Boxoft Image To PDF Demo. Purchase from www.Boxoft.com to remove the watermark

Е. Каратаев

MUMPS СУБД

Практика применения и опыт программирования

СОЛОН-ПРЕСС
2016

УДК 621.396.218
ББК 32.884.1
К 215

Каратаев Е.
MUMPS СУБД. — М.: СОЛОН-ПРЕСС, 2016. — 550 с.

Книга описывает среду исполнения, принципы организации данных в MUMPS системах и технически евопросы их применения. Описываются языковые средства, принципы обработки ошибок, алгоритмика работы с индексами, принципы внешнего взаимодействия и многое другое.

Книга может быть полезна как квалифицированным разработчикам в виде справочника, так и изучающим компьютерные технологии.

Издание интересно преподавателям и студентам профильных специальностей вузов.

УДК 621.396.218
ББК 32.884.1

Сайт журнала «Ремонт & Сервис»: www.remserv.ru
Сайт издательства «СОЛОН-ПРЕСС»: www.solon-press.ru

КНИГА — ПОЧТОЙ

Книги издательства «СОЛОН-ПРЕСС» можно заказать наложенным платежом (оплата при получении) по фиксированной цене. Заказ оформляется одним из трех способов:

1. Послать открытку или письмо по адресу: 123001, Москва, а/я 82.
2. Оформить заказ можно на сайте **www.solon-press.ru** в разделе «Книга — почтой».
3. Заказать по тел. (495) 617-39-64, (495) 617-39-65.

Каталог издательства высылается по почте бесплатно.

При оформлении заказа следует правильно и полностью указать адрес, по которому должны быть высланы книги, а также фамилию, имя и отчество получателя. Желательно указать дополнительно свой телефон и адрес электронной почты.

Через Интернет вы можете в любое время получить свежий каталог издательства «СОЛОН-ПРЕСС», считав его с адреса **www.solon-press.ru/kat.doc**.

Интернет-магазин размещен на сайте **www.solon-press.ru**.

По вопросам приобретения обращаться:
ООО «ПЛАНЕТА АЛЬЯНС»
Тел: (499) 782-38-89,
www.alians-kniga.ru

ISBN 978-5-91359-174-6

© Обложка «СОЛОН-ПРЕСС», 2016
© Каратаев Е., 2016

Оглавление

Предисловие **9**

Введение **11**

1 Среда исполнения **13**
1.1 Команды . . . 13
1.1.1 Команды присваивания . . . 18
1.1.2 Условные команды . . . 26
1.1.3 Команды передачи управления . . . 33
1.1.4 Команды ввода-вывода . . . 41
1.1.5 Служебные команды . . . 50
1.1.6 Постусловия . . . 53
1.2 Операторы . . . 55
1.3 Переменные . . . 62
1.4 Числа и строки . . . 67
1.5 Функции . . . 72
1.5.1 $DATA . . . 74
1.5.2 $GET . . . 76
1.5.3 $ORDER . . . 77
1.5.4 $NEXT . . . 79
1.5.5 $QUERY . . . 80
1.5.6 $NAME . . . 83
1.5.7 $QLENGTH . . . 85
1.5.8 $QSUBSCRIPT . . . 86
1.5.9 $ASCII . . . 88
1.5.10 $CHAR . . . 91
1.5.11 $EXTRACT . . . 93
1.5.12 $PIECE . . . 94
1.5.13 $LENGTH . . . 96
1.5.14 $REVERSE . . . 98
1.5.15 $FIND . . . 98

1.5.16 $TRANSLATE . . . 99
1.5.17 $JUSTIFY . . . 102
1.5.18 $FNUMBER . . . 104
1.5.19 $TEXT . . . 106
1.5.20 $RANDOM . . . 108
1.5.21 $VIEW . . . 109
1.5.22 $SELECT . . . 109
1.5.23 $STACK . . . 111
1.6 Списковые функции . . . 113
1.7 Битовые функции . . . 119
1.8 Модули . . . 121
1.9 Рутины . . . 125
1.10 Передача параметров . . . 131
1.11 Неопределенные значения . . . 135
1.12 Шаблоны . . . 139
1.13 Косвенность . . . 143
1.13.1 Косвенность имени . . . 146
1.13.2 Косвенность индексов . . . 147
1.13.3 Косвенность метки . . . 149
1.13.4 Косвенность аргумента . . . 151
1.13.5 Косвенность шаблона . . . 153
1.14 Интерпретатор . . . 154
1.15 Голая ссылка . . . 163
1.16 Очередность выполнения . . . 167
1.16.1 Очередность вычисления выражений . . . 168
1.16.2 Очередность вычисления имен . . . 169
1.17 Стекование $test . . . 172
1.18 Комментарий . . . 175
1.19 Стандарт и расширения . . . 177

2 Глобалы 183
2.1 B-дерево . . . 183
2.2 Кодирование индексов . . . 187
2.3 Размер блока . . . 193
2.4 Кеширование блоков . . . 198
2.5 Структуры . . . 200
2.6 Индексация . . . 207
2.7 Группировка . . . 210
2.8 Каноничность индексов . . . 220
2.9 Маппинг . . . 224

3 Индексация данных 227
3.1 Общие принципы . . . 227
3.2 Механизм поддержки индекса . . . 228
3.3 Простой индекс . . . 230
3.4 Составной индекс . . . 233
3.5 Покрывающий индекс . . . 237
3.6 Кластерный индекс . . . 241
3.7 Хеш-индекс . . . 245
3.8 Битмап индекс (bitmap) . . . 247
3.9 Битслайс индекс (bitslice) . . . 252
3.10 Нормирование значений . . . 260
3.11 Выборки по индексу . . . 262
3.12 Многоиндексная выборка (zig-zag) . . . 265
3.13 Дифференциальное индексирование . . . 269
3.14 Индексация длинных атрибутов . . . 272
3.15 Межтабличный индекс . . . 275
3.16 Индекс с условием на вставку . . . 279
3.17 Индекс на вычисляемый атрибут . . . 282
3.18 Индекс поиска по фрагменту . . . 284
3.19 Индексация для шаблона (like) . . . 287
3.20 Индексация уникального атрибута . . . 292
3.21 Массовое перестроение индексов . . . 295
3.22 Операции с древовидными индексами . . . 298
3.23 Операции с битовыми индексами . . . 303
3.24 Совмещение древовидных и битовых индексов . . . 308
3.25 Сортировка по индексу . . . 312
3.26 Статистики и кардинальность . . . 318

4 Конкурентный доступ 323
4.1 Параллельность выполнения . . . 323
4.2 Блокировки . . . 327
4.3 Транзакции . . . 333
4.4 Блокировки в транзакциях . . . 336
4.5 Функция $INCREMENT . . . 340
4.6 Функция $BIT . . . 343
4.7 Дедлоки . . . 345
4.8 TSN . . . 351

5 Обработка ошибок 353
5.1 Состояние ошибки . . . 353
5.2 ZTRAP . . . 358
5.2.1 Caché . . . 358
5.2.2 MSM . . . 364
5.2.3 DTM . . . 365
5.2.4 M3 . . . 365
5.2.5 GT.M . . . 365
5.2.6 MiniM . . . 369
5.3 ETRAP . . . 371
5.3.1 Определение . . . 371
5.3.2 $ETRAP . . . 372
5.3.3 $ECODE . . . 375
5.3.4 $ESTACK . . . 379
5.3.5 Ошибки в обработчике ошибок . . . 381
5.4 $STACK() . . . 384
5.5 Трассировка . . . 392
5.6 BREAK . . . 396
5.7 MiniM Debugger . . . 401
5.8 Caché Debugger . . . 404
5.9 Serenji Debugger . . . 406

6 Внешний мир 409
6.1 Общие принципы . . . 409
6.2 Терминальный интерфейс . . . 410
6.3 Сокеты . . . 416
6.4 WEB . . . 422
6.4.1 HTTP клиент . . . 424
6.4.2 Вебсервер на MUMPS . . . 425
6.4.3 CGI . . . 429
6.4.4 WebLink . . . 430
6.4.5 MWA . . . 433
6.4.6 CSP . . . 434
6.4.7 Проблемы HTTP . . . 437
6.4.8 Поверх HTTP . . . 441
6.5 Подключаемые DLL (SO) . . . 444
6.6 Файлы . . . 447
6.7 Внешние процессы . . . 451
6.8 Порты . . . 453

7 Практика применения 457
7.1 Терминальный режим . . . 457
7.2 Редакторы рутин . . . 459
7.3 Экспорт и импорт . . . 461
7.4 Препроцессор . . . 471
7.5 Формат $HOROLOG . . . 478
7.6 Опции устройств . . . 482
7.7 $X и $Y . . . 484
7.8 Возврат результатов . . . 486
7.8.1 Возврат по значению ($$) . . . 487
7.8.2 Возврат по ссылке . . . 488
7.8.3 Запись в предопределенную переменную . . . 489
7.8.4 Возврат значений косвенно . . . 490
7.8.5 Итеративный возврат . . . 492
7.8.6 Потоковый возврат . . . 493
7.9 %Z - рутины . . . 494
7.10 Планирование файлов . . . 496
7.11 Память и сборка мусора . . . 498
7.12 Предсказуемость . . . 504
7.13 Переносимость . . . 505

A Команды MUMPS 515

B Системные переменные MUMPS 517

C Системные функции MUMPS 519

D Стандартные коды ошибок 521

E ASCII Table 523

F Мнемоники ANSI X3.64 527

G ANSI Escape Sequences (ENG) 531

H ANSI Escape Sequences - VT100 / VT52 (ENG) 539

Предметный указатель 544

Литература 549

Издательство «СОЛОН-ПРЕСС» представляет комплект пособий по информатике и информационным технологиям

С ПОУРОЧНЫМ ПЛАНИРОВАНИЕМ

для элективного курса, профильного обучения и подготовки к ЕГЭ

КАК КУПИТЬ КНИГИ

Заказ оформляется одним из двух способов:
1) Пошлите открытку или письмо по адресу 123001, Москва, а/я 82.
2) Оформите заказ на сайте www.solon-press.ru в разделе «Книга-почтой» или «Интернет-магазин».
Бесплатно высылается каталог издательства по почте. При оформлении заказа полностью укажите адрес, фамилию, имя и отчество получателя. Желательно указать дополнительно телефон и адрес электронной почты. Каталог издательства можно получить по ссылке http://www.solon-press.ru/kat.doc
Телефон: (499) 254-44-10, (499) 252-36-96, (499) 252-25-21, e-mail: tr@coba.ru, avtor@coba.ru
Оптовые закупки у официального дистрибьютора «АЛЬЯНС-КНИГА» (www.alians-kniga.ru)

Предисловие

Во многом современный мир становится более разнообразным, чем был раньше. История знает периоды, когда становились доминирующими одни концепции систем баз данных, потом другие. И, пройдя сквозь многие годы, технология систем баз данных, заложенная в системах класса MUMPS, показала свою практичность и нацеленность на результат. Многие системы баз данных черпали из неё идеи и концепции, реализуя отдельные части её возможностей и основываясь на вполне прочном и проверенном фундаменте.

В настоящее время в практику применения входят самые разнообразные концепции систем баз данных и самые разнообразные принципы и способы их организации. И все это время, с момента создания примерно 40 лет назад, разработчики на языке MUMPS применяли развитые концепции и обеспечивали долгую жизнь своим разработкам. Видя современные, и не очень, принципы организации данных, выдаваемые за новые, многие из MUMPS программистов воспринимают их как существенно урезанные по возможностям, чем давно применяемые ими MUMPS системы.

Много лет назад, видя сколь редко выходят книги по системам баз данных MUMPS, мне хотелось, чтобы появилась такая книга, которая бы могла быть использована и как обзорная для квалифицированных разработчиков, перед которыми стоит задача выбора, и как справочная по отдельным техническим вопросам для программистов, которым необходимо разобраться в редких малоописанных темах, и как ознакомительная для изучающих компьютерные технологии в области баз данных.

Со временем пришло решение взяться за написание такой книги самостоятельно. В основу книги легли как набор статей по техническим вопросам применения MUMPS систем, так и статьи с описанием языка и среды исполнения.

Многому, отраженному в книге, я научился у тех, с кем вместе работал, и описанный в книге материал отражает опыт многих людей и решенных головоломок в самых различных проектах.

По своему образованию я не писатель, а разработчик, и, конечно, вижу мир со специфической стороны. Отдельные части книги писались намного более дотошно, чем другие, что отражает предпочтения в системе ценностей разработчика. Не являясь преподавателем, я не ставил целей получить полноценный учебник с поучающими разделами "Резюме" и с разделами контрольных вопросов. Думается, что хороший задачник с примерами еще должен быть написан.

Написание книги на языке, не являющемся языком программирования, для программистов - это большая работа, и в ней важна была моральная поддержка тех, с кем работал и с кем знаком. Как это принято при написании предисловий, выражаю благодарность всем, кто поддерживал и одобрял эту работу.

Надеюсь, что книга получилась и интересной, и полезной.

Евгений Каратаев
Москва, 2012.

Введение

Много лет назад, в середине прошлого века, когда компьютеры были большими и очень большими, методы работы с ними опирались на машинно - ориентированные способы и языки. Прикладным специалистам было необходимо такое средство, чтобы можно было описывать собственные задачи и способы обработки информации, полностью абстрагируясь от особенностей строения самих компьютеров.

Специалисты в области медицины, работавшие в Центральной больнице штата Массачусетс, коллектив разработчиков лаборатории вычислительной математики, определили основные требования к новому языку, чтобы его можно было легко применять в прикладных задачах. Так возник язык, в который были вложены множество практически востребованных технологий и методов организации вычислений. В новый язык были введены комплексно такие методы обработки информации, как работа со строками переменной длины, древовидные переменные, хранение данных в базе данных, одновременная параллельная работа заданий, встроенный механизм ввода - вывода, работа с терминальными устройствами и устройствами самых различных типов, интерпретирующая среда исполнения, возможность писать компактно и обозримо на экране размером 25*80, бестиповые данные, и многое другое.

Язык получил название MUMPS, как сокращение от названия разработки, Massachusetts General Hospital's Utility Multiprogramming System. В дальнейшем, чтобы называть язык программирования не названием определенной системы, название в стиле самого языка стали сокращать до первой буквы - M. В настоящее время названия M и MUMPS используются как синонимы для обозначения M - технологии в целом и языка в частности.

Разработка языка MUMPS была выполнена в 1967-м году. Это был период 60-х годов, годов бурного роста технологий, создания компьютеров, освоения космоса и новых видов техники, бурных событий в политике, привыкание Америки жить без расовой сегрегации, а СССР без Сталина, периода Карибского кризиса и войны во Вьетнаме, освоения

Арктики и строительства промышленных гигантов, открытий в физике элементарных частиц, феномена группы Битлз и хиппи, полетов на Луну и развитие телевидения. Умы молодых людей были наполнены идеями и верой в то, что все их задумки, выглядевшие фантастикой, могут быть выполнены, и они с открытым сердцем стремились создавать новое и совершенно шикарное по своим возможностям. Им ничто не мешало, над ними не нависал дамоклов меч маркетинга, и язык MUMPS был создан.

Разработка оказалась настолько практичной и востребованной, что была применена в самых различных медицинских учреждениях, а затем и в коммерческих, банковских. Разработка была подхвачена множеством групп разработчиков, появились самые различные реализации языка MUMPS, и молодые пытливые умы вливали в него самые разнообразные идеи. Возникла задача стандартизации языка и этим занялся комитет Mumps Development Committee, позднее был принят стандарт на язык MUMPS, выходивший со временем в более поздних редакциях.

Выход единого стандарта на язык заложил основу его распространения и применения тем, что стандарт был весьма жестким и требовательным, и соответствие реализации языка этому стандарту означало практически беспроблемный перенос программ, разработанных на MUMPS, между самыми различными компьютерами.

Со временем разработки MUMPS и M - технологии проникли и в СССР, технические и прикладные специалисты высоко оценили его возможности и различные группы разработчиков стали также применять его и в СССР. Появились также собственные реализации языка MUMPS, переводы технической документации.

В существенной степени развитие и применение M - технологий оставалось и поныне остается инициативой самих разработчиков, как системных, так и прикладных. Это живой язык, и в различных реализациях языка и сейчас производятся усовершенствования и введение различных перспективных технологий.

В этой книге освещаются вопросы самого языка, возможности среды выполнения, вопросы стандартизации, совместимости и переносимости, различные программистские трюки и головоломки. Примеры кода, примеры интерактивного исполнения команд приведены моноширинным шрифтом. Те вставки, которые содержат промпт, являются примерами интерактивной работы с MUMPS системой, остальные являются фрагментами подпрограмм.

В книге упоминаются несколько современных реализаций MUMPS систем - это Caché, GT.M, MiniM, MUMPSV1. Это названия продуктов с технической поддержкой, но книга может быть отнесена как практическое руководство также и к остальным системам.

Глава 1

Среда исполнения

1.1 Команды

По совокупности различного рода классификаций MUMPS системы относятся к процедурным, строчно - ориентированным, командно - ориентированным интерпретаторам позднего связывания.

По сути, все, что делает MUMPS система, есть исполнение заданной ей команды. В отличие от языков алгольной группы (C, Pascal), где единицей исполнения является вычисление выражения и необходимо отдельно указывать единицы исполнения не являющиеся вычислением выражения (условные операторы или операторы циклов, передачи управления), в MUMPS системе единицей исполнения всегда является выполнение команды.

В языке MUMPS различается даже очевидное для других языков действие присваивания, и требуется явно указать что именно с присваиванием необходимо сделать - присвоить только одно значение или всю переменную вместе с содержащимся в ней поддеревом.

По своей структуре команда указывает, что нужно сделать, и в аргументе указывается к чему эту команду надо применить. В отдельных случаях команды допускают применение дополнительных опций, детализирующих характер применения действия.

Команда задается ключевым словом, и это ключевое слово не является зарезервированным и может быть использовано в совершенно любом регистре. Например, несколько различных написаний одной и той же команды:

```
write
WRITE
Write
```

```
wRiTe
WRite
```

Одна из первых же неожиданностей для начинающих состоит в том, что все что есть в языке MUMPS традиционно имеет две формы написания - полную и сокращенную, или аббревиатуру-синоним. В основном, за очень редким исключением, команды сократимы до первой буквы. Например, вместо команд

```
set
merge
halt
for
write
```

можно писать то же самое в сокращенной форме:

```
s
m
h
f
w
```

Учитывая, что в давно написанных прикладных системах принято было использовать буквы верхнего регистра, и сокращение применяется также к именам функций, код на языке MUMPS для новичка может выглядеть результатом генератора случайных символов. Не нужно пугаться, уже примерно через неделю-другую после первого знакомства с MUMPS системой новички начинают писать точно также и считают MUMPS - стиль очевидным.

Другой неожиданностью для начинающих является то, что ключевые слова для команд не являются зарезервированными в языке MUMPS, и могут быть использованы в качестве имен локальных переменных. Например, такой код:

```
kill lock
```

означает удаление локальной переменной lock, а код

```
lock kill
```

означает блокирование локальной переменной kill.

Большинство команд имеет так называемую безаргументную форму. Если после имени команды не указан аргумент, то система исполняет эту команду в безаргументной форме. Безаргументная форма по своему

назначению и пониманию относится ко всему контексту исполняемого системой процесса. Например, безаргументная форма команды kill удаляет вообще все локальные переменные до которых сможет добраться.

Чтобы отличить безаргументную форму от аргументной, используется правило - если после имени команды стоит ровно один пробел и далее следует аргумент или синтаксическая конструкция подходящая по смыслу под допустимый аргумент, то это аргументная форма команды. Все остальные варианты, например отсутствие символов в строке после команды, два или более пробелов, воспринимается MUMPS системой как безаргументная форма. Например, если есть конструкция удаления локальной переменной

```
kill lock
```

то, стоит сделать не один а два пробела

```
kill  lock
```

то это уже будет две команды, одна - удаление всех локальных переменных, вторая - снятие всех блокировок.

При этом, есть команды, для которых неприменима безаргументная форма, например команды set или merge, есть команды для которых не применима аргументная форма, например команды else или halt, и есть команды которые являются различными командами в зависимости от того, указан аргумент или нет (h - команды).

H - команды это две команды, halt и hang. Первая прекращает выполнение процесса, вторая - приостанавливает выполнение на указанное число секунд. Обе имеют допустимую форму сокращения до первой буквы (h), но если аргумент указан то система считает эту команду командой hang, а если нет - то командой halt.

Все команды языка MUMPS могут быть сгруппированы во вполне традиционные для языков программирования группы команд:

1. Команды присваивания
2. Условные команды
3. Команды передачи управления
4. Команды ввода-вывода
5. Служебные команды

При этом, в силу полного отсутствия типизации переменных в языке MUMPS отсутствуют команды декларирования переменных или описания их типа.

Синтаксически языки MUMPS и ранние версии языка BASIC организованы во многом аналогично - оба основаны на командах, оба являются строчно - ориентированными, оба содержат встроенные команды ввода-вывода.

Традиционно, разработчики на языке MUMPS, работая над одним проектом, принимают набор стилистических соглашений для написания кода, или стандарты кодирования, и в различных командах могут быть приняты различные соглашения. При этом разные группы свободно понимают стили друг друга.

Автор заметил, что большинство проектов разработанных много лет назад, используют в основном соглашения о верхнем регистре команд, функций и имен переменных, а проекты разработанные недавно, уже чаще применяют нижний регистр. В языках семейства C/C++ ключевые слова являются регистрозависимыми и большинство идентификаторов по инерции используется в нижнем регистре. В языках, допускающих регистронезависимые ключевые слова и идентификаторы, чаще можно встретить смешивание регистров для большей читабельности кода. В языке MUMPS по самому его духу не принято никак препятствовать разработчикам либо навязывать какой-либо стиль, поэтому можно встретить самые различные соглашения о кодировании, при этом у разработчиков работающих в разных стилях, практически не встречается неприятия иных стилей.

При чтении кода разработчики по инерции читают сначала имя команды, затем в зависимости от того указана аргументная форма или безаргументная, читают аргумент команды, и так далее. В одной строке можно указывать подряд несколько команд, и они будут выполняться последовательно в порядке написания, слева направо. Исключение - условные команды и команды передачи управления.

Команде могут быть указаны не один аргумент, а несколько, перечисленные через запятую. В этом случае команда применяется последовательно к каждому из указанных аргументов. Например, команда присваивания

```
s a=123,b=456,c=789
```

последовательно выполняет 3 присваивания.

Вот, например, небольшая последовательность команд, имитирующая с некоторым приближением командный шелл MUMPS системы:

```
f  r !,"mumps>",l,! q:l=""  x l
```

Здесь сначала идет команда f (for), в безаргументной форме, что означает вечный цикл. Затем идет команда r (read), чтение из текущего устройства ввода-вывода с выводом сначала перевода строки (!), затем строки приглашения, затем чтения строки в локальную переменную l и затем снова вывода в текущее устройство перевода строки. Затем идет команда q (quit) с постусловием no в безаргументной форме (два пробела после постусловия), прерывающая выполнение цикла если значение переменной l получилось равным пустой строке, и далее идет команда x (xecute), выполняющая строку l как последовательность команд.

С одной стороны, пример для новичка совершенно нечитабелен и может выглядеть шпионской шифровкой, но для человека знакомого с языком MUMPS эта короткая строка читается именно как большой абзац с более чем осмысленным содержанием.

Та же самая строка, но в развернутом и приукрашенном виде может выглядеть, например, так:

```
For  Do  Quit:LineToExec=""
. Write !,"MUMPS>"
. Read LineToExec
. Write !
. If LineToExec="" Quit
. Xecute LineToExec
```

При написании кода в более традиционном, развернутом стиле, новый вариант увеличивается в объеме в разы и более-менее осмысленный код по отправке, например, сообщения по SMTP, занимающий пару экранов, может превратиться в необъятного размера монстра.

Автор считает, что есть повод задуматься о компактности и обозримости кода, если необходимо работать с действительно сложными и ответственными системами. Например, с такими как медицинская система WorldVista. Ее исходный текст на языке MUMPS занимает примерно 94 мегабайт. Думается, если бы ее писали на языке, требующем большей "разговорчивости", то она не могла бы быть написана до сих пор, в то время как применение MUMPS позволило ей быть написанной, отлаженной и работающей не одно десятилетие.

С другой стороны, свобода выбора, предоставляемая языком MUMPS, ничуть не препятствует также и написанию практически нечитабельного кода. Например, код написанный в стиле минимизации команд и демонстрирующей на экране падающие зеленые буквы в стиле заставки фильма "Матрица":

```
matrix
matrix ; k  d ^matrix w
 w *27,"[32;40m",*27,"[?25l",#
 n ch,i f  r *ch:0.15  q:ch'=-1  d
 . w *27,"[H",*27,"[1L"
 . f i=1:1:80 w $s(i#4:" ",$r(8):$c($a("A")+$r(26)),1:" ")
 w *27,"[0m",#,*27,"[?25h" q
```

для непосвященного может выглядеть также таинственно как и сам фильм.

Иногда можно встретить шутку, что текст правильной (или вернее сказать идеальной) программы на MUMPS должен занимать ровно 80x25 символов.

1.1.1 Команды присваивания

Команды присваивания изменяют значение переменной. В качестве нового значения могут быть использованы

1. Результат вычисления выражения
2. Копия другой переменной с ее поддеревом
3. Присваивание неопределенного значения

Нужно отметить, что, в отличие от систем баз данных, различающих операции insert и update в зависимости от того, существуют изменяемые данные или нет, в языке MUMPS операции манипулирования данными определены не на начало операции, а на окончание операции. Например, для присваивания значения не требуется существование или несуществование переменной, она будет или перезаписана или создана, но в любом случае будет иметь указанное значение по окончании операции. Точно также для удаления переменной не требуется ее существование или несуществование, по окончании операции удаления переменной не будет существовать.

Для присваивания значения выражения используется команда set

```
set varname=expr
```

Для присваивания команда set сначала вычисляет все значения индексов указанные для переменной varname, затем вычисляет значение expr, затем присваивает.

Одной из необычных особенностей языка MUMPS является то, что для команды set могут быть использованы левосторонние функции. Обычная функция используется для вычисления выражения по некоторому правилу и для возврата результата. В левосторонних функциях указывается что и как сделать с какой-либо частью значения переменной. Например, функция $extract возвращает подстроку значения:

```
USER>s str="abcdefgh"

USER>w $e(str,2,4)
bcd
```

При этом функция описывает позиции, с которыми проиводится операция. Точно также функция $extract может быть использована для указания позиций для присваивания, и при этом команда set присвоит не все значение целиком, а перезапишет лишь указанные позиции в строке:

```
USER>s str="abcdefgh"

USER>w $e(str,2,4)
bcd
USER>s $e(str,2,4)="FFFFFF"

USER>w str
aFFFFFFefgh
```

К таким левосторонним функциям в языке MUMPS относятся функции $extract, $piece, и в зависимости от реализации, $list, $bit и $qs.

Левосторонние функции могут быть применены лишь к локальным или глобальным переменным. Применить их к системным, даже если системная переменная допускает присваивание, нельзя.

Для левосторонних функций, выполняющих замещение в определенном формате, вводится в каждом случае специальное соглашение о дополнении строки, если значение содержит недостаточное число байт. В частности, функция $extract дополняет пробелами, функция $piece дополняет фрагментами из пустых строк, функция $bit дополняет нулевыми битами, функция $list дополняет неопределенными элементами списка, а функция $qs дополняет индексами из пустых строк.

```
USER>s str="12"

USER>s $e(str,7)=777

USER>w
```

```
str="12    777"

USER>s str="1,2,3"

USER>s $p(str,",",7)=777

USER>w
str="1,2,3,,,,777"

USER>s str=$na(a(1,2,3))

USER>s $qs(str,7)=777

USER>w
str="a(1,2,3,"","","",777)"
```

Поведение дополнений максимально приближено к трактовке недостающих элементов как значений по умолчанию с точки зрения каждой из этих функций.

Команда set имеет дополнительный синтаксис применения действия присваивания к указанному списку переменных. В этом случае значение выражения вычисляется однократно, но присваивается каждой из перечисленных в списке переменной или левосторонней функции. Общий синтаксис:

```
set (assign1,assign2,...)=expr
```

Например, присваивание пустых строк сразу трем переменным:

```
set (a,b,c)=""
```

Использование левосторонних функций в зависимости от задачи может как существенно снизить читабельность кода, так и наоборот, существенно увеличить его. Сравните, например, просто присваивание:

```
s $p(rec,4,"~")=$h
w $p(rec,4,"~")
```

и простановку значения текущей даты в поле записи:

```
s nDATE=4
s DELIM="~"
...
s $p(rec,nDATE,DELIM)=$H
w $p(rec,nDATE,DELIM)
```

где используются уже более осмысленные и значимые идентификаторы.

При использовании макросов препроцессора (в зависимости от того, поддерживается ли препроцессор в используемой MUMPS системе или нет) код может быть приведен к намного более легко воспринимаемому:

```
#define DATE(%v) $p(%v,4,"~")
...
s $$$DATE(rec)=$h
w $$$DATE(rec)
```

Что интересно, существуют реализации MUMPS, в которых трактовка левосторонних функций была расширена и в качестве аргумента им разрешено принимать не только локальную или глобальную переменную, но и указывать другую левостороннюю функцию. В частности, такая возможность была реализована в системе StarMUMPS. В ней разрешено рекурсивное левостороннее присваивание, например, так:

```
s $p($p(var,delim1,pos1),delim2,pos2)=expr
```

В этом случае система оперирует подстрокой, выделенной разделителем delim1 из подстроки переменной var, выделенной разделителем delim2, и, при необходимости, выполняет дополнение в обеих подстроках. Но такой функционал не входит в стандарт и не поддерживается современными распространенными реализациями MUMPS.

Для копирования переменной с поддеревом нужно указать не значение, а переменную, из которой необходимо скопировать. Копирование выполняет команда merge.

```
merge var1=var2
```

В правой и левой части можно указать переменные с индексами, в этом случае будет копироваться соответственно в поддерево и из поддерева.

У команды merge есть две особенности, важные для систем баз данных. Первая состоит в том, что если в переменной указанной в левой части уже было поддерево, то оно не будет предварительно удалено. Например, если есть переменная

```
USER>s a(1,1)="a11",a(1,2)="a12"

USER>s b(3,1)="b31"
```

то при копировании поддерева

```
USER>m a(1)=b(3)
```

часть поддерева a, не существующая в поддереве b, будет сохранена, та часть которая есть в обоих, будет перезаписана, а та часть которая отсутствует в a но присутствует в b, будет добавлена к a:

```
USER>m a(1)=b(3)

USER>w
a(1,1)="b31"
a(1,2)="a12"
b(3,1)="b31"
```

Поэтому, при использовании команды merge и при необходимости иметь строгую копию, разработчики на MUMPS должны убедиться, что ход выполнения программы не создает в пременной, указанной слева, посторонних записей.

Второй особенностью команды merge является отношение к неопределенным значениям. Если в поддереве источнике копирования нет записей, то они не будут копироваться. И не будут копироваться никакие записи, если такой переменной вообще не существует. В этом случае команда merge вообще ничего не делает. И, если переменной указанной слева, не существовало, то она и не будет создаваться.

```
USER>k

USER>m a(1)=b(2)

USER>w

USER>
```

Здесь предварительно были удалены все локальные переменные, затем присваивается поддерево b(2) в поддерево a(1), но поскольку поддерева b(2) не существовало, то поддерево для a(1) не создается и переменной a по-прежнему не существует. Команда merge, таким образом, при обращении к неопределенным переменным не генерирует ошибку.

Что интересно, команда merge по стандарту языка не допускает копирование поддерева в то же самое поддерево. Если переменные a и b являются по отношению к друг другу поддеревьями, то команда merge генерирует ошибку о невозможности выполнить такую операцию:

```
USER>m a(1)=a(2)

USER>m a(1)=a(1,2)

<COMMAND>
```

Здесь в первом случае переменные a(1) и a(2) не являются друг по отношению к другу поддеревьями и команда merge выполняется без ошибки, а во втором случае переменная a(1,2) является поддеревом для переменной a(1) и команда merge генерирует ошибку.

Для присваивания неопределенного значения в языке предназначена команда kill. Команда удаляет указанную переменную, в результате она становится переменной с неопределенным значением. В языке MUMPS нет отдельного понятия, что переменная существует но не имеет значения, если у переменной нет значения то она не хранится и не существует.

Поведение команды kill определено на момент ее окончания - если переменная существовала, то будет удалена, если не существовала, то команда ничего не делает. Команда может быть применена к локальным и глобальнм переменным. В зависимости от реализации MUMPS системы может дополнительно поддерживаться применение команды к структурной системной переменной, например в реализации MiniM команда

```
kill ^$JOB(jobnumber)
```

принудительно завершает процесс с номером jobnumber.

Применение команды kill к чему-либо кроме локальной или глобальной переменной, вообще говоря, не входит в стандарт, и к таким возможностям нужно относиться внимательно. В зависимости от ситуации такой случай может означать ошибку, в частности, если при использовании косвенной формы команды в прикладной системе предполагается, что команда должна генерировать ошибку если аргумент не является ни локальной ни глобальной переменной.

У команды kill поддерживается безаргументная форма - в этом случае команда применяется ко всем локальным переменным, которые существуют.

Кроме того, у команды kill поддерживается исключающая форма: если в качестве аргумента указаны имена локальных переменных перечисленные в круглых скобках, то команда применяется ко всем локальным переменным кроме указанных. При этом указанные в скобках переменные не затрагиваются и они не могут быть указаны с индексами. Например

```
kill (a,b,c)
```

удаляет все локальные переменные кроме a, b и c.

Если команде kill указывается переменная с индексами, то команда удаляет только указанное поддерево. Если указано имя без индексов, то удаляется дерево переменной целиком.

В отношении этого совмещенного поведения команды kill стандартом было предусмотрено предложение реализовать дополнительно еще две команды, kvalue (kv) и ksubscripts (ks), соответственно для удаления только указанного имени не затрагивая поддерева и для удаления только поддерева не затрагивая указанного имени. В реализации MiniM эти команды поддерживаются, но другими современными реализациями MUMPS пока еще не поддерживаются. Поэтому при их использовании нужно убедиться, будут ли команды работать на выбранной системе, либо понадобится ли их заменить на комбинацию других команд.

Другая команда, не имеющая прямой задачи сделать значение переменной неопределенным, но делающая это побочно, это команда new.

Команда new объявляет, что локальная переменная в случае, если впоследствии будет создаваться, то ее область видимости должна начинаться от текущего уровня стека. Если переменная с таким же именем уже существовала на другом уровне стека, то она перестает быть видимой и все дальнейшие операции выоплняются с новым положением локальной переменной. В случае, если переменная с этим именем уже была объявлена на этом же уровне стека, то она удаляется. В любом случае, непосредственно после команды new, эта переменная имеет неопределенное значение.

Исключением является расширенная инициализирующая форма команды new, реализованная в системе MiniM - для переменной можно указать значение, которое она должна получить:

```
new var=expr
```

при этом команда эквивалентна двум командам:

```
new var set var=expr
```

Это расширение, хотя и является практичным, не входит в стандарт языка MUMPS и в настоящее время не поддерживается другими MUMPS системами.

У команды new стандартно поддерживается безаргументная форма. В случае, если команде не указано ни одно имя, это означает что вообще все локальные переменные далее имеют область видимости начиная с текущего уровня стека.

Кроме того, у команды new стандартно поддерживается исключающая форма:

```
new (var1,var2,...)
```

В этом случае команда применяется не к перечисленным именам переменных, а ко всем, кроме перечисленных, в том числе еще не созданным. Для того, чтобы применить команду new только к определенным именам, их надо просто перечислить через запятую без взятия в скобки:

```
new var1,var2,var3,...
```

Команда new не удаляет данные локальных переменных, существовавших ранее, на другом уровне стека, и при возврате управления на предыдущий уровень стека действие команды отменяется, и видимость локальных переменных полностью восстанавливается.

Что интересно, в языке MUMPS, если для локальной переменной не объявлялось начало области видимости командой new, то видимость этой переменной всеобщая, и ее можно изменять и читать на любом уровне стека.

Команда new, что может показаться современным разработчикам на MUMPS странным, появилась в стандарте языка MUMPS далеко не сразу, а только в 1990-м году. Вполне возможно, что до сих пор находятся в эксплуатации модули, при разработке которых предполагалась общая область видимости имен локальных переменных. Вполне возможно, что при модернизации таких программ с использованием современной команды new разработчик может внести ошибку, поскольку отдельным фрагментам может понадобиться именно общая область видимости. Хотя при современных разработках команда new используется наоборот, как практически стандартное соглашение об изолировании переменных, изменяемых подпрограммой.

Кроме стандартной команды new многие современные реализации MUMPS поддерживают дополнительную расширенную команду znew. Ее отличие от стандартной команды new в том, что, создав для локальной переменной область видимости от текущего уровня стека, команда создает точную копию оригинальной переменной. Дальнейшие операции с этой переменной изменяют лишь ее, и при возврате на предыдущий уровень стека все выполненные изменения, естественно, пропадают. Команда относится к нестандартной, но многие реализации ее поддерживают. В случае ее использования в разработке, конечно, нужно проверить поддерживается ли она на целевой MUMPS системе.

Формально говоря, команда чтения из текущего устройства (read) также выполняет присваивание переменной в качестве побоного эффекта, но отнесена к командам ввода-вывода. Для нее основным действием является ввод, а присваивание - это лишь побочный, но необходимый результат.

1.1.2 Условные команды

Исполнение команд в программе на MUMPS выполняется построчно, и внутри строки слева направо. Для управления выполнением программы в зависимости от некоторых условий в языке MUMPS определены условные команды.

К условным командам относятся команды if, else и for.

Команды if в безаргументной форме и else (имеет только безаргументную форму) проверяют значение системной переменной $test. По определению операций, эта переменная может быть установлена множеством способов. Например, при неудаче получения блокировки в течении определенного таймаутом времени. Поэтому в программах MUMPS часто можно встретить команды if или else как-бы не относящиеся ни к какому явно вычисляемому условию.

Аргументная команда if вычисляет значение аргумента как число и если результат 0, то взводит системную переменную $test в значение 0, иначе взводит $test в значение 1. И выполнение команд продолжается в зависимости от результата.

У команды else есть особенность - для нее синтаксически недопустимо указание аргумента, поэтому после нее должно быть обязательно 2 пробельных символа.

Для строчно-ориентированных интерпретаторов условные команды распространяются либо на всю последующую после команды строку, либо на набор команд ограниченных операторной скобкой then - else - end. В языке MUMPS используется первый вариант и отсутствует синтаксическая конструкция then. Это весьма важная для понимания работы системы особенность. Если условный оператор выполнился, то команды расположенные далее в строке, также выполняются, а если не выполнился, то интерпретатор не выполняет дальнейшие команды, а переходит к следующей строке либо продолжает выполнять цикл, если условная команда находится в цикле.

В частности, в языке MUMPS нет смысла располагать команду else после команды if на той же строке, например так:

```
if a write 1 else  write 2
```

Если значение переменной a в этом примере не равно 0, то выполняется команда write 1, после чего команда else проверяет значение $test взведенное командой if и не выполняет команду write 2. Это как-бы ожидаемое поведение. Но если значение переменной a равно 0, то команда if не передает управление на последующие команды, расположенные в той

же строке, и управление в этом случае не будет передано на команду else.

Правильным вариантом вышеприведенного примера будет такой:

```
if a write 1
else  write 2
```

Формально говоря, в предложения комитета по стандартизации MDC входит предложение реализовать команду then как явно стекующую системную переменную $test, но в настоящее время ни одна MUMPS система такую команду не поддерживает, и программы на MUMPS её не используют.

Системная переменная $test сохраняет свое значение до следующего изменения, и её, конечно, можно использовать для передачи информации, но так обычно не поступают. Традиционно, если в программах требуется проверка переменной $test, то она взводится непосредственно перед проверкой. Поэтому операции, косвенно взводящие значение $test, но игнорирующие полученное значение, не влияют на ход выполнения программ.

Это используется многими разработчиками для вызова системных или пользовательских функций с игнорированием возврата. Вызов функции в этом случае выполняется командой if, но результат сравнения игнорируется, и далее в строке не ставятся никакие команды. В языке MUMPS, если функция возвращает значение, то оно должно быть использовано как результат выражения, и если не используется, то MUMPS система генерирует либо синтаксическую ошибку для системных функций, либо ошибку времени исполнения при выполнении команды возврата значения.

Нужно отметить, что аргументная форма команды if взводит значение $test, безаргументная форма команды if не взводит, но проверяет, и команда else не взводит, но проверяет значение $test. Если необходимо взвести значение переменной $test, то можно написать просто

```
...
if 1
...
; $test=1
...
```

или

```
...
if 0
```

```
...
; $test=0
...
```

Команда for устроена более сложно. Также как и команды if и else она распространяет свое действие на команды, размещенные после нее в этой же строке, но, в отличие от них, использует собственные правила итерации в зависимости от формы команды. В одной строке могут быть несколько команд for, в этом случае расположенные левее будут содержать в своем цикле расположенные правее.

Важно отметить, что если команды if и else управляются системной переменной $test, то команда for не реагирует на значение $test и управляется собственными правилами итерации и окончания цикла.

Команда for продолжает выполнение последующих за собой команд. Если команды расположенные на строке выполнились, то управление передается не на следующую строку, а последней выполнявшейся команде for и команда проверяет собственное условие завершения, и при необходимости, итерации. При переходе на следующую строку контекст цикла отменяется, независимо от того, произошло ли прерывание цикла или переход выполнен по команде goto.

Например, при выполнении кода

```
write "start",!
for i=1:1:5 write i," : ",i*i,!
write "end",!
```

после вывода квадрата числа не будет производиться переход на следующую строку. Вместо этого управление передается на команду for, и если ее итерации закончены, то только тогда управление перейдет на следующую строку.

У команды for в языке MUMPS есть несколько форм - безаргументная, одноаргументная, двухаргументная и трехаргументная. При этом, если есть аргументы, то должна быть указана переменная цикла и аргументы указываются через запятую не для команды, а для переменной цикла.

Безаргументная форма команды for не имеет ни условия окончания цикла, ни итерации переменной цикла. Выполнение вечного цикла может быть прервано лишь командой quit, командой goto, либо генерацией ошибки. При этом команда for одновременно налагает требование на команду quit, если она присутствует в теле цикла: допускается только безаргументная форма quit и она трактуется как прерывание текущего исполняемого контекста цикла, а не как возврат из подпрограммы.

Если команда quit выполнилась в цикле, который вложен в другой, то прерывается только этот внутренний, не затрагивая выполнения внешнего:

```
USER>f i=1:1:2 f j=3:1:5 q:j=4  w i,":",j,!
1:3
2:3
```

Традиционным идиоматическим оборотом для безаргументной формы for является итерация по перечислению, например, значений индексов. Вариант для общего примера:

```
USER>s a(123)="",a(456)="",a(789)=""

USER>s n="" f  s n=$o(a(n)) q:n=""  w n,!
123
456
789
```

Здесь используется не переменная итерации цикла, а просто обычная локальная переменная, которая принимает значения индексов. Вечный цикл прекращается командой quit если следующий индекс пустая строка. Для понимания примера важно, что команда for использована в безаргументной форме из-за того, что мы самостоятельно проверяем условие окончания и команда quit использована в безаргументной форме, поскольку из цикла не допускается возврат значения. Безаргументные формы обеих команд обозначены двойными пробелами.

Одноаргументная форма for указывается одним значением для переменной цикла:

```
for var=expr
```

При этом команда for выполнит итерацию однократно, предварительно присвоив переменной var значение вычисленного выражения expr. Например:

```
USER>for i="a" w i
a
USER>
```

У одноаргументной формы for нет итераций и условия окончания, она выполняется строго однократно. В качестве значения итерации может быть указано любое выражение, никаких сравнений или арифметических действий с ним в одноаргументной форме команда for не будет выполнять.

Двухаргументная форма команды for задается начальным значением переменной цикла и приращением. При каждой итерации переменная цикла будет получать указанное приращение. Для двухаргументной формы нет условия окончания итераций и цикл будет выполняться пока не будет прерван командой quit, командой goto или генерацией ошибки. Пример:

```
USER>for i=1:2 w i q:i>7
13579
USER>
```

Для двухаргументной формы переменная итерации уже всегда используется как число, даже если начальное значение было задано как строка:

```
USER>for i="a":2 w i q:i>7
02468
USER>
```

Переменная цикла может быть изменена на любой итерации, и команда for всегда использует строго текущее ее значение, прибавляя значение итерации именно к тому значению, которое досталось команде после очередной итерации:

```
USER>for i=1:2 w i,! s i=i+0.5 q:i>7
1
3.5
6
8.5

USER>
```

Трехаргументная форма команды for имеет дополнительно третье значение, условие окончания цикла, задаваемое после значения итерации через двоеточие:

```
USER>for i=1:1:5 w i,!
1
2
3
4
5

USER>
```

Нужно отметить, что при выполнении тела цикла сама переменная цикла может принять и неопределенное значение при выполнении команд kill или new. В этом случае при очередной итерации цикла производится ее чтение и генерируется не ошибка доступа к неопределенной переменной (код M6), а специально зарезервированная ошибка доступа к неопределенной переменной цикла (код M15).

Для трехаргументной формы for условие окончания итерации проверяется каждый раз перед выполнением команд цикла. При этом команда поступает в зависимости от знака приращения итерации - если приращение положительное, то цикл прекращается если значение переменной цикла стало больше конца цикла и если приращение отрицательное, то цикл прекращается если значение переменной цикла стало меньше конца цикла. В целом, синтаксис команды for в трехаргументной форме определен наиболее ожидаемым для разработчиков образом.

Интересно, что для двухаргументной и трехаргументной формы параметры цикла (приращение и окончание) командой for вычисляются строго однократно, перед первым выполнением цикла и далее изменение значений входящих в эти выражения, не влияют на выполнение for - команда будет строго пользоваться вычисленными первоначально значениями.

Одной из наиболее интересных особенностей команды for в языке MUMPS является возможность указать несколько аргументов, как было упомянуто ранее. Аргументы применяются последовательно к переменной цикла, один за другим. При этом разные аргументы могут иметь разную форму - одноаргументную, двухаргументную и трехаргументную. Пример трех аргументов для команды for, и все три в одноаргументной форме:

```
USER>for i="abc","def","qwe" w i,!
abc
def
qwe

USER>
```

Вариант комбинирования различных форм:

```
USER>for i="abc","def",1:1:5,"qwe" w i,!
abc
def
1
2
3
```

```
4
5
qwe

USER>
```

Возможность указать подряд несколько одноаргументных форм через запятую иногда приводит к тому, что разработчики считают это отдельной формой команды for, выполняющей итерации по перечисленному списку:

```
USER>for i="abc","def","qwe" w i,!
abc
def
qwe

USER>
```

На самом деле это не так, это несколько одноаргументных форм, и вместе с ними возможно смешивание любых других форм команды for кроме, конечно, вечного цикла.

Что еще интересно, все три условные команды языка MUMPS, if, else и for, являются строго стандартными, различные реализации языка не отклоняются от стандартного их определения и к ним не были добавлены другие условные команды.

Кроме того, что команда for может быть прервана безаргументной командой quit или генерацией ошибки, нормальный ход выполнения цикла, естественно, может быть прерван безусловным переходом к явно указанной строке. В этом случае исполняющая система MUMPS отменяет текущий контекст выполняемых циклов и выполнение команд продолжается вне контекста цикла. Разумеется, после безусловного перехода по goto никакие итерации в текущем контексте более не имеют смысла, но сохраняются изменения локальных переменных, использованных в качестве переменных цикла.

Это общее правило для любых языков програмирования, допускающих безусловный переход в контексте цикла, за исключением перехода в тот же контекст. Но для строчно-ориентированных интерпретаторов понятие перехода в тот же контекст цикла не поддерживается.

В отношении условных команд (if, else, for) действует специальное синтаксическое правило языка - для них не допускается использование постусловий. Кроме того, для команды for синтаксически не допускается косвенное задание аргумента. Косвенность допускается лишь для вычисления имени переменной цикла, начального значения, приращения

и условия окончания итерации. Задание же структуры цикла косвенно не поддерживается.

Стандарт на язык в отношении команд цикла for предусматривает, что в качестве переменных цикла могут быть использованы исключительно локальные переменные. Автор пока не встретил веских оснований, почему бы в качестве переменной цикла не использовать глобальную переменную. Формально, для виртуальной машины при исполнении команды for нет существенных отличий, с какой именно переменной надо выполнить действия, с локальной или глобальной. Но в настоящее время язык определен именно таким образом, и все MUMPS системы запрещают использовать в качестве переменной цикла что-либо иное, кроме локальной переменной.

Этот запрет поддерживается даже если переменная цикла задана косвенно. Такой код

```
USER>f ^i=1:1:2
```

приведет к синтаксической ошибке времени трансляции, а такой

```
USER>s n="^i"

USER>f @n=1:1:2
```

к синтаксической ошибке времени выполнения.

Но язык не ограничивает множество имен для переменной цикла только неиндексированными локальными переменными, и разработчики могут свободно использовать также индексированные локальные переменные. При этом все MUMPS системы вычисляют имя переменной цикла (даже если в качестве индексов указаны волатильные выражения) однократно, и далее используют только его.

То же самое правило об однократном вычислении (при начале выполнения цикла) распространяется на начальное значение, приращение и значение окончания цикла. Даже если они заданы волатильными выражениями, команда for будет использовать только те значения, которые были получены при их первом вычислении.

1.1.3 Команды передачи управления

Для изменения последовательного хода программы используются команды передачи управления. К таким относятся команды, действие которых приводит к смене текущей строки исполнения без возврата или с возвратом управления на следующую команду.

В целом, все, что исполняет MUMPS система, задается либо совокупностью имеющихся рутин, либо интерактивно введенной строкой, либо косвенным исполнением. Каждая из рутин представляет собой текст, рассматриваемый как последовательность строк. В принципе, для строчно - ориентированной машины исполнения не имеет значения, на какую из других имеющихся строк (в этой же рутине или в другой) передать управление.

Для того чтобы можно было указать строку рутины, есть в любом случае наиболее простой способ - это указать ее номер, например рутина ^ABC, строка 14.

Но такой способ неудобен в случае если рутины модифицирутся и номера строк могут измениться со временем. Для того чтобы указать определенную строку, ей приписывается так называемая метка. В MUMPS для описания метки не требуется никаких деклараций или отдельного синтаксиса - просто если строка рутины начинается не с пробельного символа, то первое слово строки и есть метка.

При исполнении строки с меткой сама метка не содержит никаких указаний исполняющей системе MUMPS за исключением того, что именует строку, в которой она находится.

Для того чтобы передать управление на другую строку, нужно указать рутину (можно не указывать если это текущая рутина), метку, либо смещение относительно начала рутины, либо метку плюс смещение относительно метки.

Механизм использования меток в MUMPS используется и для организации безусловного перехода и для перехода на подпрограмму с возвратом из нее. Одна и та же метка может быть использована в обоих случаях. Отношение к метке и ее значение определяется в отличие от других языков не тем, как она описана, а тем, как она вызвана. При трансляции рутины транслятор еще не знает, как будет использована строка с меткой, поэтому готов выполнить вызов любой строки, обеими вариантами.

Собственно, самих подпрограмм в понимании других языков программирования, как самостоятельной цельной единицы трансляции, в MUMPS не существует. Будут ли строки рутины работать как просто последовательный набор строк или как подпрограмма, определяет разработчик.

Автору доводилось работать с прикладными системами, написанными на MUMPS, часть рутин которых организована в виде логически связанного списка. Каждая из рутин (кроме последней) в последней своей строке содержит команду безусловного перехода на первую строку следующей рутины. В итоге весьма большое количество кода выполняется

последовательно. При разработке реально больших систем, где даже с применением столь компактного языка, как MUMPS, объем кода может оказаться очень большим, и такие возможности языка были использованы весьма грамотно. В любом случае, язык MUMPS не будет навязывать какой-либо стиль и лишать каких-либо возможностей.

Если после имени метки разработчик ставит в круглых скобках перечень аргументов через запятую, то такая метка становится меткой с параметрами и ее надо вызывать строго как подпрограмму, с передачей параметров. При вызове исполняющая система MUMPS сопоставляет переданные фактически параметры объявленным формальным параметрам таким образом, что следующие после метки команды (неважно в этой же строке или в следующей) уже исполняются в контексте, в котором эти формальные параметры являются локальными переменными.

Если метка была объявлена с параметрами, то на нее нельзя перейти безусловным переходом, поскольку исполняющая машина (интерпретатор) MUMPS не знает как и что сопоставить этим параметрам как локальным переменным нового контекста, при безусловном переходе не создается новый контекст или фрейм на стеке.

К командам, передающим управление, в языке MUMPS относятся:

1. HALT - команда останавливает процесс, после выполнения этой команды процесс завершает работу и далее никакие команды не выполняются.

2. DO - управление передается либо на указанную метку как на подпрограмму, либо на последующий блок строк.

3. QUIT - управление возвращается из подпрограммы либо прерывается выполнение цикла.

4. XECUTE - команда выполняет свой аргумент как последовательность строк.

5. GOTO - управление передается безусловным переходом на указанную строку.

Команда HALT (H) останавливает процесс полностью, все локальные переменные процесса пропадают, и память занятая процессом освобождается. Все устройства ввода-вывода, которые были открыты закрываются, и все блокировки, установленные процессом, снимаются. Если команда выполнилась, то дальнейшие команды в этой или последующей строке не выполняются.

Команда DO (D) имеет две формы - можно указать метку в рутине (с возможным смещением) для вызова в качестве подпрограммы либо безаргументная форма, при которой последующий блок строк рассматривается как подпрограмма. Например

```
do label^rtn
```

передает управление на рутину rtn начиная с метки label, и в новом контексте создается стековый фрейм. Если в вызванном коде будет выполнена команда quit, то эта команда вернет управление на команду, следующую после команды do, обратившейся к метке. В этом случае вызов соответствует вызову подпрограммы или процедуры.

Чтобы передать параметры подпрограмме, нужно чтобы вызываемая метка принимала параметры и чтобы вызывающий код передал их. Параметры в обоих местах указываются в круглых скобках. Если есть подпрограмма

```
label(param)
 write "param = ",param,!
 quit
```

то ее можно вызвать с передачей параметров:

```
do label^rtn(789)
```

В вызываемом коде команды следующей в строке после метки уже работают в контексте нового стекового фрейма и параметр param уже является локальной переменной с областью видимости от текущего уровня стека и далее.

Если подпрограмма была вызвана командой DO, то от подпрограммы не ожидается возврат значения и команда QUIT должна быть использована в безаргументной форме, иначе при выполнении кода система генерирует ошибку. Если требуется вызов подпрограммы с возвращаемым значением, то для возврата значения должна быть использована аргументная форма команды quit и метка должна быть вызвана как пользовательская функция, с указанием символов $$ перед именем метки. В этом случае метку можно использовать как часть вычисляемого выражения произвольной сложности. Например, функция вычисляющая куб числа:

```
CUBE(value)
 quit value*value*value
```

должна быть вызвана с указанием вызова пользовательской функции:

```
set result=123+$$CUBE^rtn(789)+456
```

Если подпрограмма вызывалась как пользовательская функция с ожиданием возврата, то управление из нее должно возвращаться командой quit с аргументом. Значение аргумента вычисляется как выражение и возвращается в место вызова. Если такая функция прерывается и управление возвращается безаргументной формой команды quit, то система генерирует ошибку.

В других языках есть возможность указать неименованные части функции как блок операций, исполняемых цельно, например в языке C++ все что ограничено фигурными скобками исполняется как единый последовательный контекст:

```
funcname( int param)
{
  // code block 0
  ...
  if(param == 2)
  {
    // code block 1
    ...
    for(int i = 1; i < 4; i++)
    {
      // code block 2
      ...
    }
  }
  else
  {
    // code block 3
    ...
  }
  return;
}
```

В языке MUMPS, как в строчно - ориентированном, нет возможности указать границы блока при трансляции текста рутины. В других аналогичных строчно - ориентированных языках могут быть использованы, например, синтаксические конструкции NEXT, являющиеся не столько командами, сколько операторными скобками ограничения блока команд или операторов.

В языке MUMPS, как строчно-ориентированном, вместо этого применяется специальный маркер в начале строки, символ точка. Если строка начинается с точки, это означает что строки с равным числом точек в начале, и идущие последовательно, образуют один уровень вложения

блока строк. Если нужно организовать блок в блоке, то ставится две точки, и так далее. При использовании безаргументной формы команды DO интерпретатор переходит в режим перехода от строки к строке с учетом числа точек в начале строки. Если начали выполнять от уровня с числом точек N, то все последующие строки идущие подряд с числом точек N + 1 считаются принадлежащими блоку. Как только интерпретатор встречает строку у которой число точек меньше или равно N это означает что выполняемый блок строк закончен. Если при этом интерпретатор встречает строки с числом точек больше N + 1, то они просто пропускаются. Например

```
write "begin",!
if a=1 do
. write "in block",!
. s a=0
write "end",!
```

Здесь две строки

```
. write "in block",!
. s a=0
```

образуют неименованный блок строк, выполняемый безаргументной формой команды do как подпрограмма. При переходе от строки

```
. s a=0
```

к следующей интерпретатор обнаруживает, что число точек стало меньше чем у выполняемого блока и выполняет неявную команду quit в безаргументной форме, и управление возвращается к команде, следущей за командой do. В строке после команды do в данном случае ничего не указано, и интерпретатор переходит к следующей строке. Но в ней число точек больше чем число точек текущего контекста (0), поэтому обе строки с точками просто пропускаются и интерпретатор переходит непосредственно к выполнению команды после строк с точками

```
write "end",!
```

После команды do в этом примере также могут стоять команды. Нужно лишь учесть, что для вызова блока строк как подпрограммы нужна именно безаргументаня форма команды DO, поэтому необходимо два пробела:

```
write "begin",!
if a=1 do  write "after do",!
. write "in block",!
. s a=0
write "end",!
```

Нужно отметить, что в системе Cache старших версий поддерживается расширенный синтаксис, который включает как свободное форматирование пробелов по контексту, допускающем их, так и применение фигурных скобок для ограничения области действия синтаксических конструкций. В частности, безаргументная команда DO и некоторые другие в этих условиях могут быть применены не к последовательности строк с точками, а к поледовательности команд, ограниченной фигурными скобками.

Команда GOTO (G) выполняет просто переход. Исполняющая система MUMPS начинает выполнять первую команду находящуюся на указанной строке. Строка может иметь метку, но не может иметь у метки параметры. Если при выполнении команды GOTO действовал контекст цикла, то он отменяется. Контекст стека полностью сохраняется. Вызываемый код может вызвать команду quit, которая вернет управление.

В отношении команды перехода GOTO язык MUMPS не имеет ограничений как в других языках. Например, в языке C или Pascal переход возможен только в пределах той же самой функции, в языке Assembler переход возможен на метку только в пределах команд ассемблерного модуля, но не в другой ассемблерный модуль, ассемблерные вставки в коде C / C++ могут как допускать, так и не допускать переход ассемблерной командой в контекст другой функции в зависимости от транслятора. В случае с MUMPS свобода перехода полная, включая возможность вычислять смещение как выражение и применять косвенные формы фрагментов метки.

Для перехода действует только одно единственное ограничение - смещение относительно метки, если оно указано, не должно быть отрицательным числом, таковы требования стандарта языка MUMPS.

Примеры команд GOTO:

```
 ; Routine R01
func(param)
 s value=123
 i +param goto param+param^R02
continue
 w value
 ...
```

```
  ; Routine R02
param
1 w "handler of 1",! goto continue^R01
2 w "handler of 2",! goto continue^R01
3 w "handler of 3",! goto continue^R01
...
```

Как и положено в компьютерной литературе, этот пример мало похож на более-менее осмысленный, но показывает примерные возможности организации переходов. Из текста первой рутины производится переход к строкам второй, и обратно из второй к строке в первой. При этом можно вычислять смещение для строки перехода. В синтаксической конструкции

```
goto param+param^R02
```

goto означает команду, первое вхождение param означает имя метки, второе вхождение param означает величину смещения в строках относительно метки, и R01 означает имя рутины, в которой эти строки нужно найти.

Кроме того, что в языке MUMPS можно организовывать переходы произвольным образом, добавлением постусловий как для команды, так и для аргумента, можно организовывать также переходы по условию. В частности, следующие синтаксические конструкции не в точности, но приблизительно эквивалентны, за исключением взведения системной переменной $TEST:

```
 i +param goto param+param^R02
 g:param param+param^R02
 g param+param^R02:param
```

Команда XECUTE (X) имеет только аргументную форму, и при выполнении команды значение аргумента используется как последовательность команд. Выполнение команды xecute приводит к выполнению команд из аргумента как если бы в текущей рутине существовала метка с именем не равным ни одной имеющейся метке:

```
uniqlabel
 xecutearg
 quit
```

Перед непосредственно выполнением команд, входящих в аргумент, команда XECUTE создает новый стековый фрейм. После выполнения

всех команд выполняется неявный безаргументный quit. Если при выполнении команд выполнится команда GOTO, то дальнейший код должен выполнить также безаргументную форму команды quit. Кроме того, в самом аргументе XECUTE может стоять команда quit, при ее выполнении выполнение самой команды xecute завершится с очисткой стекового фрейма.

К важным возможностям команды XECUTE относится то, что аргумент вычисляется как строка и может программно собираться для конструирования необходимого для выполнения набора команд и выражений. В частности, это возможность вызвать подпрограмму, имя которой приходит в виде данных, в дополнение к косвенной форме вызова подпрограмм.

При программном конструировании аргумента команды XECUTE нужно лишь следовать синтаксическим соглашениям языка MUMPS.

Важной практической возможностью команды XECUTE является возможность вызвать код, являющийся специфическим для какой-либо реализации или версии MUMPS системы. Пока аргумент является строкой, для транслятора это синтаксически корректная конструкция. Но в самой последовательности команд могут быть использованы синтаксические конструкции, не поддерживаемые текущей реализацией. В частности, это используется в прикладных системах, которые должны использовать специфический и зависимый от реализации код, но в целом комплекс должен работать на любой системе и компилироваться без ошибок. Программа самостоятельно определяет какой код исполнять в аргументе XECUTE в зависимости от текущей версии MUMPS системы.

Другой важной возможностью команды XECUTE является исполнение строки данных, например специфический для текущего устройства ввода - вывода код для выполнения определенного действия. Причем для различных устройств этот код может отличаться. Более того, выполняемый код может отличаться в зависимости от программы работающей с другой стороны ввода - вывода. Например, для одних телнет - клиентов могут потребоваться одни действия по формированию управляющих последовательностей, для других другие. При этом совокупность таких описанных действий может храниться в настройках в базе данных и использоваться в зависимости от текущего подключенного телнет - клиента.

1.1.4 Команды ввода-вывода

В MUMPS среде существует единая стандартная концепция ввода - вывода. Весь ввод-вывод производится с так называемыми устройствами

ввода - вывода посредством команд ввода - вывода. Устройство логически либо открыто и доступно процессу либо нет. Физически устройство может соответствовать самым разным способам ввода - вывода, например чтению с клавиатуры, выводу на консоль, файлу, сокету TCP/IP, межпроцессному каналу, магнитной ленте, и другим. Стандарт языка MUMPS никак не лимитирует, какие физические средства могут быть трактованы как устройства.

Все операции ввода - вывода выполняются лишь с текущим устройством. Процесс может открыть несколько устройств, но команды непосредственно посылающие или принимающие данные, будут использовать только текущий, и им нельзя явно указать, с каким из открытых устройств следует выполнять операции. Чтобы выполнить операцию с другим устройством, его нужно сначала сделать текущим.

Среда MUMPS стандартно поддерживает две системные переменные $IO и $PRINCIPAL, которые возвращают соответственно текущее устройство и устройство по умолчанию. Устройство по умолчанию - это то, которое было автоматически создано при старте процесса, и в большинстве систем MUMPS оно не может быть закрыто, но в некоторых реализациях может быть изменено. Устройство по умолчанию назначается MUMPS процессу исполняющей средой автоматически, в зависимости от способа запуска процесса.

Исключение касается лишь текущего устройства для интерактивных типов устройств типа консоли, терминала, телнета. Если пользователь вводит команды для исполнения, то при их выполнении может переключиться текущее устройство ввода - вывода, но по окончании выполнения команд необходимо снова вернуть управление пользователю, поэтому в частных случаях MUMPS система, выполнив введенные интерактивно команды, самостоятельно переключает текущее устройство в устройство по умолчанию.

Для ввода - вывода среда MUMPS использует команды:

1. OPEN - открытие устройства с необязательными опциями открытия

2. USE - сделать устройство текущим и / или изменить опции использования устройства

3. CLOSE - закрыть устройство с необязательными опциями закрытия

4. WRITE - вывести строку или спецсимвол или отдельный байт в устройство

5. READ - прочитать из устройства строку или отдельный байт

Каждое из устройств должно быть как-то именовано. Система именования стандартом не предусматривается, тут производители различных систем могут поступать как считают нужным. Состав и назначение опций также стандартом не регламентируются.

Есть MUMPS системы, где устройства нумеруются, например для открытия файла надо открыть устройство номер 15, а для принтера номер 30.

Есть MUMPS системы, где устройства строго именуются с лидирующим типом устройства, например для открытия файла нужно открыть устройство с именем

```
"|FILE|c:\temp\io.txt"
```

а для обмена по TCP/IP устройство с именем

```
"|TCP|www.yandex.ru:80"
```

Есть MUMPS системы где характер устройства определяется опцией, например открыть файл на запись в режиме добавления

```
open filename:APPEND
```

или запустить тот же файл на исполнение и обмениваться по стандартным каналам с запущенным процессом:

```
open "file":(COMMAND="cat "_filename:READONLY)::"PIPE"
```

При этом в зависимости от реализации MUMPS системы открытые устройства могут быть во владении всеми процессами MUMPS системы, или только одним тем который открыл устройство. В зависимости от реализации MUMPS система может позволить или не позволить открыть процессу устройство если устройство с таким же именем уже открыто в другом процессе.

Нечеткость требований стандарта в отношении ввода - вывода, с одной стороны, позволили разработчикам MUMPS систем поддерживать множество самых различных способов ввода - вывода, но, с другой стороны, те разработчики кто использует MUMPS, должны сверять код с документацией на используемую MUMPS систему и проверять как именно поддерживается в ней открытие или использование устройства в

зависимости от его физического типа. Обычно такие фунции разработчики выносят в отдельные библиотечные рутины, зависимые от реализации. Для одной MUMPS системы выполняется один код, для другой - другой.

Команда OPEN (O) открывает устройство. В случае если устройство с таким именем уже было открыто, то команда ничего не делает. Если не было открыто, то команда пробует открыть и при успехе имя устройства вносится в список открытых. Если указан таймаут ожидания открытия, то команда дополнительно взводит системную переменную $test в значение 1 при успехе открытия за указанное время и в значение 0 при неуспехе.

Что интересно, открытое устройство не становится текущим. Чтобы далее прочитать из устройства или записать в него, нужно его сделать текущим.

Команда открытия может поддерживать дополнительные опции открытия устройства, в зависимости от реализации MUMPS системы и в зависимости от типа устройства. Например, в опциях могут быть указаны терминатор чтения строки или режим открытия файла для совместного доступа.

Команда USE (U) делает устройство текущим. Если устройство уже было текущим, то оно остается текущим. Если такое устройство не было открыто, то команда генерирует ошибку.

Команда USE также может поддерживать дополнительные опции, чтобы указать как именно использовать устройство или применить к нему какие-то действия. Например, сменить терминатор чтения или изменить позицию в файле.

И команда OPEN и команда USE может поддерживать назначение устройству мнемоник. Это внутренний механизм MUMPS системы для обработки мнемонических имен. Механизм мнемоник может быть реализован различным образом. Во многих MUMPS системах имя обработки мнемоник означает рутину, в которой впоследствии будут искаться метки для обработки самих мнемоник. Как правило, это отдельная рутина, но это необязательное правило, различные MUMPS системы могут придерживаться слегка отличающихся соглашений. В частности, при вызове

```
write /CUP(10,20)
```

указывается что для текущего устройства должна быть вызвана мнемоника CUP с двумя параметрами. В частности именно это имя (CUP) означает позиционирование каретки на экране консоли или телнета. Если для устройства телнета и консоли назначены соответствующие рутины обработки мнемоник, то для одного и того же кода

```
write /CUP(10,20)
```

но для разных устройств будет вызываться код, зависимый от устройства.

Такой механизм позволяет организовать работу программного кода независимо от устройства и работать на самых различных типах телнет - клиентов и терминалов одними и теми же функциями, выполнив назначение рутины мнемоник при подготовке устройства.

Команда CLOSE (C) закрывает устройство и удаляет его из списка открытых. Традиционно MUMPS системы придерживаются правила, что команда CLOSE не может закрыть устройство по умолчанию, эта операция просто игнорируется. Команда CLOSE также может поддерживать дополнительные опции, означающие какие действия необходимо применить к физическому устройству при закрытии. Например, это может быть удаление используемого файла или усечение его размера по текущему положению, или по указанному размеру, или переименование, или что-то иное.

Многие современные MUMPS системы поддерживают нестандартную безаргментную форму команды CLOSE. Но эта команда применяется не к текущему устройству, а ко всем, кроме устройства по умолчанию ($PRINCIPAL). Хотя, если текущее устройство не является устройством по умолчанию, оно также неявно попадет под это правило.

При закрытии текущего устройства MUMPS системой не допускается такая ситуация, что нет текущего устройства. Команда CLOSE в этой ситуации делает текущим то устройство, которое является устройством по умолчанию. В простых случаях, когда именно это и требуется, разработчики не принимают специальных мер по переключению текущего устройства, но в реальных функциях это обычно не приветствуется. Рекомендуется либо до, либо после закрытия текущего устройства явно выполнять назначение текущим то, которое необходимо, либо то, которое было текущим на момент начала выполнения функции.

Основная модель ввода - вывода в MUMPS системах - это модель последовательных устройств. В случае если физические принципы использованного канала являются блочно - ориентированными, команды чтения и записи для такого устройства все равно приводятся к чтению порций. При этом для позиционирования на нужный блок используется команда use с опциями позиционирования, либо система автоматически переходит к следующей порции.

Команда WRITE (W) записывает в текущее устройство строку или символ или специальную управляющую последовательность, или вызывает мнемонику. Для команды write синтаксически поддерживается

несколько специальных типов управляющих последовательностей:

1. write # - очистить текущий экран или перевести страницу в принтере, в зависимости от типа устройства
2. write *code - вычислить значение выражения code как целое число и вывести в устройство один байт с этим кодом
3. write ! - вывести перевод строки
4. write ?code - вычислить значение выражения code как целое число и вывести дополнение в виде пробелов или использовать иной способ позиционирования до указанной колонки в зависимости от типа устройства
5. write /MNEM или write /MNEM(param1,param2,...) - вызвать обработчик мнемоники MNEM без параметров или с параметрами

Любые другие выражения команда write вычисляет как строку и полученную последовательность байт записывает в текущее устройство. С записью строки в различных реализациях MUMPS также могут быть нюансы, например система может поддерживать таблицу трансляций символов. В этом случае можно назначить такую таблицу трансляций что запись из программы на MUMPS ведется в одной кодировке, но благодаря автоматической трансляции реальная запись в физическое устройство (файл, сокет) ведется в другой кодировке.

Небольшой пример формирования простой странички html командами WRITE:

```
write "<html>",!
write "<head><title>Hello</title></head>",!
write "<body>",!
write "Hello, world!",!
write "</body></html>",!
```

Если такое содержание вместе с соответствующим заголовком http ответа процесс отправит web-браузеру, то он примет ее как html страницу. Например, процесс может открыть устройство типа tcp соединения и при обнаружении входящих соединений читать заголовок http запроса, выяснить что запрашивалось и сформировать командой write полный http ответ. В этом случае MUMPS процесс будет выполнять роль web - сервера.

Команда READ (R) читает строку или код одного байта из текущего устройства. Команде read можно указывать имя переменной

```
read varname
```

для чтения строки или имя переменной со звездочкой

```
read *varname
```

для чтения одного байта и возврата в указанную переменную его кода.

Кроме того, команде read можно указывать константные строки (не числа, если требуется числа то должны быть в кавычках строки) и символы специального форматирования:

1. read # - очистка экрана
2. read ! - вывод перевода строки
3. read ?code - вычисление значения code как числа и позиционирование в соответствующую колонку в зависимости от типа устройства

Также как и для команды write, аргументы команды read могут быть комбинированы через запятую последовательно:

```
read !,"Enter Your Name: ",name,!
```

Здесь команда read сначала переведет строку и с начала строки напечатает строку "Enter Your Name: ", затем будет ожидать ввода строки, и по окончании ввода запишет введенный результат в переменную name, после чего снова выполнит перевод строки на экране.

По умолчанию для интерактивных устройств типа консоли, терминала или телнета, индикатором окончания ввода служит символ с кодом 10 как подтверждение ввода (Enter) либо символ Escape с кодом 27 как отмена ввода.

Кроме того, для устройства может быть явно назначен терминатор чтения, обычно это выполняется в опциях команде use. В этом случае команда read вводит все пока не встретит терминатор чтения целиком, все входящие в него байты в нужной последовательности.

Кроме того, команде чтения можно назначить время ожидания ввода. В этом случае команда будет вводить все что окажется в устройстве в течении указанного времени. По истечении времени команда возвращает управление.

Кроме того, команде read можно назначить явно длину чтения. Например, читать первые 12 символов.

Разработчик может скомбинировать все три условия, и терминатор чтения, заданный в команде use, и время ожидания, и длину ввода. В

этом случае команда read возвращает управление как только достигнуто любое из указанных условий, в зависимости от того, какое наступило раньше.

Большинство реализаций MUMPS систем поддерживает отключение режима эха при вводе строки, чтобы иметь возможность ввести пароль, не показывая его на экране. При необходимости выполнить такую функцию необходимо проверить в документации на используемую MUMPS систему как именно в ней отключается и снова включается режим эха.

Кроме стандартных команд для ввода - вывода многие реализации поддерживают свои собственные расширенные команды, например, в зависимости от реализации это могут быть

1. PRINT, ZPRINT - вывод в текущее устройство текста или части текста рутины
2. ZWRITE - печать переменной со всеми ее узлами и значениями
3. ZZDUMP - печать значения в виде шестнадцатеричного дампа

Команды PRINT и ZPRINT относятся к устаревшим, и уже есть MUMPS системы, которые их не поддерживают из-за неактуальности. В те времена, когда основным средством доступа к компьютеру были даже не телнет по сетям TCP/IP, а соединение компьютеров по COM портам через платы мультиплексоров, алфавитно - цифровой режим доступа был основным. В настоящее время уже давно существуют полноценные развитые GUI средства разработки, показывающие рутины как необходимо, и командами типа PRINT и ZPRINT уже не пользуются.

Собственно, в сам стандарт языка команды PRINT и ZPRINT не входят, и эти команды были поддержаны различными реализациями самостоятельно с существенно совместимым друг с другом синтаксисом, чтобы дать возможность простым способом просматривать на терминальном устройстве части рутины или рутину полностью.

Команды ZWRITE и ZZDUMP относятся к командам для разработчиков, чтобы показать в специальном виде переменные и их значения. Например, пусть есть глобальная переменная

```
USER>s ^ABC(3)=456

USER>s ^ABC(8,"width")=12.7

USER>s ^ABC("total")=45
```

тогда ее можно вывести с имеющимися в ней подиндексами и значениями:

```
USER>zw ^ABC
^ABC(3)=456
^ABC(8,"width")=12.7
^ABC("total")=45
```

Команда ZZDUMP выводит шестнадцатеричную распечатку значения так что разработчик может видеть точное значение каждого из байт входящих в строку:

```
USER>zzdump "string"
0000: 73 74 72 69 6E 67              string

USER>zzdump $lb(1,2,3)
0000: 03 04 01 03 04 02 03 04 03     .........
```

В любом случае разработчик должен свериться с документацией на используемую MUMPS систему, какие дополнительные команды ввода - вывода поддерживаются.

В комплексе с командами ввода - вывода идут системные переменные ввода - вывода. К ним относятся:

1. $IO - переменная содержит имя текущего устройства по принятым в MUMPS системе правилам именования

2. $PRINCIPAL - переменная содержит имя устройства по умолчанию по принятым в MUMPS системе правилам именования

3. $X, $Y - переменные содержат значения координат каретки для текущего устройства

В зависимости от реализации также могут поддерживаться расширенные системные переменные, например $ZEOF.

Если системная переменная $zeof поддерживается, то она содержит индикатор окончания данных в устройстве. Программа может проверять значение системной переменной вместо перехватывания ошибки чтения по истечении данных. Обычным соглашением для переменной $zeof является то что она индицирует состояние конца данных для текущего устройства. Если сменить устройство на другое, то значение $zeof будет возвращаться уже для него.

Основные современные реализации MUMPS систем поддерживают вполне достаточное для комфортной работы количество различных типов устройств, включая LPT, COM порты, принтеры, сокеты, файлы, межпроцессное взаимодействие, и другие.

1.1.5 Служебные команды

К служебным командам относятся команды управления процессами, их состоянием и взаимодействием между процессами. Под процессами здесь понимается процесс MUMPS системы, или логический job. В зависимости от реализации логический job MUMPS системы может соответствовать физическому процессу, физическому потоку или самостоятельно переключаемому контексту в операционной среде, архитектурно не являющейся многозадачной, такой как DOS.

Надо отметить, что стандарт MUMPS изначально ориентирован на многозадачный контекст MUMPS системы и изначально предусматривает логическую многозадачность, даже если реализация MUMPS выполнена как однопользовательская. В этом случае система поддерживает все синтаксические конструкции, включая список процессов (состоящий из одного). Если система не может выполнить команду job, например, то она генерирует ошибку, но синтаксически эта команда поддерживается.

К служебным командам MUMPS относятся команды:

1. HANG - команда приостанавливает выполнение процесса на указанное время задержки

2. JOB - команда запускает другой процесс

3. LOCK - команда запрашивает или снимает блокировку

4. VIEW - команда обращается к внутренним зависимым от реализации структурам данных процесса или MUMPS системы

Команда HANG (H) имеет обязательный аргумент. В случае если используется аббревиатура и безаргументная форма (одна буква H), то для MUMPS такая команда считается командой HALT.

Формально, в стандарте определено, что аргумент задает время приостанова выполнения процесса в секундах. Однако, в этом пункте у различных реализаций имеются расхождения в трактовке таймаута. Существуют, по меньшей мере, такие трактовки таймаута для HANG:

1. Время используется в секундах и секунда считается единицей измерения, но величина может быть задана с точностью до миллисекунд.

2. Время округляется до целого числа секунд и секунда используется и как единица измерения и как точность задания таймаута.

Неприятность второго случая в том, что если округление дало нулевую величину, то задержка не выполняется и команда может вообще не выполнить обращение к процедуре приостанова. В случае если задержка на малую величину вставляется в цикл для отдачи процессорного времени, то в этом случае процесс может вообще не отдать процессорное время на время своего ожидания.

При этом характер отсчета задержки также может варьироваться в зависимости от реализации:

1. Задержка отсчитывается от начала выполнения команды.

2. Задержка прерывается при достижении соответствующего внутреннего кванта переключения контекстов job.

Во втором случае проявляется эффект нестабильного времени задержки с непредсказуемым характером округления как в меньшую, так и в большую сторону. И чем меньше время задержки, тем больше относительная погрешность ее выполнения.

Если частью сервера приложений, написанного на MUMPS, является фрагмент игрового характера, для которого важен таймаут с величинами менее секунды, то разработчикам необходимо проверять поведение задержек на каждой из MUMPS систем, которые предполагается использовать. Характер реализации задержки может совпадать также для других команд, использующих таймауты, для таких команд как LOCK, JOB, READ или OPEN.

Команда HANG выполняет безусловную задержку. В случае если пользователю нужно дать возможность прервать выполнение задержки, рекомендуется использовать команду чтения символа READ с задержкой. При ее выполнении, если произошел ввод символа с клавиатуры, то команда возвращает управление до истечения задержки.

Команда JOB (J) полностью аналогична команде DO (D) за двумя исключениями:

1. После выполнения команды порождается новый процесс MUMPS и управление возвращается вызвавшему процессу.

2. Нельзя передать локальные переменные по ссылке.

Если есть подпрограмма, например такая:

```
label(param,...)
 ; commands
 ...
 q
```

то ее можно вызвать как подпрограмму командой DO и тогда она выполнится как часть последовательного выполнения процесса. Либо ее же можно вызвать командой JOB, и тогда породится новый процесс и его выполнение начнется с этой метки в этой рутине и параметры param станут локальными переменными нового процесса.

Синтаксис вызова полностью аналогичен команде do:

```
do label^rtn(expr1,expr2,...)
job label^rtn(expr1,expr2,...)
```

Кроме того, команде job можно через двоеточие указать дополнительные опции запуска процесса и таймаут ожидания запуска. В случае если таймаут указан, то команда записывает в системную переменную $test значение 1 при успехе запуска дочернего процесса или 0 при неуспехе.

Команда job возвращает управление сразу как только выполнен запуск дочернего процесса либо стало известно, что такое действие невозможно по каким-либо причинам. Команда job не ожидает завершения выполнения дочернего процесса.

Состав и назначение дополнительных опций команды job определяются реализацией и версией MUMPS системы. Традиционно в них специфицируются, например, передача дополнительных локальных переменных целиком в виде дерева, передача конкурентного сокета, указание какой объем памяти необходимо использовать, в какой базе данных начать новый процесс, и другие параметры. В случае использования опций команды job необходимо проверить в документации на используемую MUMPS систему их состав и точный синтаксис.

Общий формат команды job специфицируется так:

```
job label^routine(params,...):(options,...):timeout
```

Также, как и для команды do, может быть опущено имя рутины, в таком случае для старта нового процесса используется текущая рутина.

Традиционно, различные реализации MUMPS систем поддерживают специальные расширенные системные переменные чтобы получить информацию о номере запущенного дочернего процесса в родительском процессе и о номере родительского процесса в дочернем процессе. Например, это могут быть системные переменные $zparent и $zchild.

После запуска дочерний процесс выполняется параллельно и не связанно с родительским, и любой из них может завершить работу независимо друг от друга.

Команда LOCK (L) занимает или освобождает блокировку. Сама блокировка имеет имя совпадающее с именем локальной или глобальной

переменной. При этом существование и значение этих локальных или глобальных имен для команды lock не имеют значения. Доступ к этим переменным ничуть не препятствуется.

По своему назначению блокировка - это механизм синхронизации двух или более процессов в рамках одного сервера MUMPS системы. Если процесс может получить блокировку, то он получает и продолжает выполнение. Если не может, то выполнение процесса приостанавливается до обнаружения состояния когда получить блокировку становится возможным.

Блокировки, установленные процессом, имеют время жизни либо до ее снятия самим процессом-владельцем, либо до окончания процесса. Многие современные реализации MUMPS систем имеют возможности принудительно снять блокировки, установленные в другом процессе. Этот функционал уже находится за пределами стандарта и выполняется средствами, зависимыми от реализации.

Команда VIEW (V), как и системная функция $view(), введена специально для вынесения в нее всех возможностей, специфических для реализации MUMPS системы, например, просмотр значений операционного окружения, значения внутренних структур памяти, прямого управления сбросом буферов и многого другого.

Что интересно, сама команда VIEW стандартом предусмотрена, но назначение и точный синтаксис оставлен на усмотрение реализации. При портировании программ, разработанных на MUMPS, обычно именно использование команды view, функции $view и расширенных системных переменных начинающихся на $Z, является предметом внимания для поиска эквивалентной замены по функционалу.

1.1.6 Постусловия

Постусловия - это условия выполнения или использования, записанные после команды или аргумента команды, к которым они применяются. В других языках аналогичная конструкция не встречается и может вызвать первое время непонимание. В действительности это некоторый более короткий аналог команды if, но локального действия.

Синтаксически постусловие отделяется двоеточием и является произвольным вычисляемым выражением. При выполнении программы исполняющая системы вычисляет указанное в постусловии выражение. После чего приводит его к числу и сравнивает с нулем. Если результат получится эквивалентом числового нуля, то постусловие считается ложным, если результат - числовой не ноль, то постусловие считается истинным.

Постусловие команды применяется к команде. Если команде указано несколько аргументов, то постусловие распространяется на все эти аргументы. Общий синтаксис:

```
cmd:postcond arg1,arg2,....
```

Некоторые команды, например, команды GOTO и DO с аргументами допускают указание постусловия для аргумента. В этом случае постусловие указывается через двоеточие для аргумента. Общий синтаксис постусловия для аргумента:

```
cmd arg1:postcond1,arg2:postcond2,...
```

В некотором смысле постусловие это укороченный аналог команды if но с применением не ко всей последующей строке а только к указанной команде или аргументу. В отличие от команды if, которая изменяет значение системной переменной $test, постусловие ни в какой форме его не изменяет, если конечно самостоятельно не содержит соответствующих побочных эффектов.

Если постусловие применено к команде, то при его невыполнении команда не выполняется, но управление все равно передается следующей команде.

Если постусловие применено к команде с несколькими аргументами, то оно применяется к всем указанным аргументам. Например, если при выполнении кода

```
write:a*b 123,!,456,! write 789,!
```

условие a * b вычислилось как 0, то команда

```
write 123,!,456,!
```

не выполняется полностью, но команда

```
write 789,!
```

все равно выполняется.

Постусловие для команд не может быть указано для условных команд IF, ELSE и FOR. В самом деле, положим что в языке разрешено применять постусловие к командам IF, ELSE, FOR. В этом случае мы должны в зависимости от значения постусловия применить команду к аргументу. И, если постусловие не выполнилось, то перейти к следующей команде. В этом случае мы при ложном постусловии все равно выполняем команды, расположенные после команды IF, и один раз выполняем

команды расположенные в теле цикла, после команды FOR, несмотря на собственные правила выполнения цикла, описанные в команде FOR. Таким образом, разрешение постусловий для условных команд приводит к недоразумениям.

Хотя постусловное выражение записывается *после* ключевого слова команды, но вычисляется и проверяется *перед* выполнением или невыполнением команды.

Даже в случае, если постусловное выражение не выполняется (результат вычисления равен нулю), тем не менее, выражение вычисляется, и вычисление может иметь побочные эффекты. На это разработчик обязательно должен обращать внимание.

Постусловие для аргумента указывает применять ли постусловие к этому аргументу если команда выполняется. Не все команды поддерживают постусловия для аргумента. В частности, команды do и goto поддерживают постусловия на аргументы.

Если постусловие аргумента не указано, то команда применяется к нему, иначе, сначала вычисляется постусловие аргумента, проверяется как число на равенство нулю, и в зависимости от результата команда применяется к этому аргументу, либо применяется к следующему, если он есть. При этом, если есть постусловие как на аргумент команды, так и на команду, то сначала вычисляется и проверяется постусловие на команду, а затем, если оно выполняется, вычисляются и проверяются постусловия на аргументы. Например:

```
do:pc label1:pc1,label2:pc2
```

Здесь, если выражение pc вычисляется как 0, то все аргументы и постусловия аргументов игнорируются, и управление переходит к следующей команде. Иначе, далее вычисляется и проверяется постусловие pc1, и если выполняется, то вызывается подпрограмма с метки label1. После обработки первого аргумента управление переходит ко второму, вычисляется и проверяется постусловие pc2. Если оно выполняется, то вызывается подпрограмма с метки label2.

В случае команды goto управление безусловно передается первому из аргументов, для которого выполнилось постусловие, и дальнейшие проверки не выполняются.

1.2 Операторы

Операторами называются действия над значениями, в результате которых получается также значение.

Операторы используются для укороченной записи специальных часто используемых функций, и используют специальные символы для облегчения чтения и записи действий. Например, для записи сложения можно использовать функцию, принимающую два аргумента, и имеющую, например, имя sum или summa или add, или иное, или обозначить одним символом плюс:

```
a+b
```

Конечно, вариант с записью такого действия в виде отдельно определенного оператора намного читабельнее.

В языке MUMPS используются операторы двух видов - унарные (префиксные), применяемые к одному аргументу, и бинарные (инфиксные), применяемые к двум аргументам.

Унарный оператор в префиксной форме применяется к аргументу следующим за ним. Примеры унарных операторов языка MUMPS:

```
+varname
-varname
'logical
```

Бинарные операторы в инфиксной форме применяются к двум аргументам, и располагаются между ними. Примеры бинарных операторов языка MUMPS:

```
a+b
a*b
```

Часть бинарных операторов может состоять из одного символа, часть из двух. В случае если оператор обозначен двумя символами, для него дается отдельная трактовка, например два символа используется для оператора отрицания логического оператора:

```
a'=b
a'>b
a'<b
a']b
```

Определение оператора отрицания оператора задается как последовательное применение сначала оригинального оператора, затем к результату применяется оператор логического отрицания. В частности, эквивалентны замены:

```
a'=b  ->  '(a=b)
a'>b  ->  '(a>b)
a'<b  ->  '(a<b)
a']b  ->  '(a]b)
```

При этом, некоторые современные реализации MUMPS систем, например MiniM и Caché, дополнительно поддерживают нестандартные расширенные операторы из двух символов. В частности, операторы сравнения и ленивой логики

```
a>=b
a<=b
a||b
a&&b
```

Применение расширенных нестандартных операторов, конечно, должно входить в соглашения, принятые при разработке - можно ли их применять или нет, поскольку при портировании кода на другую MUMPS систему они могут как поддерживаться, так и не поддерживаться.

Самым важным в применении операторов языка MUMPS для программистов, работавших с другими языками, является правило отсутствия приоритетов операторов. Все вычисления выполняются слева направо, как они написаны, и изменение порядка необходимо задавать явно, указав круглыми скобками очередность вычисления выражений. Так, если есть выражение

```
a+b*c
```

то интерпретатор MUMPS выполняет сначала сложение, затем умножение. Если программисту необходимо, чтобы сначала было выполнено умножение, а затем сложение, это нужно описать явно:

```
a+(b*c)
```

Для начинающих разработчиков на MUMPS этот факт некоторое время может быть источником проблем. В частности, при портировании кода вычисления выражений написанных на языке, использующем приоритеты операций, или при модификации выражения, записанного без скобок, или при модификации выражения, использующем порядок вычисления слева направо.

Так, если было условие

```
a=b
```

и к нему необходимо добавить логическое ИЛИ с условием

```
c=d
```

то недостаточно дописать

```
a=b!c=d
```

Такое выражение будет соответствовать не ожидаемому

```
(a=b)!(c=d)
```

а совсем другому. Если записать его же с указанием порядка скобками, то оно будет вычисляться так:

```
((a=b)!c)=d
```

Так что получится что значение d будет сравниваться с результатом вычисления

```
(a=b)!c
```

Что интересно, автору доводилось видеть именно такую некорректную замену, причем этот код работал не проявляя ошибки больше полугода, потому что значения переменных перед вычислением принимали такие значения, что результат, вычисленный неправильно, подходил под правильный ответ.

Кроме операторов, вычисляющих выражение, в языке MUMPS также присутствует входящий в стандарт оператор отрицания кода шаблона, сам по себе не вычисляющий выражение, а формирующий правило сопоставления шаблону:

```
USER>w "a"?1N
0
USER>w "a"?1'N
1
```

Здесь первый шаблон задает правило "один символ цифра", а второй отрицает тип символа и задает правило "один символ не цифра". С оператором отрицания типа символа в шаблоне у разработчиков могут возникнуть затруднения при его использовании, поскольку этот оператор довольно сложен в выполнении и не все реализации MUMPS его поддерживают.

В дополнение к стандартным операторам системы MiniM и Caché поддерживают операторную запись битовых операций и порядка их выполнения для битовых строк в аргументе функции $BITLOGIC:

```
& (И, бинарный)
| (ИЛИ, бинарный)
~ (НЕ, унарный)
```

Аргументами оператора могут быть любые значения, если они могут быть вычислены. Это могут быть константы, значения локальных, системных, или глобальных переменных, результат возврата системных и пользовательских функций, результаты других операторов. Если интерпретатор встречает использование оператора, то его аргументы синтаксически рассматриваются именно в этом контексте. В частности, для указания что необходимо вызвать подпрограмму с возвращаемым значением, необходимо перед меткой указать символы $$, иначе литерал метки интерпретатор будет оценивать как имя переменной или как число, если имя метки состоит из цифр.

В отличие от других языков, где есть предварительная декларация типов, в языке MUMPS все литералы и имена рассматриваются как синтаксические конструкции по контексту операции. В частности, различные элементы языка могут иметь совпадающие имена, и интерпретатор оценивает тип элемента не по его имени, а по синтаксическому способу его использования:

1. name - локальная переменная
2. name(a,b,...) - индексированная локальная переменная
3. ^name - глобальная переменная
4. ^name(a,b,...) - индексированная глобальная переменная
5. $name - системная переменная
6. $name(a,...) - системная функция
7. $$name - пользовательская переменная в текущей рутине
8. $$name(a,...) - пользовательская функция в текущей рутине
9. $$name^rname - пользовательская переменная в рутине rname
10. $$name^rname(a,...) - пользовательская функция в рутине rname

Операторы языка по своему действию могут быть сгруппированы по различным критериям, наиболее часто используется критерий группирования по типу действия - арифметические (численные), строковые (литеральные), логические (булевские).

Формально, такое деление условно и не строгое, поскольку строковые и арифметические операторы могут быть использованы для вычисления значения используемого как логическое условие, а логические операторы могут быть использованы для вычисления строкового значения ("1" или "0").

Арифметические операторы выполняют действия над операндами рассматривая их как числа и выполняют числовые операции. Множество операторов языка покрывает традиционный для языков программирования набор операций. К таким операторам в языке MUMPS относятся:

1. Унарный плюс (+), применяется к аргументу справа от оператора, приводит аргумент к числовому значению.

2. Унарный минус (-), применяется к аргументу справа от оператора, приводит аргумент к числовому значению и меняет знак на противоположный.

3. Сложение (+), возвращает арифметическую сумму операндов.

4. Вычитание (-), возвращает арифметическую разность левого и правого операндов.

5. Умножение (*), возвращает произведение операндов.

6. Деление (/), возвращает отношение левого и правого операндов.

7. Целочисленное деление (\), возвращает целую часть отношения левого и правого операндов.

8. Возведение в степень (**), возвращает результат возведения левого операнда в степень равную второму операнду.

9. Деление по модулю (#), возвращает остаток от деления нацело первого операнда на второй операнд.

10. Больше (>), возвращает булевское значение числового сравнения операндов, проверяет что левый операнд арифметически больше второго.

11. Меньше (<), возвращает булевское значение числового сравнения операндов, проверяет что левый операнд арифметически меньше второго.

При вычислении арифметического оператора интерпретатор получает число, но в некоторых случаях это технически невозможно сделать. К таким случаям относится деление на ноль, деление нацело на ноль и остаток от деления на ноль. В такой ситуации MUMPS система генерирует ошибку.

В случае если интерпретатор получает результат, который не может быть представлен во внутреннем представлении интерпретатора, например слишком маленькое или слишком большое число, интерпретатор также генерирует ошибку. Но эти значения уже определяются внутренней реализацией MUMPS системы и могут зависеть в том числе и от версии MUMPS системы.

Строковые операторы выполняют строковые операции над операндами, и перед вычислением результата приводят значения операндов к строковым. К таким операторам в языке MUMPS относятся:

1. Конкатенация (_), возвращает строку как результат литерального соединения левого и правого операнда.

2. Равно (=), возвращает булевское значение равенства операндов в их строковом представлении.

3. Проверка на вхождение ([), возвращает булевское значение содержится ли правый операнд в левом литерально, или выполняет операцию поиска.

4. Следует (]), возвращает булевское значение литерального следования литеральных представлений операндов, следует ли левый операнд в строковом сравнении после правого.

5. Сортируется после (]]), возвращает булевское значение индексного следования литеральных представлений операндов, следует ли левый операнд в индексном сравнении после правого.

6. Проверка по шаблону (?), возвращается булевское значение результата сопоставления литерального значения левого операнда шаблону задаваемому правым операндом.

Хотя оператор равенства (=) определен для литеральных представлений, его также используют для сравнения и чисел, и комбинаций чисел и строк.

Оператор "сортируется после" использует то же правило для сравнения значений, что и индексная сортировка значений индексов переменных. Зачастую используется совместно с функцией $ORDER при перечислении индексных значений.

Логические операторы приводят значения операндов к булевской величине и возвращают результат логической операции. К таким операторам в языке MUMPS относятся:

1. Унарный НЕ ('), возвращает логическую инверсию операнда.
2. Логический И (&), возвращает результат операции логического И над операндами.
3. Логический ИЛИ (!), возвращает результат операции логического ИЛИ над операндами.

В дополнение к стандартным операторам системы MiniM и Caché поддерживают расширенные арифметические и логические операторы:

1. Больше или равно (>=), возвращает булевское значение числового сравнения операндов, проверяет что левый операнд арифметически больше или равен второму.
2. Меньше или равно (<=), возвращает булевское значение числового сравнения операндов, проверяет что левый операнд арифметически меньше или равен второму.
3. Ленивый И (&&), выполняет логическую операцию И над операндами, но не вычисляет второй если результат вычисления первого определяет истинность результата.
4. Ленивый ИЛИ (||), выполняет логическую операцию ИЛИ над операндами, но не вычисляет второй если результат вычисления первого определяет истинность результата.

Ленивые логические операторы, в силу того, что часть выражения может не вычисляться, имеют отличное от строгих логических операторов поведение побочных эффектов. В случае если производится замена строгих логических операторов на ленивые, то разработчикам необходимо проверить, используется ли в выражениях побочный эффект и необходимо ли его сохранить.

1.3 Переменные

В языке MUMPS переменными называются элементы, не имеющие аргументов, и имеющие значение. В зависимости от вида переменной она

может иметь или не иметь индексов. В англоязычной литературе индексы переменных в языке MUMPS имеют отдельное название subscript. В русскоязычной литературе слово "индекс" применяется как к индексам переменных, так и к индексным структурам.

Переменные могут быть использованы в любом месте, где допускается использование значений. Значение переменной определяется тем, что это за переменная и значение может не изменяться в течение всего времени работы сервера (например, $SYSTEM), не изменяться в контексте одного процесса (например, $JOB), изменяться в течении времени без явного ей присваивания (например, $H) или изменяться в зависимости от результата работы команд (например, $TEST).

В MUMPS переменные бывают следующих видов:

1. Локальные, указываются как только имя с опциональными индексами.

2. Глобальные, указываются как имя с лидирующим символом циркумфлекса (^), и с опциональными индексами.

3. Системные, указываются как имя с лидирующим одним символом $.

4. Структурные системные, указываются как имя с лидирующими символами ^$ и имеющие определенную структуру и значение индексов.

5. Пользовательские, указываемые как имя с двумя символами $$, и не имеющие индексов.

В тех местах, где допускается использование имен переменных (взятие строки с именем по имени переменной и наоборот, косвенность имени по строке с именем), всегда можно использовать локальные и глобальные переменные. В некоторых случаях реализации MUMPS систем допускают использование также имен структурных системных переменных взаимозаменяемо с локальными и глобальными именами. Пользовательские переменные могут быть использованы лишь для получения значения. Системные переменные, в зависимости от определения и назначения, могут допускать либо только чтение, либо и чтение и запись.

Локальные переменные существуют только в контексте процесса, ими владеющем. При старте процесса он не имеет значений локальных переменных, при завершении процесса его локальные переменные пропадают.

Традиционно программы серверного класса работают на строго ограниченной области памяти, в том числе выделяемоя для локальных переменных. Локальные переменные не могут быть созданы если исчерпана память для локальных переменных.

Локальные переменные отличаются от остальных переменных тем, что для них априори применима операция new с созданием новой области видимости таким образом, что новые значения этой переменной и все ее изменения происходят лишь начиная с определенного уровня стека, а предыдущие ее же значения остаются неизменными. Этот механизм называется стекованием переменных. Кроме локальных переменных, могут также явно стековаться отдельные системные переменные ($ETRAP, $ESTACK) и автоматически стекуется системная переменная $TEST.

Локальные переменные хранятся только в определенной памяти процесса и видны только ему. Каждый из процессов имеет собственные локальные переменные, не связанные друг с другом. Стандарт языка не предусматривает доступ одного процесса к переменным другого, но отдельные реализации в отладочных целях могут предоставлять системно - зависимый способ прочитать или изменить локальные переменные другого процесса.

Глобальные переменные хранятся в базе данных и доступны всем процессам на чтение и запись. Именно наличие общего доступа к глобальным переменным и организация специальных операций по отношению к ним (бекап, транзакции, атомарность доступа) превращают MUMPS систему в СУБД.

Глобальные переменные обычно организуются по своему хранению и размещению в базах данных, состоящих из наборов файлов, в зависимости от специфики MUMPS системы. Каждая из таких баз данных имеет свой набор глобальных переменных и процессы имеют к ним всем параллельный доступ.

Системные переменные имеют определенное имя и возвращаемое значение. Например, системная переменная $JOB возвращает внутренний идентификатор текущего процесса, а системная переменная $HOROLOG возвращает текущие дату и время в виде чисел, отсчитываемые по определенному MUMPS стандарту.

Все системные и структурные системные переменные существуют у процесса всегда, и являются частью определения языка. Их нельзя удалить так, чтобы они синтаксически отсутствовали или приняли неопределенное значение.

Структурные системные переменные имеют индексы и определенное стандартом и реализацией имена и значения индексов. Например, переменная ^$JOB содержит в качестве индексов первого уровня идентифи-

каторы имеющихся в MUMPS системе процессов, а переменная ^$LOCK - имена блокировок.

Структурные системные переменные предназначены для перечисления наборов данных путем перебора значений индексов. К ним могут применяться операции взятия имени и косвенности. В некоторых случаях реализация MUMPS может дополнительно поддерживать возврат значения структурной системной переменной. В частности, MiniM в качестве значения ^$LOCK(lockname) возвращает номер процесса владельца и счетчик захвата блокировки с именем lockname.

Структурные системные переменные существуют всегда, и существование их значений определяется существованием информационных объектов, для которых они предназначены. Отдельные реализации дополнительно могут поддерживать явные операции удаления элемента из структурной системной переменной, например MiniM поддерживает

```
kill ^$JOB(pid)
```

для принудительного завершения процесса с идентификатором pid или

```
kill ^$LOCK(lockname)
```

для принудительного снятия блокировки независимо от процесса - владельца.

Но такой функционал является нестндартным и при необходимости портирования программ такой функционал нужно изолировать в рутины для замены на эквивалентный в целевой MUMPS системе.

К стандартным структурным системным переменным языка MUMPS относятся следующие:

1. ^$LOCK - в качестве индексов содержит имена блокировок, локальных или глобальных.

2. ^$ROUTINE - в качестве индексов содержит имена рутин текущей базы данных.

3. ^$GLOBAL - в качестве индексов содержит имена глобалов текущей базы данных.

4. ^$JOB - в качестве индексов содержит идентификаторы имеющихся на сервере процессов.

5. ^$DEVICE - в качестве индексов содержит идентификаторы открытых устройств (либо только для текущего процесса либо для всего сервера, в зависимости от реализации MUMPS системы).

Кроме основных стандартных структурных системных переменных комитет MDC также предлагал поддерживать дополнительные

1. ^$CHARACTER для описания наборов символов и правил их сравнения.

2. ^$EVENT для описания объявленных событий.

3. ^$LIBRARY для описания подключенных внеших библиотек.

4. ^$SYSTEM для описания системных характеристик текущей MUMPS системы.

В отношении всех остальных структурных системных переменных, если они поддерживаются MUMPS системой, должно применяться правило их именования начиная с символа Z.

Многие MUMPS системы поддерживают определение собственных Z системных переменных программистами так, что поведение этой переменной определяется кодом на языке MUMPS. В различных системах такой механизм может немного отличаться, но в целом методика аналогична - поведение определяется либо заданием метки возвращающей значение, либо метки принимающей значение.

В дополнение к стандартным, в системе Caché дополнительно существует механизм программного здания структурной системной переменной, определенной разработчиком. Её имя при использовании должно начинаться на символ Z и определение ее поведения задается рутиной со специальным именем, содержащей специальные метки.

Стандартные системные и структурные системные переменные, а также большинство дополнительно поддерживаемых различными реализациями, допускают сокращение до одной или нескольких первых символов.

Пользовательские переменные фактически являются пользовательскими функциями без аргументов. При их вызове не указываются аргументы и скобки. Для пользовательских переменных разработчик описывает рутину, метку, и код вычисления возвращаемого значения.

Различные реализации MUMPS систем поддерживают в дополнение к стандартным системным и структурным системным также собственные дополнительные переменные, имеющие имена и значения определяемые реализацией. И имена обычно начинаются на символ z. Кроме того, некоторые системы, например MiniM и Caché, предоставляют механизмы пользовательского определения системных переменных. Разработчик

может дать определение, например, системной переменной $ZPOS и далее использовать это имя также как встроенную системную переменную, в том числе и нечувствительно к регистру.

Имена локальных, глобальных и пользовательских переменных чувствительны к регистру. Имена системных и структурных системных переменных нечувствительны, и могут использоваться в любом регистре. Кроме того, для большинства из них действуют соглашения о синонимичесих аббревиатурах. Например, допускаются замены

```
$PRINCIPAL  ->  $P
$HOROLOG    ->  $H
$ECODE      ->  $EC
```

и обратно.

В отличие от многих других языков, в MUMPS отсутствуют переменные, автоматически создаваемые и доступные для процесса или модуля и имеющие определенную структуру и значения (статические переменные). Для того, чтобы локальная переменная существовала, необходимо, чтобы процесс создал ее значения самостоятельно.

1.4 Числа и строки

В языке MUMPS отсутствует типизация значений, переменных, возвращаемых значений, результатов операций. Все значения, с которыми оперирует интерпретатор языка, являются формально строками. При этом в языке действуют правила операций, определенные над строками, в совокупности с которыми язык является полноценным языком программирования.

Общее правило состоит в том, что реализация MUMPS системы скрывает от разработчика внутреннее представление данных так, что для разработчика все значения выглядят строго как строки. При этом содержимое всегда приводится к необходимому виду не при хранении данных, а при применении контекста операции.

В частности, если внутреннее значение оказалось строкой, но необходимо выполнить арифметическую операцию, система приводит строку к числу по определенным правилам. Если внутреннее значение оказалось числом, но необходимо выполнить строковую операцию, система также приводит число к строке по определенным правилам. Например, система всегда может умножить две пустые строки или взять третий байт числа.

Правила оперирования значениями и приведения числа к строке и строки к числу жестко определены стандартом и являются одной из

основ переносимости программ, написанных на MUMPS. Разработчики всегда уверены, что различные реализации MUMPS систем всегда оперируют одними и теми же соглашениями о преобразованиях данных и всегда получают одинаковый результат.

Для задания константной строки в языке MUMPS нужно ограничить последовательность символов, входящих в строку, двойными кавычками, например:

```
USER>w "this is a string"
this is a string
USER>w "this is another string"
this is another string
```

При этом в строке могут быть использованы любые символы кроме символа перевода строки. Если используемый редактор сможет ввести исходный текст со строкой с произвольными непечатными символами, то все они будут содержаться в строке. Например, символ табуляции.

Поскольку символ двойной кавычки воспринимается транслятором как символ окончания строки, для того чтобы этот символ ввести в константной строке, вместо него нужно указать удвоенный символ двойной кавычки:

```
USER>w "this is a ""string"""
this is a "string"
USER>w "this is ""another"" string"
this is "another" string
```

Для того, чтобы строковое значение содержало произвольные символы, включая символ перевода строки, который нельзя ввести в константе, разработчики комбинируют константные строки, системную функцию $CHAR и оператор конкатенации, например строка

```
"first"_$C(10)_"second"_$C(9)
```

после вычисления выражения содержит символы и переноса строки и табуляции.

Но нужно понимать, что в этом случае мы задаем лишь последовательность байт, и вывод такой строки (содержащей перевод строки) в устройство может отличаться от команды

```
write !
```

поскольку для такого форматного вывода для каждого типа устройств и даже в зависимости от выбранного для них режима могут быть определены особенности поведения и побочных эффектов.

Кроме константных строк язык MUMPS также поддерживает задание констант в виде чисел. Это удобно для разработчиков и для чисел действуют правила:

1. Числа могут начинаться на символ цифры или десятичной точки.

2. Если перед константным числом стоит знак минус то это число считается отрицательным (используется как знак числа), если два - то первый считается унарной операцией минус, второй - знаком числа.

3. Если перед константным числом стоит знак плюс, то этот знак не входит в само число, число считается положительным по умолчанию, а знак плюс считается операцией унарный плюс.

4. Последовательность цифр после первого знака входит в число.

5. Число может быть задано дробным, разделителем целой и дробной частей считается десятичная точка.

6. Числу может быть указан показатель степени в экспоненциальной форме.

Экспоненциальная форма числа задается символов "E" (латинская), после которой следует показатель, или степень в которую необходимо взвести 10. Между символом показателя "E" и значением может не ставиться никакой символ (показатель считается положительным), ставиться знак плюс (показатель считается положительным) или ставиться знак минус (показатель считается отрицательным).

Многие современные MUMPS системы могут дополнительно придерживаться правила, что показатель можно указывать не только через символ "E" но и через символ "e". В практической работе такая возможность конечно удобна, если текст набирается преимущественно в нижнем регистре, но нужно понимать, что соглашение о малой букве "e" для экспоненциальной формы не входит в стандарт и может не поддерживаться в другой реализации. Если программа на MUMPS самостоятельно выполняет парсинг чисел, то также нужно убедиться что система и передаваемые ей данные соответствуют друг другу по соглашениям о "e".

Числа в именно числовой записи, без кавычек, могут быть использованы в любом месте, где допускается указание строки или значения. Исключением является команда read - ей нельзя указать в аргументе число для того чтобы команда вывела его на печать также как константную строку. Вместо этого число необходимо заключить в двойные кавычки.

Стандарт был определен таким образом и все реализации языка MUMPS его выполняют. Возможно, это было вызвано тем, то при переводе числа из внутреннего представления в строковое может получиться не то, что указал разработчик, поскольку приведение числа к строке система выполняет самостоятельно автоматически.

При необходимости использовать строку в арифметических операциях, или число в строковых, система MUMPS автоматически выполняет приведение значений.

Если необходимо значение, вычисленное как число, например заданное числовой константой или полученное арифметически, использовать как строку, то система применяет правила канонического представления числа. В числе отбрасываются лидирующие нули, завершающие нули, не используется знак для положительных чисел и добавляется знак минус для отрицательных, и, при необходимости, формируется экспоненциальная форма с символом "E" для отделения показателя от мантиссы. Все числа при преобразовании в строку преобразуются по десятичному основанию.

Если необходимо преобразовать строку в число, то система сканирует начальную часть строки и все символы которые подходят под определение числа используются как число, а остальные, начиная с первого неподходящего, отбрасываются. При этом если первый символ это символ плюс или минус, то они считаются знаком числа. Например:

```
USER>w +"1234"
1234
USER>w +"1234ABCD"
1234
USER>w +"1234  ABCD"
1234
USER>w +"12 с половиной "
12
```

Здесь оператор унарного плюс приводит аргумент к числу чтобы показать характер приведения. После чего команда write приводит число результат к строке для вывода в текущее устройство.

Контекстное автоматическое приведение типов приводит к тому, что разработчики на MUMPS спокойно используют любые значения по необ-

ходимости. Арифметические и строковые операции могут выполняться над значениями любых переменных и над результатами любых выражений.

Дополнительно к порядку приведения чисел действует правило получения булевского значения для различных условий и постусловий. Для получения булевского значения, равного или 0 или 1, система приводит аргумент сначала к числу, потом сравнивает его с нулем. Если получилось число равное 0, то результат 0, иначе (при любом другом) результат равен 1. В частности, операция логического отрицания приводит свой аргумент к булевской величине явно. Неявно к ней приводят операции условных операторов, постусловий, соответствия шаблону, вычисление значения бита.

Что интересно, двойное применение оператора логического отрицания к аргументу приводит арумент не к инвертированному, а к прямому булевскому виду. Первый оператор приводит к булевскому, но инвертированному, второй еще раз инвертирует для получения прямого:

```
USER>w ''12
1
USER>w ''"string"
0
USER>w ''"12 с половиной"
1
```

Такая операция двойного логического отрицания используется как операция булевского нормирования - в силу того, что оператор отрицания возвращает строго предопределенные значения, нормирование также получает строго определенные значения - 0 или 1, которые можно использовать и в арифметических и в строковых операциях.

Прямым эквивалентом операции приведения к булевской величине является операция явного приведения к числу и сравнение с нулем:

```
USER>w +12'=0
1
USER>w +"string"'=0
0
USER>w +"12 с половиной"'=0
1
```

В отличие от других языков, в языке MUMPS число не всегда является самостоятельной лексемой, рассматриваемой только как синтаксическое число. Интерпретатор не только использует последовательности

цифр по контексту операции, но и анализирует исходный код по контексту команды и операции. В частности, последовательность цифр является синтаксически корректным именем метки, например при выполнении команд

```
goto 123
do 123
```

лексема 123 означает не число как значение, а допустимое имя метки.

Что интересно, допускается также вариант очень похожий на сложение:

```
goto 123+4
```

В этом случае числом здесь является лексема 4, означающая смещение от метки с именем 123, а знак + синтаксически означает запись смещения относительно метки. Так, в коде

```
do 123+4+5
```

лексемы 4 и 5 вычисляются как числа и программа вызывает подпрограмму с метки 123 со смещением 9 (сумма 4 и 5).

В большинстве случаев, конечно, разработчики на MUMPS не стремятся злоупотреблять особенностями синтаксиса, чтобы не сделать код совершенно нечитабельным.

Другим случаем использования последовательностей цифр является задание числа повторов для шаблонов, например

```
USER>w "a12"?1.2A1N1E
1
```

Здесь спецификацией шаблона является полная последовательность символов "1.2A1N1E", причем последовательность "1.2" не означает число 1.2, а указывает число повторов (означает от 1 до 2 включительно).

1.5 Функции

Стандарт языка MUMPS предусматривает поддержку определенного набора системных функций. Системные функции - это встроенные в ядро интерпретатора операции обращения к данным, по преобразованию строк, или выполняющие служебные действия.

Синтаксически системные функции задаются именем, перед которым указывается один символ $. Также как имена команд, имена системных функций используются в любом регистре и могут сокращаться до аббревиатуры, за редким исключением это первая буква имени.

Если после имени функции не ставится пара скобок с аргументами, то синтаксически это соответствует системным переменным, например если пишется функция $PIECE в сокращенной форме ($P) без скобок, это воспринимается транслятором как системная переменная. В частности, системная переменная $P действительно существует, ее полное имя $PRINCIPAL.

В действительности многие системные переменные очень похожи на функции без аргументов. Есть системные переменные, которые не изменяются все время жизни для всех процессов, например $SYSTEM. Есть системные переменные, зависящие от процесса, но не изменяющие значения, например $JOB. При этом есть системные переменные, которые могут изменять свое значение регулярно, например $HOROLOG. Такие переменные уже большинством программистов воспринимаются как функции, вычисляющие значение независимо от аргумента. Но в языке MUMPS исторически принято различать переменные и функции, и если функция не требует аргументов, то это переменная.

Более того, в терминологии языка закрепилось и различие между пользовательскими функциями и пользовательскими переменными. Фактически же между ними нет разницы, пользовательские переменные это то же самое что и пользовательские функции, если их можно вызвать без аргументов и без скобок.

В круглых скобках после имени функции через запятую перечисляются аргументы функции. В отношении аргументов для стандартных системных функций в языке есть соглашение что если аргумент не указывается, то его позиция не выделяется. Например, нельзя написать опускание аргумента так:

```
$piece(str,,3)
```

Часть функций допускает опускание аргументов с конца, в частности вместо

```
$piece(str,"~",)
```

нужно писать

```
$piece(str,"~")
```

Некоторые реализации MUMPS систем поддерживают расширенные нестандартные системные функции, допускающие опускание аргумента, в частности, функция $LB() при опускании аргумента конструирует в списке в этой позиции специальный элемент списка с зарезервированным неопределенным значением.

Все системные функции определены так, что они возвращают некоторое значение, поэтому, даже если результат возврата не важен, его в любом случае необходимо использовать, по крайней мере присвоить временной переменной или использовать результат как аргумент команды if.

Весь набор системных функций можно разделить на несколько групп по назначению функций:

1. Функции доступа к данным
2. Функции имен
3. Строковые функции
4. Служебные функции

Функции доступа к данным возвращают информацию о наличии значения переменных и наличии их индексов. К таким относятся функции:

1. $DATA - вернуть индикатор существования переменной и ее индексов.
2. $GET - вернуть значение переменной или значение по умолчанию если переменная не существует.
3. $ORDER - вернуть следующее значение индекса на том же уровне.
4. $QUERY - вернуть следующее имя переменной независимо от уровня индексов.

1.5.1 $DATA

Функция $DATA ($D) принимает в аргументе имя переменной, локальной или глобальной, и возвращает одно из следующих значений:

1. 0 - ни эта переменная, ни ее дочерние не существуют
2. 1 - переменная существует, но дочерних не имеет

3. 10 - переменная не существует, но дочерние (по меньшей мере один дочерний элемент) есть
4. 11 - и эта переменная и ее дочерние (по меньшей мере один) существуют

Возвращаемое значение строго определено стандартом и может быть использовано в арифметических операциях, что разработчики и используют при необходимости. В частности, проверка на то что имя существует независимо от существования дочерних:

```
$DATA(varname)#2
```

и проверка существования дочерних независимо от существования самой переменной:

```
$DATA(varname)\10
```

Функция $DATA в некоторых реализациях MUMPS пожет быть применена также к структурным системным переменным. Например, в этом случае можно проверить существование процесса:

```
$DATA(^$JOB(jobnumber))
```

Но в этом случае необходимо сверить с документацией на используемую MUMPS систему, поддерживается ли такой функционал.

У функции $DATA имеется побочный эффект - при обращении к глобальной переменной независимо от существования или несуществования ее или ее дочерних переменных функция взводит индикатор голой ссылки в значение, равное имени переменной, к которой обратилась. Эта особенность также может быть использована разработчиками, если необходимо присвоить индикатору голой ссылки определенное имя, но используемая реализация MUMPS системы или ее версия не поддерживает прямое присваивание системной переменной $ZREFERENCE.

Существуют реализации MUMPS систем, поддерживающие нестандартную двухаргументную форму функции $DATA - в качестве второго аргумента необходимо указать имя переменной. В этом случае если переменная из первого аргумента имеет значение, то функция одновременно присвоит это значение второй переменной:

```
USER>s a=123

USER>w $d(a,value)
1
```

```
USER>w

a=123
value=123
```

Этот функционал является нестандартным и поведение индикатора голой ссылки в случае использования во втором аргументе глобальной переменной необходимо сверять с документацией на используемую версию MUMPS системы. Наиболее ожидаемым поведением в этом случае может быть присваивание второй переменной после получения индикатора существования и присваивание индикатору голой ссылки именно второго имени.

1.5.2 $GET

Функция $GET ($G) возвращает значение для указанной переменной. Функция имеет две формы вызова - одноаргументную и двухаргументную:

```
$GET(varname)
$GET(varname,defvalue)
```

Если переменная с заданным именем имеет значение, то оно возвращается независимо от формы функции и ее второго аргумента. Если эта переменная не имеет значения, то в первом случае для нее возвращается пустая строка, а во втором случае - значение указанное как значение по умолчанию, defvalue.

Также как и функция $DATA, функция $GET имеет побочный эффект - при обращении к глобальной переменной вне зависимости от ее существования функция $GET взводит индикатор голой ссылки в значение, равное имени глобальной переменной.

Традиционным применением функции $GET является код, не использующий предварительную инициализацию

```
f i=1:1:12 s total=$get(total)+i
```

не проверяющий существование переменной

```
f i=1:1:12 s total=$get(total)+$get(^gbl(i))
```

или обращающийся к параметрам функции которые могут получить неопределенное значение

```
func(param)
 s param=$get(param,0)
 ...
```

1.5.3 $ORDER

Функция $ORDER ($O) возвращает следующее значение индекса в порядке их существования у переменной или пустую строку если следующего индекса не существует.

Для функции необходимо передать локальное или глобальное имя, указав нужное число индексов. Функция будет возвращать следующий индекс (если он существует) на уровне последнего указанного индекса.

```
USER>s a(1)=1

USER>s a(2)=1

USER>s a(3)=1

USER>s a(4)=1

USER>s a(2,1)=1

USER>s a(2,2)=1

USER>s a(3,1)=1

USER>s a(4,1)=1

USER>s n=""

USER>f  s n=$o(a(n)) q:n=""  w n,!
1
2
3
4
```

Функция возвращает значение индекса в обоих вариантах - если существует такая переменная или если существуют ее дочерние переменные. Нужно понимать, что если функция $ORDER вернула следующий индекс, то этот индекс означает следующий в понимании переменной как дерева, но само значение не обязано существовать.

У функции $ORDER существует две формы - одноаргументная и двухаргументная. Одноаргументная форма возвращает следующий индекс в порядке возрастания. Двухаргументная форма принимает вторым аргументом индикатор направления, который может принимать только одно из следующих значений:

1. 1 - перечислить по возрастанию

2. -1 - перечислить по убыванию

Любые другие значения в настоящее время зарезервированы для будущего использования и их использование должно генерировать ошибку.

Существуют реализации MUMPS систем, которые поддерживают применение функции $ORDER также и к структурной системной переменной. Этот функционал относится к нестандартному, поэтому такую возможность нужно проверить в документации. Если поддерживается, то, например, можно перечислить имеющиеся в MUMPS системе процессы

```
s job="" f  s job=$o(^$J(job)) q:job=""  w job,!
```

или открытые устройства

```
s dev="" f  s dev=$o(^$D(dev)) q:dev=""  w dev,!
```

Существуют реализации MUMPS систем, поддерживающие нестандартную трехаргументную форму функции $ORDER. В этом случае в качестве третьего аргумента указывается переменная, которой необходимо присвоить значение переменной если оно существовало. Если значения с точно таким именем не существовало, то возвращается только значение индекса, но присваивание не производится. Например, такой функционал поддерживается Caché:

```
USER>s a(1)=1,a(2,1)=21,a(3)=3,a(4)=4

USER>s n="" f  s n=$o(a(n),1,v) q:n=""  w v,!
1
1
3
4

USER>s n="" f  s n=$o(a(n),1,v) q:n=""  w n,!
1
2
3
4
```

Здесь видно, что для индекса a(2) его значение возвращается, но присваивание не выполняется.

В отношении функции $ORDER побочный эффект взведения индикатора голой ссылки в случае обращения к глобалу определен более сложно - если следующее значение индекса существует, то индикатор голой ссылки взводится в это значение. Если не существует, то в то

имя, к которому обратились первоначально. Таким образом, побочный результат зависит от того, существуют данные или нет, в отличие от функций $DATA и $GET.

Первоначально в стандарте языка MUMPS функции $ORDER не было, она появилась только в 1984-м году, и в этой редакции была принята одноаргументная форма. Двухаргументная форма, с возможностью прохода по убыванию, появилась лишь в стандарте языка 1995-го года. В настоящее время все современные реализации MUMPS систем поддерживают эту функцию в полной мере.

В техническом отношении реализация функции $ORDER, в особенности реализация прохода назад, или в направлении обратном компрессии ключей, относится к наиболее сложным для выполнения на блоках B* - деревьев. Существуют также СУБД, в которых для аналогичных к функции $ORDER операций в документации делаются отдельные оговорки о необходимости применения опций asc или desc для индексов во избежание существенного падения производительности, включая рекомендацию строить оба варианта индексов для случая если выборка с сортировкой по индексу критична для прикладной системы.

Современные MUMPS системы выполняют функцию $ORDER как в прямом направлении сортировки, так и в обратном, с примерно сопоставимой производительностью. Насколько показывает опыт применения $ORDER в различных MUMPS системах, если падение производительности и можно определить, то оно ни разу не оказалось заметным либо существенным.

1.5.4 $NEXT

До введения в стандарт языка функции $ORDER в языке использовалась функция $NEXT, в одноаргументной форме. В стандарте 1990-го года эта функция была объявлена нерекомендованной к применению, и в стандарте 1995-го года отсутствует. В настоящее время существуют реализации MUMPS систем, которые не поддерживают функцию $NEXT и которые поддерживают для совместимости.

Функция $NEXT может вернуть следующее значение индекса только по возрастанию и в случае, если следующее значение индекса не существует, то функция возвращает значение -1.

Для числовых индексов такое поведение неудобно тем, что для случая если функция вернула значение -1 необходимо различать, является ли это значение следующим существующим или это индикатор несуществования, с помощью функции $DATA.

Последующее определение функции $ORDER уже учитывало тот факт, что в MUMPS системах по умолчанию значения индексов равные пустой строке (пустые индексы) не разрешены. В случае же, если в настройках MUMPS системы пустые индексы разрешены, то такую операцию по дополнительной проверке также необходимо выполнять. Большинство прикладных систем, как существующих так и разрабатываемых, опираются на соглашение о настройках по умолчанию и о неиспользовании пустых индексов.

1.5.5 $QUERY

Функция $QUERY ($Q) принимает имя переменной и возвращает следующее существующее для этой переменной имя в порядке сортировки.

Особенность функции в том, что ей можно указать в аргументе имя, но нельзя принять само имя как значение. Поэтому функция возвращает строку с именем переменной. В случае если следующего имени функция не нашла, она возвращает пустую строку.

Такое соглашение о возврате значения приводит к использованию косвенности имени при итерациях. Если начальное значение имени может быть указано явно как имя переменной, то последующая итерация уже должна использовать строку с именем. Чтобы перевести строку в само имя, используется косвенность. Примерный типовой код применения функции $QUERY:

```
USER>s a(1)=1,a(2,2)=22,a(3,3,3)=333,a(4)=4

USER>s name=$na(a)

USER>f  s name=$q(@name) q:name=""  w name,!
a(1)
a(2,2)
a(3,3,3)
a(4)
```

В отличие от функции $ORDER функция $QUERY не рассматривает переменную как дерево, а использует как набор сортированных записей вида ключ - значение, но ключ может быть произвольным и составным.

Если функция $QUERY используется в одноаргументной форме, то она перечисляет переменную в направлении возрастания индексов в порядке их сортировки.

Функция $QUERY поддерживает так же, как и функция $ORDER, двухаргументную форму. Второй аргумент задает направление перечисления имен - если аргумент вычислен как значение 1, то функция пе-

речисляет по возрастанию, а если равно -1, то по убыванию индексов в порядке их сортировки.

```
USER>s a(1)=1,a(2,2)=22,a(3,3,3)=333,a(4)=4

USER>s name=$na(a(1000000))

USER>f  s name=$q(@name,-1) q:name=""  w name,!
a(4)
a(3,3,3)
a(2,2)
a(1)
```

Здесь в примере использовалось то, что в данных применялись числовые индексы. В реальности, при использовании строковых значений, необходимо в качестве начального значения итератора применить некое значение заведомо большее чем любое существующее, например

```
USER>s name=$na(a($c(255,255,255,255,255,255)))
```

Либо использовать начальное значение в виде имени с одним индексом пустая строка:

```
USER>s a(1)=1,a(2,2)=22,a(3,3,3)=333,a(4)=4

USER>s name=$na(a(""))

USER>f  s name=$q(@name,-1) q:name=""  w name,!
a(4)
a(3,3,3)
a(2,2)
a(1)
```

Обе функции, и $ORDER, и $QUERY, воспринимают пустую строку в качестве индекса при перечислении назад как индикатор необходимости найти самое старшее значение и вернуть его.

Функция $QUERY традиционно используется совместно с функциями $NAME, $QLENGTH и $QSUBSCRIPT и с применением операции косвенности имени. В случае если необходимо остановить итерацию ранее чем закончится переменная, или не использовать переменные с ненужным числом аргументов, применяется определение значений отдельных частей имени переменной, в частности:

```
USER>s a(1)=1,a(2,2)=22,a(3,3,3)=333,a(4)=4

USER>s name=$na(a)
```

```
USER>f  s name=$q(@name) q:name=""  w name," : ",$ql(name),!
a(1) : 1
a(2,2) : 2
a(3,3,3) : 3
a(4) : 1
```

Функция $QUERY, также как и функция $ORDER, может применяться как к локальным, так и к глобальным переменным. Причем функция $QUERY поддерживает в отношении глобальных переменных соглашение об имени базы данных:

1. Если аргумент не использует явное задание имени базы данных, то результат возвращается также без имени базы данных.

2. Если аргумент явно содержит имя базы данных, то возвращаемый результат также содержит имя базы данных, даже если эта база данных является текущей.

К дополнительным расширениям, которые могут поддерживаться отдельными реализациями MUMPS систем, относится возможность использовать функцию $QUERY для структурных системных переменных. В настоящее время этот функционал относится к нестандартному. Существуют MUMPS системы, которые не позволяют использовать $QUERY в отношении структурных системных переменных, есть которые позволяют в отношении некоторых из них. В любом случае разработчик, применяя функцию $QUERY к структурной системной переменной, должен проверить в документации на используемую MUMPS систему, как она поддерживает такую операцию.

Если используемая MUMPS система поддерживает применение функции $QUERY к структурной системной переменной, то такими переменными можно оперировать как обычными:

```
USER>s name=$na(^$J(""))

USER>f  s name=$q(@name) q:name=""  w name," : ",$ql(name),!
^$JOB("5960") : 1
^$JOB("7076") : 1
```

Функция $QUERY появилась в языке в стандарте 1990-го года, и, формально говоря, в стандарт входит лишь одноаргументная форма этой функции. Двухаргументная форма функции $QUERY входит в рекомендации комитета по стандартизации языка (MDC), но, в действительности, поддерживается всеми современными реализациями MUMPS систем. Функции $NAME, $QLENGTH и $QSUBSCRIPT входят в стандарт с 1995-го года.

Функции имен оперируют именами переменных и выполняют операцию обратную к операции косвенности имени. Частично их применение было показано при использовании функции $QUERY. К таким относятся функции:

1. $NAME - возвращает строку с именем переменной для указанной переменной.
2. $QLENGTH - возвращает число индексов у переменной заданной именем.
3. $QSUBSCRIPT - возвращает часть имени переменной заданной именем.

Функции имен оперируют так называемым каноническим представлением имен переменных. При преобразовании имя - строка или наоборот, строка - имя MUMPS система пользуется едиными соглашениями о представлении имен и значений индексов. Если в системе принято кодировать числовой индекс как число а не как строку, то при канонизации имени такое значение будет представляться именно как число, например:

```
USER>w $na(a(123))
a(123)
USER>w $na(a("123"))
a(123)
```

1.5.6 $NAME

Функция $NAME ($NA) конструирует строковое значение имени для заданной переменной. Что интересно, имя функции сокращается не до $N а до $NA, поскольку изначально до буквы $N сокращалась функция $NEXT. Хотя функция $NEXT в настоящее время не применяется и рекомендована к замене на функцию $ORDER, в целях совместимости сокращение имени $NAME до имени $NA осталось.

Функция $NAME принимает в качестве аргумента собственно имя переменной, локальной или глобальной, и возвращает строку с этим именем, представленное в канонической форме.

Что интересно, для функции $NAME не требуется существование самой этой переменной, она оперирует только ее именем. Причем само имя может быть использовано как явно, так и косвенно, например:

```
USER>w $na(abc)
abc
```

```
USER>w $na(abc(123,456))
abc(123,456)
USER>w $na(abc("ss","day"))
abc("ss","day")
USER>s name=$na(abc(1,2))

USER>w $na(@name)
abc(1,2)
USER>w $na(@name@(3))
abc(1,2,3)
```

В случае использования глобального имени функция $NAME, также как и функция $QUERY, возвращает результат с именем базы данных если имя базы было указано в аргументе и наоборот, если имя базы данных не было указано, то результат также его не содержит.

```
USER>w $na(^GBL(3,3,3))
^GBL(3,3,3)
USER>w $na(^|"any"|GBL(3,3,3))
^|"any"|GBL(3,3,3)
```

Точно также, как и сама переменная, база данных не обязана существовать, функция $NAME оперирует лишь именем переменной, не обращаясь к реальным данным.

У функции $NAME определены две формы - одноаргументная и двухаргументная. Для двухаргументной формы второй аргумент вычисляется как целое число и функция возвращает только часть имени из первого аргумента так, что число индексов в возвращаемом результате ограничено значением второго аргумента. В случае если в исходном имени недостаточно индексов, то возвращается столько, сколько их имеется:

```
USER>w $na(a(1,2,3,4,5),2)
a(1,2)
USER>w $na(a(1,2,3,4,5),8)
a(1,2,3,4,5)
```

Двухаргументная форма $NAME, судя по статистике применения, используется чрезвычайно редко. Но в тех задачах, когда надо получить только часть имени переменной, она подходит наилучшим образом.

Традиционным применением функции $NAME является ее парное применение к операциям косвенности имени, когда программный код оперирует обобщенными именами, при использовании функции $QUERY, и при конструировании более длинного имени путем добавления индексов или более короткого путем отсечения части имени.

Отдельные реализации MUMPS систем могут использовать имена структурных системных переменных в качестве аргумента функции $NAME. Этот функционал относится к нестандартному и при необходимости его использования следует обратиться к документации на применяемую MUMPS систему. В частности, если функция $NAME поддерживает такой функционал, то может конструировать имена:

```
USER>w $na(^$J(12,34,56))
^$J(12,34,56)
```

В этом случае необходимо понимать, что к реальной структурной системной переменной функция $NAME не обращается и имя, сконструированное в результате, может не соответствовать реально существующему значению либо каким-либо соглашениям и правилам для значений в этой структурной системной переменной.

При первоначальном знакомстве с системами программирования основанными на языке MUMPS новички могут путаться, где используется само имя, а где используется строка с именем переменной. Хорошим вариантом решения проблемы может быть обучение или дополнительное комментирование кода.

1.5.7 $QLENGTH

Функция $QLENGTH ($QL) возвращает число индексов у переменной заданной строкой с именем.

```
USER>s name=$na(a(1,2,3,4))

USER>w $ql(name)
4
USER>s name=$na(a(1,2,3,4,5,6))

USER>w $ql(name)
6
USER>s name=$na(a)

USER>w $ql(name)
0
```

Функция $QLENGTH не включает в результат само имя переменной и имя базы данных, если оно указывается.

Функция может быть применена к именам локальных, глобальных и в зависимости от версии MUMPS системы, к именам структурных системных переменных.

1.5.8 $QSUBSCRIPT

Функция $QSUBSCRIPT ($QS) возвращает часть имени. Первый аргумент функции это строка с именем переменной, второй аргумент - номер индекса. Если значение второго аргумента равно 1, то возвращается первый индекс, если 2 - второй, и так далее. Для несуществующих в имени индексов возвращается пустая строка.

```
USER>s name=$na(^|"any"|GBL(1,2,3,4))

USER>w $qs(name,1)
1
USER>w $qs(name,2)
2
USER>w $qs(name,3)
3
USER>w $qs(name,4)
4
USER>w $qs(name,5)

USER>
```

Функция $QSUBSCRIPT поддерживает дополнительно два специально зарезервированных значения для второго аргумента - если оно равно 0, то возвращается строка с корневым именем, а если -1 - то строка с именем базы данных, если оно было указано:

```
USER>s name=$na(^|"any"|GBL(1,2,3,4))

USER>w $qs(name,0)
^GBL
USER>w $qs(name,-1)
any
```

Все остальные отрицательные значения зарезервированы и при их использовании в настоящее время должна генерироваться ошибка.

Функции $NAME, $QLENGTH и $QSUBSCRIPT в описанном виде входят в текущий стандарт языка MUMPS и могут быть свободно использованы в целях переносимости программ.

В рекомендации комитета MDC дополнительно входит поддержка левостороннего присваивания для функции $QSUBSCRIPT, но такой функционал поддерживается не всеми реализациями. В частности, MiniM Database Server поддерживает такой функционал:

```
USER>s name=$na(a("aa","bb","cc"))
```

```
USER>w name
a("aa","bb","cc")
USER>s $qs(name,2)="dd"

USER>w name
a("aa","dd","cc")
USER>s $qs(name,0)="newname"

USER>w name
newname("aa","dd","cc")
```

Функция $QSUBSCRIPT с левым присваиванием использует в качестве первого аргумента переменную, в которой должно быть имя переменной. При присваивании это значение изменяется таким образом, что в содержащемся имени замещается соответствующий фрагмент. В случае если в исходном имени недостаточно индексов, функция дополняет имя индексами с пустыми строками:

```
USER>s $qs(name,6)="ff"

USER>w name
a("aa","dd","cc","","","ff")
```

Аналогичный к левосторонней функции $QSUBSCRIPT функционал можно получить, комбинируя косвенность имени и функцию $NAME, и выполняя в цикле итерации для каждого из значений индексов.

Строковые функции манипулируют строковыми значениями безотносительно того, хранятся они в базе данных или нет. Функции рассматривают строки как последовательность байт. В строке, согласно стандарту языка MUMPS, могут содержаться произвольные символы, включая нулевой байт.

К строковым функциям относятся функции:

1. $ASCII - возвращает код символа в указанной позиции.
2. $CHAR - конструирует строку на основе заданных кодов символов.
3. $EXTRACT - возвращает подстроку.
4. $PIECE - возвращает фрагмент строки рассматривая ее как строку с разделителями.
5. $LENGTH - возвращает число символов в строке.
6. $REVERSE - возвращает строку с обратным порядком символов.

7. $FIND - возвращает позицию подстроки в строке.
8. $TRANSLATE - выполняет замену символов.
9. $JUSTIFY - выполняет дополнение строки.
10. $FNUMBER - форматирует число по заданному формату.

Некоторые из аргументов строковые функции воспринимают как строки, некоторые строго как числа. В отличие от других языков, в MUMPS любое значение может быть использовано в обоих контекстах. В частности, строковые функции могут вычислить длину числа

```
USER>w $length(456)
3
```

или переставить в числе знаки

```
USER>w $reverse(456)
654
```

При этом любой из параметров строковыми функциями используется как выражение, которое предварительно вычисляется перед использованием. Использование в примерах констант - это частный случай использования выражений. Например

```
USER>w $length(4_5_6)
3
USER>w $length(4+5+6)
2
```

1.5.9 $ASCII

Функция $ASCII ($A) возвращает код символа из строки в указанной позиции. Строка рассматривается как последовательность байт, и один байт рассматривается в зависимости от реализации как 7-ми битный или 8-ми битный. Современные реализации MUMPS систем оперируют 8-ми битными байтами и рассматривают коды символов старше 127 как беззнаковое число от 0 до 255 включительно.

Это отношение к строке по умолчанию, поддерживаемое большинством систем. При этом в связи с расширением поддержки кодирований UTF и юникода отдельные MUMPS системы могут возвращать код не до 255 включительно, а все старшие и позиция в строке отсчитывается не в

байтах во внутреннем представлении, а в виде позиции символа в зависимости от использованного кодирования (UTF отводит под различные символы различное число байт для их представления). В любом случае, если система применяет юникод, его поддержка внешне, для разработчиков, приводится к таким же стандартным соглашениям.

Кроме того, современные реализации MUMPS придерживаются соглашений об ASCII кодировании символов, когда часть кодовой таблицы от 0 до 127 отводится под стандартную (латиница), а дополнительная часть (от 128 до 255) отводится под национальные символы. В стандарте для них используется также отдельное название - графические символы. Для них уже нет четко определенного представления и для представления символов каждая из реализаций использует дополнительное соглашение о кодировании и кодировках символов.

Для функции $ASCII определены две формы - одноаргументная и двухаргументная. Основной формой является двухаргументная. При этом первый аргумент - строка в которой надо определить код символа, второй - позиция символа в строке. Отсчет символов ведется от единицы. Например:

```
USER>w $a("123",1)
49
USER>w $a("123",2)
50
USER>w $a("123",3)
51
```

В случае если для указанной позиции в строке нет символа, функция $ASCII возвращает значение -1. Это значение определено стандартом и может быть использовано в арифметических операциях.

Если второй параметр функции опущен, то функция возвращает код первого символа, или, другими словами, использует значение параметра по умолчанию равное 1:

```
USER>w $a("123",1)
49
USER>w $a("123")
49
```

Функция $ASCII не использует никаких соглашений о строении строки, ее формате, кодировке, и рассматривает строку исключительно как просто последовательность байт, вне зависимости от того, каким образом строка была сформирована. Для систем поддерживающих юникод, строка рассматривается как последовательность символов.

Большинство современных реализация MUMPS систем поддерживают расширенные варианты функции $ASCII - в зависимости от версии это обычно $ZWASCII ($ZWA), $ZLASCII ($ZLA) и $ZQASCII ($ZQA). Они оперируют не одним байтом, а соответственно словом (2 байта), двойным словом (4 байта) и четверным словом (8 байт). Вся последовательность байт для них разбивается на последовательность слов соответствующей длины. Так, если есть строка "abcdefghijk", то такие функции дают результат:

```
USER>s str="abcdefghijk"

USER>w $a(str,1)
97
USER>w $a(str,2)
98
USER>w $zwa(str,1)
25185
USER>w $zwa(str,2)
25442
USER>w $zla(str,1)
1684234849
USER>w $zla(str,2)
1701077858
USER>w $zqa(str,1)
7523094288207667809
USER>w $zqa(str,2)
7595434461045744482
```

В случае если в строке недостаточно байт для указанной группы, то функции возвращают значение -1:

```
USER>w $zwa("1",1)
-1
USER>w $zqa(str,8)
-1
```

Расширенные варианты функций $ZWA, $ZLA и $ZQA очень практичны в задачах кодирования последовательностей чисел в строку и обратно для различных протоколов передачи данных, но нужно понимать, что они не входят в стандарт и могут не поддерживаться на выбранной реализации MUMPS системы, либо могут иметь иной результат для неопределенного значения (возвращать вместо -1 иное значение). Также нужно отметить, что позиция в строке для отсчета слова указывается не в словах, а в байтах.

При необходимости можно заменить эти функции при портировании программ на комбинацию стандартной функции $ASCII и арифметических операций:

```
zwa(str,pos)
 s pos=$g(pos,1)
 q $a(str,pos+1)*256+$a(str,pos)
zla(s,pos)
 s pos=$g(pos,1)
 q $a(s,pos+3)*256+$a(s,pos+2)*256+$a(s,pos+1)*256+$a(s,pos)
```

Здесь в последней строке используется соглашение об отсутствии приоритетов арифметических (и других тоже) операций в языке MUMPS, при расстановке скобок для соответствия арифметическим опрациям в языках с приоритетами код выглядит так:

```
zwa(str,pos)
 s pos=$g(pos,1)
 q $a(str,pos+1)*256+$a(str,pos)
zla(s,pos)
 s pos=$g(pos,1)
 q (($a(s,pos+3)*256+$a(s,pos+2))*256+$a(s,pos+1))*256+$a(s,pos)
```

Это только примерная арифметическая замена, в реальном коде необходимо также дополнительно проверить, присутствует ли в строке начиная с указанной позиции достаточное количество байт и не является ли позиция отрицательным числом.

В определенном смысле расширенные функции $ZWA, $ZLA и $ZQA используют соглашение о кодировании целых чисел в порядке от младшего к старшему, или little - endian, он же порядок кодирования байт в процессорах фирмы Intel и совместимых (x86 архитектуры).

1.5.10 $CHAR

Функция $CHAR ($C) конструирует строку из символов с указанными кодами. В качестве параметров принимает значения с кодами символов и объединяет полученные символы в одну строку. Это одна из немногих функций MUMPS определенная как функция с переменным числом аргументов.

Функция может принять минимум один аргумент. Меньшее число соответствует просто пустой строке. Максимальное число параметров определяется реализацией MUMPS системы. В частности, MiniM может принять до 255 параметров, а GT.M до 264 параметра включительно. Например:

```
USER>w $char(56)
8
USER>w $char(56,59,73)
```

```
8;I
USER>w $char(56,59,73,89)
8;IY
```

Для символов существуют естественные ограничения их кодов - от 0 до 255. Для отрицательных значений кодов поведение функции $CHAR определено как неиспользование соответствующего аргумента. Если один или более аргументов вычислены как отрицательное число, то функция пропускает этот аргумент:

```
USER>w $c(56,78)
8N
USER>w $c(56,-1,78)
8N
USER>w $c(56,78,88,89)
8NXY
USER>w $c(56,-1,78,-1,88,-1,89)
8NXY
```

В частности, если функции указаны один или более только отрицательных аргументов, то функция возвращает пустую строку:

```
USER>w 8_$c(-1)_8
88
```

Для значений аргументов больше чем 255 стандарт не дает точного определения результата. Большинство MUMPS систем в таком случае также пропускает такой аргумент. И, если MUMPS система поддерживает юникод, то наоборот, использует коды в качестве кодов символов для конструирования строки.

Возможность функции $CHAR использовать несколько параметров зачастую используется для конструирования специальных строк вместо операции конкатенации. В частности, последовательность байт, означающих перевод строки, может быть записана двумя способами:

```
$c(13)_$c(10)
$c(13,10)
```

Для функции $CHAR, как и для функции $ASCII многие реализации также поддерживают ее расширенные варианты $ZWCHAR ($ZWC), $ZLCHAR ($ZLC) и $ZQCHAR ($ZQC). Эти расширения работают соответственно с кодами в виде слова (2 байт), двойного слова (4 байт) и четверного слова (8 байт) и являются парными к соответствующим $Z расширениям для функции $ASCII. Например:

```
USER>w $zwa("AB")
16961
USER>w $zwc(16961)
AB
```

Как и $Z расширения для $ASCII, такие функции не входят в стандарт и при необходимости портировать код на систему которая их не поддерживает, необходимо проводить эквивалентную замену на функцию использующую стандартную функцию $CHAR и арифметические операции, в частности, для функции $ZWC:

```
zwc(code)
 q $c(code#256,code\256)
```

Как и в случае с заменой расширенных функций $ASCII, здесь для совместимости поведения также необходимо дополнительно проверять аргумент на отрицательную величину и на выход за пределы максимально допустимого кода.

1.5.11 $EXTRACT

Функция $EXTRACT ($E) возвращает подстроку из заданной строки. Исходная строка рассматривается как последовательность байт, и позиции в строке отсчитываются от единицы. Функция $EXTRACT имеет 3 формы - 3-х аргументную, 2-х аргументную и 1- аргументную.

Полная 3-х аргументная форма задает кроме исходной строки также позицию с которой необходимо вернуть подстроку и позицию по которую необходимо вернуть подстроку:

```
USER>s str="ABCDEFGH"

USER>w $e(str,3,6)
CDEF
USER>w $e(str,3,7)
CDEFG
```

В случае если вторая позиция (окончание подстроки) выходит за пределы строки, функция возвращает только имеющиеся в строке байты, попадающие в заданный диапазон. В случае если вторая позиция (окончание подстроки) меньше первой позиции, функция возвращает пустую строку.

Для 2-х аргументной формы, при опускании позиции окончания подстроки, функция по умолчанию возвращает подстроку лишь из одного символа в указанной позиции. Если в исходной строке такой позиции нет, то функция возвращает пустую строку.

```
USER>s str="ABCDEFGH"

USER>w $e(str,3)
C
USER>w $e(str,5)
E
USER>w $e(str,-30)

USER>w $e(str,30)

USER>
```

Одноаргументная форма функции $EXTRACT трактует поведение еще более упрощено, по умолчанию недостающие параметры трактуются как взятие только первого символа. В случае если исходная строка пустая, то возвращается пустая строка.

```
USER>w $e("")

USER>w $e("ABCD")
A
USER>w $e(789)
7
```

Функция $EXTRACT также может быть использована для левостороннего присваивания, в этом случае она специфицирует команде SET позиции для замещения.

1.5.12 $PIECE

Функция $PIECE ($P) рассматривает строку как строку с разделителями. Строка рассматривается как последовательность фрагментов, перемежающихся разделителями, в форме:

```
piece1 delim piece2 delim piece3 ... delim pieceN
```

Каждый из фрагментов, включая первый (перед первым разделителем) и последний (после последнего разделителя) может быть и пустой строкой. В частности, строка не содержащая разделитель, считается состоящей из одного фрагмента, а строка, полностью совпадающая с разделителем, считается состоящей из двух фрагментов, равных пустой строке каждый.

В отличие от многих распространенных библиотек, в языке MUMPS разделителем может являться последовательность символов из любого их числа, не только из одного символа.

Функция $PIECE имеет 3 формы - 4-х аргументную, 3-х аргументную и 2-х аргументную. Поведение этих трех форм в точности совпадает с трактовкой аргументов для функции $EXTRACT, за исключением того, что для $EXTRACT фрагментами строки являются каждый символ, а для $PIECE фрагменты разделяемые разделителем.

Для 4-х аргументной формы указывается номер фрагмента, с которого необходимо вернуть подстроку и номер фрагмента по который необходимо вернуть подстроку и все попавшие фрагменты возвращаются с тем же разделителем:

```
USER>s str="ab~cde~fgh~ijklm~"

USER>w $p(str,"~",2,3)
cde~fgh
USER>w $p(str,"~",2,4)
cde~fgh~ijklm
USER>w $p(str,"~",2,5)
cde~fgh~ijklm~
USER>w $p(str,"~",2,6)
cde~fgh~ijklm~
```

Как и для функции $EXTRACT, если в исходной строке недостаточно фрагментов, возвращается только та часть, которая присутствует.

```
USER>s str="ab~cde~fgh~ijklm~"

USER>w $p(str,"~",2,-3)

USER>
```

Если позиция последнего фрагмента (его номер, отсчитываемый от единицы) меньше первого заданного, то возвращается пустая строка. Если номера первой и последней позиции совпадают, то возвращается только этот один фрагмент.

```
USER>s str="ab~cde~fgh~ijklm~"

USER>w $p(str,"~",3,3)
fgh
```

Для трехаргументной формы функция $PIECE применяет умолчание о том, что необходимо вернуть только один фрагмент:

```
USER>s str="ab~cde~fgh~ijklm~"

USER>w $p(str,"~",2)
cde
USER>w $p(str,"~",4)
ijklm
```

И для 2-х аргументной формы применяется умолчание о том, что необходимо вернуть только один первый фрагмент.

```
USER>s str="ab~cde~fgh~ijklm~"

USER>w $p(str,"~")
ab
USER>w $p(str,"b")
a
USER>w $p(str,"c")
ab~
```

Для языка MUMPS нет отдельной типизации строки, позволяющей указать, что она организована с определенным разделителем. Какой разделитель использовать указывается для каждого вызова функции $PIECE а не для строки. Одна и та же строка различными вызовами функции $PIECE может рассматриваться как строка с различными разделителями.

Что интересно, в языке MUMPS нет эквивалентной замены функции $EXTRACT на функцию $PIECE с разделителем, равным пустой строке. В случае, если разделитель пустая строка, функция $PIECE всегда возвращает пустую строку, независимо от значения других аргументов. Такое поведение жестко определено стандартом и поддерживается всеми реализациями MUMPS.

1.5.13 $LENGTH

Функция $LENGTH ($L) имеет две формы - одноаргументную и двухаргументную. Одноаргументная форма возвращает число символов в строке. Для пустой строки возвращается число 0:

```
USER>w $l(789)
3
USER>w $l("")
0
USER>w $l("abcdef")
6
```

В двухаргументной форме функция $LENGTH рассматривает строку как строку с разделителями и второй аргумент задает значение этого разделителя. В этом случае функция возвращает число фрагментов. Для строки число фрагментов равно числу найденных в ней разделителей плюс 1. В частности, если после последнего разделителя нет символов, в этой строке последний фрагмент считается пустой строкой.

```
USER>s str="ab~cde~fgh~ijklm~"

USER>w $l(str,"~")
5
USER>w $l(str,"g")
2
```

Одним из наиболее частых применений на практике двухаргументной формы $LENGTH является подсчет числа фрагментов для последующего их использования функцией $PIECE, например:

```
USER>s str="ab~cde~fgh~ijklm~"

USER>f i=1:1:$l(str,"~") w i,":",$p(str,"~",i),!
1:ab
2:cde
3:fgh
4:ijklm
5:

USER>
```

Другим практическим применением является подсчет числа вхождений в строку заданной подстроки. Например, вычисление сколько раз входит в строку символ "8":

```
USER>s str=78457898

USER>w $l(str,8)-1
3
```

Отдельно в стандарте языка указано, что если разделитель задан пустой строкой, то функция $LENGTH возвращает значение 0 в любом случае, независимо от значения исходной строки.

```
USER>w $l(123,"")
0
USER>w $l("","")
0
USER>w $l("abcdef","")
0
```

Таким образом, дилемму "сколько в пустой строке содержится пустых строк" стандарт языка MUMPS решает в сторону значения 0, считая пустую строку особым случаем и что в пустой строке не содержится ничего, даже пустой строки. Что интересно, при любом другом значении

разделителя функция возвращает в любом случае не нулевое значение, даже если исходная строка равна пустой строке - в этом случае она считается состоящей из одного фрагмента, равного пустой строке, но этот фрагмент для непустого разделителя в любом случае существует.

И функция $PIECE и функция $LENGTH, судя по статистике их использования, применяют в качестве разделителя лишь один символ. В действительности это лишь специфика кода, а не ограничение на длину разделителей, и функции могут использовать в качестве разделителей произвольные последовательности символов.

1.5.14 $REVERSE

Функция $REVERSE ($RE) возвращает исходную строку с символами в обратном порядке.

```
USER>w $re(789)
987
USER>w $re("abcdef")
fedcba
```

В практике функция перестановки символов в обратном порядке применяется редко, но автору довелось видеть ее эффектное применение в задаче индексации в обратном порядке для поиска по окончаниям слов. В решении поиска по окончаниям слов индексируемые значения переворачиваются и по ним уже строится индекс. При поиске искомое окончание также переворачивается и выполняется обычный поиск по началам слов. В какой-то мере такое решение в паре со звуковым хешированием, аналогичным SOUNDEX, может составить мечту поэтов, затрудняющихся с рифмой.

1.5.15 $FIND

Функция $FIND ($F) ищет вхождение в строку заданной подстроки. Функция возвращает номер символа следующий непосредственно за найденной подстрокой. В случае если искомая подстрока является последней, то возвращаемый номер уже не принадлежит исходной строке. Если функция не нашла подстроку по заданным ей условиям, то возвращает значение 0.

Функция $FIND имеет две формы - 3-х аргументную и 2-х аргументную. Для двухаргументной формы функция выполняет поиск с начала строки.

```
USER>s str="abcdfffghabc"

USER>w $f(str,"a")
2
USER>w $f(str,"g")
9
```

Для трехаргументной формы фукция выполняет поиск начиная с заданной третьим аргументом позиции. Практичность определения возвращаемого значения не как позиции найденной подстроки, а как позиции, следующей после подстроки, состоит в практичности организации цикла с итератором по позиии:

```
USER>s str="abcdfffghabc"

USER>s pos=0,find="a"

USER>f  s pos=$f(str,find,pos) q:'pos  w pos,!
2
11
```

Разработчикам необходимо знать об этой особенности возвращаемого значения при портировании на язык MUMPS кода, написанного на другом языке, и использующего другие соглашения о поиске подстроки, либо использующие библиотеки с другими соглашениями. А также при портировании MUMPS кода на другие языки.

1.5.16 $TRANSLATE

Функция $TRANSLATE ($TR) выполняет посимвольную замену и возвращает строку с выполненной заменой. Замена выполняется по схеме символ из второй строки - на символ из третьей строки находящийся в совпадающей позиции.

Функция используется либо в 3-х аргументной форме, либо в 2-х аргументной форме. 2-х аргументная форма полностью эквивалентна 3-харгументной форме со значением пустая строка в качестве третьего аргумента.

Первым аргументом функция принимает исходную строку, вторым аргументом - строку таблицу символов которые необходимо заменить. Третьим аргументом функция принимает строку таблицу символов на которые следует заменить. Найдя в исходной строке один из символов принадлежащих строке сторого аргумента, функция выполняет его замену на символ из третьего аргументв в соответствующей позиции.

Например, чтобы выполнить замену всех символов по таблице

```
A -> a
B -> b
C -> c
```

нужно использовать вторым аргументом строку "ABC" а третьим строку "abc". При этом не важно в каком порядке будут в них идти символы, важно лишь чтобы символы замены находились в тех же позициях. В частности:

```
USER>s str="ABBA Call All"

USER>w $tr(str,"ABC","abc")
abba call all
USER>w $tr(str,"BAC","bac")
abba call all
```

К особенностям поведения фунции $TRANSLATE относится то, что если в третьем аргументе в нужной позиции не обнаружено никакого символа (или это пустая строка, или 2-х аргументная форма), то функция выполняет замену на пустую строку, или удаляет символ. Например, чтобы удалить все пробелы в строке нужно заменить пробел на пустую строку:

```
USER>s str="ABBA Call All"

USER>w $tr(str," ","")
ABBACallAll
USER>w $tr(str," ")
ABBACallAll
```

При этом функция не может провести замену нескольких символов на один или одного на несколько.

Функция $TRANSLATE зачастую используется разработчиками в задачах преобразования строк таким образом, какой не используется в других языках. Например, задача удаления всех цифр из строки:

```
USER>s str="Number 12-345-789 extra"

USER>w $tr(str,"0123456789")
Number -- extra
```

При этом разработчик на MUMPS может использовать этот результат для выполнения замены еще раз:

```
USER>s str="Number 12-345-789 extra"

USER>w $tr(str,"0123456789")
Number -- extra
USER>s str2=$tr(str,"0123456789")

USER>w $tr(str,str2)
12345789
```

чтобы отбросить символы, не являющиеся цифрами и оставить только цифры. Точно также можно оставить в строке только символы из определенного множества:

```
USER>s str="Number 12-345-789 extra"

USER>s str2=$tr(str,"-e")

USER>w $tr(str,str2)
e--e
```

Хотя по определению функция $TRANSLATE выполняет замену на основе исходной строки, указанной в первом аргументе, строку, с которой необходимо провести преобразования, можно использовать для преобразования иного рода, если использовать в третьем аргументе. В этом случае строка с данными выполняет роль шаблона, на который необходимо провести замену.

Например, при использовании строки в третьем аргументе можно выполнить реверс строки, указав соответствующие первый и второй аргументы:

```
USER>s str="ABBA Call All"

USER>w $tr("abcdefghijklmn","nmlkjihgfedcba",str)
llA llaC ABBA
```

Или для выполнения форматирования (дата в формате YYYYMMDD в формат DD.MM.YYYY):

```
USER>s date="20120312"

USER>w $tr("Dd.Mm.YyUu","YyUuMmDd",date)
12.03.2012
```

или выполнение перестановок фрагментов строки:

```
USER>s date="12.03.2012"

USER>w $tr("YyUu.Mm.Dd","Dd.Mm.YyUu",date)
2012.03.12
```

Здесь первый параметр описывает выходной формат, второй - входной, третий - собственно значение, содержащее символы, на которые необходимо выполнить замену.

Одним из наиболее частых применений функции $TRANSLATE является переносимый код поднимания и опускания регистра. В этом случае разработчик указывает две строки - в нижнем и верхнем регистре. Вне зависимости от реализации функция $TRANSLATE является стандартной, поэтому операция поднимания и опускания регистра, выполненная таким образом, переносима. Например, фрагмент такой функции:

```
lowercase(str)
 q $tr(str,"АБВГДЕЖ...","абвгдеж...")
uppercase(str)
 q $tr(str,"абвгдеж...","АБВГДЕЖ...")
```

Другим распространенным применением функции $TRANSLATE является перекодирование символов из одной кодировки в другую. У многих национальных алфавитов существует несколько способов кодирования символов, и в различных прикладных системах или форматах данных могут использоваться различные кодировки. При переносе данных их необходимо привести к используемой кодировке путем просто замены одних байт на другие по таблице. Задача переноса данных может возникать в различных задачах. Автор сталкивался с перекодированием из одной национальной кодировки в другую и при импорте данных в базу данных, и при выдаче данных веб - браузеру.

1.5.17 $JUSTIFY

Функция $JUSTIFY ($J) выполняет форматирование строки с выравниванием вправо с дополнением пробелами.

Первоначально, для первых реализаций MUMPS систем, основным способом ввода-вывода для них был алфавитно-цифровой, это устройства типа терминалов и консолей различного рода. Традиционно, практически все современные реализации MUMPS систем также поддерживают такой способ. Это могут быть консоли, телнет - интерфейсы, терминальные подключения. Для таких средств отображения, как и для формирования текстовых отчетов, важно показывать некоторые данные выровненно (например, при выводе в колонки).

В настоящее время функция $JUSTIFY по-прежнему применяется там, где это необходимо, в различных программах, ориентированных на терминальный алфавитно-цифровой интерфейс, или при формировании текстовых отчетов, или при формировании текста, который должен быть показан одноширинным шрифтом.

Функция $JUSTIFY возвращает строку с добавленными лидирующими пробелами так, чтобы общая длина результата была равна заданной. Второй аргумент задает ширину форматирования. Например:

```
USER>w $j("text",8)
    text
USER>w $j("text",12)
        text
```

В некоторых задачах бывает необходимо добавить пробелы не спереди строки, а сзади. Такой результат можно получить, комбинируя функцию $JUSTIFY и функцию $REVERSE:

```
USER>w $re($j($re("text"),8)),"|"
text    |
USER>w $re($j($re("other"),8)),"|"
other   |
USER>w $re($j($re("any text"),8)),"|"
any text|
```

В дополнение к двухаргументной форме функция $JUSTIFY поддерживает трехаргументную форму, специально ориентированную на форматирование чисел. Третий аргумент задает число десятичных знаков после запятой, до которого следует округлить результат. В этом случае первый аргумент вычисляется как число:

```
USER>w $j(12,8,2)
   12.00
USER>w $j(12.1356,8,2)
   12.14
USER>w $j(-12.1356,8,2)
  -12.14
```

Для трехаргументной формы важно отметить, что округление выполняется для модуля числа, а не для его значения.

Обычно функционал формирования представления данных по значениям данных выполняется непосредственно программой, производящей визуализацию результатов. В ее же функции входит использование национальных стандартов форматирования чисел по группам разрядов и использование десятичного разделителя по национальному стандарту.

В то время, когда была стандартизирована функция $JUSTIFY, она была определена так, что не использует национальные стандарты форматирования, и применяет формат, не зависящий от настроек сервера или клиента. С одной стороны, этот факт не слишком практичен для формирования полноценных отчетов в национальных стандартах, но, с другой стороны, из-за фиксированности формата позволяет применить дополнительные преобразования, например, комбинируя с функцией $TRANSLATE, для получения корректного представления данных.

1.5.18 $FNUMBER

Функция $FNUMBER ($FN) - это вторая, дополнительная к функции $JUSTIFY, функция, форматирующая числа.

Функция имеет две формы - двухаргументная задает исходное число и формат, и трехаргументная задает исходное число, формат, и число десятичных знаков.

Формат задается последовательностью специальных символов:

Код формата	**Результат выполнения функции**
`+ (плюс)`	Добавляет знак + ко всем положительным значениям числа.
`- (минус)`	Запрещает (подавляет) знак у отрицательных значений числа, т.е. выдаёт абсолютное значение числа.
`, (запятая)`	Вставляет в форматируемое число запятую через каждую третью позицию слева от десятичной точки, за исключением лидирующей позиции.
`T или t`	Показывает число со знаком + или -, идущее за Числом. Если знак подавляется (т.е. положительное число или код форматирования -), добавляется хвостовой пробел.
`P или p`	Представляет отрицательное значение числа в скобках. Если же значение числа положительное, то вместо скобок будут вставлены пробелы.

Определения форматов были стандартизованы по имевшимся американским стандартам представления чисел. Комбинируя с описанной ранее функцией $TRANSLATE, можно получить соответствие другим

национальным стандартам. Например, пусть есть значение 123456,789 и его необходимо представить по национальному стандарту с десятичным разделителем запятая, с разделителем групп разрядов пробел и с округлением до двух знаков. В этом случае применяется последовательно форматирование с помощью $FNUMBER и $TRANSLATE:

```
USER>s value=123456.789

USER>s value2=$fn(value,",",2)

USER>w
value=123456.789
value2="123,456.79"

USER>w $tr(value2,",."," ,")
123 456,79
```

Для форматирования с выравниванием вправо дополнительно нужно применить функцию выравнивания $JUSTIIFY:

```
USER>w $j($tr(value2,",."," ,"),14)
    123 456,79
```

Так же, как и в случае с функцией $JUSTIFY, современные клиент-серверные системы используют форматирование по национальным стандартам согласно настройкам клиентских компьютеров, и функция $FNUMBER используется все реже. Но, с другой стороны, эта функция, как и все определения ее форматов, являются частью жесткого стандарта и имеются во всех реализациях MUMPS систем.

В случае, если отчет необходимо формировать на сервере, разработчики либо форматируют числа в едином стандарте представления независимо от того, какому клиентскому месту предназначен отчет, либо учитывают, с какого клиентского места запрашивается формирование отчета и на сервере описывают отдельно правила форматирования по соответствующим для каждого клиентского места национальному стандарту или по его индивидуальным настройкам.

Служебные функции языка MUMPS включают функции, не оперирующие именами, переменными или строками, а выполняющие специфические действия для контекста исполнения. К таким относятся фунции:

1. $TEXT - возвращает строку текста рутины.

2. $RANDOM - генерирует следующее псевдослучайное число.

3. $VIEW - выполняет операции с контекстом процесса, сервера, компьютера, иного операционного окружения, в зависимсти от реализации.

4. $SELECT - выполняет выбор значения в зависимости от последовательного набора условий.

5. $STACK - возвращает информацию о состоянии стека процесса.

1.5.19 $TEXT

Функция $TEXT ($T) возвращает строку исходного текста программы. В аргументе нужно указать имя рутины, имя метки и смещение от метки. Любой из элементов может быть опущен, главное, чтобы по оставшемуся функция смогла понять, какая строка была идентифицирована аргументом.

В отличие от строковых функций, функция $TEXT принимает не строку с идентификацией строки, а имеет предопределенный синтаксис. Аргумент задается почти так же, как задается метка для вызова подпрограммы или для перехода по GOTO.

При этом, так же как и команды, функция $TEXT поддерживает косвенность аргумента. Если аргумент сформирован как строка, то ее можно передать функции $TEXT косвенно.

Если в аргументе опущено имя рутины, то функция ищет строку в текущей рутине. Если имя метки опущено, то функция отсчитывает строку от первой строки по порядку. Если смещение опущено, то функция возвращает строку с указанной меткой или первую строку, если опущена и метка.

Исключение из основных правил для $TEXT - опускание метки и использование смещения, равного 0. В этом случае функция $TEXT возвращает имя рутины. Например, если выполняется вызов

```
w $t(+0)
```

то функция возвращает имя текущей исполняемой рутины.

Точно так же, как и для команды перехода, смещение должно быть неотрицательным числом. Это требование стандарта переносимости. В техническом отношении реализации MUMPS систем могут выполнить отсчет и в обратном направлении, но по стандарту это запрещено и поэтому никакой из реализаций не поддерживается.

Формально говоря, система исполнения MUMPS определена так, что ей задаются для исполнения исходные коды рутин (программ). Формат и

способ хранения остаются на усмотрение реализации, но любая из них предоставляет функцию $TEXT.

Но при этом часть реализаций, например, MiniM и Caché, поддерживают компиляцию в байткод. В этом случае хранение исходного текста необязательно, более того, он может отсутствовать и в систему может быть импортирован только компилированный код без исходного текста. В этом случае функция $TEXT может не найти строку исходного текста и вернуть пустую строку.

Обе эти системы при этом поддерживают правило, что если в строке исходного кода стоит комментарий, начинающийся на не один, а на два символа точка с запятой (;;), то эта строка целиком помещается в байткод так, что в дальнейшем функция $TEXT ее может вернуть.

Получение исходного текста рутины функцией $TEXT это не основное ее применение. Разработчики прикладных систем на MUMPS часто используют возможность вернуть часть строки исходного кода для получения константных значений. Прикладная система разрабатывается так, чтобы можно было импортировать и компилировать сами исходные тексты рутин. При этом разработчики располагают тут же необходимые для работы программы константные значения, например описания экранных форм, константные справочники, заголовки, предопределенные данные с разделителями, и многое другое.

Для большинства языков программирования могут быть предусмотрены, например, таблицы констант, к которым может обратиться программа. Но в языке MUMPS нет такого элемента, как предопределенно существующая инициализированная переменная - есть либо константа в тексте программы, либо переменная, но ее необходимо предварительно создать. Выходом из ситуации является составление подпрограммы, которая по условному номеру константы вернет значение этой константы или набора подпрограмм на каждую константу. По сути, примерно в этих же целях и используются вызовы функции $TEXT.

Как пример использования массива констант приведем пример из прикладной системы WorldVista, фрагмент проверки целостности рутин:

```
...
F I=1:1 S X=$T(ROU+I),T=$P(X," ",1),U=$P(X,";;",2)
...
ROU ;;
A1B2ADM ;;78130
A1B2BGJ ;;138838
A1B2MAIN ;;211790
A1B2MSP ;;113014
A1B2MUT ;;182043
A1B2OLC ;;88060
```

```
A1B2OSR ;;173825
A1B2OSR1 ;;236003
...
```

Здесь рутина после метки ROU содержит метки, совпадающие по именам с именами проверяемых рутин, после которых через разделители указаны контрольные суммы для каждой из них. Небольшой фрагмент кода, написанный однажды, в дальнейшем, при изменении или добавлении рутин, не изменяется, просто корректируется соответствующая строка в таблице меток.

Фактически, вызов

```
$T(ROU+I)
```

и реализует аналог предопределенной таблицы значений, существующей к моменту обращения к ней, но не в виде переменных языка.

Другим вариантом выполнения тех же действий может быть создание отдельной рутины, которая должна быть выполнена при инсталляции и заполнить определенную глобаль определенными значениями, при выполнении программы она уже могла бы обратиться к этой глобали. Но такой способ гораздо менее технологичен в эксплуатации и разработке.

При этом нужно отметить, что многие sql-based прикладные системы примерно так и построены - в них ведется большое число отдельных специальных таблиц с фактически константными данными.

1.5.20 $RANDOM

Функция $RANDOM ($R) выполняет генерацию очередного псевдослучайного целого числа. Аргумент задает число - границу диапазона генерации. Функция возвращает случайное целое число от 0 включительно до значения аргумента, не включая его.

Например, чтобы сгенерировать псевдослучайную последовательность из нулей и единиц, используется серия обращений к функции

```
$random(2)
```

Значение аргумента не должно быть отрицательным числом, в этом случае функция генерирует ошибку.

Стандарт языка не специфицирует характер и тип генератора псевдослучайной последовательности. Большинство современных реализаций

MUMPS систем используют штатный генератор из стандартной библиотеки языка C. Результат такого алгоритма имеет вполне среднюю гистограмму распределения и годится для большинства технических применений. При этом, если необходимо использовать прецизионный генератор псевдослучайных последовательностей, рекомендуется либо проверить имеющийся на соответствие требованиям, либо использовать внешние средства.

1.5.21 $VIEW

Функция $VIEW ($V) предоставляет доступ к функционалу, зависящему от реализации MUMPS системы. Функция $VIEW является парной к команде VIEW, и используется, если необходимо вернуть значение, передать несколько параметров и так далее. Совместно с командой VIEW функция $VIEW реализует специфические для MUMPS системы интерфейсы, не предусмотренные стандартом.

Традиционно это, в зависимости от значений параметров, доступ к операционному окружению, внутреннему контексту процесса, внутреннему контексту сервера, внешнее взаимодействие.

Значение функции $VIEW, как и команды VIEW, необходимо смотреть в документации на используемую MUMPS систему. В целом, именно места использования функции $VIEW являются одними из кандидатов на замену при необходимости портирования кода на другую MUMPS систему. В другой системе тот же функционал может или отсутствовать и его необходимо выполнять внешними средствами, или может быть выполнен иначе.

1.5.22 $SELECT

Функция $SELECT ($S) возвращает значение вычисленное в зависимости от последовательной серии условий. Функция имеет нестандартный синтаксис и нестандартную очередность вычисления аргументов.

Функции передается один или более аргументов, перечисленных через запятую, но каждый из аргументов является парой из выражения, задающего условие, и выражения, задающего возвращаемое значение, разделенные двоеточием:

```
s $s(a=2:"two",a=1:"one",1:"other")
```

В определенном смысле функция $SELECT не является функцией в прямом понимании языка. Это выражение алгоритма вычисления выражения. Функция вычисляет условие для каждого из своих аргументов

подряд, начиная с первого. В случае если условие выполнилось, функция прекращает дальнейшие вычисления и возвращает соответствующее значение.

Также нужно отметить, что если ни одно условие не выполнилось, то функция генерирует ошибку. Разработчики на MUMPS традиционно добавляют последним аргументом условие истинности и соответствующее этому условию значение для возврата. Это значение будет возвращено если ни одно другое условие не выполнилось.

Порядок вычисления подряд один за другим важен для языка MUMPS, поскольку часть из аргументов может быть не вычислена и выполнение побочных эффектов может зависеть не от наличия, а от значений аргументов. Если функция определила, что одно из условия истинное, то остальные условия не проверяются и не вычисляются.

Функция $SELECT относится к стандартным функциям языка и поддерживается всеми современными реализациями MUMPS систем.

В дополнение к ней в системе Caché была добавлена функция $CASE. Она принимает первым аргументом значение выражения, остальные указываются парами в виде значения для сравнения и значения для возврата, если сравнение дало истинность. Для последней пары допускается опускать значение для сравнения:

```
$CASE(target,case:value,case:value,...,:default)
```

Функция $CASE выполняет наиболее часто используемую форму применения функции $SELECT - сравнение значения с перечисленными. Отличие от $SELECT в том, что функция $CASE вычисляет значение аргумента target лишь однократно, и проводит операции только сравнения. В случае если необходимо получение условия истинности не сравнением, а другим оператором или функцией, необходимо использовать функцию $SELECT.

Другим важным синтаксическим отличием от функции $SELECT является то, что функцию $CASE можно использовать в качестве аргумента команды GOTO или DO и вместо значений value в этом случае нужно указывать метки. Причем, если функция $CASE является аргументом команды DO, то метка может также содержать параметры. Таким образом, синтаксис аргументов зависит от того, аргументом чего является функция $CASE.

В случае использования функции $SELECT и необходимости выполнить переход или вызов подпрограммы в зависимости от серии условий разработчики указывают функции $SELECT возврат строки с меткой и рутиной, и применяют к результату операцию косвенности. Синтаксически можно использовать косвенность для аргумента практически любой

команды, и в этом случае строковые возвраты функции $SELECT и строковый вариант функции $CASE более гибки.

При необходимости выполнить замену функции $CASE как аргумента команды DO, например кода

```
do $case(a,1:proc1(a),2:proc2(a),:proc3(a))
```

можно использовать постусловия на аргументы команды DO:

```
do proc1(a):a=1,proc2(a):a=2,proc3(a):(a'=1)&(a'=2)
```

В этом случае необходимо учесть, что в случае использования функции $CASE будет лишь один вызов команды DO, а при перечислении аргументов через запятую команда DO проверяет каждое из постусловий слева направо и, в зависимости от их формулировки, может вызвать несколько подпрограмм из списка. Кроме того, при вычислении постусловия, на результат могут оказать влияние побочные эффекты от вычисления предыдущих постусловий.

Очередность вычисления в функции $CASE и проявление побочных эффектов не определены стандартом и фиксируются производителем системы Caché.

В отличие от конструкций переключателя switch в языке C или case в языке Pascal в языке MUMPS в функциях $SELECT и $CASE условия не обязательно константы, это произвольные вычисляемые выражения и функция $SELECT для вычисления истинности условия может использовать произвольные операторы и функции.

1.5.23 $STACK

Функция $STACK ($ST) возвращает состояние стека текущего процесса с заданного уровня.

Новый уровень стека создается вызовом подпрограммы командой DO, вызовом пользовательской функции, или выполнением команды XECUTE.

При старте процесса уровень стека равен 0, при создании каждого нового стекового фрейма уровень стека увеличивается на 1, и при возврате на предыдущий уровень уменьшается на 1.

Функция $STACK традиционно используется в паре с системной переменной $STACK, возвращающей текущий уровень стека.

Функция поддерживает две формы - одноаргументную и двухаргументную. В одноаргументной форме функция принимает номер уровня стека и возвращает строку с одним из предопределенных значений, означающих способ как был сформирован этот уровень:

DO	Если текущий контекст образован выполнением команды do с параметрами или без них.
XECUTE	Если текущий контекст образован выполнением команды xecute.
$$	Если текущий контекст образован вызовом подпрограммы с возвращаемым значением.

В двухаргументной форме функция дополнительно принимает второй аргумент, который должен быть одним из следующих значений:

PLACE	Возвращается место программы, которое исполняет строку на указанном уровне исполнения программы.
MCODE	Возвращается строка программы, которая исполняется на указанном уровне исполнения.
ECODE	Возвращается строка с ошибкой, если на этом уровне стека произошла ошибка.

В некоторых нестандартных случаях у функции $STACK действуют дополнительные правила возврата.

Если на запрошенном уровне стека исполняется команда xecute, то при задании второго параметра равном "PLACE" функция возвращает строку из одного символа @. Контекст создания стекового фрейма командой XECUTE отличается от вызова подпрограммы тем, что стек создается, но у исполняемой строки команд нет положения в рутине.

В случае если исполняется строка без метки, то при втором параметре равном "PLACE" функция формирует возврат как метка + смещение + имя рутины. Метка и смещение в этом случае отсчитываются от последней пройденной процессом метки при выполнении программы.

Функция $STACK вместе с системной переменной $STACK входит в стандартные функции, ее поведение жестко определено стандартом и поддерживается всеми современными реализациями.

У MUMPS систем есть особенность поведения функции $STACK при возникновении ошибки. От момента возникновения ошибки до присваивания переменной $ECODE пустой строки функция $STACK возвращает значения для копии стека, замороженного состояния бывшего на момент возникновения ошибки. После присваивания пустой строки переменной $ECODE функция $STACK продолжает возвращать значения о реальном состоянии стека текущего процесса.

Традиционно состояние стека анализируется при отладке и зачастую дополнительно протоколируется в обработчике ошибки, чтобы при чтении дампа состояния разработчики могли найти место проявления про-

блемы и увидеть ход выполнения процесса.

1.6 Списковые функции

Списковые функции семейства $LIST не входят в стандарт языка MUMPS и были самостоятельно разработаны и предложены в MUMPS системе Caché фирмой производителем. Разработчики высоко оценили их практичность и полноту, и списковые функции в настоящее время широко применяются в различных разработках. В дальнейшем списковые функции в точно таком же определении были реализованы также в MUMPS системе MiniM.

При использовании списковых функций и необходимости в дальнейшем портирования кода нужно проверить, поддерживаются ли они на целевой системе или версии системы.

Структурно списковые функции рассматривают строку с значением как последовательность байт, в которой присутствуют отдельные элементы, следующие друг за другом. Каждый элемент содержит специальный тег, последовательность байт, описывающую длину элемента и тип содержащегося в нем значения (для кодирования чисел).

Кодирование списков предусматривает специальный индикатор, описывающий что элемент в списке есть, но имеет неопределенное значение.

При вычислении элемента списка списковые функции последовательно просматривают элементы списка. Если задана операция с элементом списка номер N, то функция последовательно пропускает N - 1 элементов.

Каждый из элементов может иметь произвольную длину, ограничение накладывается на общую длину значения полного списка как строки. Любой элемент может содержать любые значения символов, в том числе сами в свою очередь могут рассматриваться программно как списки. Такое кодирование значения подволяет преодолеть недостатки позиционного структурирования функции $EXTRACT и резервирования специального значения для разделителя для функции $PIECE.

Пустая строка списковыми функциями считается не состоящей ни из одного элемента.

Каждая из строк, которые рассматривают списковые функции, не имеет индикатора, каким образом получена эта последовательность байт. Это могут быть константы, результат строковых функций, списковых функций, чтения из устройства, арифметических или любых других операций. Если строка по своему формату не подходит под определение списка, то списковые функции генерируют ошибку.

Основным способом получения списка является специальная функция $LISTBUILD ($LB). Эта функция принимает переменное число аргументов, в том числе нулевое. Любое число аргументов могут быть опущены. Для опущенных аргументов функция генерирует в соответствующей позиции элемент с неопределенным значением. Примеры:

```
USER>s list=$lb(1,12,123,,12345)

USER>f i=1:1:$ll(list) w $lg(list,i,"undefined value"),!
1
12
123
undefined value
12345
```

Здесь функция $LISTBUILD составила список из значений 1, 12, 123, неопределенного, и 12345. Затем в цикле по числу имеющихся элементов ($LISTLENGTH, $LL) были выведены значения списка, для неопределенных его значений использована строка "undefined value".

Структурно список есть последовательность элементов списка, следущих друг за другом. Поэтому список можно получить также конкатенацией двух списков. В результате получается новый список, в котором сначала идут элементы из первого списка, и сразу за ними элементы из второго списка, включая неопределенные.

```
USER>s list1=$lb("a","b","c")

USER>s list2=$lb(11,22,33)

USER>s list3=list1_list2

USER>f i=1:1:$ll(list3) w $lg(list3,i,"undefined"),!
a
b
c
11
22
33
```

Для получения значения элемента списка используются функции $LIST ($LI) и $LISTGET ($LG). Им нужно указать значение списка и номер позиции элемента. Позиции отсчитываются от единицы. Для этих функций действует специальное соглашение о значении -1 как о синониме позиции последнего элемента.

Обе функции возвращают значение элемента списка. Отличие в том, что если элемент имеет неопределенное значение, то функция $LIST генерирует ошибку, а функция $LISTGET возвращает значение по умолчанию. Если оно не было указано, то функция использует в качестве значения по умолчанию пустую строку.

Для обеих этих функций элемент имеет неопределенное значение, если элемент присутствует, но имеет специальное неопределенное значение и если номер позиции находится за пределами существующего числа элементов.

При этом различается неопределенное значение списка и неопределенное значение переменной, передаваемое в качестве списка. Во втором случае управление до самой списковой функции не доходит, так как MUMPS система самостоятельно обнаруживает неопределенность значения.

Для замены элемента списка нужно использовать левостороннюю форму функции $LIST и указать номер позиции для замены и значение для замены. Функция может заменить как имеющиеся, так и еще несуществующие элементы. если указанной переменной не существовало, то она автоматически будет создана и исходное состояние такого списка будет считаться пустым, не имеющим ни одного элемента.

```
USER>s list=$lb(,,)

USER>f i=1:1:$ll(list) w $lg(list,i,"no exist"),!
no exist
no exist
no exist

USER>s $list(list,2)=222

USER>f i=1:1:$ll(list) w $lg(list,i,"no exist"),!
no exist
222
no exist

USER>s $list(list,4)=444

USER>f i=1:1:$ll(list) w $lg(list,i,"no exist"),!
no exist
222
no exist
444

USER>s $list(list,7)=777
```

```
USER>f i=1:1:$ll(list) w $lg(list,i,"no exist"),!
no exist
222
no exist
444
no exist
no exist
777
```

Что интересно, функция $LIST поддерживает также трехаргументную форму, в которой функция рассматривает не одно значение, а подсписок. При присваивании в левосторонней форме функция заменяет подсписок, а при возврате значения возвращает подсписок.

В частности, поскольку нельзя присвоить неопределенное значение, можно выполнить ту же операцию путем замены на элемент с неопределенным значением:

```
USER>s list=$lb(11,22,33)

USER>f i=1:1:$ll(list) w $lg(list,i,"no exist"),!
11
22
33

USER>s $list(list,2,2)=$lb()

USER>f i=1:1:$ll(list) w $lg(list,i,"no exist"),!
11
no exist
33
```

Обе формы присваивания функции $LIST, и двухаргументная, и трехаргументная, автоматически дополняют список, если выполняется присваивание или замена несуществующих элементов списка. Дополнение производится специальными элементами неопределенного значения.

Трехаргументная форма функции $LIST при присваивании также может быть использована для удаления элемента списка или произвольного подсписка из любого числа элементов. Например, для удаления элементов с второго по четвертый включительно:

```
USER>s list=$lb(11,22,33,44,55)

USER>s $list(list,2,4)=""

USER>f i=1:1:$ll(list) w $lg(list,i,"no exist"),!
11
55
```

Для удаления одного элемента списка в качестве начального и конечного номера элемента указывается одно и то же число, например для удаления пятого и третьего элементов списка:

```
USER>s list=$lb(11,22,33,44,55,66,77)

USER>s $li(list,5,5)=""

USER>f i=1:1:$ll(list) w $lg(list,i,"no exist"),!
11
22
33
44
66
77

USER>s $li(list,3,3)=""

USER>f i=1:1:$ll(list) w $lg(list,i,"no exist"),!
11
22
44
66
77

USER>
```

Удалить элемент списка с использованием двуаргументной формы не получится, поскольку она заменяет значение в позиции списка.

С помощью трехаргументной формы функции $LIST также можно заменить элемент или несколько элементов списка на элемент с неопределенным значением, выполнив замену на список из одного или нескольких неопределенных значений:

```
USER>s list=$lb(11,22,33,44,55,66,77)

USER>s $li(list,3,3)=$lb()

USER>s $li(list,5,6)=$lb(,)

USER>f i=1:1:$ll(list) w $lg(list,i,"undefined"),!
11
22
undefined
44
undefined
undefined
```

```
77

USER>
```

К списковым функциям, поддерживаемым системами MiniM и Caché, относятся функции:

1. $LISTBUILD - создает список из указанных значений.
2. $LIST - возвращает значение элемента списка или подсписок, присваивает значение элементу списка или подсписку.
3. $LISTGET - возвращает значение элемента списка или значение по умолчанию.
4. $LISTDATA - возвращает индикатор, определен ли элемент списка или нет.
5. $LISTFIND - выполняет поиск значения среди элементов списка.
6. $LISLENGTH - возвращает число элементов списка.
7. $LISTSAME - сравнивает списки с учетом возможного различия кодирований чисел и строк.

Дополнительно система Caché в зависимости от версии поддерживает, а система MiniM не поддерживает следующие функции:

1. $LISTFROMSTRING - формирует список по исходной строке с разделителями.
2. $LISTNEXT - возвращает следующий элемент списка, используя специальный итератор, позволяющий не просматривать весь список с самого начала.
3. $LISTTOSTRING - по заданному списку формирует строку с заданным раделителем.
4. $LISTVALID - проверяет является ли значение корректным списком.

Насколько известно, другие реализации MUMPS систем не поддерживают в настоящее время списковые функции. Хотя в свое время для GT.M предлагался отдельный патч, реализующий упрощенную поддержку списковых функций, в релиз GT.M они не были внесены.

1.7 Битовые функции

Битовые функции семейства $BITXXX были разработаны производителем MUMPS системы Caché и впоследствии также и в том же виде и поведении были реализованы в MUMPS системе MiniM. Отличие этих реализаций состоит в формате хранения собственно битовых строк. Если в MiniM функции $LISTXXX были выполнены в точном соответствии и с форматом хранения Caché и данные могут быть перенесены без изменений, то при переносе данных битовых строк с MiniM на Caché или обратно они должны быть перегенерированы.

Первоначально различные реализации MUMPS систем поддерживали (и сейчас поддерживают для совместимости) битовые строки для семейства функций $ZBITXXX. Это функции, оперирующие строками как последовательностями бит. У этого семейства функций есть ключевые недостатки, не позволяющие их применять для построения битовых индексов, но допустимые для передачи битовых значений как состояния данных:

1. Любые операции над строками $ZBITXXX есть также строки, поэтому запись в базу данных результата есть запись целой строки.

2. Чтобы хранить биты, необходимо предварительно создать битовую строку необходимой длины.

Оба семейства битовых функций, и $BITXXX и $ZBITXXX не входят в стандарт, но различные реализации MUMPS систем принимают меры по переносимости кода и, во многих случаях, по совместимости форматов, таким как используемым функциями $ZBITXXX.

Основными отличиями битовых строк этих двух реализаций являются:

$ZBITXXX	$BITXXX
Для установки бита необходимо начальное значение переменной.	Для установки бита начальное значение переменной не требуется.
Битовая строка должна быть создана на необходимое количество битов.	Можно устанавливать и читать бит за пределами имеющегося количества битов.
Для изменения бита возвращается новое значение битовой строки отдельной $ZBITSET функцией.	Для изменения бита используется команда set с левосторонней формой функции $BIT, указывающей позицию для изменения.

Журналирование записи в глобал выполняется для всего нового значения целиком.	Журналирование записи в глобал выполняется только для измененного бита.
При откате транзакции откатывается все значение записанное в глобал целиком.	При откате транзакции откатывается только определенное изменение бита.
Битовые строки хранятся в несжатом виде.	Caché хранит битовые строки в компрессированном по необходимости формате.

Именно внутренние технологические решения функций семейства $BITXXX и позволили использовать битовые индексы в транзакционных системах, или в задачах OLAP в режиме read / write, а не только в традиционных задачах OLAP read only на отдельных серверах.

Для установки одного бита используется функция $BIT в двухаргументной форме. Первый аргумент - это имя локальной или глобальной переменной, второй аргумент - позиция в битовой строке. Присваиваемое значение приводится к булевскому (0 или 1) и переменная устанавливается в значение так чтобы все остальные имеющиеся биты сохранили свое значение, а бит в указанной позиции принял заданное значение. Если переменной не было, то она создается, а если позиция выходит за пределы существовавших битов, битовая строка дополняется нулевыми битами.

Та же функция $BIT в двухаргументной форме используется для чтения бита. Для единичного возвращает значение 1, для нулевого значение 0. Если переменной не существовало, то она не создается, и возвращается значение 0. Если запрашивается бит за пределами имеющегося числа битов, то возвращается значение 0.

Определение поведения функции $BIT построено так, чтобы быть наиболее практичным для битовых индексов. Традиционно в битовых индексах значение 1 означает существование записи с идентификатором с номером равным позиции бита. При создании записи записывается значение 1. При удалении - 0. Если не было записи, то это логически означает эквивалент нуля.

Теми же соглашениями пользуется функция битовых операций над битовыми строками $BITLOGIC. При этом надо быть внимательным с интерпретацией инверсии битовой строки - функция получает инверсию только для имеющегося числа битов без расширения битовой строки до максимума. В частности, если есть битовый индекс поддерживаемый для индикатора существования объектов (extent - index), то по нему нельзя определить множество объектов, в которое входят несуществующие объ-

екты, но можно проверить существование или несуществования одного объекта или множества объектов, заданных своими идентификаторами.

Для организации итераций по битовой строке поддерживаются функции $BITCOUNT и $BITFIND. Первая в одноаргументной форме возвращает общее число имеющихся битов, а в двухаргументной форме вычисляет второй аргумент как булевское значение (ноль или единица) и возвращает число соответственно нулей или единиц. Функция $BITFIND возвращает позицию бита с заданным значением (ноль или единица), следующим после заданной позиции. Кроме того, опционально может быть указано направление поиска - по возрастанию (действие по умолчанию) или по убыванию.

В совокупности с операторами логических битовых операций, поддерживаемыми функцией $BITLOGIC, этот набор функций составляет полный необходимый набор для реализации битовых индексов.

Несмотря на то, что и функции семейства $LISTXXX и семейства $BITXXX не входят в стандарт, они не имеют лидирующего символа Z и не гарантируют, вообще говоря, переносимости кода. В настоящее время поддержка и совместимость этих функций между разными MUMPS системами и разными их версиями обеспечивается только их производителями. При этом оба семейства функций получили высокую оценку у разработчиков по практичности своего применения и полноте определения операций.

1.8 Модули

Модулями в языках программирования называют программные элементы, группирующие определение алгоритмов, данных, типов в относительно цельную по смыслу группу.

Основным назначением модуля является создать такой элемент программы, чтобы он находился по иерархии между отдельной функцией и библиотекой так, чтобы перенос или использование такого элемента означало цельный перенос и готовность использования в другой программе. Отличие модуля от просто набора функций состоит в том, что модуль содержит также и определения некоторых данных. Эти данные могут быть как внутренними служебными данными модуля и невидимыми из других частей программы, так и общими данными, предоставленными другим частям, или функциям.

Понятие модуля в модульных языках выражается также в виде синтаксической конструкции - файл, как единица трансляции, или набор файлов. В случае, если такой файл содержит служебные данные, то он

уже может называться не просто набором функций, а уже модулем.

Кроме того, в модульных языках присутствует операция инициализации начальных значений данных модуля. Все модульные средства разработки так или иначе поддерживают понятие программы как цели сборки - в результате написания программистом кода должно получиться полное определение программы, ее алгоритмов и используемых данных.

При этом данные модуля в случае его использования в программе автоматически входят в данные программы. Современные компоновщики (линкеры) могут не включать в получаемую программу неиспользуемые данные или функции, даже если они объявлены в модуле. Используется ли определенная функция или пременная модуля или нет, это линкер определяет по списку взаимозависимостей.

Наличие специальных соглашений о порядке инициализации данных, объявленных модулем, означает, что перед выполнением кода, указанного программистом, программа автоматически вызывает код инициализации данных, или загружает их в память из файла с программой.

К модульным языкам относятся как компилирующие, так и интерпретирующие, или скриптовые. В случае применения интерпретатора необходимо, либо чтобы среда разработки поддерживала объединение модулей для среды исполнения так, чтобы при запуске программы среда исполнения выполнила код инициализации модулей, либо один из основных файлов итоговой программы должен вызвать определенные функции (или директивы в зависимости от языка), чтобы указать среде исполнения на применение определенных модулей.

В языке MUMPS модулей в таком определении как они используются в модульных языках, нет. Фактически, процесс в MUMPS системе не имеет такой организации кода, чтобы что-то можно было назвать единым словом "программа", или подпрограмма. Какой код является или, правильнее сказать, используется как подпрограмма или программа или библиотека функций, целиком определяется выполнением процесса.

Сервер хранит лишь совокупность алгоритмов, описанных в рутинах, и совокупность данных. Процесс может начать выполнение с любой рутины и с любой метки в любой рутине. При этом отсутствует такое понятие, как процесс инициализации используемых модулей.

Более того, если процесс вызвал рутину, то это вовсе не означает, что рутина может определить свои собственные данные, не хранимые на диске в глобалах. Рутина может использовать локальные переменные процесса, но у процесса не может быть несколько переменных, принадлежащих только рутине.

Довольно сложно представить себе ситуацию, чтобы рутина взяла и самостоятельно автоматически определила набор переменных и чтобы

процесс при запуске начал автоматически, а не по описанному разработчиком алгоритму, проводить работу по инициализации этих переменных. На сервере при эксплуатации крупных программных систем могут присутствовать тысячи рутин, и один только процесс запуска такой программы может занять длительное время. Кроме того, если множество рутин просто будет содержать (даже неинициализированными) немного собственных данных, которые должны входить в пространство запускаемого процесса, то это автоматически вызовет расход памяти до того, как она в действительности понадобится.

Если в программах клиентского класса это является совершенно нормальным из-за жесткой детерминированности общего порядка выполнения кода, то для программных систем серверного класса так не принято делать. Вместо того, чтобы поддерживать понятие модуля, поддерживается понятие функции. В отдельных случаях для практичности разработки и переноса отдельные функции могут быть перенесены пакетом. Но для разработки программ клиентского класса (запускаемый процесс точно определен по перечню функций, которые предстоит исполнить, то есть полное определение программы) модульная стратегия разработки, несомненно, чрезвычайно практична, поскольку намного проще на клиентских местах использовать несколько отдельных программ, собранных из модулей, чем полный набор из тысяч отдельных функций, которые могут быть вызваны в любое время и из любого места.

И в MUMPS системах, и в SQL системах, и во многих других системах серверного класса сознательно избегают применения понятия модуль в том виде, как он используется в системах клиентского класса. Например, в SQL системах поддерживается понятие таблицы безотносительно привязки к алгоритмам хранимых процедур, и, в зависимости от версии SQL системы, понятие пакета хранимых процедур. И наоборот, произвольная хранимая процедура может быть вызвана независимо от наличия остальных процедур. Хотя, конечно, если она использует другую, но отсутствующую, то при выполнении произойдет ошибка.

При необходимости соблюдать в MUMPS системах модульный подход требуется также, как и в модульных языках, выделить следующие элементы:

1. Какая совокупность рутин в данный момент называется программой.

2. Какие локальные переменные должен использовать модуль.

3. Какой префикс должен быть у локальных переменных модуля (предназначенные для использования модулем).

4. Какой код должен быть вызван при старте программы для выполнения инициализации локальных данных модулей.

Вообще говоря, если разработчик программы выполнит все эти условия и будет соблюдать методику модульного построения и дальше, то в итоге его программа так и будет организована. Но, если понадобится использовать отдельные фрагменты (функции, библиотеки) кода в другой программе или системе, то придется тщательно выверять, можно ли это сделать корректно, и есть ли в другой системе такое понятие как код инициализации.

К программным системам, промежуточным между серверными и клиентскими, могут быть отнесены так называемые сервера приложений, которые с одной стороны являются серверными частями для клиентского презентационного слоя, и, одновременно, клиентскими для слоя СУБД. В таких серверах приложений при использовании модулей, содержащих собственные данные с инициализацией, требующей времени, и наблюдается задержка времени при старте процесса. Кроме того, если сервера приложений используют существенные объемы внутренних данных (например, у модуля может быть буфер для форматирования значений), такие сервера приложений в случае крупной системы могут требовать очень большой памяти и выделения под такой сервер приложений отдельного компьютера.

В случае использования MUMPS систем разработчики именно их же возможности и используют в качестве серверов приложений, поскольку язык MUMPS является полноценным алгоритмическим, поддерживает множество способов ввода-вывода и содержит средства эффективного доступа к хранимым данным.

При построении серверов приложений на MUMPS системе разработчики автоматически получают масштабируемость такого сервера приложений и его переносимость на уровне самой используемой СУБД. Кроме того, в таких серверах приложений, что важно, отсутствует этап сложной трансформации данных между получением их с диска в слой самого сервера приложений.

В случае же использования сервером приложений в качестве временных данных глобалов возможности такого сервера приложений по обработке больших объемов ограничиваются не оперативной памятью, а дисковой. При этом сервер приложений для таких временных данных получает автоматически те же средства кеширования и высокоэффективного доступа для обработки огромных объемов, что и слой хранения данных в СУБД.

1.9 Рутины

Рутинами (routine) в языке MUMPS называется исходный текст программ, организованный как совокупность строк. Название давнее, и в языке используется именно такое. Для системы исполнения MUMPS рутина есть последовательность строк, объединенная заданным для них порядком и общим именем.

Текст рутины (или программы) может рассматриваться разработчиком как просто последовательность строк. При выполнении одной строки система исполнения автоматически передает управление на следующую, если не было явных команд передачи управления. Технически внутренняя организация рутин и их разбиение целиком определяется разработчиком. При этом язык MUMPS для строк рутины поддерживает дополнительное правило, не поддерживаемое для строк команд интерактивного режима - приписывание строке метки.

Метки могут быть использованы как для передачи управления, так и для вызова подпрограммы, причем одновременно. Как именно используется метка - также определяется разработчиком. Если с его точки зрения метка предназначена для вызова подпрограммы то такая подпрограмма также называется subroutine. В русском языке не используется термин подрутина или сабрутина, ближайший перевод - это подпрограмма.

При этом в русском языке принято программой называть нечто цельное, организованное единым образом. В MUMPS это называется чаще приложением, и в приложение входит набор рутин. При этом для языка нет отдельного понятия приложение, поскольку способ использования совокупности имеющихся рутин также полностью определяется разработчиком. Общая совокупность рутин в больших программных системах может использоваться и как одно большое приложение и как несколько небольших с общими подпрограммами.

С точки зрения эксплуатации MUMPS системы рутина представляет собой единицу экспорта, импорта, редактирования и компиляции (если система поддерживает компиляцию в байткод). При этом стандартные форматы экспорта и импорта допускают перенос в одном файле как одной, так и нескольких рутин. При импорте текст рутины перезаписывается целиком.

Рутина может рассматриваться как аналог модуля модульных языков программирования, или как единица трансляции, например файл cpp для трансляторов C++ или как файл pas для трансляторов Pascal. При этом, в отличие от многих других средств разработки, в языке MUMPS нет отдельного понятия начало программы, такое как в языке Pascal, например. В языке MUMPS управление может быть передано на любую

строку любой рутины из любого места, в том числе из интерактивного режима, и новый процесс может быть запущен с любой метки любой рутины.

Для эксплуатации набора рутин, таким образом, разработчики на MUMPS указывают не готовый исполняемый файл, а способ выполнения приложения, и указывают метки, рутины, характер параметров, с выполнения которых начинается, по их замыслу приложение. Например, для начала выполнения пакета AltNC указывается, что необходимо выполнить

```
d ^%aNC
```

либо набор альтернативных способов запуска с опциональной передачей параметров.

С точки зрения языка на объем рутин не накладывается никаких ограничений. При этом для языка определены так называемые требования переносимости, в частности в отношении рутин в них входят требования ограничения длины меток и идентификаторов до 8 символов включительно и не иметь строк длиннее 255 символов. При этом каждая из реализаций MUMPS системы, поскольку существует в реальном мире, поддерживает собственные ограничения, вызванные техническими особенностями реализации. Традиционно они соответствуют требованиям переносимости и имеют увеличенные лимиты как на длину идентификаторов, так и на длину строк.

Кроме того, в зависимости от реализации, ограничения могут быть косвенными, такими как в системах MiniM и Caché, где ограничение накладывается не на объем исходного текста, а на объем получаемого байткода. При том, что обе системы имеют различную архитектуру байткода, одна и та же рутина в одной из этих систем может уместиться в отведенные пределы по байткоду, а в другой нет.

С точки зрения языка и эксплуатации MUMPS системы не имеет значения, как именно рутина хранится внутри сервера. Это может быть хранение в глобалах, во внешних файлах, в специальных блоках файла данных, или еще как-то. Так же не имеет значения, как и где хранится байткод. В любом случае MUMPS система самостоятельно поддерживает правила языка по исполнению рутин.

Большинство MUMPS систем, хотя это и не оговорено отдельно в спецификациях языка, поддерживает механизм отображения так называемых системных рутин. Отображение выполняется таким образом, что для любого процесса, в какой бы текущей базе данных он не выполнялся, ему всегда доступны на исполнение системные рутины, хранящиеся

в единственном экземпляре в системной базе данных таким же образом, как если бы они находились в текущей базе данных.

В зависимости от реализации MUMPS системы это может быть отдельная спецификация для базы данных, где указывается какую из баз использовать как системную, или соглашение об именах рутин. В частности, системы MiniM и Caché поддерживают правило, что рутины, имена которых начинаются на символ процента (%), хранятся в системной базе данных и являются системными.

Первоначально в спецификации языка допускалось существование двух или более меток с одинаковыми именами в одной рутине, но впоследствии в стандарт ввели требование считать это ошибкой и при трансляции рутины с двумя или более метками компилятор должен генерировать ошибку компиляции как синтаксическую ошибку. Хотя такое существование нескольких одинаковых меток теоретически и не является препятствием для выполнения рутины, если система сможет передать управление на первую из них.

В отличие от команд, системных переменных и системных функций, имена рутин и меток, как и имена локальных и глобальных переменных, чувствительны к регистру. Разработчикам необходимо обращать внимание на регистр символов в именах рутин и меток. Традиционно разработчики принимают определенную систему именования рутин, некие мнемонические правила для суффиксов и префиксов, чтобы по имени можно было сразу определить, к чему оно относится. Например, если есть много рутин для приложения ABCD, то в нем могут быть рутины ABCDreport01, ABCDfile04 или использующие иные мнемонические соглашения.

Если процесс в какой-то момент времени выполняет код из рутины, то для него эта рутина является текущей. Для текущей рутины действует правило умолчания при использовании меток. Если у метки опущено имя рутины, то для системы это считается эквивалентом требованию использовать текущую рутину. Таким образом, в нескольких рутинах может использоваться одинаковая метка и одинаковый код вызывающий такую метку с опусканием имени рутины, но для каждой из рутин это будут разные метки. В любом случае разработчик может указать для метки имя рутины явно. В этом случае система будет искать метку в именно этой указанной рутине.

Метками называются последовательности символов, задающие именование строки в тексте программы. Метки начинаются с первого символа строки, могут начинаться на символы процента %, цифру или латинскую букву, за которыми следуют необязательные латинские буквы или цифры. Имена меток в тексте одной программы должны различаться.

Имена меток чувствительны к регистру.

Вообще говоря, в языке MUMPS все метки, если они есть, могут быть вызваны из любого контекста, из любой рутины. Что соответствует всеобщей области видимости меток. В более поздних версиях Caché также поддерживается расширенный синтаксис объявления метки внутренней, чтобы ее можно было вызвать только из той же самой рутины, а также объединение областей кода рутин так, что в каждой из таких областей может быть свой набор меток с совпадающими именами в пределах одной рутины.

Метка может иметь или не иметь параметры. В тексте программы параметры метки есть формальные параметры. Параметры заключаются в круглые скобки после имени метки.

Метку можно использовать для вызова подпрограммы командой do, для вызова как функции, возвращающей значение, через синтаксис $$ и для безусловного перехода по команде goto. Кроме того, метки могут использоваться не для передачи управления, а для задания точки отсчета для функции $TEXT. Для перехода по goto метка не должна иметь формальных параметров.

Примеры программ с метками:

```
FUNC(a,b) q a+b*2     ; 1
%NAME q $p($zv," ")   ; 2
```

Здесь в первом примере описана метка FUNC, принимающая два формальных параметра a и b. Эту метку можно вызвать как $$FUNC(), передав ей фактические параметры. Во втором примере описана метка %NAME, она не имеет параметров и её требуется вызывать без указания параметров: $$%NAME.

В той же строке, что и метка, могут находиться команды. При этом перед первой командой должен быть, по меньшей мере, один пробельный символ. При переходе управления на метку начинают исполняться команды, следующие непосредственно за описанием параметров метки.

Вызовы меток без возвращаемого значения через команду do и с возвращаемым значением определяются вызывающей стороной. Подпрограмма вызываемой стороны может определить, в каком контексте была вызвана подпрограмма, используя системную переменную $quit. В случае если ожидается возврат, подпрограмма должна вернуть значение командой quit с параметром - значением возврата, иначе подпрограмма должна вернуть управление командой quit без параметров.

На вызывающей стороне, после указания имени метки, следуют указание смещения и имя рутины, в которой эта метка находится. Если

вызывается метка с параметрами, то указывать смещение недопустимо. Имя метки, смещение и имя рутины могут быть опущены, если оставшейся части полного имени достаточно для вызова. Если имя рутины опущено, то считается, что используется текущая рутина.

Примеры:

```
do LABEL^RTN          ; 1
goto LABEL+3^RTN      ; 2
g 12+4                ; 3
do ^RTN               ; 4
w $$FUNC^RTN($h,ver)  ; 5
```

Здесь в первом примере вызывается подпрограмма без параметров в рутине RTN с метки LABEL, во втором примере выполняется безусловный переход на третью строку после строки с меткой LABEL в рутине RTN, в третьем примере выполняется переход на четвёртую строку после метки 12 в текущей рутине, в четвертом примере вызывается подпрограмма с первой имеющейся строки в рутине RTN, в пятом примере вызывается подпрограмма с метки FUNC в рутине RTN с ожиданием возврата и с передачей в качестве параметров значений системной переменной $horolog и локальной переменной ver.

Если не указаны ни метка, ни смещение, то управление передаётся на самую первую строку программы.

При вызове меток допустимо использовать косвенность, указывая косвенно любые части полного синтаксиса меток. Пусть имя метки будет в переменной label, а имя рутины - в переменной rtn, тогда предыдущие примеры будут выглядеть так:

```
do @label^@rtn          ; 1
goto @label+3^@rtn      ; 2
g @label+4              ; 3
do ^@rtn                ; 4
w $$@label^@rtn($h,ver) ; 5
```

Кроме меток, строки рутин могут иметь также дополнительный синтаксический элемент, отсутствующий у строк команд интерактивного режима - блочный синтаксис. Если набор строк, следующих подряд, начинается на символ одной или нескольких точек, то эти строки для системы выполнения считаются локальным блоком команды, и он может быть выполнен безаргументной формой команды DO. В определенном смысле такой блок строк является вложенной неименованной подпрограммой, причем в один блок могут быть вложены другие. Пример блока:

```
label(param)
 new i
 for i=1:1:param do quit:i>5
 . write i,!
 quit
```

Пример блока с дополнительным вложением другого блока:

```
 I C="@" D
 . S (XC,YC)=0,L=L+1
 . W /ATR(1),/CUP(24,30),"Ам...",/ATR(7)
 . S n=""
 . F  S n=$O(L(n),-1) Q:n=""  D
 . . S y=$P(L(n),",",1),x=$P(L(n),",",2)
 . . W /ATR(1),/CUP(y,x),P
 E  D
 . N X,Y S N=$O(L("")) Q:N=""
 . S Y=$P(L(N),",",1),X=$P(L(N),",",2)
 . W /ATR(7),/CUP(Y,X)," "
 . K L(N),WIN(Y,X)
 Q
```

Если выполняется безаргументная форма команды DO то система выполнения начинает выполнять последующие строки команд, имеющие число точек на единицу больше, чем строка, в которой находилась команда DO. В случае если встретилась строка с числом точек меньше чем необходимо, то система выполнения считает, что блок команд закончился. Если число точек больше чем необходимо, то такая строка пропускается и система переходит к следующей строке.

Используя такое простое определение, построчный интерпретатор в данном примере, если выполнилось условие, выполняет команду DO, и последовательно переходит по строкам с одной точкой, далее выполняет, при необходимости, блок с двумя точками. После окончания выполнения подпрограммы система передает управление на строку следующую после безаргументной формы команды DO, и начинает пропускать строки с числом точек больше текущего их количества (0). В логическом отношении такая организация выполнения аналогична языкам, имеющим блоки операций, в частности { / } в языках C / C++ / Java или begin / end в Pascal.

В практическом отношении на число используемых и хранимых в базе данных рутин нет никаких ограничений, главное чтобы они имели различные имена. Существуют прикладные системы, разработанные на MUMPS, содержащие десятки тысяч рутин, составляющие единый комплекс. При этом всегда можно добавить необходимые рутины.

Общие правила и возможности MUMPS систем по работе с огромными данными на ограниченных ресурсах позволяют строить прикладные системы большого или, можно сказать, сверхбольшого объема.

Исполняемый байткод кешируется и используется по необходимости, и MUMPS системе обычно не требуется хранить в оперативной памяти весь объем кода приложения. Именно то обстоятельство, что сами рутины используются системой выполнения только по необходимости, и позволяет строить большие прикладные системы, объем программного кода которых может превосходить как объем оперативной памяти, так и размер адресуемого пространства памяти.

Для редактирования исходного текста рутин большинство современных MUMPS систем имеют либо собственные средства редактирования, либо могут быть использованы сторонние редакторы. Это могут быть графические оконные программы с MDI интерфейсом, консольные с псевдографикой включая доступ по telnet, или редакторы с WEB интерфейсом.

В случае если MUMPS система хранит исходный текст во внешних файлах, также могут быть использованы текстовые редакторы общего назначения, сохраняющие файлы в каталогах файловой системы или в каталогах FTP.

Поскольку исполняющие среды MUMPS систем являются системами позднего связывания, для компиляции рутин нет необходимости в каких-либо объявлениях, что другая рутина и тем более какая-то из меток в ней существует. В отличие от языков компилирующего типа, где необходимо связывание имен подпрограмм с предварительным описанием их прототипа, в MUMPS это не требуется. В техническом отношении можно компилировать и импортировать рутины в любом порядке. Более того, можно выполнять рутину, если она обращается к еще несуществующей рутине или метке, в этом случае исполняющая система просто сгенерирует соответствующую ошибку и управление будет передано на обработчик ошибок.

1.10 Передача параметров

При вызове метки как подпрограммы ей могут быть переданы параметры. Для языка MUMPS можно передать параметры любой метке, код вызова транслируется совершенно независимо от вызываемого кода. Будет ли на самом деле метка принимать параметры, это определяется разработчиком, описывающим метку.

Для вызывающей стороны параметры являются фактическими, для

вызываемой - формальными. Синтаксически формальные параметры следуют в круглых скобках сразу после имени метки один за другим и описаны также как локальные переменные. При вызове подпрограммы формальные параметры сопоставляются фактическим. В контексте выполнения команды, следующей после метки, эти формальные параметры для этого контекста уже являются именами локальных переменных.

При этом, что интересно, ничто не мешает передать управление на строку, непосредственно следующую после строки с меткой, но в этом случае в контексте процесса не будет производиться сопоставление с формальными аргументами, и локальные переменные с теми же именами будут иметь иное значение и область видимости, то есть будет иной контекст этих переменных.

Стандарт языка определяет следующие варианты передачи фактических параметров:

1. По значению
2. По ссылке на локальную переменную
3. Опускание параметра

При этом формальные параметры могут быть либо именами локальных переменных, либо опущены. В случае если параметр опущен, с ним ничего не сопоставляется, даже если было что-то передано. При указании имени локальной переменой с ней сопоставляется соответственно способу передачи:

1. Локальная переменная принимает переданное значение.
2. Локальная переменная становится синонимом переменной, переданной по ссылке, и указывает на нее на ее уровне стека. Все операции с такой локальной переменной автоматически заменяются на операции с переданной по ссылке переменной.
3. Локальная переменная принимает неопределенное значение.

При покидании уровня стека локальные переменные формальных параметров автоматически уничтожаются, а при передаче по ссылке все выполненные изменения с переданной переменной остаются.

Положим для примера, что имеется метка с параметрами:

```
label(a,b,c)
```

Для передачи параметра по значению нужно указать значение. Это может быть произвольное вычисляемое выражение. Перед вызовом выражение вычисляется и значение передается.

```
do label(1+2,3+4,5+6)
do label($h,$$func(789),^DATA)
```

Для передачи параметра по ссылке нужно указать точку с последующим корневым именем передаваемой локальной переменной. Передача имени с индексом не допускается.

```
new local1 s local1(1)=$h
new local2 s local1(2)=$h
new local3 s local1(3)=$h
do label(.local1,.local2,.local3)
```

Для вызова с опусканием параметра в его позици ничего не пишется:

```
do label(,,)
do label(,)
do label()
```

Если указано фактических параметров меньше чем метка принимает формальных, остальные формальные параметры также принимают неопределенное значение, как если бы было указано правильное число формальных параметров, но они были бы опущены.

Язык MUMPS отличает в фактическом параметре точку как спецификатор передачи по ссылке и точку как начало спецификации числа. Во втором случае передается параметр по значению.

Эти три способа передачи параметров являются стандартными и поддерживаются всеми современными реализациями MUMPS систем. При этом производители могут дополнительно поддерживать другие способы передачи параметров. В частности, системы MiniM и Caché поддерживают дополнительно 2 способа передачи параметров:

1. Автоматическое присваивание
2. Неопределенное количество параметров

Если используется передача параметра с автоматическим присваиванием, то можно указать, какое значение в этом случае ему необходимо автоматически присвоить. Синтаксически указывается имя формального параметра, символ присваивания = и следующая за ним константа.

Пусть есть такая подпрограмма:

```
LABEL(v=123)
 q v
```

Её можно вызвать как

```
$$LABEL()    ; 1
$$LABEL(2)   ; 2
$$LABEL(3,8) ; 3
```

Здесь в первом случае не передаётся фактический параметр и переменная v принимает неопределённое значение, которой автоматически присваивается значение 123.

Во втором случае переменной v присваивается значение 2 и значение, указанное как значение по умолчанию, игнорируется. В третьем случае переменной v значение сопоставляется, но для второго параметра метка ничего не принимает, поэтому MUMPS система в этом случае генерирует ошибку.

Для того, чтобы метка могла принять неопределённое количество параметров, её последний формальный параметр специфицируется тремя точками:

```
LABEL(p1,params...)
```

В этом случае, при сопоставлении фактических параметров формальным, игнорируется передача локальных переменных по ссылке и принимаются только значения. Значение переменной, указанной тремя точками (в примере params), принимает значение количества принятых фактических параметров, начиная с этой позиции. Принимаемые значения записываются в индексы переменной (в примере это params), начиная с индекса 1 и так далее в порядке следования. Если, начиная с этой позиции, не было фактических параметров, то формальная переменная принимает неопределённое значение. Если были пропуски фактических параметров, то соответствующие узлы формальной переменной не создаются и имеют неопределённое значение.

Например, пусть есть подпрограмма:

```
LABEL(params...)
 zw params
 q params
```

Тогда при её вызове получаем:

```
USER>w $$LABEL^test(1)
params=1
params(1)=1
1
USER>w $$LABEL^test(1,2)
params=2
params(1)=1
params(2)=2
2
USER>w $$LABEL^test("we",2)
params=2
params(1)="we"
params(2)=2
2
USER>w $$LABEL^test(,2)
params=2
params(2)=2
2
```

Разработчикам, конечно, следует учитывать, что автоматическое присваивание значений и приём неопределённого количества параметров не входят с стандарт языка MUMPS, что их использование может создать препятствие при переносе программ на другую MUMPS систему.

Для всех методов передачи параметров их правила применимы к передаче параметров подпрограмме, когда вызываемый код создает контекст, принадлежащий тому же самому процессу. В случае если параметры передаются новому процессу, вызовом команды JOB, то передать можно лишь параметры по значению. При попытке передать параметр по ссылке команда JOB генерирует ошибку. Передаваемые значения MUMPS системой передаются специальным внутренним образом новому процессу.

1.11 Неопределенные значения

В языке MUMPS локальная или глобальная переменная может иметь или не иметь присвоенное значение. Если значение не было присвоено, то такая переменная не хранится и имеет неопределенное значение. При попытке чтения ее значения процесс генерирует ошибку о неопределенном значении.

В других языках для использования переменной требуется ее объявить. В этом случае под переменную отводится определенная память. До явного ей присваивания или выполнения кода инициализации объекта значение переменной по умолчанию может быть либо произвольным (мусор в памяти), либо проинициализировано заранее нулевыми значе-

ниями или указанной константой. В этом случае поведение определяется настройками соответствующего транслятора и среды исполнения или соглашениями используемого языка. В частности, отдельные компиляторы при трансляции отладочных версий инициализируют переменные, не инициализированные явно, нулевыми байтами, но при трансляции релизных версий уже этот код не вставляется и переменная содержит мусор.

В языке MUMPS в переменной не может быть произвольного значения, поскольку до использования переменная не существует и не хранится. Обращение к переменным производится по именам и по этому имени исполняющая система определяет есть ли такая переменная и какое у нее значение. При этом до присваивания переменной не существует, и переменные создаются при первом им присваивании.

Во многих случаях разработчикам необходимо определять, существует ли переменная или нет и зачастую считать, что переменная имеет значение по умолчанию. Для этого в языке есть несколько функций, которые оперируют как определенными, так и неопределенными переменными. К ним относятся:

1. $DATA - возвращает индикатор, имеет ли переменная и ее дочерние переменные значения.

2. $GET - возвращает для переменной ее значение, или если она имеет неопределенное значение то возвращает значение по умолчанию.

3. $INCREMENT - может увеличить на указанную величину неопределенную переменную, автоматически создав ее.

4. $LISTBUILD - может использовать неопределенную переменную для конструирования элемента списка, создав элемент имеющий неопределенное значение.

5. $BITLOGIC - расширенная функция, может использовать неопределенные переменные, считая их эквивалентом последовательности битовых нулей.

Функция $DATA возвращает индикатор существует ли указанная переменная и существуют ли ее дочерние переменные, хотя бы одна. Функция является основной для проверки переменной на неопределенность. Функция возвращает одновременно два индикатора в виде двух десятичных знаков, старший для дочерних и младший для самой переменной. Их можно выделить арифметическими операциями явно.

Проверка на то, что имя существует, независимо от существования дочерних:

```
$DATA(varname)#2
```

и проверка существования дочерних, независимо от существования самой переменной:

```
$DATA(varname)\10
```

Функция $GET принимает имя переменной и возвращает ее значение, если переменная была определена. Если переменная не была определена, то функция возвращает пустую строку или указанное значение по умолчанию. Это поведение функции $GET определено стандартом и им пользуются разработчики, если необходимо использовать значение по умолчанию для переменной, если ее значение может оказаться неопределенным и это не является ошибкой.

Что интересно, существовала такая реализация MUMPS системы (StarMUMPS), в которой любая неопределенная локальная переменная имела значение пустая строка и к любой из них можно было обратиться на чтение, не получив ошибку о неопределенном значении. Можно сказать, что создание такой отдельной реализации MUMPS системы действительно могло быть оправдано если необходимо было выполнять приложения, трактующие неопределенное значение именно как пустую строку. Хотя, со стороны интерпретатора, значение пустая строка ничем особым не отличается от любого другого значения.

Аналогичное реликтовое поведение в угоду прикладным системам до сих пор встречается и в СУБД отличных от MUMPS, в частности в SQL системе Oracle не выполняется различение между данными равными пустой строке и зарезервированным значением языка NULL. Вероятно, когда-то давно это решение съэкономило время при разработке какой-то из прикладных систем, и в настоящее время этот рудимент в целях совместимости до сих пор поддерживается, хотя большинство других SQL систем не содержат такой ошибки.

В современных MUMPS системах отношение к неопределенным значениям строгое и четко регламентируется стандартом, либо строго описано для расширенных функций. В тех случаях когда MUMPS система поддерживает опциональную настройку на нестандартное поведение в отношении неопределенных значений и пустых строк, тем не менее настройки по умолчанию всегда соответствуют стандартному поведению.

Функция $INCREMENT может увеличить на указанное значение (по умолчанию на 1) в том числе и неопределенную переменную. В этом

случае функция считает, что переменная имела значение, эквивалентное 0.

Функция $LISTBUILD (расширенная, не входящая в стандарт) может принимать в качестве параметра имя неопределенной переменной, в этом случае функция конструирует элемент списка в соответствующей позиции, имеющий неопределенное значение в смысле элемента списка. В какой-то степени эта функция также может быть использована, таким образом, как альтернатива функции $DATA.

Функция $BITLOGIC может оперировать именами неопределенных переменных, в этом случае она считает что переменная эквивалентна битовой строке нулевой длины и при логических операциях условно дополняет ее нулями до длины, необходимой для операции с другой переменной. В частности, битовая операция ИЛИ с неопределенной переменной возвращает исходное значение.

Такое определение функции $BITLOGIC очень практично для выполнения массовых битовых операций над строками битовых индексов, поскольку отпадает необходимость использовать дополнительный код проверки определено ли значение битовой строки в индексе.

Что интересно, в языке MUMPS неопределенные значения можно не только проверять и использовать, но также передавать как параметры пользовательским функциям. Причем, таких способов даже два:

1. Передать по ссылке неопределенную локальную переменную.

2. Опустить фактический параметр чтобы формальный параметр принял неопределенное значение.

Если мы передает по ссылке локальную переменную, имеющую неопределенное значение, например:

```
 kill accum
 set result=$$calc(.accum)
 ...
calc(acc)
 ...
 q ret
```

то при сопоставлении фактического параметра формальному производится конструирование локальной переменной (в данном случае acc) в виде синонима другой переменной, на другом уровне (в данном случае accum). Если исходная переменная имела неопределенное значение, то и формальный параметр будет иметь также неопределенное значение.

Второй способ - это опускание соответствующего фактического параметра вместо указания ссылки на переменную или вычисляемого значения:

```
 set result=$$calc()
 ...
calc(acc)
 ...
 q ret
```

В этом случае фактическому параметру также будет сопоставлено неопределенное значение.

При этом, в языке MUMPS нельзя вычислить значение, имеющее неопределенное значение, поскольку неопределенные значения могут иметь только переменные, если они не были присвоены. Но можно сделать так, чтобы переменная получила неопределенное значение, для этого ее необходимо удалить:

```
 s var="string"
 ; переменная var имеет значение
 kill var
 ; переменная var не имеет значения
```

либо применить команду new, побочно присваивающую переменной неопределенное значение, но только начиная с текущего уровня стека:

```
 s var="string"
 ; переменная var имеет значение
 new var
 ; переменная var не имеет значения
```

1.12 Шаблоны

Шаблонами (pattern) в языке MUMPS называются правила, описывающие символы, входящие в строку, для проверки соответствует или нет строка такой последовательности.

Синтаксически оператор проверки по шаблону записывается так:

```
 expr?pattern
```

Оператор возвращает значение 1 если строка соответствует шаблону и 0 если нет, причем к оператору можно применить отрицание оператора:

```
 expr'?pattern
```

что эквивалентно отрицанию логического значения результата

```
'(expr?pattern)
```

С точки зрения синтаксиса, последовательность, описывающая шаблон оценивается транслятором MUMPS на этапе трансляции. При этом те же самые действия могут быть выполнены косвенно, если шаблон задать в виде значения вычисляемого выражения и использовать косвенность шаблонов:

```
s pattern=...
expr?@pattern
```

В этом случае оценивать шаблон MUMPS система будет на этапе выполнения кода.

Структурно шаблон представляет собой последовательность из одного или нескольких следующих подряд правил. Каждое из правил называется атомарным шаблоном. Каждый из атомарных шаблонов состоит из указания числа повторов и одной из конструкций, что именно должно повторяться:

1. Код обозначающий принадлежность символа к группе символов.
2. Строка задающая последовательность символов явно.
3. Альтернатива, задающая перечисление атомарных шаблонов и означающая операцию ИЛИ.

Рассмотрим каждое из этих определений более подробно.

Для указания числа повторов можно указывать один из следующих вариантов в виде комбинации из чисел:

count	Точно указанное воличество.
.	Любое количество, включая ноль повторов.
min .	Минимум *min* раз, максимум не ограничен.
. max	Максимум *max* раз, начиная с нуля.
min . max	От *min* до *max* раз включительно.

Задание кода или класса символов производится указанием символа, соответствующего классу символов:

A	Буквы.
C	Непечатные символы. Символы с кодами от 0 до 31 и с кодом 127.
E	Любой символ.
L	Буква в нижнем регистре.
N	Цифра.
P	Знаки пунктуации и пробел.
U	Буква в верхнем регистре.

Коды классов символов могут задаваться как в верхнем, так и в нижнем регистре.

Формально, это перечень кодов, предусмотренных стандартом языка. При этом реализации MUMPS систем могут поддерживать дополнительные коды классов символов. Для уточнения полного перечня нужно обратиться к документации на используемую MUMPS систему.

Коды классов символов могут следовать один за другим. В этом случае MUMPS система строит объединенное множество из их множеств. Например, последовательность

```
AN
```

означает множество символов букв и цифр.

Формально говоря, стандарт языка MUMPS предусматривает применение оператора отрицания к коду класса символа для получения множества, не включающего указанный класс, например

```
1.3'N   - от 1 до 3 не цифр
1.3'A   - от 1 до 3 не букв
```

но такой функционал поддерживают в настоящее время не все реализации MUMPS систем. При использовании отрицания класса символов, таким образом, необходимо проверить в документации на используемую MUMPS систему, поддерживаются ли они. В случае если не поддерживаются, но шаблон был передан в виде внешних данных и применен косвенно, то возникнет ошибка на этапе выполнения, а не компиляции.

Примеры шаблонов образованных счетчиком повторов и классами символов:

2.AN	2 или больше букв или цифр
1A.E	1 буква, после которой любое число любых символов
.'C	любое число печатных символов

Второй вариант задания символов - это задание последовательности символов (одного или нескольких) явно в виде строки. Строка должна быть указана в двойных кавычках, как обычная строковая константа языка:

1"%"4AN	Один символ процента после которого 4 буквы или цифры
1.12AN1"@mail.ru"	От 1 до 12 букв или цифр после которых 1 строка "@mail.ru"

Сопоставление явно заданным символам выполняется чувствительно к регистру.

Альтернативы задаются круглыми скобками с перечислением вариантов через запятую. Сопоставление считается успешным, если система находит соответствие для подходящего участка строки по меньшей мере одной из указанных альтернатив. Например:

1.12AN1"@"1(1"mail.ru",1"yandex.ru")	От 1 до 12 букв или цифр, косле которых 1 символ "@", после которого 1 повтор из альтернативы из 1 последовательности "mail.ru" или 1 последовательности "yandex.ru"

Нужно отметить, что в приведенном примере вместо спецификации числа повторов 1 также могут быть указаны любые числа, в этом случае проверка по шаблону будет искать повторы заданных строк подряд, одна за другой.

Альтернативы шаблонов могут быть вложены друг в друга.

Механизм сопоставления строки шаблону был разработан давно, принят в стандарте 1977-го года. В стандарт тогда входило задание классов символов и строк, в качестве числа повторов можно было указывать либо число либо точку, означающие что должно быть либо строго указанное число повторов либо любое число повторов. В стандарте 1984-го года было дополнено возможностью указывать число повторов либо от и до указанного числа, либо до, либо от указанного числа. В стандарте 1995-го года были приняты альтернативы.

Намного более сложные правила задаются регулярными выражениями. В определенном смысле, что такое регулярные выражения, задает тон язык PERL, и многие другие системы ориентируются на его возможности и правила спецификации.

В системах MUMPS полновесные регулярные выражения могут быть использованы лишь в виде внешних модулей либо если MUMPS система поддерживает встроенные системные функции для регулярных выражений, в частности MiniM поддерживает семейство функций $ZPCRE для проверки по регулярному выражению, для поиска и для замены подстрок. Но все эти возможности, несмотря на поддержку и наличие, не входят в стандарт переносимости и разработчики должны проверять как именно регулярные выражения поддерживаются на целевой системе.

Поддержка полноценных регулярных выражений в синтаксисе языка PERL в той или иной функциональной форме видится существенным улучшением с точки зрения прикладных программ. В действительности, многие данные проходят или фильтр по определенным правилам проверки и нормализации, или разбор структуры также по определенным правилам. Если стандартные шаблоны MUMPS могут лишь выполнить проверку, то регулярные выражения могут также выполнить поиск с заменой и выделение набора подходящих фрагментов. Вводить регулярные выражения в язык MUMPS на уровне синтаксиса языка в настоящее время выглядит нецелесообразным, но расширенная функциональная поддержка в стиле многих других языков видится удачной.

1.13 Косвенность

Во многих языках используются синтаксические конструкции, означающие не прямое использование значения, а использование значения как указания что именно нужно использовать на самом деле, или вместо указанного. В этом случае производится непрямое использование значения (indirection). В языке MUMPS эта операция также называется косвенностью. В те годы, когда разрабатывался язык MUMPS, было принято использовать для обозначения косвенности символ "@". Во многих программных средствах он же используется и до сих пор.

Слабым аналогом косвенности является использование указателей или ссылок, когда код использует не значение самого указателя или ссылки, а тот объект, но который они указывают. Аналогом косвенности выражения является функция eval в языке JavaScript, например.

Если при чтении программного кода разработчик видит символ @, то это означает оператор косвенности. Оператор косвенности применяется к тому, что стоит непосредственно справа от него. В языке MUMPS оператор косвенности может быть применен к большинству синтаксических конструкций. Это:

1. Косвенность имени переменной
2. Косвенность индексов
3. Косвенность метки
4. Косвенность аргумента команды
5. Косвенность шаблона

Исключением являются косвенность оператора, выражения, и косвенность команды. Вместо косвенности собственно команды используется команда XECUTE, задающая выполнение не одной, а нескольких команд. Вместо косвенности выражения используется косвенность аргумента для тех команд, которые в качестве аргумента используют выражение. Например, это команда QUIT или команда IF.

Вариант косвенного выполнения команды, или задание команды в виде отдельной строки:

```
USER>s command="kill"

...

USER>x command_" varname"
```

Здесь к аргументу varname применяется команда, указанная ранее. Но полноценной заменой косвенности команды это не является, поскольку используется команда xecute, автоматически создающая новый уровень стека и выполнение, в частности, команды new будет отменено при возврате из команды xecute.

Вариант косвенности выражения сопряжен с использованием команды quit в аргументной форме и требует отдельной метки в рутине:

```
eval(expr) q @expr
```

Такую метку можно использовать для вычисления выражений, заданных косвенно:

```
 w $$eval("2*2"),!
 w $$eval("$sy_$zd($h)"),!
```

Оператор косвенности применяется к выражению после его вычисления и выражение может быть задано с произвольной сложностью. В случае использования неатомарного выражения, например составленного из операторов, необходимо все выражение заключить в скобки, чтобы

указать что оператор применяется к результату вычисления всего выражения полностью, за исключением косвенности индексов. В случае косвенности индексов индексы добавляются именно как указано.

Стандарт, вообще говоря, допускает применение косвенности произвольного уровня вложенности. Интерпретатор на этапе выполнения будет раскрывать все встреченные им синтаксические конструкции косвенности пока все выражение не будет раскрыто полностью. Выполнение этой операции с точки зрения внутренних алгоритмов и архитектуры, а также требуемых ресурсов компьютера может представлять некоторую сложность, поэтому при использовании рекурсии в операторе косвенности не все синтаксические конструкции могут быть выполнены.

В частности, система MiniM при использовании рекурсии генерирует следующую ошибку:

```
USER>s x="@x"

USER>w @x

<MAXOBJCODE>
```

система Caché генерирует ошибку:

```
USER>s x="@x" w @x

S x="@x" W @x
         ^
<FRAMESTACK>
```

и система M3-Lite генерирует ошибку:

```
3>s x="@x" w @x
<STKOV>
```

Кроме того, отдельные реализации могут содержать ограниченные возможности по рекурсивному раскрытию косвенности, либо по сочетанию видов косвенности, либо по глубине вложенности архитектурно, либо, как было показано в примерах, косвенно из-за ограниченности выделяемых на операцию раскрытия ресурсов. В частности, приведенные выше примеры показывают, что было запрошено бесконечное рекурсивное развертывание аргумента команды, определенное через само себя. В обоих случаях системы натыкаются на израсходование внутренних пределов выполнения кода, в зависимости от внутренних архитектур и методов развертывания.

Оператор косвенности не имеет синтаксических отличий для его различных форм, форма косвенности определяется по контексту. В случае если трактовка косвенности может быть многозначной, система выбирает наиболее общий вариант. Например, если указано

```
write @string
```

то синтаксически это может соответствовать как косвенности имени для вычисления значения переменной и вывода этого значения, так и косвенности аргумента команды write.

Но случай косвенности аргумента включает в себя возможный вариант, что в аргументе может быть использовано вычисляемое выражение, или имя переменной, или перечисление нескольких допустимых аргументов команды, поэтому MUMPS система выбирает наиболее общий из этих вариантов и трактует данный случай как косвенность аргумента команды.

1.13.1 Косвенность имени

Косвенность имени означает замену оператора косвенности вместе с его значением на использование того, что указано в значении. Если выражение было вычислено как строка, содержащая допустимое имя локальной, глобальной, или структурной системной переменной, то система использует это имя вместо оператора косвенности. Например:

```
USER>s name="abcdef"

USER>w
name="abcdef"

USER>s @name=123456

USER>w
abcdef=123456
name="abcdef"

USER>
```

Здесь оператор косвенности применен к переменной name, что для команды set означает применение операции не к переменной name, а к значению переменной name. Оператор может быть применен к результату выражения, вычисленному произвольным образом. В частности:

```
USER>w

USER>f i=1:1:5 s @("aa"_i_"bb")=i*i

USER>w
aa1bb=1
aa2bb=4
aa3bb=9
aa4bb=16
aa5bb=25
i=5

USER>
```

При конструировании выражения с именем переменной код может конструировать как неиндексированное имя, как в примере выше, так и индексированное, если оно построено по синтаксическим правилам языка MUMPS:

```
USER>w

USER>f i=1:1:5 s @("aa"_i_"(i)")=i*i

USER>w
aa1(1)=1
aa2(2)=4
aa3(3)=9
aa4(4)=16
aa5(5)=25
i=5

USER>
```

Конечно, в случае использования строковых индексов разработчик должен самостоятельно принять меры, чтобы кавычки были удвоены там, где это необходимо, для получения корректного синтаксического имени.

1.13.2 Косвенность индексов

Косвенность индексов задается только после косвенности имени и означает дописывание указанных индексов к вычисленному имени переменной. Имя переменной может вычисляться как локальное или глобальное имя.

Если имя было вычислено как неиндексированное, при применении косвенности индексов образуется имя с указанными индексами. Если

было вычислено индексированное имя, то индексы добавляются после его последнего индекса.

Примеры дополнения неиндексированного имени:

```
USER>w

USER>s @("abc"_123)@(11,22,33)="value"

USER>w
abc123(11,22,33)="value"

USER>s @"name"="value2"

USER>w
abc123(11,22,33)="value"
name="value2"
```

Примеры дополнения индексированного имени:

```
USER>s @("abcd(1,2,3)")@(11,22,33)="value"

USER>s name=$na(local($j,$p))

USER>s @name@(11,22,33)="value2"

USER>w
abcd(1,2,3,11,22,33)="value"
local(2560,"|CON|",11,22,33)="value2"
name="local(2560,"|CON|")"
```

При развертывании косвенности индексов MUMPS система сначала вычисляет и выполняет все необходимые подстановки для начального имени переменной, затем вычисляет значения индексов переданные после оператора косвенности индексов и получает полное имя переменной.

Результаты операций косвенности имен и косвенности индексов могут быть использованы в любом месте, где допускается применение имени переменной.

В случае если функция, принимающая имя переменной, не допускает использование структурной системной переменной, но оно было передано косвенно, MUMPS система генерирует ошибку на этапе выполнения операции в зависимости от выполняемой функции.

Операции косвенности имен и индексов часто используются разработчиками для передачи строк в качестве аргументов функций, чтобы функции могли применять обобщенные алгоритмы к произвольным переменным. Во многих случаях разработчики применяют также хранение

реальных имен переменных в глобалах с настройками прикладной программы, с тем, чтобы программа оперировала данными в указанных ей переменных, глобалах. В этом случае администратор может направить действия программы на необходимое сочетание глобалов.

1.13.3 Косвенность метки

Оператор косвенность метки определяется по контексту, в котором должна быть использована метка. Метки используются для безусловного перехода командой GOTO, для вызова подпрограмм командой DO, для вызова функций с возвращаемым значением ($$) и как аргумент функции $TEXT. Во всех этих случаях MUMPS система будет подставлять вместо метки вычисленное значение.

В каком-то смысле слабым аналогом косвенности метки является вызов функции по указателю на функцию в других языках.

Метка состоит из четырех частей - собственно имя метки, величина смещения, имя базы данных где находится рутина и имя рутины. Синтаксически имя базы данных рутины и номер смещения являются вычисляемыми значениями, а имена метки и рутины указываются как есть. Для конструирования полного имени метки и используются различные формы косвенности метки:

1. @expr - в значении expr MUMPS система ожидает полный набор из 4-х компонент, часть из которых может быть опущена на усмотрение разработчика.

2. @expr^rouname - в значении expr MUMPS система ожидает комбинирование имени метки и смещения.

3. label+@name^rouname - косвенность для смещения не поддерживается, здесь символы @name трактуются как вычисляемое выражение и синтаксически соответствуют косвенности имени. Значение смещения берется из переменной, имя которой содержится в переменной name (в данном случае).

4. label[+offset]^@expr - в значении expr MUMPS система ожидает имя рутины с опционально заданной базой данных.

5. label[+offset]^|@name|routine - косвенность для имени базы данных не поддерживается, здесь символы @name трактуются как вычисляемое выражение и синтаксически соответствуют косвенности имени. Имя базы данных рутины берется из переменной, имя которой содержится в переменной name (в данном случае).

Разработчики могут комбинировать косвенность частей метки произвольно, если после развертывания MUMPS система может получить корректный результат.

Косвенность метки в том виде как она была определена при создании языка MUMPS, позволяет использовать аналог механизма функций высшего порядка, когда функции могут передать и принять в качестве параметров другие функции. Передать непосредственно функцию в языке MUMPS нельзя, поскольку отсутствует типизация в любом виде, но можно передать строку, которую можно использовать в качестве имени функции. Более того, со строкой можно поступить как с просто данными и провести необходимые вычисления, например проверить допустимость вызова, существует ли такая метка, добавить суффикс в зависимости от контекста работы процесса и многое другое. Например, пусть есть рутина fun в виде:

```
fun
 w $$g("f",7),!
 q
f(x)
 q x+3
g(func,x)
 q $$@(func)(x)*$$@(func)(x)
```

Здесь при вызове метки fun производится обращение к функции g, и ей передаются строки "f" и 7. При этом функция g использует один параметр для косвенного вызова функции, второй как передаваемый ей параметр. При выполнении подпрограмма fun выводит в текущее устройство значение 100.

Здесь нужно обратить внимание на использование скобок в записи:

```
$$@(func)(x)
```

Такая запись означает взятие имени метки из переменной func. В случае если скобки для func будут опущены и будет использована запись

```
$$@func(x)
```

то это означает взятие имени метки из индексированной переменной

```
func(x)
```

Одним из часто используемых применений косвенности метки является выполнение перехода в зависимости от каких-либо условий в сочетании с функцией $SELECT. Функция $SELECT проверяет условия и возвращает строку с меткой для перехода или для вызова подпрограммы. К ее результату применяется оператор косвенности и результат используется как метка. Например:

```
check(value)
 d @$s(value#2:"odd",1:"even")
 q
even
 w "subroutine for even values",!
 q
odd
 w "subroutine for odd values",!
 q
```

Здесь, в зависимости от того, является значение value четным или нечетным, вызываются различные подпрограммы.

Косвенность метки позволяет разработчикам на MUMPS, в отличие от многих других языков программирования, использовать вычисляемый GOTO, когда метка для перехода вычисляется выражением или подпрограммой произвольной сложности, включая использование настроек в глобалах, и результат в силу позднего связывания системы выполнения может указывать куда угодно. В случае если в действительности к моменту выполнения такой метки не найдено, система выполнения генерирует ошибку.

В частности, такой обобщенный механизм может использоваться в системах, выполняющих проверку прав пользователей, выполнения кода в зависимости от типа применяемой MUMPS системы, выполнения меток, передаваемых от клиентской программы, и во многих других задачах.

Косвенность метки может быть использована в любом месте, где синтаксически по контексту ожидается метка. Вычисление значения происходит перед использованием метки. В случае если само выражение также содержит операторы косвенности, то выражение продолжает развертываться до тех пор, пока не будет развернуто и вычислено полностью. После этого значение проверяется синтаксически и MUMPS система использует соответствующие указанные в значении фрагменты метки.

1.13.4 Косвенность аргумента

Косвенностью аргумента команды называется применение команды к вычисленному содержанию выражения. Сама форма косвенности в свою очередь также является допустимым аргументом, поэтому язык MUMPS допускает синтаксическую рекурсию косвенности аргументов произвольной сложности.

При применении команды к косвенно заданному аргументу сначала вычисляется выражение, и затем его значение используется как аргумент

команды. В значении могут быть указаны через запятую последовательность нескольких аргументов, в этом случае команда применяется к ним последовательно слева направо так же, как если бы они были указаны явно.

Примеры косвенного указания аргумента команды:

```
s arg="#,123,!"
w @arg
w @(^Settings(123))
s if="count,total"
i @if ...
```

Все стандартные команды языка поддерживают косвенность аргумента кроме команды FOR. Кроме того, производители MUMPS систем поддерживают дух MUMPS с возможностью задать аргумент косвенно в том числе и для расширенных дополнительных команд.

Например, многие MUMPS системы поддерживают семейство команд ZLOAD / ZSAVE / ZREMOVE, в аргументе которых указывается отдельная синтаксическая конструкция, отсутствующая в определении стандартного языка - имя рутины, соответствующее фрагменту определения метки. Например:

```
ZLOAD EXTEN2
ZSAVE VIEW5
```

Здесь идентификаторы EXTEN2 и VIEW5 MUMPS система воспринимает по контесту, как имена рутин. Чтобы передать командам имя рутины, которое содержится в переменной, нужно применить косвенность аргумента команд ZLOAD и ZSAVE:

```
s name="EXTEN2"
ZLOAD @name
s name="VIEW5"
ZSAVE @name
```

Общее правило развертывания косвенностей аргументов определено в таком виде, что является некоторым (не полным) аналогом команде XECUTE со скрытым специальным конструированием аргумента путем добавления имени команды, например некоторым аналогом может быть замена

```
w @arg   ->   x "w "_arg
s @arg   ->   x "s "_arg
```

Многие реализации MUMPS систем для снижения трудоемкости выполнения системы именно таким способом и воспользовались. В случае если применяемая MUMPS система выполнена именно так, то нужно быть очень внимательным к ее реальному поведению и обязательно проверить соответствуют ли MUMPS система задачам которые необходимо выполнить. Отличие косвенности аргумента от выполнения тех же действий командой XECUTE состоит в проверке допустимости синтаксиса как аргумента именно этой команды, в создании командой XECUTE дополнительного уровня стека и в характере диагностики в случае ошибки.

Как это проверить - нужно подставить в аргумент команды, передаваемый косвенно, кроме синтаксически корректного для этой команды аргумента, дополнительно пробел и другую команду, или несколько, с возможными аргументами. Согласно стандарта языка, MUMPS система должна отказаться выполнять такой аргумент команды, поскольку он не соответствует команде синтаксически. Во-вторых, такая передача дополнительных команд случайно выполнить недопустимый с точки зрения безопасности приложения код. И, в-третьих, даже при передаче безопасного кода система может переключиться в некорректное состояние, например изменить область видимости переменных и нарушить не данные, а ход выполнения процесса.

Кроме того, при выполнении такого синтаксически некорректного аргумента MUMPS система может дать диагностическое сообщение об ошибке, реально соответствующее лишь внутреннему состоянию эмулятора, не указывая на реальную ошибку.

Другим пунктом, который может потребовать проверки соответствуют ли применяемая система и код на MUMPS который предстоит ей выполнять, является возможность вложений косвенности. Случай с бесконечной рекурсией, очевидно, не может быть выполнен в техническом отношении в конечном итоге, независимо от архитектуры MUMPS системы, но какое-то количество уровней косвенности может понадобиться для приложения. Для проверки этого пункта нужно проверить, корректно ли MUMPS система выполняет необходимые вложения сочетаний, используемых видов косвенностей, и аргументов команды XECUTE.

1.13.5 Косвенность шаблона

Косвенность шаблона является, видимо, наиболее простой формой косвенности. В этом случае вычисляемое выражение, к которому применен оператор косвенности, трактуется как шаблон. Например:

```
USER>s pat="1E"
```

```
USER>w 1?@pat
1
USER>w 12?@pat
0
```

При этом, в отличие от метки, косвенность в шаблонах синтаксически не допускается для отдельных его фрагментов, шаблон всегда должен быть задан целиком либо по месту применения, либо целиком в виде результата вычисления выражения.

Операторы косвенности вместе с командой XECUTE образуют основное препятствие для построения MUMPS системы в виде строго компилирующей системы, работающей с кодом, точный алгоритм которого может быть определен на этапе трансляции.

При этом косвенность образует очень сильное средство для построения модулей инструментального характера, модулей с обобщенными алгоритмами, с развитыми настройками. Механизм косвенности MUMPS дает намного больше возможностей, чем другие языки, включая методы, не применимые в других языках.

Как иллюстративный пример можно привести построение систем, ориентированных на консольный алфавитно-цифровой ввод-вывод. Различные терминальные устройства могут иметь различные последовательности операций, которые необходимо выполнять для одних и тех же действий. Для таких терминалов можно вести глобал с настройками, в которых хранятся выполняемые для этих терминалов команды или аргументы команд. Для добавления еще одного типа терминала в этом случае будет достаточно добавить соответствующие ему настройки.

Как еще одну иллюстрацию можно привести применение функций с обобщенными алгоритмами, которые описывают непосредственно сам алгоритм, но с использованием косвенности могут быть применены к самым различным данным. В частности, к обобщенным алгоритмам относятся операции на индексных структурах.

1.14 Интерпретатор

Как уже было описано в разделе косвенности, язык MUMPS содержит определения синтаксических конструкций, не позволяющих выполнить систему выполнения кода по схеме традиционных компиляторов с одним только неизменяемым исполняемым кодом. Этому препятствуют и операции косвенности, и команда XECUTE.

Все имеющиеся реализации MUMPS систем в той или иной мере являются интерпретаторами. Это либо интерпретаторы интерпретирующего

типа или интерпретаторы компилирующего типа или комбинированные.

Интерпретаторы интерпретирующего типа получают на входе строку, разбирают ее и выполняют. Интерпретаторы компилирующего типа получают исходный код и транслируют в промежуточный байткод, который уже может быть исполнен виртуальной машиной интерпретатора. Интерпретаторы комбинированного типа сочетают эти две возможности, например весь код может быть транслирован в команды процессора, но при выполнении косвенности или команды XECUTE система выполнения переходит к режиму интерпретатора, или одна строка может целиком транслироваться в байткод, выполняться, но полученный промежуточный байткод не сохраняется для дальнейшего использования.

Большинство современных реализаций MUMPS систем строятся как интерпретаторы компилирующего типа, включая генерацию промежуточного нехранимого байткода для команды XECUTE. Такой режим работы обеспечивает наибольшую скорость работы в сочетании с соблюдением требований языка, или наилучший баланс возможности / производительность.

При генерации байткода разработчики различных систем используют различные собственные определения состояний виртуальной машины исполнения, коды, схему преобразований, форматы компилированных рутин.

В укрупненном виде байткод, получаемый трансляцией рутины, состоит из нескольких секций - секция заголовка, описывающая размещение других секций, секция констант, секция строк помещенных целиком, секция списка меток, секция последовательности байткодов строк и другие, в зависимости от реализации.

Основная схема работы интерпретатора строится на модели виртуальной машины - регистровая или стековая. Определение языка MUMPS наиболе близко к определению стековой машины. Для такой машины исполнения кроме обычного стека фреймов для работы подпрограмм поддерживается стек вычисления. Это внутренняя структура, недоступная для языка.

Для стека вычислений поддерживаются операции положить на стек значение, целое число, метку или имя, и взять со стека значение, целое число, метку или имя. Определение значения может варьироваться также в зависимости от реализации MUMPS системы, например это может быть структура содержащая тип значения и само значение кодированное в соответствии с используемым типом.

Для всего набора элементарных действий для виртуальной машины определяются коды операций. Например, пусть будет определена виртуальная машина с кодами операций:

Код операции	Действие
op_const	Взять следующий байт и, считая его номером константы, взять из секции констант значение и поместить его на стек.
op_plus	Взять со стека два значения, сложить и поместить результат на стек.
op_mult	Взять со стека два значения, умножить и поместить результат на стек.
op_horolog	Вычислить значение даты и времени в формате $HOROLOG и поместить значение на стек.
op_concat	Взять со стека два значения, выполнить конкатенацию и поместить результат на стек.
op_writestr	Взять со стека значение, привести к строке и вывести в текущее устройство.
op_writenl	Вывести в текущее устройство перевод строки.
op_lenght_2	Взять со стека значение, привести к строке, вычислить длину и результат поместить на стек как значение.

Используя такую виртуальную машину, можно подать ей на трансляцию и затем на выполнение уже несколько заданий:

```
1) write $h,!
2) write 12+45*78,!
3) write 123_456_789,!
```

При трансляции строки

```
write $h,!
```

получаем последовательность байткодов:

```
1) op_horolog
2) op_writestr
3) op_writenl
```

При их последовательносм выполнении сначала вычисляется значение $H, заносится на стек, затем значение вынимается со стека и выводится в текущее устройство, затем в текущее устройство выводится перевод строки.

При трансляции строки

```
write 12+45*78,!
```

получаем последовательность байткодов:

```
1) op_const
2) 1
3) op_const
4) 2
5) op_plus
6) op_const
7) 3
8) op_mult
9) op_writestr
10)op_writenl
```

и соответствующую ей таблицу констант:

```
1  12
2  45
3  78
```

При их последовательном выполнении виртуальная машина выполняет следующие действия при пустом начальном стеке вычисления:

1. op_const, 1: Берет константу номер 1 (значение по таблице = 12) и помещает на стек. Состояние стека

   ```
   12
   ```

2. op_const, 2: Берет константу номер 2 (значение по таблице = 45) и помещает на стек. Состояние стека

   ```
   45
   12
   ```

3. op_plus: Берет со стека два значения, состояние стека пусто (в данном примере записи на стеке кончились). Вычисляет сумму (получается 57) и помещает значение на стек, состояние стека

   ```
   57
   ```

4. op_const, 3: Берет константу номер 3 (значение по таблице = 78) и помещает на стек. Состояние стека

   ```
   78
   57
   ```

5. op_mult: Берет со стека два значения, состояние стека пусто (в данном примере записи на стеке кончились). Вычисляет произведение (получается 4446) и помещает значение на стек, состояние стека

```
4446
```

6. op_writestr: Берет со стека значение, состояние стека - пусто. Значение приводится к строке и выводится в текущее устройство. Состояние стека - пусто.

7. op_writenl: Выводит в текущее устройство перевод строки, состояние стека - пусто.

Третий пример транслируется соответственно в последовательность байткодов:

```
1) op_const
2) 1
3) op_const
4) 2
5) op_concat
6) op_const
7) 3
8) op_concat
9) op_writestr
10)op_writenl
```

и соответствующую таблицу констант:

```
1  123
2  456
3  789
```

Соответственно, при последовательном выполнении такого байткода со стеком вычисления производятся операции:

1. На стек помещается значение 123.

2. На стек помещается значение 456.

3. Со стека снимаются два значения, стек пуст.

4. На стек помещается значение 123456.

5. На стек помещается значение 789.

6. Со стека снимается два значения, стек пуст.

7. На стек помещается значение 123456789.

8. Со стека снимается значение.

Последнее снятое со стека значение выводится в текущее устройство, и следом выводится символ перевода строки.

Примерно по такой схеме выполняется большинство стековых виртуальных машин. Реальная виртуальная машина дополнительно поддерживает более сложные записи на стек, такие как вычисленная метка для выполнения перехода или вызова подпрограммы, и вычисленное имя для взятия значения переменной или для записи.

Реальная виртуальная машина должна будет поддерживать не операцию взять со стека значение, а взять со стека то что там есть и по нему вычислить значение. В частности, там может оказаться не само значение, а имя переменной или индикатор голой ссылки. При этом для множества команд и функций требуется передавать не значения, а именно вычисленные имена переменных, например для функции $INCREMENT, $NAME или $ORDER.

В определении языка MUMPS описаны несколько функций, которые поддерживают несколько форм, например функции $GET, $PIECE. Для них виртуальная машина должна сгенерировать код исполнения или на каждую поддерживаемую форму, или единый байткод для все форм, но для сокращенных вариантов дополнительно генерировать код вычисления значения по умолчанию.

Структурно система исполнения виртуальной машины интерпретатора состоит из цикла выборки следующего байткода, помещения на стек и взятия со стека значений, имен, констант, и набора собственно исполняющих функций, выполняющих непосредственно операции со значениями и именами, например функции сложения, конкатенации, вычисления $H, вывода значения в текущее устройство.

У каждой виртуальной машины, если она не стандартизирована, определение таких операций собственное. Существуют относительно стандартизированные виртуальные машины исполнения интерпретаторов, например виртуальные машины Java, .NET, Lua. Для таких машин возможна трансляция других языков программирования в их байткод для того чтобы эти виртуальные машины использовались для выполнения как готовые. Для систем выполнения MUMPS такие стандарты на формат исполняемых бйткодов официально не поддерживаются и построение системы исполнения полностью определяет производитель MUMPS системы.

Сам генератор байткода по исходному тексту опирается на определение языка, данное для каждой синтаксической конструкции. Рассмотрим парсинг с кодогенерацией на примере разбора выражения. Выражение в языке MUMPS определено так:

```
expr := expratom [exprtail]
```

Это означает что для парсинга выражения (expr) необходимо сначала выполнить парсинг expratom, а затем, если есть символы, то парсинг exprtail. В свою очередь, определение expratom состоит из двух альтернатив:

```
expratom :=  | glvn      |
             | expritem   |
```

Здесь glvn - определение синтаксической единицы глобальной или локальной переменной, expritem - самостоятельная синтаксическая единица.

При наличии альтернатив парсер выполняет просмотр дальнейшего текста на предмет соответствует ли дальнейший текст определению первой альтернативы. При этом может выполняться также параллельная кодогенерация для вычисления этой альтернативы. Если кодогенерация альтернативы не получилась, то парсер восстанавливает свое состояние на момент начала разбора альтернативы и переходит к следующей. И так далее, пока не кончатся альтернативы.

Если ни одна альтернатива не подошла, то парсер возвращает на предыдущий уровень парсинга состояние неудачи парсинга. Если на каком - либо уровне парсинга транслятор может принять решение о том, что отсутствие альтернатив парсинга означает синтаксическую ошибку, то может вывести сообщение о синтаксической ошибке. В большинстве случаев это касается неудачи разбора последней альтернативы.

В рассматриваемом примере определение expritem состоит из перечня более детальных альтернатив, соответствующих вычислению значений:

```
expritem := | strlit           | константная строка
            | numlit           | константное число
            | exfunc           | пользовательская функция
            | exvar             | пользовательская переменная
            | svn               | системная переменная
            | function          | системная функция
            | unaryop expratom | унарный оператор с
                                  последующим expratom
            | (expr)            | открывающая круглая скобка,
                                  последующий элемент
                                  в определении
                                  expr, с последующей
                                  закрывающей круглой скобкой
```

Здесь две последних альтернативы определены рекурсивно, и парсер сначала должен определить есть ли символ унарного оператора, запомнить его, и перейти к парсингу последующего выражения как exprатom. При его разборе считается что парсер выполняет кодогенерацию для этого элемента. Поэтому после возврата из парсинга exprатom останется добавить к байткодам байткод унарной операции.

Например, при парсинге выражения

```
+$H
```

Сначала выполняется запоминание унарной операции плюс (+), затем кодогенерация для вычисления $H, затем добавляется байткод для вычисления унарной операции плюс. При выполнении байткода система исполнения уже выполнит действия в правильной последовательности, а не так как это написано - сначала оператор плюс а потом вычисление $H.

То же правило относится к альтернативе (exprг) - эта синтаксическая конструкция может целиком заменять конструкцию exprіtem.

Определение конструкции exprtail дано так:

```
exprtail := | | binaryop   | expratom |
            | | [']truthop |          |
            | [] ? pattern           |
```

К синтаксичесим конструкциям binaryop относятся бинарные операторы, например +, -, #. К синтаксическим конструкциям truthop относятся логические операторы, например >, =, <.

Таким образом, в языке MUMPS при трансляции выражения

```
a + b * c
```

Сначала выполняется вычисление a, затем вычисление b, затем сложение, затем вычисление c, затем умножение, а при трансляции выражения

```
a + ( b * c )
```

сначала вычисляется a, затем b, затем c, затем произведение b * c, затем сложение.

Определение синтаксических конструкций языка, таким образом, не содержит приоритетов операций для операторов.

В полный набор операций виртуальной машины входит кроме вычисления имен, меток и значений, также множество дополнительных служебных операций, например при трансляции кода

```
write:a+b c+d,! set ...
```

виртуальная машина должна сначала вычислить потсусловие (a+b), а затем, если оно не выполнилось, пропустить в байткоде выполнение кода для

```
c+d,!
```

для команды write и перейти сразу к вычислению аргумента следующей за ней команды set (или его постусловия, если оно было задано).

Набор таких дополнительных служебных операций полностью определяется архитектурой виртуальной машины, каким образом она была разработана производителем MUMPS системы.

Современные интерпретаторы MUMPS систем зачастую поддерживают кроме предопределенного функционала также возможность вызова внешних модулей. Обычно это выполняется динамической загрузкой динамических библиотек. В этом случае DLL (или SO в Linux) должна реализовать определенный интерфейс для того, чтобы интерпретатор мог выполнить вызов.

При выходе последующих версий MUMPS систем традиционно принято соблюдать правила совместимости используемых форматов байткодов. Если в следующей версии MUMPS системы производятся изменения, например добавляются операторы или системные функции с соответствующими байткодами для них, то они добавляются так, чтобы предыдущие компилированные байткоды для этой версии MUMPS могли исполняться без изменений. Это называется совместимостью снизу вверх.

Современные реализации MUMPS систем, поддерживающие компиляцию исходных текстов в байткод, также поддерживают возможность отдельного экспорта и импорта компилированного байткода. Такой способ позволяет компилировать исходный текст MUMPS рутины на одном сервере и исполнять его на другом без исходных текстов.

Такие меры по переносу только компилированного байткода могут приниматься, если действительно необходимо скрыть исходный текст рутин или при сокращении времени переноса программ при большом количестве кода. В случае переноса только байткода значительно сокращается объем действий на целевом сервере, и выполняется только сохранение байткода вместо сохранения исходного текста, компиляции и все равно сохранения байткода.

Одновременно с тем, разработчики должны учитывать, что часть исходных текстов может не допускать такого переноса, если программе

требуется читать строки кода функцией $TEXT, или если при компиляции макрокода производится генерация кода рутин, зависимая от конфигурации сервера, версии MUMPS системы, или времени трансляции.

Кроме того, для многих современных интерпретаторов MUMPS систем возможно выполнение декомпилятора байткода с получением исходного текста. Возможно, исходный текст не будет в точности повторять оригинальный, например декомпилятор может не определить в каком регистре была написана команда и была ли использована полная или сокращенная форма. Как правило, от версии к версии производители могут добавлять к определению байткода дополнительные элементы и декомпиляторы, тем самым, устаревают. Но в каких-то случаях они могут быть полезны для восстановления оригинального текста.

Можно отметить, что перенос только компилированного байткода в реальной практике применяется очень редко и такой код составляется с особенным вниманием к перечню версий MUMPS системы, на которой он должен работать. Если какие-то из использованных функций поддерживаются не во всех версиях, то их обычно вызывают через команду XECUTE и при выполнении соответствующих проверок.

1.15 Голая ссылка

Голая ссылка (naked indicator) - это сокращенное обозначение обращения к последнему имени глобала, к которому было обращение. Обозначается опусканием имени глобала и добавлением индексов:

```
set ^Data(id,"root","rel","sub","color")="Синий"
set ^("figure")="Квадрат" ; use naked
set ^("count")=15
```

С точки зрения выполнения кода это полный эквивалент более многословного варианта:

```
set ^Data(id,"root","rel","sub","color")="Синий"
set ^Data(id,"root","rel","sub","figure")="Квадрат"
set ^Data(id,"root","rel","sub","count")=15
```

Само значение голой ссылки хранится как часть текущего состояния процесса. При обращении к глобалам это значение меняется на имя глобала, к которому было обращение, включая значения индексов. При этом не имеет значения контекст такого обращения, это могут быть команды присваивания, системные функции, или контекст вычисления постусловия.

При старте процесса и при смене текущей базы данных значение очищается и равно пустой строке. У каждого процесса значение голой ссылки собственное, независимое от действий других процессов.

В примере выше первая строка обращается к глобалу

```
^Data(id,"root","rel","sub","color")
```

и значение голой ссылки принимает значение этого имени.

Узнать значение голой ссылки, согласно стандарта языка напрямую нельзя. Но большинство современных реализаций поддерживает специальную расширенную системную переменную $ZREFERENCE, которая возвращает имя голой ссылки.

В рекомендации стандарта (но не в сам стандарт) входит поддержка системной переменной $REFERENCE с именно таким поведением, но многие MUMPS системы могут не поддерживать такую переменную.

Обращение к голой ссылке во многом похоже на ключевое слово WITH в языке Pascal, допускающее сокращение записи для обращения к элементам структуры или объекта, отличающиеся лишь именем поля структуры. В сокращенной записи опускаются полные операции доступа к объекту или структуре и указываются лишь последние, отличающиеся друг от друга, имена.

При доступе к глобалу значение голой ссылки может принять как индексированное, так и неиндексированное значение. При использовании голой ссылки система выполнения берет имя голой ссылки, отбрасывает последний индекс и дописывает к полученному имени указанные после символа (^) индексы.

Эта операция не может быть корректно выполнена если значение голой ссылки пустая строка или неиндексированное имя. В этом случае MUMPS система генерирует стандартную ошибку (M1).

Использование голой ссылки дает существенное преимущество при записи кода, обращающегося к глобалам, имена которых отличаются лишь последними индексами. Например, типовой код добавления записи:

```
Add(Color,Figure,Count)
 n id
 s id=$i(^DATA)
 s ^DATA(id,"Color")=Color
 s ^DATA(id,"Figure")=Figure
 s ^DATA(id,"Count")=Count
 q id
```

в реальных программах может быть намного большим. Использование голой ссылки может существенно сократить как объем кода, так и количество возможных опечаток.

При этом значение голой ссылки меняется каждый раз при доступе к глобалу и является побочным эффектом многих функций и команд. В отношении порядка взведения голой ссылки и порядка вычисления выражений и имен в книге есть отдельные статьи, их также нужно изучить подробнее. При практическом использовании голой ссылки именно возможность побочного эффекта при иных операциях может создать проблему модификации кода на MUMPS.

Например, если имеем код

```
Add(Color,Figure,Count)
 n id
 s id=$i(^DATA)
 s ^DATA(id,"Color")=Color
 s ^("Figure")=Figure
 s ^("Count")=Count
 q id
```

и необходимо добавить дополнительные действия

```
  s ^("Figure")=$$Check(Figure)
```

то после возврата из функци $$Check и перед присваиванием значение голой ссылки может измениться и запись произойдет уже не в ту глобаль, которая предполагалась до модификации кода, более того, промежуточный код может взвести значение голой ссылки в значение пустой строки или неиндексированного имени. В целом, все эти варианты приводят к ошибке выполнения кода с голой ссылкой после возврата из фунции $$Check.

По требованиям совместимости с имеющимися программами разработанными на языке MUMPS в стандарт уже нельзя ввести стекование голой ссылки. Поэтому в некоторых случаях необходимо иметь механизм восстановления значения голой ссылки в подпрограммах. Это можно выполнить стандартными средствами языка, если перед выполнением подпрограммы запомнить значение ссылки, а затем, при выходе, восстановить его. Например, если есть код:

```
Check(Figure)
 n ret
 s ret=Figure_$d(^ANYGLOBBAL)
 q ret
```

то к нему можно добавить сохранение и восстановление голой ссылки:

```
Check2(Figure)
 n saveref
 ; save naked
 s saveref=$na(^("any"))
 s ret=Figure_$d(^ANYGLOBBAL)
 ; reset naked
 i $d(@saveref)
 q ret
```

Здесь сохраняется не само имя голой ссылки, а имя с затертым значением последнего индекса. В любом случае, при использовании голой ссылки, это значение не имеет применения, поскольку будет автоматически замещаться на первый индекс указанный после символа голой ссылки.

Но в этом коде есть проблема с сохранением значения. Если значение голой ссылки было пустой строкой или неиндексированым именем то операция обращения к имени

```
s saveref=$na(^("any"))
```

генерирует ошибку.

Для компенсации такой проблемы реализации MUMPS поддерживают системную переменную $ZREFERENCE или ее аналог, которая возвращает имя голой ссылки как есть.

Многие современные реализации MUMPS систем также поддерживают присваивание этой системной переменной или ее синониму $REFERENCE. В этом случае код подпрограммы можно выполнить более устойчивым к ошибке доступа к имени голой ссылки:

```
Check3(Figure)
 n saveref
 ; save naked
 s saveref=$zr
 s ret=Figure_$d(^ANYGLOBBAL)
 ; reset naked
 s $zr=saveref
 q ret
```

Чтение и возможность присваивания системным переменным $ZREFERENCE и $REFERENCE не входят в настоящее время в стандарт и поддерживаются производителями различных MUMPS систем самостоятельно. Можно отметить, то вероятность что в используемой Вами системе поддерживается переменная $ZREFERENCE, очень высока. Присваивание таким переменным поддерживают меньшее число систем. В любом случае необходимо проверить в документации, поддерживается ли системная переменная возвращающая имя голой ссылки как есть и как именно.

1.16 Очередность выполнения

Одна из основных особенностей языка MUMPS и его отличия от других состоит в недостаточно точном понимании программистами очередности вычисления выражений и выполнения операций в языке. В других языках точно так же присутствует модель вычисления выражений и определенные соглашения, которые могут поставить программиста в тупик при смене языка. Например, выражение

```
int i = 5;
i = ++i + ++i;
```

в разных языках может дать различные результаты:

```
C++       14
cl        14
bcc       14
lcc       13
gcc       13
php       13
C#        13
```

С языком MUMPS тоже не все так просто, как кажется на первый взгляд. Но, в отличие от многих других языков, в языке MUMPS полностью отсутствует undefined behavior, или поведение на усмотрение реализации.

Очередность выполнения операций и вычисления выражений в языке MUMPS строго определены стандартом. К очередности выполнения и вычисления могут быть отнесены два основных пункта:

1. Очередность вычисления выражений.
2. Очередность вычисления имен.

И различные дополнительные

1. Очередность вычисления параметров функций.
2. Очередность применения команд к аргументам.
3. Очередность определения альтернатив в функции $SELECT.

Если дополнительным правилам вычисления всегда соответствует порядок слева - направо, то первые два нужно рассматривать детально, поскольку при их применении или, что не менее важно, при модификации, могут возникнуть затруднения.

1.16.1 Очередность вычисления выражений

Выражения в MUMPS вычисляются слева направо, приоритеты операций отсутствуют. Если нужно указать иную очередность вычисления выражения, то следует поставить скобки. Это обычно первая проблема, с которой могут столкнуться программисты при прочтении кода на языке MUMPS.

Например:

```
>w 1+2*3
9
```

Если при написании программы программист обычно при первом же тестовом прогоне обнаруживает, что он неправильно записал выражение, то при внесении исправлений в имеющийся код могут возникнуть ошибки. Типичная ситуация - пусть имеется код

```
>s a=1,b=2 i a=1 w "#",!
#
```

Со временем может возникнуть необходимость усложнить условие, например a=1 и b=2 одновременно. Обычно первое что пишет программист, это:

```
>s a=1,b=2 i a=1&b=2 w "#",!
```

И условие уже не выполняется. Потому что здесь применяется очередность вычисления выражения слева направо, и сначала выполняется сравнение a и 1, результат 1. Потом вычисляется операция & с полученным результатом (1) и значением b, получаем 1, и этот результат уже сравнивается с значением 2. Получаем 0.

Решение конфликта состоит в расстановке приоритетов явным заданием порядка вычисления скобками:

```
>s a=1,b=2 i (a=1)&(b=2) w "#",!
#
```

Теперь все работает нормально. Вывод: нужно внимательно относиться как к написанию сложных выражений, так и к их изменению. Поскольку автор исходного выражения может использовать отсутствие приоритетов для расположения операций для правильного вычисления выражения. Бояться тут нечего, надо соблюдать принятые в языке соглашения.

1.16.2 Очередность вычисления имен

Имена вычисляются слева направо, значения индексов имени вычисляются в том порядке, в котором они следуют, и вычисляются как выражения.

Например:

```
>k  s a($i(a))=$i(a) w
a=2
a(1)=2
```

Здесь видно, что сначала было вычисление значения индекса (получили 1), потом значения выражения стоящего справа (получили 2). Если нужно явно задать порядок вычисления "сначала присваиваемое значение, потом имя", то для гарантирования следует явно разнести эти операции на две:

```
>k  s tmp=$i(a) s a($i(a))=tmp w
a=2
a(2)=1
tmp=1
```

Все на первый взгляд просто, но тут тоже есть нюанс, на который необходимо обратить внимание. Это голая ссылка (naked indicator) и порядок ее взведения. При вычислении имени, образованном голой ссылкой, сначала вычисляются записанные программистом индексы, потом команда использующая имя вычисляет полное имя непосредственно перед использованием, используя текущее значение голой ссылки (naked indicator).

Например, при вычислении имени

```
^($h)
```

Сначала вычисляется выражение $h. И лишь при непосредственном использовании имени выполняется подстановка голой ссылки. Пример:

```
; очищаем экспериментальную глобаль
>k ^a

; взводим индикатор голой ссылки
>i $d(^a(1))

; проверяем чему он равен
>w $zr
^a(1)
```

```
; выполняем присваивание с использованием
; naked indicator
>s ^(123,$zr)=$d(^(2,3))

; смотрим результат
>zw ^a
^a(2,123,"^a(1)")=0
```

В этом примере четко видно, в какой последовательности реально было вычислено имя глобала для присваивания. Если бы вычисление имени выполнялось слева направо, то сначала был бы взят naked indicator (значение ^a(1)), и от него было бы образовано имя

```
^a(123,"^a(1)")
```

Но, в действительности, стандарт языка MUMPS предписывает вычислять слева направо лишь значения индексов имени, и применять голую ссылку непосредственно при использовании имени.

В нашем примере выполняется сначала вычисление индексов, и имя присваивания развертывается с значения

```
^(123,$zr)
```

до

```
^(123,"^a(1)")
```

После чего вычисляется правая часть выражения - берется голая ссылка (^a(1)) и от нее вычисляется новое имя (^a(2,3)). По этому имени берется выражение ($d() в нашем случае возвращает 0). Функция $d() меняет индикатор голой ссылки на новое значение (^a(2,3)). Команда set выполняет присваивание. При этом получает на вход имя с голой ссылкой. Голая ссылка со значения

```
^(123,"^a(1)")
```

используя текущее значение naked indicator равный

```
^a(2,3)
```

развертывается до полного имени

```
^a(2,123,"^a(1)")
```

Еще к одному нюансу очередности выполнения команды set относится ее списочная форма set (name1,name2)=expr. В этом варианте сначала вычисляются последовательно слева направо значения индексов имен name1 и name2, потом значение выражения expr, потом последовательно слева направо производится развертывание имен до полных, если там есть naked indicator, и выполняется присваивание в той очередности, в которой указаны имена.

Пример:

```
>k ^a

>i $d(^a(1))

>w $zr
^a(1)

>s (^(123,$zr),^(456,$zr))=$na(^(2,3))

>zw ^a
^a(123,456,"^a(1)")="^a(2,3)"
^a(123,"^a(1)")="^a(2,3)"
```

Здесь четко видно, что сначала было проведено развертывание первого имени, а потом второго. Отметим, что очередность выполнения списочной формы присваивания

```
set (name1,name2)=expr
```

отличается от очередности выполнения присваивания по отдельности:

```
set name1=expr,name2=expr

>k ^a

>i $d(^a(1))

>w $zr
^a(1)

>s ^(123,$zr)=$na(^(2,3)),^(456,$zr)=$na(^(2,3))

>zw ^a
^a(123,456,"^a(123,""^a(1)"")")="^a(123,2,3)"
^a(123,"^a(1)")="^a(2,3)"
```

Нужно быть внимательным при модификации кода, который использует голые ссылки. Автор кода может использовать нюансы поведения интерпретатора MUMPS с целью достижения корректного результата и запись выражения и операций может оказаться неочевидной на первый взгляд. Конечно, не стоит советовать не пользоваться нюансами языка, но к соглашениям языка следует относиться с вниманием, тем более что поведение MUMPS определено стандартом сильнее, чем соответствующими стандартами для каких-либо других языков.

1.17 Стекование $test

Системная переменная $test используется для проверки выполнилось ли действие с таймаутом или нет и для выполнения условных команд. В определении языка не предусматривается прямое присваивание этой переменной командой set, но аналогичное действие можно выполнить командой if с аргументом, в частности:

```
...
if 0
; $test = 0
...
if 1
; $test = 1
...
```

Что интересно, в отличие от других языков, в MUMPS существует безаргументная форма команды if. На первый взгляд, она ничего не проверяет, но в действительности команда проверяет значение системной переменной $test, таким образом что безаргументная форма if аналогична аргументной

```
if $test
```

Точно также команда else проверяет значение системной переменной $test и условно аналогична аргументной форме команды if

```
if '$test
```

с тем отличием, что команда if имеет побочный эффект в виде установки значения $test, а команда else его не изменяет.

Системная переменная $test была введена в стандарт давным - давно, когда в языке некоторые синтаксические конструкции и определение

отдельных деталей поведения еще отсутствовали. При этом были разработаны множество прикладных программ, для работоспособности которых было критично поддерживать совместимость реализаций MUMPS снизу вверх. Со временем, когда в языке были введены понятие стека и стекования переменных (да, когда-то их в языке не было), пришлось вводить стекование системной переменной $test так, чтобы сохранить совместимость со всем объемом наработанного программного обеспечения.

Первоначально в языке MUMPS присутствовали вызовы подпрограмм через аргументную форму DO и команду XECUTE. При их выполнении для совместимости значение $test не должно было стековаться. При этом впоследствии, при введении стекования $test, в стандарте 1990-го года, одновременно ввели безаргументную форму команды DO и вызов подпрограммы как функции с возвратом значения, одновременно с аргументной формой команды QUIT.

Начиная с этой редакции языка уже стало логично поддерживать стекование переменной $test, но только для дополнительно введенных способов вызова подпрограмм.

Механизм стекования определен таким образом, что значение стекуется если при изменении величины на некотором уровне стека и при возврате на предыдущий уровень значение восстанавливается в то значение, которое было. Это поведение также называется защитой по стеку. И величина не защищается по стеку, если при ее изменении на некотором уровне стека и при возврате на предыдущий она остается измененной.

В отношении системной переменной $test было введено более сложное определение. Значение переменной $test защищается от изменений стекованием при вызове пользовательской функции и пользовательской переменной ($$) и при вызове безаргументной формы команды do. При вызове команды xecute и команды do с аргументами значение $test не защищается стекованием и может быть изменено вызываемым кодом. Пример, демонстрирующий различие в стековании системной переменной $test:

```
run()
 n tmp
 w "argumentless do",!
 i 1 w $t,! d  w $t,!
 . i 0
 w "do with argument",!
 i 1 w $t,! d proc w $t,!
 w "xecute",!
 i 1 w $t,! x "i 0" w $t,!
 w "function",!
```

```
 i 1 w $t,! s tmp=$$func() w $t,!
 q
proc
 i 0
 q
func()
 i 0
 q 1
```

При выполнении получаем:

```
argumentless do
1
1
do with argument
1
0
xecute
1
0
function
1
1
```

Поэтому хорошим практическим правилом при комбинировании команд if + else и возможности изменения значения $test является помещение кода, который может изменить $test, в блок строк под безаргументную команду do. Например если есть код

```
 if condition xecute commands
 else  xecute other
```

то при выполнении команд из строки commands может измениться значение $test и последующая команда else будет проверять уже новое значение, а не результат проверки истинности условия condition.

В этом случае правильным вариантом будет вынос кода в блок строк с безаргументной формой do:

```
 if condition do
 . xecute commands
 else  do
 . xecute other
```

В случае же, если выполняется лишь одна команда в зависимости от условия, разработчики на MUMPS зачастую применяют лишь постусловие, например:

```
xecute:condition commands xecute:'condition other
```

В этом случае, при вычислении постусловия, значение $test не изменяется.

Само значение системной переменной $test может быть либо 1 либо 0, и это строго определено стандартом, таким образом ее значение можно использовать в арифметических и строковых операциях при вычислении выражений.

В отношении начального значения переменной $test все реализации MUMPS систем придерживаются рекомендаций стандарта - при старте процесса значение $test должно быть равно 0. При этом разработчик должен понимать, что между той точкой, когда процесс был реально запущен и точкой, когда его код получил управление, может пройти несколько команд в зависимости от настроек и реализации (возможно, и версии) MUMPS системы. В практической работе не принято проверять значение $test, если оно не было взведено собственным кодом, либо кодом существенно удаленным от проверяющего, чтобы можно было при чтении кода определить правильность и логику работы программы.

1.18 Комментарий

В языке MUMPS комментарий начинается на символ точки с запятой и продолжается до конца строки. Вся последовательность комментария, включая сам символ начала, рассматривается как пробельная последовательность и может содержать произвольные символы.

```
command  ; comment
; comment
```

В стандарте на язык определено, что перед символом точки с запятой должен быть по меньшей мере один пробельный символ. При этом этот пробельный символ может быть после безаргументной формы команды. То есть символ начала коментария не может следовать сразу после выражения или команды, хотя в языке нет такого оператора. В системе GT.M есть особенность реализации комментария - в ней разрешен комментарий с начала строки без лидирующего пробельного символа. В случае портирования такого кода разработчикам необходимо обращать внимание, поддерживается ли такая форма комментирования на целевой реализации MUMPS системы.

Что интересно, хотя это явно и не оговорено, но все реализации поддерживают правило, что перед символом комментария может не ставиться пробел, если это первый символ строки команд, указанной команде

XECUTE, или введенной интерактивно для исполнения (консоль, телнет, терминал).

Система Caché дополнительно поддерживает комментарий в виде двух слешей, в си-подобном синтаксисе. Такой комментарий не противоречит определению оператора деления, но нужно понимать что при портировании кода такой комментарий на другой системе или версии Caché может не поддерживаться.

Система Caché также дополнительно поддерживает многострочный комментарий в си-подобном синтаксисе, начинающийся на символы /* и заканчивающийся символами */. При портировании такого кода также необходимо проверить, поддерживается ли он на другой системе или версии Caché.

Для многострочных комментариев у Caché действует правило независимости от препроцессора, или независимости препроцессора от многострочного комментария, и директивы препроцессора находящиеся в многострочном комментарии, обрабатываются. В частности, такое комментирование кода может быть некорректным:

```
 ...
commands
/* comment
 ...
#define ABC ^DATA
 ...
 */
 ; здесь макрос ABC уже определен
 ...
```

Таким образом, разработчику на Caché нужно быть внимательным при многострочном комментировании больших фрагментов кода, содержащего макросы препроцессора.

В дополнение к комментарию начинающемуся на символ точки с запятой (;) системы MiniM и Caché поддерживают правило, что если комментарий начинается на два символа точки с запятой, то эта строка целиком помещается в компилированный байткод в отдельной секции. В дальнейшем, при отсутствии исходного кода, эта строка может быть возвращена функцией $TEXT при наличии лишь компилированного кода. При этом не имеет значения, есть ли перед таким комментарием команды или нет, строка возвращается целиком. Традиционно, такой комментарий используется разработчиками для помещения в код наборов данных или констант. Такого же соглашения могут придерживаться и другие реализации MUMPS систем.

Для строчно-ориентированного интерпретатора, которым является любой интерпретатор MUMPS, строка, состоящая из одного комментария, является полноценной строкой выполнения, но не содержащей ни одной команды. На такую строку, точно так же как и на любую другую, можно передать управление.

При последовательном выполнении строк управление также передается от строки к строке и, если строка целиком состоит из одного только комментария, то машина исполнения интерпретатора все равно передает этой строке управление, пусть и только для проверки что строка пуста. В случае если такие строки находятся внутри циклов, эти дополнительные операции могут немного добавить ненужного времени выполнения.

Выходом из ситуации с ненужным выполнением кода пустой строки может быть использование препроцессора, поддерживаемого системами MiniM и Caché. В их препроцессорах поддерживается макрокомментарий, начинающийся на символы #;. Если строка начинается на такой комментарий, вне зависимости от числа пробельных символов перед ним, то вся строка целиком в препроцессированном коде отсутствует. Таким образом, в исходном тексте для разработчика комментарий присутствует, но для выполняющей системы такой строки нет. Обратная сторона такого удаления строки состоит в неудобстве несовпадения отсчета смещений строк, используемых реально выполняющимся кодом, и исходного текста с макросами.

1.19 Стандарт и расширения

Язык MUMPS появился в 1966 - 1967 годах как внутренняя разработка медицинской системы Massachusetts General Hospital Utility Multi-Programming System. Само название означает вполне определенную софтверную разработку, поэтому зачастую используется альтернативное название M. Сокращение до первой буквы было выполнено в духе самого языка. Авторами языка считаются Neil Pappalardo и Octo Barnett.

В 1970-х годах на основе MUMPS была успешно разработана крупнейшая медицинская система VA Vista, и многие другие разработки на MUMPS получили заметное распространение. Возникло множество различных реализаций языка MUMPS и реализаций основанных на нем, и как следствие потребность в определенной стандартизации и договоренности между различными разработчиками. В 1977-м году появился первый стандарт на язык, стандарт ANSI, и, таким образом, язык MUMPS был стандартизирован в числе первых трех - FORTRAN, COBOL, MUMPS.

Стандартизацией языка MUMPS занимается комитет Mumps Development Committee. Это общественная организация, периодически обсуждающая изменения в языке и фиксирующая свои определения в стандарте. Всего официально выходило несколько стандартов: ANSI-1977, ANSI-1984, ANSI-1990, ANSI-1995, и в настоящее время этот последний стандарт в точности совпадает с ныне действующим стандартом ISO MUMPS.

Непосредственно в самом языке уже тогда присутствовали многие из технологий, которыми пользуются в самых различных системах - строчно-ориентированный интерпретатор, хранение данных в B-Tree, составные ключи, кеширование и подкачка, параллельная работа нескольких процессов, поддержка самых различных способов ввода-вывода в едином стиле, независимость от процессорной архитектуры, и многое другое.

Строчно-ориентированный интерпретатор языка MUMPS с командами, будучи очень сильно упрощен и усечен по возможностям, может сильно напоминать ранние варианты языка BASIC. Многое из того, что было использовано в MUMPS в качестве основных возможностей, впоследствии было использовано в качестве ключевых факторов успеха других программных систем, языков, и сред исполнения.

Стандарт языка MUMPS предусматривает несколько частей, или элементов, языка:

1. Обязательные и детализированные.
2. Обязательные и не детализированные.
3. Принципы расширения языка.

К обязательным и детализированным конструкциям языка относятся синтаксические конструкции, полностью описанные по своей структуре, значению и поведению. Например, функция $PIECE определена в точности до каждого символа и детально рассматривается каждая из ситуаций с ее аргументами. Такие синтаксические элементы должны выполняться на каждой из MUMPS систем в точности так, как это описано в стандарте, без каких-либо отклонений.

К обязательным, но не детализированным, элементам языка относятся синтаксические конструкции, для которых описано их наличие и назначение. Но детали аргументов и их значения оставлены на усмотрение реализаций. Например, команда open может иметь параметры, и стандарт описывает как их необходимо синтаксически указывать, но как именно они будут поддерживаться и что именно означают - отводится на усмотрение реализации.

Соглашения о расширении синтаксических конструкций включают соглашение о том, что любая из реализаций MUMPS системы может дополнять команды, системные переменные и функции, если команды начинаются на символ Z или z, а системные переменные и функции на символы $Z или $z. Этот признак разработчики используют как один из пунктов, на которые необходимо обратить внимание при необходимости портировать прикладную программу на другую MUMPS систему.

При этом современные реализации MUMPS систем в некоторых случаях вводят синтаксические элементы, не входящие в стандарт, но не начинающиеся на символ Z. К таким относятся функции $INCREMENT, $LISTXXX, $BITXXX, появившиеся в Caché. Введение таких функций было обусловлено практической необходимостью и было встречено большинством разработчиков прикладных программ положительно. В настоящее время различные реализации самостоятельно поддерживают такие расширенные функции для обеспечения совместимости.

Понимание того, что в языке есть стандарт и расширения стандарта, необходимо для тех разработок, которые планируется запускать на различных MUMPS системах или на различных версиях одной MUMPS системы. Если поведение стандартных элементов определено одинаково для всех систем, то поведение расширенных элементов определяется только конкретной MUMPS системой и, в принципе, может отличаться от версии к версии.

При этом, зачастую даже в расширенных языковых элементах, различные MUMPS системы могут стремиться к примерно одинаковому их функциональному поведению, чтобы облегчать применение MUMPS системы. Например, может одинаково поддерживаться функциональный смысл системной переменной $ZVERSION, чтобы разработчики могли определить по ее содержанию тип текущей системы. Или поддерживаться примерно одинаковый комплект системных утилит для экспорта и импорта рутин и данных.

Объем функционала, определяемого стандартом, весьма велик, и это определяет весьма высокую степень переносимости программного обеспечения, написанного на MUMPS, между различными MUMPS системами. Нередки случаи, когда написанный код работает без изменений десятки лет, меняются лишь MUMPS системы, операционные системы, аппаратные возможности.

В отношении расширений и дополнений стандарта различные реализации MUMPS систем в разное время вводили собственные расширенные трактовки языковых конструкций, к интересным из них, например, можно отнести такие:

1. Семейство функций $LISTXXX, первоначально введенные в Caché.
2. Глубокое присваивание функцией $PIECE введенное в StarMUMPS, когда можно выполнить

```
set $piece($piece(var,...),...)=expr
```

3. Команда new с инициализацией, введенная в MiniM, когда можно выполнить new с инициализацией по месту объявления

```
new var=expr,var2=expr2
```

Кроме стандарта на язык, есть также рекомендации комитета MDC, или так называемые предложения по элементам языка. Такие конструкции не обязательны к непосредственной реализации, но в случае если поддерживаются, то предложения описывают направление и способ их трактовки. В частности, к ним относятся

1. Команды KSUBSCRIPTS и KVALUE.
2. Левостороннее присваивание функции $QSUBSCRIPT.
3. Трактовка команды THEN.
4. Функции $DPIECE и $DEXTRACT.

Нужно отметить также, что долгое время в стандарте не было четкого определения способа обработки ошибок и MUMPS системы самостоятельно поддерживали соглашение о синтаксически относительно похожей поддержке обработки ошибок в виде обработчика ZTRAP. При этом в силу отсутствия четкого единого определения такие обработчики в различных MUMPS системах получили небольшие отличия в поведении и практичности использования.

Впоследствии в стандарт было введено единое описание обработчика ошибок ETRAP, реализованное большинством MUMPS систем. И, таким образом, в нескольких MUMPS системах до сих пор присутствует два способа обработки ошибок, причем обработчик ZTRAP может иметь в каждой из них собственные нюансы. Нюанс также состоит в том, что многие программные разработки на MUMPS также содержат обработчики ошибок типа ZTRAP.

Многие практически полезные синтаксические конструкции, первоначально разработанные и поддерживаемые самостоятельно различными реализациями, впоследствии были приняты в стандарт. В частности, к таким относятся:

1. Замена ZALLOCATE / ZDEALLOCATE на различные формы команды LOCK.

2. Замена $ZREFERENCE на $REFERENCE с предложением о возможности присваивания.

И, наконец, к важным особенностям стандарта на MUMPS можно отнести его абсолютную независимость от операционной системы, на которой выполняется MUMPS система, от типа многозадачности этой операционной системы, от процессорной архитектуры и очередности байт процессора, от числа бит и байт в одном символе и от типов и способов ввода - вывода.

При появлении и необходимости использовать новый тип ввода - вывода он добавляется в MUMPS систему наравне с имеющимися. Так, в частности, произошло в свое время с появлением сетевых протоколов на основе TCP/IP, ленточных накопителей, различного рода последовательных портов. При появлении новых операционных систем, как это произошло, например, с LINUX, MUMPS системы на них могут быть портированы зачастую в точности с тем же самым функционалом и непосредственно сами программы, написанные на MUMPS, могут этого не заметить.

При появлении новых процессорных архитектур, как это произошло, например, с 64-битными процессорами, MUMPS программы точно так же этого могут не заметить, из-за того, что выполняются в стандартизированной среде выполнения.

Одним из важных отличий стандарта можно назвать его жесткую требовательность к тому, как и в каком порядке выполняются действия, но он совершенно никак не ограничивает собственно реализации по их внутреннему устройству, по форматам представления байткода или по характеру внутреннего хранения и представления данных. Это качество позволяет создавать, с одной стороны, прикладные программы, уверенно работающие на практически любой MUMPS системе, и сами реализации систем, применяющие как новейшие процессоры, так и современные технологические и алгоритмические решения и типы ввода - вывода.

Известны даже реализации, исполнявшие MUMPS программы на компьютерах на процессоре Z80, работающие в DOS на 486-х процессорах и обслуживающие параллельно десятки клиентских подключений, работающие на редких экзотических RISC процессорах, разработанные только для определенной операционной системы или для самых разных.

Стандарту на язык удалось сохранить баланс между независимостью от прикладных систем и набором высокоуровневых операций общего на-

значения достаточно небольшого количества. Действительно, все ключевые операции языка по сути описываются примерно тремя десятками элементов, и на нем можно строить как полноценные СУБД, так и сервера приложений, при этом гибко реализуя необходимые методы работы с данными, не дожидаясь пока они будут поддерживаться в самой СУБД.

Глава 2

Глобалы

2.1 B-дерево

MUMPS системы хранят данные в так называемых глобалах. Этот термин может звучать непривычно для современного разработчика, но именно в таком виде он устоялся и используется в технической документации. В качестве синонимов глобалов используются термины глобальная переменная или глобальный массив.

Глобалы являются переменными среды исполнения, и доступны процессам по имени перед которым синтаксически ставится символ циркумфлекс, например

```
^Glo
^DATA(123)
```

Глобалы отличаются от других переменных тем, что собственно они и представляют саму базу данных в MUMPS. Данные глобалов хранятся в виде файлов на диске. MUMPS системы традиционно поддерживают от одной до нескольких баз данных, в каждой из готорых находится и хранится собственный набор глобалов. Разные базы данных могут иметь совпадающие имена глобалов, но это различные глобалы.

Хранение базы данных в файле - это обычная возможность использовать операционные системы, поддерживающие файловые системы. При этом необходимо сделать отступление для полноты описания. Формально, использование именно файлов для СУБД не является необходимостью, и некоторые MUMPS системы, как и СУБД других классов, могут использовать сырые неразмеченные разделы дисков для хранения баз данных. В этом случае СУБД самостоятельно управляет таким разделом. По сути, СУБД в любом случае содержит полный функционал

кеширования блоков и их размещения в доступном пространстве на диске, и СУБД не использует и не полагается в явном виде на, например, средства кеширования операционной системы. Вполне достаточно операций чтения и записи порции данных, причем СУБД это и так всегда делает строго определенными порциями, кратными блоку.

В отличие от простейших систем баз данных, ориентированных на применение Memory Mapped Files (файлов, отображаемых в память), полноценным СУБД для доступа к базам данных эта функция не требуется, поскольку СУБД самостоятельно поддерживают довольно сложные алгоритмы синхронизации доступа к блокам, упорядочивания их чтения и записи, не полагаясь на то, как это делает операционная система.

Вернемся к глобалам MUMPS. Если используется только имя глобала, то это означает обращение к глобалу в текущей базе данных текущего процесса. Для обращения к глобалу в определенной базе данных необходимо указать имя базы данных между символом циркумфлекса и именем глобала:

```
^|dbname|DATA(132)
^|"USER"|DATA(123)
```

Само имя базы данных может быть указано как вычисляемое выражение, главное чтобы результат выражения соответствовал имени базы данных.

Различные MUMPS системы могут использовать различную систему именования баз данных, в зависимости от реализации M-системы это может быть логическое имя, путь к каталогу, сочетание логических имен, например:

```
^|"USER"|DATA(123)
^|"D:\DB\user"|DATA(123)
^|"MGR:XXX"|DATA(123)
```

Стандарт языка MUMPS не налагает ограничений или требований на способ хранения глобалов и метод кодирования данных. Каждый из производителей выбирает формат и способ кодирования самостоятельно. Но, традиционно, в силу определения операций на глобалах, производители M-систем используют организацию данных для глобалов в виде B* дерева.

Физически дерево организуется в виде сочетания блоков ссылок и блоков данных. В базе данных предусматриваются, кроме того, блоки учета занятости, блоки каталога глобалов, блоки хранения данных превышающих размер блока и образующих цепочки, и, при необходимости, другие блоки служебного назначения.

Блок данных представляет собой чередование ключей и данных для этих ключей.

```
key1
  data1
key2
  data2
...
```

Блок ссылок представляет собой чередование ключей и ссылок на дочерние блоки.

```
  ref0
key1
  ref1
key2
  ref2
...
```

Ссылки в блоке ссылок организуются таким образом, что ссылаются на дочерние блоки (которые могут быть либо также блоками ссылок либо блоками данных) в сортировке ключей. В приведенном примере ключ key1 делит дерево глобала на две части так, что все ключи меньше чем key1 следует искать по ссылке ref0, а все ключи которые больше или равен key1 следует искать по ссылке ref1. Соответственно, ключ key2 делит множество ключей на две части, ref1 указывает на ключи меньше чем key2 и ref2 указывает на ключи больше или равный key2. И так далее для всех ключей блока.

И в блоке данных и в блоке ссылок ключи хранятся сортированно, и система управления глобалами может применять специальные алгоритмы вставки, поиска, удаления и перезаписи данных по ключам.

В блоке ссылок само значение ссылки используется не для получения значения (это делается в блоке данных), а для разделения областей поиска. Конечно, если ключ в блоке ссылок образовался путем разделения пополам блока данных, а затем этот ключ был удален, то удаляется он не из блока ссылок, а из блока данных. Поэтому в результате работы в базе данных могут образовываться в блоках ссылок такие ключи, которые не соответствуют реально существующим данным, но соответствуют правилам деления блоков на области поиска.

В блоке данных может находиться не само значение данных, а ссылка на цепочку блоков для длинного значения, не помещающегося в пределы блока. В большинстве случаев M-системы оперируют достаточно небольшими по длине данными, но при необходимости могут использовать и данные, существенно превышающие размер блока.

Одним из примеров таких цепочек является хранение данных до 32 килобайт в системах MiniM или Caché в базах с размером блока 2 или 8 килобайт, а также хранение байткода и рутин в системе MSM, где объем одного элемента может составлять сотни килобайт.

Структурно имя глобала указывается как необязательное имя базы данных, имя глобала и последовательность индексов. Полная совокупность имени и индексов, фактически, представляет собой единый ключ для одного значения. И глобал физически реализует отображение ключа на значение по этому ключу.

Одна из необязательных но распространенных особенностей хранения глобалов в современных M системах состоит в том, что глобалы преимущественно хранятся в формате с компрессией ключей. Если есть два ключа, и у них начальная часть индексов на какую-то длину совпадает, то система стремится по возможности сохранить второй ключ не полностью, а указав какая часть ключа используется от предыдущего ключа и далее хранить несовпадающую часть.

Для хранения ключей различные M системы используют разные способы кодирования и определение совпадающей части ключей может выполняться по-разному - или совпадение по целым значениям индексов или совпадение по бинарному представлению индексов. Во втором случае совпадение может прийтись на часть индекса.

Положим, что в базе данных надо хранить такие ключи:

```
^Data(123,"Volgograd",789) ...
^Data(123,"Vologda",456) ...
```

Вариант разбиения ключей для первого случая:

```
^Data(123,"Volgograd",789) ...
          "Vologda",456) ...
```

Вариант разбиения ключей для второго случая:

```
^Data(123,"Volgograd",789) ...
             "ogda",456) ...
```

Применение компрессии ключей традиционно приводит к уменьшению занимаемого глобалом места на диске без потерь данных и обычно без потерь скорости сравнения ключей. В некоторых случаях при переносе данных из табличных источников в глобалы M систем может наблюдаться заметное уменьшение занимаемого данными места, если специфика хранения данных позволяет системе использовать компрессию ключей.

Применяемые в M системах деревья традиционно управляются системами хранения так, чтобы быть более балансированными. Абсолютная балансировка хранимого дерева не выполняется, поскольку могут существовать операции, при которых для сохранения полной балансировки может понадобиться переписать блоки ссылок и данных на много мегабайт. Степень сбалансированности деревьев у современных реализаций достаточно высокая и разработчики систем выбирают компромисс между сбалансированностью дерева и временем на его балансировку.

Нужно отметить, что хотя ключи в пределах одного блока следуют в определенном сортировкой ключей порядке, сами блоки дерева уже необязательно следуют друг за другом на диске в том же самом порядке. Сортированное хранение данных относится только к хранению в пределах каждого из блоков.

Размер одного блока ограничен, каким бы он ни был. Поэтому, рано или поздно, при вставке ключа в блок системе необходимо принять решение как вставить данные. В этом случае применяется расщепление блока на две части. Примерно в середине блока выбирается ключ и выносится на более верхний блок в иерархии связей, начальная часть блока сохраняется (до ключа расщепления) а вторая часть (после ключа расщепления) записывается в новый блок. Номер нового блока запоминается с ключом расщепления в блоке более верхней иерархии. Если вставка ключа расщепления в более верхний блок также требует его расщепления, то операция повторяется уже с тем блоком.

После расщепления блока часть блока остается незанятой данными. Различного рода методы перелива данных, описанные в специализированной литературе по блансировке B* деревьев, обычно, не дают полного заполнения блока в любом произвольном случае. Поэтому практически все блоки M системы используются не полностью и существует некоторое количество неиспользованных байт. Можно отметить, что среднестатистически при хранении данных B дерева в блоках степень занятости блоков составляет больше половины общего файлового пространства.

2.2 Кодирование индексов

При хранении глобалов разработчикам M систем необходимо обеспечить одновременное выполнение следующих условий:

1. Наименьшее время сравнения ключей

2. Сортировку строк согласно национальным алфавитам

3. Сортировку чисел и строк согласно стандарту языка MUMPS

Задача уменьшения времени сравнения ключей обычно сводится к такому кодированию индексов, чтобы функция сравнения выполнялась наименьшее время. При выполнении операций с базами данных наиболее целесообразно применить специальное кодирование значений индексов в форму, наиболее удобную для сравнения. Само кодирование выполняется один раз, а сравнение много раз, поэтому достигается общий компромисс по общей производительности.

К наиболее удобной форме сравнения ключей относится простое сравнение последовательности байт (в языке C это соответствует функции memcmp). В зависимости от реализации операция сравнения может быть ориентирована как на функцию memcmp для сравнения двух последовательностей байт на указанную длину, так и на функцию strcmp, ориентированную на сравнение последовательностей байт завершающихся нулевым байтом.

В любом случае, для кодирования индексов разработчики M систем стремятся выбрать такой способ, чтобы функция сравнения кодированных ключей не требовала разбирать внутренности самого ключа в какомлибо структурировании и была максимально простой.

Проблема сортировки национальных алфавитов выражается в том, что есть алфавиты в которых кодирование символов соответствует их следованию в алфавите и есть алфавиты, в которых не соответствуют. В частности, в русском языке кодирование букв "Ё" и "Е" таково, что код буквы "Ё" меньше, чем код буквы "Е" и при обычной сортировке последовательностей байт слова с буквой "Ё" сортируются не по алфвиту.

Для решения проблемы сортировки национальных алфавитов в СУБД принято давать определение набора символов (charset) для строковых значений. В определении символов дается определение, как поднимать и опускать регистр символов при необходимости сравнения нечувствительно к регистру и в каком порядке необходимо сортировать символы.

На примере проблемы буквы "Ё" опишем одно из возможных решений. Кодирование букв "ДЕЁЖЗ" в кодировке Windows-1251 определено так:

```
Буква  Код
Д      196
Е      197
Ё      168
Ж      198
З      199
```

Мы можем дать таблицу перекодирования символов так, чтобы исходным байтам ставились в соответствие другие байты, обеспечивающие корректное следование в лексикографическом порядке, например:

```
Буква  Код  Результат
Д      196    195
Е      197    196
Ё      168    197
Ж      198    198
З      199    199
```

Здесь для символов "ЖЗ" оставлены коды, буква "Ё" получила новый код, а буквы "ДЕ" получили смещение кодов.

В действительности, конечно, используется несколько иное значение кодов, поскольку с буквой "ё" надо поступить таким же образом и сдвинуть коды также всех больших букв.

Таблица кодирования составлена так, чтобы перекодировать байты для следования в нужном порядке, и по самой задаче перекодирования для такой таблицы всегда существует обратная ей.

По прямой таблице СУБД перекодирует значения индексов для хранения и последующего сравнения, а обратную таблицу использует для получения действительного значения строки, какое было записано в стандартной входной кодировке символов. Например, если входная строка была в кодировании Windows-1251, то и выходная будет также в кодировании Windows-1251, но храниться будет в кодированном представлении.

В принципе, для СУБД не имеет значения, в каких кодировках приходят данные, главное чтобы было определено, как строки следует перекодировать для сравнения. M системы могут оперировать национальными символами в произвольной кодировке и даже в зависимости от кодирования принятого на используемой операционной системе. Обычно СУБД предоставляют способ дать определение символов для произвольного кодирования или самостоятельно поддерживают большой набор встроенных кодировок или таблиц перекодирования.

Третья проблема кодирования значений индексов состоит в том, что стандартом языка MUMPS определен порядок сравнения чисел и строк в качестве значений индексов. Несмотря на то, что в языке отсутствует декларация типа, существует вполне однозначное определение что является числовым индексом а что строковым. В случае если M система обнаруживает что значение индекса подходит под определение канонического числа, то система использует это значение как число. Общие правила сравнения значений индексов:

1. Пустая строка меньше всех
2. Числа сортируются в алгебраическом порядке как числа
3. Любые числа меньше любых непустых строк
4. Строки сортируются в алфавитном порядке

Нужно отметить, что некоторые системы поддерживают возможность указать кроме стандартной сортировки также строго лексикографическую, где числа сортируются как строки. Различие способов можно увидеть на примере:

```
Значения   Сортировка MUMPS   Лексикографическая

"0"          "-20"              "-1"
"-1"         "-1"               "-20"
"12"         "0"                "0"
"20"         "12"               "100"
"-20"        "20"               "12"
"100"        "100"              "20"
```

Возможность использовать для глобала строго лексикографическую сортировку может быть использовано в задачах, где разработчики самостоятельно применяют кодирование, необходимое для решение задачи.

Вообще говоря, проблема усложняется еще и тем, что M система должна обеспечить сортировку и дробных чисел. Поэтому, если для целых чисел можно просто запомнить число в ключе, то для дробных это нельзя делать, поскольку дробные числа в общем случае нельзя сравнивать во внутреннем представлении процессора.

Особенно сильно эта проблема проявляется в интерпретаторах, где дробные числа подвергаются арифметическим преобразованиям и преобразованиям из строки и в строку. Например, простой тест показывает первые 5 найденные чисел, для которых после преобразования в строку и обратно совпадают строковые значения, но не совпадают внутренние бинарные:

```
#include <stdio.h>
#include <stdlib.h>

#pragma argsused
int main(int argc, char* argv[])
{
  double d1;
  double d2;
```

```
  int count = 0;

  char buf[ 64];

  for( d1 = 10.0; d1 < 20.0; d1 += 0.00001)
  {
    sprintf( buf, "%.15G", d1);
    d2 = atof( buf);
    if( d1 != d2)
    {
      printf( "%.15G\n", d1);
      count++;
      if( count >= 5)
      {
        break;
      }
    }
  }

  return 0;
};
```

Первые 5 найденных чисел:

```
10.00003
10.00004
10.00005
10.00006
10.00007
```

Для решения проблемы могут быть применены различные решения про лексикографическому кодированию целых и дробных чисел и строк так, чтобы при сортировке результата как массива байт значения сортировались по стандарту MUMPS.

Как один из вариантов может быть использован вариант лидирующего байта для разделения пустых строк, чисел и строк, после которого следует либо само число, либо строка:

```
Лидирующий байт
0x00        пустая строка
иное        число
0xFF        строка
```

Для чисел выполняется приведение к виду мантисса + порядок, например:

```
Значение      Кодированное
123456        .123456 * E+6
0.0078        .78 * E-2
123.456       .123456 * E+3
```

Для отрицательных чисел используются дополнения знаков до 10 для обеспечения сортировки отрицательных чисел:

```
Значение      Кодированное
-123456        .987654 * E+6
-0.0078        .32 * E-2
-123.456       .987654 * E+3
```

Для корректности сортировки используем первые байты для записи порядка, сами байты порядка также формируем так, чтобы отрицательные числа сортировались перед положительными, используя дополнение. Сами мантиссы могут быть также представлены как в десятичной, так и в любой другой системе счисления.

Различные M системы могут использовать различные основания для предствления порядка и мантиссы чисел и различные дополнения для отрицательных. Поэтому у разных M систем может варьироваться число байт, требуемых для кодирования чисел. Отдельной интересной задачей может стать исследование оптимальных параметров для дополнений и основания при лексикографическом кодировании.

Для определения длины байтовой последовательности, сколько занимает кодированное представление индекс, используются либо байты, задающие длину, либо байты - терминаторы, как в случае системы GT.M. В качестве терминатора GT.M использует нулевой байт. Поскольку он зарезервирован, то для кодирования двух специальных байт используется кодирование в виде не одного, а двух байт:

```
0     ->     1 1
1     ->     1 2
```

Соответственно, при декодировании значений эти последовательности заменяются на обратные им.

Различные M системы и другие СУБД используют различные алгоритмы и схемы кодирования чисел для обеспечения сортировки чисел. Современные СУБД скрывают сложности кодирования и представления информации так, что результатом разработчики могут пользоваться совершенно не замечая внутренних трудностей и особенностей.

Использование кодирования национальных алфавитов имеет своими следствиями то, что разработчики должны применять при чтении данных то же самое определение символов, которое было использовано при

их записи. В случае экспорта и импорта данных в стандартных форматах СУБД используют для формирования внешних файлов кодирование данных как есть, в том виде в котором они попали в базу. В случае использования бекапов или блочного экспорта и импорта используются не отдельные логические записи, а блоки целиком. Поэтому для чтения восстановленных баз и для импорта блочного экспорта нужно использовать то же самое определение символов. Если требуется сменить определение символов, то необходимо экспортировать данные в стандартном переносимом формате и импортировать на другой системе или базе данных с другим определением символов.

2.3 Размер блока

Различные СУБД, и системы класса MUMPS в том числе, управляя данными в базе данных, оперируют блоками данных. И у каждого из блоков имеется размер. Обычно он указывается справочно в описании системы и для программистов обычно не имеет особенного значения, чему он равен.

При этом для тех, кто хочет изучить и понимать работу с базами данных более углубленно, стоит обратить внимание на этот параметр и на факторы, влияющие на производительность.

Первым пунктом, который на первый взгляд вообще не упоминается, но подразумевается и является важным, это равенство размеров блоков. Каким бы ни был выбран размер, у разных блоков он одинаковый. Вообще говоря, СУБД алгоритмически может использовать и блоки разных размеров, это вопрос техники, и даже вообще не использовать блочное строение. Но если все блоки одинакового размера, то это позволяет существенно и упростить и ускорить операции вычисления положения и адресацию нужного блока, а также использовать заранее зарезервированные пространства в кеше блоков вместо постоянного динамического распределения памяти.

Размер блока технически может быть любым, но в целях оптимизации операций ввода - вывода с дисками, физически являющимися блочными устройствами, используются размеры, кратные степени 2, обычно начиная с размера 512 байт. 512 байт - это обычно минимальный размер сектора в байтах. Впоследствии размер блоков опирали не на размер сектора, а на размер кластера, а размер кластеров у современных среднестатистических компьютеров составляет обычно 4 - 8 килобайт.

Современные СУБД используют блоки размером, конечно, не такие маленькие, как один сектор или просто кластер, а начиная с размера

хотя бы 1 килобайт. Типовые размеры блоков - 1, 2, 4, 8, 16, 32, 64 килобайт. Бóльшие размеры используются редко. Кроме фиксированного для выбора размера блоков некоторые СУБД дополнительно могут предлагать индивидуально задаваемые значения для размера блока.

Изменение размера блока имеет своими последствиями улучшение или ухудшение эффективности. Если выполняется чтение блока, а это самая тормозящая работу СУБД операция, то привод диска физически возвращает операционной системе в действительности порцию размером в один кластер, и ему нет особенной разницы, был запрошен один байт или сто, или весь кластер целиком.

Несложно провести эксперимент, определяющий производительность чтения при различных размерах читаемых порций. Пусть есть файл с размером мегабайт и программа, использующая чтение разными порциями:

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

char* filename = "test.dat";
char* openmode = "rb";
int total_size = 1024 * 1024;

int read_size[] =
{
  1,
  16,
  32,
  64,
  128,
  256,
  512,
  1024,
  2048,
  4096,
  8192,
  16384,
  32768,
  65536
};

__int64 GetCurrentMilliseconds()
{
  SYSTEMTIME system_time;
  FILETIME   file_time;
  __int64    ret;
  GetLocalTime( &system_time);
```

```
  SystemTimeToFileTime( &system_time, &file_time);
  ret = *(__int64*)&file_time;
  ret /= 10000;
  return ret;
};

#pragma argsused
int main(int argc, char* argv[])
{
  static char buffer[ 1024 * 100];
  for( int i = 0; i < sizeof( read_size) /
         sizeof( read_size[ 0]); i++)
  {
    int size = read_size[ i];
    printf( "size: %6d ", size);
    __int64 start = GetCurrentMilliseconds();

    for( int j = 0; j < 100; j++)
    {
      FILE* file = fopen( filename, openmode);
      fseek( file, 0, SEEK_SET);
      for( int pos = 0; pos < total_size; pos += size)
      {
        fread( buffer, size, 1, file);
      }
      fclose( file);
    }

    __int64 end = GetCurrentMilliseconds();

    int ms = (int)( end - start);
    printf( "time: %6d\n", ms);
  }
  return 0;
};
```

Здесь собственно чтение повторяется многократно, чтобы получались значения времени заметно больше нуля и можно было сделать оценку. При работе такого теста получается следующий отчет:

```
size:      1 time:   2856
size:     16 time:    385
size:     32 time:    307
size:     64 time:    268
size:    128 time:    252
size:    256 time:    240
size:    512 time:    220
size:   1024 time:    115
size:   2048 time:     63
```

```
size:    4096 time:      35
size:    8192 time:      23
size:   16384 time:      17
size:   32768 time:      14
size:   65536 time:      12
```

Здесь выполнялось чтение файла, размещенного в файловой системе с кластером 4096 байт. Можно проанализировать соотношение времени выполнения и порции чтения, например размеры чтений 4096 и 32768 отличаются в 8 раз, но времена выполнения для них отличаются в $2,5$ раза. При этом размеры чтений 4096 и 512 также отличаются в 8 раз, но времена чтения отличаются уже в $6,2$ раз. Соотношение показывает, что используется весьма неплохой диск и контроллер диска.

Дополнительно оценим, каково соотношение времен при шаге чтения, отличающемся в 2 раза:

```
size:      32 time:     307
                            1,1
size:      64 time:     268
                            1,1
size:     128 time:     252
                            1,0
size:     256 time:     240
                            1,0
size:     512 time:     220
                            1,1
size:    1024 time:     115
                            1,9
size:    2048 time:      63
                            1,8
size:    4096 time:      35
                            1,8
size:    8192 time:      23
                            1,5
size:   16384 time:      17
                            1,4
size:   32768 time:      14
                            1,2
size:   65536 time:      12
```

Отношения показывают, то относительная эффективность чтения в данном случае группируется вокруг размера, соответствующего размеру кластера.

Таким образом, с точки зрения эффективности чтения с диска нет смысла делить работу с базой на блоки по размеру существенно меньше или существенно больше чем кластер.

Следующим фактором для оценки ценности чтения является потенциальное кеширование. Один блок содержит целый набор порций. Даже если нам необходима только одна из них, мы читаем весь блок. При этом, поскольку блок помещается в кеше, то также в кеше оказываются остальные порции, размещенные в этом же блоке. И, если у нас высока вероятность обращения к данным, находящихся далее в сортированном порядке в том же блоке, то высока вероятность их получения уже из кешированного блока без чтения с диска. Таким образом, увеличивая размер блока, мы, не сильно ухудшая относительную эффективность физического времени доступа, улучшаем коэффициент кеширования данных, или вероятность получения следующей порции без чтения.

Другим фактором для оценки эффективности работы в зависимости от размера блока является применяемое структурирование самих данных. Если наши порции (ключи в случае MUMPS) маленькие, то для поиска нужного ключа в одном блоке надо перебрать (пусть и в сортированном виде) несколько ключей, находящихся в блоке. При увеличении размера блока мы статистически увеличиваем число ключей в блоке и время поиска в одном блоке. При этом, увеличивая число ключей в блоке, мы одновременно увеличиваем число ссылок, выдаваемых этим блоком и увеличиваем коэффициент ветвления этого блока, что очень важно для быстроветвящихся деревьев, к которым и относятся B* деревья.

Грубо говоря, имея блок большего размера, мы сильнее отсекаем область поиска среди дочерних блоков, и два соседних ключа вырезают в базе меньшее число дочерних ключей, и в идеале каждая ссылка с блока ссылок должна вести на блок данных. И обратно, если в блоке ключи большие, то их в блоке помещается меньше, и их быстрее найти при поиске в блоке, но такой блок дает меньшее отсечение области поиска. В самом крайнем и экстремальном случае, если в блоке всего один ключ и две ссылки, то такой блок делит область базы всего на 2 части, и относительная эффективность такого блока (необходимо целое чтение блока для деления всего на 2) минимальна.

Есть и обратная сторона размера блока - если он слишком маленький, то на нем может вообще не поместиться три ключа (логический минимум для делящегося дерева), и не стоит применять блоки такого размера, что в них СУБД не сможет записать что-то осмысленное.

Несложно видеть, что, в действительности, вопрос размера блока непрост. Для того, чтобы СУБД работала эффективно, надо как-то выбрать такой размер. И, действительно, крупными фирмами проводились соответствующие исследования, при каких размерах блоков данных СУБД работают более эффективно.

Общее резюме таких исследований, проводимых в том числе целенаправленно фирмой Oracle, вылилось в набор простых эмпирических правил:

1. Для применяемых в настоящее время компьютеров, жестких дисков, файловых систем и среднестатистических характеристик данных оптимум размера находится примерно между 6 и 12 килобайт.

2. Если данные по размеру стремятся к малым порциям, то размер блока оптимальнее принять 2 или 4 килобайт.

3. Если данные стремятся к увеличению порций, то размер блока оптимальнее увеличить до 16 или 32 килобайт.

4. Если данные изначально составляют очень большие порции, то размер блока оптимальнее использовать в 64 килобайт или использовать отдельные файлы не блочного формата.

В настоящее время большинство СУБД класса MUMPS используют размер блока 8 килобайт в качестве основного.

2.4 Кеширование блоков

Формально говоря, системе базы данных, работающей с цепочками блоков, по каждому обращению к глобалу требуется несколько чтений с диска. Для того, чтобы доступ к глобалам был не настолько медленным, СУБД применяют механизм кеширования.

Основной принцип оптимизации состоит в том, чтобы по возможности не делать то, что уже было сделано и возможно использовать повторно. Кеширование блоков данных позволяет не читать блоки повторно, если блок уже был прочитан и его можно использовать повторно.

Современные СУБД придерживаются одного из ключевых принципов серверного программирования - возможность работать на ограниченной памяти и никогда не выходить за административно устновленные пределы. Такое же ограничение устанавливается на кеши СУБД. Их может быть несколько. M системы поддерживают кеширование различных данных - блоков файлов баз данных, байткода рутин, каталог глобалов, и другие.

Общие принципы кеширования одинаковы для разных алгоритмов кеширования. Если база данных читает блок, то помещает его в кеш блоков. Если в кеше есть еще не занятое место, то оно занимается. Если

свободных мест в кеше нет, то механизм кеширования должен решить, какое место освободить, или вытеснить из кеша. Различные алгоритмы кеширования, собственно говоря, и различаются по стратегии вытеснения из кеша. Существуют самые различные стратегии вытеснения, есть общераспространенные и общеприменяемые, есть специализированные, учитывающие специфику доступа к кешируемым элементам, и для отдельных проектов могут быть разработаны собственные стратегии.

Для кеширования блоков B-деревьев достаточно хорошо подходит алгоритм LRU. У него есть несколько модификаций (например, K-LRU, 2S-LRU), но уже применение стандартного алгоритма LRU дает очень высокий коэффициент попадания в кеш большинства блоков дерева. Большинство современных M систем для кеширования блоков использует именно стратегию LRU или его модификацию.

Название LRU означает Least Recently Used, или "наименее недавно используемый", или вытеснение элемента к которому наиболее давно не было обращений. В этой стратегии для элементов кеша ведется очередность обращений к ним таким образом, что любой, к которому произошло обращение, помещается в начало очереди. Помещается конечно не сам элемент, а его индикатор (указатель, номер). При этом последний элемент всегда оказывается наиболее долго неиспользуемым и при необходимости вытеснения вытесняется именно этот элемент.

Стратегия LRU приводит к тому, что при двух обращениях к различным веткам глобала повторно используется некоторая часть начала дерева, или его корневая часть. При достаточно большом кеше в нем могут оказаться большинство используемых блоков. Одновременно с тем, при чтении второй ветки глобала может вообще не произойти ни одного чтения с диска, если вторая ветка хранится в уже прочитанных блоках.

32-битные системы ограничены размером используемой памяти в 2 гигабайта рабочего пространства, в котором также должны разместиться и другие области памяти, кроме кеша глобалов. В случае же использования 64-битных систем СУБД может использовать огромные объемы кеша, столько сколько может быть предоставлено аппаратно.

Администратору СУБД обычно рекомендуется использовать размер кеша, достаточный для комфортной работы сервера, но не стоит полагать, что увеличением кеша всегда можно добиться роста производительности. При увеличении кеша до какого-то предела действительно можно увеличить общую производительность доступа к глобалам, но нужно понимать, что другим программам работающим на сервере также может понадобиться память и операционная система может вытеснить часть кеша СУБД в файл подкачки, что обесценит кеширование.

Минусом кеширования является обратная сторона его плюса - если

блоки отыскиваются в кеше, то производительность сервера стабильна и относительно предсказуема, чего и добиваются применением кеширования. С другой стороны, если один или несколько процессов начнут выполнять совершенно нехарактерные для выбранной стратегии кеширования глобалов операции с блоками, то кеш вымывается или обесценивается. Или, другими словами, из кеша вытесняются блоки, ценные для большинства других процессов.

К таким характерным операциям выхода за пределы стратегии могут быть отнесены работа с большими данными в условиях недостаточно большого кеша, бекап, экспорт. При активной работе таких отдельных процессов кеш, несмотря на относительную стабильность алгоритма LRU, все же может обесцениться так, что это станет заметно для других процессов. Это проявляется в том, что общая производительность сервера при обычном выполнении большинства процессов начинает непредсказуемо падать и восстанавливаться. Для решения этой проблемы администраторы систем либо выполняют такие нестандартные операции, обесценивающие кеш, в определенное регламентное время, либо добавляют объем кеша, согласуясь с аппаратными возможностями.

Что интересно, система Caché в отношении стратегии кеширования допускает применение особого режима для процесса, так называемый пакетный режим. При переводе процесса в такой режим ему ограничивается использование кеша таким образом, чтобы он вытеснял не все блоки, а только в определенных пределах, не затрагивая бóльшую часть общего кеша. Такое решение весьма перспективно, и позволяет предоставить процессам общего назначения общую стратегию, а процессам нестандартного выполнения отдельную стратегию.

2.5 Структуры

Системы MUMPS на физическом уровне оперируют только парами вида ключ-значение. Сам ключ может быть составным, и состоять из последовательности значений индексов. Значение, записанное по этому ключу, рассматривается как последовательность байт. В зависимости от реализации M системы количество байт в значении, количество байт в значении каждого индекса и количество индексов может отличаться.

Вообще говоря, этим и исчерпывается определение хранения данных в M системах. С одной стороны, определение простое, с другой стороны, определение оставляет разработчикам огромный выбор.

Пары ключ-значение в M системах во-первых, не декларированы и, во-вторых, не типизированы. По полному ключу мы можем записать в

любое время что угодно и прочитать это позднее. Не требуется давать объявление структуры ключей и их значений до того, как выполняется сама запись в глобал.

Общее правило записи в глобал таково: если значение по этому кючу не существовало, то оно создается. Если значение существовало, то оно перезаписывается. И операция записи не различает было ли по этому ключу ранее какое-либо значение. Существование или несуществование значения перед записью по ключу используется лишь при откате транзакции для восстановления предыдущего состояния ключа.

Еще одно правило M систем состоит в том, что не требуется предварительная запись по ключу с меньшим числом индексов. Если требуется запись в глобал по ключу

```
^NAME(123,456)
```

то не требуется предварительная запись или существование данных по ключам

```
^NAME
^NAME(123)
```

Каждая пара ключ-значение хранятся независимо от того, есть ли в этой же базе данных пары с похожими ключами или нет.

Одновременно с тем M системы поддерживают три стандартные команды

```
kill varname
merge varname1=varname2
lock varname
```

Эти команды различают наличие или отсутствие у имен переменных вложенных индексов и рассматривают переменные как деревья. Если одна из этих команд применена к имени переменной, то она автоматически применяется и к остальным переменным отличающимися от указанной большим числом индексов но совпадающими с исходной переменной на число ее индексов.

Например, если есть данные в глобале

```
^DATA(1)=12
^DATA(1,"name")="Phoenix"
^DATA(1,"color")="green"
^DATA(2,"name")="Tornado"
^DATA(2,"color")="red"
```

то команда удаления

```
kill ^DATA(1)
```

удалит записи

```
^DATA(1)=12
^DATA(1,"name")="Phoenix"
^DATA(1,"color")="green"
```

но оставит записи

```
^DATA(2,"name")="Tornado"
^DATA(2,"color")="red"
```

При этом команда

```
kill ^DATA(1,"name")
```

удалит лишь запись

```
^DATA(1,"name")="Phoenix"
```

поскольку у нее более нет дочерних записей и оставит записи

```
^DATA(1)=12
^DATA(1,"color")="green"
^DATA(2,"name")="Tornado"
^DATA(2,"color")="red"
```

Теми же правилами вложения имен пользуются команды lock и merge.

При проектировании структур разработчики на M традиционно считают что в действительности оперируют не отдельными парами ключ-значение, а деревьями.

Если два имени совпадают на некоторую длину, то это вопринимается как вложенные записи, например структура записей

```
^DATA(2,"name")="Tornado"
^DATA(2,"color")="red"
```

воспринимается так: значения "name" и "color" входят в запись

```
^DATA(2)
```

И, по-другому, значения "name" и "color" существуют параллельно друг другу. Иногда это соглашение может обозначаться неполной записью имени:

```
^DATA(2) =          possible value of ^DATA(2)
       ,"name") =   name of ....
       ,"color") =  color of ...
```

Разработчики на M зачастую неявно оперируют такими соглашениями для различения понятий "ключ имеет значение", "вложенные записи" (иногда называемые структурированием по ключу), "однородные записи".

При использовании структуры вида "ключ имеет значение" традиционно понимается естественное значение ключа, например

```
^DATA("Germany")="Berlin"
^DATA("France")="Paris"
```

Особенность такого строения в том, что в качестве значений ключа используется что-то содержательное, или естественный ключ, относящийся к хранимой записи.

При использовании структуры вида "вложенная запись" традиционно понимается собственно вложение по дополнительному или нескольким дополнительным индексам, например

```
       ,"name") =   name of ....
       ,"color") =  color of ...
```

означает запись собственно интересующей разработчика части безотносительно того, какие индексы будут предшествовать указанным. Эти записи могут быть вложены в разные по своей структуре переменные, например, в

```
^DATA(1)
^DATA(2)
TMP
^TEMP($J,"select",1)
^TEMP($J,"select",2)
```

При этом для разработчика не суть важно, дочерними по отношению к чему являются эти записи, а важно что несколько записей с этими ключами существуют совместно.

При использовании структуры вида "однородные записи" применяется искусственный ключ для значения индекса, чтобы различить несколько по сути одинаково структурированных записей. В примере выше это последовательность

```
^DATA(1)
^DATA(2)
```

Собственно содержательной частью в записях является то, что записано после искусственно придуманного значения 1 или 2. Эти суррогатные ключи требуются лишь для разделения нескольких записей.

Кроме различения структуры записей по индексам, разработчики на M применяют также структурирование значений, записанных по ключу.

Само значение может рассматриваться не как одна последовательность байт, а в некоем определенном разработчиком формате. Традиционно, M разработчики используют три вида структурирования значения:

1. Позиционное ($extract())
2. С разделителями ($piece())
3. Теговое ($list())

При позиционном структурировании в значении резервируется один или несколько байт по своему положению для хранения некой информации, например пусть некое значение занимает 2 символа, тогда мы можем хранить его в зарезервированных для него позициях, например

```
USER>s $e(var,3,5)="VS"

USER>w
var="  VS"

USER>w $e(var,3,5)
VS
USER>s $e(var,3,5)="MK"

USER>w
var="  MK"
```

Позиционное хранение данных довольно редко, но вероятность его применения возрастает при уменьшении длины одной порции и увеличении вероятности неизменчивости структуры. Позиционный доступ в силу своего определения выполняется довольно быстро, но если он применен, то изменить структурирование впоследствии трудно.

При хранении данных с разделителями выбирается символ (или несколько), который используется для разделения порций записи на отдельные части. Само же значение разделителя не должно использоваться в хранимых фрагментах, например:

```
USER>s $p(var,"~",2)="Germany"

USER>s $p(var,"~",3)="Berlin"
```

```
USER>s $p(var,"~",1)="Europe"

USER>w
var="Europe~Germany~Berlin"

USER>w $p(var,"~",3)
Berlin
```

Использование разделителей позволяет хранить фрагменты переменной длины. При этом, чтобы найти заданный по номеру фрагмент записи, система должна отсчитать разделители от начала значения.

Хранение значений в формате с разделителями традиционно используется в больших системах, разработанных много лет назад. При таком способе можно использовать печатаемый символ в качестве разделителя и легко видеть что записано в различных позициях. При экспорте данных в текстовый файл его можно читать и изменять, не опасаясь за нарушение формата.

При теговом хранении данных в одном значении используется соглашение что перед порцией данных записывается длина порции либо индикатор по которому можно эту длину вычислить. Система в этом случае отсчитывает положение данных, используя теги, и разработчики могут использовать любые символы в данных, поскольку для такого формата нет зарезервированного символа.

Системы Caché и MiniM используют функции $list, разработанные изначально фирмой InterSystems для Caché. В этом формате вся последовательность байт рассматривается как последовательность отдельных позиций, у каждой из которых есть тег. Таким образом, теговые структуры можно конкатенировать.

Например:

```
USER>s $list(var,2)="Germany"

USER>s $list(var,1)="Europe"

USER>s $list(var,3)="Berlin"

USER>w
var=?Europe?Germany?Berlin"

USER>w $list(var,3)
Berlin

USER>s list=$lb("Asia")_$lb("China")
```

```
USER>w $li(list,1)
Asia
USER>w $li(list,2)
China
```

Списковые структуры нельзя изменять в экспортированных данных или иными средствами, кроме набора функций $list, из-за довольно сложного определения тега одной части списка и из-за того, что теговая часть необязательно состоит только из печатных символов.

Автору в одном из проектов довелось встретить образцы кода на MUMPS, применявшие теговое структурирование самостоятельно, средствами языка, до появления расширенных функций семейства $list. И, помнится, тогда это произвело впечатление своей простотой и элегантностью решения.

Разработчики на M традиционно используют различное структурирование данных, как по именам так и по значениям, и M системы никоим образом им в этом не препятствуют. M системы не накладывают требования на предварительное описание структуры, чтобы впоследствии проверять данные по образцу. В любое время, например, разработчик может добавить еще один или несколько фрагментов к структуре. Если использованные разработчикам соглашения об обращении к несуществующим данным корректны с точки зрения разрабатываемой прикладной системы, то добавление структур не нарушает работу системы.

Например, в вышеприведенном примере со временем жизни системы к записям

```
^DATA(1)=12
^DATA(1,"name")="Phoenix"
^DATA(1,"color")="green"
^DATA(2,"name")="Tornado"
^DATA(2,"color")="red"
```

могут добавиться записи

```
^DATA(1,"color","RGB")=...
^DATA(2,"city")="Berlin"
```

И разработчик может считать что вложенные уточнения "city" могут относиться к любой записи из набора, а вложенное уточнение "RGB" может относиться только к записям "color", ввести правило трактовки отсутствия такой дополнительной записи как значение по умолчанию или как правило чтения значения из иного источника, и так далее.

В отличие от систем жесткой типизации и структуризации, M системы допускают мягкое расширение и дополнение структур на больших

данных. В отличие от sql-based или других типизованных систем, где операция alter по добавлению колонки может переформировать гигабайты данных, в M системах это просто не требуется, для хранения данных резервирования места в масштабах большой таблицы не требуется, и данные в базу будут добавлены лишь при их реальной записи.

Точно так же M система не будет препятствовать записи в локальную переменную большой сложной структуры без предварительного декларирования ее формата. Например, если выполняется команда merge

```
merge var("select",3)=^DATA(2)
```

то M система при копировании данных из ^DATA(2) полностью воспроизведет их структуру в переменной var("select",3), в ее дочерних ключах.

Для данных из вышеиспользуемого примера получим:

```
var("select",3,"name")="Tornado"
var("select",3,"color")="red"
```

или, в укороченной записи:

```
var("select",3)
              ,"name")="Tornado"
              ,"color")="red"
```

Соответственно, если в глобале присутствовала запись

```
^DATA(2,"city")="Berlin"
```

то она также будет перенесена как есть:

```
var("select",3)
              ,"name")="Tornado"
              ,"color")="red"
              ,"city")="Berlin"
```

2.6 Индексация

Одной из наиболее важных тем для СУБД является построение индексов. Индексы - это параллельно поддерживаемые структуры данных, с использованием которых можно найти нужные данные или быстро выполнить некоторые операции, такие как проверка существования определенных значений.

В отличие от других СУБД, M системы не имеют встроенных механизмов индексации и индексные структуры для M систем ничем не

отличаются от просто данных. Разработчики самостоятельно решают, какие индексы необходимо поддерживать и в каких операциях их использовать, какой тип индекса и с какими алгоритмами применять.

M системы нисколько не препятствуют написанию прикладных систем на M таким образом, чтобы исполняющая часть автоматически могла использовать описанные разработчиками определения структур данных и индексов. С другой стороны, M системы не навязывают никаких методик, и все действия СУБД по индексации данных разработчики планируют самостоятельно.

Традиционно, при разработке прикладной системы на M, разработчики либо используют уже готовые собственные наработки и библиотеки, либо прорабатывают один раз необходимый механизм и далее используют его.

Бóльшая часть индексных структур соответствует так называемым обратным, или инвертированным, спискам. Принцип их формирования в целом простой - если по идентификатору записи можно определить значение атрибута записи, то по записи в инвертированном списке по значению атрибута можно определить идентификатор записи. Это грубая формулировка в первом приближении, более развернуто индексация данных описана в главе "Индексация данных".

Если обобщенную запись обозначить структурой

```
^DATA(id)=attr1~attr2~attr3
```

то инвертированный список может быть или

```
^INDEX(attrN)=id
```

или

```
^INDEX(attrN,id)=""
```

Традиционно разработчики на M используют второй вариант как более общий и универсальный.

Соответственно, для поддержания индекса описываются функции, соответствующие операциям изменения индексируемых атрибутов - создание, перезапись, удаление.

Общий принцип поддержки индекса состоит в следующем. При создании новой записи добавляется еще одна запись в индексную структуру, при изменении записи удаляется индексная запись используя предыдущее значение атрибута и добавляется новая индексная запись для нового значения атрибута, и при удалении записи удаляется индексная запись, используя текущее значение атрибута.

Алгоритмически для разработчиков это обычно выглядит как написание соответствующего количества функций и обращение впоследствии к ним.

При этом, если разработчики видят повторяющиеся структуры, то, естественно, могут использовать один код, который автоматически определяет имя глобала для хранения данных и индексов.

Например, если есть несколько справочников имеющих одинаковую структуру, то разработчики могут просто добавить функциям изменения одного справочника один параметр, идентифицирующий справочник. Этот параметр может использоваться на усмотрение разработчиками самым различным образом. Например, пусть есть два справочника City и Country с записями, имеющими один атрибут "Name" и исходные структуры

```
^City(id)=Name
^Country(id)=Name
```

с индексными структурами

```
^IndexCity(Name,id)=""
^IndexCountry(Name,id)=""
```

и с функциями добавления данных

```
AddCity(Name)
 n id s id=$i(^City)
 s ^City(id)=Name
 s ^IndexCity(Name,id)=""
 q:$q id
 q
AddCountry(Name)
 n id s id=$i(^Country)
 s ^Country(id)=Name
 s ^IndexCountry(Name,id)=""
 q:$q id
 q
```

Для исключения дублирования кода разработчик может добавить параметр в обобщенную функцию

```
AddDict(Dict,Name)
 n id s id=$i(@("^"_Dict))
 s @("^"_Dict)@(id)=Name
 s @("^Index"_Dict)@(Name)=id
 q:$q id
 q
```

В этом случае при вызовах

```
d AddDict("City","Berlin")
d AddDict("Country","Germany")
```

будут создаваться соответствующие записи в справочниках с индексными записями для справочников City и Country. Конечно, эта же функция может быть использована для остальных справочников той же структуры.

На усмотрение разработчиков могут быть использованы различные методы, например, использование имени справочника в качестве значения индекса структур, хранение перечня индексируемых полей в отдельном определении (метаданные) и многое другое.

Нужно понимать, что для M систем нет отдельного выделения записей по их назначению - будут это оригинальные данные, служебные индексы или временные структуры для преобразований. M система будет выполнять ровно те операции и ровно таким образом, как их описал разработчик.

2.7 Группировка

Одна из наиболее востребованных практических методик после структурирования и индексирования данных - это операция группировки. В отличие от других СУБД, M системы позволяют ее выполнять максимально точно и в соответствии с решаемой задачей.

Операция группировки в базах SQL-типа объявляется опцией GROUP BY и зачастую сопровождается опцией ORDER BY. Те, кто имел дело с языком SQL, наверняка примерно представляют, что это такое и к чему приводит. Те же, кто не использует язык SQL, имеют, с одной стороны отсутствие простого декларативного объявления своих намерений, и, с другой стороны, ничем не ограничены. Рассмотрим виды группирования на языке M и их особенности, плюсы и минусы.

Группировка в общих словах - это операция выборки данных в таком виде, в котором значения колонок рассматриваются в качестве критерия объединения строк - строки с одинаковыми значениями в группирующих колонках объединяются в одну строку или один ключ, в зависимости от решаемой задачи.

Положим, что у нас есть набор исходных данных, на котором мы можем провести демонстрацию. В качестве примера выберем условную

задачу "учет новогодних елочных игрушек". Положим, что в нашем распоряжении есть несколько партий новогодних игрушек, которые мы различаем по фигуре, по цвету и в каждой партии есть некоторое количество одинаковых игрушек.

Создадим тестовые данные скриптом вида:

```
create(n)
 s:'$d(n) n=100
 s:(n<1) n=-n
 k ^group
 n i,color,colors,figure,figures,count
 s colors="красный~золотой~синий~зеленый~серебряный~желтый"
 s figures="шарик~шишка~снежинка~белка~лебедь~рыбка"
 f i=1:1:n d
 . s color=$p(colors,"~",$r(6)+1)
 . s figure=$p(figures,"~",$r(6)+1)
 . s count=$r(10)+1
 . s ^group(i)=color_"~"_figure_"~"_count
 q
```

Здесь i - это некий условный номер партии. При группировании по полям цвет и фигура часть строк с их одинаковыми значениями объединяются в одну строку: если были строки

```
красный шарик 10
красный шарик  8
синий   шарик  5
синий   шарик 15
```

то при группировании мы должны получить

```
красный шарик 10
                8
синий   шарик  5
              15
```

То есть из четырех исходных получили две выходные, причем в выходных строках в одну ячейку попали от одного до нескольких значений (количество игрушек в партии). Формально говоря, мы можем сделать с ними что хотим, но поскольку речь ведем о группировке, то в группировке принято из этих нескольких значений, попадающих в одну ячейку, составлять одно значение и приводить, таким образом, выходные данные к классическому определению таблицы с атомарными значениями в каждой ячейке.

Характер манипулирования такими наборами значений, попадающих в один ключ группирования, с целью получения лишь одного значения,

называется функцией группирования. Наиболее часто встречаются самые простейшие - вроде банального сложения в столбик или вычисления их количества.

В более сложных случаях значения, попавшие в одну ячейку, могут быть отсортированы по дополнительному выбранному критерию, например, по дате получения партии, из которой было взято это значение и в совокупности с датой получения партии может быть получена например средняя скорость поступления таких изделий.

Вообще говоря, эти функции группирования составляют совершенно отдельный интереснейший для прикладных специалистов класс задач, находящийся на стыке задач класса OLAP и Data Mining. В этой статье мы опустим их разнообразие и будем пользоваться только простейшей функцией - сложение в столбик, которой в SQL соответствует агрегирующая функция SUM. В приведенном выше примере запрос на SQL выглядел бы примерно как

```
select color, figure, SUM(count)
from NewYearToys
group by color, figure
```

Обычно совместно с группировкой используется операция сортировки. О ней мы скажем отдельно позднее. Будем считать, что общетеоретические сведения о группировке, приведенные выше, должны оказаться достаточными для ее технической реализации.

Итак, группировка может быть классифицирована по типу выборки данных и по ширине группировки. По типу выборки данных группировка делится на группировку с ориентацией на выборку с помощью функции $ORDER и с ориентацией на выборку с помощью функции $QUERY. По ширине группировки деление идет на нормальную и широкую.

Будем использовать данные, сгенерированные в вышеприведенном скрипте, и рассмотрим, как именно технически выполнить группирование. Обратим внимание на структуру выходных данных и заметим, что сочетание группирующих полей для каждой строки образует уникальное значение. Следовательно, в MUMPS базах данных это сочетание должно стоять слева от знака равенства:

```
переменная( группирующее поле 1 ...
            группирующее поле 2 ...
            группирующее поле N ) = набор негруппирующих полей
```

Здесь под таинственными символами (...) и обозначены различия типов группировки - в случае использования функции $ORDER используем конкатенацию значений группирующих полей, в случае использования функции $QUERY используем обычные запятые, рассматривая

значения группирующих полей в качестве значений индексов соответствующего уровня.

Выполним группировку для функции $QUERY:

```
GroupQ()
 ; group to use $QUERY function
 ; use SUM function
 k group
 n i,color,figure,count
 s i=""
 f  s i=$o(^group(i)) q:i=""  d
 . s color=$p(^group(i),"~",1)
 . s figure=$p(^group(i),"~",2)
 . s count=$p(^group(i),"~",3)
 . s group(color,figure)=count+$G(group(color,figure),0)
 q
```

Здесь выполняется проход по исходным данным, для наглядности значения полей сохраняются в отдельных переменных, после чего выполняется сложение. Функция $GET используется для случая, если это сложение выполняется первый раз. Вместо сложения можем использовать любую иную функцию группирования, но в нашем примере будем пользоваться для простоты только одним сложением в столбик. После того, как данные сгруппированы в виде

```
group(color,figure)=SUM(count)
```

мы можем их получить одним проходом с помощью функции $QUERY:

```
WriteGroupedQ()
 d QroupQ()
 n color,figure,count,cf
 s cf="group"
 f  s cf=$Q(@cf) q:cf=""  d
 . s color=$qs(cf,1)
 . s figure=$qs(cf,2)
 . s count=@cf
 . w color,?15,figure,?30,count,!
 q
```

Здесь значения отдельных полей из группирующего ключа получаются с помощью функции $QSUBSCRIPT. В случае использования этого типа группировки мы можем использовать несколько полей группирования и все равно сможем получить результат одним проходом. В целях создания более-менее формализованной обобщенной функции мы можем

использовать номера в аргументах $QS. Если их получать из формальной спецификации запроса, то нет необходимости организовывать вложенные циклы прохода по уровням индексов.

Рассмотрим парный вышеприведенному метод группирования, ориентированный на использование функции $ORDER:

```
GroupO()
 ; group to use $ORDER function
 ; use SUM function
 k group
 n i,color,figure,count
 s i=""
 f  s i=$O(^group(i)) q:i=""  d
 . s color=$p(^group(i),"~",1)
 . s figure=$p(^group(i),"~",2)
 . s count=$p(^group(i),"~",3)
 . s group(color_$C(10)_figure)=
    count+$G(group(color_$C(10)_figure),0)
 q
```

Здесь результат получается в виде переменной с одним значением индекса, в котором с помощью разделителей используется символ $C(10). Для получения результата группировки можем использовать также только один проход, но с использованием функции $ORDER:

```
WriteGroupedO()
 d GroupO()
 n color,figure,count,cf
 s cf=""
 f  s cf=$O(group(cf)) q:cf=""  d
 . s color=$P(cf,$C(10),1)
 . s figure=$P(cf,$C(10),2)
 . s count=group(cf)
 . w color,?15,figure,?30,count,!
 q
```

Здесь мы также можем составить обобщенную функцию группировки, если получим номера полей из формального запроса и подставим их в аргумент функции $PIECE. В обоих типах группировки в правой части может стоять не одно значение негруппирующего (агрегирующего) поля, а несколько. Их можно хранить как в формате с разделителями, так и в списочном виде. В приведенном примере использовалось только одно негруппирующее поле, поэтому в случае если их несколько, код следует соответственно подправить.

Отметим плюсы и минусы обоих методов группирования. В первом случае (ориентация на $QUERY) результат выдается в отсортированном

виде, и порядок сортировки определяется индексным порядком сортировки. Каких-либо дополнительных пересортировок уже не требуется. При этом следует помнить, что операция $QS может занять больше времени, чем $P во втором случае. К тому же обязательно следует скорректировать код для случая получения в качестве значения поля пустой строки. Например, всегда дополнять строку пробелом при группировании и удаления этого пробела при выдаче результата. Во втором случае, вообще говоря, отсортированность результата не гарантируется и определяется выбранным символом - разделителем. Если он меньше пробела, то результат будет отсортирован. И, так же как в первом случае, следует дополнять индексное значение неким символом на случай получения группировки только по одному полю и при возможности получения в качестве значения поля пустой строки.

В случае использования групировки, ориентированной на функцию $ORDER, результат, конечно, будет неким образом отсортирован, но результат врядли будет удовлетворительным, поскольку будет применяться индексная сортировка к агрегату полей, которые скорее всего сортируются строковой сортировкой.

В случае использования нестроковых (числовых) значений полей следует приводить их значения к строкам таким образом, чтобы сортировка проводилась в правильном порядке, соответствующем типу данных. Например, в случае использования целых чисел их следует заменять примерно как: число 123 заменяем на строку "+00000123". То есть во-первых добавляем символ знака, во-вторых, дополняем нулями до некоторой выбранной длины. В случае использования дробных чисел ситуация усложняется - следует в строку вносить символ знака числа, десятичный символ, дробную часть, знак и величину порядка. Причем расположить эти части следует в порядке, обеспечивающем именно строковую сортировку. После проведения группировки с такой сортировкой в функции визуализации также следует провести соответствующую коррекцию данных, чтобы убрать нагромождение дополняющих нулей.

Впрочем, в ситуации с особой трудоемкостью дополнений полей с целью совмещения группировки с сортировкой ничто не мешает выполнить сортировку в виде операции, отдельной от группирования. Об этом тоже не следует забывать - сортировка как отдельная операция может понадобиться в ситуации, когда следует выполнить сортировку по негруппирующим полям.

Рассмотрим другое деление группировки - на нормальную и широкую. Проблемой, породившей такое деление, является ограниченность длины значения индекса. В нашем случае это существенно, поскольку в индексные значения пишутся значения полей. Каким бы ни было ма-

гическое число этого ограничения, в целях эффективности реализации СУБД в каждой реализации оно есть. В отдельных реализациях размер индекса совпадает с величиной группировки, в других это две разные величины, но в любом случае предполагаются ограничения на максимальную величину индекса и группирующих полей.

Вообще говоря, в большинстве случаев несложного применения и несложного анализа двух вышеприведенных способов группирования вполне хватает. Поэтому оба они называются нормальной группировкой, поскольку в обоих случаях слева от символа равенства стоят именно значения группирующих полей.

Но если все хорошо работает, то программисты этим, как правило, не занимаются, и нас более интересует случай, когда не все хорошо работает. Или, в случае с группировкой, стоит вопрос - как провести группирование в ситуации, когда величина группирующих полей не уместилась в ограничение индекса.

В этой ситуации помогает условная замена значений полей на соответствующие этим значениям числовые идентификаторы. Скажем, цвету красный сопоставляется число 1, цвету синий - 2 и так далее, после чего в группировании принимают участие не длинные поля типа названия организации, а короткие числа.

Эти промежуточные идентификаторы значений должны быть числами, для которых можно задать, во-первых, взаимно однозначное соответствие между значением и числом и, во-вторых, на наборе чисел должен быть определен порядок, соответствующий порядку значений полей.

Для этого выполняем два прохода. В первом получаем список значений группирующих полей, попавших в выборку, во втором проводим собственно группировку. При выдаче результата используем отображение числовых значений на значения полей. Примерный код получения списка значений:

```
WideGroup()
 k group,map
 n i,color,figure,count
 s i=""
 f  s i=$O(^group(i)) q:i=""  d
 . s color=$p(^group(i),"~",1)
 . s figure=$p(^group(i),"~",2)
 . ; save colors and figures into special lists
 . s map("color",color)=""
 . s map("figure",figure)=""
```

После этого в локальной переменной map содержатся два списка с цветами и фигурами. Отметим, что до полного прохода по результатам выборки данных, попавших на группировку (в нашем случае это

```
$O(group(i))
```

мы просто не можем построить сортированного списка числовых идентификаторов значений, поскольку данные приходят в заведомо несортированном виде.

После получения списков значений группирующих полей можем построить отображение на соответствующие числовые значения:

```
 s color=""
 f  s color=$O(map("color",color)) q:color=""  d
 . ; map color to ordered number
 . s map("color",color)=$I(map("color"))
 . ; map ordered number to color
 . s map("Ncolor",map("color",color))=color

 s figure=""
 f  s figure=$O(map("figure",figure)) q:figure=""  d
 . ; map figure to ordered number
 . s map("figure",figure)=$I(map("figure"))
 . ; map ordered number to figure
 . s map("Nfigure",map("figure",figure))=figure
```

После этого в локальной переменной map имеем отображение значений цветов и фигур на числа, причем числа благодаря использованию индексной сортировки в

```
$O(map("color",color))
и
$O(map("figure",figure))
```

упорядочены в том же порядке. После этого, используя отображения значений на числа, можем провести широкую группировку:

```
 n ncolor,nfigure
 s i=""
 f  s i=$O(^group(i)) q:i=""  d
 . s color=$p(^group(i),"~",1)
 . s figure=$p(^group(i),"~",2)
 . s ncolor=map("color",color)
 . s nfigure=map("figure",figure)
 . s count=$p(^group(i),"~",3)
 . s group(ncolor,nfigure)=
     count+$G(group(ncolor,nfigure),0)
 q
```

Объединив образцы кода вместе, получим функцию, которая выполняет широкую группировку. Отметим, что никакой оптимизации здесь не приводилось, а получение данных, попадающих на группирование, не всегда такая простая операция, как просто проход по глобали. И, чтобы не выполнять ее дважды, имеет смысл в реальном коде сохранить выборку во временной глобали. И использовать глобали для группирования и отображения группирующих значений на числа, поскольку данных может оказаться столь много, что они просто не поместятся в области данных процесса.

При выводе сгруппированных данных следует, конечно же, помнить, что группировали мы не значения, а их номера, поэтому используем построенное ранее отображение:

```
WriteWideGrouped()
 d WideGroup()
 n color,ncolor,figure,nfigure,count,cf
 s cf="group"
 f  s cf=$Q(@cf) q:cf=""  d
 . s ncolor=$qs(cf,1)
 . s nfigure=$qs(cf,2)
 . s count=@cf
 . s color=map("Ncolor",ncolor)
 . s figure=map("Nfigure",nfigure)
 . w color,?15,figure,?30,count,!
 q
```

Сложно говорить о группировке и не затронуть группировку с подытогами. Например, получение той же группировки, но в которую вставлены данные отдельно по цветам безотносительно фигур игрушек, а также общая величина. Ничего сложного в этом нет. Конечно же, следует использовать тот же механизм группирования, но для каждой строки писать суммирование не только со строкой, идентифицируемой группой полей, но и идентифицируемой специальным маркером подитога вместо группирующего поля. Например, выбрав в качестве маркера подитога символ $C(11), получим группирование с подытогами по цвету:

```
 ; group to use $QUERY function
 ; use SUM function
 k group
 n i,color,figure,count
 s i=""
 f  s i=$o(^group(i)) q:i=""  d
 . s color=$p(^group(i),"~",1)
 . s figure=$p(^group(i),"~",2)
 . s count=$p(^group(i),"~",3)
```

```
. s group(color,figure)=
    count+$G(group(color,figure),0)
. s group(color,$C(11))=
   count+$G(group(color,$C(11)),0)
q
```

Здесь в переменной

```
group(color,figure)
```

накапливается группировка по цвету и фигуре, а в переменной

```
s group(color,$C(11))
```

накапливается группировка по цвету, она же является подитогом по фигуре.

Получение подитогов при группировке представляется весьма занятным, поскольку мы можем скомбинировать те же действия в более общем виде:

```
. s group(color,figure)=
    count+$G(group(color,figure),0)
. s group(color,$C(11))=
   count+$G(group(color,$C(11)),0)
. s group($C(11),figure)=
    count+$G(group($C(11),figure),0)
. s group($C(11),$C(11))=
    count+$G(group($C(11),$C(11)),0)
```

и получить группировку с комбинациями подитогов и общего итога.

В приведенном варианте вместо одинаковых предопределенных тегов $C(11) могут быть использованы различные символы, чтобы отделить просто значения от подитогов и итогов.

В целом, можно отметить, что разработчики на MUMPS никаким образом не ограничены в средствах и функционале для построения прикладных систем. При планировании уровня системы, будет это визуализация, прикладная логика, общесистемные алгоритмы или нижний уровень хранения данных, разработчик может выбирать, где и как выполнить отдельные операции, при этом оставаясь в едином согласованном контексте.

2.8 Каноничность индексов

Для корректной разработки программных систем на MUMPS очень важным моментом является понимание того, что именно M система использует в качестве индекса и что именно программа передает M системе в качестве такого значения.

С точки зрения человека или правил, принятых в прикладной системе, несколько физически различных значений (строк) могут означать логически одно и то же. Для M системы это не так. Вот пример, когда физически различные значения означают одно и то же с точки зрения человека:

```
USER>s a("100")=123

USER>s a("10e1")=456

USER>s a("1e2")=789

USER>w
a(100)=123
a("10e1")=456
a("1e2")=789
```

Здесь все три значения индексов означают различную запись числа 100, но для M системы числом в индексном значении является только одно из них. Это происходит потому, что M системы используют определенное правило каноничности чисел. Если строка подходит под определение канонического числа, то значение используется как число и к нему применяется правило сортировки как числа. Иначе значение считается строкой и к нему применяется правило сортировки строк.

Перечень правил, которыми пользуются M системы для определения каноничности строкового представления числа:

1. Строка не является каноническим числом, если содержит лидирующие нули и не является нулем.

2. Строка не является каноническим числом если терминируется символом ноль и он не является частью порядка и число не является нулем.

3. Строка не является каноническим числом, если оканчивается символом десятичной точки, после которой не следует ни одной цифры.

4. Строка не является каноническим числом, если содержит лидирующий знак "+" или более чем один знак "-" или знак "-" с одним или более знаками "+" или знак "-" после которого следует лидирующий ноль.

5. Строка не является каноническим числом, если содержит иные буквы или цифры, не формирующие экспоненциальную форму числа.

6. Экспоненциальная форма числа не является канонической, если число после представления в виде строки не совпадает с исходной.

7. Значение подходит под определение канонического числа, если задано числовой а не строковой константой или вычислено арифметически.

Пропустив через такое небольшое сито определений, M система относит строку либо к числам, либо оставляет строкой.

Вот несколько примеров, демонстрирующих различную запись единицы, но представляющие различные с точки зрения M систем индексные значения:

```
USER>s a("1.0")="1.0"

USER>s a("01.")="01.0"

USER>s a("01.0")="01.0"

USER>s a("1.")="1."

USER>s a("1")="1"

USER>s a("01")="01"

USER>s a("-1")="-1"

USER>s a("+1")="+1"

USER>w
a(-1)="-1"
a(1)="1"
a("+1")="+1"
a("01")="01"
a("01.")="01.0"
a("01.0")="01.0"
a("1.")="1."
a("1.0")="1.0"
```

Здесь под определение канонически заданного числа подошли только две записи:

```
s a("-1")="-1"
s a("1")="1"
```

Отметим, что большинство M систем при выводе имен переменных отмечает кавычками те индексы, которые она используют как строковые, и без кавычек те, которые использует как числа.

Приведем также пример, показывающий каноничность экспоненциальной формы:

```
USER>s a("1e100")="1e100"

USER>s a("1E+100")="1E+100"

USER>w
a(1E+100)="1E+100"
a("1e100")="1e100"
```

Здесь каноничной формой M системы считают только второй вариант, с явным указанием знака показателя.

Для выяснения того, является ли строка каноническим представлением числа, разработчики на M используют оператор унарного плюс: если применение унарного плюс к строке дает ту же строку, то строка представляет собой каноническое число:

```
USER>w 1E+100
1E+100
USER>w +"1E+100"
1E+100
USER>w +"1E100"
1E+100
```

Поэтому, для того чтобы гарантированно использовать в качестве значений индексов именно числа, разработчики используют при индексации унарный плюс. В этом случае M система приводит строку к числу по правилам приведения и используется результат. Зачастую, пользователи при вводе значений могут набрать числа, соответствующие допустимой записи с точки зрения человека, но не каноничное с точки зрения M системы, и унарный плюс канонизирует полученное значение.

Другой особенностью, которую следует учитывать, является возможность формирования логически эквивалентных, но физически различных строк. Есть функции, которые могут дать логически эквивалентные но

физически различные результаты в зависимости от особенностей внутреннего формата кодирования.

К функциям, формирующим физически различные значения, относятся функции семейства $list. Если элемент списка получен в виде числовой константы либо был вычислен арифметически то функции формируют числовой элемент, иначе строковый. Например:

```
USER>s list=$lb(123.456)

USER>w $list(list,1)
123.456
USER>s list=$lb("123.456")

USER>w $list(list,1)
123.456
USER>w $lb(123.456)=$lb("123.456")
0
```

Здесь элементы списка в одних случаях числовые, в других строковые, но во внутреннем кодировании списков формируются различные последовательности байт.

Автору приходилось сталкиваться с использованием списковых структур в качестве составных значений индексов в большой программной системе. Система прекрасно работала до тех пор, пока значения в список попадали, только будучи вычисленными как числа. Как только системе были переданы строковые значения, тут же произошла ошибка. В качестве исправления ошибки был выбран отказ от использования списковой структуры в индексе в пользу разделителей. Хотя автор и не уверен, что такое решение проблемы может быть единственным. Нормализация (или канонизация) числовых значений там, где известно, что они должны быть числовыми, а также принудительное приведение чисел к строкам конкатенацией с пустой строкой там, где должны быть строки, также могло бы быть решением.

К функциям, формирующим физически различные последовательности байт для логически эквивалентных значений, также относятся функции $bit, поскольку логический результат проверки бита определяется не только тем, был ли он записан в строку, но и тем правилом, что незаписанные биты рассматриваются как нулевые. Например, различное формирование битовых строк из нулей:

```
USER>s $bit(bits1,100)=1

USER>s $bit(bits1,100)=0
```

```
USER>s $bit(bits2,10000)=1

USER>s $bit(bits2,10000)=0

USER>w bits1=bits2
0
```

К общим рекомендациям для разработчиков на M можно добавить рекомендацию не использовать форматы кодирования индексных значений, если они не дают канонического представления, иначе работоспособность кода будет зависеть от определенного прикладной системой и комплексом тестов способа попадания данных в систему.

2.9 Маппинг

Маппинг глобалов или отображение глобалов - это механизм замены обращения к глобалу на физическое обращение к глобалу в определенной базе данных или группе файлов (томов).

Маппинг может выполняться на уровне имен глобалов и на уровне индексов. Физическая трансляция имени глобала в другую базу данных может выполняться в глобал с тем же именем, с другим именем, с другими индексами.

Традиционно, маппинг используется разработчиками, как минимум, для так называемых системных и временных глобалов. Большинство современых ралиазаций поддерживает соглашение о так называемых системных рутинах и системных глобалах. В действительности, это обычные глобалы, но для них поддерживается специальное соглашение об отображении на системную базу данных. Традиционно, системные глобалы и системные рутины первым символом имени имеют символ процент (%).

Общепринятое соглашение позволяет различным разработчикам понимать друг друга с первого символа имени. Если имя системное, то, из какой бы текущей базы данных к ним ни обратились, физически они (глобалы и рутины) располагаются в одной единственной системной базе данных. Такое соглашение приводит к тому, что системные рутины и глобалы доступны всем процессам в единственном экземпляре, процессы из различных баз данных могут обмениваться данными, и использовать единые для системы рутины общего назначения.

Таким же правилам подчиняются так называемые временные глобалы. Для них общеиспользуемого соглашения о формировании имени нет, но принцип тот же - они отображаются в специальную базу данных, к

которой применены облегченные настройки записи и журналирования. Временные данные, оставшиеся от прошлого сеанса работы сервера, при его старте могут быть не только удалены, но и база может быть пересоздана. Для использования временных глобалов и формирования имени такого глобала необходимо обратиться к соответствующей части документации на используемую M систему.

Маппинг в зависимости от применяемой M системы может быть предопределенным (как в MiniM), так и полностью настраиваемым (как в Caché). Если в MiniM понятие текущей области и текущей базы данных совпадают, то в Caché это отдельные понятия. В Caché процессы логически обращаются к области, но область как таковая существует лишь как набор правил отображения глобалов на системную, временную и обычно специально для этой области созданную базу данных.

Caché позволяет организовать маппинг весьма сложно и использовать множество физически различных баз данных. Нередко применяется конфигурация области, состоящая из отображения части глобалов на системную базу, на временную, на базу для глобалов содержащих данные, и на базу для рутин и специальных справочных данных. При этом база данных с рутинами может передаваться целиком от разработчиков в эксплуатацию, минуя процесс импорта.

При создании новой области средства Caché учитывают собственные соглашения об отображении глобалов и рутин по умолчанию и автоматически создают их для новой области, но эти настройки всегда можно изменить.

Пример маппинга на уровне имени:

```
^%SRV
^%WM
```

Здесь процентное имя глобала полностью отображается в системную базу данных.

Пример маппинга на уровне индексов:

```
^ROUTINE("%RI")
^ROUTINE("%RO")
^ROUTINE("%BACKUP")
```

Здесь первый символ имени рутины - это процент (%), поэтому ветка глобала для хранения рутин ^ROUTINE физически отображается в системную базу данных. Такое же соглашение о маппинге на уровне индексов применяется к глобалам хранящим макрорутины и компилированный байткод.

Практически любая многопользовательская СУБД поддерживает в той или иной форме понятие функций и данных общего назначения, или системную базу данных. В системах MUMPS это выполняется по первому символу, или специальным соглашением для временных глобалов.

Разумеется, у разработчика есть возможность всегда обратиться к рутинам или глобалам хранящимся в другой базе данных, задав их место хранения явно, и указав базу данных. Например

```
^|"%SYS"|COMMON(...)
^|"TEMP"|SELECT($J,...)
```

В этом случае M система обращается к глобалу в указанной базе, но к используемому имени также применяется правило отображения. Например, имена

```
^|"USER"|%COMMON
^|"TEMP"|%COMMON
```

приводят к обращению к базам USER и TEMP, но для них также применяются соглашения об отображении системного имени и физически используются глобалы в области %SYS. Такое соглашение позволяет не нарушить правила отображения вне зависимости от того, как были специфицированы имена глобала или рутины.

В зависимости от реализации M системы, если она поддерживает журналирование, различные базы данных могут иметь различные настройки журналирования. Разработчикам необходимо учитывать правила настройки журналирования и маппинга для корректной работы приложений. Например, чтобы не возникла ситуация что при откате транзакций одни глобалы были восстановлены в начальное значение, а другие нет, но приложение использует их взаимозависимо, что и приводит к ошибке.

Глава 3

Индексация данных

3.1 Общие принципы

В этой главе речь пойдет об алгоритмах и структурах данных для индексов, их организации, поддержке и применении.

Термин индекс далее используется строго в целях обозначения дополнительных поисковых или оптимизирующих структур. Основным языком примеров выбрано стандартное подмножество языка MUMPS. Но, хотя по возможности применяется страндартный синтаксис, в некоторых исключительных случаях для большей читаемости применяются Cache Object Script и MiniM Database Server - расширения. Их применение ограничено и допускает альтернативную замену на эквивалентные выражения в иных диалектах MUMPS. Применение битмап индексов ограничено теми MUMPS системами, которые поддерживают расширенные $BIT функции.

Индексы - это структуры данных, размещаемые параллельно и поддерживаемые синхронно основным структурам данных и имеющие основным назначением поддержание структур данных, ориентированных на ускорение поиска или оптимизацию хранения основных данных. Здесь под основными данными понимаются данные, хранение и работа с которыми является основным назначением системы базы данных.

При использовании основных данных система базы данных выполняет операции вставки, поиска, удаления и изменения в общем массиве их хранения. При использовании дополнительных индексных структур система параллельно обновляет индексные структуры при изменении (вставке, изменении и удалении) основных данных и в некоторых случаях получает возможность использовать индексные структуры, ориентированные на поиск данных. Наличие такой возможности определяется

характеристиками и структурой индекса.

Как следует из вышеприведенного, введение индексов в систему базы данных утяжеляет операции, связанные с изменением данных, но ускоряет операции связанные с поиском и, как обычно, в следствии этого, с выборкой данных.

Индексные структуры сами по себе обычно не являются необходимыми для основной работы системы базы данных. И их применение определяется программистом или администратором системы.

В большинстве общераспространенных систем баз данных поддержка индексных структур и их использование выполняется автоматическими средствами. В этой главе мы будем составлять структуры и алгоритмы, которые можно использовать вне автоматики и пользоваться всеми возможностями безотносительно ограничений системы базы данных. Примерно как если бы по частям реализовали внутренние механизмы большой системы, но в несколько упрощенном варианте.

3.2 Механизм поддержки индекса

Индексная структура по своему состоянию должна соответствовать состоянию индексируемых данных. Поэтому операции обновления индексов обычно делят на две группы - динамическое обновление индексных структур при обновлении одной записи и массовые операции удаления / построения индексов.

Далее будем рассматривать строки данных, устроенные для простоты следующим образом:

1. Идентификатор записи получаем инкрементом узла ^Data
2. Значение записи хранится в узле ^Data(id)
3. Запись состоит из полей с разделителем ~ (тильда)
4. Индексные записи храним с глобале ^Index
5. В записи предполагаем поля - фигура, цвет, количество
6. Общее строение записи: ^Data(id)=Figure~Color~Count

Операции динамического обновления индексов могут вызываться из операции обновления записи, и либо предшествовать собственно сохранению основной записи, либо последовать ему, либо обрамлять.

Например:

```
 ; просто сохранение объекта
SaveObject(id,ObjVal)
 i '+$g(id) s id=$i(^Data)
 s ^Data(id)=ObjVal
 q
 ; обновление индексов перед сохранением
SaveObject(id,ObjVal)
 n OldValue
 i '+$g(id) s id=$i(^Data)
 s OldValue=$g(^Data(id))
 d DeleteIndices(id,OldValue)
 d InsertIndices(id,ObjVal)
 s ^Data(id)=ObjVal
 q
 ; обновление индексов после сохранения
SaveObject(id,ObjVal)
 n OldValue
 i '+$g(id) s id=$i(^Data)
 s OldValue=$g(^Data(id))
 s ^Data(id)=ObjVal
 d DeleteIndices(id,OldValue)
 d InsertIndices(id,ObjVal)
 q
 ; обрамление обновления индексов при сохранении
SaveObject(id,ObjVal)
 i '+$g(id) s id=$i(^Data)
 d DeleteIndices(id,$g(^Data(id)))
 s ^Data(id)=ObjVal
 d InsertIndices(id,ObjVal)
 q
```

Здесь DeletIndices удаляет индексные записи по этому объекту, а InsertIndices их создает. В данном случае подразумевается простой формат хранения записи - одной строкой, которая трактуется либо как строка содержащая одно значение.

Несмотря на то, что три метода в итоге дают одинаковый результат, между ними есть разница в том, насколько правильно будет работать конкурентный (одновременный для нескольких процессов) доступ к данным и индексам. В случае хранения только данных этот вопрос практически не стоит, поскольку операция set атомарная в том смысле, что в операции выполняется только одно изменение в глобалах. В случае же применения параллельных структур индексов существует момент между состояниями, когда записи нет, но индекс есть, или наоборот, индекс есть но записи нет. Этот вопрос решается обычно с помощью применения блокировок. Операция set нового значения записи обрамляется командами

```
l +^Data(id)
s ^Data(id)=ObjVal
l -^Data(id)
```

И внутри функций удаления / вставки индексных записей также вставляются обрамляющие блокировки. Наличие блокировок особенно критично в случае исполнения кода в контексте транзакции и возможности выполнения операции trollback.

Различие в режиме перестроения индекса, а именно что раньше появится в базе - индексная запись или запись с данными, позволяет построить в некотором смысле самовосстанавливающуюся систему, которая будет иметь возможность восстановиться в случае сбоя при записи строки данных. Если индекс построен раньше, то при выборке по индексу функция выборки данных может определить, что индексная запись существует, но ей не соответствует строка данных.

В случае применения блокировок в операции обновления записи мы в функции выборки можем также попытаться заблокировать эту же запись и, если блокировка оказалась успешной, но записи нет, или ее состояние не соответствует индексным значениям, то значит что операция записи самой строки данных была неуспешной и следует просто удалить индексную запись. Механизм довольно громоздкий, но в ситуации, когда из соображений эффективности не хочется применять транзакции, может оказаться полезным. Вопрос выбора стратегии обновления индекса при обновлении записи оставим программисту.

Операция перестроения индекса сводится к удалению всех индексных записей и перебору всех имеющихся записей с данными и построения индексных записей по каждой имеющейся записи данных. Полагаем, что есть функции DeleteIndex для удаления всех индексных записей по одному индексу. Тогда перестроение индекса может выглядеть как

```
UpdateIndex(IndexName)
 d DeleteIndex(IndexName)
 n id,ObjValue
 s id="" f  s id=$o(^Data(id),ObjValue) q:id=""  d
 . d InsertIndex(IndexName,id,ObjVal)
 q
```

3.3 Простой индекс

Простой индекс в некоторой литературе ещё называется обратным списком. Если структуры основных данных отображают идентификатор записи (назначенный программистом или поддерживаемый автоматически

системой) на запись как совокупность значений атрибутов, то простой индекс отображает значение индексируемых атрибутов на идентификатор или набор идентификаторов записей.

Наличие такого быстрого отображения и реализует способ быстрого поиска записи по значению атрибута. Положим, что в нашем случае записи имеют структуру

```
^Data(id)=Figure~Color~Count
```

Тогда при наличии индекса по фигуре мы должны поддерживать актуальность дополнительной структуры данных

```
^Index(Figure,id)=""
```

Эта структура позволяет, задав значение фигуры, получить набор идентификаторов записей, в которых встречается это значение фигуры. В случае поддержки нескольких индексов, конечно, структура должна быть усложнена:

```
^Index("Figure",Figure,id)=""
^Index("Color",Color,id)=""
^Index("Count",Count,id)=""
```

с целью повторного использования имени индексной глобали.

Можно также выбрать вариант хранить индексы по разным атрибутам в разных глобалях, но обычно придерживаются использования одной. Это сокращает список используемых глобалов и упрощает сопровождение и модификацию программ.

Простой индекс используется как в задачах выборки данных по условию, так и определения существования записей с заданными значениями атрибутов. Что используется при реализации ограничений различного вида.

Небольшой пример реализации простых индексов:

```
ind01 ; простые индексы, автоматический идентификатор
 q
CreateRecords()
 k ^Index
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
 s Counts="2~5~12~8"
 f i=1:1:12 d
 . s Figure=$p(Figures,"~",$r(4)+1)
```

```
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
DeleteRecord(id)
 q:'$d(^Data(id))
 l +^Data(id)
 n RecordValues s RecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(id)
 l -^Data(id)
 q
UpdateRecord(id,RecordValues)
 q:'$d(^Data(id))
 l +^Data(id)
 n OldRecordValues s OldRecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,OldRecordValues)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q
DeleteIndex(IndexName)
 k ^Index(IndexName)
 q
InsertIndexRecords(id,RecordValues)
 d InsertIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d InsertIndexRecord("Color",id,$p((RecordValues),"~",2))
 d InsertIndexRecord("Count",id,$p((RecordValues),"~",3))
 q
DeleteIndexRecords(id,RecordValues)
 d DeleteIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d DeleteIndexRecord("Color",id,$p((RecordValues),"~",2))
 d DeleteIndexRecord("Count",id,$p((RecordValues),"~",3))
 q
InsertIndexRecord(IndexName,id,Value)
 l +^Index(IndexName,Value,id)
 s ^Index(IndexName,Value,id)=""
 l -^Index(IndexName,Value,id)
 q
DeleteIndexRecord(IndexName,id,Value)
 l +^Index(IndexName,Value,id)
 k ^Index(IndexName,Value,id)
```

```
l -^Index(IndexName,Value,id)
q
```

3.4 Составной индекс

Если простой индекс отображает значение одного атрибута на идентификатор или набор идентификаторов строк, то составной индекс отображает совокупность значений двух или более атрибутов на идентификатор или набор идентификаторов строк.

Структурно индексные записи содержат значения не одного, а двух или более атрибутов в определенном порядке. При этом значения атрибутов могут быть как указаны по отдельности в подъиндексах узлов, так и может быть применена функция, получающая по двум или более значениям одно значение, которое не можут быть получено при других значениях атрибутов. Приведем простой пример с одним составным индексом по атрибутам Figure и Color. В первом случае, при раздельном указании значений атрибутов в индексе:

```
^Index("FigureColor",Figure,Color,id)=""
```

Во втором случае при использовании, например, функции $listbuild:

```
^Index("FigureColor",$lb(Figure,Color),id)=""
```

Или при использовании одного значения в виде строки с разделителем:

```
^Index("FigureColor",Figure_"~"_Color,id)=""
```

Вопрос выбора функции получения составного значения по нескольким оставим программисту. Думаю, что это лучше решать в каждом конкретном случае. В расчет следует принимать не только уникальность её результата для значений аргументов, но и возможность использовать индексную сортировку при выборке данных, а также время вычисления индексирующего значения.

Составные индексы применяются в выборках данных, где задано условие на два атрибута одновременно. Фактически, имея составной индекс по этим атрибутам, мы получаем максимальную скорость выборки.

Кроме указания идентификатора записи в подъиндексе узла индексной записи можно также применить хранение набора идентификаторов в одном значении. Например:

```
^Index("FigureColor",Figure,Color)=idlist
```

Здесь idlist - список идентификаторов строк. Такая организация индекса позволяет организовать эффективное покрытие. Покрывающим индексом называется индекс, который может быть применен в другом типе выборки. В случае составного индекса например это могут быть узлы двух видов:

```
^Index("FigureColor",Figure)=FigureIdList
^Index("FigureColor",Figure,Color)=FigureColorIdList
```

Например, пусть есть записи

```
1 шарик    красный   20
2 шарик    зеленый   12
3 квадрат  синий     5
```

Им будут соответствовать индексные записи:

```
^Index("FigureColor","шарик")=$lb(1,2)
^Index("FigureColor","шарик","красный")=$lb(1)
^Index("FigureColor","шарик","зеленый")=$lb(2)
^Index("FigureColor","квадрат")=$lb(3)
^Index("FigureColor","квадрат","синий")=$lb(3)
```

Второй вариант организации покрывающего индекса есть применение операции не $o() для выборки данных, а операции $q(). При ее использовании в качестве покрывающего индекса может быть использована начальная простая структура вида

```
^Index("FigureColor",Figure,Color,id)=""
```

В данном случае индекс покрывает две выборки - при указании как обеих атрибутов Figure и Color, так и при указании только атрибута Figure. Приведем пример применения такого покрытия:

```
n Data,id,ref
s Data("FigureColor","шарик","красный",1)=""
s Data("FigureColor","шарик","зеленый",2)=""
s Data("FigureColor","квадрат","красный",3)=""
s Data("FigureColor","квадрат","синий",4)=""
s Data("FigureColor","шарик","красный",5)=""
w "Select шарик + красный",!
s id=""
f  s id=$o(Data("FigureColor","шарик","красный",id)) q:id=""  d
. w id,!
```

```
 w "Select шарик",!
 s ref=$na(Data("FigureColor","шарик")) »
        f  s ref=$q(@ref) q:ref=""  d
 . w $qs(ref,4),!
 q
```

Первая выборка выдает по условию по двум атрибутам, вторая - по одному.

Использование и поддержка покрывающих индексов может быть как очень эффективна по быстродействию и задействованию одного индекса в нескольких задачах, так и может оказаться неэффективна в случае катастрофического роста объема данных, занимаемого самим индексом и фактическим обесцениванием кеширования базы данных. Фактически, составной индекс является прямым конкурентом методу многоиндексной выборки.

В случае если движок базы данных не умеет использовать многоиндексную выборку, то составному индексу альтернатив, конечно, нет. В случае же возможности выбора следует оценить затраты на поддержку составного индекса. При высокой кардинальности атрибутов составной индекс, видимо, может оказаться менее эффективным, чем многоиндексная выборка по совокупным затратам.

Оценку затратности и предпочтительности разных вариантов лучше производить в реальном проекте на реальных объемах данных, сопоставив с используемым аппаратным обеспечением. На общую оценку может повлиять и общая использованная алгоритмика, и объем кеша, и объем данных подлежащих индексированию, и объем получающихся в итоге выборок.

Небольшой пример реализации составного индекса:

```
ind03 ; составной индекс по Figure м Color
 q
CreateRecords()
 k ^Index
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
 s Counts="2~5~12~8"
 f i=1:1:12 d
 . s Figure=$p(Figures,"~",$r(4)+1)
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
```

```
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
DeleteRecord(id)
 q:'$d(^Data(id))
 l +^Data(id)
 n RecordValues s RecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(id)
 l -^Data(id)
 q
UpdateRecord(id,RecordValues)
 q:'$d(^Data(id))
 l +^Data(id)
 n OldRecordValues s OldRecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,OldRecordValues)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q
InsertIndexRecords(id,RecordValues)
 n Figure,Color
 s Figure=$p((RecordValues),"~",1)
 s Color=$p((RecordValues),"~",2)
 l +^Index("FigureColor",Figure,Color,id)
 s ^Index("FigureColor",Figure,Color,id)=""
 l -^Index("FigureColor",Figure,Color,id)
 q
DeleteIndexRecords(id,RecordValues)
 n Figure,Color
 s Figure=$p((RecordValues),"~",1)
 s Color=$p((RecordValues),"~",2)
 l +^Index("FigureColor",Figure,Color,id)
 k ^Index("FigureColor",Figure,Color,id)
 l -^Index("FigureColor",Figure,Color,id)
 q
```

Здесь поддерживается только один составной индекс.

То, что индекс является составным, одновременно с тем не мешает ему быть простым или кластерным. Эти возможности можно совмещать. В приведенном примере показан простой составной индекс.

Можно также обратить внимание на то, что в индексных записях для большого объема данных оказывается много записей, имеющих дублирующиеся фрагменты. В частности, могут совпадать следующие фрагменты

```
^Index("FigureColor",Figure,Color)
^Index("FigureColor",Figure)
^Index("FigureColor")
```

В практически всех MUMPS системах, в отличие от многих простых NoSQL систем, применяется компрессия ключей, и в большинстве случаев такие дублирования фрагментов не приводят к какому-либо существенному росту объема индексов.

3.5 Покрывающий индекс

Покрывающим индексом (cover index) называется индекс, который может быть использован для нескольких задач выборки данных, в том числе для тех, для которых он изначально не планировался. Название отражает не вид структуры, а различные возможности ее применения.

Индекс в силу своей структуры есть отображение значений атрибутов на идентификаторы записей. Поэтому содержит как набор значений атрибутов, так и набор отображений. Практически любой индекс в силу своего строения является покрывающим для по крайней мере нескольких задач:

1. Поскольку содержит набор значений атрибутов, может быть использован для выборки списка различных имеющихся значений атрибутов.

2. Поскольку каждое значение атрибута отображается на набор идентификаторов записей, можно найти записи по заданному значению.

3. Если значения атрибута находятся в некотором упорядочении, то можно получить сортировку по значениям атрибутов в этом упорядочении.

4. Возможно определить, существует ли запись с заданным значением, и тогда можно реализовать ограничения различного рода, накладываемые на данные. В частности, запрет иметь значение "синий" более чем 5 записям одновременно.

Каждая возможность применения индекса вызвана некоторым отношением - либо внутри однородного набора, либо между значением и набором. И, чем больше отношений содержится в индексе, тем больше покрытий он обеспечивает. Например, пусть есть таблица с атрибутами Color и Figure. Составной индекс по Color и Figure покрывает уже следующие задачи:

1. Выбрать набор различных значений Color
2. Выбрать по заданному Color идентификаторы записей (применять вместо функции $O функцию $Q)
3. Выбрать по заданному Color и Figure идентификаторы записей (применять функцию $O)
4. Отсортировать идентификаторы по Color и Figure ($O)
5. Выбрать идентификаторы для диапазона Color ($Q)
6. Выбрать идентификаторы для диапазона Figure при задании Color ($O)
7. Выбрать идентификаторы при задании диапазонов Color и Figure ($O по Color + вложенный $O по Figure)
8. Для заданного Color отсортировать по Figure.

При этом нужно отметить, что составной индекс по Color и Figure уже не может быть использован, если условие наложено только на атрибут Figure. Поскольку в таком индексе значения Figure находятся в подчиненной, а не независимой иерархии по отношению к Color. Если необходимо использовать значения Figure независимо от значений Color, то необходимо поддерживать второй индекс соответствующей структуры.

С точки зрения эффективности применение покрывающих индексов вполне оправдано, поскольку одна индексная структура может быть использована многократно в различных задачах. При этом чем больше атрибутов записи объединяет индекс, тем больше покрытий он обеспечивает. В случае применения межтабличных индексов эффективность покрывающего свойства многократно увеличивается, поскольку позволяет сократить число операций соединений.

Приведем демонстрационный пример поддержания структуры данных для записей с атрибутами Color и Figure, после чего приведем примеры использования одного и того же индекса в различных задачах:

```
CreateRecords()
 k ^Index
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
 s Counts="2~5~12~8"
 f i=1:1:120 d
```

```
 . s Figure=$p(Figures,"~",$r(4)+1)
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
DeleteRecord(id)
 q:'$d(^Data(id))
 l +^Data(id)
 n RecordValues s RecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(id)
 l -^Data(id)
 q
UpdateRecord(id,RecordValues)
 q:'$d(^Data(id))
 l +^Data(id)
 n OldRecordValues s OldRecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,OldRecordValues)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q
DeleteIndex(IndexName)
 k ^Index(IndexName)
 q
InsertIndexRecords(id,RecordValues)
 d InsertIndexRecord("ColorFigure",id,
   $p((RecordValues),"~",2),$p((RecordValues),"~",1))
 q
DeleteIndexRecords(id,RecordValues)
 d DeleteIndexRecord("ColorFigure",id,
   $p((RecordValues),"~",2),$p((RecordValues),"~",1))
 q
InsertIndexRecord(IndexName,id,ColorValue,FigureValue)
 l +^Index(IndexName,ColorValue,FigureValue,id)
 s ^Index(IndexName,ColorValue,FigureValue,id)=""
 l -^Index(IndexName,ColorValue,FigureValue,id)
 q
DeleteIndexRecord(IndexName,id,ColorValue,FigureValue)
 l +^Index(IndexName,ColorValue,FigureValue,id)
 k ^Index(IndexName,ColorValue,FigureValue,id)
 l -^Index(IndexName,ColorValue,FigureValue,id)
```

```
 q
```

Приведем несколько примеров таких выборок различного назначения по одному и тому же индексу.

Выбрать набор различных значений Color

```
Select1()
 n Color
 s Color=""
 f  s Color=$o(^Index("ColorFigure",Color)) q:Color=""  d
 . w Color,!
 q
```

Выбрать по заданному Color идентификаторы записей (применять вместо функции $O функцию $Q)

```
Select2(Color="красный")
 n ref
 s ref=$na(^Index("ColorFigure",Color))
 f  s ref=$q(@ref) q:$s(ref="":1,$qs(ref,2)'=Color:1,1:0)  d
 . w $qs(ref,4),!
 q
```

Выбрать по заданному Color и Figure идентификаторы записей (применять функцию $O)

```
Select3(Color="синий",Figure="квадрат")
 n id
 s id=""
 f  s id=$o(^Index("ColorFigure",Color,Figure,id)) q:id=""  d
 . w "Found id: "_id,!
 q
```

Отсортировать идентификаторы по Color и Figure ($O)

```
Select4(COrder=1,FOrder=-1)
 n c,f,id
 s c="" f  s c=$o(^Index("ColorFigure",c),COrder) q:c=""  d
 . s f="" f  s f=$o(^Index("ColorFigure",c,f),FOrder) q:f=""  d
 . . s id="" f  s id=$o(^Index("ColorFigure",c,f,id)) q:id=""  d
 . . . w c_" "_f_" "_id,!
 q
```

3.6 Кластерный индекс

Кластерный индекс отображает совокупность значений одного или нескольких атрибутов на совокупность значений остальных атрибутов. В случае применения простого индекса он также может быть назван кластерным, если значение id задается неавтоматически и является равнозначным с остальными атрибутом, и некластерным если поддерживается автоматически и не входит в перечень атрибутов. То есть хранение в предыдущем случае структуры данных как

```
^Data(id)=Figure~Color~Count
```

Является кластерным индексом если значение id задается неавтоматически и id является одним из атрибутов записи.

Интересна история с развитием идей SQL в отношении кластерных индексов. Первоначальная формулировка SQL касалась такого объединения строк в таблице, что все значения в строке равнозначны и строка объединяет их в одну запись. Каким бы образом не выполнялось хранение записей, во внутренних механизмах любой СУБД присутствует способ адресации и идентификации любой записи. При этом практически сразу перед разработчиками встала проблема того, что этот внутренний идентификатор на языке SQL напрямую практически недоступен и для указания определенной записи среди набора имеющих одинаковые значения необходимо было применить дополнительное поле данных, в которое самостоятельно писать условный идентификатор такой записи, но к которому СУБД относится так же, как и к любому другому полю записи.

Структуру вида

```
IntId  Color     Figure
F23H   Синий     Круглый
FM90   Синий     Круглый
```

было необходимо трансформировать в

```
IntId  Id    Color     Figure
F23H   14    Синий     Круглый
FM90   15    Синий     Круглый
```

Впоследствии достаточно быстро обнаружилось, что именно эти условные дополнительные значения, добавляемые к каждой из записей, и

идентифицируют запись в практически всех операциях, и всегда на такое искусственное поле добавляется индекс. Получалось, то из-за недоступности внутреннего идентификатора разработчики заставляли систему выполнять то же самое врукопашную.

Далее разработчики таких SQL СУБД переизобрели заново кластерный индекс и исключили внутренний идентификатор, заменив структуру на

```
Id     Color      Figure
14     Синий      Круглый
15     Синий      Круглый
```

и используя поле Id (указываемое при объявлении как кластерного хранения, так и кластерного индекса) вместо своего внутреннего идентификатора IntId, хотя многие СУБД и приписывают свой идентификатор записи для выполнения внутренних системных операций.

В настоящее время большинство коммерческих SQL СУБД либо уже добавили поддержку кластерного хранения записей и кластерных индексов, либо ими пока не восхищаются. Хотя при этом в большинстве случаев применений в SQL системах кластерный индекс используется не столько для действительно индексирования, сколько для улучшения эффективности структуры хранения, и в качестве кластерного поля используется именно искусственно добавленный ключ.

Построение кластерного индекса необязательно означает уникальность индексируемого атрибута. Но, в случае такой уникальности, применение кластерного индекса более оправдано. Пусть мы строим кластерный индекс по атрибуту Figure. Поскольку атрибут неуникален, то мы должны отобразить значение Figure на совокупность Color + Count:

```
^Data(Figure,internalid)=Color~Count
```

Здесь значение internalid является вспомогательным строго в целях создания уникальности отображения Figure на Color + Count. Вообще говоря, значение internalid по его назначению даолжно быть уникально лишь в пределах каждого Figure. Но также можно использовать уникальность в пределах всего набора записей.

Вообще говоря, большинство систем баз данных поддерживают создание только одного кластерного индекса на массив. По своему смыслу кластерный индекс сочетает в себе и простой индекс и саму запись данных. В отличие от простого идентификатора в случае простого индекса при применении кластерного индекса идентификатор строки становится составным. В нашем случае это совокупность значений атрибутов, входящих в индекс, и внутреннего дежурного идентификатора.

Внутрисистемная адресация строки в случае применения кластерного индекса везде заменяется с использования простого id на совокупность Figure + internalid.

Операции выборки в случае использования кластерного индекса, естественно, должны учитывать, что хранение массива данных уже не одноуровневое, а двухуровневое и, скажем, при выборке всех записей использовать либо вложенный цикл с перечислением по $o(), либо применять перечисление по $q().

Применение кластерного индекса нисколько не мешает применять также и простые индексы на тот же массив данных по другим атрибутам, а также и по тому же самому. Опять же, в отличие от простого индекса, следует использовать составной идентификатор. Например, дополнительный индекс по цвету может выглядеть так:

```
^Index("Color",Color,Figure,internalid)=""
```

Или можно использовать некоторую функцию, которая по значениям Figure и internalid составляет значение, которое не может быть получено при иных значениях Figure и internalid.

```
^Index("Color",Color,$$func(Figure,internalid))=""
```

Или в случае уникальности internalid в пределах массива данных

```
^Index("Color",Color,internalid)=Figure
```

Структура второго варианта позволяет сохранить уникальность отображения и одновременно получить составной идентификатор. По значению Color получаем набор internalid и для каждого из них восстанавливаем полный идентификатор, используя значение Figure.

Основными преимуществами кластерных индексов являются эффективность использования дискового пространства в случае уникальности атрибута кластерного индекса и эффективность кеширования блоков данных при кластеризации таблиц.

Небольшой пример реализации кластерного индекса:

```
ind02 ; кластерный индекс по атрибуту Figure
 q
CreateRecords()
 k ^Index
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
```

```
 s Counts="2~5~12~8"
 f i=1:1:12 d
 . s Figure=$p(Figures,"~",$r(4)+1)
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
 n internalid,Figure,id
 s Figure=$p(RecordValues,"~",1)
 s internalid=$i(^Data)
 l +^Data(Figure,internalid)
 s $p(RecordValues,"~",1)=""
 s ^Data(Figure,internalid)=RecordValues
 s id=$lb(Figure,internalid)
 d InsertIndexRecords(id,RecordValues)
 l -^Data(Figure,internalid)
 q id
DeleteRecord(id)
 n RecordValues,Figure,internalid
 s Figure=$lg(id,1),internalid=$lg(id,2)
 q:'$d(^Data(Figure,internalid))
 l +^Data(Figure,internalid)
 s RecordValues=$g(^Data(Figure,internalid))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(Figure,internalid)
 l -^Data(Figure,internalid)
 q
UpdateRecord(id,RecordValues)
 d DeleteRecord(id)
 q $$InsertRecord(RecordValues)
DeleteIndex(IndexName)
 k ^Index(IndexName)
 q
InsertIndexRecords(id,RecordValues)
 d InsertIndexRecord("Color",id,$p(RecordValues,"~",2))
 d InsertIndexRecord("Count",id,$p(RecordValues,"~",3))
 q
DeleteIndexRecords(id,RecordValues)
 d DeleteIndexRecord("Color",id,$p(RecordValues,"~",2))
 d DeleteIndexRecord("Count",id,$p(RecordValues,"~",3))
 q
InsertIndexRecord(IndexName,id,Value)
 n Figure,internalid
 s Figure=$lg(id,1),internalid=$lg(id,2)
 l +^Index(IndexName,Value,Figure,internalid)
 s ^Index(IndexName,Value,Figure,internalid)=""
 l -^Index(IndexName,Value,Figure,internalid)
 q
```

```
DeleteIndexRecord(IndexName,id,Value)
 n Figure,internalid
 s Figure=$lg(id,1),internalid=$lg(id,2)
 l +^Index(IndexName,Value,Figure,internalid)
 k ^Index(IndexName,Value,Figure,internalid)
 l -^Index(IndexName,Value,Figure,internalid)
 q
```

В этом примере при обновлении строки данных производится её удаление, потом вставка новой. Это связано с тем, что новое значение атрибута Figure может быть изменено. В более практичной реализации лучше определять именилось ли значение кластерного атрибута. Эта тема рассматривается позднее, в теме дифференциального перестроения индекса. Построение остальных двух индексов такое же как в предыдущем случае, поскольку они также простые.

3.7 Хеш-индекс

Хеш-индекс - это структура данных, отображающая на идентификатор строки не значение или совокупность значений атрибутов, а хеш-код этого значения, или совокупности значений. Выбор хеш-функции и сама необходимость применения хеш-индексов в каждом конкретном случае выбирается программистом отдельно. В расчет должны браться факторы как возможности применения индекса для выборки, так и скорость кода поддержки обновления индексных записей.

Структурно хеш индекс выглядит также как и другие:

```
^Index("Figure",$$hash(Figure),id)=""
```

Также хеш-индекс может быть составным или хеш строится по двум или более атрибутам. В этом случае хеш-функция применяется к совокупности значений атрибутов:

```
^Index("FigureColor",$$hash(Figure,Color),id)=""
```

Поскольку, по определению, хеширование не является однозначным отображением, применив вместо значений атрибутов их суррогаты в виде хеш кодов мы получаем грязную выборку. То есть, зафиксировав значение Figure, идентификаторы которых мы хотим, получим хеш-код Figure. Но в силу неуникальности хеширования в подъиндексах узла ^Index("Figure",$$hash(Figure)) будут идентификаторы также записей с другим значением атрибута Figure.

Применяется хеш-индекс в случаях когда допустимо построение временных структур, и быстрое деление записей на группы существенно ускоряет какую-либо операцию. Применение хеш-индексов весьма эффективно если необходимо соединить две выборки по длинному атрибуту или по совокупности атрибутов, поскольку хеш-структура занимает в памяти гораздо меньше места и для относительно небольших выборок (несколько сот записей) обеспечивает достаточно быстрый поиск.

При соединении двух таблиц, для который нет индексов для полей соединения, в которых есть, скажем, N и M записей, выполняется сканирование первой и для каждой найденной записи производится полное сканирование второй. Получается N*M сравнений.

В случае применения хешированного соединения сервер строит временный хеш-индекс на вторую таблицу. Это M операций. Положим, что хеширование дает приблизительно равномерное разбиение на n групп. Проходом по записям первой таблицы получаем значение атрибута соединения, вычисляем его хеш-код, и перебираем группу из примерно M/n записей с этим хеш-кодом. На одну запись из N таким образом приходится примерно M/n сравнений. То есть количество сравнений мы уменьшили примерно пропорционально n. В силу того, что обычно в выборках участвует конечное число элементов (M), используя характерный для такой выборки параметр разбиения хеша n мы сводим временную сложность от квадратичной к почти линейной. Если число n сопоставимо с M, то мы можем получить почти простой индекс.

В случае если число n намного меньше кардинальности атрибута, такой способ гораздо эффективнее, чем обычный индекс, по расходам памяти. То есть построение временного полноценного индекса на вторую таблицу в приведенном случае может быть менее эффективным из-за расхода памяти и обесценивания кеша.

Другим способом является получение выборки сразу с хешированием атрибутов, участвующих в соединении.

Применение хеш-индексов отличается от применения обычных, опять же, в силу неуникальности хеширования. Они обязательно требуют применения пусть некоторого, но дополнительного перебора записей в хеш-группе, или, другими словами, выборки с фильтрацией.

Эффективность хеш-функций, вообще говоря, не предмет индексации, поэтому ниже приведем только простейшую. Сами же хеширующие функции тем более качественны, чем более равномерное разбиение по группам дают для входного набора строк. Ко второй качественной характеристике хеширования относится число групп, на которое хеш разбивает входные данные. Третья качественная характеристика хеширования - скорость вычисления хеша. Во многих случаях оправдано применение

встроенных системных функций вычисления CRC.

```
hash(Value,groupsize)
        n ret,i,magic1,magic2
        s magic1=0
        s magic2=142
        s:'+$g(groupsize) groupsize=255
        s ret=magic1
        f i=1:1:$l(Value) d
        . s ret=((ret+magic2)*$a($e(Value,i)))#groupsize
        q ret
```

Здесь каждой строке ставится в соответствие одно из чисел в диапазоне от 0 до groupsize, где groupsize по умолчанию 255.

На практике хеш-индексы редко используются в качестве постоянных структур, и чаще применяются для соединения временных выборок. В случае, если строится выборка для дальнейшего соединения с другой выборкой по хешу, то при их построении сразу применяется построение выборки в хеш-структуре.

В действительности, приведенная структура хеш-индекса это лишь приближение, поскольку MUMPS всегда физически оперирует парами ключ - значение, объединенными в деревья. Любое обращение по номеру хэш-кода в любое дерево это в действительности итеративная процедура поиска с отсечением, пусть и довольно быстрая, в то время как хеш-таблицы - это массивы прямой адресации, где хэш-код используется в качестве номера начала списка коллизий.

3.8 Битмап индекс (bitmap)

Битовый индекс является дополнительной структурой данных к основной, отображающей значения индексируемого атрибута на набор идентификаторов записей. В целом определение такое же, как и у простого индекса, но набор идентификаторов записей формируется иначе. Если в случае простого индекса хранятся значения идентификаторов и, вообще говоря, они могут быть произвольными, в том числе строками или составными, то в случае битовых индексов идентификаторы рассматриваются строго как целые числа.

Набором идентификаторов является битовая последовательность, в которой идентификатору строки соответствует положение бита в соответствии с его величиной. Наличие записи с заданным значением атрибута отмечается 1, отсутствие - 0 или пустым хвостом. Положим, что есть три записи:

```
1    шарик     красный     12
2    шарик     синий        5
3    кубик     красный      7
4    кубик     синий        3
```

В этом случае есть 4 идентификатора, их значения рассматриваются как позиции битов в битовой карте. В карте получается в данном случае 4 бита. По атрибуту фигура два различных значения, поэтому в этом индексе будет две карты. То же самое по атрибуту цвет.

```
Индекс по фигуре
значение атрибута      биты  0  1  2  3  4
шарик                        0  1  1  0  0
кубик                        0  0  0  1  1
Индекс по цвету
значение атрибута      биты  0  1  2  3  4
красный                      0  1  0  1  0
синий                        0  0  1  0  1
```

Чтобы использовать битовые карты, база данных должна поддерживать операции с битовыми строками. Это может быть либо встроенный функционал, либо быть реализован с помощью какого-либо модуля расширения. В примере будем использовать функции $bit из состава Caché или MiniM.

Операции получения идентификаторов строк по значениям атрибутов сводятся к использованию битовых операций с битовыми картами и выборке из получившейся битовой карты позиций ненулевых битов. Значения этих позиций считаются идентификаторами строк.

К особенностям битовых индексов относится то, что они могут быть, как и простые индексы, составными, но не могут быть кластерными. Другим ограничением является то, что идентификатором строки данных должно быть строго натуральное число. В большинстве систем второе ограничение совершенно незаметно, поскольку идентификаторы и без того поддерживаются в автоинкрементном режиме.

Битовые индексы и простые индексы по одной и той же таблице могут быть совмещены. Но если для таблицы используется кластерный индекс, то битовый индекс для нее уже не может быть применен, поскольку идентификатор строки фактически нечисловой. Совмещение и одновременное применение простых и битовых индексов выполняется алгоритмически. Также может быть организована многоиндексная выборка совместно из нескольких простых и битовых индексов.

Небольшой пример реализации битового индекса:

```
ind04 ; битовые индексы, автоматическое поддержание идентификатора
 q
CreateRecords()
 k ^Index
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
 s Counts="2~5~12~8"
 f i=1:1:12 d
 . s Figure=$p(Figures,"~",$r(4)+1)
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
DeleteRecord(id)
 q:'$d(^Data(id))
 l +^Data(id)
 n RecordValues s RecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(id)
 l -^Data(id)
 q
UpdateRecord(id,RecordValues)
 q:'$d(^Data(id))
 l +^Data(id)
 n OldRecordValues s OldRecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,OldRecordValues)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q
DeleteIndex(IndexName)
 k ^Index(IndexName)
 q
InsertIndexRecords(id,RecordValues)
 d InsertIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d InsertIndexRecord("Color",id,$p((RecordValues),"~",2))
 d InsertIndexRecord("Count",id,$p((RecordValues),"~",3))
 q
DeleteIndexRecords(id,RecordValues)
 d DeleteIndexRecord("Figure",id,$p((RecordValues),"~",1))
```

```
 d DeleteIndexRecord("Color",id,$p((RecordValues),"~",2))
 d DeleteIndexRecord("Count",id,$p((RecordValues),"~",3))
 q
InsertIndexRecord(IndexName,id,Value)
 s $bit(^Index(IndexName,Value),id)=1
 q
DeleteIndexRecord(IndexName,id,Value)
 s $bit(^Index(IndexName,Value),id)=0
 q
```

Сравнив с рутиной ind01 для простых индексов, несложно понять, что единственные отличия содержатся в функциях InsertIndexRecord и DeleteIndexRecord. Это простой учебный пример, здесь не проверяется что идентификатор действительно является натуральным числом.

Также строится только одна битовая карта. В реальном приложении, несмотря на то, что хранится только один бит на запись, битовые карты должны быть сегментированы, поскольку работа с неограниченными строками обычно не предусматривается. Разбиение по сегментам выполняется вычислением номера сегмента и позиции в сегменте по значению идентификатора. В системе должна быть выбрана одна и та же гранулярность сегмента, чтобы битовые логические операции выполнялись корректно. Точное значение размера битового сегмента, вообще говоря, лучше определять экспериментально. Об этом пойдет речь в части сравнения битовых и простых индексов.

Простой пример сегментирования битового индекса:

```
#define BITSEGMENT 32000
InsertIndexRecord(IndexName,id,Value)
 n seg,pos
 s seg=id\$$$BITSEGMENT
 s pos=id#$$$BITSEGMENT
 s $bit(^Index(IndexName,Value,seg),pos)=1
 q
DeleteIndexRecord(IndexName,id,Value)
 n seg,pos
 s seg=id\$$$BITSEGMENT
 s pos=id#$$$BITSEGMENT
 s $bit(^Index(IndexName,Value,seg),pos)=0
 q
```

Здесь номер сегмента получается делением идентификатора нацело на размер сегмента, а позиция в сегменте определяется остатком от деления нацело идентификатора на размер сегмента. Здесь, поскольку значения идентификаторов рассматриваются как позиции битов, величина сегмента BITSEGMENT означает число бит, а не байт. В зависимости

от применяемой версии сервера, размера блока базы данных и метода применяемой упаковки битовой карты эту величину можно подобрать экспериментально более оптимальной, или обратиться в техподдержку MUMPS системы за рекомендациями.

В действительности, это тоже еще не все - у реализации битовых функций в Caché и MiniM есть техническая особенность - нельзя использовать нулевое число в качестве позиции в сегменте (третий параметр). При этом число ноль получается для идентификаторов, кратных размеру сегмента. Чтобы избежать этой проблемы, можно поступить просто - после вычисления правильной позиции в сегменте к ней прибавить единицу, например, и во всех функциях, использующих сегменты, учитывать эту неправильную единицу. Например:

```
#define BITSEGMENT 32000
InsertIndexRecord(IndexName,id,Value)
 n seg,pos
 s seg=id\$$$BITSEGMENT
 s pos=id#$$$BITSEGMENT+1
 s $bit(^Index(IndexName,Value,seg),pos)=1
 q
DeleteIndexRecord(IndexName,id,Value)
 n seg,pos
 s seg=id\$$$BITSEGMENT
 s pos=id#$$$BITSEGMENT+1
 s $bit(^Index(IndexName,Value,seg),pos)=0
 q
```

К обычным достоинствам битовых индексов относится то, что битовые операции могут быть оптимизированы и могут выполняться значительно быстрее, чем операции с простыми списками. Другим достоинством является относительно меньший объем используемой дисковой памяти, чем в случае обычных индексов.

К недостаткам битовых индексов относятся необходимость применения битовых операций к всей битовой карте независимо от того, сколько реально там содержится значимых бит. Поскольку в большинстве случаев процессы обращаются к объектам в относительно небольшом диапазоне идентификаторов, вероятность что эти идентификаторы находятся в одном сегменте увеличивается. В случае же уменьшения величины сегмента битовый индекс по структуре и характеристикам приближается к простому обратному списку. В ситуации "размер сегмента равен единице" битовый индекс полностью вырождается в обычный.

К как преимуществам, так и недостаткам битовых индексов, в зависимости от характера их применения может быть отнесено сочетание их

реализации с транзакционностью прикладной системы. Для отдельных реализаций битовых индексов транзакционность системы будет негативным фактором, для других она не имеет значения. Более подробной этот вопрос описывается в главе о тарнзакциях.

В любом случае выбор вида индекса оставим за программистом и руководителем проекта. Отметим, что сейчас мы занимаемся только техническими вопросами, на основе которых и можно сделать аргументированный выбор. В документации и рекомендациях фирм - производителей движков баз данных также можно найти советы по выбору применяемых индексов. Важно понимать, на основе чего они сделаны, каковы характеристики движков этих баз данных и для каких характерных операций этот выбор предлагается.

3.9 Битслайс индекс (bitslice)

Резаный битовый индекс представляет собой совокупность структур, отображающих фрагменты значений атрибутов на набор идентификаторов строк данных. Структурно аналогичен битовому индексу (bitmap), но представляет собой N битовых индексов, работающих для всех значений атрибутов. Индексации подвергаются только числовые атрибуты либо те, которые могут быть приведены к числовому виду. Значение N зависит от типа выбранного атрибута.

В карту входят отдельные карты по каждому биту значения атрибута. В примере рассмотрим тип натуральное число. Само число в нашем примере рассматривается как 32-битное без знака. Берутся отдельные биты числа и по этим битам строится 32 отдельных битовых индекса.

При поиске по индексу производится обратная операция - выясняется какие биты в значении атрибута должны быть единичными и какие нулевыми и конструируется соответствующая битовая операция. Операция применяется к всем 32 битовым картам. В результате производится хранение всех значений индексов в фактически ограниченном дисковом пространстве. Причем объем хранения примерно пропорционален количеству строк данных и не зависит от кардинальности индексируемых атрибутов.

Приведем пример раскладки значений атрибутов по bit-slice индексу.

```
Строки данных
id   Count      Битовое представление атрибута
 1      17      00000000000000000000000000010001
 2       6      00000000000000000000000000000110
 3       5      00000000000000000000000000000101
```

```
 4        8      00000000000000000000000000001000
 5       14      00000000000000000000000000001110
 6       18      00000000000000000000000000010010

Индексные записи
биты       битовые карты, младшие разряды слева
   0       0101000
   1       0010011
   2       0011010
   3       0000110
   4       0100001

В остальных битовых картах нули
```

Использование bit-slice индексов в силу его структуры может быть оправдано не только для числовых атрибутов, но и для любых, для которых возможно битовое представление фиксированной длины, например, даты, номера телефонов, условные коды в виде нескольких символов. Для использования нечислового атрибута для индексации просто потребуются две операции:

1. определение максимального количества бит (для корректного дополнения нулями) для выбранного типа атрибута

2. получение значащей битовой последовательности по значению атрибута

Как и для битмап (bitmap) индексов, при реальном построении движка, использующего bit-slice индексы, из-за априори большого числа записей следует прибегать к сегментированию битовых карт. Схема сегментирования может быть выбрана, например, такой же как и в предыдущем случае с битмап (bitmap) индексами. Формально говоря, bit-slice индексы могут быть совмещены с традиционными bitmap индексами напрямую при выборе совпадающего параметра размера битового сегмента.

К достоинствам bit-slice индексов по сравнению с bitmap индексами относится меньший занимаемый объем при высоко-кардинальных атрибутах. Фактически, из-за своего структурирования bit-slice индексы стремятся повторно использовать дисковое пространство и битовые карты. При снижении кардинальности атрибутов bit-slice индексы фактически приближаются по соотношению нулевых и единичных битов к битмап индексам или имеют худшие характеристики. На этот факт следует обратить внимание при реализации модулей расширения системы базы данных при реализации битовых операций внешними средствами.

К недостаткам относится большее время на выполнение разборок по битовым картам и сборок обратно, а также ещё большая по сравнению с битовыми индексами вероятность взаимоблокировок процессов. Также к недостаткам относятся увеличенные в 32 раза (в приведенном примере) затраты на распаковку сжатых битовых карт для получения того же результата, что и при использовании битовых индексов.

То преимущество bit-slice индексов, что они потенциально могут занимать меньше места чем битовые, в каких-то случаях может перевесить недостаток, связанный с усложнением выборки. При уменьшении занимаемого дискового пространства ускорение достигается меньшим количеством дисковых операций и, во-вторых, в силу совмещения в битовых картах информации о различных значениях атрибутов, значительно более эффективным использованием кеширования.

Проведенный эксперимент с размером, занимаемым индексами на диске в СУБД Caché подтверждает теоретические рассуждения:

Число записей	Число различных значений атрибута	Объем хранения, байт	
10000	4	bitmap	116
		bit-slice	5396
10000	10000	bitmap	120272
		bit-slice	17452
Объем хранения индексируемых данных,			260000

То есть, при увеличении кардинальности атрибута bit-slice индекс намного эффективнее по объему хранения, чем bitmap индекс, по объему хранения. Приведенные числа не пропорциональны параметрам исходных условий, поскольку приведены не информационная емкость индекса, а объем, занимаемый структурой данных на диске. Функции $bit в Caché оперируют сжатыми битовыми строками. При чтении строка распаковывается, при записи снова сжимается. Поэтому объем хранения может варьироваться еще и в зависимости от состояния индекса, насколько хорошо он сжимается реализованным алгоритмом.

Впрочем, пока Caché и MiniM не имеют такой встроенной поддержки битовой нарезки, но специализированные движки баз данных или специальный модуль расширения вполне могут выполнять такие битовые операции не прибегая к интерпретируемому режиму. В качестве примера применения bit-slice индексов приведем воспроизводимый код для Caché и MiniM, построенный на $bit функциях. Код приводится только в учебных целях.

```
CreateRecords()
 k ^Index
```

```
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
 s Counts="2~5~12~8"
 f i=1:1:12 d
 . s Figure=$p(Figures,"~",$r(4)+1)
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
DeleteRecord(id)
 q:'$d(^Data(id))
 l +^Data(id)
 n RecordValues s RecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(id)
 l -^Data(id)
 q
UpdateRecord(id,RecordValues)
 q:'$d(^Data(id))
 l +^Data(id)
 n OldRecordValues s OldRecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,OldRecordValues)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q
DeleteIndex(IndexName)
 k ^Index(IndexName)
 q
InsertIndexRecords(id,RecordValues)
 d InsertIndexRecord("Count",id,$p((RecordValues),"~",3))
 q
DeleteIndexRecords(id,RecordValues)
 d DeleteIndexRecord("Count",id,$p((RecordValues),"~",3))
 q
InsertIndexRecord(IndexName,id,Value)
 n i
 l +^Index(IndexName)
 f i=0:1:31 d
 . s $bit(^Index(IndexName,i),id)=(Value\(2**i))#2
```

```
 l -^Index(IndexName)
 q
DeleteIndexRecord(IndexName,id,Value)
 n i
 l +^Index(IndexName)
 f i=0:1:31 s $bit(^Index(IndexName,i),id)=0
 l -^Index(IndexName)
 q
```

Здесь приведен вариант не сегментированного индекса, для построения сегментированного нужно дополнить структуру и алгоритм вычислением номера сегмента и смещения в сегменте:

```
#define BITSEGMENT 32000
InsertIndexRecord(IndexName,id,Value)
 n i
 n seg,pos
 s seg=id\$$$BITSEGMENT
 s pos=id#$$$BITSEGMENT
 l +^Index(IndexName)
 f i=0:1:31 q:'Value  d
 . s $bit(^Index(IndexName,seg,i),pos)=Value#2
 . s Value=Value\2
 l -^Index(IndexName)
 q
DeleteIndexRecord(IndexName,id,Value)
 n i
 n seg,pos
 s seg=id\$$$BITSEGMENT
 s pos=id#$$$BITSEGMENT
 l +^Index(IndexName)
 f i=0:1:31 s $bit(^Index(IndexName,seg,i),pos)=0
 l -^Index(IndexName)
 q
```

Кстати, можно отметить интересную особенность, что при удалении записи собственно значение атрибута при использовании bit-slice (как и bitmap) индексов, вообще говоря, не требуется, поскольку проставление нулевых значений выполняется на все биты, а их количество определяется не значением атрибута, а его типом, тем, сколько бит используется при кодировании атрибута этого типа. Использование значения атрибута позволяет сократить число обращений к индексной глобали, если не проставлять несуществующие значения:

```
DeleteIndexRecord(IndexName,id,Value)
 n i
 n seg,pos
```

```
 s seg=id\$$$BITSEGMENT
 s pos=id#$$$BITSEGMENT
 l +^Index(IndexName)
 f i=0:1:31 q:'Value d
 . s:Value#2 $bit(^Index(IndexName,seg,i),pos)=0
 . s Value=Value\2
 l -^Index(IndexName)
 q
```

При замене типа с числового на иной, как было описано ранее, нужно использовать не явно заданные выражения получения количества битов, а функции, которые а) определяют число бит для типа и б) получают значения бит для значения атрибута, примерно это может выглядеть так:

```
DeleteIndexRecord(IndexName,id,Value)
 n i,len
 s len=$$BitCount(IndexName)
 l +^Index(IndexName)
 f i=0:1:len s $bit(^Index(IndexName,i),id)=0
 l -^Index(IndexName)
 q
InsertIndexRecord(IndexName,id,Value)
 n i,len
 s len=$$BitCount(IndexName)
 l +^Index(IndexName)
 f i=0:1:len d
 . s $bit(^Index(IndexName,i),id)=$$GetBit(IndexName,Value,i)
 l -^Index(IndexName)
 q
BitCount(IndexName)
 q:IndexName="Count" 31
 q 0
GetBit(IndexName,Value,i)
 q:IndexName="Count" (Value\(2**i))#2
 q 0
```

В таком варианте от функций построения битовых карт отделяются две функции, специфичные для используемого типа атрибута. При добавлении еще одного типа нужно будет просто изменить соответственно функции BitCount и GetBit. При использовании иных типов чем числовые можно значительно сократить битовое представление значения атрибута.

Это пример для односегментных bit-slice индексов. Код приведен без оптимизации, только в учебных целях. Для поддержания разделения по сегментам карт нужно просто скорректировать функции изменения карт:

```
#define BITSEGMENT 32000
InsertIndexRecord(IndexName,id,Value)
 n i,seg,pos
 s seg=id\$$$BITSEGMENT
 s pos=id#$$$BITSEGMENT
 l +^Index(IndexName)
 f i=0:1:31 d
 . s $bit(^Index(IndexName,i,seg),pos)=(Value\(2**i))#2
 l -^Index(IndexName)
 q
DeleteIndexRecord(IndexName,id,Value)
 n i,seg,pos
 s seg=id\$$$BITSEGMENT
 s pos=id#$$$BITSEGMENT
 l +^Index(IndexName)
 f i=0:1:31 s $bit(^Index(IndexName,i,seg),pos)=0
 l -^Index(IndexName)
 q
```

Наиболее распространенной операцией, где bit-slice индексы дают существенное преимущество, является операция суммирования значений атрибутов в столбик.

Продемонстрируем принцип вычисления суммы по bit-slice индексу на примере найденной bit-slice карты:

```
Строки данных
id   Count      Битовое представление
 1      17      010001
 2       6      000110
 3       5      000101
 4       8      001000
 5      14      001110
 6      18      010010
```

Колонки справа являются двоичным представлением значений атрибутов. Математически каждой из позиций соответствует позиция в разложении числа по степеням двойки:

```
Строки данных
id   Count      Битовое представление
 1      17      10001
 2       6      00110
 3       5      00101
 4       8      01000
 5      14      01110
 6      18      10010

Степени двойки  43210
```

Каждое из значений атрибутов есть сумма битов умноженных на степень двойки, соответствующей его позиции. Таким образом, полную сумму всех значений можно получить как сумму сумм битов по каждой позиции (здесь в виде колонки) умноженных на степень двойки для этой позиции. Для получения суммы битов по каждой колонке используется встроенная системная функция $bitcount. Таким образом, для вычисления общей суммы необходимо 5 вызовов функции $bitcount и 5 произведений на степени двойки.

Пусть есть число A, разложимое по основанию 2:

$$A = \sum_{n=0}^{N} a_n 2^n$$

здесь все коэффициенты a_n являются значениями либо 0 либо 1. Тогда сумма чисел $A + B + C$ представима как сумма:

$$A + B + C = \sum_{n=0}^{N} a_n 2^n + \sum_{n=0}^{N} b_n 2^n + \sum_{n=0}^{N} c_n 2^n =$$

$$= \sum_{n=0}^{N} (a_n + b_n + c_n) 2^n$$

И для вычисления суммы всех чисел необходимо вычислить суммы битов при каждой из степеней двойки, что и выполняется системной функцией $bitcount.

Битовое системное расширение, функция $bitcount, выполняется на нижнем уровне так, что просто возвращает число единиц или нулей. Выполнение счета проводится на нижнем уровне, в кодах процессора, и очень быстро. При увеличении числа записей выигрыш во времени может достигать сотен раз по сравнению с простым сложением в столбик чисел, на получение каждого из которых выполняется обращение к глобали функцией $order.

Одним из наиболее традиционных применений bit-slice индексов для получения ключевого преимущества по скорости работы являются аналитические системы класса OLAP. В них операции выборки и суммирования применяются одновременно к большому числу записей и их атрибутов.

Для битмап и битслайс индексов в примерах используются блокировки, но для операций выполняемых функциями $bit они не требуются, поскольку эти операции проставляют единичный или нулевой бит атомарно. Во время непосредственно выполнения функций $bit никакие

другие процессы не могут изменить значение переменной в то же самое время. При откате транзакции операции, выполненные функциями $bit, также откатываются атомарно, не затрагивая другие биты.

К интересным особенностям алгоритма поддержки bit-slice индексов можно отнести то, что обе системы, и Caché и MiniM, при отсутствии установки бита явным образом используют эту позицию как нулевой бит. Эквивалент нулевого бита используется как при дополнении битовой строки, так и при чтении бита за ее пределами. Поэтому, если разделить операции обновления индексов на три вида

1. При добавлении новой записи.
2. При изменении записи.
3. При удалении записи.

то, несмотря на то, что сама операция простановки битов должна распространяться на все биты получаемые по значению атрибута, при добавлении новой записи нет необходимости проставлять нулевые биты, сократив таким образом число дисковых операций, или операций с глобалом. Даже если произойдет операция отката транзакции, то система возвращает значение бита из явно заданного нулевого значения в такое же значение, эквивалентное нулевому биту. При обновлении или удалении записи необходимо менять лишь те биты, которые отличны от имевшихся, неважно было ли значение бита 0 или 1. Такие особенности обновления bit-slice индексов могут заметно сократить время обновления индекса относительно прямолинейного алгоритма полного обновления всех битов.

3.10 Нормирование значений

Для индексирования данных важным моментом является понимание чувствительности атрибутов к регистру.

Ключевой операцией для использования индекса является обращение к искомому значению в глобал или локаль, и поиск выполняет непосредственно СУБД, поскольку значения индексов в MUMPS хранятся сортированно.

В обычных случаях используются значения атрибутов как есть. При этом, для того, чтобы найти необходимую запись в индексе, нужно чтобы она существовала в точности в том же виде. Если мы не предпринимаем

никаких дополнительных мер, то такой индекс будет чувствителен к регистру. Или, иными словами, сравнение будет выполняться "как есть".

При необходимости искать по индексу нечувствительно к регистру мы должны оба значения - и искомое, и записанное в индексе, привести к единым соглашениям по регистру символов.

Традиционно, большинство систем баз данных использует приведение строк к верхнему регистру, но это необязательное правило.

Одна из проблем, вытекающих из построения регистронезависимого поиска - это то, что после приведения к единому регистру несколько различных значений могут стать равны, например:

Исходное	Нормированное
Шар	ШАР
шар	ШАР
ШАР	ШАР

и, в этом случае, при необходимости использовать такой индекс для поиска чувствительно к регистру, необходимо применять дополнительную операцию фильтрации. В ней нужно проверить равенство искомого атрибута атрибуту найденной записи.

Чувствительность к регистру является, вообще говоря, частным примером общей нормировки значений. При внесении данных в базу данных могут нормироваться как сами значения атрибутов, так и их индексные значения. Вот несколько распространенных примеров нормирования значений атрибутов:

1. Приведение к единому регистру
2. Отбрасывание лидирующих и завершающих пробельных символов
3. Замена последовательности пробельных символов на один пробел
4. Удаление непечатных символов
5. Капитализация названий

Разработчик должен самостоятельно решить, на каком из этапов должна производиться нормировка значений - при изменении значения атрибута или при вычислении индексного значения. Очевидно, что применение нормировки, если она использовалась хотя бы раз, далее всегда необходимо при обращении к индексу.

В случае применения нормированных значений индексов для уникального индекса разработчик должен принять соответствующие меры

для разрешения конфликта - что является для базы данных уникальным - оригинальное значение атрибута или его нормированное представление. В такой ситуации традиционным выбором разработчиков является опора на оригинальное значение, и ведение в индексных записях в действительности неуникальных записей, и обеспечение уникальности не структурно, а алгоритмически. В любом случае, у разработчиков обычно есть выбор, что является более важным критерием - ведение уникальности собственно индекса или уникальности значений записей при сохранении нормированности.

3.11 Выборки по индексу

Выборкой по индексу (возможно, термин не совсем соответствует русскому языку, но так он употребляется очень часто в среде разработчиков) называется использование дополнительно поддерживаемых индексных структур для получения данных либо непосредственно, либо косвенно в сочетании с самими записями данных. Собственно говоря, именно эта операция и является целью применения параллельных структур данных.

По своему строению индексные структуры дают возможность тем или иным способом указав искомое значение атрибута получить ноль, одно или несколько идентификаторов записей или значения других атрибутов записей. Индексные записи отображают индексируемое значение на один или несколько идентификаторов. Поэтому операции выборки могут быть нескольких видов:

1. Проверка существования в индексе заданного значения атрибута.
2. Выборка значений атрибутов для заданного диапазона значений атрибута.
3. Выборка значений идентификаторов для заданного значения атрибута.
4. Выборка значений идентификаторов для заданного диапазона значений атрибута.

Структурно строение индексных записей схематично выглядит как:

```
^Ind("IndName",IndValue,id)=""
```

При этом первой задаче соответствует применение операции $d() для проверки существования узла, второй соответствует применение $o()

для выборки значений IndValue, третьей - операция $o() для выборки набора id при фиксировании значения IndValue и для четвертой - либо операция $q() либо два вложенных друг в друга цикла - один по IndValue, второй - по id.

Для демонстрации выборок используем структуру хранения простого индекса из первого примера.

Выборка диапазона значений атрибута

```
SelectFigures(from="",to="")
 i from'="" s from=$o(^Index("Figure",from),-1)
 i to'="" s to=$o(^Index("Figure",to))
 f  s from=$o(^Index("Figure",from)) q:(from=to)!(from="") d
 . w from,!
 q
```

Здесь функция SelectFigures выбирает различные значения атрибута Figure от from до to включительно. Если аргумент опущен, то считается пустой строкой и производится выборка с открытым концом: если from не указан, то выдаются все с начала, если to не указан, то выдаются все до конца. В реализации функция корректирует значения from и to для того чтобы цикл for был применим к всем комбинациям.

Выборка идентификаторов по значению атрибута

```
SelectByFigure(Figure)
 n id s id=""
 f  s id=$o(^Index("Figure",Figure,id)) q:id=""  d
 . w id,!
 q
```

Выборка значений идентификаторов для заданного диапазона значений атрибута с использованием вложенного цикла:

```
SelectByFiguresl(from="",to="")
 n id
 i from'="" s from=$o(^Index("Figure",from),-1)
 i to'="" s to=$o(^Index("Figure",to))
 f  s from=$o(^Index("Figure",from)) q:from=to  d
 . s id="" f  s id=$o(^Index("Figure",from,id)) q:id=""  d
 . . w id,!
 q
```

Выборка значений идентификаторов для заданного диапазона значений атрибута с использованием итераций по $q:

```
SelectByFigures2(from="",to="")
 n ref
 s ref=$na(^Index("Figure",from))
 f  s ref=$q(@ref) q:$s(to="":ref="",1:$qs(ref,2)]]to)  d
 . w $qs(ref,3),!
 q
```

Здесь в качестве условия прекращения цикла стоит сложное условие: в зависимости от непустоты хвоста to либо продолжать пока ссылка не станет пустой, либо учитывать порядок индексной сортировки.

Во всех приведенных случаях есть возможность указывать направление выдачи результата - по возрастанию или убыванию значений атрибутов и / или идентификаторов. Для этого можно либо явно указать в необязательном аргументе функций $o и $q направление траверса либо передавать его из необязательного параметра функций SelectBy... Приведенные функции выдают в направлении возрастания значений в отношении индексной сортировки. Для полноты и строгости изложения также следует отметить, что возможность задать направление выборки для функции $query может поддерживаться не всеми MUMPS системами, для которых разрабатывается программа, и такую возможность необходимо проверить по документации. В частности, система GT.M такой параметр не поддерживает.

В третьем случае, при выборке идентификаторов при наложении ограничений на диапазон значений атрибутов, идентификаторы выдаются не в порядке сортировки чисел, а в порядке сортировки атрибутов. Для выдачи идентификаторов также в сортированном виде следует применить дополнительную сортировку, хотя обычно в упорядочении значений идентификаторов нет особого смысла кроме соответствия порядку создания записи.

Приведенные случаи рассчитаны на выборку включительно указанные значения. Во многих случаях требуется выдавать значения со строгими неравенствами. Для этого нужно соответственно модифицировать приведенные функции.

Примечание - в англоязычной литературе и справочниках описанные выборки могут называться так: 2, 4 - range scan, 1, 3 - index scan.

Механизм выборки данных структурно обычно выполняют одним из двух видов:

1. Итератор непосредственно применяет операцию к полученному идентификатору

2. Итератор записывает найденный идентификатор в набор для возврата

Использование первого способа продемонстрировано на приведенных выше примерах выборки данных. Команда write выводит найденные данные непосредственно на экран.

Второй способ используется в библиотечных или обобщенных функциях. Итерации по данным выполняются по тем же самым алгоритмам, но вместо применения какой-либо операции к найденным данным эти данные записываются в указаную переменную.

В простых случаях переменная для получения набора найденных данных может передаваться по ссылке и использоваться локальная переменная. В общем же случае используются временные глобальные переменные. Для формирования имени используют, например, номер текущего процесса, внутренний идентификатор выборки или что-то еще.

Для демонстрации примера выборки в указанную переменную:

```
indselect ;
 n Search
 s Search=$na(^TempResult($j))
 d SelectByFigure("квадрат",Search)
 w "Результат поиска:",!
 zw @Search
 k @Search
 q
SelectByFigure(Figure,Result)
 n id s id=""
 k @Result
 f  s id=$o(^Index("Figure",Figure,id)) q:id=""  d
 . s @Result@(id)=""
 q
```

В некоторых из примеров используется первый способ, для простой демонстрации, а в более сложных случаях, или для выполнения операций над результатами поиска, используется второй вариант.

Использование временных глобальных переменных обычно предпочтительнее из-за возможно большого объема данных, попадающих в выборку. В тех случаях, когда по самой формулировке задачи объем выборки небольшой, то также применяются и локальные переменные.

3.12 Многоиндексная выборка (zig-zag)

Многоиндексной выборкой, иначе называемой шаговой или зиг-загом, называется выборки идентификаторов записей по нескольким индексным структурам одновременно. Также часто встречается название zig-zag ordered scan. Принципиальным моментом является слово ordered,

или упорядочение искомых идентификаторов. Применяется для выборки из двух или более индексов.

Индексные структуры отображают значение атрибута на набор идентификаторов. При этом набор идентификаторов может быть упорядочен в каком-либо упорядочении. Если порядок сортировки в нескольких индексах одинаков, то к ним может быть применена многоиндексная выборка.

При такой выборке алгоритм выбирает по какому-то критерию один из индексов, например первый попавшийся и получает идентификатор записи. После чего идет по списку индексов и сверяет есть ли в них соответствующая запись с соответствующими этим индексам значениями атрибутов. Если во всех заданных индексах такая запись есть, то она считается найденной. Если по какому-то из индексов запись не найдена, то по этому индексу выполняется сдвиг в наборе идентификаторов и этот индекс считается начальным, все повторяется. Если на каком-то шаге идентификатор стал пуст, то выборка закончена.

Приведенный алгоритм является одним из вариантов алгоритма соединения сортированных списков, называемый merge join или sort merge join, и может объединять, вообще говоря, произвольное число индексных структур.

Условно представим схему выборки. Положим, что нужно найти серую кошку. При этом располагаем двумя индексами - по цвету и по виду.

```
^Index("color","белый",1)=""
^Index("color","белый",2)=""
^Index("color","серый",3)=""
^Index("color","серый",4)=""

^Index("type","кошка",1)=""
^Index("type","кошка",4)=""
^Index("type","кошка",5)=""
^Index("type","собака",2)=""
```

В качестве начального условия выбираем индекс по цвету и позиционируемся на первое значение идентификаторов 3. Получаем в переменную id = 3. Переходим к индексу по виду и проверяем есть ли такой идентификатор в отображении значения "кошка". Такого нет. Делаем текущим индекс по виду и выполняем выборку следующего идентификатора по виду относительно текущего значения id, id принимает значение 4.

Поскольку идентификатор не пуст, не прекращаем поиск и проверяем существование отображения следующего индекса (это индекс по цвету).

Такая запись есть, и других индексов для проверки нет, поэтому идентификатор считается удовлетворяющим заданным условиям и отмечается каким-либо способом как найденный.

С этого же индекса (или с любого другого) выполняем сдвиг на следующий идентификатор. Сдвинувшись по индексу по цвету на следующий идентификатор после 4, получаем пустой идентификатор, следовательно поиск закончен.

В найденный набор идентификаторов таким образом должен попасть только один идентификатор id = 4.

Обобщенная функция многоиндексной выборки может выглядеть например таким образом:

```
AND(ret,names...)
 n id,i,j,place
 ; идем по всем и запоминаем. контекст выборки - в переменных
 ; id -         текущий идентификатор
 ; i - номер индекса в наборе индексов
 ; j - внутренная переменная прохода по набору индексов
 ; place - внутренняя переменная
 s id="" f  s i=1,id=$$ANDnext() q:id=""  s @ret@(id)=""
 q
ANDnext()
ANDrep
 s id=$o(@names(i)@(id))
 q:id="" ""  ; кончилось хотя бы по одному - кончилось совсем
 ; и идем по всем кроме полученного и проверяем существование
 ; если хотя бы один не существует,
 ; то с него делаем следующий шаг
 f j=i+1:1:i+names-1 s place=((j-1)#names)+1 »
       i '$d(@names(place)@(id)) s i=place g ANDrep
 q id ; по всем есть, значит подходит
```

Здесь символом » обозначен перенос строки, которого в реальном коде не должно быть. В функции AND используется передача переменного числа параметров. То же самое можно организовать и на стандартном M:

```
ANDv(ret,names) g AND+1
```

Но при этом следует самостоятельно сформировать набор индексов. Приведенные функции используют соглашения что в передаваемых переменных передаются имена и функции должны использовать косвенность. Это соглашение позволяет составить функцию выборки по индексам в достаточно общей форме.

Приведем примеры получения данных в соглашениях из первого примера:

```
SelectFigureColor(Figure,Color)
 n res
 d AND($na(res),$na(^Index("Figure",Figure)), »
     $na(^Index("Color",Color)))
 s res="" f  s res=$o(res(res)) q:res=""  d
 . w res,!
 q
```

Здесь формируется набор имен, куда следует вренуть результат и имена индексных записей. Тот же вариант без использования переменного числа аргументов:

```
SelectFigureColor(Figure,Color)
 n res,ind
 s ind(1)=$na(^Index("Figure",Figure))
 s ind(2)=$na(^Index("Color",Color))
 s ind=2  ; два индекса
 d ANDv($na(res),.ind)
 s res="" f  s res=$o(res(res)) q:res=""  d
 . w res,!
 q
```

Многоиндексная выборка по своему характеру не является наиболее оптимальной по числу выполняемых с глобалами действий, поскольку часть шагов выполняется впустую. Количество лишних операций, или операций не приведших к получению искомого идентификатора, сильно зависит от состояния используемых индексов. Вполне возможны ситуации, когда при большом количестве данных по обеим индексам в искомое множество попадает очень малое число идентификаторов, но в этом случае многоиндексная выборка тем не менее предпримет попытки прохода по неинтересующим идентификаторам в том числе.

Одновременно с тем в большинстве случаев многоиндексная выборка намного предпочтительнее использования только одного индекса и фильтрации значений по остальным атрибутам. В определенной степени это утверждение опирается на наиболее распространенные статистики применяемых данных. Соотношение эффективности различных методов поиска по нескольким индексам, безусловно, зависит от состояния данных.

Многоиндексную выборку можно применять также при смешанных индексах - одновременно использовать древовидные, составные и битовые индексы. Условием их совместности является одинаковость упорядочения искомых идентификаторов. В случае применения индексов разного вида соответственно следует изменить алгоритм выборки, чтобы он обращался к соответствующим операциям получения следующего идентификатора и для проверки существования идентификатора в индексных

структурах. Вышеприведенные коды использовали простые индексы и соответственно использовали функции $o() и $d().

Замечание относительно одинаковости упорядочения важно также в отношении применения баз данных с различными соглашениями о сравнении символов (character collation), а также для тех MUMPS систем, в которых сортировка локальных переменных может отличаться от сортировки глобалов в случае если производится выборка из индексов или промежуточных индексных структур, расположенных в смешанном виде хранения - и в локальных и в глобальных переменных одновременно.

3.13 Дифференциальное индексирование

Дифференциальным индексированием называется такое перестроение индексных элементов, при котором изменению подвергаются индексные элементы только для тех атрибутов записи, которые изменились. Введение дифференциального индексирования является оптимизирующей операцией, поскольку в большинстве случаев позволяет уменьшить количество операций с диском, не изменяя саму логику работы и структуру данных. В большинстве случаев записи данных действительно редко меняются целиком, зачастую меняется лишь часть атрибутов.

Введение дифференциального индексирования принципиально сказывается только в одном месте кода поддержки индексов - при обновлении записи. При вставке новой и при удалении записи данных индексные записи должны быть строго либо добавлены либо удалены соответственно. Только при обновлении записи есть возможность сверить значения атрибутов в текущем значении записи и в новом значении. Сравнение делается бесхитростно, повальным сканированием всех атрибутов попадающих под индексирование, либо используется информация от функции, изменяющей значение атрибута.

На примере ведения простого индекса покажем изменения кода, которые нужно сделать для введения дифференциального перестроения индекса.

```
CreateRecords()
 k ^Index
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
 s Counts="2~5~12~8"
 f i=1:1:12 d
 . s Figure=$p(Figures,"~",$r(4)+1)
```

```
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
DeleteRecord(id)
 q:'$d(^Data(id))
 l +^Data(id)
 n RecordValues s RecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(id)
 l -^Data(id)
 q
 ; эта функция начинает различать
 ; предыдущие и новые значения атрибутов
UpdateRecord(id,RecordValues)
 q:'$d(^Data(id))
 l +^Data(id)
 n OldRecordValues s OldRecordValues=$g(^Data(id))
 n Figure,Color,Count,FigureOld,ColorOld,CountOld
 ; check Figure
 s FigureOld=$p((OldRecordValues),"~",1)
 s Figure=$p((RecordValues),"~",1)
 i Figure'=FigureOld d
 . d DeleteIndexRecord("Figure",id,FigureOld)
 . d InsertIndexRecord("Figure",id,Figure)
 ; check Color
 s ColorOld=$p((OldRecordValues),"~",2)
 s Color=$p((RecordValues),"~",2)
 i Color'=ColorOld d
 . d DeleteIndexRecord("Color",id,ColorOld)
 . d InsertIndexRecord("Color",id,Color)
 ; check Count
 s CountOld=$p((OldRecordValues),"~",3)
 s Count=$p((RecordValues),"~",3)
 i Count'=CountOld d
 . d DeleteIndexRecord("Count",id,CountOld)
 . d InsertIndexRecord("Count",id,Count)
 s ^Data(id)=RecordValues
 l -^Data(id)
 q
DeleteIndex(IndexName)
 k ^Index(IndexName)
```

```
 q
InsertIndexRecords(id,RecordValues)
 d InsertIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d InsertIndexRecord("Color",id,$p((RecordValues),"~",2))
 d InsertIndexRecord("Count",id,$p((RecordValues),"~",3))
 q
DeleteIndexRecords(id,RecordValues)
 d DeleteIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d DeleteIndexRecord("Color",id,$p((RecordValues),"~",2))
 d DeleteIndexRecord("Count",id,$p((RecordValues),"~",3))
 q
InsertIndexRecord(IndexName,id,Value)
 l +^Index(IndexName,Value,id)
 s ^Index(IndexName,Value,id)=""
 l -^Index(IndexName,Value,id)
 q
DeleteIndexRecord(IndexName,id,Value)
 l +^Index(IndexName,Value,id)
 k ^Index(IndexName,Value,id)
 l -^Index(IndexName,Value,id)
 q
```

В приведенном примере код UpdateRecord явно содержит перечень имен всех свойств и способ получения их значений. В структуре алгоритма дифференциального перестроения индексов явно просматривается возможность написания более обобщенного кода. В более общем случае имеет смысл применять обобщенные функции вроде

```
GetAttrCount(rectype)
GetAttrName(rectype,attrnumber)
GetAttrValue(rec,attrnumber)
GetRecType(rec)
IsIndexed(rectype,attrnumber)
```

То есть тем или иным образом ввести а-ля объектный подход и информацию о метаданных. Это позволит реализовать код индексирования для общего случая и применять его для множества различных структур.

В целом, введение дифференциального перестроения индекса может приводить к уменьшению накладных расходов на индексирование в разы по сравнению с неоптимизированным обновлением индексных записей "в лоб". Поскольку данное усовершенствование относится к оптимизации, его стоит отнести на последние этапы разработки.

3.14 Индексация длинных атрибутов

В реализации любых M систем в целях повышения эффективности реализаций вводятся ограничения на длину индекса. Ограничения такого же характера присутствуют и в других, не-M реализациях СУБД. В распространенных реализациях M длина индексов включая имя переменной ограничена 255 байт.

В случае применения сложных структур индексирования, составных индексов или просто длинных строковых атрибутов возникает задача выполнить индексацию в указанных ограничениях. Для решения этой задачи может быть применено сегментирование значений атрибутов. В индексе вместо отображения значения атрибута на набор идентификаторов прописывается отображение сегментов атрибута на набор идентификаторов.

Получить сегменты можно например так:

```
f i=0:1:$l(value)/n s @ind@(i+1)=$e(value,i*n+1,i+1*n)
```

Здесь n - число символов в одном сегменте.

При использовании сегментирования значения атрибута поиск выполняется указанием значений сегментов. При этом может возникнуть ситуация поиска по значениям с длиной, кратной величине сегмента. При этом возможно появление ситуации нахождения лишних записей, имеющих первую часть значения атрибута совпадающего с искомым. Чтобы исключить появление такой ситуации вводится дополнительная структура - индекс на длину атрибута. И либо длина искомого фрагмента должна обязательно указываться при поиске, либо необходимо использовать полный набор всех сегментов, которые могут принадлежать используемому значению.

После получения сегментов для каждого из них прописывается отображение сегмента на идентификатор. Например, используя модифицированный пример с простым индексом, для демонстрации используя сегменты длиной по 12 символов:

```
CreateRecords() ; k  d CreateRecords^ind09() w
k ^Index
k ^Data
n i,Figures,Colors,Counts,Figure,Color,Count,id
s Figures="квадрат с такими разными и неровными »
краями~круг тоже не очень ровный~отрезок ну очень »
прямой, но только жаль что в плоскости~треугольник »
вообще какой-то очень правильный"
s Colors="красный~зелёный~синий~белый"
```

```
 s Counts="2~5~12~8"
 f i=1:1:12 d
 . s Figure=$p(Figures,"~",$r(4)+1)
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
DeleteRecord(id)
 q:'$d(^Data(id))
 l +^Data(id)
 n RecordValues s RecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(id)
 l -^Data(id)
 q
UpdateRecord(id,RecordValues)
 q:'$d(^Data(id))
 l +^Data(id)
 n OldRecordValues s OldRecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,OldRecordValues)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q
DeleteIndex(IndexName)
 k ^Index(IndexName)
 q
InsertIndexRecords(id,RecordValues)
 d InsertIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d InsertIndexRecord("Color",id,$p((RecordValues),"~",2))
 d InsertIndexRecord("Count",id,$p((RecordValues),"~",3))
 q
DeleteIndexRecords(id,RecordValues)
 d DeleteIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d DeleteIndexRecord("Color",id,$p((RecordValues),"~",2))
 d DeleteIndexRecord("Count",id,$p((RecordValues),"~",3))
 q
InsertIndexRecord(IndexName,id,Value)
 l +^Index(IndexName,id)
 n i,segment,n
 s n=12
 f i=0:1:$l(Value)/n d
```

```
 . s segment=$e(Value,i*n+1,i+1*n)
 . s:segment'="" ^Index(IndexName,i+1,segment,id)=""
 s ^Index(IndexName_" size",$l(Value)+1,id)=""
 l -^Index(IndexName,id)
 q
DeleteIndexRecord(IndexName,id,Value)
 l +^Index(IndexName,id)
 n i,segment,n
 s n=12
 f i=0:1:$l(Value)/n d
 . s segment=$e(Value,i*n+1,i+1*n)
 . k:segment'="" ^Index(IndexName,i+1,segment,id)
 k ^Index(IndexName_" size",$l(Value)+1,id)
 l -^Index(IndexName,id)
 q
```

Здесь символом » обозначен перенос строки, которого в реальном коде не должно быть.

В результате получаем индексные записи с сегментированными значениями атрибутов. Обратим внимание, что можно упростить схему блокировок, используя более короткое имя блокировки - означающее не столько блокируемую переменную, сколько смысл операции - в данном индексе идет операция с данным идентификатором.

Для выборки данных из индекса следует, конечно, учитывать, что атрибуты сегментированы и искать следует те идентификаторы, для которых полностью совпадут все сегменты. Примерный код выборки из одного сегментированного индекса, использующий многоиндексную выборку, может быть таким:

```
 Select()
 n Figure,ind,i,n,segment,res s n=12
 s Figure="квадрат с такими разными и неровными краями"
 f i=0:1:$l(Figure)/n d
 . s segment=$e(Figure,i*n+1,i+1*n)
 . s:segment'="" ind(i+1)=$na(^Index("Figure",i+1,segment))
 s ind(i+1)=$na(^Index("Figure size",$l(Figure)+1))
 s ind=$l(Figure)/n+1
 ; используем zig-zag ordered scan
 d ANDv($na(res),.ind)
 s res="" f  s res=$o(res(res)) q:res=""  d
 . w res,!
 q
ANDv(ret,names) g AND+1
AND(ret,names...)
 n id,i,j,place
 ; идем по всем и запоминаем. контекст выборки - в переменных
 ; id -        текущий идентификатор
```

```
 ; i - номер индекса в наборе индексов
 ; j - внутренная переменная прохода по набору индексов
 ; place - внутренняя переменная
 s id="" f  s i=1,id=$$ANDnext() q:id=""  s @ret@(id)=""
 q
ANDnext()
ANDrep
 s id=$o(@names(i)@(id))
 q:id="" ""  ; кончилось хотя бы по одному - кончилось совсем
 ; и идем по всем кроме полученного и проверяем существование
 ; если хотя бы один не существует, то с него делаем следующий шаг
 f j=i+1:1:i+names-1 s place=((j-1)#names)+1 »
        i '$d(@names(place)@(id)) s i=place q ANDrep
 q id ; по всем есть, значит подходит
```

Применение сегментирования значений атрибутов может быть вызвано не только длинными значениями самих атрибутов, но и служебными значениями в индексе, также занимающими место.

Такой метод может быть применен также для поиска записей по блобам, поскольку теоретически величина атрибута не ограничена. Поиск выполним так же, как при использовании атомарного строкового атрибута - при точном совпадении значения атрибута.

3.15 Межтабличный индекс

Межтабличным индексом называется индексная структура, объединяющая в себе данные не для одного, а для двух или более логических наборов данных. Записи в межтабличном индексе перестраиваются при добавлении, изменении и удалении записей в наборах записей, данные из которых входят в этот индекс. Основным назначением межтабличного индекса является сокращение операций соединения таблиц.

Рассмотрим структуру данных вида Отдел - Подразделение - Сотрудник. Та сущность что правее в этой цепочке входит в ту что левее. Пусть у каждой сущности есть два атрибута - идентификатор записи и название, или ФИО для сотрудника. Схема существенно упрощенная по отношению к реальной, но приведена исключительно в демонстрационных целях. Сущности Подразделение и Сотрудник имеют соответственно дополнительные атрибуты - ссылки на Отдел и Подразделение. Структурно схема данных может быть обозначена так:

```
Отдел
^Data("Otdel",id)=$lb(Name)
Подразделение
```

```
^Data("Podr",id)=$lb(Name,IDOtdel)
Сотрудник
^Data("Sotr",id)=$lb(Name,IDPodr)
```

Эта схема существенно нормализована, не содержит избыточной информации, и значения атрибутов зависят только от идентификаторов строк. По приведенным данным можно выполнить любые операции - добавления, удаления и изменения. Рассмотрим две тяжелых операции - вывести фамилии сотрудников для заданного отдела и вывести наименование отдела для заданного сотрудника. Обе задачи решаются пошаговым прохождением через промежуточные структуры подразделения. Межтабличные индексы могут решить задачу ускорения вывода. В нашем случае индексов будет два - по одному для каждой из задач.

```
 CreateRecords()
 k ^Data
 n id
 ; сделаем отделы
 f id=1:1:12 d InsertOtdel("Отдел "_id)
 ; сделаем подразделения
 f id=1:1:30 d InsertPodr("Подразделение "_id,$r(12)+1)
 ; сделаем сотрудников
 f id=1:1:300 d InsertSotr("Сотрудник "_id,$r(30)+1)
 q
InsertOtdel(Name)
 n id
 s id=$i(^Data("Otdel"))
 s ^Data("Otdel",id)=$lb(Name)
 q:$Q id  q
InsertPodr(Name,fk)
 n id
 s id=$i(^Data("Podr"))
 s ^Data("Podr",id)=$lb(Name,fk)
 q:$Q id  q
InsertSotr(Name,fk)
 n id
 s id=$i(^Data("Sotr"))
 s ^Data("Sotr",id)=$lb(Name,fk)
 q:$Q id  q
```

Здесь при создании записи сотрудника нам становится известным полный путь от сотрудника к отделу. На самом деле это очень серьезное соглашение - может ли быть запись которая ни на что не ссылается. В нашем примере пусть будет нельзя. Также пусть будет нельзя удалять те записи, на которые имеются ссылки. В нашем примере полный путь между сотрудником и отделом определяется, таким образом, в момент создания записи о сотруднике, при удалении записи о сотруднике, а также

при изменениях внешних ссылок подразделения на отдел и сотрудника на подразделение.

Структура поддерживаемого индекса будет фактически так или иначе отражать путь в условном графе принадлежности объектов одного другому:

```
Для получения сотрудников по отделу
^Data("ind1",otdelid,podrid,sotrid)=""
Для получения отдела по сотруднику
^Data("ind2",sotrid,podrid,otdelid)=""
```

Пусть в нашей схеме данных поддерживается строгая дисциплина ссылочности: не может быть подразделения без отдела и сотрудника без подразделения. Исходя из этой упрощенной дисциплины значения в индексах меняются при

1. при создании сотрудника
2. при удалении сотрудника
3. при изменение ссылочного значения сотрудника на подразделение
4. при изменении ссылочного значения подразделения на отдел

Соответственно для этих событий вводим функции поддержания индекса:

```
 indAddSotr(idsotr)
 n idotdel,idpodr
 s idpodr=$li(^Data("Sotr",idsotr),2)
 s idotdel=$li(^Data("Podr",idpodr),2)
 s ^Data("ind1",idotdel,idpodr,idsotr)=""
 s ^Data("ind2",idsotr,idpodr,idotdel)=""
 q
indDelSotr(idsotr)
 n idotdel,idpodr
 s idpodr=$li(^Data("Sotr",idsotr),2)
 s idotdel=$li(^Data("Podr",idpodr),2)
 k ^Data("ind1",idotdel,idpodr,idsotr)
 k ^Data("ind2",idsotr,idpodr,idotdel)
 q
indChangeSotrFK(idsotr,newpodr)
 n oldpodr,oldotdel,newotdel
 s oldpodr=$li(^Data("Sotr",idsotr),2)
 s oldotdel=$li(^Data("Podr",oldpodr),2)
 s newotdel=$li(^Data("Podr",newpodr),2)
 k ^Data("ind1",oldotdel,oldpodr,idsotr)
```

```
 s ^Data("ind1",newotdel,newpodr,idsotr)=""
 k ^Data("ind2",idsotr,oldpodr,oldotdel)
 s ^Data("ind2",idsotr,newpodr,newotdel)=""
 q
indChangePodrFK(idpodr,newotdel)
 n oldotdel,idsotr
 s oldotdel=$li(^Data("Podr",idpodr),2)
 m ^Data("ind1",newotdel,idpodr)=^Data("ind1",oldotdel,idpodr)
 k ^Data("ind1",oldotdel,idpodr)
 s idsotr=""
 f  s idsotr=$o(^Data("ind1",newotdel, »
       idpodr,idsotr)) q:idsotr=""  d
 . k ^Data("ind2",idsotr,idpodr,oldotdel)
 . s ^Data("ind2",idsotr,idpodr,newotdel)=""
 q
```

Приведенная схема отображения отдела на сотрудников (индекс ind1) использует составной индекс. Он неудобен для многоиндексной выборки, поскольку отображает соответственно отдел на неупорядоченный набор сотрудников. Для индекса ind2 отображение единственно. Для того, чтобы можно было применять многоиндексную выборку сотрудников лучше ввести дополнительный индекс, отображающий отдел на упорядоченный набор сотрудников:

```
^Data("ind3",otdelid,sotrid)=""

 indAddSotr(idsotr)
 n idotdel,idpodr
 s idpodr=$li(^Data("Sotr",idsotr),2)
 s idotdel=$li(^Data("Podr",idpodr),2)
 s ^Data("ind1",idotdel,idpodr,idsotr)=""
 s ^Data("ind3",idotdel,idsotr)=""
 s ^Data("ind2",idsotr,idpodr,idotdel)=""
 q
indDelSotr(idsotr)
 n idotdel,idpodr
 s idpodr=$li(^Data("Sotr",idsotr),2)
 s idotdel=$li(^Data("Podr",idpodr),2)
 k ^Data("ind1",idotdel,idpodr,idsotr)
 k ^Data("ind3",idotdel,idsotr)
 k ^Data("ind2",idsotr,idpodr,idotdel)
 q
indChangeSotrFK(idsotr,newpodr)
 n oldpodr,oldotdel,newotdel
 s oldpodr=$li(^Data("Sotr",idsotr),2)
 s oldotdel=$li(^Data("Podr",oldpodr),2)
 s newotdel=$li(^Data("Podr",newpodr),2)
 k ^Data("ind1",oldotdel,oldpodr,idsotr)
```

```
 k ^Data("ind3",oldotdel,idsotr)
 s ^Data("ind1",newotdel,newpodr,idsotr)=""
 s ^Data("ind3",newotdel,idsotr)=""
 k ^Data("ind2",idsotr,oldpodr,oldotdel)
 s ^Data("ind2",idsotr,newpodr,newotdel)=""
 q
indChangePodrFK(idpodr,newotdel)
 n oldotdel,idsotr
 s oldotdel=$li(^Data("Podr",idpodr),2)
 m ^Data("ind1",newotdel,idpodr)=^Data("ind1",oldotdel,idpodr)
 k ^Data("ind1",oldotdel,idpodr)
 s idsotr=""
 f  s idsotr=$o(^Data("ind1",newotdel, »
       idpodr,idsotr)) q:idsotr=""  d
 . k ^Data("ind2",idsotr,idpodr,oldotdel)
 . s ^Data("ind2",idsotr,idpodr,newotdel)=""
 . k ^Data("ind3",oldotdel,idsotr)
 . s ^Data("ind3",newotdel,idsotr)=""
 q
```

В случае применения межтабличных индексов следует внимательно пересмотреть политику отношения к ссылкам, а также степень востребованности запросов "через голову", требующих без межтабличных индексов долгих соединений. Возможно, что негативный фактор усложнения операции добавления / удаления и увеличение объема хранения может перевесить выгоды от эпизодического использования прямого индекса между таблицами. В случае использования в индексе идентификаторов записей операции обновления индексов по причине смены внешней ссылки в одной из промежуточных таблиц будут происходить нечасто, поэтому алгоритмическая нагрузка может быть совершенно незначительной.

Применение межтабличных индексов видится чрезвычайно оправданным для построения ROLAP хранилищ данных по схеме сильно разветвленной звезды с частично нормализованными данными. По сути, применение именно таких индексов и упрощает структуру схемы для операции выборки данных, оставляя сами данные нормализованными.

3.16 Индекс с условием на вставку

При поддержке индексных структур параллельно со структурами данных может быть использовано хранение не всех индексных записей, а лишь некоторых. Механизм поддержки индексов может использовать некоторое заданное условие для того, чтобы определить, следует ли вставлять индексную запись или нет.

При применении этого механизна следует обращать внимание, зависит ли условие вставки индексной записи только от значений атрибутов или также от иных факторов. Механизм удаления и обновления индексной записи может в первом случае использовать то же самое условие, либо производить удаление индексной информации всегда и независимо от результата вычисления условия.

Использование условной вставки индексной записи приводит к тому, что в индексы попадает информация не о всех записях данных. Этот факт может быть использован при выборке по условному индексу - при выборке мы автоматически получаем только те записи, которые удовлетворяют заданному условию. Этот метод может оказаться во много раз эффективнее применения сложной индексной структуры или сложных алгоритмов выборки по нескольким индексам.

Кроме простой реализации достаточно сложного условия можно отметить другие плюсы условной вставки индексов, такие как уменьшение объемов журналирования, ресурсов кеширования и дисковых операций.

Приведем пример условной вставки индексных записей. Здесь индексная запись по атрибуту Figure вставляется только если выполняется условие. При этом в условии проверяется что значение атрибута Count больше 1. При использовании индекса по атрибуту Figure мы автоматически получаем выборку с дополнительным условием Count > 1 - в выборку попадут только записи попадающие под это условие.

```
 CreateRecords()
 k ^Index
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
 s Counts="2~5~12~8"
 f i=1:1:12 d
 . s Figure=$p(Figures,"~",$r(4)+1)
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
DeleteRecord(id)
 q:'$d(^Data(id))
```

```
 l +^Data(id)
 n RecordValues s RecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(id)
 l -^Data(id)
 q
UpdateRecord(id,RecordValues)
 q:'$d(^Data(id))
 l +^Data(id)
 n OldRecordValues s OldRecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,OldRecordValues)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q
DeleteIndex(IndexName)
 k ^Index(IndexName)
 q
Condition(RecordValues)
 n Count
 s Count=$p(RecordValues,"~",3)
 q Count>1
InsertIndexRecords(id,RecordValues)
 i $$Condition(RecordValues) d
 . d InsertIndexRecord("Figure",id,$p(RecordValues,"~",1))
 d InsertIndexRecord("Color",id,$p(RecordValues,"~",2))
 d InsertIndexRecord("Count",id,$p(RecordValues,"~",3))
 q
DeleteIndexRecords(id,RecordValues)
 d DeleteIndexRecord("Figure",id,$p(RecordValues,"~",1))
 d DeleteIndexRecord("Color",id,$p(RecordValues,"~",2))
 d DeleteIndexRecord("Count",id,$p(RecordValues,"~",3))
 q
InsertIndexRecord(IndexName,id,Value)
 s ^Index(IndexName,Value,id)=""
 q
DeleteIndexRecord(IndexName,id,Value)
 k ^Index(IndexName,Value,id)
 q
```

Здесь мы при удалении индексной записи не проверяем условие, поскольку для выполнения команды kill не требуется существования соответствующего узла.

При использовании условной вставки может поддерживаться несколько индексов по одному атрибуту с различными условиями вставки. Таким образом, может быть реализован механизм очень простой выборки с несколькими сложными условиями. Технически оказывается проще сформулировать условие отбора в виде отдельной функции и использо-

вать ее при вставке индексных записей, чем то же самое условие описывать в виде алгоритма выборки. В частности, в условие вставки хорошо вынести тяжелые для оптимизации элементы, например, операцию логического ИЛИ.

Как видно, наличие условия на вставку индексной записи никак не связано со структурой индекса - условие может быть комбинировано с любым типом индекса.

Применение индексов с условием на вставку, в действительности, исходя из характеристик таких индексов - это довольно тонкая задача. Если при проектировании системы вообще сначала неизвестно, какие именно задачи будет выполнять структура данных, то применение условного индекса может быть не оправдано из-за возрастания сложности разработки (увеличивается количество используемых сущностей). При тонкой настройке и оптимизации системы наоборот - один универсальный полный индекс может быть заменен на несколько условных. При такой замене можно прогнозировать небольшое замедление при обновлении записей данных и существенное ускорение при поиске по ним с использованием «специализированных» индексов.

3.17 Индекс на вычисляемый атрибут

Индекс на вычисляемый атрибут - это индексная поисковая структура, использующая не хранимые значения атрибутов, а вычисляемые на основе значения одного или совокупности значений нескольких атрибутов. Индексная структура соответствует обычному индексу, но отображает на набор идентификаторов условный виртуальный атрибут, значение которого вычисляется.

При индексировании вычисляемых атрибутов используется соглашение, что одним и тем же правилом вычисления должны пользоваться функции вставки и удаления индексной записи. Это необходимо для того, чтобы поддержка поисковых структур была корректной. При этом функция вычисления значения такого вычисляемого атрибута должна зависеть лишь от значений другого или совокупности других атрибутов. Это требуется для корректного удаления индексной записи и при обновлении строки данных.

Индексация вычисляемых атрибутов имеет характеристики:

1. Значение индексируемого атрибута не хранится.

2. Функция вычисления может быть довольно сложной.

3. Функция вычисления может приводить вычисляемое значение к индексной сортировке.

4. Функция вычисления может использовать не только логические атрибуты самой строки данных, но и атрибуты других объектов, жестко с ней связанных.

Исходя из характеристик индексов на вычислямые атрибуты, они применяются в соответствующих специфических задачах:

Поиск по специфически заданному условию на хранимый атрибут

Например если у объекта есть атрибут дата, но поиск накладывает ограничения на день недели. В этом случае мы можем поддерживать индекс по вычисляемому атрибуту "день недели", значение которого вычисляется на основе поля даты.

Сортировка объектов в сложном порядке

Например, вывод объектов имеющих иерархическую организацию взаимного отношения. При поддержке индекса, ориентированного на специальную сортировку, мы можем многократно использовать хранимый индекс вместо того, чтобы строить его каждый раз.

Поиск по значениям атрибутов подчиненных объектов

В этом случае значения подчиненных объектов рассматриваются как основа для вычисления вычисляемого атрибута. Поиск может выполняться наложением условия как на значение атрибута индексируемого объекта, так и на значения подчиненных объектов одновременно, что позволяет путем поддержки пусть и сложной функции вычисления вычисляемого атрибута добиться очень простого и быстрого поиска.

Реализация быстрого сложного поиска

Возможность вычислять значение вычисляемого атрибута позволяет объединить в функции вычисления довольно сложное условие, в том числе «тяжелые» для поиска операции, например логическое ИЛИ и использовать небольшое утяжеление действий по поддержке такого индекса, но существенно выиграть при поиске данных.

Индексация вычисляемого атрибута может быть выполнена также во многих не-MUMPS системах баз данных путем добавления еще одного хранимого атрибута и поддержкой индекса для него. Этот добавляемый атрибут просто должен автоматически поддерживаться в соответствующем состоянии триггерами. При изменении значений атрибутов, на основе которых вычисляется этот дополнительный атрибут, триггеры просто должны обновить его значение. В этой ситуации хранение дополнительного атрибута просто технически избыточно и логически необязательно,

хотя во многих системах баз данных оно и не может быть полностью устранено.

Один из примеров применения вычисляемых атрибутов - вхождение объекта в иерархическое отношение с другими объектами. Например, если в системе вводится условное деление значений атрибутов на группы. В обычном варианте применяется введение дополнительной связывающей таблицы, отображающей номер группы на допустимые значения, входящие в нее. При этом возникает проблема поддержки полного перечня такого отображения, и, кроме того, при выборке возникает относительно тяжелая операция соединения двух таблиц. В то время как введение вычисляемого атрибута и индексация по нему решает вопрос поиска наиболее элегантным образом.

Кроме технических проблем в данном случае упрощается сопровождение и развитие системы - со временем сложность классификации может быть существенно изменена, и в классификацию может попасть очень сложное условие, поддерживать же в этом случае структуру данных для отображения будет еще сложнее, если вообще это будет целесообразно.

3.18 Индекс поиска по фрагменту

Для поиска по фрагменту значения атрибута такой индекс запоминает набор фрагментов и по каждому фрагменту поддерживается отображение на набор идентификаторов записей, в которых он встретился. Структурно такой индекс может быть как инвертированным списком, так и битовым.

При построении индекса должна быть задана схема выделения фрагментов из значения атрибута. Простой индекс при таком подходе тоже может быть назван индексом поиска по фрагменту, просто на одно значение атрибута в данном случае приходится один фрагмент. Сложность выполняемого поиска по такому индексу определяется сложностью алгоритма, задающего разбиение значения на фрагменты.

Одной из самых тяжелых операций поиска в базах данных является поиск по фрагменту значения. При использовании специального индекса для такого поиска, разумеется, поиск можно существенно облегчить. Но также внимательно следует отнестись к составлению алгоритма выделения фрагментов, поскольку именно совпадение этого алгоритма и поискового шаблона определяет применимость индекса.

Приведем простой пример построения индекса по фрагменту, с поиском по любой заданной части значения атрибута. Алгоритм выделения

фрагментов просто берет подстроки значения атрибута по схеме:

```
USER>s Value="green blue"

USER>f i=1:1:len w $e(Value,i,len),!
green blue
reen blue
een blue
en blue
n blue
 blue
blue
lue
ue
e
```

Задание фрагментов в таком виде, используя соглашения индексного упорядочения, позволяет быстро найти любой объект, содержащий искомый фрагмент. Приведем небольшую учебную реализацию с такой индексацией:

```
CreateRecords()
 k ^Index
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
 s Counts="2~5~12~8"
 f i=1:1:12 d
 . s Figure=$p(Figures,"~",$r(4)+1)
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
DeleteRecord(id)
 q:'$d(^Data(id))
 l +^Data(id)
 n RecordValues s RecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(id)
 l -^Data(id)
```

```
 q
UpdateRecord(id,RecordValues)
 q:'$d(^Data(id))
 l +^Data(id)
 n OldRecordValues s OldRecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,OldRecordValues)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q
InsertIndexRecords(id,RecordValues)
 d InsertIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d InsertIndexRecord("Color",id,$p((RecordValues),"~",2))
 q
DeleteIndexRecords(id,RecordValues)
 d DeleteIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d DeleteIndexRecord("Color",id,$p((RecordValues),"~",2))
 q
InsertIndexRecord(IndexName,id,Value)
 n i,len s len=$l(Value)
 f i=1:1:len s ^Index(IndexName,$e(Value,i,len),id)=""
 q
DeleteIndexRecord(IndexName,id,Value)
 n i,len s len=$l(Value)
 f i=1:1:len k ^Index(IndexName,$e(Value,i,len),id)
 q
FindFigure(part,ids)
 n ref,id
 s ref=$na(^Index("Figure",part))
 f  s ref=$q(@ref) q:(ref="")!($qs(ref,2)'[part)  d
 . s id=$qs(ref,3)
 . s ids(id)=""
 q
```

Здесь функция FindFigure выдает в подиндексы переменной ids найденные идентификаторы, например

```
  d FindFigure^indpart("тре",.ids)
```

находит объекты, у которых атрибут Figure имеет значение "отрезок" и "треугольник".

В действительности иногда приходится искать по гораздо более сложному условию, в котором используется не один фрагмент, а несколько, и накладывается условие на взаимное расположение найденных фрагментов, на расстояние между фрагментами в значении. В этом случае требуется внимательное составление правила получения фрагментов по значению атрибутов, чтобы учесть все предъявляемые требования. В

SQL - ориентированных базах данных есть выражение для условия поиска LIKE. Приведенная техника индекса поиска по фрагменту примерно соотносится именно с таким условием поиска. Но условие LIKE, конечно, имеет гораздо большую сложность.

3.19 Индексация для шаблона (like)

Индексация для поиска по шаблону представляет собой одну из наиболее интересных и занятных задач-головоломок. Общая формулировка задачи такова: построить индекс со структурой и алгоритм поиска так, чтобы поиск по шаблону значения атрибута выполнялся наиболее быстро.

Прямым применением традиционных индексных структур такая задча в общем случае не решается. Обычные индексы могут быть использованы только если шаблон значения атрибута начинается на какие-то символы. Если начало шаблона определено, то мы можем спозиционироваться в индексе на это начало и дальше пройти перебором значений до тех пор пока начальная часть значений в индексе не перестанет соответствовать шаблону.

Конечно, во-первых, в общем случае это условие не выполняется и, во-вторых, перебор индексированных значений по начальной части поиска может пройтись по десяткам мегабайт диска. Для малых баз данных перебор обычно не критичен, но для больших реальных инсталляций такой подход существенно увеличивает число операций с глобалами и может прокачать через кеш множество бесполезных блоков базы данных.

Более того, можно сказать, что для малых баз данных и сама индексация скорее может быть вредной. Если данных в базе данных мало, то намного эффективнее выполнять просто переборы по кешу, если сами данные помещаются в памяти, чем регулярно обращаться к индексным структурам, которые вымывают кеш и усложняют логику работы. Индексация для поиска по шаблону по самой постановке вопроса и по степени его привередливости традиционно относится к реально большим базам данных, где построение индексов соответствующих выполняемой задаче действительно дает преимущество.

В качестве одного из вариантов индексации для поиска по шаблону может быть использован принцип максимального отсечения неподходящих вариантов. Для этого индексируемое значение разбивается на последовательности в 1, 2, 3 и т.д. символов. Например, строка

```
abcd
```

разбивается на подстроки

```
a
b
c
d
ab
bc
cd
abc
bcd
```

При задании шаблона поиска шаблон трансформируется также в последовательность подстрок, которые он содержит, например шаблон

```
ab*d
```

разбивается на подстроки

```
a
b
d
ab
```

После этого производится поиск аналогичный многоиндексной выборке, как если бы у искомой записи было несколько атрибутов, с применением алгоритма zig-zag. Основной задачей такого подхода является пропуск заведомо неиспользуемых подстрок и отсечение как можно большего числа неподходящих записей.

Для построения индекса по шаблону используются служебные функции разбиения слова и шаблона на части:

```
DEPTHS() q 3  ; use 1,...,3 symbols
MAKEPARTS(str,parts) ; use as parts all symbols
 n depth=$$DEPTHS(),d,pos,i,substr
 f i=1:1:depth d
 . f pos=1:1:$l(str)-i+1 d
 . . s substr=$e(str,pos,pos+i-1) s:substr'="" parts(substr)=""
 q
MAKEPATPARTS(str,parts)
 ; use as parts only symbols between * and ?
 q:str=""
 n i f i=1:1:$l(str,"*") d MAKEPATPARTS2($p(str,"*",i),.parts)
 q
MAKEPATPARTS2(str,parts)
 q:str=""
 n i f i=1:1:$l(str,"?") d MAKEPATPARTS3($p(str,"?",i),.parts)
 q
MAKEPATPARTS3(str,parts)
```

```
 q:str=""
 n depth=$$DEPTHS(),d,pos,i,substr,start=$l(str)
 i $l(str)>=depth s start=depth
 f i=1:1:depth d
 . f pos=1:1:$l(str)-i+1 d
 . . s substr=$e(str,pos,pos+i-1) s:substr'="" parts(substr)=""
 q
PAT(mask)   ; make MUMPS pattern based on the LIKE's template
 n quote,i,ret,char
 s quote=0,ret=""
 f i=1:1:$l(mask) d
 . s char=$e(mask,i) s:char="""" char=char_char
 . i "*?"'[char s ret=ret_$s(quote:char,1:"1""_char),quote=1 q
 . i quote s ret=ret_""""
 . s ret=ret_$s(char="*":".E",1:"1E")
 . s quote=0
 i quote s ret=ret_""""
 q:$q ret  q
```

Структурно пример использует набор слов и индексов в виде:

```
^LIKEDATA(id)=word
^LIKEIND(part,id)=""
```

Здесь id - идентификатор слова, word - значение слова, part - часть слова.

Для построения индекса используется набор функций добавления и удаления индексных записей:

```
ADDWORD(word)
 n id=$i(^LIKEDATA)
 s ^LIKEDATA(id)=word
 n parts
 d MAKEPARTS(word,.parts)
 n part="" f  s part=$o(parts(part)) q:part=""  d
 . s ^LIKEIND(part,id)=""
 q
DELWORD(id)
 n word=$g(^LIKEDATA(id)),parts
 d MAKEPARTS(word,.parts)
 n part="" f  s part=$o(parts(part)) q:part=""  d
 . k ^LIKEIND(part,id)
 k ^LIKEDATA(id)
 q
REINDEX
 k ^LIKEIND
 n i="" f  s i=$o(^LIKEDATA(i)) q:i=""  d
 . n parts d MAKEPARTS(^LIKEDATA(i),.parts)
```

```
 . n part="" f  s part=$o(parts(part)) q:part=""  d
 . . s ^LIKEIND(part,i)=""
 q
```

Пример использует контрольный набор данных:

```
INIT ; clear and recreate all data
 k ^LIKEDATA,^LIKEIND
 n word
 f word="abc def","def ghj","rty iop","789 hjk","abdefghj" d
 . d ADDWORD(word)
 q
```

Для выборки по шаблону используется алгоритм многоиндексной выборки адаптированный с использованием нескольких индексов по разным атрибутам к использованию одного индекса по нескольким фрагментам:

```
LIKESELECT(mask,ids)
 n pat=$$PAT(mask),parts,inames,part,id
 ; make template parts
 d MAKEPATPARTS(mask,.parts)
 ; make index names
 s part="" f  s part=$o(parts(part)) q:part=""  d
 . s inames($i(inames))=$na(^LIKEIND(part))
 ; call index search
 k ids
 d AND($na(ids),.inames)
 ; use filter to remove unneed
 s id="" f  s id=$o(ids(id)) q:id=""  d
 . i ^LIKEDATA(id)'?@pat k ids(id)
 q
AND(ret,names) ; make zig-zag ordered selection
 n id,i,j,place
 s id="" f  s i=1,id=$$ANDnext() q:id=""  s @ret@(id)=""
 q
ANDnext()
ANDrep
 s id=$o(@names(i)@(id))
 q:id="" ""
 f j=i+1:1:i+names-1 s place=((j-1)#names)+1 >>
   i '$d(@names(place)@(id)) s i=place g ANDrep
 q id
```

Поскольку при поиске по индексу находятся записи, подходящие по индексу, но не подходящие по шаблону, применяется дополнительная фильтрация по шаблону

```
 s id="" f  s id=$o(ids(id)) q:id=""  d
 . i ^LIKEDATA(id)'?@pat k ids(id)
```

И, наконец, контрольный тест для запуска:

```
test ;
 n ids,like
 f like="* *","*ef","*hj?" d
 . k ids
 . d LIKESELECT(like,.ids)
 . d dump
 q
dump
 w "like template=""",like,""""",!
 n id="" f  s id=$o(ids(id)) q:id=""  d
 . w "found id=",id,", string=""",^LIKEDATA(id),""""",!
 q
```

При необходимости разработчик может вставить трассировку выполнения или счетчики операций и, последовательно совершенствуя алгоритм, добиться большей оптимизации, например выполнять поиск по индексу не в индексной сортировке, а использовать сначала отсечение по фрагментам максимальной длины, например построить структуру

```
^LIKEDATA(id)=word
^LIKEIND(len,part,id)=""
```

Здесь id - идентификатор слова, word - значение слова, part - фрагмент слова, len - длина фрагмента.

Соответственно может быть модифицировано и применение алгоритма поиска $$ANDnext() с тем, чтобы он выполнял проход сначала по самым длинным фрагментам.

Кроме того, каждый разработчик видит проблему по-своему, и у каждого могут быть свои варианты, как построить индексацию для поиска по шаблону, чтобы она выполнялась максимально быстро. Вполне возможно, что разработчик может использовать специфику разрабатываемой системы и / или особенности индексируемых данных.

Интересно отметить, что у поиска по шаблону может быть найдена масса вариаций. В одном из проектов, разработанных на MUMPS с прямым программированием индексных структур и нестандартными выборками из индексов было применено решение, содержащее разбиение значения атрибута на фрагменты и построение индексов по фрагментам. Далее, при вводе искомой строки она также разбивалась на фрагменты. Поиск выполнялся по начальным частям полученных фрагментов и

нечувствительно к регистру. Можно было ввести одну - две буквы из начала каждого из необходимых фрагментов и система автоматически и весьма быстро находила в огромном массиве те записи, в которых присутствовали фрагменты начинающиеся на те же символы.

Такой поиск решал проблему поиска по названиям или именам, когда одно название или полное имя может включать много слов, но затруднительно набрать в тоности то же самое, что было введено при записи данных в базу. В частности, название

```
ООО "Белый Медведь"
```

может быть введено в базу данных в разном виде, но логически эти названия эквивалентны для пользователя:

```
ООО "Белый Медведь"
ООО Фирма "Белый Медведь"
Белый Медведь, ООО
ООО БЕЛЫЙ МЕДВЕДЬ
```

После разбиения на отдельные слова и приведения к общему регистру в индексах системы использовались значения

```
ООО
БЕЛЫЙ
МЕДВЕДЬ
```

Разумеется, что при применении такого механизма поиска, когда пользователь мог просто ввести в строке поиска

```
бе ме
```

система оперативно отыскивала также и необходимое ему название ООО "Белый Медведь", и, возможно, еще пару подходящих под такой же шаблон.

Скорость поиска по описанным алгоритмам такова, что поиск первых 10 - 20 строк может выполняться интерактивно, при очередном вводе символа в строке поиска.

3.20 Индексация уникального атрибута

Одной из интереснейших и спорных тем в индексации является тема индексации уникального атрибута. Индексы в данном случае используются по меньшей мере в двух различных задачах:

1. Для поиска нужной записи по заданному значению атрибута.

2. Для поддержания условия уникальности значения атрибута среди всех имеющихся записей.

Сам факт использования уникальных значений атрибутов вызывает споры, подобные религиозным - использовать ли их в качестве естественных ключей или нет, и на каком уровне реализации системы должна происходить поддержка уникальности - на уровне сервера приложения (прикладная логика) или на уровне сервера базы данных (нижний уровень хранения). Чтобы не вызывать не относящихся к теме вопроса противоречий, выберем условный пример, с которым могут согласиться сторонники различных подходов. Положим, что у нас есть учетная система, в которой сохраняются сведения о пользователях и нам предстоит сохранить в ней атрибуты "номер паспорта" и "логин".

Оба атрибута всегда должны быть уникальны среди всех записей, никак не связаны друг с другом, и номер паспорта может со временем меняться. Кажется очевидным, что при удалении записи пользователя из системы его логин впоследствии также не может быть использован повторно, и что у одного пользователя могут быть два или более паспортов, с единственным ограничением, что только один из них является применимым для совершения новой сделки. Назовем его текущим активным. Кроме того, пользователь системы может перестать в ней существовать в том смысле, что с одной стороны он может быть как-бы удален, с другой стороны имеющиеся на него ссылки должны остаться действующими для расшифровки имевшихся на какой-то предыдущий момент времени в системе взаимосвязей.

Чтобы не возникло желания использовать логин в качестве естественного ключа (догматический подход на практике обычно не работает). Мы наложим еще одно условие - логин также может со временем меняться, также как номер паспорта, логин не может быть впоследствии повторен и всегда должен быть уникален среди всех логинов, известных системе. Приведенный пример выглядит вполне разумным для того, чтобы согласиться с мыслью о правомерности существования уникальных атрибутов.

Чтобы жизнь медом не казалась, придумаем еще один уникальный атрибут - номер служебного мобильного телефона, выдаваемого фирмой своему сотруднику. Номер может со временем быть передан другому сотруднику, он может отсутствовать, и одновременно не может принадлежать двум сотрудникам. В качестве значения такого атрибута нам без разницы что использовать - сам номер или идентификатор записи об

отдельном объекте учета. В любом случае среди всех записей о пользователях заданный номер телефона может или не быть ни у кого, или быть только у одного сотрудника.

В примере номер телефона, наверное, наиболее типичный пример уникального атрибута. Часть схемы данных о пользователе будет такой:

```
Запись о сотруднике:   ^D(iduser)=$lb(phone)
Индексная запись:      ^I(phone,iduser)=""
```

На каком уровне будет проводиться проверка уникальности атрибута, на уровне логики или на уровне хранения, видимо, не принципиально, поскольку действия в принципе должны быть выполнены одни и те же.

Рассмотрим, какие операции могут привести к модификации записи и какие из них к нарушению уникальности атрибута:

Вставка новой записи

При вставке может возникнуть конфликт уникальности, если такое значение уже было. Если не было, то вставка безопасна.

Удаление имеющейся записи

Если запись была, то ее удаление гарантированно оставляет условие "либо один либо ноль" истинным, поэтому удаление записи всегда безопасно с точки зрения уникальности атрибута.

Изменение атрибута

При изменении атрибута может произойти конфликт, если новое значение уже существует. Если такого не было, то изменение безопасно.

Откат транзакции

При откате транзакции сервер базы данных восстанавливает предыдущее значение. В случае, если оно было (операции удаления и изменения), то сервер базы данных самостоятельно выставит значения данным, не согласуясь с поддерживаемым нами ограничением уникальности. Для того, чтобы обезопасить систему от нарушения уникальности при откате транзакции с изменением значения, необходимо применять блокировки.

В целом, уникальность значения атрибута есть лишь один из очень простых видов ограничений целостности базы данных - они могут быть объявлены гораздо более комплексно и затрагивать большое количество сущностей в весьма сложной взаимосвязи.

Для того, чтобы гарантировать, что значение всегда одно, может быть использован индекс вида

```
^Index(IndexName,Value)=id
```

В этом случае в индексе физически не может одновременно существовать две записи с разным id.

Вне зависимости от использования механизмов транзакций, разработчик должен применять блокировки, если налагаются условия не на отдельное значение атрибута, а на его зависимость от значений атрибутов других записей. Проверять и модифицировать записи, если значения атрибутов взаимозависимы с другими записями можно, только если наложена блокировка, означающая, что это условие проверяется текущим процессом. Блокировки могут быть выполнены как монопольно, так и с различением блокирования на операции чтения и записи в зависимости от выполняемой операции.

Имя блокировки не обязано соответствовать имени глобала индекса, это может быть в целом любое имя, главное чтобы система именования отображала блокируемое условие. Зачастую строение индексного глобала достаточно близко к имени блокировки, и именно его разработчики и используют в типовых задачах.

Поддержание уникальности значения атрибута есть частный случай наложения условий на взаимные значения различных записей. В частности, при традиционной трактовке уникальность означает существование записи с данным значением атрибута в количестве от 0 до 1. Значение 1 может быть лишь частным случаем, одним из значений для более общего параметра уникальности N. При замене N на, например, 7, получаем условие чтобы указанное значение присутствовало в записях числом от 0 до 7.

При выполнении таких более общих условий уникальности в других, не MUMPS системах, необходимо приводить условие к условию простой уникальности. Например, завести отдельный справочник допустимых значений, и ссылаться на записи из этого справочника, но с ограничением что на каждую запись справочника может сослаться лишь одна запись. Например, если необходимо выполнить условие "не более 7 синих одновременно", то придется завести 7 записей в справочнике с значением "синий" и ссылаться на такие записи, поддерживая особую дисциплину добавления и удаления записей в справочнике административно. Кроме того, понадобится реализовать стратегию монопольного захвата одной из свободных записей справочника, например с применением опций

```
SELECT ... FOR UPDATE
```

3.21 Массовое перестроение индексов

Операция поддержания индексов в актуальном состоянии обычно состоит в обновлении индексных структур при изменении основных, индекси-

руемых. Некоторым особняком стоит операция массового перестроения индекса по каким-либо причинам. При ее исполнении перестраиваются индексные записи для одновременно большого числа записей. В эксплуатационном отношении это массовое перестроение может быть оптимизировано как программно, так и административно (если такая возможность предусмотрена).

К массовому перестроению индексов / одного индекса приводят задачи:

Создание индекса

При создании индекса СУБД должна для каждой имеющейся индексируемой записи выполнить вставку индексных записей. При этом, пока идет их вставка, сами данные не изменяются.

Изменение кластерного индекса

При изменении кластерного индекса все записи меняют идентификаторы, поэтому все другие индексные записи должны быть перестроены. При перестроении записей сами данные не изменяются.

Массовый импорт данных

Обычно перед импортом данных индексы отключают, если есть такая возможность, проводят импорт, потом включают индекс. Пока индекс выключен, он просто накапливает информацию о необходимости переиндексации определенных записей - вместо внесения индексных структур делается отметка о том, что после включения индекса может быть не полное перестроение индексных записей, а только для измененных / добавленных / удаленных записей. После включения (или активации) индекса перестроение идет не по всем записям, а только по тем, о которых есть отметка.

Удаление данных таблицы

При удалении индексируемых записей СУБД может выполнить как позаписное удаление (delete * from table), так и массовое освобождение занимаемого пространства (truncate table). В случае M-систем полного аналога truncate не существует, поэтому оптимизировать можно лишь удаление индексных структур - при полном удалении данных таблиц полностью удалять все индексные деревья.

Возможное изменение структуры индексируемых данных

Если в структуре индексных записей как-то использовалась информация о структуре индексируемой записи, то в случае ее изменения такие индексы также должны быть перестроены - имеющиеся записи удалены и построены новые.

Изменение определения индекса

При поддержании индекса с условием на вставку или индекса с вычисляемым значением это условие и выражение вычисления могут изме-

ниться. В этом случае все индексные записи по такому индексу должны быть удалены и построены заново. В целях оптимизации по времени выполнения операции массового перестроения индексов мы можем использовать специфические для этих операций признаки:

1. Индексируемые данные не меняются. При этом мы выполняем цикл. Следовательно, у нас могут оказаться инварианты цикла, которые мы можем вынести за пределы цикла. Например, мы можем эскалировать блокировки до более общих, сэкономить на создании и очистке временных переменных и так далее. Вынос инварианта за пределы цикла - это оптимизационная задача, и она может быть выполнена обычно только после составления самого кода, который нужно оптимизировать.

2. При массовой вставке индексных записей мы предполагаем, что эта операция может быть длительной и затронуть значительное количество узлов. Поэтому мы может использовать наиболее общую блокировку индексных и индексируемых записей. Более того, такая эскалация может сделать бессмысленной работу пользователей, и могут быть предприняты специальные административные меры по отключению пользователей от системы на время таких длительных операций.

3. В случае если идет массовое удаление индексных записей мы в итоге должны получить состояние их полного отсутствия. Поэтому нет нужды удалять каждый узел - можно удалить все дерево целиком.

4. В случае если идет массовая вставка индексных записей то в большинстве случаев мы можем использовать специфические для СУБД средства, например $SortBegin / $SortEnd в СУБД Caché.

5. Мы можем программно включить или отключить опцию журналирования / не журналирования. Чтобы при выполнении операции администратор мог выбрать - использовать журналирование каждой операции или выключить журналирование, провести перестроение записей и снова включить журналирование, с тем чтобы после этого выполнить бекап базы - полный или инкрементный. В первом случае для восстановления базы используется предыдущий бекап плюс накат журнала, во втором - предыдущий бекап плюс повтор перестроения индексов либо просто бекап после перестроения индексов. Второй вариант многие администраторы выбирают

по причине его более быстрой работы - бекап выполняется оптимизированно, поблочно, а перестроение индексных записей - позаписно, поэтому операции с бекапом могут быть более эффективны чем операции с журналом.

Программно для массового перестроения индексов предпочтительно иметь специальную утилиту, которую можно вызвать из других утилит для составления простого средства управления сложной системой.

Поскольку при массовом перестроении индексов индексные структуры не соответствуют данным, выполнение операций перестроения для проектируемой системы может быть специальной операцией, выполняемой регламентно, в период неактивности пользователей. Иначе для работы системы во время перестроения индексов разработчикам придется реализовать временные алгоритмы поиска не по индексам, а по самим данным.

3.22 Операции с древовидными индексами

К операциям с древовидными индексами относятся операции, использующие деревья с индексами и реализующие над ними теоретико - множественные операции. К таким операциям относят логические операции над множествами: OR (ИЛИ), AND (И), и другие. При этом деревья используются для хранения множеств - операндов и результата операции.

Структурно древовидный индекс удерживает как значения атрибутов, так и идентификаторы записей. Поэтому в операциях над индексными деревьями есть две группы операций - как над деревьями значений атрибутов, так и над деревьями, включающими идентификаторы записей.

Для простоты будем оперировать разработанным ранее примером поддержки простых индексов:

```
CreateRecords()
 k ^Index
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
 s Counts="2~5~12~8"
 f i=1:1:12 d
 . s Figure=$p(Figures,"~",$r(4)+1)
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
```

```
InsertRecord(RecordValues)
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
DeleteRecord(id)
 q:'$d(^Data(id))
 l +^Data(id)
 n RecordValues s RecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(id)
 l -^Data(id)
 q
UpdateRecord(id,RecordValues)
 q:'$d(^Data(id))
 l +^Data(id)
 n OldRecordValues s OldRecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,OldRecordValues)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q
InsertIndexRecords(id,RecordValues)
 d InsertIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d InsertIndexRecord("Color",id,$p((RecordValues),"~",2))
 q
DeleteIndexRecords(id,RecordValues)
 d DeleteIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d DeleteIndexRecord("Color",id,$p((RecordValues),"~",2))
 q
InsertIndexRecord(IndexName,id,Value)
 s ^Index(IndexName,Value,id)=""
 q
DeleteIndexRecord(IndexName,id,Value)
 k ^Index(IndexName,Value,id)
 q
```

Рассмотрим операции над деревьями значений атрибутов. К ним можно отнести операции выборки поддерева с заданным значением атрибута, выбрать поддерево со значениями атрибута меньшими чем заданное, со значениями большими чем заданное, со значениями между двумя заданными, вычесть поддерево из другого поддерева, и получить число различных значений атрибутов. Приведем примерные реализации этих операций и как их использовать в контрольном примере.

Выборка поддерева с заданным значением атрибута.

```
EQ(ret,name)
 m @ret=@name
 q ret

USER>d EQ^TREEOP($NA(a),$na(^Index("Figure","круг")))
```

Получить поддерево из заданного поддерева кроме указанного значения.

```
NE(ret,name,value)
 m @ret=@name
 k @root@(value)
 q
NE2(ret,name,value...)
 m @ret=@name
 n i f i=1:1:value k:$d(value(i)) @ret@(value(i))
 q

USER>d NE^TREEOP($na(res), »
   $na(^Index("Figure")),"круг")
USER>d NE2^TREEOP($na(res), »
   $na(^Index("Figure")),"круг","квадрат")
```

Получили всё поддерево с индексом кроме поддерева с указанным значением. В первом случае стандартный M код, с вычитанием одного значения, во втором - с расширением COS, с вычитанием значений, заданных списком.

Выборка поддерева со значениями атрибута, меньшими заданного.

```
LT(ret,name,value)
 n i
 s i=value f  s i=$o(@name@(i),-1) q:i=""  m @ret@(i)=@name@(i)
 q

USER>d LT^TREEOP($na(r),$na(^Index("Figure")),"отрезок")
```

Выборка поддерева со значениями атрибутов, большими чем заданное.

```
GT(ret,name,value)
 n i
 s i=value f  s i=$o(@name@(i)) q:i=""  m @ret@(i)=@name@(i)
 q

USER>k  d GT^TREEOP($na(r),$na(^Index("Figure")),"отрезок")
```

Выборка поддерева со значениями между двумя заданными.

```
BT(ret,name,begin,end)
 n i
 s i=begin f  s i=$o(@name@(i)) q:(i="")!(i]]end)!(i=end)  d
 . m @ret@(i)=@name@(i)
 q
```

```
    USER>d BT^TREEOP($na(r),$na(^Index("Figure")), »
    "квадрат","треугольник")
```

Получить число различных значений атрибутов.

```
COUNT(name)
 n ret,id s ret=0
 s id="" f  s id=$o(@name@(id)) q:id=""  s ret=ret+1
 q ret
```

```
USER>w $$COUNT^TREEOP($na(^Index("Figure","треугольник")))
```

Рассмотрим операции с поддеревьями идентификаторов. К ним можно отнести трансформирование дерева - списка значений + списки идентификаторов в дерево списка идентификаторов, операцию объединения множеств (OR), операцию пересечения множеств (AND) и операцию дополнения множеств (SUB).

Сократить дерево на один уровень, или трансформировать дерево - список значений + список идентификаторов в дерево - список идентификаторов:

```
SIMPLE(ret,name)
 n i
 s i="" f  s i=$o(@name@(i)) q:i=""  m @ret=@name@(i)
 q
```

Получить идентификаторы, содержащиеся в индексе по полю Figure:

```
USER>d SIMPLE^TREEOP($na(r),$na(^Index("Figure")))
```

Получить идентификаторы, у которых значение Figure лежит между заданными значениями:

```
USER>d BT^TREEOP($na(r2), »
  $na(^Index("Figure")),"квадрат","треугольник")
USER>d SIMPLE^TREEOP($na(r),$na(r2))
```

Получить объединение множеств:

```
OR(ret,names...)
 n i
 f i=1:1:names m:$d(names(i)) @ret=@(names(i))
 q
```

Используя эту операцию, получить идентификаторы записей, у которых значение Figure равно "круг" или "отрезок":

```
d OR^TREEOP($na(r),$na(^Index("Figure","круг")), »
     $na(^Index("Figure","отрезок")))
```

Найти пересечение множеств:

```
AND(ret,names...)
 n id,i,j,k,place
 s id="" f  s i=1 s id=$$ANDnext() q:id=""  s @ret@(id)=""
 q
ANDnext()
ANDrep
 s id=$o(@names(i)@(id))
 q:id="" ""
 f j=i+1:1:i+names-1 s place=((j-1)#names)+1 »
    i '$d(@names(place)@(id)) s i=place g ANDrep
 q id
```

Используя операцию пересечения найти красные круги:

```
 d AND^TREEOP($na(r),$na(^Index("Figure","круг")), »
 $na(^Index("Color","красный")))
```

Найти дополнение множеств:

```
SUB(ret,from,what...)
 n i,id
 m @ret=@from
 f i=1:1:what d:$d(what(i))
 . s id="" f  s id=$o(@what(i)@(id)) q:id=""  k @ret@(id)
 q
```

Используя операцию дополнения, найти круги и не красные:

```
d SUB^TREEOP($na(r),$na(^Index("Figure","круг")), »
    $na(^Index("Color","красный")))
```

Кроме приведенных операций в практике также встречаются другие, такие как взять первое или последнее значение, или взять начиная с n-го m значений. Для них также можно составить обобщенные операции, использующие косвенность аргумента.

Несложно видеть, что собственно сами операции на индексных структурах просты, если их описывать обобщенно, в косвенной форме, и сложность использования индексов проявляется тогда, когда их жестко прописывают в коде программ, используя предопределенные магические константы.

3.23 Операции с битовыми индексами

Здесь поведем речь о реализации теоретико-множественных операций над битовыми индексами.

Битовый индекс структурно представляет собой отображение значений атрибутов на набор идентификаторов в виде сопоставления с каждым значением битовой последовательности. При этом в силу того, что технически неограниченная последовательность не может быть реализована, она разбивается на сегменты. Единичному биту в такой последовательности сопоставляется признак, что номер этого бита в общей последовательности (с учетом всех предшествующих сегментов) представляет собой числовой идентификатор индексированной записи.

Для выполнения операций поиска в таких индексах строится дерево логического выражения и каждой операции сопоставляется логическая операция над битовыми последовательностями. Каждая из операций имеет результатом битовую последовательность такой же структуры - совокупность битовых сегментов.

При этом, что важно, семантика битовой последовательности является открытой - если есть единичный бит, то ему соответствует объект. Если его нет - то объекта нет. При этом под состояние "нет единичного бита" попадает как нулевой бит, так и отсутствие сегмента как такового. Над непосредственно битовыми последовательностями, таким образом, отсутствует операция унарной инверсии (отрицания), поскольку нет признака окончания последовательности. Таких же принципов открытости логических множеств придерживаются большинство других программных систем, например, в языке Prolog нет операции получить множество неизвестных высказываний.

При этом, хотя нет операции унарной инверсии, в модели открытого мира присутствует операция вычитания, которая в логике закрытого мира эквивалентна операции AND (И) первого операнда с результатом

инверсии второго операнда. В модели открытого мира такая операция не раскладывается на отдельные и применяется как есть, без промежуточной инверсии вычитаемого множества.

С точки зрения применения в базах данных логическая операция "исключающее или" над индексами весьма спорна, поскольку пока не было разумных примеров ее использования. Видимо, именно поэтому в битовых функциях Caché и MiniM этой операции нет. Впрочем, если она потребуется (поскольку мир разнообразен), ее можно выразить через имеющиеся базовые логические операции.

Рассмотрим остальные операции:

1. Логическое И или пересечение множеств.
2. Логическое ИЛИ или объединение множеств.
3. Логический бинарный НЕТ или вычитание множеств.

Структурно, поскольку битовые последовательности имеют сегментное строение, каждая из этих операций должна принять два или больше аргументов и выполнить соответствующую логическую операцию над каждым из сегментов и вернуть набор сегментов с результатом.

Для испытаний воспользуемся тестовыми данными, сделанными с помощью:

```
#define BITSIZE (260000)

CreateRecords()
 k ^Index
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
 s Counts="2~5~12~8"
 f i=1:1:12 d
 . s Figure=$p(Figures,"~",$r(4)+1)
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
```

```
InsertIndexRecords(id,RecordValues)
 d SET($na(^Index("Figure",$p(RecordValues,"~",1))),id)
 d SET($na(^Index("Color",$p(RecordValues,"~",2))),id)
 d SET($na(^Index("Count",$p(RecordValues,"~",3))),id)
 q

SET(name,id,bit=1)
 n seg,pos
 s seg=id\$$$BITSIZE s pos=id#$$$BITSIZE+1
 s $bit(@name@(seg),pos)=''bit
 q
```

Для реализации операции И или пересечения множеств:

```
AND(out,in1,in2)
 n i k @out m @out=@in1
 s i="" f  s i=$o(@out@(i)) q:i="" d
 . s @out@(i)=$bitlogic(@out@(i)&@in2@(i))
 q
```

Для реализации операции ИЛИ или объединения множеств:

```
OR(out,in1,in2)
 n i k @out m @out=@in1
 s i="" f  s i=$o(@in2@(i)) q:i="" d
 . s @out@(i)=$bitlogic(@out@(i)!@in2@(i))
 q
```

Для реализации операции SUB или дополнения множеств:

```
SUB(out,in1,in2)
 n i k @out m @out=@in1
 s i="" f  s i=$o(@in2@(i)) q:i="" d
 . s @out@(i)=$bitlogic(@out@(i)&~@in2@(i))
 q
```

При этом этим операциям не требуется использовать размер сегмента - они должны провести операции с теми сегментами, которые имеются.

Для операций с сегментами битовых карт есть нюанс, неуловимый на первый взгляд - работа с пропущенными сегментами. При простановке битов может оказаться, что по каким-либо сегментам совсем не было операций, и такой сегмент будет отсутствовать. Поэтому операции OR и SUB опираются на первый операнд, но проход выполняют по второму, а операция AND выполняет проход по первому операнду, игнорируя наличие сегментов второго.

Отличительной особенностью реализации операций на битовыми индексами также является отсутствие проверки существования самих битовых сегментов, поскольку реализация функций $bit отсутствующие сегменты (неопределенные значения) считает как не содержащие единичных битов.

Интересной задачей по оптимизации работы приведенных функций AND, OR, SUB выглядит переход от операции merge к проходам по имеющимся сегментам.

Кроме описанных, для типовых операций с данными часто требуется еще две операции - операция получения количества единичных битов:

```
COUNT(in)
 n i,ret s ret=0
 s i="" f  s i=$o(@in@(i)) q:i=""  d
 . s ret=ret+$bitcount(@in@(i),1)
 q ret
```

и получение следующего единичного бита после заданного:

```
NEXT(name,from)
 n ret,seg,pos s ret=0 s from=+from
 s seg=from\$$$BITSIZE s pos=from#$$$BITSIZE+2
 f  s:(seg'="") ret=$bitfind(@name@(seg),1,pos) »
      q:(ret)!(seg="")  d
 . s seg=$o(@name@(seg)) q:seg=""
 . s pos=0
 q:ret seg*$$$BITSIZE+ret-1
 q ""
```

С использованием приведенных функций уже очень легко составить довольно сложные запросы, например запрос "выдать идентификаторы с заданными Figure и Color":

```
Select(Figure,Color)
 n id,result
 d AND($na(result),$na(^Index("Figure",Figure)), »
    $na(^Index("Color",Color)))
 s id="" f  s id=$$NEXT($na(result),id) q:id=""  d
 . w id,!
 q
```

Если провести сравнение простоты и удобства пользования битовыми и древовидными индексами, то у различных систем баз данных, несмотря на общие принципы, реализация битовых индексов различна. И, вероятно, это также является весомым фактором в сравнении. Например,

Caché и MiniM имеют перед другими СУБД чисто технические преимущества - 1) битовые операции атомарны, несмотря на то что операции различных процессов могут прийтись на один и тот же сегмент, 2) откат транзакций работает вполне корректно, 3) блокирование сегментов не требуется в силу пункта 1, 4) применение встроенной компрессии при хранении сегмента и 5) действительно малый объем журнала по сравнению с другими реализациями.

В целом же, если сравнить битовые индексы с древовидными, то древовидные несомненно богаче по функциональности, но битовые намного проще в реализации и, видимо, несколько быстрее в работе. Особенно если используются встроенные и характерные для битовых сегментов операции, например $bitcount. Видится перспективным, что если для древовидных структур в MUMPS появятся аналогичные специализированные функции, то быстродействие древовидных индексов также можно ускорить.

Как возможные направления в развитии Caché и MiniM с целью ускорения древовидных индексов можно было бы назвать встроенные системные функции или команды

Получить число потомков у узла

Системная функция, похожая на $data, но возвращающая число различных значений непосредственных потомков узла, например если есть

```
s ^d("s",1)=""
s ^d("s",4)=""
s ^d("s",78)=""
s ^d("s",54,789)=""
s ^d("s",100)=""
s ^d("k",3)=""
s ^d("k",8)=""
s ^d("k",12,456)=""
```

то такая новая функция вернет:

```
w $data2(^d("s"))
5
w $data2(^d("k"))
3
```

Это позволит упростить часто используемую операцию оценки количества элементов.

Выполнить логический И между потомками двух или больше заданных узлов

Команда, похожая на merge, но оставляющая в результате пересечения подиндексов, например если есть

```
s ^d("s",1)=""
s ^d("s",4)=""
s ^d("s",12)=""
s ^d("k",1)=""
s ^d("k",12)=""
s ^d("k",34)=""
```

то такая новая команда сделает

```
merge2 ^d("R")=^d("s")&^d("k")
zw ^d("R")
^d("R",1)=""
^d("R",12)=""
```

В остальном же сравнение битовых и древовидных индексов скорее всего в той или иной форме приходит к тому, что битовые индексы имеют намного более бедные покрывающие свойства (меньшее число различных операций, в которых они могут быть применены). Если к используемой системе предъявляются довольно сложные требования по запросам, то применение исключительно битовых индексов вместо древовидных может в некоторых случаях даже ухудшить результаты. С другой стороны, если можно обойтись только структурно простыми запросами, легко реализующимися на битовых операциях, это скорее всего приведет к ускорению работы системы, особенно если операция применяется к очень большому объему данных.

В оценке какие именно индексы лучше использовать наилучшим критерием выглядит практика. В задачах OLTP обычно проще применять древовидные, а в задачах OLAP битовые индексы. Преимущество битовых индексов существенно возрастает при увеличении объемов данных, и некоторые системы OLAP построенные не на MUMPS, имели в качестве ключевого преимущества по скорости именно реализацию битовых индексов.

3.24 Совмещение древовидных и битовых индексов

Рассмотрим технические детали совмещения в одной выборке битовых и древовидных индексов. Для такого совмещения следует построить модель абстрагирования от реализации, то есть рассматривать индексные операции как таковые безотносительно их реальной внутренней реализации. И, используя абстрактные операции, реализовать механизм совмещения двух разнородных структур.

В случае древовидных и битовых индексов такой абстрацией может быть абстрагирование до уровня отображения вообще, следования вообще, проверки существования вообще и другие. Составив алгоритм оперирования индексами вообще, его просто нужно адаптировать до уровня, способного использовать конкретную реализацию, не зная ее деталей.

Основной операцией выборки по индексам является операция И. Рассмотрим на ее примере данную методику. Для этого мы располагаем двумя абстрактными операциями - теоретико-множественной операцией битового И и многоиндексной шаговой выборки. Приведем операции с обоими видами индексов к одинаковым абстракциям.

Многоиндексная операция выборки, как было рассмотрено, использует такие ключевые элементы:

1. Упорядоченный набор имен индексов. Упорядочение требуется чтобы выбирать их из используемого набора.

2. Одинаковое упорядочение (сортировка) выбираемых из индекса искомых значений.

3. Наличие у абстрактного индекса операции "взять следующий идентификатор".

4. Наличие у абстрактного индекса операции "проверить существование идентификатора".

Под все четыре приведенных условия мы уже можем подвести операции и с битовым и с древовидным индексом. Сам алгоритм, выполняя собственно выборку, не должен знать реальную реализацию каждого индекса, а должен оперировать каждым из индексов в обобщенной форме. Для реализации такого полиморфизма используем косвенность - будем передавать в структуру, описывающий индекс, кроме имени самого индекса, также еще два имени - один как имя функции, выполняющей получение следующего идентификатора, и второй как имя функции выполняющей проверку существования заданного идентификатора в индексе.

```
CreateRecords()
 k ^Index
 k ^Data
 n i,Figures,Colors,Counts,Figure,Color,Count,id
 s Figures="квадрат~круг~отрезок~треугольник"
 s Colors="красный~зелёный~синий~белый"
 s Counts="2~5~12~8"
```

```
 f i=1:1:24 d
 . s Figure=$p(Figures,"~",$r(4)+1)
 . s Color=$p(Colors,"~",$r(4)+1)
 . s Count=$p(Counts,"~",$r(4)+1)
 . s id=$$InsertRecord(Figure_"~"_Color_"~"_Count)
 q
InsertRecord(RecordValues)
 n id s id=$i(^Data)
 l +^Data(id)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q id
DeleteRecord(id)
 q:'$d(^Data(id))
 l +^Data(id)
 n RecordValues s RecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,RecordValues)
 k ^Data(id)
 l -^Data(id)
 q
UpdateRecord(id,RecordValues)
 q:'$d(^Data(id))
 l +^Data(id)
 n OldRecordValues s OldRecordValues=$g(^Data(id))
 d DeleteIndexRecords(id,OldRecordValues)
 s ^Data(id)=RecordValues
 d InsertIndexRecords(id,RecordValues)
 l -^Data(id)
 q
InsertIndexRecords(id,RecordValues)
 d InsertIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d SET($na(^Index("Color",$p((RecordValues),"~",2))),id)
 q
DeleteIndexRecords(id,RecordValues)
 d DeleteIndexRecord("Figure",id,$p((RecordValues),"~",1))
 d DEL($na(^Index("Color",$p((RecordValues),"~",2))),id)
 q
InsertIndexRecord(IndexName,id,Value)
 s ^Index(IndexName,Value,id)=""
 q
DeleteIndexRecord(IndexName,id,Value)
 k ^Index(IndexName,Value,id)
 q
#define BITSIZE (260000)
DEL(name,id) s bit=0 g SET+1
SET(name,id,bit=1)
 n seg,pos
 s seg=id\$$$BITSIZE s pos=id#$$$BITSIZE+1
```

```
 s $bit(@name@(seg),pos)=''bit
 q
```

Здесь по атрибуту Color строится обычный битовый индекс, а по атрибуту Figure простой древовидный. Теперь модернизируем алгоритм многоиндексной выборки таким образом, чтобы он использовал не предопределенные операции $O(), $D(), $BIT(), а заданные извне имена функций. И составим сами функции, реализующие соответственно выборку следующего идентификатора и проверку существования идентификатора.

```
Select(Figure,Color)
 n res,names
 ; ставим функцию next и data для простого индекса
 s names(1,"next")="OrderSimple"
 s names(1,"data")="DataSimple"
 s names(1)=$na(^Index("Figure",Figure))
 ; ставим функцию next и data для битового индекса
 s names(2,"next")="OrderBit"
 s names(2,"data")="DataBit"
 s names(2)=$na(^Index("Color",Color))
 s names=2
 d ANDvi($na(res),.names)
 s res="" f  s res=$o(res(res)) q:res=""  d
 . w res,!
 q
 ;
ANDvi(ret,names) g ANDi+1
ANDi(ret,names...)
 n id,i,j,place
 s id="" f  s i=1,id=$$ANDnexti() q:id=""  s @ret@(id)=""
 q
ANDnexti()
ANDrepi
 s id=$$@names(i,"next")(names(i),id)
 q:id="" ""
 f j=i+1:1:i+names-1 s place=((j-1)#names)+1 »
  i '$$@names(place,"data")(names(place),id) s i=place g ANDrepi
 q id
 ;
#define BITSIZE (260000)
OrderSimple(ref,id) q $o(@ref@(id))
DataSimple(ref,id) q $d(@ref@(id))
DataBit(ref,id) q
$bitfind(@ref@(id\$$$BITSIZE),1,id#$$$BITSIZE+1)=(id#$$$BITSIZE+1)
OrderBit(name,from)
 n ret,seg,pos s ret=0 s from=+from
 s seg=from\$$$BITSIZE s pos=from#$$$BITSIZE+2
```

```
f  s:(seg'="") ret=$bitfind(@name@(seg), »
    1,pos) q:(ret)!(seg="")  d
. s seg=$o(@name@(seg)) q:seg=""
. s pos=0
q:ret seg*$$$BITSIZE+ret-1
q ""
```

Здесь вызовы

```
$$@names(i,"next")(names(i),id)
$$@names(place,"data")(names(place),id)
```

выполняют косвенный вызов метки по ее имени для выполнения необходимых операций, абстрагируясь от действительного типа индекса.

Анализ показывает, что мы действительно справились с задачей совместить в одной выборке разноструктурные битовые и древовидные индексы, и при этом количество избыточных операций увеличилось не намного, а именно осталось в линейных пределах.

В действительности, нам здесь очень сильно помог тот факт, что битовые индексы оперируют идентификаторами как натуральными числами (целыми и даже не отрицательными). Для них порядок арифметического следования совпадает с индексным упорядочением. Поэтому для обоих типов индексов удалось составить достаточно небольшое число абстракций и легко и понятно их реализовать.

Для операции ИЛИ выборка из индекса сводится к простой операции выборки идентификаторов по перечню индексов в одну структуру со слиянием. Результат также может быть по своей структуре как древовидной, так и битовой структурой.

Операция вычитания может быть сделана как операция выборки первого операнда в результирующую структуру, с последующей операцией выборки по второму операнду (вычитаемому), но вместо слияния выполняется удаление из результата.

3.25 Сортировка по индексу

В отличие от систем, имеющих встроенные специализированные механизмы сортировки строк, M-системы имеют единственную возможность произвести сортирование - это построить сортирующий индекс. Такие экзотические случаи, как сортировка фрагментов строки с разделителями, видимо, не будем относить к более-менее реальным случаям для мира баз данных с их объемами. Поскольку в M-системах определено единственное упорядочение - это индексная сортировка, то для построения

сортирующего индекса требуется введение промежуточной процедуры приведения сортируемых значений к соглашениям индексной сортировки.

В индексной сортировке используется соглашение следования - если оба значения являются строковыми представлениями числа, то они сравниваются как числа в арифметическом упорядочении. Если не являются представлениями числа, то сортируются как строки побайтно. Если одно значение является представлением числа, а другое нет, то числовое значение следует перед нечисловым. Кроме того, в большинстве реализаций M-систем существует возможность тонкой настройки правила сортировки путем определения сортирующих таблиц. Для Caché это выполняет утилита nls. Для MiniM сортировка символов задается файлом определения символов (nat файл).

Сортирующий индекс может быть постоянным или временным - постоянный обычно используется для упорядоченной выборки постоянно хранимых данных, временный - для упорядоченной выборки временных данных, например, уже выбранных по какому-либо условию.

Временный сортирующий индекс, также как и постоянный, может быть организован в глобали в целях избежания переполнения локальной памяти процесса. В этом случае, естественно, принимаются меры к различению данных между процессами, их использующими. Обычно в практике используется либо номер процесса либо условный очередной номер выборки.

Для сортировки значений атрибутов их следует либо приводить к числовому значению, либо к строковому. Операция приведения к сортирующему значению не обязательно обратима - это следует иметь ввиду при использовании сортирующего индекса для поиска. Также, в отличие от поискового индекса, сортирующий индекс зачастую логически сортирует сами объекты, а не значения атрибутов - используется вычисляемый атрибут, для которого задается функция возвращающая сортирующее значение. При этом сортирующее значение может зависеть от нескольких атрибутов.

Традиционным способом приведения к числовому сортирующему значению является операция плюс - ее результат всегда числовое представление значения или первой его части с отбрасыванием незначащей для интерпретации строки как числа. К числовым сортировкам относят как сами числовые величины, так и другие, которые могут быть трактованы в некотором условном числовом пространстве, например даты.

Более общей является строковая сортировка - путем приведения к строке можно сортировать составное значение. Традиционными средствами приведения к строковой сортировке являются методы добавления

лидирующего пробела, использования непечатного символа в качестве внутреннего разделителя групп, приведение чисел к строковому представлению путем выравнивания разрядов, приведение к одному регистру, введение фиктивного поля со значением, производным от идентификатора строки.

Добавление лидирующего пробела приводит возможное числовое значение атрибута к гарантированному строковому. Вместо пробела можно использовать любой символ, не являющийся цифрой - последующие за ним цифры будут сортироваться как строки.

Выравнивание числовых значений используется в сложной сортировке для того, чтобы значение, будучи числовым, и сортируемым как строка, тем не менее давало именно числовую сортировку - то есть при строковой сортировке порядок следования чисел должен сохраниться.

Для этого применяют обычно классические методы - выравнивание по ширине, дополнение незначащими лидирующими и завершающими нулями, проставление знака числа. Так же нормируются дробные числа, отдельно форматируются мантисса и отдельно порядок.

Приведем простой пример выравнивания чисел:

```
ViewSorted()
 n a
 s a($$ToSortVal(-100))=""
 s a($$ToSortVal(-1.345))=""
 s a($$ToSortVal(-1.0))=""
 s a($$ToSortVal(-5.0))=""
 s a($$ToSortVal(0))=""
 s a($$ToSortVal(1.0))=""
 s a($$ToSortVal(1.345))=""
 s a($$ToSortVal(3.0))=""
 s a($$ToSortVal(100.5))=""
 w "Как значения легли в индексе",!
 zw a
 w "Как значения выбираются из индекса",!
 s a="" f  s a=$o(a(a)) q:a=""  d
 . ; удаляем лидирующие нули
 . n z s z=a f  q:$e(z)'="0"  s $e(z)=""
 . w +z,!
 q
ToSortVal(val)
 n p1,p2
 s p1=$p(val,".",1)
 s p1=$j(p1,10," ") s p1=$tr(p1," ","0")
 s p2=$p(val,".",2)
 s p2=$re($j($re(p2),10," ")) s p2=$tr(p2," ","0")
 q p1_"."_p2
```

здесь сортируемые значения приводятся к строкам, но при этом, сортируясь как строки они сортируются также как числа, сохраняя взаимное арифметическое упорядочение.

Это качество позволяет использовать такой фрагмент для соединения числа, приведенного к строковой сортировке, в качестве части строки для сложной сортировки, скажем одновременно по нескольким атрибутам, которая, безусловно, физически является строковой сортировкой.

Результат работы этого теста такой:

```
USER>d ViewSorted^indsort()
Как значения легли в индексе
a("000000-100.0000000000")=""
a("00000000-1.0000000000")=""
a("00000000-1.3450000000")=""
a("00000000-5.0000000000")=""
a("0000000000.0000000000")=""
a("0000000001.0000000000")=""
a("0000000001.3450000000")=""
a("0000000003.0000000000")=""
a("0000000100.5000000000")=""
Как значения выбираются из индекса
-100
-1
-1.345
-5
0
1
1.345
3
100.5
```

Здесь отрицательные числа сортируются неправильно, и для компенсации такого эффекта необходимо приведение всех чисел к положительным путем прибавления константы к сортируемому значению для получения сортирующего:

```
ViewSorted()
 n a
 s a($$ToSortVal(-100))=""
 s a($$ToSortVal(-1.345))=""
 s a($$ToSortVal(-1.0))=""
 s a($$ToSortVal(-5.0))=""
 s a($$ToSortVal(0))=""
 s a($$ToSortVal(1.0))=""
 s a($$ToSortVal(1.345))=""
 s a($$ToSortVal(3.0))=""
 s a($$ToSortVal(100.5))=""
```

```
 w "Как значения легли в индексе",!
 zw a
 w "Как значения выбираются из индекса",!
 s a="" f  s a=$o(a(a)) q:a=""  d
 . ; удаляем лидирующие нули
 . n z s z=a-1000000000 f  q:$e(z)'="0"  s $e(z)=""
 . w +z,!
 q
ToSortVal(val)
 n p1,p2 s val=val+1000000000
 s p1=$p(val,".",1)
 s p1=$j(p1,10," ") s p1=$tr(p1," ","0")
 s p2=$p(val,".",2)
 s p2=$re($j($re(p2),10," ")) s p2=$tr(p2," ","0")
 q p1_"."_p2
```

Здесь значение 1000000000 было использовано в качестве сортирующего дополнения, и эта величина существенно зависит от характера данных. В реальных случаях выбору величины дополнения необходимо уделять особое внимание.

Теперь, после применения дополнения, результат работы правильный, и числовое значение приведено к строковой сортировке, включая как положительные, так и отрицательные величины.

```
USER>d ViewSorted^indsort()
Как значения легли в индексе
a("0999999900.0000000000")=""
a("0999999995.0000000000")=""
a("0999999998.6550000000")=""
a("0999999999.0000000000")=""
a("1000000000.0000000000")=""
a("1000000001.0000000000")=""
a("1000000001.3450000000")=""
a("1000000003.0000000000")=""
a("1000000100.5000000000")=""
Как значения выбираются из индекса
-100
-5
-1.345
-1
0
1
1.345
3
100.5
```

Отметим, что аналогичным способом отображения чисел из одного множества в другое но уже с требуемыми сортирующими свойствами

используют также для дат. Стандартно в M дату используют в формате $H, но это число, и для приведения к строковой сортировке используют замену, например в виде функции $zdt($h,8) в Caché. Это дает приведение даты и времени к замечательно сортирующемуся виду в фиксированном формате

```
"YYYYMMDD HH:MM:SS"
```

В таком виде использовать дату в качестве части сортирующей строки даже вполне читабельно при отладке.

Введение непечатного символа позволяет ввести в сортировку правило разбиения на группы - до непечатного символа значения сортируются в рамках одной группы, после - в рамках подгруппы группы. Естественными причинами введения групп являются иерархические отношения - для упорядочения, скажем, номеров субсчетов в рамках субсчета, в который они сами входят. В качестве непечатных символов используют символы меньшие пробела. Это гарантирует, что сортирующий механизм MUMPS системы будет считать более короткие группы сортирующимися перед более длинными. При необходимости каждая из групп также может приводиться к фиксированному по длине форматированию.

Важным моментом сортировки является случай совпадения сортирующей последовательности для различных объектов. В этом случае следует либо ввести фиктивную группу со значением, производным от идентификатора объекта, либо строить дополнительный подиндекс.

Например, пусть есть два объекта

```
^Data(1)="Белый~Волк"
^Data(2)="Белый~Гусь"
```

здесь при сортировке следует выбрать либо вариант групп, либо вариант подиндекса.

Первый вариант, с применением групп:

```
структура:
 ^sort(цвет_$C(1)_идентиифкатор)=""
пример данных:
 ^sort("Белый"_$C(1)_"1")=""
 ^sort("Белый"_$C(1)_"2")=""
```

Второй вариант, с применением подиндекса:

```
структура:
 ^sort(Цвет,идентификатор)=""
пример данных:
 ^sort("Белый",1)=""
 ^sort("Белый",2)=""
```

При этом для выборки следует использовать в первом случае проход по $O() и получать значение искомого идентификатора функцией $P(), а во втором - проход по $Q() и получение искомого идентификатора функцией $QS().

Нужно помнить, то способ сортирования объектов полностью определяется функцией получения сортирующего значения. При ее формировании следует быть особенно внимательным и обеспечить правильность ее работы.

3.26 Статистики и кардинальность

Статистики индексов и кардинальность атрибутов - это числовые характеристики индексов и записей данных, помогающие автоматическому оптимизатору запроса выбрать наиболее оптимальный план выполнения этого запроса. В собственно самих MUMPS системах нет механизма оптимизатора, встроенных операций с индексами и выполнение программы полностью определяется программистом. Поэтому далее речь идет об условном оптимизаторе, который может быть выполнен программно, либо разработчик может определить стратегии самостоятельно.

Кардинальностью атрибута называется количество его различных значений. Скажем, у атрибута "пол" значений может быть всего три - мужской, женский и не указан. Поэтому при среднестатистической выборке по этому индексу СУБД фактически вернет треть таблицы. Если требуется сузить результат и есть возможность использовать иной план, то лучше его также оценить и выбрать лучший. Если же значением атрибута является дата, то, скажем, если в таблице за год сделать выборку на указанную дату, то СУБД вернет примерно одну трехсотую часть таблицы. Конечно, во втором случае кардинальность выше и избирательность индекса по атрибуту дата намного лучше, чем по атрибуту пол. Иными словами, при выборке по высококардинальному атрибуту производится большее отсечение данных, чем при выборке по низкокардинальному.

Статистиками индекса называются внутренние дежурные данные, которые ведет (или может вести и поддерживать по различным стратегиям) СУБД, и в которых отражается статистическая информация по индексам. Большинство СУБД поддерживают гистограммные или более сложные статистики. В этих дежурных данных движок СУБД запоминает избирательность индекса вообще, избирательность отдельных значений (несмотря на то, что может быть много различных дат, одна из них может быть у больше чем половины записей), частота использования этого индекса и относительная эффективность его кеширования,

объем загружаемых с диска данных при использовании этого индекса, относительная производительность операций при использовании индекса определенной структуры, и многое другое.

Реальная работа оптимизатора и устройство статистик зачастую полностью производителями СУБД не раскрываются, зачастую составляют предмет гордости производителя и предмет особого внимания, поскольку именно эта часть наиболее уязвима при сравнении систем по скорости выполнения запросов и чувствительна к изменению как характера данных, так и настроек сервера. Но для администраторов предоставляются дополнительные административные средства поддержки не только индексов, но и их статистик. Если наличие статистик по отношению к таблицам является средством оптимизации доступа, то статистики по отношению к индексам являются средством оптимизации доступа к самим индексам.

Зачастую полноценная статистическая информация для своей поддержки может требовать колоссальных расходов - изменение одной лишь записи данных может повлиять на многие числовые характеристики, большинство из которых могут быть составлены как интегральные. При этом для обновления статистик в живую, на работающей базе, может понадобиться гораздо большая вычислительная мощность, чем на поддержание собственно самих обслуживаемых данных.

Для того, чтобы не перегружать сервер избыточными рассчетами статистик, большинство СУБД имеют отдельно запускаемую утилиту обновления статистик. При этом штатным режимом сервера является работа некоторое время в режиме неизменных статистик и периодическое их перестроение. При работе в режиме необновления статистик оптимизатор полагается на частично устаревшие данные и может выбрать неоптимальный план исполнения, хотя в большинстве случаев статистики отражают ситуацию относительно адекватно. Обычно уже небольшая часть записей может приблизительно охарактеризовать и характер распределения последующих данных.

Многим администраторам и программистам известна ситуация, когда сервер с автоматичесим определением плана запроса начинает существенно снижать производительность, хотя на первый взгляд ничего странного с ним не сделали. Во многих случаях это именно проявление неактуальности статистик индексов - планы выполнения запросов выбираются на основании неактуальных данных. Для исправления такой ситуации рекомендуется периодически запускать утилиту обновления статистик - или вручную, или вносить ее исполнение в регламент автоматической работы сервера.

При использовании MUMPS систем программисты, используя пря-

мой доступ, никогда не сталкиваются с проявлением автоматического оптимизатора за его отсутствием. Но для создания общей картины приведем пример, в котором может проявиться работа оптимизатора в других СУБД либо если он выполнен в прикладной программе. Или, по крайней мере, проиллюстрировать его макет.

Пусть есть схема из двух таблиц - документ и строка документа. В документе пусть есть атрибуты дата, номер и название. У строки документа пусть есть порядковый номер, название изделия и количество. В задаче требуется выполнить выборку таких документов, на которые наложены условия:

1. Документы имеют заданную дату.

2. В строках документа используется заданное изделие.

Для реализации мы делаем две таблицы - одну для документа и вторую для строк документа. При этом вторая имеет ссылочный ключ на первую. Кроме того, поскольку в условии выборки фигурирует дата документа, то мы хотим использовать индекс на атрибут дата документа, и создаем его. Кроме того, по тем же соображениям создаем индекс по полю изделие у таблицы строк документов. Для поддержания ссылочной целостности поддерживаем индекс на атрибут - внешний ключ строки документа на документ.

Для реализации запроса у нас есть на выбор по меньшей мере два плана:

1. По индексу на дату документа выбрать подходящие документы и получить набор строк документа. Используя этот набор и индекс на ссылочный ключ строк документа, и второй индекс по атрибуту изделия строк документа, используя операцию пересечения множеств, получить набор ссылочных ключей строк документа. Это будет набор идентификаторов документов.

2. Используя индекс по атрибуту изделия строк документа, получить набор строк документа. Используя набор идентификаторов документов и индекс по атрибуту даты документа и используя операцию пересечения множеств, получить набор идентификаторов документов.

Стоит вопрос - какой из двух этих планов выполнения выбрать. Автоматический оптимизатор может использовать кардинальность индексов

по атрибуту даты и изделия, чтобы выбрать вариант, имеющий наибольшее сужение первичной области поиска.

Пусть в начале жизни системы мы ввели два документа и в каждом по пять строк с различными изделиями. В этом состоянии индекс по дате содержит два различных значения, индекс по изделию содержит десять различных значений. В данной ситуации оптимизатор, скорее всего, выберет второй вариант - использовать сначала индекс по изделию, потом индекс по дате с пересечением.

Со временем в систему вводятся новые документы. Каждый день число различных значений дат документов увеличивается, но число различных изделий в строках документов ограничено спецификой документов и изделий, и со временем растет все меньше, и через некоторое время перестает расти. Через некоторое время наступает момент, когда число различных значений дат документов становится больше, чем различных изделий в строках документов. Но, если статистики не обновлены и отражают начальное состояние, то оптимизатор по инерции может выбирать все тот же план. Если выполнить перестроение статистик, то при следующем таком запросе оптимизатор уже может изменить план выполнения на более быстродействующий.

Это, в целом, упрощенная ситуация с использованием статистик автоматическим оптимизатором. В действительности современные оптимизаторы используют большое количество низкоуровневой информации, как например распределение записей по блокам и эффективности кеширования индексных записей, частота использования индекса и вероятность его кеширования, и другие.

При программировании MUMPS систем, традиционно, программисты не используют автоматических оптимизаторов и пишут код выборки в традиционном навигационном стиле. При этом, в отличие от автоматического режима, есть возможность попасть в противоположную неприятность - не учтя статистик данных, использовать план выборки, который уже не сможет быть изменен без перепрограммирования. Но, одновременно с тем, при правильном выборе плана система будет работать намного более стабильно и предсказуемо, не будет зависеть от таинственных и неуправляемых факторов. Иллюстрируя вышеприведенным примером, разработчики могут изначально ориентироваться на то, чего будет больше в реальной системе - различных дат или различных изделий.

Кроме того, при прямом доступе есть возможность использования сложных индексов, использовать которые автоматический оптимизатор врядли сможет, такие как, скажем, индексы с условием на вставку или индексы для поиска по фрагменту.

Что касается именно приведенного примера, то, в отличие от табличноориентированных систем с автоматическим поддержанием индексов, более эффективным вариантом может оказаться поддержание межтабличного индекса, объединяющего изделие в строке документа и дату документа и возвращающего пару - идентификатор документа и номер строки документа, или внутри табличный индекс, отображающий изделие на внешний ключ - идентификатор документа, или, что проще, составной индекс на атрибуты изделие и внешний ключ. В последнем случае, если автоматический оптимизатор увидит покрывающие свойства такого индекса и сможет использовать многоиндексную выборку (zig-zag ordered scan), то это будет, вероятно, хорошим решением.

При применении MUMPS систем, в отличие от систем с автоматическими оптимизаторами, работающими по декларативно заданным запросам, присутствуют, таким образом, как плюсы, так и минусы. К минусам можно отнести то, что при разработке прикладных систем необходимо уделить определенное внимание и затратить некоторые усилия на введение в систему индексов. К плюсам можно отнести то, что у разработчиков прикладных систем на MUMPS практически отсутствуют ограничения и на типы возможных к применению индексов, и на алгоритмы их поддержки и использования. Хотя базовый набор действий выполняемых MUMPS системой прост, на нем возможно построение от самых простых до самых сложных систем.

Глава 4

Конкурентный доступ

4.1 Параллельность выполнения

MUMPS системы изначально предусматривают выполнение чего-либо в рамках процессов (называемых job, или задание), существующих в контексте M системы. Процесс может быть запущен, выполняться, или завершить работу. Если на компьютере установлено и работает несколько M систем, то в контексте каждой из них набор процессов и их окружение будут собственными.

Физический способ реализации процесса по стандарту MUMPS не лимитирован никаким образом, и, в зависимости от реализации, это могут быть действительно процессы операционной системы, или потоки, или собственный механизм переключения контекстов. Стандарт MUMPS преполагает, что M система выполняет процессы параллельно, хотя в силу особенностей реализации физический способ такой параллельности может отличаться.

Примеры физически различной реализации процессов M системы:

1. В MiniM и Caché M процесс соответствует процессу операционной системы
2. В MiniMono и M3-Lite M процесс соответствует потоку операционной системы
3. В MSM-DOS M процесс соответствует внутреннему виртуальному контексту

Каждый из процессов M системы имеет собственный текущий номер. Этот номер возвращается системной переменной $JOB. При старте процесса M система назначает ему номер не совпадающий ни с одним уже

имеющимся таким образом, что никакие два выполняющиеся на одной M системе (это не относится к процессам нескольких M систем на том же компьютере) процесса не имеют одинакового номера.

Номер процесса, полученный им при старте, не изменяется в течении всего времени его работы.

Собственно само значение переменной $JOB также не определено по формату, это может быть сочетание цифр или букв. Это может быть длинное или короткое число. Но в любом случае это число отличает текущий процесс M системы от любых процессов этой же системы.

Вообще говоря, после завершения M процесса его номер процесса может быть использован повторно при старте другого M процесса, но это необязательное условие. В частности, в системе M3-Lite выделение процессам номеров никак не привязано ни к номерам процессов операционной системы, ни к номерам потоков операционной системы, а выделяется последовательно.

Физическое соответствие значения переменной $JOB объектам операционной системы также зависит от реализации M системы:

1. В MiniM и Caché $JOB соответствует номеру процесса операционной системы
2. В MiniMono $JOB соответствует номеру потока операционной системы
3. В M3-Lite и MSM-DOS $JOB соответствует внутреннему виртуальному номеру

Разработчикам M программ, таким образом, не следует полагать, что номер $JOB связан с каким-либо объектом операционной системы. Более того, значение номеров $JOB и способ их выделения процессу может меняться от версии к версии одного производителя.

Согласно стандарта MUMPS, все имеющиеся в M системе процессы перечисляются в структурной системной переменной ^$JOB в качестве индексов первого уровня и в отношении этой переменной должен поддерживаться минимальный набор операций, такие как $ORDER и $DATA.

Например, перечислить имеющиеся в M системе процессы можно так:

```
s j="" f  s j=$o(^$J(j)) q:j=""  w j,!
```

Кроме операций предусмотренных стандартом, различные реализации M систем могут дополнительно поддерживать расширенный набор операций со структурными системными переменными, например в MiniM поддерживается операция

```
kill ^$JOB(job)
```

принудительно завершающая выполнение указанного процесса.

Для разработчиков на M важным моментом является характер выполнения процессов и переключение их контекстов и характер чтения значений переменных.

Формально, стандарт MUMPS никак не регламентирует характер переключения контекстов M процессов при выполнении программ и отдельных команд или функций. При выполнении кода, состоящего из вызовов функций, чтения и записи переменных и передаче управления от команды к команде, в любой момент времени может произойти переключение контекста процессора на другой процесс. Если есть, например, команда

```
set a=a+1
```

то процесс может прочитать значение переменной a, после чего контекст выполнения процессора переключится на другой M процесс, затем продолжится выполнение с операции сложения, затем снова прервется и затем продолжится операцией записи. Те же самые соглашения о неопределенности прерывания выполнения для переключения контекстов касаются и других многозадачных систем и средств программирования.

Но, одновременно с тем, что M системы выполняют процессы параллельно, для них, как для процессов СУБД, отдельно определено дополнительное соглашение об атомарности обращений к переменным.

Для каждого процесса в M системах определены 3 вида переменых:

1. Локальные переменные, принадлежащие и видимые только текущему процессу.

2. Глобальные переменные, принадлежащие M системе в целом и видимые всем процессам.

3. Системные и структурные системные переменные, существующие у каждого процесса, но значение которых определяется для каждой переменной индивидуально.

Для каждого из этих видов переменных определено, что процесс получает значение переменной целиком. В отличие от других средств разработки, в M системах может быть предусмотрена возможность получить значения локальных переменных одного процесса другим процессом и чтение состояния его системных переменных. Это делается в целях отладки, например, или при выполнении административных задач.

Атомарность доступа к значениям переменных означает, что если переменная имеет какое-то значение, то процесс читает его целиком. Если процесс изменяет значение, то оно заменяется целиком. Например, пусть два процесса выполняют операции

```
первый
s ^A="11111111111111111111"

второй
s ^A="22222222222222222222"
```

При этом соглашение об атомарности доступа приводит к тому, что третий процесс прочитает либо значение из всех единиц, либо из всех двоек, но никогда не прочитает смесь из байт, записанных разными процессами.

Отдельные функции языка выглядят так что они как будто меняют только часть переменной, например

```
set $extract(^A,4,7)=1234
```

В действительности, для таких функций не определено, как именно M система будет выполнять конкурентный доступ. Есть два варианта выполнения, первый - выполняется сначала чтение значения, потом его модификация во временной памяти, и затем запись строки опять же целиком. Но параллельность выполнения процессов приводит к тому, что между операцией чтения и записи может произойти перезапись этой же переменной другим процессом. Второй вариант - система блокирует доступ к этой переменной, не давая другим процессам изменить переменную, выполняет изменение и отпускает блокировку. Большинство M систем реализует первый вариант. Те системы баз данных, которые используют второй вариант, обрекают себя на возможность коллизий взаимоблокировок, если для блокирований используют те же механизмы блокировок, которые используются программистами.

Что интересно в M системах, это то, что паралельность выполнения команд и функций различными M процессами обеспечивается вне зависимости от физической реализации системы и от типа базовой операционной системы. В любом случае разработчикам предоставляется набор соглашений о поведении таких виртуальных процессов и стандартные соглашения о переносимости программ, даже если целевая M система работает в однозадачной операционной среде.

4.2 Блокировки

Для обеспечения корректности одновременного доступа различных M процессов в условиях параллельности их выполнения M системы реализуют различного рода блокировки. Часть этих блокировок выполняется автоматически внутренними механизмами системы, часть предусмотрены стандартом языка и предоставлены разработчикам.

Блокировка - это независимо от других объектов существующая запись в контексте M системы. Блокировки не хранимы на дисках, а существуют в специально отведенной области памяти M системы. Формально говоря, запрета на хранение блокировок в файле на диске нет, но все имеющиеся M системы используют только блокировки, хранящиеся в оперативной памяти.

При блокировании указывается имя, структурно соответствующее имени локальной или глобальной переменной. Блокирование выполняется командой блокировки lock, например:

```
lock abc(123)
lock ^ABC(456)
lock ^|"%SYS"|COMMON(789)
```

Для большинства M систем нет разницы между локальными и глобальными блокировками. Обычно разработчики выбирают, какие имена необходимо блокировать - локальные или глобальные и далее пользуются внутренними соглашениями.

При блокировании локальных и глобальных имен, даже если они совпадают, производятся различные блокирования, поскольку локальное имя не равно глобальному.

Различие между локальными и глобальными именами блокировок возникает при использовании распределенных систем, когда сервера M систем объединены и процессы одной M системы могут использовать базу данных другой, удаленной M системы. В этом случае блокирование глобального имени, соответствующего удаленной системе, выполняется на том сервере, где должна храниться эта глобальная переменная, если бы она существовала. При этом блокировки с локальным именем будут храниться на том сервере, где выполняется процесс. И в этих двух случаях M системы ищут блокировки в разных областях, так как они находятся на разных серверах.

Поэтому, если два процесса должны синхронизироваться по доступу к глобалу, то обычно выбирается блокирование глобального имени, а если должны синхронизироваться по доступу к строго локальному ресурсу, то выбирается блокирование локального имени.

Несмотря на то, что аргументом блокировки является локальное или глобальное имя, это имя никак не связано с точно таким же именем локальной или глобальной переменной соответственно. Это разные объекты M систем. Наличие или отсутствие блокировки, установленной в одном из процессов, другим процессам никак не мешает получить доступ к этой переменной чтобы прочитать или записать, а также не требуется само существование таких переменных.

Блокировки необходимы лишь для взаимной синхронизации выполнения процессов. То есть два или более процесса, которые должны иметь корректный доступ к глобалам, должны установить необходимые блокировки, а потом их снять. Или, другими словами, управление должно пройти через команды блокирования.

Для блокировок в M системах действует простое правило: два или более процессов не могут захватить блокировки имен, если эти имена являются вложениями друг друга в иерархии дерева имени или в точности совпадают. Не имеет значения, какое из имен было захвачено первым, более длинное или более короткое. Иначе могут.

Примеры взаимоисключающих блокировок:

```
^ABC и ^ABC(123)
^DATA и ^DATA
```

Здесь в первом случае имена находятся в отношении иерархии, во втором случае совпадают.

Примеры взиморазрешающих блокировок:

```
^ABC и ^DATA
^DATA(3) и ^DATA(5)
```

Здесь в первом случае имена не совпадают, во втором случае имена не находятся в отношении иерархии.

В частности, из соглашений об именовании блокировок вытекает возможность реализовать блокирование на чтение-запись. Это такой вариант блокирования объекта, при котором доступ может получить либо лишь один пишущий процесс и никто более, либо один или более читающий и ни одного пишущего.

Для реализации такой стратегии для пишущего процесса блокируемое имя совпадает с именем ресурса, а для читающего добавляется к имени номер его процесса.

```
Имя ресурса:
^DATA("year",45,"details")
```

```
Блокирование на запись:
lock ^DATA("year",45,"details")

Блокирование на чтение:
lock ^DATA("year",45,"details",$J)
```

Здесь каждый из блокирующих на чтение использует имя, параллельное другому блокирующему на чтение и эти имена находятся в иерархии с именем блокирования на запись.

При необходимости выполнить синхронизацию доступа к ресурсам, не являющимся глобалами, разработчикам необходимо договориться о трансформации имени ресурса в локальное или глобальное и использовать уже их.

Как именно выполняется синхронизация процесса: команда блокирования lock проверяет возможность выполнить блокирование и в случае возможности вносит в имена блокировок новую блокировку, процесс продолжает выполнение. Если блокирование невозможно и одна или более имеющихся блокировок запрещают существование новой, то команда lock приостанавливает выполнение процесса до наступления условия возможности блокирования либо до истечения времени ожидания блокировки, если оно было установлено.

Блокировки выполняют как бы операторные скобки окружения таким образом, что между командами lock выполняемый процессом код гарантирован от того, что другой процесс получит доступ к тем же переменным. Разумеется, оба процесса должны использовать одинаковые соглашения об именах блокировок. Например, при выполнении кода

```
lock ^A
set value=^A
set value=value+1
set ^A=value
lock
```

для каждого из процессов, использующих синхронизацию, гарантировано получение следующего номера, для каждого из процессов своего, и не получится так что пока один процесс вычислял новое значение, другой уже также перезаписал его, вычислив на основе того же значения, и оба бы в результате использовали одинаковое значение value.

Команда блокирования имеет два варианта - полную и инкрементную форму. В вышеприведенных примерах использовалась полная форма. В этом варианте перед тем как выполнить блокирование указанного имени команда снимает все имевшиеся блокировки. Например, при первом прочтении кода

```
lock ^A,^B,^C
```

может показаться, что команда применена к трем именам и после ее выполнения будут блокированы все три имени, но это не так. Сначала команда снимет все блокировки, потом выставит блокировку на ^A, затем снимет ее, выставит блокировку на ^B, затем снимет ее и выставит блокировку на ^C. И в результате будут блокированы не имена ^A + ^B + ^C, а только имя ^C.

Для того, чтобы снять все блокировки, используется безаргументная форма команды:

```
lock
```

поэтому надо быть внимательным к числу пробелов, поскольку последующее имя может оказаться синтаксически значимой для M конструкцией.

Для того, чтобы добавлять или снимать блокировки независимо от уже существующих имен блокировок, команда lock используется в инкрементной форме. Для добавления блокировки инкрементно перед именем ставится символ +, для снятия инкрементно перед именем ставится символ -. Например

```
lock +^A
set value=^A
set value=value+1
set ^A=value
lock -^A
```

В этом случае блокирование имени ^A никак не затрагивает остальные блокированные имена.

Что интересно, инкрементная форма блокирования использует внутренний счетчик блокирования таким образом, что инкрементная блокировка при первом блокировании создает имя со счетчиком 1, а при последующих увеличивает счетчик этого имени на 1. При инкрементном снятии блокировки счетчик уменьшается на 1. Если счетчик становится равен 0, то блокировка снимается совсем и блокируемое имя удаляется из имен блокировок.

Такое определение поведения инкрементных блокировок позволяет создавать библиотечные подпрограммы, которые должны оперировать лишь определенной им частью системы, не затрагивая остальную.

По своему общему назначению блокировка - это объект синхронизации процессов. Для того, чтобы определить необходимо ли использовать

блокировку при доступе к какому-либо ресурс, нужно определить, присутствует ли признак одной из ситуаций необходимости блокирования. По своему назначению блокировки могут использоваться для случаев:

1. Доступ к ресурсу, который по своему физическому устройству может быть использован строго монопольно только одним процессом. Например, процессы могут определить, что COM порт занят именно в прикладной программе по соглашению об имени соответствующей блокировки, не обращаясь к команде открытия порта.

2. Изменение данных таким образом, что новое состояние зависит от предыдущего состояния этой же записи или от состояния других записей.

Наиболее часто блокировки используются для второй задачи. Предположим, что процессу необходимо получить следующее значение идентификатора. Для этого берется предыдущее, увеличивается на единицу и записывается новое. Полученное значение используется далее в программе. Например:

```
NextId()
 n id
 l +^DATA
 s id=$g(^DATA)+1
 s ^DATA=id
 l -^DATA
 q id
```

Здесь программа меняет значение записи в глобале на основе предыдущего значения этой же записи. Если не выполнять блокирование, то между операцией получения предыдущего значения и записью нового вычисленного может произойти такая же операция в другом процессе, и действия одного из этих процессов будут утрачены и процессы будут оперировать не взаимосогласованными значениями.

Другим примером является построение индексной записи. В этом случае значение записи в индексном глобале выполняется на основе записи в глобале, содержащем индексируемые данные. Если не выполнять блокирование, то другой процесс может получить из индексных записей информацию, не соответствующую реально существующим данным.

По умолчанию команда блокирования ожидает возможности блокировки бесконечное время, хотя корректнее говорить о неограниченном времени. Во многих практических задачах обычно можно определить некое разумное время ожидания блокировки, по истечении которого

необходимо принять решение о продолжении выполнения программы и о передаче управления на другую команду.

Время ожидания блокировки указывается после имени блокировки:

```
lock name:timeout
```

где время ожидания указывается в секундах.

В зависимости от реализации или версии M системы может распознаваться дробное значение таймаута. Например, в MiniM поддерживается указание с точностью до миллисекунд. При использовании конкретной M системы нужно проверить, если это необходимо, поддерживается ли дробное число в качестве таймаута.

Одна из основных задач таймаута - определить либо то, что, возможно, было возникновение дедлока и дальнейшее выполнение программы невозможно, либо что другой процесс не выполнил отпускание блокировки и сделал блокирование невозможным.

Если указан таймаут ожидания блокировки, то M система приостанавливает выполнение процесса либо до получения блокировки либо до истечения таймаута. В случае если блокирование оказалось неуспешным, то для процесса взводится значение системной переменной $TEST в значение 0. Если успешно, то в значение 1.

Одно из простых правил разработки на M состоит в том, что если был указан таймаут ожидания для какой-либо операции, то следующей выполняемой командой обычно должна стоять команда проверки значения $TEST, например командой IF или командой ELSE.

Все установленные процессом блокировки принадлежат этому процессу. В случае если процесс завершился каким-либо образом, аварийно или самостоятельно командой HALT, и не снял установленные им блокировки, то M система автоматически снимает все установленные им блокировки.

Механизм снятия блокировок в этом случае работает в зависимости от примененной MUMPS системы, в разных системах он может быть выполнен по-разному и может содержать некоторый период, в течении которого блокировка может существовать, но принадлежать несуществующему процессу. В определенной степени верно то, что при аварийном завершении процесса данные, принадлежащие ему, но находящиеся в общей области памяти сервера, в любом случае какое-то время продолжают там находиться и не удаляются в тот же самый физический момент времени. Традиционно MUMPS системы стремятся сократить этот период очистки до физически предельного минимума с тем, чтобы при запуске следующего процесса, в случае если его номер совпадет

с закончившимся, блокировки не были переданы новому процессу как нежданное наследство.

4.3 Транзакции

Для MUMPS систем, как для СУБД, одним из весьма практичных механизмов является механизм транзакций. Первоначально, когда язык MUMPS был только стандартизирован, механизм транзакций и его поведение не были определены. При этом для MUMPS были разработаны огромное количество программ, выполняющих критически важные задачи. И, для совместимости с предыдущими реализациями по поведению, те M системы, которые поддерживают транзакции, поддерживают их так, что если процесс не выполнил команды начала транзакции, то по умолчанию он работает вне транзакционных скобок.

M системы поддерживают, как минимум, команды начала транзакции, подтверждения транзакции и отката транзакции. В зависимости от внутренней архитектуры, используются ли оптимистические или пессимистические блокировки данных, система может поддерживать также команду повтора транзакции. Если M система не поддерживает транзакции, либо какую-либо из команд, то при попытке выполнить такую команду должна генерироваться ошибка о том, что эта команда не поддерживается.

Например, системы MiniM и Caché реализуют команды tstart, tcommit и trollback и парную им системную переменную $tlevel, но на команду trestart генерируют ошибку о том, что команда не поддерживается. Система GT.M в силу внутренней архитектуры дополнительно поддерживает команду trestart и парную ей системную переменную $TRESTART. Более старые системы MSM могут не генерировать ошибку при выполнении транзакционных команд, но при этом команды могут не выполняться, поскольку система так и осталась недоработанной в отношении транзакций. Система M3-Lite при использовании транзакционных команд также ничего не делает и системная переменная $tlevel всегда возвращает значение 0. Таким образом, при необходимости использовать транзакции разработчики должны проверить характер и, возможно, особенности поведения транзакций на используемой M системе.

Для M системы транзакция - это с одной стороны состояние процесса, с другой стороны совокупность сделанных от начала транзакции изменений в глобалах.

По умолчанию при старте процесса его уровень транзакции 0, что означает что сделанные в этом состоянии изменения не откатываются.

При каждом вызове команды tstart уровень транзакции увеличивается на единицу. Начиная со значения 1 изменения в глобалах могут быть возвращены в состояние на начало транзакции, когда значение уровня транзакции было 0. Произвольные библиотечные функции и подпрограммы, таким образом, могут просто использовать транзакционные скобки tstart - tcommit по необходимости, не влияя на контекст транзакции вызвавшего их кода.

При каждом вызове команды tcommit система уменьшает на единицу уровень транзакции. При снижении до 0 процесс переходит снова в состояние вне транзакции.

При вызове команды trollback M система откатывает все изменения глобалов сделанные процессом от входа в состояние транзакции 1, в предыдущее состояние и переводят уровень транзакции в состояние 0. Например, если есть код

```
s ^a(0)=0
tstart  ; $tlevel = 1
s ^a(1)=1

tstart  ; $tlevel=2
s ^a(2)=2
tcommit ; $tlevel=1

tstart  ; $tlevel = 2
s ^a(3)=3
trollback  ; $tlevel = 0
```

то команда rollback вернет предыдущее состояние и для ^a(3) и для ^a(2) и для ^a(1), поскольку все они были изменены в состоянии $tlevel>0, но не вернет предыдущее состояние для ^a(0).

Пока выполняется изменение глобалов в контексте транзакции, все эти изменения могут быть видны другим процессам или не видны. Системы MiniM и Caché работают в режиме видимости всех сделанных изменений, то есть в пессимистической блокировке. Система GT.M может поддерживать оптимистическую блокировку и другие процессы могут не видеть изменений, сделанных в контексте транзакций.

Первый из этих случаев ориентирован на большую работу при откате сделанных изменений (trollback) и имеет легковесное подтверждение (tcommit), второй ориентирован на большую работу при подтверждении сделанных изменений (tcommit) и легковесный откат (trollback) и, в принципе, может потребовать очень больших ресурсов как оперативной памяти, так и дисковой.

При использовании второго случая разработчики должны соразмерять операции, которые предстоит выполнить их прикладной системе, с

аппаратными возможностями сервера, на котором программа будет выполняться. Традиционным решением проблемы в таких случаях является разбиение выполняемых действий, одной длинной транзакции, на несколько коротких транзакций. С другой стороны, это, теоретически, уже не транзакция, и нет необходимости входить в такой режим. Обычно проблема длинных транзакций возникает в тех СУБД, где нет возможности отключить контекст транзакции или переключиться на пессимистическую блокировку.

В техническом отношении режим оптимистической блокировки выполняется ведением версий данных, а режим пессимистической блокировки выполняется ведением отката по журналу выполненных действий. Таким образом, для корректной работы транзакций во втором случае (как в MiniM и в Caché) необходимо, чтобы работало журналирование для используемой базы данных.

Интересной особенностью систем MiniM и Caché является возможность на время отключить и снова включить журналирование для процесса. При его выключении не для базы данных, а для процесса, сделанные им изменения попадают в базу данных, но не попадают в журнал. А, поскольку откат транзакции работает по журналу, все изменения, сделанные при временном выключении журналирования, не откатываются.

Другой интересной особенностью, поддерживаемой Caché, но не поддерживаемой MiniM, является возможность административно явно указать, для каких именно глобалов в одной и той же базе данных использовать или не использовать журналирование.

При изменении глобалов процесс может выполнить всего 4 различных с точки зрения транзакционного механизма действия:

1. Создать запись переменной, которая не существовала.

2. Перезаписать существовавшую переменную.

3. Удалить существовавшую переменную.

4. Удалить не существовавшую переменную.

В первом случае откат транзакции возвращает переменную в состояние неопределенного значения, но только для именно этого имени, не удаляя вложенных имен. Во втором случае откат возвращает предыдущее значение. В третьем случае откат возвращает существование переменной и ее предыдущее значение. Если было удаление с вложенными именами, то они также возвращаются в свое значение.

Четвертый пункт ни в MiniM, ни в Caché, ничего не делает. Глобалы являются деревьями и операция удаления это удаление также всех вложенных переменных. Но в журнале среди записей, выполненных процессом, не содержится, с какими из них выполнялись операции, поскольку при удалении несуществовавшей переменной удалять было нечего. Если в течении транзакции после удаления несуществовавшей переменной другой процесс запишет то-то в этот глобал, то эти записи останутся и глобал не будет переведен в состояние несуществующего.

В отношении четвертого пункта у различных СУБД может быть расхождение во мнениях, то необходимо предпринять - ничего не делать или восстановить значение в неопределенное. В случае MUMPS систем переменные являются деревьями и восстановление в неопределенное значение лишь одного корневого имени без его дочерних имен выглядело бы неестественным и не цельным действием. Удаление же тех узлов, которые процесс не изменял, означает изменение тех узлов, которые не были затронуты, что противоречит принципу отката транзакции откатывать именно выполненные процессом изменения.

Для MUMPS систем, поддерживающих транзакции, поддерживается поведение по умолчанию для завершения процесса, если он завершился в контексте транзакции. В зависимости от версии и реализации это могут быть либо откат сделанных изменений, либо игнорирование отката, что в силу пропадания процесса эквивалентно подтверждению сделанных изменений. Также поведение MUMPS системы может зависеть от архитектуры, используется ли оптимистическая или пессимистическая блокировка.

4.4 Блокировки в транзакциях

MUMPS системы, реализующие транзакции, имеют особенность поведения блокировок, выполненных в транзакциях. Эти два механизма выполнены взаимосвязанно, хотя, казалось бы, на первый взгляд это различные механизмы.

Если выполняется блокирование глобалов, то для СУБД это означает, что производится важная для взаимосогласованности данных операция. Если эта операция выполняется в контекте транзакции, то выполненные изменения могут быть впоследствии либо отменены целиком, либо подтверждены целиком.

Оба эти механизма, и транзакции, и блокировки, ориентированы на корректное выполнение конкурентного доступа к данным. Поэтому, для того, чтобы дать другим процессам доступ либо к подтвержденным, либо

к отмененным данным, транзакционные MUMPS системы удерживают блокировки до окончания транзакции независимо от команды удаления блокировки, если блокировка была выполнена в транзакционном контексте.

Удержание блокировки выражается в том, что при выполнении команды снятия блокировки в действительности эта блокировка не снимается физически, а лишь маркируется для снятия по окончании транзакции.

Различные реализации MUMPS могут по-разному выполнять нюансы этого механизма, поскольку стандартом не описано точное поведение системы при удержании блокировки до окончания транзакции. Например, некоторый список отличий:

1. Система MiniM удерживает блокировку до конца транзакции, если она была установлена при ненулевом $tlevel. Система Caché удерживает любые блокировки, пока текущий $tlevel не нулевой, в том числе установленные при нулевом $tlevel.

2. Система MiniM при команде снятия всех блокировок (безаргументный lock) физически снимает блокировки независимо от контекста транзакции. Система Caché продолжает удерживать блокировки при безаргументном lock.

Кроме того, различные MUMPS системы могут давать специфичный от реализации механизм принудительного физического удаления блокировок, в том числе установленные другим процессом.

В любом случае, различные реализации MUMPS систем ориентированы на типовое использование блокировок в транзакциях в инкрементной форме. При написании типового кода разработчики на M получают наиболее ожидаемое поведение СУБД и конкурентного доступа для различных процессов.

Механизм взаимозависимости блокировок имеет очень важное следствие в том, что касается массовой обработки данных в транзакции. Для пояснения нужно сделать небольшой отступ к упрощенной классификации СУБД с нечеткими границами между различными типами СУБД:

1. Малые базы. Данных так мало, что для данных не требуются ни специальные форматы, ни специальные алгоритмы. Например, для хранения данных могут использоваться плохоструктурированные с точки зрения СУБД форматы - INI, XML.

2. Средние базы. Данных уже много, для них применяются специальные форматы, характерные для СУБД, и зачастую индексные структуры и алгоритмы. Например, форматы DBF.

3. Большие базы. Данных столь много, что к специальным алгоритмам добавляются специальные методы кеширования и подкачки, поскольку объем обрабатываемых за одну операцию данных превышает размер оперативной памяти.

4. Очень большие базы (VLDB). К функциям СУБД добавляется кеширование и подкачка также и служебных данных, поскольку и их объем также не умещается в оперативной памяти.

Здесь важным пунктом является четвертый. Блокировки, по сути, являются служебными данными, существующими временно. В СУБД класса VLDB служебные данные изначально предполагаются в таких объемах, что их сами необходимо хранить на диске и кешировать. Например, служебные данные о блокировках могут быть записаны в служебные поля базы данных или в отдельном файле. В определенной степени суждение о том, то задача относится к классу VLDB, определяется соотношением объемов обрабатываемых данных и аппаратными возможностями. Первые реализации VLDB систем обрабатывали такие объемы данных, с которыми современные 64-битные компьютеры могут справляться совершенно непринужденно, имея объем оперативной памяти больше, чем объем дисковой для тех первых систем.

Разработчики должны понимать, что в настоящее время MUMPS системы не относятся к СУБД класса VLDB, поскольку блокировки хранят строго в оперативной памяти, в пределах указанных администратором в настройках сервера.

Важным следствием является то, что разработчикам для выполнения в транзакции массового изменения данных необходимо учитывать общее ограничение на объем блокировок. Это является общей проблемой не MUMPS систем, а различных СУБД в целом, вне зависимости от архитектуры.

Традиционно разработчики выбирают один из вариантов:

1. Выполнение массовых операций в специальном внетранзакционном контексте.

2. Выполнение массовых операций без блокировок.

3. Разбиение массовой операции на меньшие порции.

Еще одним внесистемным решением проблемы большого числа блокировок при выполнении массовых операций является метод эскалации блокировок. Существуют также такие реализации СУБД, которые поддерживают этот метод автоматически. Суть метода эскалации состоит в

том, чтобы, немного нарушив принцип презумпции блокирования, заменить большое число детальных блокировок на меньшее число логически включающих их укрупненных блокировок.

Предположим, что для выполнения действия необходимо блокирование большого числа однородных узлов

```
^DATA("abc",123)
^DATA("abc",456)
^DATA("abc",789)
^DATA("abc",...)
^DATA("def",123)
^DATA("def",456)
^DATA("def",789)
^DATA("def",...)
^DATA("tyu",123)
^DATA("tyu",456)
^DATA("tyu",789)
^DATA("tyu",...)
...
```

Этот список мы можем заменить на меньшее число блокировок более высокого уровня

```
^DATA("abc")
^DATA("def")
^DATA("tyu")
...
```

Либо на совсем радикальную общую блокировку

```
^DATA
```

Соответственно, в методе эскалации различают решение по степени детализации эскалирования, по возможности попадания процессов в дедлок и по возможности выполнить эскалацию в автоматическом режиме. В разработках на MUMPS обычно разработчики не применяют метод автоматической эскалации из-за трудоемкости ее выполнения и существенной зависимости снятия блокировок в контексте транзакции от реализации MUMPS системы, а реализуют работу массовой операции в контексте изначально укрупненных блокировок.

При необходимости обеспечить целостность всей базы до и после массовой операции разработчики могут использовать замену транзакций на бекап. При выполнении бекапа до массового перестроения данных у администратора существует возможность вернуть базу данных в предыдущее корректное состояние в случае сбоя.

4.5 Функция $INCREMENT

Транзакции с их возможностью выполнить отмену произведенных действий это совсем не безобидный механизм, просто сам по себе что-то улучшающий, и его нельзя просто добавить к коду, разработанному для контекста без транзакций.

Рассмотрим для примера операцию добавления записи. При добавлении записи необходимо получить ее очередной номер, или суррогатный ключ:

```
NextId()
 n id
 l +^DATA
 s id=$g(^DATA)+1
 s ^DATA=id
 l -^DATA
 q id
```

Этот код вполне корректен для работы вне транзакций. Пока мы оперируем арифметикой, вычисляем значение следующего идентификатора, записываем, код блокирует глобал и несколько процессов получают каждый для себя следующий номер.

Но этот код некорректен с точки зрения работы в транзакции. Если первый процесс получит значение 1, второй процесс значение 2, а третий процесс значение 3, то при выполнении вторым процессом отката транзакции произойдет возвращение значения

```
^DATA
```

к значению, бывшему перед получением значения 2, то есть откат транзакции установит этот счетчик в значение 1.

Следующий, четвертый процесс, получит соответственно номер 2, а пятый номер 3, что очевидно не соответствует корректной работе функции получения следующего номера суррогатного ключа.

Для корректной работы конкурентного доступа процессов в контексте транзакции СУБД традиционно предусматривают механизм, парный к механизму отката транзакции, или операции, действие которых откатом транзакции не возвращается назад.

В MUMPS системах наиболее часто используемой операцией, для которой требуется невозврат значения в предыдущее состояние, является операция получения следущего идентификатора. В других СУБД также предусматриваются различные аналогичные механизмы, например в Oracle это sequence, в INTERBASE это generators.

Для СУБД, основанных на SQL, также зачастую поддерживается функция автоинкрементного поля как упрощенный и специализированный для прикладного применения механизм. Автоинкрементные поля автоматически совмещают в себе как безоткатное в транзакции получение следующего идентификатора, так и неявную систему именования таких идентификаторов. Те СУБД, которые поддерживают автоинкрементные поля, также должны поддерживать и механизм получения последнего сгенерированного текущим процессом такого значения, либо, если было автоматическое увеличение для нескольких таблиц, то получение полученного номера для указанной таблицы.

Технически в MUMPS системах безоткатное действие выполняется в виде системной функции $increment(). Функции указывается переменная и необязательный второй аргумент, на сколько следует увеличить значение. Если второй аргумент не указан, то функция увеличивает на 1. Если такой переменной не было или было нечисловой значение, то функция считает что было значение 0 и создает переменную с новым значением. В большинстве случаев разработчиками используется одноаргументная форма функции.

Вышеприведенный код с применением этой функции уже может выглядеть так:

```
NextId()
 n id
 l +^DATA
 s id=$increment(^DATA)
 l -^DATA
 q id
```

И уже такой вариант обеспечивает корректный сценарий получения следующего идентификатора для различных процессов, в том числе выполняющих откаты транзакций. Но, в силу того, что эта операция особенная, MUMPS реализации не требуют выполнения отдельной специальной блокировки используемой переменной и выполняют корректный атомарный доступ с выполнением арифметических операций самостоятельно. Также корректным поэтому является такой код:

```
NextId()
 q $increment(^DATA)
```

При выполнении функции $increment MUMPS реализация гарантирует, что, пока эта операция выполняется, ни один другой процесс не сможет изменить это значение даже если будет устанавливать блокировки или также одновременно выполнит $increment. Во втором случае

система обеспечивает строго последовательное выполнение увеличения значения и оба процесса получат различные значения идентификаторов.

В техническом отношении функция $increment предусматривает явное задание значения, на сколько следует увеличить значение переменной, и, в том числе, это приращение может быть дробным, нулевым и отрицательным.

Важным моментом для более глубокого понимания работы функции $increment является то, что ее нельзя полноценно заменить на отключение журналирования. В основную формулировку входит требование невозврата значения при откате. И, казалось бы, если разработчик отключит журналирование на период вычисления нового значения, то эта запись не попадет в журнал и откат транзакции не выполнит возврат для инкрементируемого значения.

С точки зрения работы прикладной системы такая замена может оказаться корректной заменой, но с точки зрения работы сервера - нет. Если мы будем выполнять восстановление базы данных после сбоя, то часть данных сервер берет из бекапа, а часть данных дописывается из журнала в том, что касается изменений, еще не попавших в базу данных из журнала. Поэтому, если в журнале не будет записей об изменении инкрементируемой переменной, то она может быть восстановлена на одно из предыдущих состояний по файлу бекапа, но счетчик не восстановится в корректное состояние соответствующее уже использованным номерам идентификаторов, содержащимся в журнале.

Поэтому, для корректного выполнения операции $increment, в журнал пишется специальная запись, которую игнорирует операция отката транзакции, но не игнорирует операция восстановления базы данных по журналу. И в запись пишется не предыдущее значение, а значение, которое получено после изменения переменной.

Для тех MUMPS систем, которые используют автоматическую репликацию по журналу, например Caché, также важно использовать именно функцию $increment, а не отключение журналирования, поскольку результат инкремента переменной также должен быть доставлен в реплицируемую базу.

Одним из важных выводов для разработчиков на M при использовании транзакций является то, что следует внимательно пересмотреть операции получения идентификатора и стратегии использования переменных для их хранения. Нужно иметь в виду также и тот факт, что смешивание операций set и $increment никак не пресекается самой СУБД. Если происходит их смешивание, то система при откате вернет значение, бывшее предыдущим для set, несмотря на то, что в транзакции также была использована и операция $increment.

И еще одним немаловажным фактом применения функции $increment является то, что она не входит в стандарт языка MUMPS и стандарт, хотя и предусматривает откат транзакций, но не предусматривает механизма безоткатных изменений. Сама функция $increment введена в различные реализации MUMPS систем их производителями самостоятельно и согласованно между собой, опираясь на наиболее приемлемые практики и методики.

4.6 Функция $BIT

При использовании битовых индексов в MUMPS системах важным для понимания работы системы и ее применения является дополнительный к механизму битовых функций механизм конкурентного доступа, выполненный специально для них.

Битовый индекс представляет собой совокупность сегментов, каждый из которых это последовательность байт, и каждый бит в ней значим, может иметь значение либо "объект с идентификатором, равным номеру бита, существует" (1), либо "не существует" (0). Кроме того, применяется правило умолчания, что если бит находится за пределами реально физически существующих байт, либо строка байт вообще не существует, то логически для битовых операций это эквивалентно последовательности нулей.

В техническом отношении битовые операции могут быть добавлены к MUMPS системе внешними по отношению к ней функциями в динамической библиотеке. Автору довелось участвовать в проектах, где использовался именно такой вариант. Разработки блестяще работали в режиме OLAP и имели некоторые непреодолимые недочеты в режиме OLTP.

Если операции с битами выполняются внешними по отношению к MUMPS системе средствами, то для записи в глобал остается использовать операцию set. Для MUMPS системы это операция полной перезаписи всего значения. Конечно, есть MUMPS системы в которых применяется дифференциальное журналирование, например как в MiniM, и в журнал записывается по возможности лишь изменение строки байт, а не вся строка, но в целом, вообще говоря, журналируется именно операция set.

В логическом отношении значимым изменением является изменение одного бита, а физически это для MUMPS системы целый большой полноценный set. Это приводит к двум проблемам:

1. Для простановки одного бита требуется взять полное значение сегмента, изменить в нем 1 бит и записать новое полное значение сегмента.

2. Для MUMPS системы видна лишь операция set.

Эти проблемы приводят к следующим последствиям: первая требует блокировать весь сегмент на время изменения, чтобы другой процесс не перезаписал значение сегмента ранее и его изменения не были утрачены. Поэтому возникает конфликт блокировок - хотя разным процессам требуется изменить разные биты, им необходимо использовать одно и то же имя блокировки. При высоконагруженной работе это приводит к увеличению вероятности взаимоблокировок. Вторая проблема приводит к большому объему журналирования, хотя из значимых изменений - всего один бит.

Эти две проблемы структурно не являются характерными для какой-либо определенной архитектуры или типа СУБД. В случае применения битовых индексов в любой другой системе они также существуют. Можно увидеть в рекомендациях, в том числе и для других типов СУБД, рекомендации использовать битовые индексы лишь для хранилищ данных, приближенных по своему режиму работы к режиму read-only, и рекомендации по возможности не использовать битовые индексы для задач класса OLTP.

В современных MUMPS системах, таких как Caché и MiniM, эти обе проблемы были решены на уровне СУБД введением двух дополнительных механизмов, работающих при использовании битовых функций:

1. При изменении бита система выполняет эту операцию атомарно, с внутренней синхронизацией, не попадающей в множество блокировок команды lock, и не удерживающейся до окончания транзакции.

2. При изменении бита система использует специальную запись в журнале.

В совокупности эти обе меры приводят к тому, что один единичный бит может быть проставлен независимо от других и при откате транзакции именно этот бит будет возвращен в предыдущее состояние, даже если другие процессы продолжают изменять битовую строку. Точно также значимые биты будут проставляться по отдельности при восстановлении из бекапа с дополнением по журналу.

Выполненные архитектурные меры по отношению к функции $bit снимают необходимость использовать блокировки индексных структур

при перестроении битмап индексов, хотя в примерах в целях методологии они могут присутствовать.

Кроме того, при использовании битовых индексов в современных реализациях MUMPS систем также снимается рекомендация не использовать битмап индексы в OLTP задачах, а использовать по возможности только в OLAP задачах. В силу транзакционности таких битмап индексов они точно также применими в любых OLTP задачах, как и индексы других типов. Это дает разработчикам свободу выбора при построении качественно других прикладных систем.

Точно так же, как и в случае с функцией $increment, разработчики должны разделить глобалы на те, к которым применяются операции $bit и те, к которым применяется прямое изменение другими формами команды set. В случае смешивания способов изменения байтовой строки MUMPS система будет журналировать именно использованную операцию вне зависимости от того, было ли изменение этой же строки иными способами.

Нужно отметить, что функции семейства $bit не являются частью стандарта MUMPS, соглашения принятые в одних системах, могут не поддерживаться в других. Этот функционал не входит в уровень переносимости. Кроме того, различные MUMPS системы могут использовать различные методы компрессии и кодирования битовых строк для уменьшения общего объема хранения. При переносе данных, таким образом, нельзя переносить битмап индексы как есть, их необходимо перестроить на целевой системе заново.

4.7 Дедлоки

Конкурентный доступ нескольких процессов к одним и тем же данным содержит крупную неприятность, называемую отдельным термином мертвой блокировки. Алгоритмически эта проблема не является спецификой MUMPS систем или систем основанных на других архитектурах или методах, а характерна для конкурентного доступа как такового.

Дедлоки (deadlock), или мертвая блокировка - это явление, событие или состояние двух или более процессов, один из которых, заблокировав ресурс 1, ожидает освобождения ресурса 2, но ресурс 2 в свою очередь заблокирован другим процессом, и он ожидает освобождения ресурса 1. Цепочка взаимоблокировок может быть более длинной, с вовлечением нескольких процессов.

Для иллюстрации примера приведем код на MUMPS, условно воспроизводящий ситуацию с обновлением нескольких (для простоты двух)

объектов, имеющих два простых атрибута, имеющих всего два значения. Функция action выполняет обновление как строки данных

```
^zAug("data")
```

так и двух индексов

```
^zAug("prop1")
^zAug("prop2")
```

Функция run1 выполняет имитацию изменения объектов.

Основной задачей примера является поиск решения проблемы с deadlock, возникающим при работе с битовыми индексами, в который вероятность конфликта доступа при обновлении индексов возрастает в тысячи раз. Поэтому в приведенном примере воспроизведения deadlock блокировка узла данных не выполняется.

```
run1()
 n id,prop1,prop2
 f  d
 . w "."
 . s id=$r(2),prop1=$r(2),prop2=$r(2)
 . d action1(id,prop1,prop2)
 q
action1(id,prop1,prop2)
 n oldprop1,oldprop2
 ;
 s oldprop1=$lg($G(^zAug("data",id)),1)
 s oldprop2=$lg($G(^zAug("data",id)),2)
 ;
 l:oldprop1'="" +^zAug("prop1",oldprop1,id)
 l:oldprop2'="" +^zAug("prop2",oldprop2,id)
 l +^zAug("prop1",prop1,id)
 l +^zAug("prop2",prop2,id)
 ;
 k:oldprop1'="" ^zAug("prop1",oldprop1,id)
 k:oldprop2'="" ^zAug("prop2",oldprop2,id)
 ;
 s ^zAug("prop1",prop1,id)=""
 s ^zAug("prop2",prop2,id)=""
 ;
 s ^zAug("data",id)=$lb(prop1,prop2)
 h .5
 ;
 l:oldprop1'="" -^zAug("prop1",oldprop1,id)
 l:oldprop2'="" -^zAug("prop2",oldprop2,id)
 l -^zAug("prop1",prop1,id)
 l -^zAug("prop2",prop2,id)
 q
```

После запуска

```
d run1
```

в двух процессах они в течении некоторого времени уверенно встают в deadlock. Будем использовать этот код как базовый для модернизации с целью нахождения решения.

Для возникновения ситуации мертвой блокировки необходимо возникновение четырех условий:

1. Процессы требуют предоставления им права монопольного управления ресурсами, которые им выделяются (условие взаимоисключения).

2. Процессы удерживают за собой ресурсы, уже выделенные им, ожидая в то же время выделения дополнительных ресурсов (условие ожидания ресурсов).

3. Ресурсы нельзя отобрать у процессов, удерживающих их, пока эти ресурсы не будут использованы для завершения работы (условие неперераспределяемости).

4. Существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более ресурсов, требующихся следующему процессу цепи (условие кругового ожидания).

В задачах конкурентного доступа к базе данных все эти условия присутствуют. Для решения проблемы необходимо разорвать одно (или более) из перечисленных условий. Приведенное далее решение проблемы использует изменение программистом выполнения процесса так, чтобы разорвать второе условие. Решение состоит в разбиении имен ресурсов на группы, с тем, чтобы блокировать имена ресурсов одной группы списком (или одновременно), и имена групп ресурсов упорядочить. При блокировке списком отпадает наращивание блокировок той же группы в дополнение к тем, которые могут ожидаться другими процессами.

Общий метод выглядит примерно так: разбиваем ресурсы на иерархические группы. Назовем их группа1, группа2, ... группаN. При этом вводится дисциплина блокирования: ресурсы группы n+1 могут быть блокированы только если нет других блокировок группы n+1 или старше и если есть блокировка имен группы n или младше. Группой ноль будем считать отсутствие блокировок. В рамках одной группы блокировку выполняем списком.

Группы следует организовать таким образом, чтобы имена блокировок группы n+1 были известны после блокирования имен группы n. В примере используется инкрементальная блокировка списком чтобы удержать имевшиеся ранее блокировки и добавить к ним новые.

Текст примера для воспроизведения / тестов:

```
 run3()
 n id,prop1,prop2
 f  d
 . w "."
 . s id=$r(2),prop1=$r(2),prop2=$r(2)
 . d action3bit(id,prop1,prop2)
 q
 ; то же самое что action3 но имитируем битовые сегменты
action3bit(id,prop1,prop2)
 n str,old,oldprop1,oldprop2
 s str=$na(^zAug("prop1",prop1,1))_","_
    $na(^zAug("prop2",prop2,1))
 l +^zAug("data",id)
 s oldprop1=$lg(^zAug("data",id),1)
 s oldprop2=$lg(^zAug("data",id),2)
 ; это на случай запуска теста без созданных данных
 s:oldprop1="" oldprop1=" "
 s:oldprop2="" oldprop2=" "
 s old=$na(^zAug("prop1",oldprop1,1))_","_
       $na(^zAug("prop2",oldprop2,1))
 x "l +("_old_","_str_")"
 ; ничего не значит, это имитация.
 ; реально тут битовая операция сброса бита
 x "s ("_str_")="""""
 s ^zAug("data",id)=$lb(prop1,prop2)
 h 0.4
 ; тоже ничего не значит, это имитация.
 ; реально тут битовая операция установки бита
 x "s ("_str_")="""""
 x "l -("_old_","_str_")"
 l -^zAug("data",id)
 q
 ; обычный обратный индекс
action3(id,prop1,prop2)
 n str,old,oldprop1,oldprop2
 s str=$na(^zAug("prop1",prop1,id))_","_
     $na(^zAug("prop2",prop2,id))
 l +^zAug("data",id)
 s oldprop1=$lg(^zAug("data",id),1)
 s oldprop2=$lg(^zAug("data",id),2)
 ; это на случай запуска теста без созданных данных
 s:oldprop1="" oldprop1=" "
```

```
s:oldprop2="" oldprop2=" "
s old=$na(^zAug("prop1",oldprop1,id))_","_
   $na(^zAug("prop2",oldprop2,id))
x "l +("_old_","_str_")"
x "k "_old   ; ну просто совпало так
s ^zAug("data",id)=$lb(prop1,prop2)
h 0.4
x "s ("_str_")="""""  ; тут тоже удачно совпало
x "l -("_old_","_str_")"
l -^zAug("data",id)
q
```

Нетрудно убедиться, что замена блокировок списком на последовательные тут же приводит к ситуации deadlock в течении довольно короткого времени.

В приведенном примере две группы:

1. ^zAug("data",...) - узел данных
2. ^zAug("prop1",...) и ^zAug("prop2",...) - узлы индексов

Некоторый условный пример с более осмысленным содержанием: предположим, что есть объект документ с объектами - строки документа. Для операций с объектом делим операции и имена блокировок на 4 группы:

1. Сохранения документа, вход - идентификатор документа, дальше передаются имена для группы2 (имена индексов) и группы3 (идентификаторы объектов - строк документа).
2. Индексы документа.
3. Объекты строк документа, вход - идентификатор объекта строки, дальше передаются имена для группы4 (имена индексов по строкам).
4. Индексы строк.

Ставим блокировку на идентификатор документа, выполняем операции с объектом документа. По его содержанию определяем имена блокировок для индексов документа и объектов строк документа. Блокируя индексы, выполняем изменение индексов. Блокируя строки документа, выполняем операции со строками. По блокированным строкам можем определить имена индексов. Блокируем индексы, выполняем обновление индексов.

Также следует отметить, что при использовании в битовых операциях функций семейства $bit в современных версиях Caché и MiniM проблема deadlock автоматически снимается, поскольку операция

```
s $bit(^data,n)=0
s $bit(^data,n)=1
```

корректно отрабатывается без необходимости блокирования всего сегмента, и корректно работает откат транзакции. При использовании в качестве реализации битовых операций функций семейства $bit получаем выигрыш как в более свободной параллельной работе процессов, так и в существенном снижении объемов журналирования изменений индексных сегментов.

Другим решением проблемы дедлоков является разрыв четвертого условия, кольцевого ожидания. Для этого выполнение процессов планируется таким образом, чтобы захват ресурсов выполнялся в строго определенном отношении условных имен этих ресурсов. Для этого вводится правило упорядочивания ресурсов и дисциплина доступа, при которой захват ресурса должен производиться строго в определенном направлении этого упорядочения.

Для вышеприведенного примера, для исключения взаимоблокировки, нужно построить такой же список ресурсов, но блокировать не единым списком, а провести сортировку имен в некотором обще-оговоренном порядке и выполнять блокирование в порядке полученной очередности.

Большинство программных систем, в которых возможно возникновение взаимоблокировок, обычно применяют именно разрыв четвертого условия и упорядочивают захват ресурсов. При этом разработчики прорабатывают ход выполнения для каждой из операций и при необходимости корректируют алгоритмы так, чтобы сохранить порядок блокирований.

Для тех ситуаций, где применение методов разрыва условий неприменимо или применение решений слишком трудоемко, разработчики должны для избежания останова системы вводить компенсатор для второго условия (условие ожидания). Это выполняется указанием времени ожидания. Если по истечении определенного времени ресурс не был предоставлен, то процесс должен принять решение об откате выполняемого действия и либо счесть обнаруженное состояние непреодолимым, либо попытаться повторить операцию снова.

В MUMPS системах время ожидания указывается в одном из параметров команды lock:

```
lock +name:timeout
```

Разработчики всегда должны проверять состояние системной переменной $TEST, чтобы проверить, была ли блокировка с таймаутом успешной или нет, с помошью команд IF или ELSE, например:

```
lock +name:5 e  w "lock failed",!
```

Существуют также системы баз данных и СУБД, которые применяют компенсатор по таймауту автоматически, и при возникновении ситуации невозможности получить захват ресурса, автоматически откатывают транзакцию и повторяют ее действия снова. В случае если рестарт транзакций поддерживается используемой MUMPS системой, это также может быть использовано разработчиками.

При использовании рестарта транзакций может также возникнуть ситуация, когда при выполнении N рестартов операцию выполнить все равно не удалось. В этом случае разработчики на MUMPS должны использовать значение системной переменной $TRESTART, показывающей число рестартов, чтобы процесс мог принять решение стоит ли продолжать операцию снова. Вообще говоря, если алгоритм допускает дедлоки, то причина дедлоков не устраняется повторным запуском алгоритма, и дедлоки могут возникать при каждом его выполнении.

4.8 TSN

При выполнении транзакций СУБД должна реализовать механизм различения записей в журнале транзакций, какие записи к какой транзакции относятся. Для этого используется так называемый номер последовательности транзакций, или Transaction Sequence Number (TSN).

Каждая транзакция должна получить собственный уникальный номер, и этот номер должен отличаться как для различных процессов одного экземпляра сервера СУБД, так и между процессами сервера до его останова и после последующего старта. Второе условие необходимо для корректного продолжения журнала транзакций и процедуры автоматического восстановления сервера при старте.

К наиболее распространенным и достаточно устойчивым механизмам генерации номера транзакций относится алгоритм основанный на времени генерации с синтетическим номером выравнивания. В номер транзакции входит, таким образом, два числа - дата с временем и синтетическое число. Общий алгоритм состоит в следующем:

1. Сервер ведет используемую дату и синтетический номер, эти данные используют процессы сервера для генерации нового значения TSN.

2. При старте сервера сервер отыскивает в журнальных записях наибольший из последних номеров TSN и выделяет в нем дату и синтетический номер.

3. Если текущая дата больше последней использованной, то она становится используемой и синтетический номер обнуляется.

4. Если текущая дата меньше или равна используемой, то для генерации TSN используется используемая, а синтетический номер увеличивается.

5. При переполнении синтетического номера используемая дата увеличивается на одну секунду и синтетический номер обнуляется.

При такой схеме генерации значения TSN поддерживается правило их непрерывного возрастания, что позволяет корректно использовать журнальные записи, оставшиеся от предыдущего сеанса работы сервера.

Кроме того, механизм генерации основанный на добавлении к текущей дате синтетического номера застрахован от одинаковых значений TSN, полученных при одинаковом значении времени для различных процессов.

Наиболее важным следствием и задачей алгоритма является устойчивость к переводу времени на компьютере как в течении сеанса работы сервера, так и между сеансами. Такой перевод времени может выполняться как вручную для исправления ошибки хода часов, так и автоматически при смене летнего и зимнего отсчета времени.

Задачей синтетического дополнения TSN является ожидание даты, когда текущая дата и время компьютера догонит используемое. В случае переполнения синтетического числа к используемому времени автоматически добавляется секунда и синтетический довесок обнуляется. Таким образом, если на компьютере был произведен перевод часов на N секунд назад, то сервер СУБД в течении этих N секунд будет использовать для вычисления TSN более позднее время, а когда время сравняется, начнет использовать текущее.

В случае если в реализации СУБД содержится ошибка в отношении генерации очередного значения TSN, это может проявиться самым различным образом. В первую очередь либо некорректным откатом транзакции, в течении которой был произведен перевод времени, либо некорректным процессом старта сервера. Если администратор системы сможет определить ситуацию сбоя, то может выполнить перевод времени на компьютере соответственно либо между сеансами работы СУБД, либо во время сеанса. В любом случае, механизм генерации TSN является строго внутренним для СУБД, и при обнаружении проблем в работе сервера необходимо обращаться в техподдержку СУБД.

Глава 5

Обработка ошибок

5.1 Состояние ошибки

MUMPS системы относятся к средам исполнения со встроенными языковыми средствами обработки и генерации ошибок. Уже на уровне языка присутствуют средства реагирования на нарушение обычного хода выполнения программы либо на невозможность его продолжения по каким-либо причинам.

Если сравнить с контекстом выполнения программы, не имеющей языковых средств обработки ошибок, то это может быть, к примеру язык C, в котором обработка отдельных типов ошибок выполняется поразному в зависимости от типа ошибки, операционной среды, и, возможно, процессора. Например, обработка прерывания Ctrl+C и переполнения числа с плавающей точкой выполняются совершенно разными способами, как и описание желания программы реагировать на них.

В MUMPS системах к средствам обработки ошибок относится само понятие состояния процесса при генерации ошибки, специальные команды и системные переменные и функции, используя которые разработчик определяет поведение программы при возникновении ошибки. В MUMPS системах обработка ошибок, последовательность ее выполнения, набор средств обработки ошибок не зависят от типа ошибки.

Штатное выполнение программы проходит в состоянии отсутствия ошибки и последовательность выполнения определяется набором указанных для выполнения команд и функций. При возникновении же ошибки система выполнения должна прервать штатное выполнение программы и, не продолжая его, выполнить обработку ошибок. Обычное течение программы далее невозможно. Например, при выполнении деления на ноль или при попытке записи в базу данных находящейся в состоянии

только на чтение.

Состоянием процесса в состоянии ошибки называется перевод процесса в состояние невозможности продолжить выполнение следующей операции - команды, функции, вычисления оператора. Система исполнения при выполнении кода проверяет условия выполнимости операций, и при невозможности дальнейшей нормальной работы переводит процесс в состояние ошибки. Все непреодолимые для выполнения системой события сводятся в одно состояние - состояние ошибки. К ним относятся множество самых различных типов событий, обнаруживаемые системой исполнения, либо генерируемые программно командой генерации ошибки. Это состояние существует у процесса до того момента, как будет отменено программно определенным программистом образом. И, пока процесс находится в состоянии ошибки, процесс должен предпринять специальные меры. Для того, чтобы программа не остановилась в нерешительности, разработчик указывает, что именно необходимо предпринять среде исполнения MUMPS системы. К таким указаниям относится комплекс управления системой исполнения:

1. На какой из предыдущих уровней стека необходимо перейти, или необходимо ли продолжить обработку на этом же уровне стека.

2. Какой код необходимо выполнить, или на какую строку передать управление.

3. Критерий окончания обработки ошибок и перевод состояния среды исполнения MUMPS системы в состояние отсутствия ошибок.

Характер обработки ошибок указывается не всей MUMPS системе целиком с распространением требований на все выполняемые процессы, а индивидуально каждому процессу. У каждого процесса свои собственные значения системных переменных, описывающих характер, или способ обработки ошибок.

Конечно, MUMPS система в действительности никогда не останавливается в нерешительности и в любом случае предпринимает действия по продолжению выполнения. И, в случае, если разработчик не указал что делать процессу при возникновении ошибки, и система исполнения не может нормально продолжить выполнение программы, то начинает искать инструкции по обработке ошибок на текущем стеке. Не найдя их, выполняет возврат на предыдущий уровень стека и снова ищет.

В случае, если система вернулась на самый верхний уровень стека в состоянии ошибки, то выполняется поведение по умолчанию - производится переключение на устройство по умолчанию, вывод в текущее

устройство диагностического сообщения и процесс переходит к ожиданию ввода следующих команд. При этом система переводит процесс в состояние отсутствия ошибки, но сохраняет информацию об ошибке в специальных системных переменных. Формат диагностического сообщения для каждой из MUMPS системы свой и содержит порцию описания, которую система может предоставить. Обычно по такому диагностическому сообщению понятно, что произошло, и зачастую где именно или с какими переменными.

Исторически сложилось так, что, хотя стандарт предусматривал состояние ошибки, стандартизированные средства и определение их поведения стандарт языка MUMPS начал предусматривать лишь с редакции 1995-го года.

MUMPS системы, разработанные до этого года, вводили обработку ошибок на свое усмотрение, и в существенной степени совместимо друг с другом по принципам поведения. При этом отдельные нюансы поведения различных MUMPS систем отличаются. К таким отличиям относится множество сочетаний собственно обработчика ошибок с другими компонентами MUMPS системы:

1. Соотношение обработчика ошибок и состояния транзакций.
2. Сочетания поведения достандартного обработчика и стандартного, их приоритеты.
3. Возможность обработки ошибок возникших в самом обработчике ошибок.
4. Поведение встроенных команд отладки, например BREAK, ZGOTO, ZQUIT и др.
5. Наличие и содержание дополнительных системных переменных, например $ZSTATUS, $ZERROR и др.

В случае если разработчики прикладной системы используют лишь определенный стандартом обработчик ошибок, то такие программы переносимы. В случае если используется достандартный, то при переносе программ разработчикам необходимо сверить поведение используемых ими нюансов таких обработчиков.

В настоящее время разработчики прикладных программ обычно выбирают то, что им удобнее, или то к чему они привыкли, и, зачастую, таким средством оказывается именно достандартный характер обработки ошибок.

Поведение достандартных обработчиков ошибок в некоторых MUMPS системах разрабатывалось зачастую с ориентацией на практичность их применения в прикладных программах, наиболее подходящих для применения этих MUMPS систем. Конечно, поведение таких обработчиков ошибок не было привязано к определенным прикладным программным пакетам, и было достаточно универсально, но при реализации нюансов производители MUMPS систем могли учесть наиболее предпочтительные умолчания, подходящие определенным программам.

Для тех программистов, кто ранее работал со строчно - ориентированными языками, принцип построения обработчиков ошибок в MUMPS будет знаком. В частности, в языках BASIC группы до сих пор используется синтаксическая конструкция указания что делать системе исполнения при возникновении состояния ошибки:

```
On Error GoTo ОбработкаОшибок
```

для указания куда перейти для обработки ошибок и

```
On Error Goto 0
```

для отключения установленного ранее обработчика ошибок и перевода процесса в состояние обработки ошибок по умолчанию.

После взведения такого указания процесс уже имеет инструкцию, описывающую, что необходимо предпринять в случае возникновения ошибки. При этом инструкция обработки ошибок распространяется в зависимости от особенностей языка либо на все дальнейшее выполнение, либо на все выполнение на текущем уровне стека. При необходимости ограничить действие обработчика ошибок нужно его взвести в соответствующее состояние.

Более поздние языки с поддержкой обработки ошибок на уровне языка также содержат средства ограничения области действия обработчика ошибок путем синтаксического указания этой области, например, конструкции вида:

```
try - catch
try - except
try - finally
```

Что интересно, в старших версиях Caché при введении блочного синтаксиса с ограничением действия фигурными скобками также был введен и обработчик ошибок с ограничением области его действия:

```
TRY {
     protected statements
} CATCH [ErrorHandle] {
     error statements
}
further statements
```

В частности, обработчик деления на ноль в старших версиях Caché синтаксически может быть описан с использованием нового расширенного синтаксиса в стиле блоков try - catch и команды throw так:

```
div(num,div) public {
 TRY {
  SET ans=num/div
 } CATCH errobj {
  IF errobj.Name="<DIVIDE>" { SET ans=0 }
  ELSE { THROW }
 }
 QUIT ans
}
```

В языке MUMPS, определенном как строчно - ориентированном языке, вообще говоря, отсутствует такое синтаксическое понятие, как строки принадлежащие или не принадлежащие функции или процедуре. Поэтому в стандартном языке отсутствует как область действия обработчика ошибок, так и автоматическое прекращение его действия какими либо синтаксическими элементами. Область действия определяется лишь состоянием стека. При этом программа может выполнять на одном и том же уровне стека самые различные строки, в том числе принадлежащие и различным рутинам.

Вообще говоря, обработка ошибок включает в себя в качестве составной части определенный алгоритм передачи управления, и этот механизм может быть использован программой в качестве штатной составляющей работы процесса. Процесс можно перевести в состояние ошибки программно, если процесс выполнит команду генерации ошибки. В этом случае разработчики должны понимать, что они используют механизм обработки непреодолимого состояния для возврата по стеку, который можно выполнить обычными командами. Сложно сказать, может ли это быть рекомендовано или не рекомендовано. В некоторых случаях это действительно может существенно как упростить код, так и усложнить его понимание и модернизацию.

5.2 ZTRAP

Механизм обработки ошибок, применявшийся в MUMPS системах до введения единого стандартизированного определения, называется обобщенным термином ZTRAP, поскольку производители различных MUMPS систем использовали именно этот термин.

В различных реализациях его определение и поведение различно, хотя и сохраняет общие принципы обработки ошибок - он указывает что необходимо выполнить в случае возникновения состояния ошибки и документация на конкретную MUMPS систему описывает каким именно образом он будет выполняться.

Для указания обработчика ошибок используется специальная системная переменная $ZTRAP, доступная на чтение и запись. В случае если значение пусто, то это означает что система должна выполнять обработку ошибок по умолчанию. Иначе система должна использовать это значение в качестве инструкции что необходимо предпринять.

Очень важно отметить, что различные реализации MUMPS систем используют различные механизмы при обработке ошибок по ZTRAP, поэтому далее будет описываться механизм с указанием MUMPS системы, для которой он приведен. Кроме того, что поведение обработчиков ZTRAP в различных MUMPS системах отличается, также отличается их поведение при комбинировании с обработчиком ETRAP и их одновременном использовании в одном процессе.

5.2.1 Caché

В системе Caché системной переменной $ZTRAP необходимо присвоить имя метки. В случае возникновения ошибки процесс выполняет неявную команду goto и переходит на эту метку. Пример:

```
proc ; k  d proc^ZTRAP w
 s $ztrap="err"
 w "in proc, $st=",$st,!
 w 1/0
 w "exit proc, $st=",$st,!
 q
err
 w "in err, $st=",$st,!
 q
```

При выполнении в терминале Caché получаем:

```
USER>k  d proc^ZTRAP w
in proc, $st=1
in err, $st=1
```

Здесь иллюстрируется, что обработчик ошибок выполнился, не дойдя до отметки о выходе из подпрограммы, и данный обработчик выполнился на том же уровне стека.

В Caché есть дополнительная особенность обработчика ZTRAP: перед именем метки может быть указан символ "*". Отсутствие этого символа означает, что система выполнения должна вернуться до того уровня стека, на котором произошло присваивание $ZTRAP и вместо выполняемого кода выполнить неявную команду GOTO. В случае если происходит ошибка на более глубоком уровне стека, то процесс возвращается по стеку и уже на именно том уровне, где было присваивание $ZTRAP, выполняет обработчик ошибки. Для иллюстрации модифицируем пример:

```
proc ; k  d proc^ZTRAP w
 s $ztrap="err"
 w "in proc, $st=",$st,!
 d trap
 w "exit proc, $st=",$st,!
 q
trap
 w "in trap, $st=",$st,!
 w 1/0
err
 w "in err, $st=",$st,!
 q
```

При выполнении получаем:

```
USER>k  d proc^ZTRAP w
in proc, $st=1
in trap, $st=2
in err, $st=1
```

Здесь обработчик выполнился именно на том же уровне стека, где был установлен, независимо от уровня стека, где произошла ошибка - на том же или более глубоком.

В случае если перед именем метки ставится символ "*", то это служит процессу Caché инструкцией не выполнять развертывание стека до того уровня где был установлен обработчик, а выполнить его на том уровне стека где произошла ошибка. Проиллюстрируем еще раз модифицированным примером, указав уже символ "*" перед именем метки:

```
proc ; k  d proc^ZTRAP w
 s $ztrap="*err"
 w "in proc, $st=",$st,!
 d trap
```

```
 w "exit proc, $st=",$st,!
 q
trap
 w "in trap, $st=",$st,!
 w 1/0
err
 w "in err, $st=",$st,!
 q
```

При выполнении примера получаем:

```
USER>k  d proc^ZTRAP w
in proc, $st=1
in trap, $st=2
in err, $st=2
exit proc, $st=1
```

Для того, чтобы продолжить выполнение программы с команды, следующей после вызвавшую ошибку, при использовании обработчика ZTRAP в Caché нужно:

1. Внести код, который может вызвать ошибку, в аргумент команды XECUTE.
2. Перед именем метки с обработчиком указать символ "*".

Проиллюстрируем это на примере:

```
proc ; k  d proc^ZTRAP w
 s $ztrap="*err"
 w "in proc, $st=",$st,!
 x "w 1/0"
 w "exit proc, $st=",$st,!
 q
err
 w "in err, $st=",$st,!
 q
```

При выполнении такого кода получаем:

```
USER>k  d proc^ZTRAP w
in proc, $st=1
in err, $st=2
exit proc, $st=1
```

В определение обработчика ошибок входит 3 пункта: 1) что делать со стеком, 2) какой код нужно выполнить и 3) критерий окончания обработки. В случае использования обработчика ZTRAP в Caché третий пункт выполняется автоматически, как только система выполнит переход на обработчик ошибок по неявной команде GOTO.

К отличительной особенности перехода по GOTO относится возможность перейти на метку из различных точек выполняемого кода (строк), на которые мы можем попасть в различном контексте вызова - с ожиданием возвращаемого значения или без. И, для того, чтобы обработчик корректно отнесся к возврату при выполнении неявной команды GOTO, нужно указать возврат в зависимости от значения системной переменной $QUIT. Покажем, как может сработать обработчик, не учитывающий контекст возврата:

```
proc() ; k  w $$proc^ZTRAP() w
 s $ztrap="err"
 w "in proc, $st=",$st,!
 w 1/0
 w "exit proc, $st=",$st,!
 q 11
err
 w "in err, $st=",$st,!
 q
```

Это тот же самый код, но предназначенный для вызова с возвратом значения:

```
USER>k  w $$proc^ZTRAP() w
in proc, $st=1
in err, $st=1

K  W $$proc^ZTRAP() W
   ^
<COMMAND>
USER>
```

Здесь команда QUIT в обработчике ошибок вошла в конфликт по своей форме (безаргументной) с контекстом вызова с ожиданием возврата. Чтобы исправить ситуацию и сделать обработчик ошибки более универсальным, модифицируем его, чтобы учитывалось значение $QUIT:

```
proc() ; k  w $$proc^ZTRAP() w
 s $ztrap="err"
 w "in proc, $st=",$st,!
 w 1/0
```

```
 w "exit proc, $st=",$st,!
 q 11
err
 w "in err, $st=",$st,!
 q:$Q "was error" q
```

И этот вариант уже корректно срабатывает в обоих случаях вызова:

```
USER>k  w $$proc^ZTRAP() w
in proc, $st=1
in err, $st=1
was error
USER>d proc^ZTRAP
in proc, $st=1
in err, $st=1
```

В случае если в обработчике ошибки также происходит ошибка, то при его срабатывании также вызывается обработчик ошибок. Если программист не изменил значение $ZTRAP, то при его выполнении программа зацикливается. Приведем модифицированный пример для иллюстрации:

```
proc() ; k  w $$proc^ZTRAP() w
 s count=0
 s $ztrap="err"
 w "in proc, $st=",$st,!
 w 1/0
 w "exit proc, $st=",$st,!
 q 11
err
 i count=1 w "already in err",! q:$Q "was error" q
 s count=1
 w "in err, $st=",$st,!
 w 1/0
 q:$Q "was error" q
```

При его выполнении получаем отчет о ходе обработки ошибки:

```
USER>k  w $$proc^ZTRAP() w
in proc, $st=1
in err, $st=1
already in err
was error
count=1
USER>
```

Другим способом избежать зацикливание в обработчике ошибок является самостоятельная очистка текущего обработчика ошибок $ZTRAP в самом обработчике:

```
proc() ; k  w $$proc^ZTRAP() w
 s $ztrap="err"
 w "in proc, $st=",$st,!
 w 1/0
 w "exit proc, $st=",$st,!
 q 11
err
 s $zt=""
 w "in err, $st=",$st,!
 w 1/0
 q:$Q "was error" q
```

В этом случае процесс, попав в обработчик, в нем использует обработчик ошибок, установленный на более высоком уровне, или, если он не был установлен, то переходит к обработке ошибки по умолчанию:

```
USER>k  w $$proc^ZTRAP() w
in proc, $st=1
in err, $st=1

 w 1/0
 ^
<DIVIDE>err+3^ZTRAP
USER 2e0>
```

При использовании системы Caché и обработчика в стиле ZTRAP программисты чаще всего выбирают типовой способ его использования с такими условиями:

1. Обработчик используется без указания символа "*", чтобы вне зависимости от глубины стека где произошла ошибка, управление было передано на код, указанный программистом при установке обработчика.

2. Обработчик устанавливается во всех функциях, где это важно, и с самого верхнего уровня. Во вспомогательных функциях-утилитах из разряда преобразований данных обработчик обычно не указывается.

3. В самом коде обработчика ошибок значение обработчика $ZTRAP предварительно очищается во избежание зацикливания и чтобы код мог перейти на обработку ошибки на более верхнем уровне стека.

При этом нужно понимать, что выбор характера обработки ошибок в каждом проекте может отличаться из-за различных методик или необходимости предпринимать какие-то особенные для прикладной программы действия. Для этого изучаются требования к проекту, особенности работы обработчика ошибок в определенной MUMPS системе или даже ее версии, и составляются правила его применения, чтобы группа программистов - участников проекта им далее следовала.

Старшие версии Caché также дополнительно проверяют синтаксис значения $ZTRAP при присваивании и в случае недопустимого синтаксиса генерируют ошибку <SYNTAX>:

```
USER>s $zt="w w"

S $ZT="w w"
^
<SYNTAX>
```

5.2.2 MSM

В системе MSM поведение обработчика ZTRAP почти в точности аналогично его поведению в системе Caché, но с такими отличиями:

1. Не поддерживается символ "*" перед меткой обработчика ошибок и MSM всегда передает управление на тот уровень стека, где он был установлен последним.

2. Перед выполнением неявной команды GOTO для передачи управления на код обработчика ошибок значение обработчика $ZTRAP автоматически сбрасывается в значение пустая строка.

И, также как и в Caché, в системе MSM критерием выхода процесса из состояния ошибки является передача управления неявной командой GOTO на метку обработчика.

Таким образом, для организации полностью совместимой между Caché и MSM обработки ошибок с помощью ZTRAP нужно, чтобы програмисты не использовали символ "*" перед именем метки в значении $ZTRAP, и в коде обработчика сбрасывали значение $ZTRAP в значение пустая строка. И большинство программ, разрабатываемых как для MSM, так и для Caché, действительно придерживается таких правил.

5.2.3 DTM

В системе DTM (Data Tree MUMPS) обработчик ZTRAP также аналогичен по поведению системе MSM, но у программиста дополнительно есть команда ZQUIT. Если код обработчика ошибок заканчивается командой QUIT, то управление передается на тот уровень стека, на котором произошла ошибка, а если командой ZQUIT, то управление передается на обработчик ошибок, установленный раньше по стеку.

5.2.4 M3

В MUMPS системе M3 (в настоящее время доступна реализация M3-Lite) обработчик ZTRAP используется в зависимости от синтаксиса содержания системной переменной $ZTRAP.

Если значение $ZTRAP синтаксически соответствует имени метки, то при обработке ошибки процесс выполняет возврат по стеку на уровень где было выполнено присваивание $ZTRAP и выполняет неявную команду GOTO на указанную метку.

В случае если значение $ZTRAP синтаксически соответствует последовательности команд, то они выполняются вместо строки, на которой произошла ошибка. В этом случае код команд должен выполнить передачу управления на иную строку командой GOTO или возврат командой QUIT.

В случае если присваивается пустая строка, то обработчик на этом уровне стека отменяется и далее процесс будет использовать обработчик, указанный ранее по стеку.

5.2.5 GT.M

В системе GT.M обработчик ZTRAP устроен, видимо, наиболее сложным образом из современных MUMPS систем. Ключевая часть его функционала, видимо, и послужила прототипом для определения стандартного обработчика ошибок ETRAP.

К формальному определению $ZTRAP в системе GT.M можно отнести то, что эта системная переменная должна содержать последовательность команд, которые будут выполнены на манер команды XECUTE. К дополнительным синтаксическим возможностям относится то, что она может быть аргументом команды NEW и система GT.M различает, на каком уровне стека был определен обработчик ошибок - если была применена команда NEW явно, то на этом же уровне стека, иначе присваивание системной переменной $ZTRAP изменяет значение на том уровне

стека где она была заведена ранее.

По умолчанию команда NEW, примененная к переменной $ZTRAP, автоматически сбрасывает ее в значение пустая строка, поэтому обычно следующей же командой идет присваивание $ZTRAP обработчика ошибок. В системе GT.M, в отличие от других, пустое значение не является признаком обработчика по умолчанию. Если процесс попал в состояние ошибки при пустом $ZTRAP, то он завершает работу. В начальном состоянии процесса после его старта значение $ZTRAP равно "B", что означает необходимость выполнить при ошибке команду BREAK, то есть перейти в непосредственный режим управления процессом (Direct Mode), и управление передается оператору для непосредственного ввода команд. При этом в системе GT.M имеется управление поведением присваивания $ZTRAP через переменную окружения - если значение переменной окружения gtm_ztrap_new вычисляется как TRUE, YES или ненулевое число, то любое присваивание $ZTRAP приводит к неявному выполнению команды NEW для $ZTRAP перед присваиванием. Применение NEW к переменной $ZTRAP позволяет сохранить предыдущие определения обработчиков ошибок и автоматически восстановить предыдущее значение при возврате по стеку. В отличие от определения ZTRAP в Caché, в GT.M обработчик ошибок всегда выполняется на том же уровне стека где произошла ошибка, без автоматического возврата по стеку к месту установки $ZTRAP.

Формально говоря, при возникновении ошибки в исполняемой строке, процесс GT.M переходит в состояние выполнения строки $ZTRAP как если бы она была вместо исполняемой строки, или XECUTE без организации нового уровня стека. Критерий окончания состояния ошибки - начало выполнения этой строки. При этом система GT.M различает, в каком качестве и как использовать содержание переменной $ZTRAP.

В зависимости от значения переменной окружения gtm_ztrap_form, процесс GT.M использует значение $ZTRAP следующими способами (по умолчанию поведение GT.M соответствует значению code):

code	Значение $ZTRAP используется как последовательность команд, выполняемых при ошибке. Если управление не было передано командами GOTO, ZGOTO или QUIT какой-либо другой строке кода, то после выполнения кода $ZTRAP управление снова передается на начало строки, вызвавшей ошибку.

entryref	Значение $ZTRAP используется как имя метки для передачи управления неявной командой GOTO.
adaptive	Если значение $ZTRAP синтаксически является последовательностью команд, то используется как было описано для "code", иначе используется как было описано для "entryref". В этом случае процесс GT.M применяет распознавание синтаксиса и действует адаптивно.
popentryref	Процесс GT.M возвращается по стеку на уровень, где было произведено последнее присваивание $ZTRAP и использует его значение как имя метки для выполнения неявной команды GOTO для передачи управления.
popadaptive	Процесс GT.M возвращается по стеку на уровень, где было произведено последнее присваивание $ZTRAP и использует его значение в зависимости от синтаксиса - если это последовательность команд то они выполняются, если имя метки то на нее передается управления неявной командой GOTO.

Кроме того, процесс GT.M использует еще одну переменную окружения процесса - переменную gtm_zyerror. Если ее значение синтаксически соответствует имени метки, то при значениях переменной gtm_ztrap_form, равным popentryref или popadaptive, процесс после возврата по стеку выполняет неявную команду DO для метки, указанной в переменной окружения gtm_zyerror, и лишь после возврата из нее передает управление либо последовательности команд, либо метке, указанным в $ZTRAP.

В качестве иллюстрации поведения $ZTRAP приведем пример из документации GT.M при присваивании переменной $ZTRAP последовательности команд:

```
GTM>ZPRINT ^EP6
EP6     WRITE !,"THIS IS "_$TEXT(+0)
        NEW
        NEW $ZTRAP SET $ZTRAP="DO ET"
        SET (CB,CE)=0
BAD     SET CB=CB+1 WRITE A SET CE=CE+1
        WRITE !,"AFTER SUCCESSFUL EXECUTION OF BAD:",!
        SET A="A IS NOT DEFINED"
        ZWRITE
        QUIT
```

```
ET      W !,"CONTINUING WITH ERROR TRAP AFTER AN ERROR",!
        ZWRITE
        SET A="A IS NOW DEFINED"


GTM>do ^EP6

THIS IS EP6
CONTINUING WITH ERROR TRAP AFTER AN ERROR
CB=1
CE=0
A IS NOW DEFINED
AFTER SUCCESSFUL EXECUTION OF BAD:
A="A IS NOT DEFINED"
CB=2
CE=1

GTM>
```

Здесь при выполнении строки BAD генерируется ошибка неопределенного значения A, вызывается обработчик ошибок в виде последовательности команд

```
DO ET
```

в котором корректируется ошибка с определенностью переменной A.

Затем снова управление снова передается на строку BAD и в примере показывается, что оператор увеличения CB выполнился дважды, а увеличения CE - единожды.

Тот же алгоритм обработки ошибки с использованием стандартного обработчика ETRAP, с возвратом управления на строку, вызвавшую ошибку, выполняется таким способом:

```
GTM>ZPRINT ^EP6A
EP6A    WRITE !,"THIS IS "_$TEXT(+0)
        NEW
        NEW $ETRAP SET $ETRAP="GOTO ET"
        SET (CB,CE)=0
BAD     SET CB=CB+1 WRITE A SET CE=CE+1
        WRITE !,"AFTER SUCCESSFUL EXECUTION OF BAD:",!
        ZWRITE
        QUIT
ET      W !,"CONTINUING WITH ERROR TRAP AFTER AN ERROR",!
        ZWRITE
        SET A="A IS NOW DEFINED"
        SET RETRY=$STACK($STACK,"PLACE")
        SET $ECODE=""
```

```
        GOTO @RETRY


GTM>DO ^EP6A

THIS IS EP6A
CONTINUING WITH ERROR TRAP AFTER AN ERROR
CB=1
CE=0
A IS NOW DEFINED
AFTER SUCCESSFUL EXECUTION OF BAD:
A="A IS NOW DEFINED"
CB=2
CE=1
RETRY="BAD^EP6A"

GTM>
```

Здесь для выполнения возврата на строку имя строки с ошибкой вычисляется обращением к системной переменной $STACK и системной функции $STACK. И в обработчике ошибок $ETRAP, и в конце кода обработчика используются команды GOTO, сохраняющие уровни стека.

5.2.6 MiniM

В системе MiniM обработчик ZTRAP также выполнен с отличительными особенностями поведения, не совпадающими в точности с другими MUMPS системами. Вообще говоря, в системе MiniM отсутствует отдельный механизм обработки ошибок ZTRAP как таковой, и присутствует только обработчик ETRAP. Системная переменная $ZTRAP в MiniM используется исключительно для определенного уровня синтаксической совместимости с имеющимся кодом и при присваивании $ZTRAP в действительности выполняется присваивание системной переменной $ETRAP в зависимости от содержания $ZTRAP.

В системе MiniM значение $ZTRAP сохраняется единственным для всех уровней стека, вне зависимости от уровня на котором производилось присваивание.

При присваивании значения системной переменной $ZTRAP производится анализ его содержания и выполняются операции с присваиванием и команда new с системными переменными $ESTACK и $ETRAP в зависимости от содержания $ZTRAP. В значении системной переменной $ZTRAP допускается либо имя метки для перехода по команде goto, либо метка и дополнительный символ "*" перед ней.

Если значение $ZTRAP при присваивании содержит только метку, то при этом выполняется код эквивалентный коду:

```
new $estack
new $etrap
set $etrap="g:'$es "_$ztrap
```

Это соответствует раскручиванию стека выполнения до того уровня, на котором был установлен обработчик, и переходу на обработчик по команде goto.

Если значение $ZTRAP при присваивании содержит символ "*" перед меткой, то при этом выполняется код, эквивалентный коду:

```
set $etrap="g "_$e($ztrap,2,$l($ztrap))
```

Это соответствует переходу на обработчик ошибок на уровень стека, на котором возникла ошибка:

```
USER>s $zt="err^errhandler"

USER>w $et
g:'$es err^errhandler
USER>s $zt="*err^errhandler"

USER>w $et
g err^errhandler
```

В случае, если системной переменной $ZTRAP присваивается значение не соответствующее ни одной допустимой форме, то процесс MiniM генерирует ошибку <SYNTAX>.

Поскольку в MiniM обработчик ZTRAP есть только переходник на обработчик ETRAP, для выполнения кода совместимого по поведению с другими MUMPS системами необходимо в коде обработчика очистить значение системной переменной $ECODE. Эту очистку надо разместить после того, как значение $ECODE было использовано, например, так:

```
n ec s ec=$ec s $ec=""
```

В этом случае обработка ошибки будет закончена выполнением кода обработчика. Иначе, без очистки значения $ECODE при возврате из обработчика, будет продолжена раскрутка стека до передачи управления обработчику, установленному ранее.

5.3 ETRAP

Обработчик ошибок ETRAP был определен в стандарте языка MUMPS ANSI 1995-го года и в настоящее время входит в ныне действующий стандарт ISO MUMPS. В существенной степени его функционал был выбран похожим на наиболее сильные стороны обработчика ZTRAP, реализованного в системе GT.M.

5.3.1 Определение

Основу обработчика ETRAP составляет комплекс следующих соглашений:

1. Схема обработки ошибок стандартизирована и переносима.
2. Обработчик задается не меткой, а строкой команд, которая выполняется при переходе процесса в состояние ошибки.
3. Имеется возможность изменять установленный ранее обработчик $ETRAP.
4. Имеется возможность указать явно область видимости действия обработчика $ETRAP командой NEW.
5. Имеется возможность вести независимый отсчет по стеку системной переменной $ESTACK, применяя по необходимости к ней команду NEW.
6. Имеется системная переменная $STACK и парная ей системная функция $STACK() для анализа стека.
7. Имеется системная переменная $ECODE для чтения текущего состояния ошибки, для генерации ошибки при присваивании непустой строки и для окончания обработки ошибки при присваивании пустой строке.

В целом, весь этот комплекс соглашений реализует более мощную и гибкую схему обработки ошибок, чем простые правила обработчиков ZTRAP, связанные с неявной передачей управления командой GOTO на метку.

Алгоритм выполнения обработчика ошибок по схеме ETRAP, вообще говоря, стандартен, но, поскольку в каждой из MUMPS систем присутствует в той или иной мере схема обработки ZTRAP, в каждой из них

могут быть отдельные нюансы по сочетанию двух этих обработчиков, по приоритету их действия в состоянии ошибки, и по переключению процесса в зависимости от последнего присваивания обработчика - ETRAP или ZTRAP. Далее будет рассматриваться схема чистого обработчика ETRAP, в состоянии, когда процесс не определял обработку по ZTRAP. В случае, если прикладная программа сочетает оба варианта, программистам необходимо сверить по документации на используемую MUMPS систему характер обработки ошибок для применяемой MUMPS системы, включая возможные особенности ее версии.

5.3.2 $ETRAP

Для описания действий, которые необходимо предпринять при обработке ошибки, используется присваивание специальной системной переменной $ETRAP последовательности команд.

При этом системная переменная $ETRAP может быть ограничена по области действия командой NEW. При присваивании выполняется изменение значения $ETRAP на том уровне стека, где к $ETRAP последний раз была применена команда NEW, а если не было, то изменяется ее значение на нулевом, начальном уровне стека. При чтении переменной $ETRAP возвращается значение присвоенное на последнем уровне стека, независимо от текущего уровня стека. В определенном смысле такое поведение системной переменной $ETRAP аналогично обычной локальной переменной, с тем исключением что эта переменная всегда существует и при старте процесса по умолчанию имеет значение пустая строка. Присваивание значения $ETRAP само по себе не выполняет автоматического ограничения области действия.

Дополнительно можно отметить, что зачастую после применения команды NEW к переменной производится ее присваивание, и для сокращения таких операций система MiniM поддерживает расширенную (нестандартную) инициализирующую форму команды NEW. В MiniM можно использовать оба варианта как равноценные:

```
new $etrap set $etrap=handler
new $etrap=handler
```

Интересно, что, в отличие от локальных переменных, команда NEW в применении к системной переменной $ETRAP не изменяет ее значение, а создает копию на текущем уровне стека, и дальнейшие команды SET изменяют значение $ETRAP уже начиная с текущего уровня. Пример рутины для выполнения NEW применительно к переменной $ETRAP:

```
setnew ; k  d setnew^ETRAP w
 s $et=123
 w "1: ",$et,!
 d subset
 w "4: ",$et,!
 q
subset
 n $et
 w "2: ",$et,!
 s $et=456
 w "3: ",$et,!
 q
```

При выполнении рутины получаем последовательность значений $ETRAP:

```
USER>k  d setnew^ETRAP w
1: 123
2: 123
3: 456
4: 123
```

Здесь видно, что в точке 2 значение $ETRAP осталось неизменным, и, несмотря на изменение в точке 3, при возврате в точку 4 предыдущее значение было восстановлено.

При переходе к обработке ошибки процесс выполняет значение переменной $ETRAP вместо текущей выполняемой строки. Процесс выполняет ее на манер команды XECUTE, но без организации нового уровня стека.

После окончания выполнения строки команд, указанной в $ETRAP, и, выполнив возврат по стеку, процесс проверяет значение системной переменной $ECODE, и, если оно пустое, то выполнение продолжается со следующей строки. Если не пустое, то процесс возвращается по стеку на предыдущий уровень и снова выполняет значение $ETRAP как последовательность команд. На предыдущем уровне стека значение $ETRAP может уже быть другим.

После окончания выполнения строки команд выполняется неявно последовательность команд возврата по стеку в следующей форме:

```
 quit:$quit ""
 quit
```

Это означает выполнить безаргументный возврат, если возврат по контексту не ожидается, и вернуть пустую строку, если возврат ожидается. При этом значение системной переменной $TEST такой логической проверкой не затрагивается и не изменяется.

Для задания обработчика ошибок необходимо, как минимум, выполнить два действия:

1. Присвоить переменной $ETRAP строку с последовательностью команд для обработки ошибки.
2. Эта последовательность команд должна явно или неявно изменить или не изменить значение переменной $ECODE.

Остальные системные переменные и системные функции служат коду обработчика ошибок индикатором состояния, по которому он принимает решение.

Простой вариант, показывающий возврат управления по стеку при обработке ошибок:

```
proc() ; k  d proc^ETRAP() w
 s $etrap="w $st,!"
 w 1/0
 q
```

При его выполнении получаем сначала срабатывание обработчика на последнем, первом уровне, далее на предыдущем, нулевом:

```
USER>k  d proc^ETRAP() w
1
0
```

При этом значение $ECODE не изменяется, и все время сохраняет свое значение:

```
USER>w $ec
,M9,
```

Рассмотрим вариант остановки обработки ошибки в коде обработчика:

```
proc() ; k  d proc^ETRAP() w
 s $etrap="w ""stack="",$st,! s $ec="""""
 w $$sub()+123,!
 d sub()
 w "last",!
 q
sub()
 w "sub",!
 w 1/0
 q
```

При выполнении такого кода ошибка возникает в функции sub. Если она вызывается в контексте возврата значения, то возвращается пустая строка, иначе выполняется просто возврат. Дальнейшее выполнение продолжается на предыдущем уровне стека:

```
USER>k  d proc^ETRAP() w
sub
stack=2
123
sub
stack=2
last
```

5.3.3 $ECODE

Присваивание системной переменной $ECODE пустой строки для системы исполнения MUMPS процесса означает, что обработка ошибки закончена и дальнейшую раскрутку стека не нужно выполнять.

Присваивание системной переменной $ECODE непустой строки, наоборот, приводит к генерации ошибки с записыванием этого значения в переменную $ECODE. Заменим код генерации ошибки на принудительное взведение процесса в состояние ошибки таким присваиванием:

```
proc() ; k  d proc^ETRAP() w
 s $etrap="w ""stack="",$st,!,""$ec="",$ec,! s $ec="""""
 w $$sub()+123,!
 d sub()
 w "last",!
 q
sub()
 w "sub",!
 s $ec="trap in sub"
 q
```

При выполнении такого кода получаем последовательность работы:

```
USER>k  d proc^ETRAP() w
sub
stack=2
$ec=trap in sub
123
sub
stack=2
$ec=trap in sub
last
```

Здесь происходит все то же самое, что и при ошибке деления на ноль, но мы явно задаем, какую ошибку надо генерировать. Формально говоря, мы можем присвоить переменной $ECODE произвольное значение. Но стандарт содержит явные рекомендации использовать соглашения по формированию такого значения:

1. Содержание $ECODE задается последовательностью кодов ошибок, разделенных запятой.

2. При происхождении очередной ошибки, если значение $ECODE не пусто, код ошибки надо дописывать в конец этого списка.

3. Коды ошибок следует начинать с символа M для ошибок, зарезервированных и определенных стандартом, с символа Z для расширенных ошибок реализации MUMPS системы и с символа U для ошибок прикладной программы.

Список стандартных ошибок, начинающихся с символа M, приведен в приложении. В рассмотренном примере код ошибки "M9" соответствует ошибке деления на ноль.

Кроме того, некоторые MUMPS системы поддерживают, для совместимости с обработчиком ZTRAP, также команду ZTRAP для явной генерации ошибки, и в этом случае в значение $ECODE должнен быть занесен код ошибки, начинающийся с символа Z, поскольку такая команда не описана в стандарте, генерируемая ей ошибка не входит в описанные, и поддерживается в качестве расширенной команды MUMPS системы:

```
USER>s $et=""

USER>ztrap "custom"

<Zcustom>

USER>w $ec
,ZZTRAP,
USER>w $ze
<Zcustom>
```

Здесь стандарт определяет поведение переменной $ECODE, а значение расширенных $Z переменных остается на усмотрение реализации и в разных MUMPS системах может как совпадать, так и незначительно отличаться.

Что важно отметить, в языке допускается присваивание переменной $ECODE, и содержание преимущественно трактуется как список разделенный запятыми, но присваивание системной переменной в левосторонней форме функцией $PIECE() в языке синтаксически не поддерживается.

```
USER>s $p($ec,",",2)=45

S $P($EC,",",2)=45
^
<SYNTAX>
```

при выполнении в Caché и

```
USER>s $p($ec,",",2)=45

<SYNTAX> :SET: *s $p($ec,",",2)=45
```

при выполнении в MiniM.

Способ создания нового уровня стека для обработчика $ETRAP не важен, это может быть вызов метки командой DO, вызов функции с ожиданием возврата, команда XECUTE, или блочная форма команды DO:

```
proc() ; k  d proc^ETRAP() w
 s $etrap="w ""stack="",$st,!,""$ec="",$ec,! s $ec="""""
 d
 . w 1/0
 x "w 1/0"
 w "last",!
 q
```

При выполнении такого кода управление продолжается на уровне стека на уровень выше, чем уровень произошедшей ошибки:

```
USER>k  d proc^ETRAP() w
stack=2
$ec=,M9,
stack=2
$ec=,M9,
last
```

Для того, чтобы продолжить выполнение программы на том же уровне стека, необходимо передать управление из обработчика командой GOTO, например:

```
proc() ; k  d proc^ETRAP() w
 s $etrap="w $st,! s $ec="""" goto continue"
 w 1/0
continue
 w "last",!
 w $st,!
 q
```

В случае покидания управления из последовательности обработки ошибок, заданной в обработчике $ETRAP командой GOTO, неявный возврат управления на уровень стека выше не выполняется, при выполнении такого кода исполнение программы продолжается с заданной метки:

```
USER>k  d proc^ETRAP() w
1
last
1
```

В частности, обработчик ошибок может скорректировать ситуацию с данными, приведшими к ошибке и продолжить выполнение программы, в том числе с места, вызвавшего ошибку:

```
proc() ; k  d proc^ETRAP() w
 s $etrap="goto err"
 k a
do
 w a,!
 w 1/0
 q
err
 w "err trap at $st=",$st,", $ec=",$ec,!
 i '$d(a) s $ec="" s a=123 goto do
 q:$q "" q
```

Выполнение такого кода показывает последовательность обработки обеих ошибок - сначала вызывается обработчик из-за неопределенности переменной a, значение a корректируется, затем выполнение продолжается с использования переменной a, затем генерируется ошибка деления на ноль, и обработчик в этом случае не останавливает обработку очисткой переменной $ECODE, управление возвращается на уровень выше, снова срабатывает обработчик $ETRAP, но на верхнем уровне, вне контекста текущей рутины, конечно нет возможности определить метку err.

```
USER>k  d proc^ETRAP() w
err trap at $st=1, $ec=,M6,
```

```
123
err trap at $st=1, $ec=,M9,

goto err
^
<NOLINE>
```

Последнее сообщение в этом примере может отличаться для разных MUMPS систем. Одни могут запретить переход по GOTO по причине отсутствия текущей рутины, другие могут запретить выполнение GOTO на нулевом уровне стека, не проводя проверок на существование рутины и метки.

Также, как и последовательность команд, выполняемая в качестве аргумента командой XECUTE, всегда заканчивается неявной командой QUIT, даже если это не было указано, также и последовательность команд, задаваемая обработчику ошибок $ETRAP, всегда заканчивается неявной последовательностью команд

```
quit:$quit ""  quit
```

Важно отметить, что в случае, если обработчику ошибок $ETRAP был указан код, явно передающий управление на другую строку командой GOTO, уровень стека сохраняется и ответственность за стек и управление возвратом по стеку берет на себя разработчик. Если управление было явно передано по стеку командой QUIT, то именно эта команда и будет использована для возрата по стеку и дальнейшие команды, указанные в $ETRAP, в том числе и неявный возврат, исполняться не будут, до них просто не дойдет управление.

5.3.4 $ESTACK

Парной к системной переменной $ETRAP является системная переменная $ESTACK. Эта переменная используется в качестве маркера стека для обработки ошибки в форме, где это важно. Типовое назначение переменной $ESTACK - маркировать уровень стека, чтобы выполнить раскрутку стека до определенного уровня, отмеченного этим маркером, и только после этого выполнить обработчик ошибок.

Переменная $ESTACK была введена в стандарт языка в 1995-м году, наряду с системной переменной $STACK и системной функцией $STACK(). Все три используются для программного анализа состояния стека.

Переменная $STACK при старте процесса всегда равна 0, при увеличении уровня стека при вызове подпрограмм, блочной командой DO

или командой XECUTE, всегда увеличивается на 1 и при возврате по стеку командой QUIT либо неявной командой QUIT всегда принимает предыдущее значение, или возвращается к значению на 1 меньше. В определенном смысле переменная $STACK показывает абсолютное значение уровня стека.

Переменная $ESTACK также при старте процесса равна 0, и при каждом увеличении уровня стека увеличивается на 1. Но при возврате по стеку принимает предыдущее значение. Отличие от переменной $STACK в том, что если к переменной $ESTACK применена команда NEW, то на этом уровне стека переменная $ESTACK обнуляется и ее рост продолжается снова на 1 на каждый уровень стека. Но при возврате по стеку значение $ESTACK восстанавливается. Таким образом, если на уровне N к переменной $ESTACK применить команду NEW, то на нем она станет равной 0, и при возврате по стеку станет равной $N - 1$.

Пример такого поведения показывает программа:

```
estack ; k  d estack^ETRAP w
 w "1: ",$es,!
 d subes
 w "7: ",$es,!
 q
subes
 w "2: ",$es,!
 n $es
 w "3: ",$es,!
 d subes2
 w "6: ",$es,!
 q
subes2
 w "4: ",$es,!
 n $es
 w "5: ",$es,!
 q
```

При выполнении получаем последовательность значений $ESTACK:

```
USER>k  d estack^ETRAP w
1: 1
2: 2
3: 0
4: 1
5: 0
6: 0
7: 1
```

Переменная $ESTACK по своему поведению предназначена для относительной маркировки стека, и, в частности, для обработки ошибок,

поскольку ведет свой отсчет не в абсолютном выражении, а в относительном, начиная с последнего применения команды NEW. Обработчик ошибок может быть составлен таким образом, чтобы при возврате по стеку игнорировать все уровни стека до определенного и выполнить код обработки ошибки только на нужном уровне. Более грубым вариантом будет использование в таком случае значения $STACK, если указывать его в коде команд обработчика ошибок:

```
1) n $et s $et="q:$st'="_$st_"  g err"
2) n $es,$et s $et="q:$es  g err"
или
3) n $es,$et s $et="g:'$es err"
```

Второй вариант выглядит более эстетично и легче читается и понимается программистами.

Переменная $ESTACK может быть маркирована командой NEW столько раз, сколько это необходимо, при каждой такой операции отсчет снова начинается с нулевого значения.

5.3.5 Ошибки в обработчике ошибок

При выполнении кода обработчика ошибок также может возникнуть ошибка и для этого случая также должны быть предусмотрены правила.

Сам обработчик ошибок при выполнении также может назначить обработку своих ошибок, при выполнении назначенного им обработчика ошибок также могут возникнуть ошибки, и так далее.

К основной проблеме обработки ошибки, возникшей в обработчике ошибок, в MUMPS системах можно отнести определение момента, когда продолжение назначенных обработчиков ошибок далее не имеет смысла. К такой ситуации относится ситуация, когда при выполнении обработчика ошибок произошла ошибка, произошел вызов обработки ошибок и он закончился неуспехом. Критерий успеха в обработке ошибок - присваивание пустой строки системной переменной $ECODE.

Продемонстрируем пример программой, генерирующей такие вложенные ошибки:

```
proc7() ; s $ec=""  d proc7^ETRAP()
 n $et s $et="d err1 w 11,!"
 w "trap1",!,1/0
 q
err1
 n $et s $et="d err2 w 22,!"
 w "trap2",!,1/0
```

```
 q
err2
 n $et s $et="d err3 w 33,!"
 w "trap3",!,1/0
 q
err3
 w "stack = ",$st,!
 q
```

При выполнении этой программы трижды назначаются обработчики ошибок и трижды для них создается ситуация ошибки. При выполнении такой программы на экран выводятся сообщения о порядке выполнения:

```
USER>s $ec=""  d proc7^ETRAP() w 77,!
trap1
trap2
trap3
stack = 4
33

USER>
```

Здесь последовательно срабатывают обработчики ошибок и в итоге управление спускается командой

```
d err3
```

на 4-й уровень стека.

После выполнения последнего обработчика ошибок, или в данном случае кода

```
d err3 w 33,!
```

управление покидает уровень стека, на котором выполнялся обработчик ошибок. Управление в данном случае покидается неявными командами

```
  q:$q ""  q
```

выполняемыми, когда управление передается после последней команды на окончание строки команд.

И в этой ситуации неважно, как именно управление покидает этот уровень стека - будет выполнен неявный возврат, явная команда QUIT или управление покинет строку команд из $ETRAP командой GOTO, а далее выполнит команду QUIT.

Все эти случаи проверяют, что управление покидает уровень обработчика ошибок без назначения переменной $ECODE пустой строки, и

ранее по стеку уже выполнился предыдущий обработчик (или несколько обработчиков) ошибок. Если эта ситуация обнаружена, то процесс выполняет возврат по стеку столько раз, сколько необходимо, чтобы оказаться на том уровне стека, где обработчик ошибок не срабатывал. Важно понимать, что самый первый уровень стека, на котором сработал обработчик ошибок, также будет пропущен и управление будет передано на имеющийся на том уровне стека обработчик ошибок или, если он не назначен, будет продолжена раскрутка стека.

В этой ситуации также могут найтись расхождения в разных реализациях. В частности, в MUMPS системе Caché раскрутка стека будет продолжаться с вызовом обработчиков $ETRAP, если они назначены на каждом уровне стека, пока системная переменная $ECODE не примет значение пустая строка. MUMPS система GT.M при раскрутке стека остановит раскрутку на том уровне, на котором значение $ETRAP окажется равно пустой строке, то есть присутствует критерий останова, который не опирается на выполнение кода указанного программистом, а использует отсутствие указаний по обработке ошибок. В этом случае сохраняется риск, что сработавший в обработчике ошибок вложенный обработчик ошибок может привести к состоянию зацикливания внутреннего обработчика из-за неподтвержденного программистом состояния окончания обработки ошибки.

Интересной в данном случае ситуацией является прохождение управления как-бы между командами. В случае, если в обработчике ошибок указаны команды

```
command1 arg1 command2 arg2
```

то при происхождении ошибки при выполнении первой из них, срабатывании обработчика ошибок, в котором также назначается обработчик ошибок и который также закончится непустым значением $ECODE, то вторая команда не будет выполнена, поскольку процесс автоматически раскрутит стек мимо нее на предыдущий уровень, хотя при обычном выполнении программ команды выполняются одна за другой подряд, слева направо.

Другим состоянием ошибки процесса, приводящем к раскрутке стека свыше уровня, вызвавшего ошибку, является ошибка в командах самого обработчика ошибок. Посмотрим такое выполнение на программе, где в командах самого обработчика ошибок генерируется ошибка:

```
proc8() ; s $ec="",$et="" d proc8^ETRAP()
 s $et="w ""$st="",$st,!,1/0"
 d
```

```
 . w 1/0
 q
```

При выполнении получаем:

```
USER>s $ec="",$et="" d proc8^ETRAP()
$st=2
$st=1
$st=0

w "$st=",$st,!,1/0
               ^
<DIVIDE>^ETRAP
USER>
```

Формально, при происхождении ошибки обработчик ошибок должен выполниться вместо этой строки. Но, если сама строка содержит ошибку выполнения, то такая ситуация рекурсивно зацикливает процесс. Для выхода из этой проблемы процесс проверяет, что состояние ошибки возникло непосредственно при выполнении команд обработчика ошибок, не дойдя до его окончания.

В этой ситуации процесс также переходит к состоянию раскрутки стека - сначала пропускает все уровни, включая вызвавший ошибку, и далее вызывает обработчики ошибок для каждого из уровней стека, пока по окончании какого-либо из них переменная $ECODE не примет значение пустая строка, либо пока процесс не перейдет к самому верхнему уровню стека.

5.4 $STACK()

Системная функция $STACK() возвращает информацию о состоянии стека текущего процесса. Используя ее, можно определить, в какой последовательности и каким способом были созданы отдельные фреймы стека.

У процесса есть текущее состояние стека и состояние, доступное для анализа возникшей ошибки. При нормальном выполнении процесса, когда состояние ошибки отсутствует, эти оба состояния стека совпадают, и функция $STACK() возвращает только текущее состояние, как оно есть на момент выполнения.

В случае, если процесс переходит в состояние ошибки, то функция $STACK() предоставляет больше информации. На стеке сохраняются все уровни, которые могут быть предоставлены для анализа. Например, если ошибка произошла на уровне стека N, а обработчик ошибки выполняется на уровне $N - 4$, то функция $STACK() предоставляет информацию

также до уровня N включительно, и разработчики могут уточнить, как именно и где произошла ошибка.

После отмены состояния ошибки присваиванием

```
set $ecode=""
```

функция $STACK() возвращает только текущее состояние стека.

В отличие от системной переменной $STACK, возвращающей текущий уровень стека, функция $STACK() в одноаргументной форме с параметром -1 возвращает доступное для анализа число стековых фреймов. Таким образом, это значение может быть равно $STACK в состоянии отсутствия ошибки, в состоянии максимального номера фрейма или быть больше, чем $STACK в состоянии ошибки.

У системной функции $STACK() есть две формы - одноаргументная и двухаргументная. Одноаргументная форма принимает в качестве параметра целое число, означающее:

< -1	Зарезервировано, при таких значениях в зависимости от реализации MUMPS системы могут генерироваться ошибки или возвращаться пустая строка.
-1	Возвращает число стековых фреймов, доступных для анализа, может быть больше чем $STACK.
0	Возвращает строку с зависимым от реализации MUMPS системы способом старта процесса.
> 0, <= $STACK(-1)	Возвращает результат в зависимости от состояния стекового фрейма с указанным номером. Если фрейм создан командой (в настоящее время это XECUTE или DO), то возвращается название команды в верхнем регистре. Если фрейм создан вызовом функции с возвращаемым значением, то возвращается строка $$.
> $STACK(-1)	Возвращает пустую строку.

Покажем пример выдачи результатов одноаргументной формы функции $STACK() без состояния ошибки программой:

```
proc ; k  d proc^STACK w
 x "d sub"
```

```
 q
sub
 n v s v=$$func()
 q
func()
 d show
 q 123
show
 w "available: ",$st(-1),!
 n i f i=0:1:$st(-1) w i,": ",$st(i),!
 q
```

При выполнении этой программы в Caché получаем:

```
USER>k  d proc^STACK w
available: 5
0:
1: DO
2: XECUTE
3: DO
4: $$
5: DO
```

И при выполнении этой же программы в MiniM получаем:

```
USER>k  d proc^STACK w
available: 5
0: XECUTE,CONSOLE
1: DO
2: XECUTE
3: DO
4: $$
5: DO
```

Здесь показывается, в каких контекстах были созданы каждый из фреймов и отличие выдачи программы в части, зависимой от реализации.

Двухаргументная форма команды $STACK() возвращает более детальную информацию о заданном уровне стека. Она принимает первым аргументом номер уровня стека и вторым аргументом строку с типом состояния этого уровня. Тип детализации или код состояния может быть:

ECODE	Код ошибки, произошедшей на этом уровне стека или пустая строка если ошибки на этом уровне не происходило.
MCODE	Если система может показать исполнявшуюся строку команд, то возвращает ее. Если не может, то возвращает пустую строку. Строкой команд может быть как строка рутины, так и аргумент исполнявшейся команды XECUTE.

PLACE	Положение строки кода в рутине, если строка принадлежит рутине, или символ @, если исполнялась команда XECUTE.

Второй аргумент функции $STACK() используется нечувствительно к регистру. Остальные значения кодов зарезервированы для будущего использования, а коды начинающиеся на символ Z, зарезервированы за реализациями MUMPS систем и могут быть использованы на их усмотрение.

Модифицируем программу так, чтобы она показывала также коды состояния стека:

```
proc ; k  d proc^STACK w
 x "d sub"
 q
sub
 n v s v=$$func()
 q
func()
 d show
 q 123
show
 w "available: ",$st(-1),!
 n i,c
 f i=0:1:$st(-1) d
 . w i,": ",$st(i),!
 . f c="ecode","place","mcode" w c,": ",$st(i,c),!
 q
```

При ее выполнении показывается состояние стека с детализацией, какой код и на какой строке исполнялся:

```
USER>k  d proc^STACK w
available: 5
0: XECUTE,CONSOLE
ecode:
place: @
mcode: k  d proc^STACK w
1: DO
ecode:
place: proc+1^STACK
mcode:  x "d sub"
2: XECUTE
ecode:
place: @
mcode: d sub
3: DO
```

```
ecode:
place: sub+1^STACK
mcode:  n v s v=$$func()
4: $$
ecode:
place: func+1^STACK
mcode:  d show
5: DO
ecode:
place: show+3^STACK
mcode:  f i=0:1:$st(-1) d
```

Сами строки выдачи результата

```
 . w i,": ",$st(i),!
 . f c="ecode","place","mcode" w c,": ",$st(i,c),!
```

в собственно выдачу результата не попали, поскольку предельный уровень стека для анализа был определен на предыдущем по отношению к этим командам уровне стека.

Модифицируем программу так, чтобы при ее выполнении возникла ошибка и программа показала состояние стека с учетом ошибки:

```
proc ; k  d proc^STACK w
 n $es,$et s $ec=""
 s $et="q:$es  d show s $ec="""" d show"
 x "d sub"
 q
sub
 w 1/0
 q
show
 w "current: ",$st,!
 w "available: ",$st(-1),!
 n i,c
 f i=0:1:$st(-1) d
 . w i,": ",$st(i),!
 . f c="ecode","place","mcode" w c,": ",$st(i,c),!
 w "----------------------------",!
 q
```

При ее выполнении показывается сначала состояние стека в состоянии ошибки, а затем, после сброса ошибки

```
s $ec=""
```

состояние стека без состояния ошибки (работа в MiniM):

```
USER>u $p:/lines=50 k  d proc^STACK w
current: 2
available: 3
0: XECUTE,CONSOLE
ecode:
place: @
mcode: u $p:/lines=50 k  d proc^STACK w
1: DO
ecode:
place: proc+3^STACK
mcode:  x "d sub"
2: XECUTE
ecode:
place: @
mcode: d sub
3: DO
ecode: ,M9,
place: sub+1^STACK
mcode:  w 1/0
------------------------------
current: 2
available: 2
0: XECUTE,CONSOLE
ecode:
place: @
mcode: u $p:/lines=50 k  d proc^STACK w
1: DO
ecode:
place: proc+3^STACK
mcode: q:$es  d show s $ec="" d show
2: DO
ecode:
place: show+4^STACK
mcode:  f i=0:1:$st(-1) d
------------------------------
```

Работа той же программы в Caché немного отличается:

```
USER>k  d proc^STACK w
current: 2
available: 3
0:
ecode:
place: @ +2
mcode: K  D proc^STACK W
1: DO
ecode:
place: proc+3^STACK +1
mcode:  x "d sub"
```

```
2: XECUTE
ecode:
place: @ +1
mcode: d sub
3: DO
ecode: ,M9,
place: sub+1^STACK +1
mcode:  w 1/0
-----------------------------
current: 2
available: 2
0:
ecode:
place: @ +2
mcode: K  D proc^STACK W
1: DO
ecode:
place: show+6^STACK +4
mcode: q:$es  d show s $ec="" d show
2: DO
ecode:
place: show+4^STACK +2
mcode:  f i=0:1:$st(-1) d
-----------------------------
```

Ключевое отличие состоит в том, как определяется положение строки, вместо которой исполняется строка с командами обработчика ошибок

```
mcode: q:$es  d show s $ec="" d show
```

С одной стороны, команды обработчика ошибок исполняются вместо строки рутины, с другой стороны они сами по себе не имеют положения в тексте рутины. Думается, что стандарт языка в данном случае дает неоднозначность толкования и неполноту определения функции $STACK() для случая, если выполняется обработчик ошибок $ETRAP. Для данного случая имеет смысл добавить индикатор, что на этом уровне продолжает выполняться строка обработчика ошибок, например

```
$STACK(level) = COMMAND, MODE
```

чтобы вместо строки

```
1: DO
```

получить индикатор текущего контекста

```
1: DO,ETRAP
```

или дополнительный код состояния стека

```
$STACK(level,"ETRAP")
```

чтобы можно было определить, что строка команд не принадлежит рутине, а задана в обработчике ошибок, выполняющегося в этот момент, в $ETRAP.

Этот пример показывает отличие системной переменной $STACK и максимально доступного для анализа уровня $STACK(-1). Показано, что в состоянии ошибки система удерживает максимально доступную для анализа ошибки информацию о стеке.

При этом выполнение обработчика ошибки

```
d show
```

в состоянии стека не отображается, хотя при его выполнении производится изменение состояния стека.

Этот нюанс очень важен, поскольку именно обработчик ошибки, строка команд указанная в $ETRAP, выполняется вместо строки, вызвавшей ошибку, и, если бы $STACK() возвращала информацию о стеке как он исполняется, то информация о собственно месте происхождения ошибки была бы утрачена.

К ключевой проблеме функции $STACK(), как видно из ее определения и поведения, можно отнести то, что она возвращает данные в зависимости от того, находится ли процесс в состоянии ошибки или нет. Технически, в процессе могут возникнуть также ошибки в обработчике ошибок, сам обработчик ошибок может использовать и перезаписывать фреймы стека и использовать большие уровни, чем сохраняется для функции $STACK() в момент ошибки.

Выглядит более разумным такое определение функции $STACK(), чтобы ее поведение было точным и однозначным. Например, для того, чтобы $STACK() возвращала реальное состояние стека как он исполняется на текущий момент, а дополнительная функция $ESTACK() возвращала бы состояния стека в зависимости от заданного первым параметром номера ошибки. В этом случае программисты могут получить точные состояния стека для каждой из последовательности произошедших ошибок, пока процесс не выйдет из состояния ошибки присваиванием:

```
set $ecode=""
```

5.5 Трассировка

Трассировка, и как механизм, и как методика, относится к простейшим и общедоступным отладочным средствам. В некоторой степени трассировка является намного более детализированным вариантом логгирования работы программы.

В книге часто применяется трассирование выполнения программ для иллюстрирования их работы. В своей основе трассировка программ есть внедрение в программу команд вывода информации, необходимой и понятной для разработчика, а логгирование это внедрение в программу команд вывода информации, необходимой для администратора.

По логгированию обычно видна работы программы в сильно укрупненном виде, например операция старта или останова, как именно они закончились, происходили ли критические ошибки.

По трассировке программ видна последовательность их выполнения, значения необходимых для анализа величин - локальных, глобальных и системных переменных, вычисленные параметры оценки работы, например общее время или количество чего-то.

В отличие от логгирования, команды трассировки удаляются из готовой программы и, наоборот, добавляются для проведения отладки.

При локализации ошибки общий прицип поиска ошибок состоит в объединении следующих пунктов:

1. Составление контрольного теста для воспроизведения проблемы.

2. Определение критерия момента перехода программы в некорректное состояние.

3. Установка контрольных меток трассировки с выводом индикатора корректности.

4. Найдя моменты, между которыми возникает проблемное состояние, этот участок кода детализируется более подробно.

Последовательность таких шагов локализует место происхождения проблемы, последовательно сужая интервал между найденными состояниями "до ошибки" и "после ошибки".

В MUMPS системах зачастую в комплект инсталляции входит рутина или набор рутин для проведения трассировки состояния виртуальной машины исполнения. Обычно такие рутины именуются с префиксами, производными от Error Trapping:

```
^%ET
^%ETN
^%ETL
```

Рутины трассировки состояния виртуальной машины обычно имеют две точки входа - трассирование с завершением состояния обработки ошибок и трассирование без завершения состояния обработки ошибок. В этом случае их можно использовать в зависимости от необходимости.

При этом рутины трассировки по умолчанию производят сохранение состояния процесса в относительно универсальном виде, включая, например, все значения локальных переменных. В реальных прикладных системах сама такая информация может быть недостаточно информативной для анализа ошибки и многие прикладные системы содержат собственные средства трассировки состояния процесса и его отображения. В этом случае разработчики могут вывести более информативно состояние контекста процесса, например, кроме идентификатора текущего пользователя его более осмысленное описание, прочитанное из глобалов, а также состояние критичных для анализа глобалов.

Как вариант трассировки состояния виртуальной машины можно посмотреть пример вывода состояния стека с использованием функции $STACK().

Собственно сам вывод информации о состоянии процесса в осмысленном виде может занять большой объем, поскольку в него могут включаться значения $STACK(), системных переменных, локальных переменных, обращение к различного рода системно-зависимым функциям $VIEW() и командам VIEW, критичные для понимания работы программы записи в глобалах.

В большинстве случаев вывод данных производится не совсем так, как приведено в книге (в виде простого вывода на экран), хотя во многих случаях для локализации проблем и такого варианта может быть достаточно. Трассировка зачастую выполняется либо в файл, либо в глобал. После чего можно просмотреть состояние файла или глобала или применить к глобалу специальные средства поиска.

В некоторых случаях отладочная информация о выполнении программы может занимать мегабайты относительно одинаковых строк, в которых необходимо что-то найти или сопоставить. Поэтому, в тех случаях, когда по характеристикам выполнения программы ожидается очень большой вывод, разработчики чаще применяют вывод в текстовые файлы, а когда небольшой - в глобалы.

При выводе в текстовый файл (и вообще при любой трассировке) разработчики должны понимать, то функция трассировки должна изменить контекст выполнения процесса наименьшим образом, и необходимо

сохранять и восстанавливать значения системных переменных, которые могут быть критичны для выполнения процесса. К таким в первую очередь относятся:

1. Текущее устройство ввода-вывода, значение $IO.
2. В случае если значение $PRINCIPAL может быть изменено, то его также надо восстанавливать.
3. Значение последней глобальной ссылки (naked indicator), значение $REFERENCE или $ZREFERENCE.
4. Значение системной переменной $TEST.
5. Все используемые функцией трассировки локальные переменные не должны сохраниться после выхода из трассировки и, желательно, не должны измениться критические для работы процесса локальные переменные.

При многопользовательской работе необходимо также различать различные трассировки, выполненные разными процессами, запущенные разными пользователями. В этом случае разработчики при выводе в файл получают следующий номер файла, либо при выводе в глобал, получают следующий индекс глобала, например, системной функцией $INCREMENT(), и используя в имени индекса или глобала что-то идентифицирующее или пользователя, запустившего процесс, или компьютер с которого подключились к MUMPS системе. В этом случае в общем списке имеющихся трассировок можно относительно просто найти интересующую.

При работе функции трассировки могут возникнуть проблемы, которые она может быть неспособна преодолеть и функция должна корректно отнестись к ситуации невозможности выполнить трассировку. К таким условиям относится работа в состоянии нехватки ресурсов.

Нехватка ресурсов может наступить как до срабатывания трассировки (работа в контексте обработчика ошибки, возникшей из-за нехватки ресурса), так и во время (программа работает на пределе доступных ресурсов).

В практической работе автор сталкивался с такими ситуациями, когда программа исчерпывает доступный ей ресурс, переходит к обработке ошибок, но обработчик ошибок сам не может выполниться корректно, поскольку ему также необходимы какие-то ресурсы. Обычно такая ситуация характеризуется тем, что одновременно и программа не работает

корректно и, при этом, нет и не может появиться никаких диагностических сообщений.

В такую ситуацию могут попасть не только программы на MUMPS, но и любые другие, которые в обработчике ошибок полагают, что все необходимые им ресурсы доступны. Как раз при исчерпании таких ресурсов эти обработчики не могут выполниться корректно.

Те случаи, когда программа не выполняет необходимые действия и не сообщает об этом, хотя имеет такую возможность, относятся к ошибкам программистов. В частности, если утилита копирования файлов может ничего не сообщить на экран о неудаче копирования файла в случае полного исчерпания места на диске.

В MUMPS системах к типовым ситуациям нехватки ресурсов в контексте работы функций трассировки состояния относятся:

1. Исчерпание памяти для локальных переменных (ошибка STORE в системах Caché и MiniM).

2. Исчерпание доступного места в базе данных и невозможность его расширения из-за административного ограничения.

3. Исчерпание доступного места в базе данных и невозможность его расширения из-за исчерпания дискового пространства.

4. Исчерпание лимита на число одновременно открытых устройств ввода-вывода.

5. Исчерпание таблицы блокировок, если они используются при трассировке.

При правильном проектировании функции трассировки и обработчика ошибок такие ситуации должны определяться программно и, при невозможности выполнить трассировку обычными средствами, должны использоваться аварийные средства логгирования, не расходующие ресурсы MUMPS системы.

К таким могут быть отнесены встроенные в MUMPS системы средства логгирования в текущий лог самой MUMPS системы или средства логгирования, предоставляемые операционной системой. В Windows системах, в частности, к таким средствам могут быть отнесены средства Event Logging.

И, к числу проблем, специфических для трассировки в контексте СУБД, относится работа трассировки с сохранением информации в глобал в контексте транзакции. В случае, если после функции трассировки

процесс вызывает откат транзакции, могут быть также отменены и все записи, сделанные в глобал трассирования. В этом случае разработчики должны выбрать подходящее решение, поддерживаемое используемой MUMPS системой:

1. Трассирование в глобал, на который не распространяется откат транзакции.

2. Трассирование в глобал в базе данных, на которую не распространяется откат транзакции.

3. На время трассировки отключать режим журналирования для процесса.

Разумеется, что при использовании отключения режима журналирования он, после выполнения трассировки, также должен быть восстановлен в предыдущее состояние.

Несмотря на то, что общий принцип трассировки выполнения процесса и состояния процесса на первый взгляд очень простой, реальное и полноценное его исполнение может потребовать достаточно больших усилий из-за необходимости учитывать множество нюансов и самой разрабатываемой прикладной системы, и используемой MUMPS системы, и операционной системы.

5.6 BREAK

Команда BREAK относится к встроенным отладочным средствам. Команда декларирована в стандарте языка, но ее реальное поведение, параметры и использование никак не определены и отводятся на усмотрение реализации MUMPS системы.

Общий принцип работы команды BREAK состоит в применении терминального или консольного режима, интерактивном вызове команды для выполнения отладочных операций и организации ввода-вывода в консоли непосредственно оператором, разработчиком или администратором.

К отладочным операциям, выполняемым командой BREAK, относятся установка и снятие точек или условий останова, либо для явного перехода в режим интерактивной отладки.

В каждой из MUMPS систем реализация команды BREAK выполнена по-своему. MUMPS системы сохраняют общий стиль и назначение

команды BREAK, поэтому у различных реализаций могут быть и совпадающие нюансы этой команды.

Принцип использования BREAK состоит в расстановке в нужных программисту местах точек останова либо задании таких точек выполнением команды BREAK. В зависимости от параметров команды процесс либо продолжает выполнение, либо останавливается в нужном месте и переходит в интерактивный режим. Программист может вывести на экран значения переменных, вызвать системные функции $VIEW() и команду VIEW для определения состоянияи продолжить выполнение программы либо по одной команде либо далее целиком как есть, либо прервать полностью.

Синтаксис применения BREAK в MUMPS системе Caché:

```
BREAK:pc status
BREAK:pc "extend"
```

В первой форме команда отключает либо включает прерывания по комбинации Ctrl+C, а во второй форме устанавливает тип останова для перехода в интеративный режим.

В зависимости от значения параметра "extend" процесс может перейти в режим исполнения по одной команде, по одной строке команд, либо отключить остановы по команде BREAK и далее продолжить выполнение программы. В состоянии ожидания ввода программиста система поддерживает безаргументную форму команды GOTO для продолжения выполнения программы.

Положим, что есть программа для демонстрации BREAK:

```
STEPS
 B "S+"
 N I
 F I=1:1:3 W I,!
 Q
```

При ее выполнении получаем пошаговое прохождение ее выполнения, фрагмент работы программы:

```
USER>d STEPS^BREAK

 B "S+"
 ^
<BREAK>STEPS+1^BREAK
USER 2d0>g

 N I
```

```
 ^
<BREAK>STEPS+2^BREAK
USER 2d0>g

 F I=1:1:3 W I,!
 ^
<BREAK>STEPS+3^BREAK
USER 2d1>g

 F I=1:1:3 W I,!
           ^
<BREAK>STEPS+3^BREAK
USER 3f2>g
1
 F I=1:1:3 W I,!
             ^
<BREAK>STEPS+3^BREAK
USER 3f2>
```

При этом MUMPS система Caché дополнительно предлагает расширенную и намного более мощную по своим возможностям команду ZBREAK, которая работает также в контексте терминального режима:

```
ZBreak location[:action:condition:execute_code]
```

Значения параметров команды:

location	Задает место останова в виде метки или, если начинается с символа "*", задает имя локальной переменной для останова при изменении этой локальной переменной.
action	Задает режим перехода в отладочное состояние с ожиданием ввода от программиста, аналог параметра "extend" для команды BREAK.
condition	Задает условие срабатывания точки останова, выражение на языке MUMPS. Если параметр опущен, то условие считается истинным по умолчанию.
execute_code	Строка команд, которую надо выполнить при переходе в режим останова.

Например, задание точки останова для переменной A и с проверяемым условием останова:

```
ZBreak *A::"'$D(A)"
```

Эта точка останова срабатывает при таком изменении локальной переменной A, после которого значение выражения

```
'$D(A)
```

становится истинным, например:

```
%SYS>K A
K A
^
<BREAK>
%SYS 1s0>
```

Еще несколько примеров установки точек отладки из документации по Caché:

Установить точку останова при переходе на строку TAG^ROU с переходом в режим построчного прохождения:

```
ZBreak TAG^ROU:"L"
```

Установить точку останова при переходе на строку TAG^ROU, но не останавливаться (action="N"), и если выполняется условие $X < 1$, то записать в переменную FLAG значение переменной X:

```
ZBreak TAG^ROU:"N":"X<1":"S FLAG=X"
```

Полный перечень параметров и опций команд BREAK и ZBREAK приведен в документации на MUMPS систему Caché. При использовании команд отладки также необходимо изучить способы продолжения выполнения программы, команды GOTO, QUIT и ZQUIT, зависимые от реализации и версии системы.

В MUMPS системе GT.M команда BREAK задает останов выполнения программы и выполнение указанной строки команд перед переходом в интерактивный режим отладки:

```
B[REAK][:tvexpr] [expr[:tvexpr][,...]]
```

Если выполнение программы доходит до команды BREAK, то проверяется ее постусловие, и, если оно выполняется, то процесс переходит в интерактивный режим. При этом выполняются задаваемые строки команд.

Выдержка из документации на команду BREAK в системе GT.M:

```
GTM>ZPRINT ^br
br;
 kill
 for i=1:1:3 do break;
 quit
break;
 write "Iteration ",i,?15,"x=",$get(x,"<UNDEF>"),!
 break:$data(x) "write ""OK""",!":x,"write ""Wrong again""",!":'x
 set x=$increment(x,$data(x))
 quit
GTM>DO ^br
Iteration 1    x=<UNDEF>
Iteration 2    x=0
%GTM-I-BREAK, Break instruction encountered
            At M source location break+2^br
GTM>ZCONTINUE
Wrong again
%GTM-I-BREAK, Break instruction encountered
            At M source location break+2^br

GTM>ZCONTINUE
Iteration 3    x=1
OK
%GTM-I-BREAK, Break instruction encountered
            At M source location break+2^br

GTM>ZCONTINUE
%GTM-I-BREAK, Break instruction encountered
            At M source location break+2^br

GTM>ZCONTINUE

GTM>
```

Здесь при выполнении кода процесс переходит в интерактиный режим отладки и сообщает о причине останова, а продолжение выполнения процесса происходит по команде ZCONTINUE.

Кроме команды BREAK MUMPS система GT.M предлагает также более мощную расширенную команду ZBREAK, имеющую больше возможностей, чем команда BREAK:

```
ZB[REAK][:tvexpr] [-]entryref[:[expr][:intexpr]][,...]
```

Команда ZBREAK может установить и снять точку останова на указанной строке, проверить выполняется ли выражение и назначить счетчик на срабатывание точки останова, чтобы пропустить первые N остановов.

В этом разделе описания команд BREAK и ZBREAK приведены иллюстративно, чтобы составить предствление о характере их применения. Для применения на практике отладчика, основанного на командах BREAK и ZBREAK, программисту необходимо ознакомиться с документацией на используемую MUMPS систему, включая возможные особенности ее версии.

Принцип отладки, основанный на применении команд BREAK, в действительности, несмотря на кажущуюся сложность, прост и позволяет отлаживать CHUI программы или отдельные фрагменты кода в традиционном для MUMPS систем консольном режиме. К такому режиму относятся консоли, терминалы и различные телнет-клиенты. Фактически, программист может вести отладку, соединившись с MUMPS системой удаленно, и для работы такого отладчика не требуются дополнительные программные средства.

5.7 MiniM Debugger

Отладчик MiniM Debugger встроен в редактор исходного кода MiniM Routine Editor. Отладчик поставляется бесплатно в составе инсталлятора MiniM и всегда соответствует по своей версии той же версии MiniM. Использование отладчика MiniM Debugger из одной версии MiniM с другой версией MiniM не рекомендуется из-за возможных отличий в нюансах версий. При этом в остальном работа самого редактора MiniM Routine Editor возможна с другими версиями сервера MiniM.

Клиентская часть отладчика MiniM Debugger работает под операционной системой Windows и выполнена как Win32 приложение. При этом серверная часть может работать как на этом же, так и на другом компьютере и быть как 32 битной так и 64 битной.

Отладчик MiniM Debugger может подключиться как к имеющемуся процессу, так и запустить новый в виде консоли или фонового процесса, причем для запуска консоли необходимо, чтобы серверная часть работала на том же компьютере. Кроме того, поддерживается опция автоматического присоединения к следующему процессу MiniM, который будет запущен. В этом случае отладчик присоединяется к процессу независимо от способа его запуска, это может быть телнет доступ, обслуживание страницы MWA, подключение через MiniM Server Connect от любой клиентской программы, использующей его, или любой другой способ запуска процесса.

Отладчик MiniM Debugger не использует устройства ввода-вывода отлаживаемого процесса, и процесс исполняется в своем естественном

контексте, как есть.

Отсоединение от отлаживаемого процесса производится независимо от способа его запуска, был ли запущен новый процесс или отладчик присоединился к уже запущенному.

Точно также, независимо от способа запуска отлаживаемого процесса производится его принудительное завершение по команде отладчика. Таким образом, отладчик MiniM Debugger может терминировать выполнение произвольного процесса.

К функциям отладчика MiniM Debugger относятся:

1. Attach - присоединение к имеющемуся процессу или запуск нового, или ожидание запуска нового.

2. Run - продолжить выполнение процесса пока он не остановится на точке останова.

3. Pause - остановить выполнение процесса в его текущей точке выполнения.

4. Reset - принудительно завершить процесс.

5. Detach - отсоединиться от отлаживаемого процесса, после чего процесс продолжает свое выполнение естественным образом, без остановов на точках останова.

6. Step Over - выполнить строку команд, либо выполнить строку команд без входа на более глубокий уровень стека, либо выйти на предыдущий уровень стека в зависимости от хода выполнения процесса.

7. Step Into - выполнить строку команд, либо войти при ее выполнении на более глубокий уровень стека, либо выйти на предыдущий уровень стека, в зависимости от хода выполнения процесса.

8. Run to Cursor - выполнять процесс пока не остановится на строке где назодится курсор, без создания отдельной точки останова, либо пока не остановится на иной заданной точке останова.

Отладчик MiniM Debugger показывает о процессе его состояние в виде текущего стека исполнения, список установленных точек останова, список вычисляемых выражений и их значения в процессе в состоянии останова, и предопределенный список системных переменных процесса.

Точки останова могут быть объявлены или сняты независимо от того, исполняется ли отлаживаемый процесс, или остановлен отладчиком.

Если точки останова объявлены в редакторе MiniM Routine Editor, то по завершению процесса и старте нового все они будут применены также к новому процессу и не будут утрачены с локальными переменными завершившегося процесса.

Точки останова могут быть установлены только на строку в рутине, и точке нельзя добавить условие срабатывания. Точки останова на изменение переменных не могут быть установлены.

Отладчик MiniM Debugger может отлаживать только непосредственный исполняемый код, и точки останова могут быть установлены только в рутинах типа INT.

Выражения для просмотра их значений (Watch) могут быть добавлены как из списка выражений, так и из редактора кода командой основного меню.

Отладчик MiniM Debugger может показывать значения локальных переменных и выражений во всплывающей строке подсказки в состоянии останова. Значение показывается при наведении курсора на имя локальной переменной или при наведении на выделенный фрагмент текста. В случае если синтаксически это выражение соответствует значимому с точки зрения языка MUMPS и вычисляется как строка, то значение показывается во всплывающей подсказке (Tooltip Evaluation).

Эта функциональность использует лишь фрагмент текста, если он синтаксически является именем локальной переменной (в этом случае выделение текста необязательно), либо выделенный фрагмент текста без учета какой именно синтаксической конструкции он принадлежит. В частности, может быть выделен фрагмент имени рутины как несколько символов, но Tooltip Evaluation получит в качестве синтаксической конструкции последовательность латинских символов, что синтаксически соответствует имени локальной переменной. Tooltip Evaluation может вычислить как простые значения, так и значения выражений произвольной сложности.

Вычисление выражений производится в контексте отлаживаемого процесса, и, если при вычислении выполняется побочное действие, то оно также останется в контексте процесса, например модификация глобальных или локальных переменных при вычислении выражения с функцией $INCREMENT(). Таким образом, программист может написать дополнительные собственные функции на языке MUMPS, чтобы при их вычислении производился побочный эффект в виде присваивания, для модификации переменных из отладчика, равно как и специальные функции декодирования данных для показа значений в необходимом программисту виде.

Отладчик MiniM Debugger работает в режиме только построчного

пошагового исполнения отлаживаемого процесса, или переходит либо на уровень стека ниже, либо на уровень стека выше.

По клавишным комбинациям отладчик MiniM Debugger максимально приближен к продуктам Borland (Delphi, C++ Builder).

5.8 Caché Debugger

Отладчик Caché Debugger встроен в редактор исходного кода Caché Studio. Отладчик поставляется бесплатно в составе инсталлятора Caché и всегда соответствует по своей версии той же версии Caché. Использование отладчика Caché Debugger из одной версии Caché с другой версией Caché не рекомендуется из-за возможных отличий в нюансах версий.

Интерфейс отладчика выполнен как часть Caché Studio, соответственно вне зависимости от версии операционной системы на которой работает сервер, отладчик работает на операционной системе Windows. При этом отладчику не важно, выполняется отлаживаемый процесс на том же или на другом компьютере.

Отладчик Caché Debugger может подключиться к имеющемуся на сервере процессу Caché, запустить новый процесс, или запустить новый для выполнения ZEN или CSP страницы.

Новый процесс, запущенный с указанной метки, выполняется как новый JOB, и имеет устройством ввода-вывода по умолчанию такое, которое используется отладчиком Caché Debugger для показа вывода процесса. Обычно таким устройством является устройство типа |TCP| или терминал. Если отлаживаемый процесс обращается к команде READ, то выполнение такой команды останавливает процесс и переводит в ожидание, в котором программист не может ничего ввести. Для этого случая есть команда отладчика Interrupt, выполнение которой прерывает выполнение текущей исполняемой команды с переходом к следующей. Команды WRITE выполняются корректно, и весь вывод, включая служебный вывод отладчика, отображается в специальном окне Output.

К функциям отладчика Caché Debugger относятся:

1. Attach - присоединиться к указанному имеющемуся процессу.

2. Go - запустить выполнение процесса далее пока не сработает условие останова. Если не было текущего отлаживаемого процесса, то отладчик запускает новый, как указано в опциях запуска нового процесса.

3. Restart - завершает выполнение текущего процесса и запускает новый как указано в опциях запуска нового процесса.

4. Stop - заканчивает текущую сессию отладки в зависимости от способа присоединения к отлаживаемому процессу - если было присоединение к уже имеющемуся, то процесс продолжает работу без отладчика, если был запущен новый, то процесс завершает работу.

5. Break - остановить выполнение процесса в его текущей точке выполнения, где бы он ни находился.

6. Interrupt - прервать выполнение текущей выполняемой команды.

7. Step Into - выполнить очередную команду и остановиться перед следующей, а также выполнить вход в процедуру или функцию на один уровень стека глубже или на следующую итерацию цикла.

8. Step Over - выполнить текущую команду и остановиться перед следующей, при этом пропустить вызов процедуры или функции, создающей при выполнении уровень стека, или остановиться на команде, следующей за блоком команд (после безаргументной формы DO).

9. Step Out - выполнить шаг на один уровень стека выше для блока команд или вызова функции или процедуры, и остановиться на следующей команде на предыдущем уровне стека.

10. Run to Cursor - выполнять процесс пока не остановится на строке где назодится курсор, без создания отдельной точки останова, либо пока не остановится на иной заданной точке останова.

Отладчик Caché Debugger показывает о процессе его состояние в виде текущего стека исполнения, список установленных точек останова, список вычисляемых выражений и их значения в процессе в состоянии останова.

Точки останова могут быть установлены как на место в коде, так и на локальные переменные. Точка останова по строке проверяется при передаче управления на первую команду в этой строке, а точка останова по локальной переменной проверяется при любом изменении этой локальной переменной, любого ее узла. Оба типа точек останова могут иметь необязательные условия останова, при выполнении которых отладчик останавливает процесс. Если условие останова не указано, то условие считается всегда истинным. Если указано, то рассматривается

как выражение на языке Cache Object Script и вычисляется. В этом случае точка останова активируется, если это выражение вычислилось как истина.

Точки останова могут быть объявлены или сняты независимо от того, исполняется ли отлаживаемый процесс, или остановлен отладчиком. Если точки останова объявлены в редакторе Caché Studio, то по завершению процесса и старте нового все они будут применены также к новому процессу и не будут утрачены с локальными переменными завершившегося процесса.

Старшие версии Caché Debugger могут использовать для установки точек останова не только рутины типа INT, но и рутины препроцессора типа MAC, и корректно относиться к строкам, которые после препроцессирования не присутствуют в получаемой INT рутине.

Для добавления в список выражений для просмотра их значений надо включить показ окна Watch, и переместить путем Drag-n-Drop фрагмент текста в список окна Watch. Этот фрагмент текста впоследствии можно отредактировать.

Отладчик Caché Debugger работает в режиме пошагового прохождения по отдельной команде, или переходит либо на уровень стека ниже, либо на уровень стека выше.

По клавишным комбинациям отладчик Caché Debugger максимально приближен к продукту Microsoft Visual Studio.

Отладчик Caché Debugger всегда доступен разработчику при установке клиентской части Caché под Windows и при работе с сервером Caché той же версии. Сервер Caché при этом может работать под любой поддерживаемой Caché операционной системой.

5.9 Serenji Debugger

Serenji Debugger - это самостоятельный коммерческий продукт, появившийся много лет назад, и работающий с различными MUMPS системами. На момент написания книги он поддерживает следующие MUMPS системы:

1. Caché 4.0 или старше
2. GT.M 4.3-000 или старше
3. DSM 7.1 или старше
4. M21 2.14 или старше

5. MSM 4.4 или старше

Кроме функций отладчика этот продукт является также полноценным GUI редактором исходного кода с подсветкой синтаксиса и может работать с поддерживаемыми MUMPS системами как локально, так и удаленно по TCP/IP.

К возможностям отладчика Serenji Debugger относится возможность отладки самых различных типов процессов, это могут быть терминальные процессы, фоновые процессы, а также процессы, обслуживающие WEB страницы ZEN, CSP и WebLink в Caché, MSM-Activate в MSM, и приложения Visual M.

Клиентская часть Serenji Debugger выполнена в виде стандартного многооконного Windows приложения и работает на множестве операционных систем Windows. В настоящее время продукт непрерывно развивается и совершенствуется, и часть ранних операционных систем Windows снята с поддержки из-за отказа Microsoft их далее официально поддерживать.

По клавишным комбинациям отладчик Serenji Debugger максимально приближен к продукту Microsoft Visual Basic.

Отладчик поддерживает установку и снятие отладочной точки останова, просмотр стека, локальных переменных, список установленных точек останова, и возможность ввода и исполнения команд в контексте останова. При этом вывод исполняемых команд может быть показан в окне отладчика.

К функциям прохождения отлаживаемого процесса относятся функции:

1. Step in
2. Step over
3. Step out
4. Run to cursor
5. Go
6. Fast forward

К интересной возможности отладчика относится возможность переключения пошагового исполнения между режимами по каждой команде и по каждой строке команд.

Точки останова отладчика могут быть установлены с условиями:

1. Условие в виде выражения, при выполнении которого точка останова приводит к останову и переключению в отладчик.
2. Отложенный останов на заданное число срабатываний, после которого останов выполняется.
3. Точка останова на условие изменения данных.
4. Обычная точка останова, срабатывающая всегда.
5. Останов при переходе процесса в состояние ошибки.
6. Останов при начале выполнения процесса.
7. Однократный останов без запоминания точки останова (Run to cursor).

Точки останова могут быть установлены или сняты независимо от того, исполняется ли отлаживаемый процесс, или остановлен отладчиком, при этом нет ограничения на число установленных точек останова. Если точки останова объявлены в редакторе Serenji Debugger, то по завершению процесса и старте нового все они будут применены также к новому процессу и не будут утрачены с локальными переменными завершившегося процесса.

Как уже было сказано, отладчик Serenji Debugger является активно развиваемым продуктом и его разработчики тщательно и очень внимательно относятся к выявляемым в продукте проблемам и к самым малейшим нюансам совместимости с различными реализациями как MUMPS систем, так и операционных систем. Продукт выпускается независимо от производителей MUMPS систем, и с выходом новых их версий при необходимости повторно адаптируется под возможные изменения в версиях MUMPS систем и операционных систем.

Глава 6

Внешний мир

6.1 Общие принципы

С самого начала в языке MUMPS присутствовали встроенные в язык команды ввода-вывода, команды открытия, закрытия и переключения устройств ввода-вывода. За многие годы развития компьютерных систем MUMPS системы применяли появлявшиеся новые технологии и способы взаимодействия программ в виде все тех же устройств ввода-вывода. При появлении в будущем других новых технологий взаимодействия они также могут быть использованы в MUMPS системах путем добавления еще одного типа устройства и, с большой долей вероятности, с полным сохранением общей модели ввода-вывода.

Как уже было описано в разделе команд ввода-вывода, в MUMPS системах основная модель ввода-вывода - это модель последовательных устройств. Устройство в зависимости от его физического типа и способа открытия может поддерживать запись (данные выходят за пределы процесса), чтение (данные поступают в процесс) или и чтение и запись.

При и чтении и записи устройство производит автоматическое позиционирование в потоке на длину прочитанных или записанных данных. В отдельных специальных случаях так называемых блочных устройств или при открытии устройства в блочном режиме чтение и запись приводят к чтению и записи одного блока. Размер одного блока задается либо типом устройства либо режимом его открытия. Вне зависимости от того, что устройство является последовательным, команде чтения строки можно указать длину последовательности, которую необходимо прочитать.

В отличие от обычных функций нижнего уровня распространенных языков программирования, команда чтения строки в MUMPS при ука-

зании длины чтения действительно ожидает приема входных данных на всю указанную длину, а команда записи записывает всю указанную строку.

Что интересно, для команды чтения может быть указан таймаут ожидания, и команда вернет прочитанную последовательность либо если получена указанная длина, либо если истек таймаут. Таким образом, в MUMPS можно соединить пространство и время. Чтение с таймаутом используется, например, в интерактивном вводе в терминальных программах - процесс время от времени читает что было нажато с указанием таймауа, и если не было нажатий то продолжает отображение или обработку данных, иначе переходит к обработке нажатой клавиши.

Кроме штатно поддерживаемых устройств ввода-вывода, для тех случаев, когда нет способа организовать обмен данными последовательно из-за физического принципа действия, например UDP/IP, или когда необходимо обращаться к устройству для чтения его параметров состояния, в MUMPS системах используются либо специальные системные функции, либо обращение к функциям во внешних подключаемых модулях (DLL, SO), либо MUMPS система может предоставить тот же интерфейс через SSVN.

В модели устройств MUMPS инициатором обмена данными всегда выступает процесс MUMPS, переходя к чтению или записи данных. В случае если необходимо организовывать обмен данными по инициативе сторонней программы, нужно организовать процесс MUMPS так, чтобы он проверял, нет ли входных данных для обработки. Нужно учитывать, что в модели процессов MUMPS в настоящее время отсутствует многопоточность в понимании выполнения нескольких последовательностей команд в рамках одного процесса или событийность с прерываниями.

Модель событий и асинхронной активации MUMPS процессов предлагалась и рассматривалась в комитете по стандартизации MDC, и частично модель описана в стандарте как предложение к применению, но ни одна MUMPS система в действительности таких способов активации и выполнения не поддерживает.

6.2 Терминальный интерфейс

Отдельно и несколько особняком от общего ввода-вывода стоит терминальный интерфейс. В нем команда чтения выполняется внутренним кодом процесса, при этом для терминального интерфейса переход к исполнению введенной команды и возврат управления заканчивается восстановлением текущего устройства, чтобы можно было ввести следующую

строку данных и, при необходимости, восстановление режима отображения эха на экране (повтор, показ введенных символов). Эти переключения выполняются неявно. Кроме того, для терминальных устройств поддерживается специальная обработка входного потока данных таким образом, чтобы сигнал от нажатия Ctrl+C распознавался корректно и приводил к прерыванию по ошибке.

Кроме того, после возврата управления после исполнения введенной команды терминальные режимы могут проводить дополнительные действия по обработке ошибок по умолчанию, обычно это вывод диагностического сообщения в зависимости от типа и версии MUMPS системы.

К терминальным интерфейсам относятся способы ввода, комбинирующие ввод с клавиатуры в качестве входного потока для чтения и вывод на экран в качестве потока записи. Это могут быть в зависимости от типа MUMPS системы либо специальные программы, как терминал в Caché, либо консоль в понимании операционной системы, либо телнет-интерфейс, либо реальный физический терминал, подключенный по последовательному каналу.

При использовании MUMPS систем разработчики пользуются терминальным интерфейсом как основным для выполнения подпрограмм или запуска служебных утилит и первое, что разработчик ищет при использовании новой MUMPS системы или системы новой версии - это способ запуска предпочтительного для него терминала. Это может быть или встроенная консоль или телнет клиент. При запуске такой программы в MUMPS системе для ее обслуживания автоматически запускается новый процесс. Для выхода из программы и завершения процесса (но не завершения работы сервера в целом) используется команда HALT (H).

В отношении терминальных интерфейсов поддерживается соглашение о специальной трактовке управляющих последовательностей команды WRITE:

Команда	Действие
WRITE #	Очищает экран полностью и позиционирует каретку ввода в левый верхний угол.
WRITE !	Перевод строки с прокруткой экрана при необходимости.
WRITE ?N	Позиционирование каретки вправо по горизонтали до указанной позиции.

В определенном смысле, в качестве терминала могут быть использованы также электрические печатающие машинки, вывод в этом случае производится на бумагу и команда WRITE # приводит к переходу на

следующую страницу.

Также, для терминальных устройств поддерживается буквальная трактовка системных переменных $X и $Y. Они возвращают текущее положение каретки на экране и присваивание значений этим переменным перепозиционирует каретку на экране в указанное место. Применение терминальных устройств при использовании MUMPS систем было столь важно (и остается и сейчас), что специально для них в язык были введены эти системные переменные. Думается, что при появлении в будущем устройств ввода-вывода, сравнимых по важности с терминальными, для них также могут быть введены специальные и стандартные системные переменные, возвращающие характеристики состояния отображения, вместо набора отдельных системных функций.

При использовании стандартных терминальных устройств экран представляется как матрица символов 80x25. При выполнении команд записи терминальному устройству передается последовательность символов, которую он использует для отображения - либо выводит на экран, либо изменяет характеристики отображения на экране, режим показа символов. Размер экрана 80x25 это обычный размер по умолчанию, но во многих терминальных клиентах этот размер может быть изменен.

К первой группе передаваемых последовательностей от сервера к клиенту относятся печатные символы, отображаемые как есть, выводимые последовательно с переносом вывода на следующую строку, если текущая строка закончена. Это основной режим вывода данных, используемый для терминальных устройств. Например, набор команд вывода строк и комбинирование с форматом перевода строки производит вывод вполне читаемого текста.

Ко второй группе относится набор эскейп последовательностей. Эскейп последовательность состоит в целом из символа ESC, за которым следуют опции и последовательность завершается распознаваемым символом. От символа ESC (десятичный код 27, шестнадцатеричный 1B) отсчитывается следующий за ним символ. Если это символ окончания последовательности, то терминал выполняет команду соответствующую этому символу, иначе до символа завершения распознает значения параметров команды.

Набор поддерживаемых эскейп последовательностей зависит от типа устройства. Обычно устройства поддерживают наборы эскейп последовательностей VT, самым распространенным по применению из них является набор команд VT-100. Различные телнет-клиенты могут поддерживать как определенный набор эскейп последовательностей, так и несколько, и переключаться с одного на другой. Полные списки управляющих последовательностей приведены в соответствующих справочни-

ках.

Некоторые эскейп-команды, иллюстрирующие их применение:

EscB	Переместить каретку на одну строку вниз.
Esc[ValueB	Переместить каретку на Value строк вниз.

При выполнении таких последовательностей на экране форматируется:

```
USER>w 123,$c(27),"B",456
123
   456
USER>w 123,$c(27),"[3B",456
123


   456
USER>
```

Общий принцип применения эскейп последовательностей прост, и по сути, выполняется передача специального символа (ESC), при необходимости символа начала списка параметров, значения параметров и управляющего символа команды.

Передача нужного набора последовательностей обычно оформляется в виде библиотек соответствующих функций. При использовании таких библиотек разработчики могут управлять экраном и отображением на нем в разнообразных режимах псевдографики, управлять цветом символов и фона, организовывать псевдо - оконный терминальный интерфейс, отображать на экране разнообразные управляющие элементы ввода (рамки, заголовки, кнопки, строки ввода, списки).

По умолчанию MUMPS системы поддерживают набор так называемых мнемоник, которые по сути и являются такими функциями, применяющимися к текущему устройству вывода. Мнемоника используется в командах чтения и записи как параметр команды, содержит имя и опционально собственные параметры, например:

```
W /CUP(5,12)
```

Из-за разнообразия наборов эскейп последовательностей и типов терминальных устройств разработчики зачастую поддерживают также наборы управляющих последовательностей, описывающие устройство. Такие наборы хранятся в глобалах и при необходимости выполнения действия библиотека читает из глобала необходимую для текущего типа устройства последовательность и выполняет ее. Если используется программа

с такими настройками, то подключение к ней еще одного типа терминального устройства становится простым делом - достаточно следовать инструкции на программу в части как описать или определить нужные последовательности.

Что интересно, различные терминальные устройства могут также возвращать различные последовательности символов для служебных и управляющих клавиш (Ctrl, Shift, Alt, F1-F12 и др.). На эти последовательности также может производиться настройка в зависимости от типа устройства.

Много лет назад, при появлении первых MUMPS систем в России (это была система MSM фирмы Micronetics), Рустем Османов, талантливый программист, разработал терминальную библиотеку под названием %WM. В нее входили функции чтения символов, управление выводом и организация примитивов - псевдоокна с рамками и заголовками, разнообразные управляющие элементы. На основе этой библиотеки был написан терминальный менеджер рутин и глобалов в стиле двухпанельного Norton Commander, и эта программа работала на множестве терминальных устройств, включая различные телнет - клиенты. Также эта библиотека использовалась, и сейчас используется, в прикладных разработках.

После того, как Рустем Османов ушел от нас, эту библиотеку использовали и модернизировали несколько различных фирм и в настоящее время существует несколько независимых форков этого проекта, пошедших различными путями развития и в некоторых случаях уже не совместимые друг с другом.

Независимо от библиотеки %WM, но на основе тех же принципов (Рустема Османова) и полностью с нуля другой талантливый программист, Андрей Вологдин, разработал аналогичную псевдооконную библиотеку %aWM и построил на ней как двухпанельный менеджер рутин и глобалов AltNC, так и несколько прикладных разработок. Библиотека %aWM отличается полнотой поддержки самых различных MUMPS систем, операционных систем и терминальных устройств. При появлении новой MUMPS системы эта разработка (%aWM) может быть портирована на нее достаточно просто и разработчики могут достаточно быстро получить сильный и гибкий инструмент.

Терминальные приложения отличаются от привычных в настоящее время двухуровневых клиент - серверных и трехуровневых с использованием промежуточных серверов приложений. Перед пользователем - экран с псевдографикой, достаточно крупные шрифты, и клавиатурный ввод. На экране размещается объем информации, необходимый для выполнения операции ввода, переход от одного управляющего элемента к другому выполняется обычно нажатием клавиши Enter.

Отсутствие необходимости попадать курсором мышки в небольшие поля и вообще отсутствие использования мыши приводит к существенному увеличению скорости ввода. Операторы, которым приходится вводить множество информации, предпочитают (при наличии альтернативы) именно псевдографические интерфейсы с клавиатурным вводом. Такие интерфейсы, в частности, используются в терминалах кассиров супермаркетов, на рабочих местах банковских сотрудников, при продаже билетов. Конечно, не все такие программы построены на MUMPS системах, и часть из них построены как толстый клиент, но принципы отображения информации те же.

Общий объем передаваемой от терминала к серверу и обратно информации столь мал по сравнению с другими способами организации клиентских мест, что множество терминалов могут обслуживаться как одним сервером с не самой высокой производительностью, так и связываться по каналам связи с практически произвольной пропускной способностью. Даже если серверу необходимо передать полностью все изображение экрана, в самом наихудшем случае это составляет $25*80 = 2000$ байт значимых данных. При использовании управляющих эскейп последовательностей объем может быть увеличен до примерно $8000 - 10000$ байт. Такой объем может быть передан в течении времени реакции оператора по практически любым каналам связи. При этом обычно нет необходимости передавать такие объемы непрерывно, и, в действительности, за время реакции оператора передаются в сотни раз меньшие объемы, для обновления только изменяемых позиций.

В терминальных программах нажатие символа обрабатывается на сервере - каждое нажатие отсылается серверу, в ответ терминалу возвращается инструкция что необходимо вывести на экран. И, фактически, уже в течении ввода строки в дополнительных управляющих элементах программа может сразу отображать изменение данных, например, при организации инкрементального ввода автоматически подбирать по индексу подходящие варианты или альтернативы.

Как следствие технических особенностей терминальных программ, они отличаются существенно меньшими потребностями вычислительных ресурсов, пропускной способности каналов связи, уменьшенным временем реакции сервера. При этом эргономика таких программ для операторов ввода практически оптимальна - в большинстве случаев операторам достаточно использовать малую часть клавиатуры, keypad, с цифрами и клавишей ввода.

В качестве операционных систем на клиентской стороне могут быть использованы практически любые, на которых может быть запущена терминальная программа, обычно это или телнет или SSH клиент с шифро-

ванием канала обмена. Поэтому терминально ориентированные программы обладают значительным временем жизни, существенно большим чем время жизни отдельных версий операционных систем, и существенной независимостью от аппаратных возможностей клиентских компьютеров - они могут быть изменены независимо от основной программы. И, кроме того, терминально ориентированные программы всегда работают с текущей, актуальной версией программы, размещенной на сервере и обновляемой централизованно.

6.3 Сокеты

С появлением в массовом использовании сетевого протокола TCP/IP он стал использоваться в MUMPS системах также, как и другие типы устройств, в качестве одного из типов, и в настоящее время все MUMPS системы поддерживают этот тип обмена в качестве одного из стандартных типов.

Протокол TCP/IP представляет собой протокол доставки пакетов в определенной последовательности с богатыми возможностями адресации и маршрутизации, поддерживающийся на множестве операционных систем.

С точки зрения программистов, это протокол передачи байт, двунаправленный, на каждом из концов соединения работает программа, используя сокет. Доставкой информации (последовательности байт) от одного сокета до другого занимается сетевая аппаратура и серверное программное обеспечение.

Для установления соединения между двумя программами необходимо, чтобы одна из них ожидала соединения. Соединение для такой ожидающей программы с точки зрения программистов характеризуется номером порта TCP/IP, в протоколах IpV4 их может быть всего до 65535. Часть из этих портов зарезервирована RFC для программ определенного назначения, остальные могут быть использованы на свое усмотрение. В принципе, с технической точки зрения, резервирование номеров портов не означает запрета на их использование, это только ожидаемый номер порта (или порт по умолчанию). Разумеется, все порты могут быть использованы программами на MUMPS, если они уже не заняты другими процессами.

Другая программа подключается к ожидающей на указанном компьютере и на указанном порту. Далее производятся операции чтения, записи, и соединение разрывается. В этом сценарии программа, ожидающая подключение, называется сервером, а подключающаяся к нему -

клиентом.

Все, что передается через сокеты, уже определяется договоренным протоколом обмена между этими двумя программами. Существует как множество стандартных протоколов обмена, так и частных, разработанных для применения в определенных программах. Например, протокол HTTP описывает что должен передать клиент и в каком кодировании, и что должен ответить сервер. В этом случае обе стороны называются соответственно HTTP - клиентом и HTTP - сервером.

Большинство протоколов семейства RFC в укрупненном виде построены на одних и тех же принципах - сначала передающая сторона (неважно, будь то клиент или сервер) передает заголовок запроса / ответа, за ним тело запроса / ответа с его данными. В заголовке описывается кому передаются данные, что необходимо сделать, как рассматривать способ кодирования данных, учетная информация о промежуточных серверах, и другое. При этом практически все протоколы и способы кодирования RFC используют одинаковые соглашения о способах кодирования, о кодировании времени, о формировании адресов, о шаблонах ответа и принципах формирования сообщений об ошибках.

Стандарт языка MUMPS не специфицирует точного синтаксиса именования устройств и в каждой из MUMPS систем такой способ именования устройств типа TCP/IP может быть свой собственный.

В качестве примера рассмотрим простенькую программку отправки сообщения по SMTP протоколу:

```
send
 n dev,str,to,saveio,err s to=15,saveio=$io
 ; mail message depended magics
 n From,To s From="mail@address.here"
 n subject s subject="This is mail subject"
 n message s message="This is mail message"
 s To=From
 ; server's depended code
 n smtpsrv,computer s smtpsrv="smtp.mail.address.here"
 s computer="computernamehere"
 ; vendor depended code
 s dev="|TCP|"_smtpsrv_":25"
 o dev:("rwt"):to  e  s err="open" g err
 ; common code here
 u dev
 r str:to i $e(str)'="2" s err="connect" g err
 w "HELO ",computer,!
 r str:to i $e(str)'="2" s err="helo" g err
 w "MAIL From: ",From,!
 r str:to i $e(str)'="2" s err="mail" g err
 w "RCPT To: ",From,!
```

```
 r str:to i $e(str)'="2" s err="rcpt" g err
 w "DATA",! r str:to i $e(str)'="3" s err="data" g err
 w "From: ",From,!,"To: ",To,!
 w "Subject: ",subject,!
 w message,!
 w ".",! r str:to i $e(str)'="2" s err="end" g err
 w "QUIT",! r str:to
 u saveio c dev
 q
err
 u saveio c dev
 w "failed at ",err,!
 w
 q
```

Здесь отдельно собраны магические значения с именами серверов для отправки сообщения, с адресом SMTP почты и зависимый от производителя код открытия TCP устройства. Устройство типа TCP открывается, делается текущим, в него согласно RFC на SMTP сообщения записываются данные, читается ответ от принимающего SMTP сервера, и соединение разрывается.

Это простейший код передачи сообщения. Все более мощные возможности SMTP могут быть добавлены, согласно RFC, по тем же принципам - запись в устройство и чтение из устройства. К ним относятся кодирование национальных символов, передача сообщения в формате HTML, передача приложений к сообщению.

Собственно сокеты TCP/IP для программиста различаются на три большие группы по способу их использования:

1. Клиентский.
2. Серверный с собственным обслуживанием (listen).
3. Серверный с передачей обслуживающему процессу (accept).

Клиентский способ использования состоит в указании к какому серверу необходимо соединиться и по какому номеру порта. Обе характеристики указываются способом, зависимым от производителя MUMPS системы. Такой клиентский сокет традиционно используется для передачи запроса серверу и получения ответа, либо для повторения этой операции многократно в рамках одного соединения. После выполнения задачи клиентская сторона закрывает соединение, а серверная продолжает ожидать дальнейших подключений. В зависимости от организации серверной стороны она может как обслуживать только одно соединение,

так и несколько - пока клиентское соединение обслуживается, сервер может принять и другие входящие соединения. Пример такого простого клиентского соединения и был приведен ранее, с отсылкой почтового SMTP сообщения.

Серверный сокет характеризуется лишь номером порта. Процесс открывает этот сокет на своем компьютере на заданном порту и ожидает входящие соединения. Способ ожидания входящих соединений также может отличаться для различных MUMPS систем. Часто ожидание входящего соединения для TCP устройства выполняется в виде команды чтения. Если команда вернула управление, то это означает, что соединение выполнено. Реже операцию ожидания входящих соединений выполняют в виде команды USE со специальной опцией устройства. Если команда выполнилась, то дается способ проверить было ли выполнение прервано или было выполнено соединение.

Далее серверные сокеты уже различаются по способу обслуживания запроса - будет ли выполнять запрос и отвечать командами записи текущий же процесс (второй из перечисленных способов использования) или эта функция будет передана специально созданному для этих целей процессу (третий способ).

Для выполнения третьего типа (такую операцию также называют операцией accept) в MUMPS системах используется команда JOB с опцией передачи сокета новому процессу. Сокет, получивший входящее соединение на операции accept, как-бы разъединяется на два на стороне такого серверного MUMPS процесса, и вторая его копия передается в управление новому дочернему процессу. После этого серверный процесс также снова может перейти к операции accept и ожидать нового соединения. Способ передачи accepted - сокета новому процессу также зависит от применяемой MUMPS системы и описывается в документации.

Кроме того, возможен вариант, что в MUMPS системе и операции listen и accept выполняются в виде мнемоник /LISTEN или /ACCEPT. Хотя на нижнем уровне API операционной системы запуск нового процесса с передачей сокета существенно зависит от операционной системы, на уровне среды исполнения MUMPS в рамках MUMPS системы одного производителя такое расхождение может отсутствовать, и операции могут различаться не между операционными системами, а между MUMPS системами различных производителей. Поэтому при написании переносимого кода такие операции локализуют, чтобы их можно было легко заменить и использовать другие.

Серверный сокет обычно используется в интерфейсе подключения клиентских программ. На стороне MUMPS системы запускается процесс, ожидающий подключения в режиме accept. При подключении кли-

ента процесс при необходимости выполняет действия по проверке прав, возможности обслуживания, и запускает дочерний процесс с передачей ему сокета. И уже дочерний процесс обменивается с клиентской программой, принимая запрос и отправляя ответ. Характер и способ обмена образуют прикладной протокол. В случае если клиентский процесс отсоединился или соединение разорвано по иным причинам, серверный процесс выполняет действия по завершению работы.

Протокол обмена между клиентом и сервером задается передаваемыми заголовками запроса и параметрами запроса. Обычно для передачи данных используется один из способов определить конец передачи порции и для разделения одной порции данных от другой:

1. Критерий окончания - специальный символ, например перевод строки или другой.

2. Критерий окончания - тип команды, когда длина параметров определяется командой.

3. Критерий окончания - переданная ранее длина.

И в техническом отношении ничто не препятствует применению всех трех способов в одном протоколе. Например, пусть клиентской программе необходимо передать команду выполнить строку команд. В этом случае возможна такая договоренность:

1. Передать байт "X".

2. Передать два байта, означающих длину последущей строки команд.

3. Передать саму строку на указанную ранее длину.

На серверной стороне процессу остается прочитать байт, и если это код символа "X" то далее прочитать два байта, считая их длиной строки команд, и далее прочитать строку на указанную длину.

Также возможно, например, такое соглашение для команды выполнения строки команд:

1. Передать строку "XECUTE", завершающуюся переводом строки.

2. Передать строку команд, завершающуюся переводом строки.

Здесь на серверной стороне необходимо читать не на определенную длину, а до договоренного разделителя (в данном случае перевод строки). Естественно, что в этом случае в передаваемых данных не должен содержаться сам символ перевода строки.

По способу использования сокетов они также могут быть разделены на две группы, хотя между ними и нет четко определенной границы:

1. Серверные.

2. Сервисные.

К серверным сокетам могут быть отнесены те, которые используются скорее для выполнения одного запроса и этот запрос полностью решает задачу, либо для выполнения в рамках одного соединения нескольких запросов. В этом случае соединение удерживается для полного решения задачи.

К сервисным сокетам могут быть отнесены те, которые используются для только одного запроса, даже если он не решает задачу относительно целиком.

В частности, отправка SMTP сообщения - это набор сообщений в рамках одного соединения как от клиента серверу, так и ответы сервера клиенту. В случае же использования модели HTTP запросов один запрос может и не решать поставленную задачу целиком, и получение всей информации о странице может потребовать нескольких запросов, например отдельно для самой страницы, для входящих в нее скриптов Javascript, стилей, картинок. Поэтому в случае SMTP обмена протокол по способу своего использования скорее серверный, а в случае HTTP скорее сервисный.

Основным различием между способами использования - серверный или сервисный является общее время выполнения задачи. В случае если требуется установление нескольких соединений, то каждое из них требует собственные накладные расходы. Если задача может быть выполнена в виде одного запроса объемом в 10 килобайт, или в виде 10-ти запросов по 1 килобайт, то первый способ может работать в разы быстрее. Это особенно заметно при снижении объемов передаваемых за один раз (за одно соединение) данных.

Косвенно это также проявляется при использовании собственных протоколов обмена MUMPS систем с клиентскими программами. Хотя соединение установлено один раз и данные передаются в пределах одного соединения, возможна передача порций данных как по отдельности, так и единой строкой, например в виде строки с разделителями

или в виде списка. Второй вариант показывает обычно намного лучшее время, чем последовательные запросы на каждую из порций.

6.4 WEB

С появлением и распространением в сети Internet HTTP клиентов и HTTP серверов MUMPS системы также стали использовать и для написания как клиентских, так и серверных частей приложений, для интеграции приложений и обмена данными по HTTP протоколу.

В настоящее время распространенность HTTP клиентов такова, то многие пользователи именно запуск браузера считают синонимом слов "войти в интернет".

В этом разделе опишем основные принципы и способы интеграции сервера приложений с каналом связи по протоколу HTTP с точки зрения программистов MUMPS систем.

Соединение клиента и сервера выполняется по сетевому протоколу TCP/IP. Протокол HTTP это соглашение о передаче и формировании HTTP запроса и получении и трактовке ответа. Обе части, и запрос и ответ, состоят из двух частей - заголовок и тело. Заголовок формируется как набор строк. После заголовка следует одна пустая строка, отделяющая заголовок от содержания. Далее следует само содержание. В заголовках описывается, что запрашивается, что отвечается, в каком кодировании, информационные поля.

Чтобы выполнить запрос, одна из сторон, называемая HTTP клиентом, открывает сокет, и в параметрах соединения указывает с каким сервером и на какой порт следует соединиться. По умолчанию для HTTP серверов используется порт номер 80. Далее, при установлении соединения, клиент отсылает запрос и ожидает ответ. В ответ сервер отсылает набор заголовков, пустую строку и содержание ответа.

Полностью все соглашения о кодировании и передаче описываются в соответствующих документах RFC.

Обе стороны взаимодействуют через сетевой протокол TCP/IP, и неважно, какая программа работает с той стороны, на каком процессоре, на какой архитектуре, на какой операционной системе, и по каким промежуточным каналам производится пересылка сетевых пакетов. По сути, если среда разработки может получить программу, работающую с сокетами TCP/IP, то ее можно использовать для написания модулей интеграции не только для HTTP, но и для HTTPS, WAP, и других, а также тех, которые еще появятся. Поверх этих протоколов работают такие технологии, как SOAP, JSON, AJAX, RSS и другие, постоянно

находящиеся в наше время на слуху.

Заголовок запроса (ответа) состоит из набора строк, каждая из которых означает один из заголовков запроса (ответа). В начале строки пишется имя заголовка, далее двоеточие с пробелом или пробел, далее значение, далее после пробела с разделителями "," или ";" могут следовать детализирующие значения заголовка.

Например, запрос на получение страницы index.html, находящейся в корне сайта, выглядит так:

```
GET /index.html HTTP/1.1
```

Здесь GET - это типа запроса, /index.html - это имя ресурса на сайте, HTTP/1.1 - это уточнение версии протокола для запроса.

Вообще говоря, с точки зрения принципов построения HTTP взаимодействия для программистов, это и все. Остальное - это детали, описанные в справочниках RFC на протоколы и способы кодирования, в документации на готовые WEB сервера и их версии и на готовые WEB браузеры или другие HTTP клиенты и их версии. Строго говоря, детальный состав заголовков, их порядок и само и их наличие в запросе - это все носит рекомендательный характер. В развитием Интернет появляются новые версии как браузеров, так и серверов, самостоятельно вводящие практически полезные заголовки, которые далее могут быть в точности поддержаны и другими производителями. Программисты - народ пытливых умов, и они постоянно придумывают что-то новое.

Далее рассмотрим способы, как может быть организовано взаимодействие MUMPS системы с HTTP клиентом или HTTP сервером. MUMPS процесс может быть как клиентом, запрашивающим данные у HTTP сервера, так и сервером, отдающим данные HTTP клиенту. При этом серверная сторона может быть выполнена намного более разнообразнее, чем просто открытие сокета, запись - чтение и закрытие.

На серверной стороне требуется получить соединение, разобрать параметры, сформировать ответ, переслать ответ обратно. MUMPS система может быть включена в этом процессе на произвольном этапе, и в зависимости от выбранной точки последовательности в этой цепочке может отличаться способ ее применения.

Перечислим несколько из возможных и реализованных различными производителями вариантов генерации HTTP ответа в контексте MUMPS процесса:

1. MUMPS процесс может открыть серверный сокет TCP/IP, получить входящее соединение и сгенерировать ответ, отправив его обратно. Это вариант вебсервера, написанного на MUMPS.

2. Готовый WEB сервер, используя соглашения CGI, можно настроить на обработку запросов специального вида так, чтобы WEB сервер запускал непосредственно MUMPS процесс. Это вариант прямого вызова MUMPS процесса.

3. Промежуточная DLL для WEB сервера, поддерживающая протокол ISAPI или NSAPI, самостоятельно обращающаяся к MUMPS системе по обычному для нее протоколу подключения. MUMPS процессу при этом передаются параметры и контекст запроса (переменные окружения) в виде его локальных переменных. MUMPS процесс при этом, используя обычные команды WRITE, генерирует ответ. Сгенерированный ответ промежутоный DLL модуль возвращает WEB серверу, а тот уже возвращает его HTTP клиенту. При этом могут быть использованы уже готовые модули промышленных WEB серверов, такие как шифрование по протоколу HTTPS, компрессия ответа, трансформация параметров по определенным правилам. В таком варианте выполнен модуль WebLink.

4. Промежуточный CGI модуль (фильтр файлов) может обрабатывать специальный файл таким образом, чтобы специальные синтаксические конструкции в нем обрабатывались MUMPS процессом, а остальное передавалось как есть. В таком варианте выполнен модуль MWA.

5. Промежуточный DLL или CGI модуль (фильтр файлов) может обрабатывать специальный файл таким образом, чтобы специальные синтаксические конструкции в нем обрабатывались MUMPS процессом, так же как и в предыдущем варианте, но при первой обработке файла чтобы генерировалась специальная функционально эквивалентная рутина. И далее модуль уже может вызывать сгенерированную рутину, чтобы она командами WRITE генерировала содержание HTTP ответа. В таком варианте выполнен модуль CSP.

6.4.1 HTTP клиент

Интеграция MUMPS процесса в качестве HTTP клиента с WEB сервером для получения данных выглядит наиболее просто. Достаточно использовать поддерживаемые MUMPS системой устройства типа TCP и соглашения протокола HTTP.

Приведем вариант простого HTTP запроса к одному из серверов новостей, отвечающих данными по RSS:

```
HTTPREAD ; HTTP client demo
 q
RSS ; k  d RSS^HTTPREAD w
 n dev="|TCP|news.yandex.ru:80"
 n header,rss,str
 o dev:("rwt"):10 e  w "failed to connect",! q
 u dev
 w "GET /index.rss HTTP/1.0",!
 w "Host: news.yandex.ru:80",!!
 n $et="g exit"
 ; skip headers
 f  r str q:str=""  d
 . s header($i(header))=str
 ; read entire content
 f  r str q:str=""  d
 . s rss($i(rss))=str
exit
 u $p
 w "exit",!
 c dev s $ec=""
 w
 q
```

Этот пример приведен на языке MUMPS в диалекте MiniM.

Процесс открывает сокет в текстовом режиме, пишет в него два HTTP заголовка, идентифицирующих запрос новостей RSS с дополнительной пустой строкой, далее переходит к чтению ответа. Полный ответ состоит из двух частей - заголовок ответа (сохраняется в переменную header) и собственно содержания (сохраняется в переменную rss).

Здесь не производится отслеживание длины содержания, передаваемой в поле Content-Type, и по окончании входных данных (WEB сервер разрывает соединение) генерируется ошибка чтения. По ошибке чтения сокет закрывается и выводятся значения как заголовков, так и содержания ответа.

Конечно, для более реалистичного варианта получения ответа необходимо обратить внимание как на значение самих заголовков, так и на кодировку ответа, указываемую в самом ответе (RSS ответ возвращается в формате XML).

6.4.2 Вебсервер на MUMPS

Для организации вебсервера на MUMPS необходимо открыть согласно документации на используемую MUMPS систему серверный сокет, и, дождавшись подключения, принять заголовки HTTP запроса. После них,

в зависимости от типа запроса, могут быть данные запроса (например, в запросах типа POST или PUT). Разобрав содержание заголовков, надо сгенерировать заголовки ответа и содержание ответа.

Приведем простой вариант вебсервера, реагирующего на запросы максимально примитивно, но уже демонстрирующего принципы работы:

```
RUN ; run http listener
 n $es,$et="g:'$es exit",str
 n dev="|TCP|:81"
 o dev:("rwt"):2 e  w "open failed",! g exit
rep
 u dev:/ACCEPT
 j responce:(:$io)
 u $p w "responce was run",! g rep
 q
exit
 u $p c
 q
responce
 u $p:("rwt")
 n str,header
 f  r str:10 q:str=""  d
 . s header($i(header))=str
 n ans,crlf=$c(13,10)
 s ans($i(ans))="<html><head>"_crlf
 s ans($i(ans))="<title>Demo Http Responce</title>"_crlf
 s ans($i(ans))="</head>"_crlf
 s ans($i(ans))="<body>"_crlf
 s ans($i(ans))="This is demo http responce.<p>"_crlf
 s ans($i(ans))=$zcvt($zv,"o","xml")_"<p>"_crlf
 s ans($i(ans))="Now: "_$h_"<p>"_crlf
 s ans($i(ans))="Headers:<p>"_crlf
 s i="" f  s i=$o(header(i)) q:i=""  d
 . s ans($i(ans))=$zcvt(header(i),"o","xml")_"<p>"_crlf
 s ans($i(ans))="</body></html>"_crlf
 n len=0,i=""
 f  s i=$o(ans(i)) q:i=""  s len=len+$l(ans(i))
 w "HTTP/1.0 200 OK",!
 w "Content-Type: text/html",!
 w "Content-Length: ",len,!!
 s i="" f  s i=$o(ans(i)) q:i=""  w ans(i)
 h
```

Этот пример, как и предыдущий с HTTP клиентом, приведен на языке MUMPS в диалекте MiniM.

Здесь процесс открывает серверный сокет, и ждет подключения к нему извне. При подключении сокет просто передается дочернему процессу без дополнительных проверок.

Дочерний процесс переводит сокет в текстовый режим, чтобы операции и чтения и записи использовали стандартные соглашения о переводе строки.

Далее процесс вычитывает все заголовки HTTP запроса в локальную переменную header.

После чего генерирует ответ в локальную переменную ans, чтобы иметь возможность по окончании полной генерации ответа получить его общую длину в байтах. Вообще говоря, в протоколах HTTP предусмотрена возможность не передавать длину ответа, в этом случае клиент должен получать все, что следует далее, но часть HTTP клиентов не придерживаются этого правила.

По окончании генерации ответа процесс передает HTTP клиенту ответ, состоящий из заголовков и самого содержания ответа.

В таком исполнении программистам на MUMPS, в принципе, доступны как все возможности по генерации произвольного ответа с самыми сложными правилами и нюансами, так и появляется необходимость дополнительной работы в случае если необходимо применять парные к HTTP запросам темы - шифрование HTTPS, авторизация, компрессия, кодирование, генерация графического содержимого.

Как именно относиться к параметрам запроса в таком вебсервере - полностью определяет программист, например запрос

```
http://server/abc/def/klm?123&678
```

Может быть тактован так:

1. Использовать abc как имя базы данных, в которую необходимо переключиться.
2. Использовать def как имя рутины.
3. Использовать klm как имя метки.
4. Использовать 123 и 678 как значения первого и второго параметров.

Или использовать иные произвольные и непротиворечивые соглашения.

При обращении к такому простому HTTP серверу браузером IE9 получаем ответ вида:

```
This is demo http responce.
MiniM for Windows 32 bit 1.14 release build
Now: 62697,72396
```

```
Headers:
GET / HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: ru-RU
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0;
Windows NT 6.1; WOW64; Trident/5.0)
Accept-Encoding: gzip, deflate
Host: localhost:81
DNT: 1
Connection: Keep-Alive
```

При обращении иным браузером заголовки могут быть другими, в частности при запросе браузером Opera9 получаем:

```
This is demo http responce.
MiniM for Windows 32 bit 1.14 release build
Now: 62697,72389
Headers:
GET / HTTP/1.1
User-Agent: Opera/9.63 (Windows NT 6.1; U; en) Presto/2.1.1
Host: localhost:81
Accept: text/html, application/xml;q=0.9,
application/xhtml+xml, image/png, image/jpeg,
image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: ru-RU,ru;q=0.9,en;q=0.8
Accept-Charset: iso-8859-1, utf-8, utf-16, *;q=0.1
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Connection: Keep-Alive, TE
TE: deflate, gzip, chunked, identity, trailers
```

Конечно, такой вариант вебсервера довольно трудоемок и требует написания специфических для HTTP протокола и способов кодирования модулей, равно как и получение дополнительной информации о полученном подключении обращением к системным функциям. Реализация грамотного WEB сервера на MUMPS это действительно трудоемкая задача, из-за обилия различных нюансов кодирований и трактовок параметров и заголовков HTTP запросов.

Все остальные варианты генерации HTTP ответа с использованием MUMPS - это различного рода упрощения с получением параметров запроса и характеристик соединения таким образом, что разработчику остается все меньше работы либо таким образом, чтобы использовались уже готовые модули кодирований и трансформации.

Применение готовых WEB серверов дает бóльшую возможность в независимой административной их настройке и подключении сложных подсистем - шифрования, компрессии, подстановки правил, кеширования и проксирования.

С одной стороны, возможность административно подключить или отключить, или перенастроить по справочной документации дополнительный модуль промышленного WEB сервера может оказаться существенным плюсом. С другой стороны, применение сторонних программных комплексов по отношению к MUMPS системе также влечет и необходимость заниматься намного большим комплектом программного обеспечения - инсталлировать и обновлять версии, равно как и искать возможные проблемы работы в общей, интегральной схеме. Настройка большого комплекта программ с их собственными нюансами может оказаться намного более разнообразной задачей, чем просто запуск MUMPS процесса при старте MUMPS системы.

6.4.3 CGI

Классический интерфейс CGI состоит в использовании программой стандартных каналов ввода вывода stdin + stdout. Вебсервер, получив запрос от браузера, запускает указанный процесс и формирует ему необходимые параметры окружения, через которые и передаются параметры запроса из URL.

Обычно в каталог cgi-bin или функционально аналогичный ему не размещаются исполняемые файлы серверов СУБД, поэтому в них размещают специальные командные файлы, в которых первой строкой указывают через шебанг что необходимо выполнить. В запросе вебсерверу указывают уже этот командный файл. Например, организуем запрос к файлу demo.m размещенном в каталоге cgi-bin:

```
http://localhost/cgi-bin/demo.m?param=123
```

Сам файл demo.m составим с указанием правила запуска MUMPS процесса. Разумеется, тут могут быть использованы только те MUMPS системы, которые поддерживают ввод и вывод данных через стандартные каналы stdin и stdout. В параметрах запуска нужно указать, что необходимо выполнить. Параметры процесса у каждой MUMPS системы отличаются и необходимо определить по документации правила их описания. Положим, что необходимо выполнить на сервере рутину DEMO, тогда файл demo.m может выглядеть так:

```
#! serverdir\mumps -xecute "d ^DEMO"
```

Здесь точное значение параметров зависит от MUMPS системы.

Сама выполняемая рутина DEMO уже полностью формирует HTTP ответ для вебсервера:

```
WEB
 w "Content-Type: text/html",$c(10,10)
 w "<html>",!
 w "<head><title>Web Demo Response</title></head>",!
 w "<body>",!
 w ....
 w "</body>",!
 w "</html>",!
 q
```

Также, в зависимости от MUMPS системы, необходимо определить по документации на нее, как получить переменные окружения.

Такой способ запуска, если он осуществим, прост и не требует настройки дополнительного программного обеспечения. Программист может программно генерировать произвольные вебстраницы, произвольного типа и полностью реализовывать сложные формы взаимодействия с вебклиентом (браузером).

6.4.4 WebLink

WebLink это ISAPI модуль для промышленных WEB серверов. В выполнении запроса используется обращение к dll с передачей параметров, но при разработке используются файлы asp. Хотя технически и используется расширение для файлов asp, но WebLink работает с такими файлами самостоятельно, без использования модулей Microsoft Active Server Pages.

Файл WebLink содержит в качестве основного код в разметке HTML, а специальные теги ASP обрабатываются компилятором приложений WebLink. Результат обработки таких файлов получается в виде рутин MUMPS системы. После первой обработки файла уже вызываются сгенерированые рутины, отдающие содержание вебсерверу. Все обращения (ссылки) к станицам asp, входящим в приложение, компилятором WebLink автоматически заменяются на обращение к ISAPI модулю с подстановкой необходимых параметров.

Кроме того, сгенерированные или иные рутины могут быть вызваны непосредственно через DLL модуль.

Набор файлов и рутин объединяется в один пакет, называемый приложением. Все входящие в него элементы могут быть откомпилированы.

WebLink при компиляции файлов рассматривает специальные теги

```
<% %>
```

и символы подстановки

```
|expression|
```

как инструкции по генерации кода. Пример синтаксиса из документации:

```
<%@Language=Cache@%>
<%
 Set date=""
 For  Set date=$Order(^Contact(date)) Quit:date=""  D List
 G End ; jump past the HTML that lists the contacts
 ;
List ; get the details
 Set d=^Contact(date)
 Set VisitDate=date
 Set Contact=$P(d,"#",1)
 Set Notes=^Contact(date,"notes")
 ; now display the list
%>

<hr>
Date: |VisitDate|<br>
Our contact: |Contact|<br>
Notes of visit: |Notes|<br>
<br>

<%
 Quit
End ;
%>
```

При обработке такой страницы фактически генерируется рутина на MUMPS так, что все содержание страницы, не входящее в служебные теги, заменяется на команды вывести это содержание в текущее устройство. В служебных тегах могут быть организованы подпрограммы, переходы на метки, может быть использован блочный синтаксис. В приведенном примере вывод данных Date, Contact и Notes производится в цикле, как часть подпрограммы List.

По той же схеме выполняется обработка условных тегов, в них пишется переход на метку с служебном теге:

```
<%@Language=Cache@%>
<%
 If condition1 goto Cond1
%>
<p>This is some HTML to display
...etc
<%
 Goto End
```

```
Cond1 ;
%>
<p> Here is the alternative HTML to display
...etc
<%
End ;
%>
<p>
continue with the rest of the page
```

WebLink также поддерживает генерацию HTML кода для элементов управления путем присваивания в коде MUMPS значений специальным зарезервированным локальным переменным.

При процессировании всех страниц WebLink производит замену имен ссылок со страниц PageName.asp на вызов ISAPI модуля так, что все необходимые служебные параметры автоматически передаются в URL в виде:

```
http://server/scripts/mgwms32.dll?MGWLPN=My_LPN&wlapp=Contacts
```

При работе этот модуль обработки автоматически определяет по скомпилированному WebLink приложению и его служебным данным, какую рутину необходимо вызвать, как и какие параметры передать.

Страница WebLink может управлять генерацией заголовков HTTP ответа, используя специальный тег <HTTP>, при обработке которого также выполняется подстановка параметров:

```
<HTTP>
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: MyCookieName=MyCookie|number|;
  path=/; expires= Fri 08-Jan-1999 13:00:00 GMT
</HTTP>
<HTML>
```

Перед выполнением страницы в контексте MUMPS процесса создаются необходимые служебные локальные переменные с переменными окружения, которые страница может использовать при необходимости.

К интересным возможностям приложений WebLink относятся механизм передачи событий и параметров с клиентской стороны (браузер) на сервер (MUMPS). Например, если есть поле ввода, то можно установить обработчик события его изменения на серверной стороне и процессор WebLink при компиляции приложения автоматически сгенерирует код передачи таких событий и их параметров:

```
<INPUT TYPE=TEXT NAME=Username
   ActionOnChange=Mytest(Info,Password)>
...
<SCRIPT Language=Cache>
 //type=actions
Mytest(Info,Password) ; Test the username
 ...
 ...
 Quit
</SCRIPT>
```

WebLink работает с InterSystems Caché (NT, UNIX), а также с системами DSM и MSM (NT). Модуль разрабатывается фирмой M/Gateway Developments Ltd и, кроме него, фирма M/Gateway также выпускает множество модулей для WEB разработки с использованием MUMPS систем.

Что интересно, WebLink комплектуется программой для разработчиков WebLink Developer, который выполняет все необходимые действия по подготовке веб приложения для WebLink. Сам WebLink Developer выполнен также в веб интерфейсе.

6.4.5 MWA

MWA (MiniM Web Access) это модуль для промышленных WEB серверов, таких как Apache, IIS, и других, поддерживающих стандартные соглашения о CGI модулях.

Технологически модуль MWA является CGI фильтром - браузер получает запрос к файлу, но вместо возврата его содержания этот файл передается модулю, модуль его использует для генерации ответа, и уже ответ передается вебсервером обратно HTTP клиенту.

Для генерации ответа модуль запускает MUMPS процесс и, в зависимости от встреченных синтаксических конструкций, передает управление процессу, либо самостоятельно отдает в содержание какие-то фрагменты обрабатываемого файла.

В настройках вебсервера производится ассоциирование типа файла по расширению (mwa) с обработчиком MWA CGI и запрос к странице производится указанием файла mwa:

```
http://server/dir/page.mwa?param=value&...
```

Сам файл mwa является текстовым, и в нем все содержание отдается вебсерверу как ответ, за исключением синтаксических конструкций выделенных тегами

```
<?name arguments ?>
```

Здесь name - имя MWA тега, а arguments - аргументы тега. Все встреченные MWA теги обрабатываются MWA процессором и, в зависимости от тега, процессор либо самостоятельно заменяет тег на соответствующее ему содержание, либо передает управление MUMPS процессу, чтобы вычислить значение либо выполнить MWA тег.

MWA теги образуют по совокупности возможностей небольшой язык программирования, где часть операций выполняется на стороне процесса MiniM на языке MUMPS. Например, тег include включает указанный другой файл, тег eval вычисляет значение выражения и оно подставляется вместо тега, тег if пропускает или включает часть тела страницы до следующего условного тега, а тег while организует циклы обработки.

Кроме генерации содержания, MWA теги управляют заголовками ответа, чтобы задать веб баузеру, что он получает страницу HTML, данные XML, файл PDF, графический файл или другое.

Структурно выполнение MWA страниц полностью аналогично таким технологиям как PHP или ASP. При обработке файлов не генерируются промежуточные элементы, рутины, временные файлы, файлы MWA не компилируются, и для обновления приложения необходимо только обновить MWA файлы (останов вебсервера не требуется).

Например, страница с содержанием

```
<html><body>
Сегодня: <b><?eval $zd($h,13) ?></b>
</body><html>
```

генерирует HTML (тип ответа по умолчанию) страницу с текущей датой.

В контексте MUMPS процесса, обслуживающего страницу при генерации, все параметры запроса и переменные окружения доступны в виде локальных переменных.

6.4.6 CSP

CSP (Cache Server Pages) это технология как фильтра файлов (CSP страницы), так и обработка специального расширения (CLS), соответствующего сущности базы данных (класс). В настройках вебсервера (поддерживаются и IIS и Apache, причем на различных операционных системах) указывается что определенные расширения должны обрабатываться процессором CSP. Это может быть, в зависимости от версии или операционной системы, как стандартный CGI, так и ISAPI модуль.

Страница может быть создана как файл формата HTML, при первом к нему обращении процессор генерирует эквивалентный по функциональности код в виде класса Cache Object Script. Дальнейшая работа по генерации HTTP ответа выполняется этим классом. Если по странице CSP сгенерирован и скомпилирован соответствующий ей класс, то дальше можно обращаться не только к файлу, но и к самому классу, точно таким же образом, но указывая расширение CLS. Более того, если класс создан по определенным правилам, то файл CSP не требуется.

Все HTML теги, встреченные в странице процессором CSP, обрабатываются с заменой на вызов соответствующих этим тегам классов. Любой из HTML тегов можно переопределить. В зависимости от определения того, на что заменять тег, он может быть заменен на вывод просто текста командой WRITE или на обращение к определенному классу, чтобы его функции вывели необходимое содержание. У тега поддерживаются атрибуты, также передаваемые соответствующим классам.

Разработчики могут определить собственные правила как необходимо обрабатывать определенные теги, задав эти правила в CSR. Это файл определения класса в специальной разметке, где указано при каких ситуациях какие действия по генерации кода необходимо выполнять. Действия могут быть отнесены как на этап компиляции, так и на этап выполнения. Правило обработки тега CSR получает и атрибуты, и содержание тега, и контекст его работы. Также можно определять обработчик тега в зависимости от того, в какой из тегов он вложен.

Разработчики могут определить через CSR большой набор компонентов таким образом, чтобы указывать на странице CSP не громоздкий код, а компактный собственный тег. В какой-то мере стандартные теги HTML также заданы набором таких правил CSR.

Генерируемый по файлу CSP класс содержит набор методов, вызываемых для обработки определенного фрагмента страницы. Разработчики в самой странице CSP могут также определить собственные методы для будущего класса. В теле самой страницы разработчики могут свободно смешивать код HTML, код JavaScript, передаваемый клиентской стороне и код Cache Object Script, используемый либо для генерации страницы, либо для обработки событий страницы.

В случае если по CSP странице получается очень большой класс (по объему кода), то компилятор классов автоматически переносит часть кода по нескольким рутинам.

В состав технологии CSP входит кроме обработки страниц также работа с авторизацией, поддержка сессий, передача обработки событий с клиента на сервер.

Процессор CSP кроме обработки тегов также поддерживает макро-

подстановки и специальные макроподстановки CSP.

Положим, в коде CSP страницы есть фрагмент:

```
<b>Hello!</b>
```

При его обработке получается выполняющийся на стороне сервера при генерации HTTP ответа код

```
Write "<b>Hello!</b>",!
```

Этот код уже генерирует отдаваемый клиенту фрагмент

```
<b>Hello!</b>
```

Если встречаются специальные символы макроподстановок то они рассматриваются либо как подстановки этапа компиляции либо подстановки этапа выполнения. Если встречается код вида

```
 ##(expr)##
```

То он вызывается однократно на этапе компиляции и результат вставляется в вывод страницы:

```
This page was compiled on:
<b>##($ZDATETIME($H,3))##</b>
```

заменяется на

```
Write "This page was compiled on:",!
Write "<b>2000-08-10 10:22:22</b>",!
```

Если же встречается конструкция вида

```
 #(expr)#
```

то она заменяется на вычисление выражения expr при выполнении каждый раз:

```
This page was executed on:
<b>##($ZDATETIME($H,3))##</b>
```

заменяется на

```
Write "This page was compiled on:",!
Write "<b>",$ZDATETIME($H,3),"</b>",!
```

Кроме кода на языке Cache Object Script, разработчик может добавить классу SQL запрос, объявив его тегом

```
<script language=SQL>
```

и используя в дальнейшем в функциях на COS.

За счет того, то каждый из тегов HTML, встреченных при обработке, заменяется на соотвветствующим образом определенный ему код, поддерживается набор тегов управления CSP:IF / CSP:WHILE / CSP:LOOP.

Кроме генерации содержания HTTP ответа, разработчику CSP страниц также доступно управление заголовками HTTP ответа с тем, чтобы веб клиент колучал корректное описание ответа. В частности, можно указать что передается не HTML страница (тип ответа по умолчанию), а XML или графический файл. Путем обращения к методам базового класса, от которого наследуется CSP класс, можно управлять куками, перенаправлять на другую страницу, и управлять другими операциями, характерными для HTTP протокола.

Одной из значимых возможностей технологии CSP является встроенная поддержка HiperEvent - передача обработчиков событий с клиента на сервер. Если на CSP странице процессор встречает обращение из JavaScript кода к серверному методу, то на страницу автоматически помещается код вызова такого события. Код обращается к серверу без перезагрузки всей страницы, вызывая специальную страницу брокера запросов. CSP процессор автоматически генерирует для него код передачи параметров и приема возвращаемого значения.

Технология CSP используется во множестве как Intranet проектов, так и Internet сайтов.

6.4.7 Проблемы HTTP

К общим проблемам применения MUMPS систем для генерации HTTP ответов относятся следствия самой природы обмена данными по HTTP и они не связаны со спецификой собственно самих MUMPS систем. К ключевым проблемам могут быть отнесены следующие, хотя в каких-то ситуациях могут иметь значение и другие:

1. Работа в недоверительной среде Internet, когда в параметрах запроса могут прийти непредусмотренные разработчиками данные.

2. Непредсказуемое количество процессов для обслуживания потока запросов, в отличие от Intranet приложений с прогнозируемой активностью.

3. Отсутствие постоянного соединения в течении всего времени работы Internet - приложения и соединение с сервером только для генерации отдельных порций.

При работе в среде интернет к приложению предъявляются дополнительные требования, о которых разработчики иногда не задумываются. При небрежном отношении к проектированию программного интерфейса и к входным данным могут создаваться уязвимости. Входные данные, принимаемые через параметры запроса, присылаются внешними средствами. Неправильно будет полагать, что параметры будут присланы только и исключительно со страниц разрабатываемого веб приложения. Формально говоря, на стороне сервера невозможно достоверно установить даже то, что запрос пришел именно от браузера, поскольку произвольные запросы могут быть сформированы любой программой, работающей с сокетами TCP/IP.

На серверной стороне при приеме параметров необходимо в любом случае, даже в самой незначительной на первый взгляд странице, внимательно относиться к использованию значений параметров. В среде выполнения MUMPS, как и во многих других с возможностью динамически конструировать выполняемое или вычисляемое выражение, зачастую с целью облегчения работы параметры могут быть использованы как часть синтаксической конструкции. На языке MUMPS к таким конструкциям относятся команды XECUTE, косвенность команд и меток, а также возможные нюансы поведения определенных версий MUMPS систем.

После написания обработчика веб страницы, а еще лучше до, необходимо проверить, как обработчики используют недоверительные данные. Нужно проверить все команды XECUTE, которые выполняют строки команд, сконструированные с использованием входных параметров. Программисту нужно оценить, можно ли в такие параметры вписать код, не предусмотренный разработчиком так, чтобы он выполнился. Точно также необходимо проверить все синтаксические конструкции, использующие косвенность.

Приведем простой пример. Положим, что в параметре запроса

```
http://server/dir/page?id=1234
```

значение параметра id используется для подстановки

```
; значение переменной id получено из параметра
x "s abc="_id
```

В этом случае в параметрах можно передать произвольные выполняемые команды, например

```
http://server/dir/page?id=1234%20k%20^DATA
```

что в результате выполнения команды XECUTE приведет к выполнению кода

```
s abc=1234 k ^DATA
```

В случае использования косвенности команд также существует возможность выполнения постороннего кода. У MUMPS систем могут быть нюансы выполнения косвенности команд, в частности существуют системы, которые выполняют косвенную форму команды как вариант команды XECUTE, например в одной из систем может выполняться:

```
USER>s @"a=123 w 123"

<SYNTAX> :SET ARGUMENT: *a=123 w 123
```

и при этом тот же код в другой системе:

```
USER>s @"a=123 w 123"
123
```

Кроме того, в стандартном выполнении косвенности имен допускается синтаксическая конструкция с вычислением значений индексов, например:

```
s @"a($h)"=123
```

Конечно, существует возможность принять параметр веб страницы для конструирования полного имени:

```
s @"a("_id_")"=123
```

Но нужно понимать при этом, что в такой конструкции может исполняться произвольный передаваемый извне код, например пусть передано

```
http://server/dir/page?id=$v(FormatDiskCommand)
```

тогда будет выполнена системная функция с нехорошим побочным эффектом.

Если нужно подставить значение параметра в качестве индекса, то нужно использовать значение переменной строго как значение индекса:

```
s a(id)=123
или
s name=$na(a)
...
s @name@(id)=123
```

В таких случаях значение параметра не исполняется как команда и не вычисляется как выражение.

При просмотре кода на предмет скользких мест необходимо также просмотреть его на предмет остальных форм косвенности, включая применение косвенности меток, и сопоставить возможность подстановки в параметры непредусмотренных данных с поведением применяемой MUMPS системы в таких случаях.

Уязвимости такого рода обычно называют уязвимостями класса Code Injection, например SQL Injection или JavaScript Injection. В данном случае это будет MUMPS Injection.

Проблема непрогнозируемого количества запросов вызвана тем, что в среде интернет может произойти наплыв посетителей на страницы сайта по ссылкам с популярных ресурсов и на обслуживание могут понадобиться тысячи процессов сервера. Кроме проблемы физической способности используемого в качестве сервера компьютера выполнять столько процессов одновременно, также стоит вопрос о пределах лицензии на применяемую MUMPS систему если это коммерческая система. При эксплуатации в среде интернет серверов с лицензированием, конечно, необходимо проверить, подходит ли выбранная лицензия или программные решения по ограничению входящих соединений для обслуживания пика запросов.

К третьей проблеме веб приложений относится проблема отсутствия постоянного соединения в том виде, как оно используется в клиент - серверных или терминальных приложениях. Любая страница разработанного веб приложения может быть запрошена в любое время. И, вообще говоря, нет возможности принудить пользователей (а тем более поисковые системы) попадать на страницы приложения строго в определенной последовательности и передавать строго определенные и только предусмотренные параметры.

Передаваемые параметры могут не соответствовать реальным данным в базе данных, и приложение должно относиться к ним как к возможно несуществующим.

В приложениях, где используются авторизация пользователя либо произвольная форма связывания контекста веб приложения с текущим пользователем, необходимо применять дополнительные меры по работе с куками или передавать дополнительные параметры.

И, если веб странице для работы необходимо строго работать только с авторизованным пользователем либо со связанным с ним контекстом, во всех страницах необходимо это предусматривать. Например, перенаправлять на страницу авторизации, либо возвращать содержание по умолчанию.

Очевидно, что и вторая и третья проблема протокола HTTP не относятся напрямую к MUMPS системам, и тут могут быть использованы точно такие же типовые решения проблем, как и при использовании других веб технологий.

6.4.8 Поверх HTTP

На основе протокола HTTP строится множество дополнительных технологий взаимодействия веб клиентов и сервера. В них используется тот факт, что если административно настроен канал взаимодействия по TCP/IP с прохождением HTTP запросов, то поверх HTTP протокола также строится набор дополнений по передаче параметров и получению результата и используется в точности та же самая инфраструктура.

К таким технологиям относятся AJAX, SOAP, JSON, XML-RPC и другие, зачастую не получающие отдельного названия. По сути, все они построены на том, что создается дополнительный объект HTTP запроса, через который передаются параметры и полученное содержание анализируется в зависимости от вида технологии. AJAX и SOAP обмениваются данными в формате XML. Если AJAX передает произвольный XML, то SOAP дополнительно использует определенную схему определения передаваемого XML, выделяя в ответе определенные теги самостоятельно как ответ. В определенной мере SOAP можно рассматривать как один из вариантов, или аналог взаимодействия XML-RPC.

Взаимодействие JSON основано на том, что в качестве ответа сервер возвращает код на JavaScript, который выполняется непосредственно, и по правилам JavaScript автоматически создается объект JavaScript. Вызывающий код уже обращается к его свойствам - атомарным, или массивам, или спискам.

Пример SOAP-запроса на сервер интернет-магазина:

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetails
       xmlns="http://warehouse.example.com/ws">
       <productID>12345</productID>
     </getProductDetails>
```

```
   </soap:Body>
</soap:Envelope>
```

Пример ответа:

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetailsResponse
       xmlns="http://warehouse.example.com/ws">
       <getProductDetailsResult>
         <productID>12345</productID>
         <productName>Стакан граненый</productName>
         <description>Стакан граненый. 250 мл.</description>
         <price>9.95</price>
         <currency>
             <code>840</code>
             <alpha3>USD</alpha3>
             <sign>$</sign>
             <name>US dollar</name>
             <accuracy>2</accuracy>
         </currency>
         <inStock>true</inStock>
       </getProductDetailsResult>
     </getProductDetailsResponse>
   </soap:Body>
 </soap:Envelope>
```

Для протокола SOAP при его разработке были изначально дополнительно определены средства прототипирования и в нем присутствует две отдельные части: 1) URL для получения прототипа функции с перечислением параметров и структуры возвращаемого значения и 2) URL для собственно вызова функции. Наличие прототипов позволяет строить визуальные средства с возможностью выбора функции и с автоматическими генераторами SOAP запросов, включая парсинг ответа.

Пример JSON ответа:

```
{
   "firstName": "Иван",
   "lastName": "Иванов",
   "address": {
       "streetAddress":
          "Московское ш., 101, кв.101",
       "city": "Ленинград",
       "postalCode": 101101
   },
   "phoneNumbers": [
```

```
        "812 123-1234",
        "916 123-4567"
    ]
}
```

Для языка JavaScript такой ответ синтаксически является представлением JavaScript объекта с объявлением и одновременно инициализацией полей. В этом примере поля firstName и lastName специфицированы как атомарные строковые, поле address как также объект со своими полями, и поле phoneNumbers как массив.

В своей технологической основе все эти способы построены на том, что по HTTP передается набор параметров и принимается ответ, но ответ отдается парсеру ответа для выделения его фрагментов. В случае AJAX / SOAP / XML-RPC ответ парсится как XML, а в случае JSON ответ парсится как JavaScript. Что интересно, для обмена данными по таким соглашениям уже не суть важно, какой именно из передающе - принимающих протоколов будет использован - будет это HTTP, или SMTP или другой.

Несложно также построить такой запрос, чтобы не только вебсервер отвечал на языке вебклиента, но и чтобы вебклиент отсылал вебсерверу данные в характерном для него формате. В частности, в SOAP содержание передается как XML. Но вебклиент может также сформировать содержание HTTP запроса и как JavaScript, и на языке MUMPS, например передавать данные в виде

```
data(78)
Berlin~Germany~Europe
```

в теле запроса типа POST или PUT.

Такое содержание на стороне MUMPS процесса может рассматриваться как просто перечисление пар ключ - значение, и расходы на парсинг данных фактически отсутствуют из-за встроенных в язык возможностей косвенности.

Разумеется, при применении такого метода передачи данных сохраняется опасность уязвимости при приеме данных от неавторизованного источника. Если серверный код примет, например, такие данные:

```
a($h)
123
```

и строку с ключем будет использовать как есть, то рискует выполнить системные функции как побочный эффект конструирования имени.

Конечно, для страховки от такой возможности необходимо поставить простейшую проверку на допустимость полученного ключа. Например, применить функции $QS - $QL для проверки синтаксической допустимости имени и не является ли имя глобалом.

Весьма привлекательным выглядит вариант обмена данными в формате, наиболее удобном для принимающей стороны - вебсервер отвечает вебклиенту на языке JavaScript, как в протоколе JSON, а вебклиент отправляет данные вебсерверу на языке MUMPS. В этом случае на принимающей стороне не используются громоздкие сторонние библиотеки парсинга абстрактных форматов данных, а используются встроенные в язык средства косвенности и использования данных динамически, как кода или фрагментов выражений. Теоретически, и для вебклиента на JavaScript, и для вебсервера на MUMPS (или на другой NoSQL СУБД) передача данных в виде просто пар ключ - значение выглядит более практичным форматом, чем громоздкая XML разметка.

При передаче, например, онлайн магазину запроса он мог бы выглядеть например так:

```
function
getProductDetails
productID
12345
```

Или так:

```
function("name")
getProductDetails
param("productID")
12345
```

6.5 Подключаемые DLL (SO)

Большинство современных MUMPS систем, а из промышленно используемых все, допускают подключение и использование внешних динамических библиотек. Это DLL в системах Windows и SO в Linux.

Сама библиотека получается компилирующими трансляторами, обычно это C / C++ / Pascal / ASM. Каждая из MUMPS систем декларирует правила, которым должна соответствовать такая библиотека и какой программный интерфейс она должна предоставить. У каждой из MUMPS систем такие правила собственные и они не входят в стандартные соглашения.

Вызов DLL начинается с вызова специальной или набора специальных расширенных функций MUMPS системы на языке MUMPS, в каждой из систем такой набор собственный, хотя по общей структуре характер вызовов в целом отвечает одним и тем же целям.

На стороне DLL она должна предоставить одну или несколько функций (экспортировать) в соглашениях и в прототипе, объявленном MUMPS системой. При необходимости вызова внешней DLL система определяет, в какой файле находятся необходимые функции, загружает его динамически и отыскивает в экспортированных функциях специальную, которая объявляет наличие и способ вызова внутренних функций этой DLL. Далее отыскивается необходимая, имя которой было указано в MUMPS коде, и она вызывается с передачей ей аргументов и получением возвращаемого значения.

В зависимости от типа применяемой MUMPS системы начальная функция DLL, объявляющая поддерживаемые ей функции, может возвращать как набор указателей на функции, так и набор имен экспортированных функций. Для каждой из таких поддерживаемых внешних функций DLL также должна объявить прототип, количество и, при необходимости, характер передачи аргументов.

Перед вызовом внешней функции MUMPS система преобразует при необходимости передаваемые функции значения из своего внутреннего представления в формат для передачи, формирует стек вызова и передает управление вызываемой функции.

Укрупненно, и в абстрактных обозначениях, это может выглядеть так:

1. MUMPS система выполняет код

   ```
   $zcalldll("filename","funcname",param1,param2)
   ```

2. MUMPS система по параметру filename отыскивает файл DLL и динамически загружает его.

3. В списке экспорта отыскивается оговоренная этой MUMPS системой специальная функция декларирования поддерживаемых этой DLL внешних функций, пусть например это будет функция DECLARED.

4. Для такой функции определяется прототип, согласно которому MUMPS система может получить список внешних функций этой DLL с их прототипами. Пусть для примера эта функция возвращает список определений для функций "transform", "add" и "remove".

5. MUMPS система просматривает этот список объявлений и находит необходимую, например, "transform".

6. По объявленному прототипу функции готовит значения параметров и формирует стек вызова. Значения param1 и param2 переводятся при необходимости из внутреннего представления в передаваемое внешней функции. Это может быть строка завершающаяся нулем, число, структура, или иное.

7. Внешняя функция вызывается и принимается значение возврата. Это значение преобразуется к внутреннему представлению, используемому MUMPS системой и возвращается как возврат функции $zcalldll на уровне исполнения языка.

Обычно предоставляется набор вариантов вызова внешней функции:

1. Загрузить DLL, вызвать функцию, выгрузить DLL.

2. Отдельно операции загрузки DLL, возможность вызова функций у уже загруженной DLL, и операция выгрузки DLL.

3. Загрузка DLL, вызов функции с оставлением DLL загруженной, с отдельной операцией выгрузки DLL.

Вариант с возможностью оставить DLL загруженной между различными вызовами функций в ней дает возможность разработчикам использовать внутренние данные такой DLL, ее собственное состояние, многократно обращаться к ее внутренним объектам.

На основе внешних DLL обычно выполняются модули, написание которых на языке MUMPS очень трудоемко, либо для использования уже написанных модулей, либо для использования специальных API операционных систем, не доступных готовыми для использования в MUMPS вариантами. Также это могут быть функции, для которых критична скорость выполнения, например компрессоры данных, парсеры специальных форматов данных, или интерпретаторы других языков.

К неудобствам таких внешних функций можно отнести то, что в случае смены как MUMPS системы, так и операционной системы такие модули необходимо перекомпилировать под другую MUMPS- или операционную систему (выполнить портирование). К большим плюсам внешних DLL относится то, что в прикладной системе можно использовать произвольный функционал и расход ресурсов на вызов функции в DLL обычно значительно меньше, чем на вызов иных внешних средств.

К интересным возможностям вызова внешних функций в DLL можно отнести не только то, что в таких функциях разработчик ничем не ограничен по возможностям, но и то, что большинство MUMPS систем также предоставляют интерфейс обратного вызова процесса на языке MUMPS из контекста вызванной DLL. При этом разработчик может в таком обратном вызове выполнить набор команд или вычислить значение выражения. При выполнении кода на MUMPS, в свою очередь, также возможен вызов DLL, и так далее. Используя возможности обратного вызова, разработик может, например, вернуть из DLL большое количество данных, записав их в локальную или глобальную переменную или запросить дополнительные значения в зависимости от переданных аргументов.

В практическом применении чаще встречаются внешние DLL с функциями, которых не хватает в самой используемой MUMPS системе с точки зрения разработчиков прикладных систем, либо выполняющие сложные преобразования, либо предоставляющие доступ к специфическому функционалу операционных систем. В частности, это функции кодирования данных в специальных форматах, функции криптографических модулей, модулей регулярных выражений, модулей компрессии данных, дополнительное управление аппаратным обеспечением сервера.

Также отдельно необходимо отметить, что именно внешние DLL используются для интеграции MUMPS систем с другими СУБД, как используя обобщенные драйверы (ODBC, OLEDB, BDE), так и специализированные DLL для определенных СУБД.

6.6 Файлы

Доступ к файлам поддерживается во всех MUMPS системах. Файл во многих системах рассматривается как основное средство передачи данных, как больших, так и малых объемов, как в простых, так и в сложных форматах.

Файл доступен MUMPS процессу как устройство. При открытии файла поддерживаются опции его открытия - нужно ли открыть только имеющийся, или создать новый, какой должен использоваться доступ - текстовый или бинарный, какой должен использоваться терминатор. Операции чтения и записи автоматически смещают текущую позицию в файле на количество прочитанных или записанных байт, при записи после последней позиции файл автоматически увеличивается в размере. И так далее. Не будет ошибкой сказать, что в целом MUMPS системы поддерживают практически все операции с файлами, характерные для

работы с файлами в других программных средах.

Способ открытия файла на чтение или на запись в каждой из используемых MUMPS систем отличается в силу того, что стандарт не специфицирует в точности все необходимые опции. Это может быть открытие файла по номеру устройства с указанием в опции имени файла, или указание в качестве имени устройства самого имени файла, или совмещение в имени устройства и типа устройства и имени файла.

Набор поддерживаемых опций, их имена и значения также могут отличаться в разных MUMPS системах. Поэтому для написания переносимого кода такие операции необходимо выносить в отдельный код.

Несмотря на некоторое расхождение в способе поддержки файлов в виде устройств языка, общие правила у различных MUMPS систем совпадают. Для того, чтобы процесс имел доступ к файлу, используется команда OPEN, чтобы больше не использовать файл - команда CLOSE, чтение выполняется командой READ, запись командой WRITE, а смена опций использования командой USE. Чтобы выполнить портирование кода на иную MUMPS систему, необходимо проверить по документации на нее либо работу с этими командами, либо отдельное описание типа устройства, где указываются поддерживаемые опции команд при работе с файлами.

К особенностям применения файлов в практике можно отнести два пункта: 1) работа в текстовом или бинарном режиме и 2) обработка достижения конца файла.

Для текстового и бинарного режима различаются команды записи

```
write !
```

и чтения строки.

При записи символа перевода строки в Windows-based системах производится вывод последовательности

```
$C(13,10)
```

а в UNIX-like системах выводится последовательность

```
$C(10)
```

При чтении строки система также различает что является символом окончания строки в зависимости от операционной системы. Для UNIX-like систем символ $\$C(13)$, предшествующий символу $\$C(10)$, считается частью строки, хотя он и относится к непечатным, а в Windows-based

системах, если он присутствует, то отбрасывается и в строку не включается. Но это относится только к последнему символу $C(13) - если перед ним есть еще такие же, то они уже включаются в строку.

При работе в текстовом и бинарном режиме (если MUMPS система поддерживает их различение) символ перевода строки рассматривается описанным выше способом, в зависимости от типа операционной системы. В бинарном режиме выводится символ $C(10) вне зависимости от типа операционной системы. Также для бинарного режима необходимо задавать терминатор чтения. Иначе система будет читать строку на максимально доступную длину, например 32 килобайт.

Естественно, что если в файле находится строка с длиной превышающей максимальную длину строки MUMPS системы, то она прочитает строку на доступную длину и последующие чтения строки будут производиться далее, из той же самой строки находящейся в файле. Таким образом, программа получит разбиение исходной длинной строки на несколько строк. Если применяются файлы с такими данными, то разработчикам необходимо учитывать фактор максимальной длины строки. И, возможно, в зависимости от применяемой MUMPS системы.

Для текстового и бинарного режима могут отличаться также поведение команд табулирования

```
write ?NN
```

В этом случае MUMPS система может как вести отсчет текущей позиции в строке и корректно вывести в файл нужное число пробелов для табулированного дополнения, так и не вести такой отсчет, и даже игнорировать такую команду. Для каждой из MUMPS систем при необходимости применения такой команды по отношению к файлу необходимо отдельно проверить, какие операции система выполняет. Кроме того, если выводящий код оперирует системными переменными $X и $Y при работе с текущим устройством, то также необходимо проверить, какие значения возвращаются для файла, как в текстовом, так и в бинарном режиме.

Основной режим доступа к файлам в MUMPS предполагается через модель последовательных устройств - текущий указатель автоматически смещается на количество прочитанных или записанных байт. Но при работе со многими форматами файлов необходимо явно позиционироваться в файле. Для этого практически все системы поддерживают опции команды USE, чтобы задать, как именно необходимо установить текущую позицию в файле. Либо этот же функционал доступен через специальные расширенные системные функции.

К такой же операции, использующей позицию и длину, относится операция явной блокировки участка файла на чтение или на запись. Такие операции уже могут поддерживаться мéньшим числом MUMPS систем, и характерны скорее для файл-серверных систем, чем для клиент-серверных.

Многие MUMPS системы поддерживают отдельную опцию, задающую длину строки для чтения без указания терминатора чтения. С тем, чтобы команда чтения строки всегда выполняла чтение строго указанной порции байт. Таким образом, доступ к файлу из последовательного превращается в некоторым образом блочный. В любом случае, даже если применяемая MUMPS система не поддерживает такой режим, у программистов есть опция команды чтения строки READ, указывающая длину чтения.

Обработка конца файла по умолчанию определена во всех системах как генерация ошибки. В стандарт входит именно такое поведение, и все MUMPS системы его поддерживают, но стандарт не описывает код такой ошибки, и в разных системах содержание ошибки может отличаться.

При выполнении команды чтения (как строки, так и одного символа) MUMPS система проверяет, есть ли в файле еще какие-то байты. Если есть, то выполняет чтение на доступную длину. Например, если программа запрашивает чтение данных на 100 байт, а в файле только 40, то команда вернет только 40 байт. Если же на момент выполнения команды чтения в файле нет доступных байт (предыдущие команды чтения его исчерпали), то команда не возвращает пустую строку (иначе команды чтения зациклились бы), а генерирует ошибку достижения конца файла.

Нужно отметить, что многие современные MUMPS системы поддерживают отдельный режим процесса для обработки конца файла. Используя системную функцию, можно переключить работу процесса так, чтобы явно указать - будет ли генерироваться ошибка достижения конца файла или будет взводиться определенная расширенная системная переменная. Многие MUMPS системы используют для такой переменной имя

```
$ZEOF
```

При переводе процесса в такой режим программа должна реагировать не на генерирующуюся ошибку, а на значение этой системной переменной.

В практическом применении файлы используются для экспорта и импорта данных, рутин, как в стандартных для MUMPS систем форматах, так и в сторонних форматах (XML, DBF, INI, CSV, и другие), а

также для подготовки временных данных в виде файлов для внешних трансляторов (LaTeX, XSL, языки программирования, и другие) и для использования результатов работы внешних процессов. Во втором случае разработчики применяют специальные библиотеки обработки таких специальных форматов файлов и приведения их кодировок к внутренней текущей кодировке MUMPS системы.

6.7 Внешние процессы

Обращение к внешним процессам - это возможность, относящаяся не столько к функционалу собственно СУБД, сколько к задаче интеграции с другими компонентами или организация взаимодействия с уже готовыми подсистемами.

Внешний процесс запускается для выполнения определенной задачи. Ему могут быть переданы параметры, получены результаты и вариантов организации взаимодействия может быть множество, в зависимости от исполнения запускаемой программы.

К вариантам запуска внешнего процесса могут быть отнесены следующие:

1. Запустить процесс и не дожидаться окончания его работы.
2. Запустить процесс и дождаться окончания его работы.
3. Запустить процесс и взаимодействовать с его стандартными каналами ввода - вывода как с устройством MUMPS.

При этом все три варианта допускают передачу процессу параметров командной строки, а также возможность запуска тенологически различных процессов - exe, скриптов, командных файлов.

Первые две возможности обычно выполняются в MUMPS системах как системные расширенные функции.

Третья возможность более интересная. Запускаемый процесс принимает аргументы командной строки и читает свой стандартный входной канал stdin, а результат пишет в свой стандартный выходной канал stdout. Для MUMPS процесса открытие внешнего процесса как устройства означает запуск внешнего процесса с организацией взаимодействия с его каналами. Запускающий MUMPS процесс пишет в его канал stdin, а читает из его канала stdout.

Приведем пример такого устройства, запускающего встроенную в операционную систему команду DIR, и читающий результат.

Для системы Caché это может быть такой вариант:

```
cmd
 s dev="dir c:\ /w /b"
 o dev:("QR")
 u dev
 s $zt="end"
 f  r line u $p w line,! q:line=""  u dev
end
 c dev
 q
```

Здесь ключевой элемент состоит в том, что устройство открывается с опцией Q, поэтому читается не из файла dir, а выходной канал процесса, запущенного внешней программой dir.

Для системы MiniM это может быть такой вариант:

```
dir ;
 n zeof=$v("proc",5,0)
 n dev="|PIPE|dir /w"
 o dev:("rwt")
 n result
 u dev f  q:$zeof  r result($i(result))
 u $p
 c dev
 i $v("proc",5,zeof)
 zw result
 q
```

Здесь ключевой элемент состоит в том, что используется устройство типа |PIPE|, что и означает запуск внешнего процесса с организацией чтения его выходного канала.

В системе GT.M это может быть такой вариант:

```
Set dev="MyProcs"
OPEN dev:(COMMAND="ls":READONLY)::"PIPE"
```

Здесь ключевой элемент состоит в том, что устройству указывается область обработки "PIPE" и отдельно в опции устройства указывается выполняемая команда.

Таким образом, используя запуск внешнего процесса, разработчики могут подключать к разрабатываемой прикладной системе множество уже готовых программ.

К плюсам выполнения задачи внешним процессом относится его готовность её выполнить и разработка запускаемой программы независимо от контекста её использования. В частности, трансляторы языков программирования работают независимо от того, из какой программы

их вызвали. Ко второму плюсу относится использование запущенным процессом собственного адресного пространства, изолированного от запускающего. К третьему плюсу можно отнести возможность запускать процесс, работающий в другой архитектуре, например из 64-х битной MUMPS системы запускать и использовать процессы, выполненные как 32-х битные и наоборот, а также независимость от того, что внешний процесс может быть как самостоятельным и запущенным из исполняемого файла, так и интерпретатором скриптового языка.

К минусам использования внешних процессов можно отнести возможно увеличенное время их запуска по сравнению с обращением к такому же коду, выполненному в виде подключаемой DLL, и необходимость специальным образом готовить параметры и принимать результаты работы. Возможно, может понадобиться специальный небольшой парсер для разбора результата работы программы.

При использовании как внешних подключаемых DLL, так и внешних процессов в некоторых MUMPS системах может присутствовать приостанов работы остальных JOB, если эта MUMPS система выполнена на собственном механизме переключения контекстов исполняющихся JOB. При передаче управления внешним средствам из контекста, контролируемого таким интерпретатором, управление ему не возвращается, пока запущенная функция не отработает полностью до возврата или до возврата следующей порции.

Такие устаревшие и немасштабируемые MUMPS системы уже не развиваются и сняты с производства, хотя в эксплуатации еще могут находиться сервера под их управлением. Первоначально такие системы были разработаны для работы под управлением операционных систем без встроенной многозадачности, например класса DOS, и унаследовали определенные архитектурные решения.

6.8 Порты

Под применением портов в отношении MUMPS систем понимают в широком смысле и последовательные и параллельные порты, а в узком смысле практически всегда последовательные (а в еще более узком - RS-232).

Первоначально, когда еще толком не было разработано сетевых соединений компьютеров с полноценной сетевой инфраструктурой, к компьютеру подключались терминалы. Это монитор с клавиатурой и приемопередающим блоком, а связь выполняется кабелем по последовательному каналу.

Характеристики связи были таковы, что именно последовательные каналы связи с хорошими токами позволяли связать терминалы с самим компьютером на довольно большие расстояния. Объем передачи был небольшой и пропускной способности таких каналов вполне хватало.

В настоящее время такие подключения практически не встречаются, терминалы практически не используются, их заменили полноценные персональные компьютеры с полноценной сетевой инфраструктурой, поэтому первоначальное назначение последовательных портов как основного коммуникационного средства ушло, но осталось применение последовательных портов для другого оборудования.

При появлении полноценных сетевых протоколов TCP/IP на смену железным терминалам пришли телнет - клиенты, эмулирующие их работу и совместимые с передаваемыми форматами за тем исключением, что данные передаются по каналу TCP/IP, а не по последовательному каналу, и телнет клиент это не отдельная аппаратная конструкция, а программа, которая может быть запущена на компьютере во множестве экземпляров и для соединения с различными серверами по одному и тому же сетевому подключению.

Самые различные устройства могут быть подключены к компьютеру по последовательному порту - модемы, контрольно - кассовые машины, считыватели штрихкодов, денежные ящики, приемники купюр.

Последовательные порты также могут быть использованы для оборудования, не работающего непосредственно через COM порты, в виде виртуальных последовательных портов. В этом случае устанавливается специальный драйвер виртуального порта. Также могут использоваться аппаратные переходники с USB оборудования на COM порты.

Общий принцип работы с оборудованием по последовательному порту состоит в том, что используется устройство для COM порта. Устройству передаются опции открытия в команде OPEN или, при необходимости, изменяются опциями команды USE, и чтение - запись выполняются командами READ - WRITE. При этом MUMPS система направляет все указания устройству соответствующему аппаратному последовательному порту.

Для организации обмена с контрольно - кассовой аппаратурой нужно записывать и читать из порта байты, описанные в протоколе обмена на используемую аппаратуру. Обычно он выглядит примерно таким образом:

1. Записать байт начала команды.

2. Записать байт кода команды.

3. Записать байт конца команды.

4. Записать 2 байта контрольной суммы.

После записи последовательности команд нужно прочитать из того же устройства ответ о выполнении как определенную последовательность байт. В ней содержатся в указанном в протоколе кодировании данные с ответом.

Различные ККМ используют различные протоколы, но общие принципы полностью аналогичны описанному - это определенная последовательность байт. Для корректной работы аппаратуры она также может иметь особенности по настройке параметров порта для передачи - наличие стоповых бит, число бит в байте, паритетность, скорость передачи, интервалы задержек. Эти параметры устанавливаются или в опциях команды OPEN, или в опциях команды USE, или вызовом отдельных системных функций.

Несмотря на то, что последовательные терминалы в настоящее время считаются устаревшим оборудованием, тем не менее такое оборудование до сих пор находится в эксплуатации как чрезвычайно надежное и простое. Для подключения такой аппаратуры к MUMPS серверам используются различного рода мультиплексоры, создающие необходимое для аппаратного подключения число последовательных портов, и в MUMPS системе запускаются процессы, обслуживающие обмен данными по последовательным портам.

В настоящее время техническая поддержка такого последовательного оборудования, подключение различного рода железных терминалов, является отдельной и непростой задачей в силу очень малой распространенности такого оборудования, и в настоящее время под терминалами с последовательными каналами специалисты скорее будут иметь в виду различного рода платежные терминалы, внутри которых находится компьютер с набором последовательных портов и ККМ - оборудование стойки, подключенное через них.

Отдельно нужно сказать о работе с принтерами. Работа с принтерами может быть выполнена как вывод текста на тексто - ориентированное устройство и как вывод PCL или PostScript команд на графически - ориентированное устройство. Традиционно под работой с принтерами по портам подразумевается первый вариант, когда программа на MUMPS просто выводит текст в устройство, связанное с принтером. При этом принтеры могут быть аппаратно выполнены как соединяемые и через последовательный, и через параллельный порт.

В настоящее время современные MUMPS системы могут поддерживать специальные устройства для работы с принтерами с поддержкой

специализированных для принтеров опций - управление ориентацией страницы, качеством печати, ее цветностью и другими. Тем не менее, если принтер работает через обычный последовательный или параллельный порт, его можно использовать открыв программно соответствующий порт и посылая необходимые управляющие последовательности. Для каждого из принтеров, по крайней мере, выпускавшихся раньше, можно найти описание команд принтера для вывода на печать и текста и графических примитивов.

В некоторой степени исторический факт, что именно такая технология использования новых моделей принтеров, без использования драйверов операционной системы, и применялась в различных прикладных MUMPS системах. При необходимости подключения новой модели и для графической печати для нее описывались команды вывода примитивов, посылающие последовательности байт в порт, или в некотором роде драйвер принтера, написанный на языке MUMPS. По сути, вывод на принтер производится либо выдачей текста, либо команд языка PCL или PostScript. Для вывода же просто текстовых документов вполне достаточно обычных текстовых управляющих символов - перевод строки, табулятор, перевод страницы.

Глава 7

Практика применения

7.1 Терминальный режим

Название терминальный происходит от терминалов, это такие полукомпьютеры, имеющие монитор, канал связи, клавиатуру, и простейшие средства сопряжения по линии связи, например по COM портам.

Такие клиентские места подключались к компьютеру, который стоял в отдельном помещении, или в большой стойке, или даже в нескольких. Терминал отсылал компьютеру нажатие клавиш, а компьютер отвечал командами что сделать с видеопамятью. В результате на экране отображались алфавитно-цифровые символы.

Обычно такие терминалы выполнялись на монохромной электронно-лучевой трубке с зеленым люминофором. Но были терминалы и другого физического исполнения, вместо экрана могли использоваться даже печатающие машинки. Общее название терминального интерфейса так и было закреплено.

В настоящее время терминальные устройства почти не используются, их заменили программные средства, работающие на множестве операционных систем. При этом полностью сохранились принципы взаимодействия - отсылка нажатия клавиши от клиентской части серверу и отсылка команды изменения экрана от сервера к клиентской части.

В настоящее время также можно встретить термин CHUI - CHaracter User Interface как альтернатива названию GUI - Graphical User Interface. В практически любой операционной системе присутствуют такие программы, в частности это обычные консоли, DOS - окна, графические телнет клиенты, имитирующие алфавитно-цифровой интерфейс.

В таких программах взаимодействие с компьютером выполнено в виде вывода символов, и, зачастую, с применением псевдографики, и ввода с

клавиатуры. Терминальный интерфейс, по оценкам экспертов, позволяет строить программы с наиболее оптимальным для ввода данных интерфейсом. В частности, при работе со счетами в банках, при вводе чека в супермаркетах, при выписке билетов многие разработчики стараются применить именно алфавитно-цифровой режим, как оптимальный для клавиатурного ввода с большой скоростью.

Такие же программы можно выполнять на языке MUMPS, поскольку клиентской части в общем-то все равно, какими программными средствами принимается символ и отдается команда изменения изображения. Точно так же как на других языках, на языке MUMPS можно формировать символами псевдографики оконный интерфейс, поля ввода, кнопки, и многое другое.

Современные MUMPS системы поддерживают терминально - ориентированные средства взаимодействия как минимум, зачастую с несколькими вариантами и способами физического соединения и исполнения.

В настоящее время все меньше MUMPS систем поддерживает ставшие физически экзотическими средства вроде COM портов на мультиплексорах, или LAT интерфейс. При этом как минимум поддерживаются консольные и телнет-интерфейсы.

Консольным интерфейсом называется окно, принадлежащее непосредственно запущенному процессу, когда MUMPS процесс физически выводит в это окно и читает ввод с клавиатуры компьютера. Телнет интерфейсом называется программное взаимодействие MUMPS процесса с телнет клиентом. Телнет клиент это специальная программа, отображающая окно и программно взаимодействующая с серверной частью по сетевому протоколу поверх TCP/IP.

Запуск консольного варианта (если он поддерживается) выполняется или непосредственным запуском исполняемого файла с, возможно, указанием необходимых параметров. Для запуска телнет клиента необходимо запустить саму программу и указать опции соединения - имя компьютера и порт который обслуживается телнет-сервером MUMPS системы. Например, запустив в Windows cmd.exe, в его окне набрать:

```
telnet localhost 23
```

Здесь localhost - это имя компьютера (локальный), 23 - номер обслуживаемого порта. При успешном соединении телнет клиент может вводить команды и отображать результат работы.

В качестве телнет клиентов могут быть использованы произвольные телнет клиенты, это могут быть как программы непосредственно консольного исполнения, так и графические, показывающие окно эмуляции.

Многие MUMPS системы предоставляют уже настроенное инсталлятором меню запуска терминального входа в MUMPS систему.

Собственно, сам запуск такой программы, соответствующей входу в MUMPS систему, и является обычным и ежедневным делом для тех, кто работает с MUMPS системами, и может вызвать затруднение у разработчиков, привыкших к только графическим средствам разработки. При использовании MUMPS системы первое, что необходимо найти, это способ входа в нее в зависимости от версии, от используемой операционной системы. Далее, в силу традиций и совместимости различных MUMPS систем, все может выглядеть совершенно одинаково на совершенно разных реализациях.

Обычно слева в начале строки MUMPS система обычно пишет промпт и сообщает в нем где находится текущий процесс, в какой из баз данных. После него ожидает ввода команд. При этом можно, нажав стрелки вверх / вниз, выбрать предыдущий ввод. После ввода строки команд разработчик нажимает Enter и система выполняет введенные команды.

Протокол для телнет клиентов является открытым и данные передаются по сети как есть. Кроме него существуют возможности обратиться к MUMPS системе по закрытому каналу, используя протокол SSH, и различные SSH клиенты. По сути это примерно те же телнет клиенты, но использующие шифрование для закрытия канала обмена данными. Для настройки SSH канала необходимо либо обратиться к руководству на используемую MUMPS систему, либо настроить SSH шлюз внешними средствами.

Обычно окно терминального интерфейса имеет стандартные размеры 80 по горизонтали на 25 по вертикали символов. При этом многие системы имеют возможность увеличить размер по вертикали, в том числе с прокруткой.

Кроме символов псевдографики для оформления рамок и элементов интерфейса CHUI программы могут использовать также цвета. Например, пакет программ AltNC может работать с рутинами и глобалами различных MUMPS систем в стиле двухпанельного интерфейса Norton Commander, как в консоли так и в телнете, с раскраской панелей и подсветкой синтаксиса рутин на языке MUMPS.

7.2 Редакторы рутин

Вторым основным средством для разработчиков на MUMPS являются редакторы рутин. Рутина представляется в виде текста, как обычный текстовый файл. Редактирование такого текста может выполняться как

отдельной графической программой с полновесным MDI интерфейсом, так и редатором написанным на самом MUMPS и редактирующим программу в терминальном окне.

Для практически всех современных MUMPS систем существуют средства редактирования текста, либо устанавливаемые в составе MUMPS системы, либо устанавливаемые и настраиваемые отдельно.

Для систем Caché и MiniM в комплект MUMPS системы входят графические MDI редакторы текста, для GT.M, Caché, DSM, M21 и MSM возможно использование внешних средств Serenji, а также для большинства MUMPS систем может быть использован редактор рутин, входящий в консольный AltNC.

Если MUMPS система поддерживает хранение исходных текстов рутин в виде файлов файловой системы, то можно использовать произвольный текстовый редактор.

В большинстве случаев редакторы рутин поддерживают редактирование с подсветкой синтаксиса языка MUMPS и, возможно, с учетом специфики MUMPS системы, ее дополнительных возможностей синтаксиса.

Кроме просто редактирования рутин редакторы также поддерживают обращение к компилятору рутин и показ результата трансляции с возможными ошибками синтаксиса.

Редакторы рутин в большинстве случаев соединяются с MUMPS системой по протоколу выполненному поверх TCP/IP и могут работать как с сервером запущенным на том же компьютере, так и с любым другим, если к нему есть доступ по TCP/IP.

Зачастую редакторы рутин также содержат функцию экспорта и импорта рутин. При этом нужно понимать, на каком из компьютеров будет сохраняться файл экспорта - на локальном, где работает редактор, или на сервере, где работает MUMPS система.

Для организации работы нескольких разработчиков над одним проектом могут применяться самые различные способы. Необходимо учитывать, что исходный текст рутин не является файлами как таковыми, чтобы использовать готовые средства контроля версий и, в зависимости от принятых в компании - разработчике программного обеспечения внутренних правил это могут быть, например, такие:

1. Разработчики работают в одной базе данных и редактируют только рутины, закрепленные за ними, либо те рутины, которые определены руководителем.

2. Разработчики работают в различных базах данных, в том числе и на различных серверах и проводят экспорт и импорт измененных

рутин в виде файлов и согласуют файлы через систему контроля версий самостоятельно.

3. Разработчики работают в различных базах данных и используют специальные средства сопряжения рутин и файлов с системами контроля версий.

Одним из очень практичных свойств MUMPS систем является то, что рутины могут быть изменены и компилированы независимо друг от друга, в том числе в эксплуатирующейся базе данных. При этом приложение, их не использующее во время редактирования и компиляции, продолжает корректно работать. Из-за того, что рутины используются только по необходимости, возможна работа с огромными объемами программного кода, и, одновременно с тем, отсутствуют сложные процедуры сборки приложения в один исполняемый файл.

MUMPS системы, хотя это и не определено стандартом явно, поддерживают выполнение рутин в режиме позднего связывания. В этом режиме выполнения кода переход к нужной строке выполняется как есть, и проверка существования вызываемой строки проверяется только при исполнении. При трансляции программ и исполняемых строк существование вызываемых строк или их корректность не требуются. Их можно написать, импортировать или отредактировать позже, чем редактирование и компиляцию вызывающего кода. В системах исполнения позднего связывания не используется линкер или построитель связей.

При разработке большой системы группа разработки выбирает одну из моделей разработки и далее ей следует. В большинстве случаев используется редактирование рутин в отладочных базах данных (в общей или индивидуальной), перенос рутин в тестовую базу для проверки (также может быть общей или индивидуальной) и передача готового комплекта рутин в рабочую базу. При необходимости вместе с исходными текстами рутин могут передаваться файлы экспорта глобалов и инструкции по выполнению кода настройки после импорта рутин.

В качестве средств контроля версий обычно используются традиционные файловые средства и разработчики выполняют синхронизацию системы контроля версий с нужным каталогом и выполняют импорт и экспорт рутин в этот каталог.

7.3 Экспорт и импорт

Экспорт и импорт рутин и данных для MUMPS систем являются основным средством переноса рутин и данных между различными серверами.

Одним из ключевых отличий MUMPS систем от других систем баз данных является то, что стандарты применения MUMPS систем предусматривают стандартные форматы переноса рутин и данных, и они поддерживаются практически всеми реализациями MUMPS систем. Кроме стандартных форматов MUMPS реализации зачастую поддерживают дополнительные собственные расширенные форматы файлов экспорта и импорта.

Для рутин есть, по сути, один стандартный формат экспорта, с вариациями на усмотрение реализаций. Файл экспорта рутин состоит из последовательности следующих элементов:

1. Заголовок экспорта.
2. Последовательность рутин.
3. Пустая строка как индикатор окончания экспорта.

Последовательность рутин хранится как простая последовательность следующих строк:

1. Строка с именем рутины.
2. Последовательность строк рутины как есть.
3. Пустая строка как индикатор окончания рутины.

Для того, чтобы отличить пустую строку рутины и индикатор окончания, при экспорте пустой строки рутины экспортируется строка из одного пробела или одной табуляции. Поэтому, если в рутине разработчик поставил пустую строку, экспортировал рутину, потом импортировал, то эта пустая строка будет заменена на строку из пробельной последовательности.

Вторым нюансом, в котором различные реализации MUMPS систем могут проявить особенности, является заголовок экспорта. Это две текстовые строки, в которых программа экспорта пишет служебную информацию о проведенном экспорте. Очередность и формат этого заголовка не фиксированы, и различные MUMPS системы могут писать заголовки с различном порядке. В любом случае, для импорта рутин эти обе строки заголовка используются лишь в информационных целях.

Третьим нюансом является возможность размещения в первых двух строках заголовка автоимпорта. Это последовательность команд MUMPS,

выполняющих импорт оставшейся части файла импорта. Заголовок автоимпорта является специфическим для каждой из MUMPS систем, поскольку используется информация о деталях хранения рутин, а в каждой из MUMPS систем они могут отличаться, кроме того в каждой из MUMPS систем используются различные способы компиляции рутин. Заголовок автоимпорта используется для передачи файла с экспортированными рутинами MUMPS системе как последовательности команд для исполнения, например:

```
mumps.exe < routines.rtn
```

При этом процесс считывает команды для выполнения из переданного файла. Эти команды уже самостоятельно продолжают последующее чтение и выполняют импорт. Реальное имя исполняемого файла и необходимые дополнительные опции командной строки нужно определить по документации на используемую MUMPS систему или обратиться в ее техническую поддержку.

Вариант заголовка автоимпорта, используемый системой Caché:

```
N IO,D,P,I,A,X,L S IO=$I R D U 0 W !,D,! S P="^"   >>
  F  U IO R A Q:A=""  I $P(A,P,3) S L=$P(A,P,5)    >>
  S:L X=$ZU(55,L) ZL  ZS @$P(A,P) S:L X=$ZU(55,0)  >>
  U 0 W !,$P(A,P,1,2),?20," loaded" ;(Self-loading)
%RO on 05 Oct 2012   2:27 PM
ETRAP^INT^1^62735,51942^0^1
ETRAP ; etrap examples
 q
 ...
```

Здесь символами » обозначено продолжение одной строки. При выполнении такого файла экспорта система Caché выполняет команды импорта рутин из того же самого файла.

Вариант заголовка автоимпорта, используемый системой MiniM:

```
n h,r,l r h f  r r q:r=""  s h=$p(r,"^",4)          >>
  s h=$s(h:h,1:$h) s r=$p(r,"^") f  r l             >>
  i l="" q:l=""                                     >>
  s ^ROUTINE(r,$i(^ROUTINE(r)))=l s ^ROUTINE(r,0)=h
14:31 5-окт-2012 MiniM Routine Editor export
ETRAP
ETRAP ; etrap examples
 q
 ...
```

Здесь символами » также обозначено продолжение одной строки. При выполнении такого файла экспорта система MiniM выполняет команды импорта рутин из того же самого файла.

Вариант заголовка автоимпорта, используемый системой MUMPSV1:

```
S %I=$I R B X B U 0 C %I D ^%C K ^$R("%C")  >>
  W !,"Done",! Q
F  R R Q:R=""  U 0 W R,! U %I K A F I=1:1   >>
  R A(I) I A(I)="" M ^$R(R)=A Q
ETRAP
ETRAP ; etrap examples
 q
 ...
```

Здесь символами » также обозначено продолжение одной строки. При выполнении такого файла экспорта система MUMPSV1 выполняет команды импорта рутин из того же самого файла.

В принципе, эта же методика может быть использована для размещения в этом же файле не только рутин, но и глобалов, и иных выполняемых действий. В частности, после того, как процесс MUMPS выполнит команды, он продолжит считывание входных команд. Если далее снова встретит команды для выполнения, то снова их будет выполнять. Таким образом, можно в одном файле переносить множество различных фрагментов. При размещении в одном файле экспорта лишь одной последовательности рутин с одним заголовком автоимпорта после выполнения команд процесс MUMPS считает в качестве дальнейших команд пустую строку, поэтому закончит работу.

Заголовок автоимпорта традиционно составляется так, чтобы занимать только первые две строки для того, чтобы рутины могли быть импортированы традиционными средствами. В этом случае в качестве служебных данных экспорта системы считают и покажут последовательности команд, но на содержание импортируемых рутин это не оказывает никакого влияния. В том случае, если разработчики используют механизм автоимпорта для собственных целей, не предназначенных для стандартных средств импорта рутин или глобалов, то могут быть использованы произвольное число строк в начале файла и произвольные операции.

У механизма автоимпорта есть еще один нюанс, касающийся ошибок импорта. При передаче команд исполнения через стандартный вход команды выполняются в контексте текущего устройства с разделенными каналами входа и выхода (stdin + stdout), поэтому, если при импорте или компиляции импортированной рутины произойдет ошибка и будет производиться вывод диагностики в текущее устройство, то вывод будет

направлен в канал вывода, а не в канал ввода, и файл экспорта не будет поврежден операциями записи. Ошибки при импорте могут произойти даже при записи очередной строки рутины, например, при недостатке места в базе данных. Большинство современных MUMPS систем принимают меры к тому, чтобы при импорте не допускать порчи исходного файла диагностическими сообщениями об ошибках в случае их происхождения, не утрачивать диагностические сообщения и продолжать импорт до его полного окончания с получением полного отчета.

Имена файлов с экспортированными рутинами могут иметь любое расширение файла, зачастую разработчики используют один из нескольких: R, ROU, RTN, M, RSA (Routine Save Archive) или другие, читаемые как "это файл, содержащий рутины".

Распространенной ошибкой является использование форматов экспорта рутин для хранения в репозиториях систем контроля версий. Причин две:

1. Формат экспорта содержит не одну целостную сущность редактирования, а несколько.

2. Формат экспорта содержит дополнительную информацию, не входящую в сущность, редактируемую разработчиками.

Первая причина приводит к тому, что в файле могут оказаться не одна сущность редактирования, а несколько, и их состав, вообще говоря, может быть произвольным и определяется каждый раз при экспорте. Кроме того, порядок экспорта рутин, а значит и построчное содержание такого файла экспорта, в общем случае, не детерминированы, поскольку средства экспорта не гарантируют порядок следования экспортируемых сущностей. В большинстве случаев они следуют в лексикографической сортировке имен сущностей, но это не гарантируется.

Вторая причина приводит к тому, что утилита экспорта автоматически формирует первые две строки с добавлением служебной информации, и в нее могут входить дата, время, версия, название программы экспорта, и другие данные, отличающиеся как на разных серверах, так и в разное время экспорта.

Обе проблемы приводят к тому, что для систем контроля версий такие файлы вызывают проблемы коллизий изменений строк, не относящиеся к разработке.

Для задачи сопряжения с системами контроля версия наиболее подходит формат хранения рутин в файлах, используемый в системе GT.M. Он решает обе проблемы. Система хранит рутины в виде файлов файловой системы как есть. В одном файле хранится одна рутина, и в файле

не хранится никакая другая автоматически добавляемая информация, которую не редактирует разработчик.

У этого формата имеется лишь один недостаток - имя рутины совпадает с именем файла. При использовании файловых систем, не различающих регистр символов в именах файлов (например, в Windows), это приводит к невозможности хранить две рутины, отличающиеся в имени регистром символов, и к неопределенности регистра символа для импорта. При преодолении проблемы регистра, например, при соглашении использовать лишь верхний, или при специальном декорировании имен, такой формат, тем не менее, наиболее удобен для систем контроля версий по содержанию файлов.

MUMPS системы, поддерживающие препроцессор, также дополнительно различают типы рутин - непосредственный код MUMPS языка, получаемый после препроцессирования, макрокод рутин с директивами препроцессора, и включаемые рутины с макрокодом. При необходимости экспортировать рутины с разным типом применяется формат RSA, в котором в качестве имени рутины хранится не только имя, но также дополнительно расширение, указывающее на тип рутины, и другая дополнительная информация, например время последней модификации этой рутины. Таким образом, макрорутины не могут быть перенесены в MUMPS систему, не поддерживающую препроцессор и такой специальный формат экспорта с сохранением типа рутины.

Данные глобалов, так же как и рутины, экспортируются и импортируются через стандартные форматы файлов. Данные могут быть перенесены между различными MUMPS системами, разных версий, работающими на разных операционных системах и от разных производителей.

Принцип организации форматов экспорта глобалов тот же, что и для рутин. В файле записывается заголовок экспорта, и перечисляются имена глобалов и их значения.

Так же как для файлов экспорта рутин, для файлов экспорта глобалов могут быть применены заголовки автоимпорта и такие файлы могут быть использованы в пакетных операциях.

В отличие от рутин, глобалы могут содержать произвольные символы, в том числе символы используемые текстовыми файлами как символы окончания строки. Поэтому для глобалов различают два формата экспорта - потоковый и переменной длины.

Потоковый формат представляет собой простой текстовый формат, как последовательность пар строк - первая из них содержит полное имя глобала с индексами, вторая полное значение. Поскольку потоковый формат рассматривается как текстовый, его нельзя использовать для переноса тех данных, в индексах или в значениях которых могут

быть нетекстовые байты.

Формат переменной длины для кодирования пар имя - значение использует для каждой записи специальный маркер длины. В этом случае и индексы и данные глобалов могут содержать произвольные байты, поскольку количество байт для использования в качестве имени и данных определяется маркерами.

В отношении глобалов, в отличие от рутин, неприменимо понятие целостности сущности. Если рутина как совокупность строк после выполнения импорта целиком принимает новое значение, то в отношении глобалов такое соглашение действует только для каждой из перечисленных пар имя - значение. При импорте глобалов действуют правила:

1. Имеющиеся записи в глобалах не удаляются, независимо от совпадения значений индексов, в том числе при частичном их совпадении.

2. Если имеется запись с тем же именем (полное совпадение значений индексов), то эта запись принимает указанное значение независимо от того, существовала ли она ранее, и от того, какое имела значение.

Таким образом, при импорте данных из файла экспорта глобалов действуют правила, используемые командой merge. Если для операции переноса данных необходимо обеспечить взаимную целостность между различными записями в глобалах, то перед импортом необходимо самостоятельно удалить имеющиеся в этих глобалах записи, чтобы их совокупность после импорта приняла строго то же значение, и чтобы они не содержали иных записей, бывших ранее и не имеющихся в файле экспорта.

Автоматические средства ни одной из MUMPS систем такого предварительного удаления данных не выполняют.

Организация файлов экспорта глобалов технически допускает их использование в системах контроля версий, но разработчики очень редко прибегают к такому способу. Согласно общему принципу разработки, в системах контроля версий хранится только то, что пишут программисты. Поэтому, если необходимо в разрабатываемой системе иметь кроме рутин также и данные, то разработчики помещают эти данные в рутины и программный код получает их через функцию $TEXT.

В среде MUMPS разработчиков многие годы также действовало правило хранить в глобалах, подлежащих переносу через экспорт, только текстовые символы, как в значениях индексов, так и в значениях записей. При этом также налагалось ограничение на длины и индексов и

данных. Такой набор соглашений, при их соблюдении, позволял свободно переносить глобалы между произвольными MUMPS системами. И все данные, экспортированные в соответствии с требованиями стандарта переносимости, и сейчас могут быть импортированы любой MUMPS системой.

Что интересно в применении файлов экспорта глобалов, это то, что в такой формат могут быть экспортированы данные из самых различных источников данных и впоследствии импортированы в MUMPS систему. Разработчик, описывающий такой экспорт из не-MUMPS системы, должен лишь определить соответствие пар ключ-значение и соблюсти синтаксис языка MUMPS для кодирования индексов и значений.

Для разработки прикладных программных систем зачастую используются также сущности, не являющиеся рутинами и данными глобалов, но рассматриваемые как целостный элемент. К таким относятся, например, определения экранных форм, отчетов, классов. Хотя вся информация хранится в виде глобалов, к таким сущностям зачастую применяются специализированые средства редактирования и их экспорт и импорт также должны выполняться соответствующим способом, как целостной сущности.

При импорте рутин и данных важным моментом эксплуатации может оказаться, в зависимости от предъявляемых требований к системе, режим транзакционности при импорте. При импорте как рутин так и глобалов MUMPS системы, даже если поддерживают транзакционность, ее не применяют. В случае неудачи импорта, например, при замещении рутины или при ошибке ее компиляции, в базе данных останется состояние рутины на момент ошибки. Такой вариант используется по умолчанию всеми MUMPS системами в целях совместимости поведения. В случае если разработчикам необходимо выполнять импорт с возможностью восстановления состояния рутин и данных на момент начала импорта если произошла ошибка, то такие средства импорта необходимо составить самостоятельно и применить стратегию по правилам, применяемым прикладной системой.

Другим нюансом является возможность административно указать, какие глобалы или базы данных не журналируются и при откате транзакции импорта сделанные изменения не будут возвращены. Некоторые MUMPS системы и, возможно, в зависимости от версии, могут применять такие соглашения, в частности, к глобалам хранения компилированного байткода. В таких системах, если разработчик выполняет импорт с компиляцией в транзакции, то при ошибке компиляции и откате транзакции исходный код рутин может быть возвращен в предыдущее состояние, а компилированный байткод нет, или наоборот.

Кроме того, часть MUMPS систем применяет хранение рутин не в глобалах, а, например, во внешних файлах или в блоках базы данных специального типа. В этом случае при применении импорта в контексте транзакции надо проверить поведение системы при выполнении команды trollback, и как именно система выполняет откат изменений сделанных при импорте рутин и при компиляции в байткод. Кроме того, возможность выполнить откат сделанных изменений может определяться особенностями поведения и ограничениями журнала для команды kill, выполняющей предварительное удаление импортируемых сущностей. Если поведение команды kill несовместимо с предъявляемыми требованиями по откату больших изменений, необходимо заменить одну команду kill на соответствующую серию так, чтобы выполняемые удаления могли быть отменены командой trollback.

Конечно, если необходимо обеспечить целостность импорта на используемой MUMPS системе, не имеющей поддержки транзакционности, то необходимо самостоятельно принять меры к сохранению информации о состоянии до импорта, о выполняемых при импорте изменениях и, при необходимости, программно отменить выполняемые изменения и вернуть состояние импортируемых сущностей (глобалы, рутины, формы, отчеты и т.д.) на начало импорта.

Хотя большинство прикладных разработок для MUMPS систем не столь критичны к целостности импорта, в случае наличия особых требований к системе по качеству разработчикам необходимо детально проверить все нюансы импорта с обеспечением целостности. К отличительным качествам MUMPS систем относится то, что на них возможно выполнение прикладных систем удовлетворяющих самым высоким критериям и разработчики имеют возможность определять и контролировать каждое выполняемое системой действие.

Кроме стандартных форматов экспорта глобалов как последовательности пар ключ - значение часть MUMPS систем поддерживают блочный экспорт глобалов. В основе такого экспорта лежит факт, что глобалы хранятся в виде блоков диска и большинство MUMPS систем выполняет хранение данных по схеме B*-tree. В такой схеме глобал состоит из дерева блоков, в котором есть два типа блоков - блоки ссылок и листовые, или конечные блоки.

Блоки ссылок содержат значения индексов, разделяющие области поиска и указывающие на дочерние блоки - они могут быть как блоками ссылок так и листовыми блоками. В результате изменений базы данных вполне может оказаться, что ключи хранящиеся в блоках ссылок даже не соответствуют реально существующим данным. Их задача в том, чтобы разделить области поиска. Листовые блоки содержат последова-

тельности пар ключ - значение.

Если абстрагироваться от реального формата кодирования такого блока, то, по сути, занимаемое им пространство это уже и есть готовая последовательность хранения данных. Поэтому часть MUMPS систем использует прямой экспорт таких блоков. Вместо многократных проходов за каждой парой ключ - значение при таком способе система просто рекурсивно проходит дерево глобала не как логическое дерево пар ключ - значение, а как физическое дерево блоков, и экспортирует занимаемую листовыми блоками последовательность байтов целиком. В экспорте блоков ссылок нет никакой необходимости, поскольку они не содержат данные, а действительные ключи разделения областей поиска после импорта могут оказаться совершенно другими, и ключи блоков ссылок для экспорта просто не требуются.

Каждый из производителей использует свой собственный формат блоков, способы кодирования данных и ключей. В результате, естественно, получаемый файл зависит от используемой MUMPS системы и не может быть использован на системах других поизводителей, но может быть использован на системах того же производителя.

При импорте система проходит по каждому из записанных в файле образов блоков, разделяет пары ключ - значение и записывает их в базу данных по месту так, как эта пара должна храниться в текущей базе. Стратегия записи соответствует команде merge - данные, уже присутствующие в базе, но отсутствующие в файле экспорта, будут сохранены.

У такого блочного экспорта глобалов есть две вытекающие из его принципа особенностей. Первая особенность - это то, что экспорт выполняется очень быстро. Системе нет необходимости выполнять промежуточные преобразования значений ключей и формировать полное имя для стандартного формата экспорта. Но, при этом, импорт не избавлен от парсинга блоков на пары и выполняется несколько медленнее, чем экспорт, хотя во многих случаях и быстрее, чем импорт из стандартных форматов, поскольку ключи хранятся в образе блока в уже кодированном виде.

Вторая особенность блочного экспорта - это то, что при экспорте система использует блоки целиком, не разделяя на пары ключ - значение, поэтому такому способу экспорта нельзя указать, что необходимо экспортировать только определенные значения, задав индексы, из всего глобала. Таким образом, блочный экспорт применяется всегда целиком к глобалу.

Блочный экспорт глобалов, естественно, не ограничен различением, текстовый это формат или бинарный, поскольку он всегда используется как бинарный и экспортирует всегда все символы, содержащиеся как в

данных, так и в значениях индексов.

В практическом применении блочный экспорт используется, насколько автору известно, редко и, в основном, для переноса больших объемов данных, либо при необходимости выполнять быстрый экспорт. Традиционно таким способом переноса данных пользуются в основном разработчики и администраторы для переноса данных между собственными серверами.

7.4 Препроцессор

Препроцессор относится к нестандартным дополнениям MUMPS систем, не предусмотренным стандартом, но поддерживаемым различными производителями из соображений практической полезности.

В отличие от языков семейства Си, где препроцессор является неотъемлемым атрибутом, изначально и по умолчанию присутствующим в распоряжении программиста, в языке MUMPS его нет, и поддержка выполняется в каждой из систем самостоятельно производителями. При этом различные реализации поддерживают большинство основных особенностей для синтаксической совместимости исходных текстов и различные собственные особенности препроцессинга рутин.

Основной единицей трансляции в строчно - ориентированном языке является совокупность строк рутины. Именно по этой совокупности строк система исполнения отсчитывает смещения относительно меток. И задачей препроцессора является получение такого непосредственного кода на языке MUMPS. Если в языке Си номера строк до препроцессирования сохраняются для отладчика и директивы __LINE__, то в системах MUMPS такое соответствие не сохраняется и нет прямого соответствия между номером строки в рутине с макросами и номером строки для системы построчного исполнения. В определенном смысле это может доставить непривычные неудобства при первых применениях.

Препроцессор, несмотря на то, что появился в MUMPS системах много лет назад, до сих пор применяется редко и большинство разработок, особенно ориентированных на возможность портирования на различные реализации, его не используют.

По организации работы большинство препроцессоров использует три вида рутин - стандартные рутины, определенные в языке MUMPS, называемые также рутинами непосредственного кода (INTermediate routines), рутины содержащие макросы - макрорутины (MACro routines) и рутины, предназначенные для включения при препроцессировании - включаемые рутины (INClude routines).

Имена различаются при их использовании по условному расширению, например:

```
ROUNAME.MAC
ROUNAME.INC
ROUNAME.INT
```

Существуют также реализации MUMPS систем, в которых организация макрокода иная, и различаются только два типа рутин - INT и MAC, при этом считается, что в качестве включаемых (INC) рутин используются MAC рутины с соответствующим именем.

Поддерживаются правила компиляции: 1) INC рутины не компилируются и не порождают исполняемый байткод, но используются MAC рутинами для включения, 2) INT рутины компилируются в исполняемый байткод и 3) MAC рутины транслируются препроцессором макросов в INT рутину с тем же именем, после чего она компилируется в исполняемый байткод.

Макрорутины по своему назначению являются исходным текстом для получения INT рутины и последующего исполняемого байткода. При препроцессировании препроцессор сканирует строку за строкой последовательно и, если в строке есть директивы препроцессора, то строка соответствующим образом изменяется или не включается в выходную INT рутину. Рутина с макросами может содержать просто текст на языке MUMPS без директив препроцессора, смешивание MUMPS кода с директивами препроцессора, или одни только директивы на усмотрение программиста.

Рутины разных типов могут иметь одинаковые имена. Текст рутин хранится в разных глобалах. После трансляции MAC рутины полученный INT код по умолчанию доступен для редактирования и просмотра, в некоторых реализациях может использоваться дополнительная опция не сохранять сгенерированный промежуточный код INT рутины.

Макросы поддерживаются только в макрорутинах. В косвенных выражениях, в аргументе команды XECUTE, и в командном режиме макросы не поддерживаются. В этом случае код и подстановки косвенности выполняются вне контекста препроцессора.

Для совместимости с традиционной разработкой без использования макросов обычно поддерживается два вида трансляции - ориентированный на чистый MUMPS код и ориентированный на препроцессор. При этом, если присутствует MAC рутина, то она сначала препроцессируется для получения INT кода, и дальше вызывается транслятор INT кода. Если было указано явно расширение (или тип) рутины, то компилируется именно этот вариант.

В общем случае инструкции препроцессора состоят из директив препроцессора и макроподстановок.

При трансляции макрокода директивы выполняются, изменяя состояние внутренних определений препроцессора, или управляя условием обработки кода. Если препроцессор встречает макроподстановку, то заменяет ее по месту на ее определение с возможными указанными для нее параметрами.

Директивы практически всех препроцессоров состоят из условных директив

1. #if
2. #ifdef
3. #ifndef
4. #else
5. #endif

директив определения макроподстановок

1. #define
2. #undef

и директив управления

1. #include
2. #execute

В зависимости от реализации MUMPS системы могут поддерживать также дополнительные директивы и способы макроподстановок.

Условные директивы указывают препроцессору на необходимость или продолжить обработку или не выполнять обработку группы строк до окончания действия директивы или до переключения условия директивой #else. При этом директивы #ifdef и #ifndef проверяют существует ли определение макроподстановки, а директива #if вычисляет MUMPS выражение и программист может обращаться ко всем возможностям MUMPS системы, например вызвать функции или прочитать значение глобалов.

Директивы #define и #undef создают или удаляют определение макроподстановки.

Директива #include включает указанную директиве INC рутину и препроцессор продолжает обработку кода с первой строки этой рутины, как если бы ее текст был целиком вставлен вместо строки с директивой #include.

Директива #execute выполняет аргумент как последовательность команд языка MUMPS, как если бы они были аргументом команды xecute. В этой директиве, как и в директиве #if, разработчик может обращаться к функциям, глобальным или локальным переменным.

Макроподстановки развертываются в их определение, и в определении макроподстановки и в качестве их аргументов также могут использоваться другие макроподстановки. Генерируемый код INT рутины порождается на момент препроцессирования так, как определено макросом. В частности, если макрос вычисляет значение, зависимое от времени, или от версии, или от состояния глобалов на момент трансляции, то порожденный INT код будет содержать именно эти значения.

В большинстве случаев препроцессор макросов используется в качестве простого инструмента подстановки. Но при этом механизм препроцессирования довольно мощный, и может порождать другие рутины, или генерировать текст, вызывая сложные функции. Технически разработчику доступны все возможности самой MUMPS системы для генерации подстановок или выполнения произвольных действий на момент трансляции макрокода.

Общая схема подстановок состоит в определении имени макроса и его аргументов и в использовании его там где необходимо выполнить такую подстановку. Например, код:

```
#define DGLO(%id) ^AR67ED("tools",46,%id)
 ...
 s $$$DGLO(idrec)=$$value(idrec)
```

развертывается в код

```
 s ^AR67ED("tools",46,idrec)=$$value(idrec)
```

Основной задачей препроцессора макросов, таким образом, является помощь разработчику в упрощении разработки и в сокрытии длинных и, возможно, не очень читабельных строк в осмысленные с точки зрения разработчика синтаксические конструкции. В техническом отношении сложность макросов может быть произвольной.

Рутины, предназначенные для включения в макрорутины, используются преимущественно для создания набора определений макросов. Но также могут содержать и традиционный MUMPS код. В этом случае

он будет включаться в генерируемую INT рутину как он указан. Включаемые INC рутины могут включать другие включаемые INC рутины и макрорутина (MAC) может включать несколько включаемых INC рутин.

Кроме того, что определение макросов создает более читабельный для разработчика словарь терминов, упрощая написание программ и устраняя возможность опечаток, вторым практическим преимуществом прероцессора является согласованность MAC рутин по использованию магических констант, определенных в одном месте, во включаемой INC рутине.

К третьему практическому преимуществу относится то, что препроцессоры в MUMPS системах используют проверку, существует ли определение для подстановки, прежде чем ее выполнить. В случае если ее не существует, препроцессор выдает диагностическое сообщение об ошибке трансляции. Это поведение дает разработчикам механизм проверки на опечатки в именах переменных или функций или рутин. Для самого языка MUMPS при трансляции строки не важно, существует ли такая переменная или рутина, он не имеет такой информации. Но препроцессор уже может проверить опечатки по текущему набору определений макроподстановок. При разработке крупных прикладных систем, использующих множество имен переменных, функций и рутин, такие проверки могут существенно сократить число ошибок уже на этапе кодирования.

Хотя препроцессор и поддерживается различными системами, но к его недостаткам можно отнести то, что в MUMPS системах не встречается утилита, аналогичная утилите MAKE, чтобы автоматически перекомпилировать MAC рутины, если изменились включаемые ими INC рутины или перекомпилировать INT рутины если изменились соответствующие им MAC рутины. Тем не менее, такую утилиту можно составить самостоятельно, с учетом особенностей прикладного проекта.

Рассмотрим в качестве примера построение таких зависимостей. Для того, чтобы вести информацию о зависимостях MAC файлов от INC файлов, необходимо где-то дополнительно сделать отметку о том, что при трансляции MAC файла были транслированы включенные INC файлы. Пусть такая зависимость ведется в глобали

```
^DEPENDS(macroutine,incroutine)=""
```

При препроцессировании INC рутины необходимо выполнить отметку о том, что она препроцессировалась:

```
#execute s ^mtemp1("INC",$j,incroutine)=""
```

Эту строку вставляем в текст INC рутины.

При препроцессировании MAC рутины просто вносим записи, полученные при обработке INC рутин:

```
#execute k ^DEPENDS(macroutine)
#execute m ^DEPENDS(macroutine)=^mtemp1("INC",$j)
#execute k ^mtemp1("INC",$j)
```

Эти строки вставляем в текст MAC рутины. Удаление пройденных INC рутин нужно для того, чтобы информация о транслированных INC рутинах не использовалась при последующей трансляции другой MAC рутины.

При трансляции таких рутин будут автоматически заполняться зависимости MAC рутин от INC рутин в глобале

```
^DEPENDS(macroutine,incroutine)=""
```

Далее остается получить имена MAC рутин и INC рутин либо автоматически используя особенности препроцессора, либо явно указав текущее имя, например если рутина ETRAP.INC, то вставить строку

```
#execute s ^mtemp1("INC",$j,"ETRAP")=""
```

Далее, при реализации утилиты MAKE, необходимо просто соблюсти правила трансляции:

1. Проверять необходимость перекомпиляции по списку рутин, входящих в определенный проект.
2. Если есть INT рутина, но нет ее байткода, то транслировать.
3. Если есть MAC рутина но нет INT рутины и байткода, то транслировать.
4. Если дата изменения байткода раньше чем INT рутины или MAC рутины то транслировать.
5. Если MAC рутина зависит от какой-либо INC рутины и есть INC рутина с датой изменения позже чем MAC рутина, то транслировать.

Кроме того, нужно определить по документации на используемую MUMPS систему, как именно можно получить дату и время последнего изменения INC, MAC, INT рутин и байткода, а также как именно программно вызвать трансляцию MAC и INT рутин. Эти несколько правил

приведены навскидку и в реальном проекте и утилите перекомпиляции, конечно, могут учитываться более сложные правила, включая ведение возможных зависимостей MAC рутин от версии, от изменений управляющих данных в глобалах, и так далее.

При выполнении препроцессирования разработчик может использовать информацию о текущей версии MUMPS системы и, в зависимости от нее, управлять генерацией MUMPS кода. Например, директивами препроцессора определять, для какой MUMPS системы или операционной системы выполняется трансляция, и использовать более удачные, или специфические, или оптимизированные для нее возможности. Например, генерация кода блокировки на чтение в зависимости от версии MUMPS системы:

```
#if $zv["MiniM"
#define MINIM
#else
#define CACHE
#endif
 ...
 ...
 ...
#ifdef MINIM
 l +^A4("WH","MD","F",cubeId,$$LockName^ST51()):1
#else
 l +(^A4("WH","MD","F",cubeId)#"S"):1
#endif
 ...
```

Здесь предполагается, что код может выполняться либо на системе MiniM, либо на системе Caché.

Поскольку рутины для макропроцессора различаются по типу, то к традиционным форматам экспорта рутин системы, поддерживающие препроцессор, дополнительно поддерживают расширенный формат экспорта рутин, сохраняющий информацию о типе рутины. Это формат RSA (Routine Save Archive).

MUMPS системы, поддерживающие препроцессор макросов, также, в принципе, могут быть использованы для кроссразработки для получения кода для целевой MUMPS системы, не поддерживающей препроцессор. Те места кода, которые должны отличаться в зависимости от версии, должны генерироваться по условию какую из целевых систем использовать. Например, перед трансляцией пакета рутин можно внести запись в глобал о версии и директивой #if проверить имя целевой системы.

Полученный пакет INT рутин затем может быть экспортирован в стандартном формате и перенесен на целевую MUMPS систему.

Вариант кроссразработки под систему, не поддерживающую макросы, очень экзотичен и специфичен, а вот вариант с кроссразработкой под другую систему или под другую версию системы в практике встречается чаще. Но традиционно, при необходимости иметь код, работающий на различных реализациях MUMPS, планируется иначе - либо выполнением специфичного от версии кода через xecute, либо комплектованием прикладной системы набором специфичных для целевой системы рутин, либо, в крайнем случае, комбинированием соответствующих команд if или постусловий.

7.5 Формат $HOROLOG

Системная переменная $HOROLOG возвращает значение текущей даты и времени в локальном времени системы (с учетом часового пояса).

Дата и время возвращаются в виде двух чисел, разделенных запятой. Первое число - количество дней, прошедших начиная с пятницы, 31 декабря 1840 года. Второе число показывает количество секунд дня, прошедших с полуночи.

Формат и точку отсчета для переменной $HOROLOG выбрал James M. Poitras, один из разработчиков системы MDH, ранней версии современных MUMPS систем, в 1969-м году. MDH - это крупная автоматизированная система для учета медицинских сведений. С его слов:

- Я вспомнил, что старейший (видимо, наиболее старейший) гражданин США, ветеран Первой Мировой Войны, имел к тому времени возраст 121 год. Поэтому я захотел представить дату в юлианском календаре так чтобы возраст можно было легко вычислять и представить любую дату в виде числа. Я решил, что начальной даты отсчета 1840 должно быть достаточно.

Другая легенда выбора отсчета 1840 года гласит, что в этом году была произведена первая запись в систему MDH. Но это шутка.

В частности, отсчет 60000 приходится на 10 апреля 2005-го года.

Формат системной переменной $HOROLOG является наиболее распространенным из всех используемых форматов дат и времени в прикладных системах на MUMPS. С этим форматом работает множество расширенных $Z функций различных расширений, и дополнительные расширенные системные переменные, например $ZTIMESTAMP, также придерживаются выбранного формата.

При возврате значение переменной выглядит так:

```
USER>w $h
62542,57317
```

Разделитель запятая не должен при этом восприниматься как десятичная точка, это только разделитель количества дней и количества секунд. Число секунд не дополняется лидирующими нулями. Это обстоятельство необходимо учитывать при сортировке по дате и времени.

Для приведения формата $HOROLOG к виду, допускающему сортировку, применяется или метод дополнения обоих чисел до строки или приведение к числу секунд. При дополнении до строки обе части дополняются так, чтобы формат строки при строковой сортировке приводил к корректной сортировке дат, например такой:

```
USER>s h=$h w $j($p(h,","),6)_","_$j($p(h,",",2),6)
 62542, 57693
```

При приведении к числу секунд часто используется два метода. Первый основан на том, что в сутках содержится:

```
USER>w 24*60*60
86400
```

секунд, поэтому общее число секунд вычисляется по формуле:

```
USER>s h=$h w $p(h,",")*86400+$p(h,",",2)
5403686644
```

Для получения значений даты и времени в формате $HOROLOG из такого числа применяются операции деления нацело и взятие остатка от деления нацело на 86400.

Второй метод основан на условном числе секунд 100000 так, чтобы при вычислении в результате было видно обе части - и дата и время, каждая из частей в формате $HOROLOG:

```
USER>s h=$h s total=$p(h,",")*100000+$p(h,",",2)

USER>w
h="62542,58032"
total=6254258032
```

Для получения значений даты и времени в формате $HOROLOG обратно из такого числа применяется деление нацело и остаток от деления нацело на 100000.

Первый вариант приведения к числу (домножение номера даты на 86400), кроме того, используется для вычисления разности дат и времени в секундах для двух дат, заданных с указанием времени.

Нужно обратить внимание на то, что значение $HOROLOG, как системной переменной, волатильно, и два различных обращения к этой переменной могут дать не только различные значения секунд, но и различные значения дат (при работе программы около полуночи). Поэтому на практике применяется взятие значения $HOROLOG однократно, а затем использование этого значения в нескольких операциях вычислений.

В настоящее время современные MUMPS системы строго поддерживают лишь положительный отсчет числа дней в $HOROLOG, но в случае, если необходимо оперировать также отрицательными значениями, необходимо проверить, как эта операция поддерживается на применяемой и целевой MUMPS системах, или написать функции преобразования дат самостоятельно.

Нужно отметить, что корректное преобразование даты, заданной числом относительно точки отсчета в номер года, месяца и дня, а также обратно, является не совсем тривиальной задачей, поскольку такая операция должна учитывать, что каждый 4-й год високосный, при этом каждый 100-й не високосный, но каждый 400-й високосный.

В частности, на языке MUMPS один из вариантов декодирования значения даты в формате $HOROLOG в значения года, месяца и дня, может быть таким:

```
DATEDECO ; $horolog decoding to year, month, day
 ; MiniM internals
 ; http://www.minimdb.com
 q
IsLeapYear(Y)
 q (Y#4=0)&((Y#100)!'(Y#400))
DECODE(H,Year,Month,Day) ; d DECODE^DATEDECO($H,.Y,.M,.D)
 n D1,D4,D100,D400
 s D1=365,D4=D1*4+1,D100=D4*25-1,D400=D100*4+1
 n Y,M,D,I,T,DayTable
 s T=H+672046
 ; return zeroes if date before 0 year
 i T'>0 s Year=0,Month=0,Day=0 q
 s T=T-1,Y=1
 f  q:T<D400  s T=T-D400,Y=Y+400
 s I=T\D100,D=T#D100
 i I=4 s I=I-1,D=D+D100
 s Y=Y+(I*100),I=D\D4,D=D#D4,Y=Y+(I*4),I=D\D1,D=D#D1
 i I=4 s I=I-1,D=D+D1
 s Y=Y+I,DayTable="31,28,31,30,31,30,31,31,30,31,30,31"
 i $$IsLeapYear(Y) s $p(DayTable,",",2)=29
 s M=1
 f  s I=$p(DayTable,",",M) q:D<I  s D=D-I,M=M+1
 ; return decoded day
```

```
 s Year=Y,Month=M,Day=D+1
 q
```

А вариант кодирования значений года, месяца и дня в соответствующее значение по отсчету системной переменной $HOROLOG на языке MUMPS может быть таким:

```
IsLeapYear(Y)
 q (Y#4=0)&((Y#100)!'(Y#400))
ENCODE(year,month,day)
 n ret,i,MonthDays
 s ret=day,MonthDays="31,28,31,30,31,30,31,31,30,31,30,31"
 i $$IsLeapYear(year) s $p(MonthDays,",",2)=29
 f i=2:1:month s ret=ret+$p(MonthDays,",",i)
 s i=year-1
 q ret+(i*365)+(i\4)-(i\100)+(i\400)-672046
```

Приведенные функции кодирования и декодирования дат по шкале $HOROLOG оперируют также и отрицательными значениями по шкале $HOROLOG и используются на языке C в системе MiniM. В случае если применяемая MUMPS система по каким-либо причинам не поддерживает работу с отрицательными значениями дат по шкале $HOROLOG, но это необходимо в прикладной задаче, то могут быть использованы эти функции на языке MUMPS.

Конечно, внутри системы исполнения MUMPS нет специальных маленьких и очень точных часов, и отсчет времени системная переменная $HOROLOG производит по текущему времени, установленному на компьютере. В случае перевода даты и времени на компьютере переменная будет возвращать значение по новому отсчету. Поэтому, строго говоря, прикладная программа не должна полагать, что одно событие произошло действительно позже другого, если значение $HOROLOG стало для него больше. Но на практике именно так и поступают.

Многие современные реализации MUMPS поддерживают расширенные системные функции $ZDATE, $ZDATEH, $ZTIME, $ZTIMEH или аналогичные для преобразования значения даты и времени между различными форматами. Для них опорным является формат $HOROLOG. Например:

```
USER>f i=1:1:14 w i,": ",$zd($h,i),!
1: 03/26/2012
2: 26 мар 2012
3: 2012-03-26
4: 26/03/2012
5: мар 26, 2012
```

```
6: мар 26 2012
7: мар 26 2012
8: 20120326
9: Март 26, 2012
10: 1
11: Пн
12: понедельник
13: 26 Март 2012
14: 26.03.2012
```

Кроме того, многие библиотеки функций написанные на MUMPS, входящие в большие программные пакеты, также оперируют форматом даты и времени, заданном переменной $HOROLOG.

Точка отсчета времени переменной $HOROLOG ведется по локальному времени. При этом компьютеры, находящиеся в различных часовых поясах, в один и тот же момент времени имеют различные показания $HOROLOG. Для выравнивания общего отсчета по UTC нужно дополнительно применять отсчет часового пояса по расширенным системным переменным $ZTIMEZONE и $ZTIMESTAMP. В зависимости от реализации MUMPS такие переменные могут отличаться именем или тот же функционал может быть выполнен в виде системных функций. Особенно важно использование единого времени по UTC при объединении в единую систему территориально удаленных серверов, а также при передаче в единый центр данных из территориально удаленных серверов.

7.6 Опции устройств

Одним из препятствий для полной переносимости программ, написанных на MUMPS, между различными реализациями, является различная поддержка опций устройств.

Стандарт отводит поддержку типов устройств и их опций самим производителям реализаций. Это приводит к тому, что в различных MUMPS системах одни и те же операции, например, открыть файл - прочитать строки - закрыть файл, выполняются различным кодом. Код отличается не только различным соглашением об именовании устройств, но и различным синтаксисом указания, что является опцией.

Опции устройства (или параметры устройства) указываются после имени устройства и могут перечисляться через двоеточие. Синтаксически опция указывается согласно стандарту 1995-го года (ныне действующий) так:

```
deviceparameters := | deviceparam                              |
                    | ( [ [ deviceparam ] : ] ... deviceparam |
```

```
deviceparam := | expr               |
               | devicekeyword      |
               | deviceattr = expr  |

devicekeyword := name
deviceattr := name
```

Синтаксически это позволяет указать, например, такие способы:

```
use filedev:(truncate)
use filedev:(mode="w")
```

При этом стандарт ставит транслятор в неоднозначное положение, как отличить в записи опции (на диаграмме это deviceparam) ключевое слово опции, имя локальной переменной и выражение как результат сравнения локальной переменной с другим выражением. При этом стандарт перечисляет альтернативу deviceparam := expr первой, отдавая ей приоритет, но в expr как раз входит, как альтернатива, имя локальной переменной.

В рекомендации стандарта, а не в сам стандарт, входит другой вариант определения devicekeyword и deviceattr:

```
devicekeyword := [ / ] name
deviceattr := [ / ] name
```

Применение слеша для указания имени опции или ключевого слова уже позволяет транслятору определить точно, что это именно имя опции, а не локальная переменная и что это имя опции со значением параметра, а не выражение сравнения.

Примеры, приведенные выше, уже выглядят так:

```
use filedev:(/truncate)
use filedev:(/mode="w")
```

При этом сложилась ситуация, что среди современных реализаций MUMPS есть такие, которые поддерживают первый вариант, и те, которые поддерживают второй. К первым относится, в частности, GT.M, ко вторым Caché и MiniM.

В силу сложившихся особенностей различных систем и для обспечения совместимости с имеющимися разработками разработчики прикладных программ должны применять методы изолирования кода, если необходимо выполнять программы на различных MUMPS системах. К ним относятся, например, такие:

1. Вынос операций с устройствами в мнемоники.
2. Выполнение операций с устройствами в команде xecute при выборе надлежащего синтаксического варианта.
3. Поддержка различных библиотечных рутин для устройств для различных реализаций MUMPS.

Кроме различного синтаксиса опций, различные MUMPS системы, даже поддерживающие одинаковый способ синтаксического указания имени опции, поддерживают устройства ввода-вывода по-разному и с разными именами опций и трактовкой их значений.

7.7 $X и $Y

Системные переменные $X и $Y были введены в язык в качестве встроенных и стандартных системных переменных с самого начала, и по своему назначению отражают положение каретки ввода на терминальных устройствах. При эксплуатации MUMPS систем эти переменные, при этом, разработчики используют с большой осторожностью.

Системные переменные $X и $Y доступны и на запись и на чтение. При записи в них MUMPS система принимает меры к установке позиции каретки в указанное положение. Синтаксически эта операция выглядит так:

```
set $X=12
set $Y=5
```

При этом язык на логическом уровне не специфицирует отличие физического поведения таких переменных для различных терминальных устройств. В зависимости от реального физического способа отображения и типа терминального устройства MUMPS система выполняет действия, соответствующие этому типу устройства. В частности, для Windows консоли выполняется позиционирование консольного курсора

```
SetConsoleCursorPosition
```

используя функционал WinAPI, а для телнет клиентов выполняется отсылка команд позиционирования в виде эскейп-последовательности.

Для терминальных устройств иного типа, например, для принтеров и терминалов, выполняется передача специфических для них команд позиционирования каретки.

В зависимости от типа такого терминального устройства содержание эскейп-последовательности для позиционирования также, в принципе, может отличаться, хотя большинство терминальных протоколов по базовым эскейп-последовательностям совпадают.

При разработке под терминальные устройства при использовании переменных $X и $Y разработчики, безусловно, должны проверить, как именно выполняется управление позиционированием на всех типах устройств, с которыми необходимо работать.

Более сложный вопрос стоит с чтением системных переменных $X и $Y. Проблема в том, что MUMPS системе не всегда может быть известно, где реально находится каретка ввода на терминальном устройстве после того, как была выполнена какая-либо из команд. Вообще говоря, MUMPS система может гарантировать корректный пересчет координат лишь при выполнении команды write с выводом строки, состоящей из печатных символов.

Все остальные случаи вывода уже не гарантируют корректного значения $X и $Y для произвольно взятого терминально-ориентированного устройства. К таким операциям относятся операции управления форматом

```
write !
write #
write ?NN
```

операции вывода отдельного символа, даже если это код печатного символа

```
write *code
```

и операция вывода с помощью эскейп-последовательностей или мнемоник, например

```
write /MNEMNAME(params)
```

В этих случаях, вообще говоря, система может лишь гарантировать, что управляющая последовательность была передана отображающему устройству.

При выполнении операций управления терминальным устройством через мнемоники или через явную передачу эскейп-последовательностей, вообще говоря, перед каждой мнемоникой стоит задача выполнить определенное действие, например удалить строку. При этом не для каждой такой мнемоники может быть гарантировано, как именно данный тип терминального устройства спозиционирует каретку по выполнению

операции. Хотя для большинства мнемоник, косвенно меняющих или могущих изменить положение каретки, такое определение дано для эскейп-последовательности.

Существует довольно большое число типов терминальных устройств, отличающихся в деталях протокола управления, набором поддерживаемых эскейп-последовательностей и побочным эффектом позиционирования каретки при выполнении команд управления. И, зачастую, для корректной работы с различными типами устройств, прикладные программы, ориентированные на сложный терминальный вывод, поддерживают специальную библиотеку с настройками действий, которые необходимо выполнить в зависимости от типа терминального устройства.

Вообще говоря, MUMPS система может поддерживать отсчет координат $X и $Y даже для нетерминальных устройств, например для текстовых файлов. Но и в этом случае не выполняется пересчет положения условной каретки при выводе кода символа.

Положение с чтением значений $X и $Y во многом сходно с чтением значения $TEST. Общая практика состоит в том, что фрагмент кода не должен полагаться на значение такой системной переменной, если не он привел к изменению значения. И, для того, чтобы использовать переменные $X и $Y на чтение, разработчики предварительно выполняют явное позиционирование, и лишь затем код выполняется в условиях относительно корректного позиционирования и корректного отсчета по координатам терминального устройства.

Таким образом, разработчики, при необходимости использования значений $X и $Y на чтение, должны в любом случае протестировать поведение кода для различных терминальных устройств, которые предстоит эксплуатировать и проверить, после каких команд вывода они могут полагаться на корректное значение $X и $Y, а после каких нет.

7.8 Возврат результатов

При планировании прикладных систем разработчики на MUMPS имеют возможность выбирать из множества вариантов, как именно вычисляющая функция или алгоритм будут возвращать результат.

Первое, на что нужно обратить внимание при выборе метода - это поддерживаемое используемыми MUMPS системами ограничение на длину строки. В случае если результат вычисления гарантированно укладывается в этот лимит, то обычно применяется функция, возвращающая значение, и вызываемая через $$.

При выборе варианта в виде возврата одной строки остается определиться, будет ли применено структурирование результата - будет ли строка одним цельным значением, или следует применить структурирование в виде строки с разделителями (для $PIECE()) или в виде списка (для $LIST()).

Если результат вычислений не укладывается в поддерживаемое системой ограничение по длине, или необходимо иное структурирование, то разработчики должны выбирать из более сложных случаев. Приведем основной список возможностей вернуть результат вычисления алгоритмом:

1. Возврат по значению ($$).
2. Возврат по ссылке в локальную переменную.
3. Запись в предопределенную переменную.
4. Возврат значений в переменную, имя которой передается косвенно.
5. Итеративный возврат.

Теперь рассмотрим варианты возврата значений и их особенности подробнее.

7.8.1 Возврат по значению ($$)

Для возврата по значению используется вызов вычисляющей функции в контексте возврата ($$) и возврат вычисленного значения командой QUIT. Простой пример возврата одной строки по значению:

```
ByValue(param) ; k  w $$ByValue^RETURN(123) w
 n ret
 s ret="Calculate, param = "_param
 q ret
```

При вызове просто выполняется команда QUIT с аргументом:

```
USER>k  w $$ByValue^RETURN(123) w
Calculate, param = 123
USER>
```

К подводным камням такого возврата можно отнести поддерживаемое используемой MUMPS системой ограничение на длину строки и возможность вызова без контекста возврата командой DO. Для второго случая

у программистов есть системная переменная $QUIT, при ее равенстве 1 возврат командой QUIT с аргументом необходим, а если значение 0, то возвращать не следует, нужна команда QUIT без аргумента.

К договоренности по формату вызова в этом случае также нужно отнести ожидаемое структурирование возвращаемой строки: будет ли это цельное значение, строка с разделителями или список. В разработках при применении строки с разделителями можно встретить два варианта: использование предопределенного разделителя и запись используемого разделителя первым символом возвращаемой строки. Второй вариант встречается намного реже, но обеспечивает корректную работу кода в случае смены разделителя и возможность для вызываемой функции использовать наиболее подходящий разделитель в зависимости от специфики данных.

В случае использования списковой структуры (для $LIST()) нужно обязательно проверить, поддерживается ли такой функционал в целевой MUMPS системе или ее версии.

Возрат по значению обычно используется для возврата атомарных или структурированных простым способом значений, если объем возврата небольшой и при развитии системы и предъявляемы к ней требований этот объем также не выйдет за пределы ограничения на длину строки.

7.8.2 Возврат по ссылке

Для возврата результата вычисления по ссылке вычисляющей функции передается ссылка на локальную переменную (через точку). Могут быть переданы также несколько ссылок, на несколько переменных. Простой пример такой передачи с использованием структурирования по индексам:

```
RunByRef ; k  d RunByRef^RETURN w
 n a,b
 d ByRef(.a,.b)
 w
 q
ByRef(param1,param2)
 s param1(1)="param1 1"
 s param1(2)="param1 2"
 s param2(1)="param2 1"
 s param2(2)="param2 2"
 q
```

При выполнении кода вызывающая функция передает ссылки на свои локальные переменные (фактические параметры), а вызываемая в них записывает, используя свои имена формальных параметров:

```
USER>k  d RunByRef^RETURN w

a(1)="param1 1"
a(2)="param1 2"
b(1)="param2 1"
b(2)="param2 2"
USER>
```

Возврат значений по ссылке можно свободно комбинировать с возвратом по значению.

К подводным камням такого способа относится ограничение на общий объем памяти, занимаемый локальными переменными процесса и невозможность передать ссылку на глобал, так как этот вариант не поддерживается синтаксически.

При таком способе может быть применено произвольное структурирование вычисленных результатов: цельная строка, строка с разделителями, список, структурирование по индексам, а также могут быть использованы несколько разных локальных переменных.

К плюсам или минусам (в зависимости от ситуации) может быть отнесено то, что перед вызовом вычисляющей функции в передаваемые локальные переменные уже могут быть записаны некоторые значения и они будут доступны вычисляющей функции. В случае если вызываемой функции необходимо гарантировать, что весь возврат в переменную принадлежит ей, перед записью нужно удалить из этой переменной все значения:

```
ByRef(param1,param2)
 k param1,param2
 s param1(1)="param1 1"
 ...
```

Возврат по ссылке обычно используется тогда, когда необходимо передать больше чем одно атомарное значение, или необходимо структурировать ответ по индексам, но возвращаемый объем заведомо умещается в области локальных переменных процесса.

7.8.3 Запись в предопределенную переменную

Запись в предопределенную переменную применяется из-за возможностей языка MUMPS видеть все созданные ранее по стеку локальные переменные, если они не были экранированы командой NEW. Многие разработки на языке MUMPS, выполнявшиеся много лет назад, использовали такую возможность как за неимением команды NEW, так и для организации взаимодействия без передачи параметров на стеке.

Для вычисляющей функции просто резервируется набор имен переменных (локальных или глобальных), в которые ожидается возврат.

Для современных методик программирования такой способ выглядит несколько архаично, но, тем не менее, имеет свои плюсы - возможно применять произвольное структурирование.

Как пример можно привести фрагмент кода работы с датами, который часто приводится как демонстрационный по языку MUMPS:

```
7    S %=%H>21608+%H-.1,%Y=%\365.25+141,%=%#365.25\1
     S %D=%+306#(%Y#4=0+365)#153#61#31+1,%M=%-%D\29+1
     S X=%Y_"00"+%M_"00"+%D Q
     ;
YX   D YMD S Y=X_% G DD^%DT
YMD  D 7 S %=$P(%H,",",2) D S K %D,%M,%Y Q
```

Здесь даже не используются параметры меток и возврат по значению, код просто использует текущие локальные переменные на чтение или на запись. С тем же успехом он мог бы использовать и глобальные переменные. Фактически, в этом случае вызываемые метки являются частью общего вычисляющего алгоритма.

К плюсам такого способа относится практически полное отсутствие каких-либо ограничений, а к недостаткам - необходимость согласовывать вызывающий и вызываемый код по именам локальных переменных и несколько большая трудность чтения такого кода современными программистами.

Предопределенные переменные с точки зрения классического MUMPS используются всегда, поскольку язык сам по себе определен как строчно-ориентированный и структурирование кода на функции это лишь взгляд программиста, а в самом языке нет таких синтаксических конструкций, как функция, содержащая собственные переменные функции.

7.8.4 Возврат значений косвенно

Для косвенного возврата значений вызываемой функции передается имя (или несколько имен) локальных или глобальных переменных. Вызываемая функция для записи результата использует это имя косвенно.

В качестве демонстрационного примера приведем такой вариант:

```
RunIndir ; k  d RunIndir^RETURN w
 n var
 s var("data")="123"
 d Indir($na(var))
 w
```

```
 q
Indir(name)
 s @name@("ret")="Calculate "_@name@("data")
 q
```

Здесь вычисляющая функция записывает результат, используя косвенность имени переменной. Кроме того, заодно в этой же переменной ей передается исходное значение:

```
USER>k  d RunIndir^RETURN w

var("data")=123
var("ret")="Calculate 123"
USER>
```

Такой сособ возврата значений, также как и передачу по ссылке, можно свободно комбинировать с возвратом по значению.

К подводным камням такого способа можно отнести случайную возможность передать в вычисляющую функцию имя локальной переменной, которое уже используется ей внутри. В этом случае произойдет применение операции косвенности к внутренней переменной:

```
RunIndir ; k  d RunIndir^RETURN w
 n var
 s var("data")="123"
 d Indir($na(var))
 w
 q
Indir(name)
 n var s var=456
 s @name@("ret")="Calculate "_$g(@name@("data"))
 s @name@("ret")="Calculate "_var
 q
```

И вызываемая функция не получит результат:

```
USER>k  d RunIndir^RETURN w

var("data")=123
USER>
```

Такая ситуация может случайно произойти при использовании часто употребимых имен временных локальных переменных.

Возврат значений косвенно обычно применяется при необходимости вернуть большой объем данных, и зачастую для такого способа используются не локальные, а глобальные переменные. Объем возврата, в принципе, может быть любым, в пределах, доступных базе данных. Разумеется, при передаче имени глобала коллизий по именам с локальными переменными произойти не может.

7.8.5 Итеративный возврат

Под итеративным возвратом понимается возврат из вычисляющей функции очередной порции результата. Для получения общего результата в этом случае нужно вызвать вычисляющую функцию несколько раз, обычно в цикле.

Для организации итеративного возврата ключевым моментом является соглашение о передаче состояния очередной итерации таким образом, чтобы обе стороны могли определить, является ли вызов начальным, завершающим, или необходимо продолжить итерацию далее.

В простых случаях таким индикатором является само значение итератора, которое создается и проверяется вызывающей стороной, например, на равенство пустой строке или иному предопределенному значению.

Приведем простой пример итеративного возврата, где критерием как начала так и окончания итерации является равенство пустой строке самого итератора:

```
RunIter ; k  d RunIter^RETURN w
 n iter,value s iter=""
 f  s value=$$Iter(.iter) q:iter=""  d
 . w "Next: ",value,!
 q
Iter(index)
 s index=$o(^rMAC(index))
 i index="" q ""
 q index_": "_$g(^rMAC(index,0))
```

При выполнении такого кода возвращается перечень имеющихся макрорутин с датами их модификации:

```
USER>k  d RunIter^RETURN w
Next: BREAK: 62682,53568
Next: DTC: 62623,48148
Next: ETRAP: 62671,63182
Next: ETSTACK: 62659,70354
Next: EXTRUN: 62686,58570
Next: RETURN: 62690,54306
Next: STACK: 62682,82491
Next: ZTRAP: 62651,61619
```

В более сложных случаях может быть использовано усложнение соглашений: использование структурного итератора, специальной функции создания начального значения итератора и, возможно, специальных временных служебных данных, использование специальной функции проверки критерия окончания итерации, использование специальной функции завершения итерации с удалением временных данных.

В определенном смысле такое полное построение итеративного возврата аналогично запросу с соответствующим набором функций:

```
RunIter ; k  d RunIter^RETURN w
 n iter,value
 d ICreate(.iter)
 f  d INext(.iter) q:$$IEOF(.iter)  d
 . s value=$$IValue(.iter)
 . w value,!
 d IClose(.iter)
 q
ICreate(index)
 s index=""
 q
INext(index)
 s index=$o(^rMAC(index))
 q
IValue(index)
 q index_": "_$g(^rMAC(index,0))
IEOF(index)
 q index=""
IClose(index)
 q
```

С переходом к такому обобщенному итератору его набор функций может быть изменен в любое время с использованием новых требований, предъявляемых к запросу, но все вызывающие функции останутся неизменными, поскольку не используют явных критериев начала и окончания итераций.

Итеративный возврат применяется обычно в ситуациях, когда объем возвращаемых данных непредсказуем, но передавать результат через предварительную запись в промежутоный временный глобал с последующим его анализом по каким-либо причинам нецелесообразно.

7.8.6 Потоковый возврат

Потоковый возврат применяется для возврата результата в устройство с помощью команд записи в устройство.

Такие виды возврата вычисленных значений применяются при генерации ответа клиентским программам, например при обслуживании TCP - соединения клиентской программы с сервером, при генерации WEB страницы, при генерации файла.

Результат вычислений в этом случае возвращается не вызывающей функции, а в устройство ввода-вывода.

При использовании потокового возврата, конечно, вызывающая функция должна придерживаться одинаковых соглашений с вызываемой функцией об устройстве ввода-вывода для получения результата: используется ли текущее устройство, или следует передать имя устройства, которое будет открыто и использовано на запись.

7.9 %Z - рутины

При эксплуатации MUMPS систем и при разработке программного обеспечения для них разработчикам может понадобиться размещать часть своих рутин так, чтобы они были доступны на выполнение из любой базы данных. Практически все реализации MUMPS систем поддерживают правило отображения рутин из некоей особой базы данных на остальные так, что рутины можно вызывать, не указывая явно базы данных где они хранятся, как если бы они находились в текущей базе данных.

Практически все современные СУБД, поддерживающие хранение и исполнение подпрограмм в базе данных, в той или иной мере поддерживают такую функциональность. Такую базу данных обычно называют системной или аналогиным пользуются термином (в зависимости от предпочтений производителя), и в ней размещают рутины, относящиеся ко всем базам данных, а не только к определенной прикладной программе. Либо относящиеся к прикладной программе, которая используется различными процессами, работающими в различных базах данных.

При этом возникает вопрос коллизий имен рутин между различными разработчиками. При эксплуатации СУБД при выполнении апгрейда на последующую версию инсталлятор устанавливает в системную базу данных комплект рутин, входящий в эту версию. Для того, чтобы не возникло коллизий с именами рутин и с замещением рутин с таким же именем, не принадлежащих самой СУБД, производители рекомендуют использовать специальные соглашения об именовании рутин, размещяемых в системной базе данных.

Рутины, которые должны отображаться на другие базы данных, должны начинаться на символ процент (%). При этом производители MUMPS систем не используют имена рутин, начинающиеся на символы %Z, в стандартной комплектации.

Группа имен рутин, начинающихя на символы %Z, таким образом, отводится для разработчиков прикладных программ или для дополнительных модулей третьих лиц.

Исторически сложилось так, что практически все системные программы производители MUMPS систем именуют в верхнем регистре. В

каких-то случаях это делается из соображений совместимости, в каких-то из сохранения общего стиля.

И, как следствие этого неформализованного правила, независимые разработчики также могут использовать процентные рутины с именами, содержащими символы в нижнем регистре, не опасаясь коллизий со стороны процесса апгрейда.

Кроме проблемы коллизий с набором имен, используемым применяемой СУБД, также стоит вопрос коллизий между различными производителями прикладных программ общего назначения и различных библиотек. Для этого случая также не существует официально формализованного правила, но большинство разработчиков прибегает к механизму разделения имен таких рутин библиотечного назначения путем использования префиксов.

В качестве префикса обычно используются символы сокращения от названия библиотеки, например

```
%xdXXX
%iaXXX
%aXXX
```

где вместо символов XXX используется уже содержательное имя, как-бы уже имя рутины в самом этом пакете рутин.

Интересно то, что автору действительно довелось встретить случай, когда разработчики рутин общего библиотечного назначения не следовали правилам именования для избежания коллизий, и при переносе таких рутин на другую реализацию MUMPS системы действительно возникла коллизия по именам, причем устранить проблему оказалось непросто - имя рутин использовалось также в нескольких приложениях на других языках. В этом случае было принято административное решение - после апгрейда снова устанавливать пакет библиотеки поверх имеющихся рутин и не использовать системную рутину, входящую в комплект СУБД. Проблема коллизии не возникала до тех пор, пока не произвели перенос проекта на другую СУБД. Такой случай показывает неверность утверждения, что если прикладная система работает нормально, то в ней нет ошибок.

Общепринятой и сложившейся практикой среди разработчиков на MUMPS стало то, что те рутины, которые должны быть импортированы в одну базу данных, готовятся для импорта в виде одного комплекта файлов. Соответственно, для %Z рутин и библиотечных, устанавливаемых в системную базу данных, также готовится отдельный комплект файлов импорта и инструкция с указанием в какую базу данных их необходимо импортировать. Отдельные реализации MUMPS систем допускают

просто перенос подготовленного файла базы данных с необходимыми рутинами, но такой перенос для системной базы данных в общем случае не выполним.

7.10 Планирование файлов

СУБД оперирует данными, организованными в базы данных. При этом физическая организация баз данных может отличаться для СУБД разных типов. Данные могут размещаться и храниться физически в памяти, поступать из внешних источников, храниться в виде файлов операционной системы, использоваться сырые неразмеченные разделы дисков, храниться на лентах. Применяемые в практике СУБД обычно комбинируют эти способы организации.

Пример организации виртуальной СУБД, не хранящей собственные данные, а получающей их из сторонних источников - WMI, когда приложение обращается с запросом, характерным скорее для СУБД, но с целью определить, например, температуру процессора или список работающих процессов.

Пример организации СУБД, работающей только с файлом - это парсер XML или INI файла, когда приложение обращается к данным, хранящимся в одном файле.

Промышленные СУБД обычно применяют хранение данных в виде файлов операционной системы, различные методы кеширования хранимых данных и дополнительные служебные файлы для поддержания операций транзакций, бекапа, процедуры восстановления после аппаратных и программных сбоев.

Один экземпляр промышленной СУБД оперирует обычно множеством файлов одновременно, выполняя при необходимости чтение и запись. И, для корректного планирования файлов данных, нужно понимать принципы и способы организации работы дисковых накопителей.

Дисковый накопитель упрощенно представляет собой действительно диск или несколько, привод, перемещающуюся головку и систему управления всем механизмом, заключенные в корпус. Ключевым элементом является головка чтения - записи. У каждого из шпинделей она одна, и, если программе необходимо выполнить чтение - запись, то головка должна выполнить физическое позиционирование к необходимому месту на диске. После выполнения операции чтения - записи головка должна спозиционироваться в другое место. Поскольку это процесс механический, то на каждое такое перепозиционирование уходит время. Если программа оперирует одним сектором, то коэффициент использования головки

высокий. Если несколькими, включая принадлежащие нескольким файлам, то низкий.

В идеале аппаратура должна предоставить по отдельной головке для каждого сектора, но такого не бывает, хотя промышленность уже нашла решение, но в виде выпуска твердотельных накопителей.

Для общего улучшения производительности (если имеется такая возможность) лучше разделять всю совокупность используемых файлов между как можно бóльшим числом шпинделей. В идеале нужно поместить каждый из используемых файлов на отдельный накопитель и базу данных разделить на несколько файлов на физически различных накопителях.

Физически это выполняется либо планированием размещения файлов на физически разных накопителях, либо применением различного рода RAID массивов, когда файлы автоматически аппаратно разделяются между несколькими шпинделями.

Для планирования размещения файлов нужно определить, какие из файлов какими операциями используются - последовательными или произольными позиционированиями. В частности, если есть процесс записи журнала, то лучше ему предоставить под журнал отдельный диск, чтобы он не беспокоил головки дисков файлов данных.

Для составления списка файлов для размещения на физически различных дисках нужно обратиться к описанию используемой СУБД, и определить способ переноса файлов и изменения конфигурации СУБД.

Исполняемые файлы (exe) после запуска процессов обычно более не используются, и образы процессов используются только в памяти. Поэтому, если на сервере имеются более медленные и более быстрые диски, то лучше разместить исполняемые файлы СУБД на медленных, а файлы данных на быстрых дисках.

Как показывает практика применения различных дисковых конфигураций, даже распределение файлов СУБД между несколькими физически различными одиночными дисками может поднять производительность серверной системы в 2 - 4 раза. Применение RAID массивов соответствующих типов или твердотельных дисков также является предпочтительным способом улучшения производительности.

В случае если на компьютере достаточно оперативной памяти, может быть организован также RAM - диск, и на него могут быть перенесены те файлы, которые могут быть потеряны без потери значимой информации (различного рода временные файлы).

7.11 Память и сборка мусора

Одним из важных практических вопросов применения MUMPS систем является отношение к памяти, принятое в системах такого класса и исторически сложившиеся традиции или поведение систем, наиболее ожидаемое разработчиками.

MUMPS системы по своей организации относятся к серверным системам, или, другими словами, к системам серверного класса. При этом они выполняют как задачи сервера баз данных, так и сервера приложений. К ключевым требованиям систем серверного класса относится гарантированное обслуживание заданного числа процессов и выполнение задач в прогнозируемое время.

Для этого система должна обслуживать задачи на ограниченных ресурсах как в целом, так и для каждого из выполняемых процессов. К ограничению ресурсов относится использование файлов, сокетов, портов, и оперативной памяти.

В случае, если система для выполнения задачи какого-либо из процессов начнет захватывать нерегламентированное или непредусмотренное поставленной задачей количество ресурсов, это может приводить к непредсказуемому переходу к подкачке с диска, что негативно сказывается на быстродействии как текущего, так и соседних процессов, и, зачастую, может быть признаком ошибок в исполняемой процессом программе или некорректного отношения к объемам данных.

В MUMPS системах, как и в других системах серверного класса, принято отводить на каждую из областей используемой памяти определенные пределы. При этом система, выполняя задачи, не выходит за эти пределы.

Выход за пределы расходования памяти может быть двух типов:

1. Процесс захватывает дополнительное пространство памяти для размещения данных.

2. Процесс захватывает дополнительное пространство памяти из-за фрагментации используемого пространства.

Оба случая MUMPS системы стремятся предотвратить и применяют улучшенные алгоритмы повторного использования памяти для снижения фрагментации, а также генерируют ошибку невозможности размещения дополнительных данных в случае исчерпания отведенных пределов.

Традиционно при работе с локальными переменными программист оценивает необходимый программе объем, и обычные переменные для

обработки данных по общему объему находятся в предсказуемых пределах. В большинстве случаев достаточно тестового прогона программы для того, чтобы убедиться в достаточности выделенной памяти для локальных переменных.

Кроме предсказуемого и прогнозируемого объема локальных переменных могут встречаться случаи использования локальных переменных для хранения временной копии данных, хранящихся в глобалах. В этом случае в локальные переменные могут попасть, вообще говоря, непредсказуемые объемы, зависящие от того, какие данные и какого объема оказались на текущий момент в базе данных.

В этом случае разработчики должны принять решение и оценить, являются ли используемые в локальных переменных данные (или могущие в них попасть при обработке) ограниченными по объему. В случае, если их объем может выходить за разумные для локальных переменных пределы, необходимо в качестве временных переменных использовать глобалы. В отличие от локальных переменных, MUMPS системы могут обрабатывать практически неограниченные, по сравнению с локальными переменными, объемы данных, размещенные в глобалах, из-за применения кеширования и подкачки блоков в кеш по необходимости. В случае с глобалами системе необходимо для одновременного использования лишь несколько блоков в кеше, в то время как для работы с локальными переменными процесс хранит их все в памяти одновременно.

Если разработчики обнаруживают, что административно установленные ими ограничения для локальных переменных недостаточны, пределы могут быть изменены. В некоторых реализациях MUMPS систем возможно программное управление объемом локальных переменных для запускаемого процесса, его можно указать в качестве параметра команды job.

Сами по себе MUMPS системы также применяют меры по ограничению внутренних пределов памяти для служебных целей, стремясь не выходить на неограниченное ее потребление.

По определению языка MUMPS нигде явно не указано использование указателей или иных структур, фиксирующих объекты языка в памяти. Поэтому, чисто технически, MUMPS системы могут применять внутри сборку мусора, хотя большинство систем такого механизма либо не используют либо используют весьма ограниченно. В частности, сборка мусора может быть применена к локальным переменным с ограничением видимости при покидании процессом этой области видимости, либо при удалении локальной переменной.

Механизм сборки мусора, или отложенного возврата использованных фрагментов памяти, может приводить к непредсказуемому измене-

нию производительности из-за срабатывания сборщика мусора и нерегламентированному приостанову выполнения процесса чтобы дождаться доступности памяти локальных переменных. В таких случаях может наблюдаться выполнение процесса некоторыми рывками.

Насколько известно автору, современные MUMPS системы таким механизмом либо не пользуются, либо используют весьма ограниченно и симптомы сборщика мусора, если он присутствует, на практике себя практически не проявляют. В случае если в используемой системе проявляются симптомы сборки мусора и его поведение начинает мешать, рекомендуется обратиться к документации на используемую MUMPS систему или в техподдержку и отрегулировать его поведение.

Кроме выполнения кода, написанного на языке MUMPS, сервера приложений или центры интеграции данных могут обращаться к дополнительным модулям, написанным на иных средствах разработки, в том числе содержащих встроенные сборщики мусора. В частности, такие среды исполнения как Java и .NET используют сборщик мусора изначально. При использовании таких систем в качестве динамических библиотек нужно понимать, что в них отношение к используемым пространствам памяти может отличаться от характерного для серверных систем. В частности, модули могут начать использовать неограниченное или все доступное пространство памяти, или внезапно начать расходовать процессор на работу сборщика мусора.

В случае если применяемый модуль, содержащий сборку мусора, начинает создавать нехарактерное для серверного поведения препятствие, необходимо обратиться к документации на используемые средства и попытаться принять меры к регулированию их поведения или заменить на предсказуемые модули. Во многих случаях, перед использованием в качестве среды модулей расширения систем со сборкой мусора, рекомендуется тщательно обдумать последствия такого шага и оценить возможность описать функционал модуля на языке MUMPS.

Первоначальные варианты MUMPS систем были ориентированы на работу в качестве серверов баз данных и приложений на весьма скромных по нынешним меркам ресурсах, как дисков, так и оперативной памяти. Благодаря продуманному отношению к ресурсам вычислительной системы такие сервера весьма уверенно обслуживали задачи, недоступные для систем других типов, работающих на той же аппаратуре. И в настоящее время, при сохранении аккуратного отношения к ресурсам, программные системы, работающие на MUMPS, продолжают характеризоваться как наименее проблемные и наиболее прогнозируемые в плане отношения к памяти и к времени отклика.

В каждой из MUMPS систем набор административных настроек пре-

делов ресурсов выполняется в зависимости от особенностей системы, но при этом практически все они имеют настройку на предел объема локальных переменных и кеша глобалов. Кроме таких настроек часто используется настройка объемов кеша байткода рутин и объем памяти для блокировок. Наличие других пределов использования служебных данных зависит от конкретной используемой MUMPS системы.

Общий необходимый объем памяти оценивается как суммарный из областей общего пользования (кеши глобалов, байткода, блокировки, и другие), и областей для каждого из процессов (локальные переменные, служебные данные) и количества выполняемых процессов. В случае использования внешних модулей необходимо также учитывать используемый ими объем памяти в пространстве каждого из процессов.

Исходя из поставленных задач, в большинстве случаев администратор может оценить раскладку выделяемой MUMPS системе оперативной памяти по отдельным областям или оценить необходимые аппаратные возможности, или оценить возможность запуска на этом же компьютере других серверов или приложений. Возможно, что иные приложения могут использовать другие стратегии расходования ресурсов, это также необходимо учитывать.

В частности, типичное клиентское приложение может захватывать непредсказуемые объемы памяти и руководствоваться иными правилами обработки данных. Например обрабатывать большие объемы в памяти, не используя подкачку и кеширование, как это делают СУБД. Что может приводить к непредсказуемому переходу множества процессов в свап и к резкому замедлению общей работы. Многие программы, написанные в стиле клиентского приложения, могут вообще не содержать обработки исчерпания ресурса или выхода за поддерживаемые пределы, и аварийно завершают работу при работе с данными, если их объем стал чуть больше чем тот, который они могут обработать в обычном стиле.

Еще одним фактором, который может повлиять на расчет памяти и величин отдельных фрагментов памяти сервера, является тип архитектуры MUMPS системы - является ли она многопроцессной или многопоточной. Как это можно установить - прочесть в документации или посмотреть на список запущенных процессов операционной системы. Если в списке запущенных процессов при еще одном запуске job MUMPS системы появляется еще один процесс - значит это многопроцессная система. Если появляется еще один поток (или несколько) у одного из процессов - то это многопоточная система.

Кроме них также могут существовать реализации с самостоятельным переключением контекста job-ов, не использующие ни потоки, ни процессы операционных систем. Для таких систем расчет памяти делается,

по сути, также, как и для многопоточных.

Различие многопроцессных и многопоточных систем в отношении к памяти состоит в том, что потоки живут внутри адресного пространства процесса и пользуются его единым адресным пространством, а различные процессы пользуются собственными адресными пространствами. В 32-битных системах, весьма распространенных ранее, ныне, и, весьма вероятно, некоторое время еще, общее пространство составляет 4 гигабайта. Из них 1 гигабайт отводится на область, управляемую автоматически операционной системой, где размещаются кодовые сегменты, как самого процесса так и служебных библиотек. 1 гигабайт по умолчанию не используется, но может быть использован для данных, если включить в операционной системе специальную опцию и отметить исполняемый файл специальным образом. Но, по умолчанию, тиражное и массовое программное обеспечение такими опциями не пользуется. И оставшиеся 2 гигабайта отдаются процессу на пространство данных. Обычная практика применения систем серверного класса состоит в том, что возможность увеличить что-либо в полтора раза обычно не является заметным практическим решением. Вот в этих пределах процесс сервера должен разместить вообще все данные, которыми он пользуется в памяти. Точнее говоря, может использовать только это адресное пространство.

Если MUMPS система выполнена в многопоточной схеме или в схеме с самостоятельным переключением контекста job, то все, то есть вообще все области данных, все кеши и все области локальных переменных, должны разместиться в пределах этих 2 гигабайт. В случае, например, если нужно отвести 8 мегабайт на локальные переменные для одного job, то 128 таких job уже должны занять пространство адресов 1 гигабайт. Если надо иметь возможность запустить больше job, то расход пространства соответственно увеличивается. Оставшееся пространство также должно быть занято под различного рода кеши. Таким образом, в случае применения многопоточной схемы администратор при росте числа выполняемых job должен либо сокращать области кешей, либо такой сервер не сможет в итоге обслужить больше чем определенное число job.

В случае если MUMPS система выполнена по многопроцессной схеме (а по такой схеме в настоящее время выполнены большинство современных MUMPS систем), то часть адресного пространства процесса каждого из job занимается отображенными на его пространство общими областями кешей, а локальные переменные и служебные данные занимают собственное пространство. Поэтому в отношении пределов по пространству многопроцессные системы не имеют естественного ограниче-

ния. Ограничение, если и есть, то аппаратное, это предел установленного объема оперативной памяти с добавленным файлом подкачки.

В случае применения 64-битной архитектуры такие различия между многопоточными и многопроцессными реализациями стираются из-за практически неограниченного 64-битного пространства. Хотя, если система изначально была разработана для того, чтобы работать и на 32-битной архитектуре, то ее 64-битный вариант не подвергается архитектурной переделке на многопоточную схему из-за отсутствия такой необходимости. Что интересно, встречаются разработки, выполненные на вполне современных интерпретирующих языках и успешно работающие на современных 64-битных системах, но содержащие проблемы при работе на 32-битных реализациях тех же интерпретаторов именно из-за отношения к объему памяти, доступному одному процессу.

Другой косвенной, но важной особенностью различия MUMPS систем, выполненных как многопроцессные, многопоточные и системы с самостоятельным переключением контекстов, является их работа с внешними модулями.

Если многопроцессная система вызывает внешний модуль, то он располагается в контексте только этого процесса и не затрагивает иные процессы и остальные процессы не зависят от того, передано ли управление внешнему модулю (dll или so).

Если многопоточная система вызывает внешний модуль, то он располагается в контексте всего пространства MUMPS системы и, возможно, могут существовать особенности его применения в многопоточной системе, например невозможность обратиться к нему из двух или более потоков. При этом, если поток обратился к внешнему модулю, то остальные потоки выполняются параллельно и не приостанавливают свою работу.

Если система с самостоятельным переключением контекста job обратилась к внешнему модулю, то могут возникнуть проблемы с инициализацией таких обращений из нескольких различных job, сможет ли такой внешний модуль отличить такие различные контексты, если это один и тот же процесс и один и тот же поток. Другой проблемой является то, что если один из job вызвал внешний модуль, то остальные job не могут выполняться, поскольку управление передано от системы самостоятельного переключения контекстов внешнему модулю.

Первые два случая, многопроцессный и многопоточный, используют вытесняющую многозадачность, а второй, с самостоятельным переключением контекстов job, кооперативную многозадачность.

В практическом отношении, для стабильного и предсказуемого обслуживания большого числа job наилучшим выбором видится вариант многопроцессной MUMPS системы.

7.12 Предсказуемость

При разработке прикладных программ на языке MUMPS программисты в большинстве случаев используют лишь примерно три десятка системных функций и системных переменных. Весь остальной код - это функции, состоящие из элементарных операций и действий.

При создании самой MUMPS системы её разработчикам необходимо обеспечить отлаженную и четкую работу именно небольшого количества функционала. Конечно, во внутренних механизмах любой MUMPS системы содержится множество технологических решений головоломок самого различного рода и очень много труда. При этом перед MUMPS системой ставится задача четко и предсказуемо выполнить определенный перечень действий. В этом MUMPS системы могут быть выделены среди систем управления базами данных других классов.

Элементарные действия, выполняемые MUMPS системами, не столь элементарны, как просто набор арифметических и строковых функций. Они содержат также полный набор функций обращения к хранимым данным, реализуя собственно СУБД, а не только несущий слой для сервера приложений.

Одновременно с этим, каждая из этих простых функций действительно может быть до конца отлажена и выполнена с высоким уровнем надежности и предсказуемостью поведения. Более того, с поведением, четко определенном стандартом на язык - все программы на стандартном множестве языка MUMPS выполняются одинаково.

В отличие от MUMPS систем, СУБД на основе автоматических процессоров запросов (SQL / QBE) содержат в себе в одном действии огромную функциональность. Фактически, даже для выполнения простого оператора INSERT СУБД запускает на выполнение большую, функциональную и сложную фабрику кода.

Если в SQL - based системах любое, даже простое действие с данными выполняется обращением к сложному действию, или простое рассматривается как частный случай более общего сложного, то в MUMPS СУБД сложные действия описываются как последовательность простых. При этом программист явным образом описывает, что именно необходимо сделать и каждый из использованных элементов, системные функции, работают с высоким уровнем надежности и предсказуемости.

При изменении требований к прикладной системе и к набору выполняемых ей операций в первом случае необходимо выяснить, является ли простое действие частным случаем какого-либо из поддерживаемых сервером сложных или как именно добавляемое действие необходимо разложить на составные, которые уже являются поддерживаемыми. В

случае с MUMPS системой разработчики априори уверены, что практически любая модернизация прикладной системы выполнима.

В действительности автор сталкивался с прикладными системами, разработанными на основе MUMPS систем, практически всех типов: от высоконагруженного OLTP до массивного OLAP со сложными статистическими функциями.

В целом, при эксплуатации, прикладные системы на основе MUMPS показывают себя обычно так, что у пользователей не на слуху название самого используемого сервера, а все операции они называют на своем, понятном им языке, например "пробить по базе", "ввести в базу" и т.д.

При необходимости подключения к системе еще одного или нескольких пользователей администраторы прикладных систем на основе MUMPS систем обычно не затрудняются в оценках, справится ли сервер или нет. Благодаря предсказуемости такие системы очень хорошо прогнозируемы по необходимому аппаратному обеспечению и администраторы не ждут от серверов внезапных и необъяснимых провалов производительности.

7.13 Переносимость

Одной из ключевых практических особенностей MUMPS систем является наличие соглашений о переносимости. В различных других языках и СУБД они также есть, и следование им обеспечивает долгую жизнь как прикладным программам, разработанным для работы на MUMPS, так и самим MUMPS системам. Кроме того, при обсуждении различных вопросов разработчики могут проиллюстрировать ситуацию фрагментом кода и быть уверенными, что их поймут в точности разработчики, использующие иные MUMPS системы или операционные системы, и у других разработчиков приведенный код будет исполняться в точности также.

В понятие переносимости в данном случае не входит портируемость собственно самих MUMPS систем, и описывает только факторы, которыми руководствуются разработчики прикладных программ.

Переносимость программ достигается тем, что разработчики этих программ принимают меры к применению в своей работе определенных рамок, или общего знаменателя возможностей, которые есть у различных MUMPS систем, соответствующих уровню переносимости. И прикладная программа, которая по своим запросам находится в договоренных пределах, имеет все шансы быть корректно перенесенной на иные MUMPS системы или работающие на иных операционных системах.

Стандарт на MUMPS как язык и среду исполнения программ сам по себе довольно жесткий, и включает требование отсутствия undefined behavior, или неопределенного поведения на усмотрение реализации. Вообще говоря, если MUMPS система разработана в соответствии со стандартом, то программы, написанные на MUMPS, уже имеют очень высокие шансы быть перенесенными на эту MUMPS систему. И наоборот, если программа написана на MUMPS, то имеет очень высокие шансы работать в точности так же и на самых разных MUMPS- и операционных системах.

Следование стандарту, однако, еще не гарантирует того факта, что программа сразу заработает на другой MUMPS системе, поскольку кроме языковых конструкций любая система использует определенные количественные, или измеримые, ограничения. В частности, ранее MUMPS разработчикам откровенно мешало ограничение на максимальную длину строки, принятое в 255, а позднее в 510 символов, или ограничение на количество строк в рутине. Прогресс не стоит на месте и разработчики MUMPS систем идут навстречу практическим потребностям прикладных программ, превышая определенные уровнем переносимости пределы.

Первоначально разработанные MUMPS системы были призваны обрабатывать информацию на компьютерах примерно 40-летней давности, имевшие, мягко говоря, очень скромные по современным меркам аппаратные возможности. В настоящее время аппаратные возможности типовых компьютеров намного больше. Сама запись программ в сокращенной форме по своей информационной емкости может быть в каких-то случаях даже более компактной, чем соответствующий порождаемый ей байткод.

Список измеримых ограничений, составляющих требования переносимости MUMPS программ, составляют вторую часть стандарта на язык MUMPS. Эти требования описывают минимальные возможности как для собственно самих MUMPS систем, так и максимальные пределы, внутри которых должны оставаться прикладные программы, чтобы соответствовать уровню переносимости.

Важно отметить, что пределы даются не в байтах, а в символах. Соответствие одного символа числу бит (или байт) определяется MUMPS системой. Кроме того, стандарт уточняет, что могут указываться как явные, так и неявные, или побочные, ограничения. Например, длина командной строки неявно ограничивает длину языковой конструкции, которая может быть использована в качестве исключительно одного аргумента команды.

Список требований переносимости стандарта, действующего на текущий момент для языка MUMPS, таков:

Вид ограничения	Требования
Набор символов	Набор символов, который должен использоваться в рутинах и в данных, ограничен наборами символов ASCII, JIS90, ISO-8859-UAA, ISO-8859-1-USA/M (должен использоваться один из них).
Имена локальных переменных	Использование идентификаторов в именах ограничено символами в верхнем регистре. На полную длину индентификатора ограничения нет, но распознаются только первые 8 символов идентификатора. Это накладывает неявное ограничение на общее количество уникальных имен.
Количество локальных переменных	Явного ограничения на количество локальных переменных нет, но есть неявное ограничение, исходя из ограничения на общий объем для хранения локальных переменных.
Число индексов локальной переменной	Нет явного ограничения на число различных узлов локальной переменной, но есть неявное ограничение на число индексов, которые могут быть использованы для произвольной локальной переменной. Сумма длин всех индексов плюс удвоенное количество индексов плюс длина имени плюс 15 не должны превышать максимально допустимую длину строки.
Значения индексов	Индексы должны быть непустыми строками, состоящими только из печатных символов. Явного ограничения на длину одного индекса нет, но есть неявное ограничение на общую длину полного имени (см. ранее). Если значение индекса соответствует числовому значению, то число должно находиться в определенных пределах. Использование значений индексов, не соответствующих этим критериям, не определено, за исключением пустой строки в качестве начального и конечного значения функциями $ORDER и $QUERY.
Число глобальных переменных	Ограничение на общее число различных имен глобальных переменных явно не накладывается.

Число индексов глобальной переменнной	Длина имени базы данных плюс 3 плюс длина имени глобала плюс длина всех индексов плюс удвоенное число индексов плюс 15 не должно превышать допустимую длину строки.
Количество узлов глобальной переменной	Нет явного ограничения на общее число узлов глобальной переменной.
Тип данных	Стандарт определяет только один тип данных, строка переменной длины. В зависимости от контекста, в котором требуется использование значений в качестве чисел, целых чисел, булевских значений, интерпретация строк определяется соответствующими правилами отображения строк на числа, целые числа и булевские значения. MUMPS системы никак не ограничены порядком внутреннего использования и представления данных.
Точность чисел	Все числа, используемые в арифметических операциях, или в любом другом контексте, требующем приведения к числу, должны находиться в пределах $[-10^{25}, -10^{-25}]$ или $[10^{-25}, 10^{25}]$, или быть нулем. Точность представления чисел как минимум с 15 значащими знаками. Относительная ошибка вычисления в операциях сложения, вычитания, умножения, деления, целочисленного деления или деления по модулю не должна превышать 10^{15}. Относительная ошибка возведения в степень не должна превышать 10^7.
Целые числа	Точность операций с целыми числами ограничивается точностью операций с числами (см. ранее).
Строки	Ограничения на строки состоят из строгих и нестрогих. Строгие: длина строки ограничена 255 символов и допускаются только символы из набора ASCII. В нестрогие ограничения (рекомендации к применению) входит: длина локальных и ssvn ограничена 32767, а глобальных 510 символов, все символы также должны быть из набора ASCII (ANSI X3.4–1986).

Специальные переменные	Переменные $X и $Y должны быть неотрицательными целыми числами. Увеличение значения $X и $Y за пределы максимально выполнимых приводит к их неопределенному значению. Значение переменной $SYSTEM должно соответствовать ограничениям на индексы локальных переменных.
Вложенность выражений	Явного ограничения на глубину вложенности выражений нет. Неявное ограничение содержится в ограничении на длину строки, содержащую это выражение.
Результат вычисления	Ошибочным должен считаться любой результат, не соответствующий ограничениям на длину строки. Числовые и целочисленные результаты вычислений должны считаться ошибочными если они не соответствуют диапазону допустимых чисел.
Внешные ссылки	Внешние ссылки не переносимы и в переносимых программах не должны использоваться. Внешние ссылки это ссылки специального вида на внешние модули, именуемые именами специального вида, начинающимися на символ амперсанд (&Package.Function).
Строка рутины	Длина строки рутины должна соответствовать ограничению на длину строки (см. ранее). Длина вычисляется не включая символ конца строки. Символы строки рутины ограничены печатными символами из набора ASCII.
Число строк в рутине	Явные ограничения не накладываются, неявные определяются ограничениями на объем хранения рутины.
Число команд в строке	Явного ограничения на число команд в строке не накладывается, неявное ограничение вытекает из ограничения на длину строки рутины.
Метки	Имена меток, как в строковом так и в числовом варианте, должны удовлетворять требованиям на имена локальных переменных (см. ранее) по ограничению длины и использованию печатных символов.

Число меток	Явных ограничений на число меток в рутине не накладывается. Неявные ограничения: 1) строка может иметь не более одной метки и 2) любые две строки рутины должны иметь различные метки.
Число рутин	Явного ограничения на число рутин не накладывается, неявное ограничение вытекает из ограничения на длину имени (см. ранее), совпадающее с ограничением на длину имени локальной переменной.
Косвенность	Значения аргументов косвенности и команды XECUTE должны удовлетворять ограничениям на длину строки и по набору символов соответствовать ограничениям на командную строку.
Ограничения на объемы хранения	Размер одной рутины не должен превышать 10000 символов. Общий объем рутины вычисляется как сумма объемов всех ее строк. Объем строки вычисляется как длина строки плюс 2. Размер пространства локальных переменных не должен превышать 10000 символов. Объем определяется как сумма объемов всех локальных переменных безотносительно области видимости определяемой командой NEW.
Объем локальной переменной	Вычисляется как длина имени переменной плюс длина ее значения плюс 4.
Объем локального массива	Вычисляется как длина имени массива плюс сумма длин всех значений плюс длина каждого индекса плюс 2 для каждого узла, для которого значение $DATA равно 10 или 11.
Стек	Система обеспечивает минимум 127 уровней стека исполнения процесса. Неявное ограничение может быть вызвано ограничением хранения.
Обработка транзакций	Общая длина имен и значений глобальных переменных, изменяемых в одной транзакции, не должно превышать 57343 символов.
Вложения транзакций	Одна транзакция не может содержать более чем 126 вложенных транзакций.

Другие требования	Нецелые значения таймаутов в команде HANG и в командах с таймаутами могут приводить к неопределенному времени выполнения ожидания.

Последнее из указанных требований неявно вводит требование использовать таймауты в виде целого числа секунд, если требуется соответствие уровню переносимости.

Описанные требования ориентированы на применение абстрактной MUMPS системы таким образом, чтобы разрабатываемые MUMPS программы могли быть перенесены как на одну из имеющихся систем, так и на разработанную в будущем и не определенную в настоящем времени.

Последняя редакция стандарта датируется 1995-м годом и относится к стандарту ANSI. В настоящее время этот стандарт перестал действовать юридически, но его полное содержание входит в текущий действующий ISO стандарт на MUMPS системы. Его разработкой и дополнениями занимается комитет MDC (Mumps Development Committee).

С последних изменений прошло много лет и в настоящее время комитет рассматривает вопрос о возможном изменении стандарта и уровня переносимости не в виде описания абстрактных требований, а в виде описания общего знаменателя имеющихся возможностей существующих и применяемых в настоящее время MUMPS систем. Такой перечень мог бы составить более практический список требований реальной переносимости программ и данных.

В большинстве случаев именно такими неявными правилами и пользуются разработчики программ в практической работе. Выбирается набор MUMPS систем, на которые планируется перенос программ и составляется общий перечень ограничений, при соблюдении которых программы будут работоспособны на всех них.

Приведенный список требований переносимости в таких случаях служит основой для определения ограничений, и к нему добавляются уточненные ограничения. Кроме этого списка определяется набор функций и технологий, которые можно использовать. Например, при планировании переноса программы с Caché на MSM нельзя использовать классы, а при обратном переносе нельзя использовать слишком большие рутины.

Нюансы поведения различных MUMPS систем могут проявиться, в частности, и в характере исчисления таймаута. Возможно различное поведение системы:

1. Таймаут исчисляется от начала выполнения команды на весь указанный период.

2. Окончание таймаута округляется до целого числа секунд.

3. Окончание таймаута округляется до ближайшего окончания кванта переключения контекста job.

Во многих случаях разработчики проверяют используемые целевыми MUMPS системами ограничения и применяют стандартные возможности языка плюс дополнительные расширенные функции, поддерживаемые различными реализациями. В тех случаях, когда расширенные функции выполняются на различных MUMPS системах по-разному, разработчики применяют изолирование такого кода таким образом, чтобы при исполнении программы выполнялся код, соответствующий текущей MUMPS системе.

К таким способам могут быть отнесены:

1. Комбинирование кода и управление выполнением командами IF или функцией $SELECT.

2. Различная трансляция с использованием препроцессора и подстановкой соответствующего кода.

3. Использование различных рутин для различных MUMPS систем.

4. Помещение синтаксически зависимого от системы кода в аргумент команды XECUTE.

5. Помещение зависимого от системы кода в глобалы и выполнение его при необходимости командой XECUTE.

В случае необходимости переноса программ из числа уже написанных, разработчики обращают внимание на использование элементов, которые могут зависеть от реализации MUMPS системы или даже от ее версии:

1. Применение функций, не являющихся $Z функциями, например $INCREMENT, $LIST, $BIT.

2. Применение фунции $VIEW.

3. Применение $Z функций.

4. Применение Z команд.

5. Применение $Z системных переменных.

6. Применение команды VIEW.
7. Применение команды PRINT.
8. Применение команды BREAK.
9. Использование устройств, соглашений об их именовании и параметров.
10. Синтаксические расширения стандартных команд, например применение опции "S" у команды lock в Caché.
11. Применение свободного синтаксиса, новых команд циклов и фигурных скобок в новых версиях Caché.
12. Применение классов.
13. Использование устройств с одинаковыми именами различными процессами, принадлежат ли устройства открытые процессом, именно этому процессу или всему серверу.
14. Сколько одновременно может быть открыто устройств одним процессом и одновременно всеми процессами сервера.
15. Ограничение на предельный объем рутины.
16. Ограничение на предельный объем байткода.
17. Применение транзакций.
18. Применение команды TRESTART.
19. Применение системной переменной $TRESTART.
20. Присваивание системной переменной $REFERENCE.
21. Применение расширенных особенностей препроцессора.
22. Применение внешних DLL.
23. Применение обработки ошибок по ZTRAP и комбинирование с обработкой ошибок по ETRAP.
24. Применение расширенных структурных системных переменных.

Как показала практика применения MUMPS систем, практически любая из них может быть использована для выполнения крупнейших и сложнейших приложений, если разработчики принимают меры к соблюдению требований переносимости. Одновременно с тем, такие разработчики могут легко заменить одну MUMPS систему на другую при соответствующей замене системно-зависимого кода, либо на систему другой версии, либо работающую на другой операционной системе.

Приложение A

Команды MUMPS

Список команд языка MUMPS, входящих в стандарт ANSI X11.1-1995 и ISO MUMPS.

Команда	**Действие**
BREAK	Переход в отладочный режим, поведение и спецификация аргументов определяются реализацией.
CLOSE	Закрывает указаное устройство ввода-вывода.
DO	Передает управление подпрограмме или блоку строк.
ELSE	Выполняет строку в зависимости от условия $TEST.
FOR	Выполняет цикл.
GOTO	Передает управление на другую метку.
HALT	Останавливает выполнение процесса.
HANG	Приостанавливает выполнение процесса на указанное время.
IF	Выполняет строку в зависимости от значения аргумента или условия $TEST.
JOB	Запускает новый процесс.
KILL	Удаляет переменную.
LOCK	Устанавливает или отпускает блокировку.
MERGE	Копирует переменную с подиндексами.
NEW	Создает новую область видимости переменной.
OPEN	Открывает устройство ввода-вывода.
QUIT	Возврат из подпрограммы.
READ	Читает из текущего устройства строку или символ.
SET	Присваивает переменной или левосторонней функции.
TCOMMIT	Завершает транзакцию успехом.
TRESTART	Повторяет транзакцию сначала.
TROLLBACK	Откатывает транзакцию, завершение неуспехом.

TSTART	Начинает новую транзацкию.
USE	Делает устройство ввода-вывода текущим, передает устройству параметры.
VIEW	Доступ к внутренним функциям и структурам данных, поведение и спецификация аргументов определяются реализацией.
WRITE	Вывод в текущее устройство значений, управление форматом.
XECUTE	Выполнение аргумента как последовательности команд.

Приложение B

Системные переменные MUMPS

Список системных переменных языка MUMPS, входящих в стандарт ANSI X11.1-1995 и ISO MUMPS.

Переменная	Значение
$DEVICE	Статус текущего устройства, состояние ошибки ввода-вывода.
$ECODE	Значение последних ошибок процесса.
$ESTACK	Номер уровня стека от послденего изменения его области видимости.
$ETRAP	Код - обработчик ошибок.
$HOROLOG	Значение текущего локального времении даты.
$IO	Текущее устройство ввода-вывода.
$JOB	Номер текущего процесса.
$KEY	Управляющая последовательность, завершившая последнее чтение из текущего устройства.
$PRINCIPAL	Устройство ввода-вывода по умолчанию.
$QUIT	Индикатор необходим ли возврат значения из функции или только выход из подпрограммы.
$STACK	Уровень стека.
$STORAGE	Объем памяти доступный локальным переменным.
$SYSTEM	Идентификатор текущего выполняющегося экземпляра MUMPS системы.
$TEST	Значение условия последней команды IF или команды с таймаутом (OPEN, LOCK, JOB, READ).
$TLEVEL	Уровень транзакции.
$TRESTART	Число рестартов транзакции.
$X	Позиция каретки по горизонтали на терминальном устройстве.

$Y	Позиция каретки по вертикали на терминальном устройстве.

Приложение C

Системные функции MUMPS

Список системных функций языка MUMPS, входящих в стандарт ANSI X11.1-1995 и ISO MUMPS.

Функция	**Действие**
$ASCII	Возвращает код символа в указанной позиции строки.
$CHAR	Возвращает строку образованную из указанных кодов символов.
$DATA	Возвращает индикатор существования переменной и ее дочерних.
$EXTRACT	Возвращает подстроку с заданного места.
$FIND	Возвращает позицию после найденного фрагмента.
$FNUMBER	Форматирует число.
$GET	Возвращает значение переменной или значение по умолчанию если переменная не определена.
$JUSTIFY	Возвращает дополненную до нужной длины строку.
$LENGTH	Возвращает длину строки или длину строки с разделителями.
$NAME	Возвращает строку с именем переменной.
$ORDER	Возвращает следующий существующий у переменной индекс.
$PIECE	Возвращает часть строки с разделителями.
$QLENGTH	Возвращает число индексов в имени.
$QSUBSCRIPT	Возвращает значение части имени, индекс.
$QUERY	Возвращает следующее имя переменной с индексами.
$RANDOM	Возвращает псевдослучайное число в указанном диапазоне.
$REVERSE	Возвращает строку в обратном порядке.

$SELECT	Возвращает значение в зависимсоти от серии условий.
$STACK	Возвращает информацию о состоянии стека исполнения.
$TEXT	Возвращает строку текста рутины.
$TRANSLATE	Замена символов по указанным таблицам.
$VIEW	Доступ к внутренним функциям и структурам данных, поведение и спецификация аргументов определяются реализацией.

Приложение D

Стандартные коды ошибок

Текущий стандарт языка MUMPS предусматривает перечень ситуаций, для которых определены стандартные коды ошибок. Ошибки с кодами MDC (MUMPS Development Committee) при возникновении ситуаций, которые они описывают, записываются в системную переменную $ecode через запятую с предшествующей буквой M, что означает ошибку, описанную в стандарте языка.

Например, при обращении к переменной с неопределенным значением код ошибки равен M6:

```
USER>w und12345

<UNDEFINED> *und12345

USER>w $ec
,M6,
```

M1	Naked indicator undefined.
M2	Invalid $FNumber code string combination.
M3	$Random argument less than 1.
M4	No true condition in $Select.
M5	Line reference less than 0 (zero).
M6	Undefined local variable.
M7	Undefined global variable.
M8	Undefined special variable.
M9	Divide by zero.
M10	Invalid pattern match range.
M11	No parameters passed.
M12	Invalid line reference (negative offset).
M13	Invalid line reference (line not found).

M14 Line level not one (1).
M15 Undefined index variable.
M16 Quit with an argument not allowed.
M17 Quit with an argument required.
M18 Fixed-length Read not greater than 0 (zero).
M19 Cannot merge a tree or subtree into itself.
M20 Line must have a formal list.
M21 Formal list name duplication.
M22 Set or Kill to ^$Global structured system variable name (SSVN) when data in global.
M23 Set or Kill to ^$Job structured system variable name (SSVN) for non-existent job number.
M24 Change to collation algorithm while subscripted local variables defined.
M26 Non-existent environment (non-existent namespace).
M27 Attempt to roll back a transaction that is not re-startable.
M28 Mathematical function, parameter out of range.
M29 Set or Kill on structured system variable name (SSVN) not allowed by implementation.
M30 Reference to global variable with different collating sequence within a collating algorithm.
M31 Device control mnemonic expression used for a device without a mnemonic space being selected.
M32 Device control mnemonic used in user-defined mnemonic space which has no associated line.
M33 Set or Kill to ^$Routine when the routine specified exists.
M35 Device does not support mnemonic spaces.
M36 Incompatible mnemonic spaces.
M37 Read from device identified by null string.
M38 Invalid structured system variable name (SSVN) subscript.
M39 Invalid $Name argument.
M40 Call by reference in the actual parameter list in Job command.
M41 Invalid Lock argument within a transaction.
M42 Invalid Quit within a transaction.
M43 Invalid range value ($X,$Y).
M44 Invalid command outside a transaction.
M45 Invalid Goto reference.
M57 A label is defined more than once in a routine.
M58 Too few formal parameters.

Приложение E

ASCII Table

Список кодов символов ASCII с указанием кодов шаблонов (patcode), как они используются в MUMPS:

Decimal	Character	Patcode
0	NUL	C,E
1	SOH	C,E
2	STX	C,E
3	ETX	C,E
4	EOT	C,E
5	ENQ	C,E
6	ACK	C,E
7	BELL	C,E
8	BS	C,E
9	HT	C,E
10	LF	C,E
11	VT	C,E
12	FF	C,E
13	CR	C,E
14	SO	C,E
15	SI	C,E
16	DLE	C,E
17	DC1	C,E
18	DC2	C,E
19	DC3	C,E
20	DC4	C,E
21	NAK	C,E
22	SYN	C,E
23	ETB	C,E

24	CAN	C,E
25	EM	C,E
26	SUB	C,E
27	ESC	C,E
28	FS	C,E
29	GS	C,E
30	RS	C,E
31	US	C,E
32	SP (space)	P,E
33	!	P,E
34	"	P,E
35	#	P,E
36	$	P,E
37	%	P,E
38	&	P,E
39	' (apostrophe)	P,E
40	(	P,E
41	)	P,E
42	*	P,E
43	+	P,E
44	, (comma)	P,E
44	- (hyphen)	P,E
46	.	P,E
47	/	P,E
48	0	N,E
49	1	N,E
50	2	N,E
51	3	N,E
52	4	N,E
53	5	N,E
54	6	N,E
55	7	N,E
56	8	N,E
57	9	N,E
58	:	P,E
59	;	P,E
60	<	P,E
61	=	P,E
62	>	P,E
63	?	P,E
64	@	P,E

65	A	A,U,E
66	B	A,U,E
67	C	A,U,E
68	D	A,U,E
69	E	A,U,E
70	F	A,U,E
71	G	A,U,E
72	H	A,U,E
73	I	A,U,E
74	J	A,U,E
75	K	A,U,E
76	L	A,U,E
77	M	A,U,E
78	N	A,U,E
79	O	A,U,E
80	P	A,U,E
81	Q	A,U,E
82	R	A,U,E
83	S	A,U,E
84	T	A,U,E
85	U	A,U,E
86	V	A,U,E
87	W	A,U,E
88	X	A,U,E
89	Y	A,U,E
90	Z	A,U,E
91	[	P,E
92	\	P,E
93	]	P,E
94	^	P,E
95	_ (underscore)	P,E
96	`	P,E
97	a	A,L,E
98	b	A,L,E
99	c	A,L,E
100	d	A,L,E
101	e	A,L,E
102	f	A,L,E
103	g	A,L,E
104	h	A,L,E
105	i	A,L,E

106	j	A,L,E
107	k	A,L,E
108	l	A,L,E
109	m	A,L,E
110	n	A,L,E
111	o	A,L,E
112	p	A,L,E
113	q	A,L,E
114	r	A,L,E
115	s	A,L,E
116	t	A,L,E
117	u	A,L,E
118	v	A,L,E
119	w	A,L,E
120	x	A,L,E
121	y	A,L,E
122	z	A,L,E
123	{	P,E
124	|	P,E
125	}	P,E
126	~(tilde)	P,E
127	DEL	C,E

Приложение F

Мнемоники ANSI X3.64

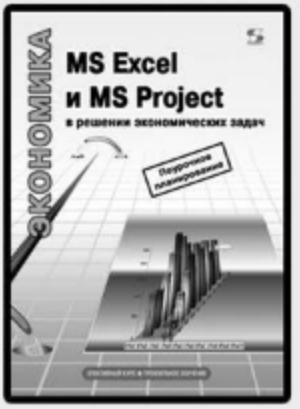
Calling Sequence	Name
APC	Application Program Command
BEL	Ring the bell
CBT(%1)	Cursor Backward Tabulation
CCH	Cancel Character
CHA(%1)	Cursor Horizontal Absolute
CHT(%1)	Cursor Horizontal Tabulation
CNL(%1)	Cursor Next Line
CPL(%1)	Cursor Preceding Line
CPR	Cursor Position Report
CTC(%1,%2,%3,%4,%5,%6,%7,%8,%9)	Cursor Tabulation Control
CUB(%1)	Cursor Backward
CUD(%1)	Cursor Down
CUF(%1)	Cursor Forward
CUP(%1,%2)	Cursor Position
CUU(%1)	Cursor Up
CVT(%1)	Cursor Vertical Tabulation
DA	Device Attributes
DAQ(%1,%2,%3,%4,%5,%6,%7,%8,%9)	Define Area Qualification
DCH(%1)	Delete Characters
DCS	Device Control String
DL(%1)	Delete Lines
DMI	Disable Manual Input
DSR(%1)	Device Status Report
EA(%1)	Erase in Area
ECH(%1)	Erase Characters
ED(%1)	Erase in Display
EF(%1)	Erase in Field

EL(%1)	Erase in Line
EMI	Enable Manual Input
EPA	End of Protected Area
ESA	End of Selected Area
FNT	Font Selection
GSM	Graphic Size Modification
GSS	Graphic Size Selection
HPA(%1)	Horizontal Position Attribute
HPR(%1)	Horizontal Position Relative
HTJ	Horizontal Tab with Justify
HTS	Horizontal Tab Set
HVP(%1,%2)	Horizontal and vertical position
ICH(%1)	Insert Characters
IL(%1)	Insert Lines
IND	Index
INT	Interrupt
JFY	Justify
MC	Media Copy
MW	Message Waiting
NEL	Next Line
NP(%1)	Next Page
OSC	Operating System Command
PLD	Partial Line Down
PLU	Partial Line Up
PM	Privacy Message
PP(%1)	Preceding Page
PU1	Private Use 1
PU2	Private Use 2
QUAD	Quad
REP(%1)	Repeat
RI	Reverse Index
RIS	Reset to Initial State
RM(%1,%2,%3,%4,%5,%6,%7,%8,%9)	Reset Mode
SEM	Select Editing Extent Mode
SGR(%1,%2,%3,%4,%5,%6,%7,%8,%9)	Select Graphic Rendition
SL	Scroll Left
SM(%1,%2,%3,%4,%5,%6,%7,%8,%9)	Set Mode
SPA	Start of Protected Area
SPI	Spacing Increment
SR	Scroll Right
SS2	Single Shift Two

SS3	Single Shift Three
SSA	Start of Selected Area
ST	String Terminator
STS	Set Transmit State
SU	Scroll Up
TBC	Tabulation Clear
TSS	Thin Space Specification
VPA(%1)	Vertical Position Attribute
VPR(%1)	Vertical Position Relative
VTS	Vertical Tab Set

Издательство «СОЛОН-ПРЕСС» представляет
комплект пособий по информатике и информационным технологиям
С ПОУРОЧНЫМ ПЛАНИРОВАНИЕМ

для элективного курса, профильного обучения и подготовки к ЕГЭ



КАК КУПИТЬ КНИГИ

Заказ оформляется одним из двух способов:
1) Пошлите открытку или письмо по адресу 123001, Москва, а/я 82.
2) Оформите заказ на сайте www.solon-press.ru в разделе «Книга-почтой»
или «Интернет-магазин».
Бесплатно высылается каталог издательства по почте. При оформлении заказа полностью укажите адрес, фамилию, имя и отчество получателя. Желательно указать дополнительно телефон и адрес электронной почты. Каталог издательства можно получить по ссылке http://www.solon-press.ru/kat.doc
Телефон: (499) 254-44-10, (499) 252-36-96, (499) 252-25-21, e-mail: tr@coba.ru, avtor@coba.ru
Оптовые закупки у официального дистрибьютора «АЛЬЯНС-КНИГА» (www.alians-kniga.ru)

Приложение G

ANSI Escape Sequences (ENG)

These sequences define functions that change display graphics, control cursor movement, and reassign keys.

ANSI escape sequence is a sequence of ASCII characters, the first two of which are the ASCII "Escape" character 27 (1Bh) and the left-bracket character "[" (5Bh). The character or characters following the escape and left-bracket characters specify an alphanumeric code that controls a keyboard or display function.

ANSI escape sequences distinguish between uppercase and lowercase letters.

Esc**[**Line;Column**H** Esc**[**Line;Column**f**	**Cursor Position:** Moves the cursor to the specified position (coordinates). If you do not specify a position, the cursor moves to the home position at the upper-left corner of the screen (line 0, column 0). This escape sequence works the same way as the following Cursor Position escape sequence.
Esc**[**Value**A**	**Cursor Up:** Moves the cursor up by the specified number of lines without changing columns. If the cursor is already on the top line, ANSI.SYS ignores this sequence.
Esc**[**Value**B**	**Cursor Down:** Moves the cursor down by the specified number of lines without changing columns. If the cursor is already on the bottom line, ANSI.SYS ignores this sequence.

Esc[Value**C** **Cursor Forward:**
Moves the cursor forward by the specified number of columns without changing lines. If the cursor is already in the rightmost column, ANSI.SYS ignores this sequence.

Esc[Value**D** **Cursor Backward:**
Moves the cursor back by the specified number of columns without changing lines. If the cursor is already in the leftmost column, ANSI.SYS ignores this sequence.

Esc[**s** **Save Cursor Position:**
Saves the current cursor position. You can move the cursor to the saved cursor position by using the Restore Cursor Position sequence.

Esc[**u** **Restore Cursor Position:**
Returns the cursor to the position stored by the Save Cursor Position sequence.

Esc[**2J** **Erase Display:**
Clears the screen and moves the cursor to the home position (line 0, column 0).

Esc[**K** **Erase Line:**
Clears all characters from the cursor position to the end of the line (including the character at the cursor position).

Esc[Value;...;Value**m** **Set Graphics Mode:**
Calls the graphics functions specified by the following values. These specified functions remain active until the next occurrence of this escape sequence. Graphics mode changes the colors and attributes of text (such as bold and underline) displayed on the screen.

Text attributes

0	All attributes off
1	Bold on
4	Underscore (on monochrome display adapter only)
5	Blink on
7	Reverse video on
8	Concealed on

Foreground colors

30	Black
31	Red
32	Green
33	Yellow
34	Blue
35	Magenta
36	Cyan
37	White

Background colors

40	Black
41	Red
42	Green
43	Yellow
44	Blue
45	Magenta
46	Cyan
47	White

Parameters 30 through 47 meet the ISO 6429 standard.

Esc**[**=Value**h** **Set Mode:**
Changes the screen width or type to the mode specified by one of the following values:

Screen resolution

0	40 x 25 monochrome (text)
1	40 x 25 color (text)
2	80 x 25 monochrome (text)
3	80 x 25 color (text)
4	320 x 200 4-color (graphics)
5	320 x 200 monochrome (graphics)
6	640 x 200 monochrome (graphics)
7	Enables line wrapping
13	320 x 200 color (graphics)
14	640 x 200 color (16-color graphics)
15	640 x 350 monochrome (2-color graphics)

16 640 x 350 color (16-color graphics)
17 640 x 480 monochrome (2-color graphics)
18 640 x 480 color (16-color graphics)
19 320 x 200 color (256-color graphics)

Esc[=Value**l**

Reset Mode:
Resets the mode by using the same values that Set Mode uses, except for 7, which disables line wrapping (the last character in this escape sequence is a lowercase L).

Esc[Code;String;...**p**

Set Keyboard Strings:
Redefines a keyboard key to a specified string.
The parameters for this escape sequence are defined as follows:
Code is one or more of the values listed in the following table. These values represent keyboard keys and key combinations. When using these values in a command, you must type the semicolons shown in this table in addition to the semicolons required by the escape sequence. The codes in parentheses are not available on some keyboards. ANSI.SYS will not interpret the codes in parentheses for those keyboards unless you specify the /X switch in the DEVICE command for ANSI.SYS.
String is either the ASCII code for a single character or a string contained in quotation marks. For example, both 65 and "A" can be used to represent an uppercase A.
IMPORTANT: Some of the values in the following table are not valid for all computers. Check your computer's documentation for values that are different.

Key	**Code**	**SHIFT+ Key**	**CTRL+ Key**	**ALT+ Key**
F1	0;59	0;84	0;94	0;104
F2	0;60	0;85	0;95	0;105
F3	0;61	0;86	0;96	0;106
F4	0;62	0;87	0;97	0;107

F5	0;63	0;88	0;98	0;108
F6	0;64	0;89	0;99	0;109
F7	0;65	0;90	0;100	0;110
F8	0;66	0;91	0;101	0;111
F9	0;67	0;92	0;102	0;112
F10	0;68	0;93	0;103	0;113
F11	0;133	0;135	0;137	0;139
F12	0;134	0;136	0;138	0;140
HOME (num keypad)	0;71	55	0;119	–
UP ARROW (num keypad)	0;72	56	(0;141)	–
PAGE UP (num keypad)	0;73	57	0;132	–
LEFT ARROW (num keypad)	0;75	52	0;115	–
RIGHT ARROW (num keypad)	0;77	54	0;116	–
END (num keypad)	0;79	49	0;117	–
DOWN ARROW (num keypad)	0;80	50	(0;145)	–
PAGE DOWN (num keypad)	0;81	51	0;118	–
INSERT (num keypad)	0;82	48	(0;146)	–
DELETE (num keypad)	0;83	46	(0;147)	–
HOME	(224;71)	(224;71)	(224;119)	(224;151)
UP ARROW	(224;72)	(224;72)	(224;141)	(224;152)
PAGE UP	(224;73)	(224;73)	(224;132)	(224;153)
LEFT ARROW	(224;75)	(224;75)	(224;115)	(224;155)
RIGHT ARROW	(224;77)	(224;77)	(224;116)	(224;157)
END	(224;79)	(224;79)	(224;117)	(224;159)
DOWN ARROW	(224;80)	(224;80)	(224;145)	(224;154)
PAGE DOWN	(224;81)	(224;81)	(224;118)	(224;161)
INSERT	(224;82)	(224;82)	(224;146)	(224;162)

DELETE	(224;83)	(224;83)	(224;147)	(224;163)
PRINT SCREEN	–	–	0;114	–
PAUSE/BREAK	–	–	0;0	–
BACKSPACE	8	8	127	(0)
ENTER	13	–	10	(0
TAB	9	0;15	(0;148)	(0;165)
NULL	0;3	–	–	–
A	97	65	1	0;30
B	98	66	2	0;48
C	99	66	3	0;46
D	100	68	4	0;32
E	101	69	5	0;18
F	102	70	6	0;33
G	103	71	7	0;34
H	104	72	8	0;35
I	105	73	9	0;23
J	106	74	10	0;36
K	107	75	11	0;37
L	108	76	12	0;38
M	109	77	13	0;50
N	110	78	14	0;49
O	111	79	15	0;24
P	112	80	16	0;25
Q	113	81	17	0;16
R	114	82	18	0;19
S	115	83	19	0;31
T	116	84	20	0;20
U	117	85	21	0;22
V	118	86	22	0;47
W	119	87	23	0;17
X	120	88	24	0;45
Y	121	89	25	0;21
Z	122	90	26	0;44
1	49	33	–	0;120
2	50	64	0	0;121
3	51	35	–	0;122
4	52	36	–	0;123
5	53	37	–	0;124
6	54	94	30	0;125
7	55	38	–	0;126

8	56	42	–	0;126
9	57	40	–	0;127
0	48	41	–	0;129
-	45	95	31	0;130
=	61	43	—	0;131
[	91	123	27	0;26
]	93	125	29	0;27
	92	124	28	0;43
;	59	58	–	0;39
'	39	34	–	0;40
,	44	60	–	0;51
.	46	62	–	0;52
/	47	63	–	0;53
`	96	126	–	(0;41)
ENTER (keypad)	13	–	10	(0;166)
/ (keypad)	47	47	(0;142)	(0;74)
(keypad)	42	(0;144)	(0;78)	–
- (keypad)	45	45	(0;149)	(0;164)
+ (keypad)	43	43	(0;150)	(0;55)
5 (keypad)	(0;76)	53	(0;143)	–

Издательство СОЛОН-ПРЕСС представляет

Непрерывное ТРИЗ-образование

Первичные инструменты ТРИЗ. Справочник практика

Автор: М. Орлов
Серия: Библиотека создания инноваций
ISBN: 978-5-91359-081-7
Формат: 70 x 100/16

Справочник является составной частью учебно-методического комплекса, состоящего из трех учебных курсов и применяется при освоении материала поддерживающих их учебников:

1. «Азбука ТРИЗ. Основы изобретательного мышления»
2. «Нетрудная ТРИЗ»
3. «Основы классической ТРИЗ»

Справочник предназначен для применения при решении сложных «неразрешимых» задач с острыми противоречиями в любой прикладной области.

Книга содержит первичные таблицы, каталоги и иллюстрирующие примеры для решения самых распространенных – стандартных и радикальных – противоречий и поддерживает основные учебники Модерн ТРИЗ Академии. Поскольку справочные материалы вынесены в это отдельное небольшое издание, они не дублируются в учебниках.

Изданный отдельной компактной книгой, справочник эффективен сначала для обучения и тренинга вместе с учебниками, а затем, при профессиональной работе – без обращения к учебникам, то есть с работой только с этой небольшой книгой, что, несомненно, очень удобно для пользования.

Материал справочника стандартизован и полностью согласован с софтвером Модерн ТРИЗ Академии под брендом EASyTRIZ™ (Нетрудная ТРИЗ) в трех вариантах для поддержки всех трех уровней обучения, сертификации и последующей практической работы.

Азбука ТРИЗ. Основы изобретательного мышления

Автор: Орлов М. А.
Серия: Библиотека создания инноваций
ISBN: 978-5-91359-080-0
Формат: 70x100/16

Она предназначена для школьников и учителей, студентов и всех начинающих для правильного – с первых шагов! – тренинга изобретательности, для воспитания любознательности и творческой устремленности, для увлечения фантазией и мечтой, которые могут стать реальностью.

Эта книга поддерживается специальной справочной книгой автора «Первичные инструменты ТРИЗ. Справочник практика», а также программными средствами – софтвером EASyTRIZ™.

Данная книга является составной частью учебно-методического комплекса, состоящего из трех учебных курсов:

1 «Азбука ТРИЗ. Основы изобретательного мышления»
2 «Нетрудная ТРИЗ»
3 «Основы классической ТРИЗ

Учебник эффективен как для самостоятельного освоения основ ТРИЗ, так и для обучения и сертификации на уровень Apprentice in MTRIZ Science and Arts (МТРИЗУченик) и не требует дополнительной подготовки, поэтому доступен широким массам читателей.

Приложение H

ANSI Escape Sequences - VT100 / VT52 (ENG)

This document describes how to control a VT100 terminal.

ANSI escape sequence is a sequence of ASCII characters, the first two of which are the ASCII "Escape" character 27 (1Bh) and the left-bracket character "[" (5Bh). The character or characters following the escape and left-bracket characters specify an alphanumeric code that controls a keyboard or display function.

Esc**[20h**	Set new line mode
Esc**[?1h**	Set cursor key to application
Esc**[?3h**	Set number of columns to 132
Esc**[?4h**	Set smooth scrolling
Esc**[?5h**	Set reverse video on screen
Esc**[?6h**	Set origin to relative
Esc**[?7h**	Set auto-wrap mode
Esc**[?8h**	Set auto-repeat mode
Esc**[?9h**	Set interlacing mode
Esc**[20l**	Set line feed mode
Esc**[?1l**	Set cursor key to cursor
Esc**[?2l**	Set VT52 (versus ANSI)
Esc**[?3l**	Set number of columns to 80
Esc**[?4l**	Set jump scrolling
Esc**[?5l**	Set normal video on screen
Esc**[?6l**	Set origin to absolute
Esc**[?7l**	Reset auto-wrap mode
Esc**[?8l**	Reset auto-repeat mode
Esc**[?9l**	Reset interlacing mode

Esc=	Set alternate keypad mode
Esc>	Set numeric keypad mode
Esc**(A**	Set United Kingdom G0 character set
Esc**)A**	Set United Kingdom G1 character set
Esc**(B**	Set United States G0 character set
Esc**)B**	Set United States G1 character set
Esc**(0**	Set G0 special chars. & line set
Esc**)0**	Set G1 special chars. & line set
Esc**(1**	Set G0 alternate character ROM
Esc**)1**	Set G1 alternate character ROM
Esc**(2**	Set G0 alt char ROM and spec. graphics
Esc**)2**	Set G1 alt char ROM and spec. graphics
Esc**N**	Set single shift 2
Esc**O**	Set single shift 3
Esc**[m**	Turn off character attributes
Esc**[0m**	Turn off character attributes
Esc**[1m**	Turn bold mode on
Esc**[2m**	Turn low intensity mode on
Esc**[4m**	Turn underline mode on
Esc**[5m**	Turn blinking mode on
Esc**[7m**	Turn reverse video on
Esc**[8m**	Turn invisible text mode on
Esc[Line;Line**r**	Set top and bottom lines of a window
Esc[Value**A**	Move cursor up n lines
Esc[Value**B**	Move cursor down n lines
Esc[Value**C**	Move cursor right n lines
Esc[Value**D**	Move cursor left n lines
Esc**[H**	Move cursor to upper left corner
Esc**[;H**	Move cursor to upper left corner
Esc[Line;Column**H**	Move cursor to screen location v,h
Esc**[f**	Move cursor to upper left corner
Esc**[;f**	Move cursor to upper left corner
Esc[Line;Column**f**	Move cursor to screen location v,h
Esc**D**	Move/scroll window up one line
Esc**M**	Move/scroll window down one line
Esc**E**	Move to next line
Esc**7**	Save cursor position and attributes
Esc**8**	Restore cursor position and attributes
Esc**H**	Set a tab at the current column
Esc**[g**	Clear a tab at the current column
Esc**[0g**	Clear a tab at the current column

Esc**[3g**	Clear all tabs
Esc**#3**	Double-height letters, top half
Esc**#4**	Double-height letters, bottom half
Esc**#5**	Single width, single height letters
Esc**#6**	Double width, single height letters
Esc**[K**	Clear line from cursor right
Esc**[0K**	Clear line from cursor right
Esc**[1K**	Clear line from cursor left
Esc**[2K**	Clear entire line
Esc**[J**	Clear screen from cursor down
Esc**[0J**	Clear screen from cursor down
Esc**[1J**	Clear screen from cursor up
Esc**[2J**	Clear entire screen
Esc**5n**	Device status report
Esc**0n**	Response: terminal is OK
Esc**3n**	Response: terminal is not OK
Esc**6n**	Get cursor position
EscLine;Column**R**	Response: cursor is at v,h
Esc**[c**	Identify what terminal type
Esc**[0c**	Identify what terminal type (another)
Esc**[?1;**Value**0c**	Response: terminal type code n
Esc**c**	Reset terminal to initial state
Esc**#8**	Screen alignment display
Esc**[2;1y**	Confidence power up test
Esc**[2;2y**	Confidence loopback test
Esc**[2;9y**	Repeat power up test
Esc**[2;10y**	Repeat loopback test
Esc**[0q**	Turn off all four leds
Esc**[1q**	Turn on LED #1
Esc**[2q**	Turn on LED #2
Esc**[3q**	Turn on LED #3
Esc**[4q**	Turn on LED #4

Codes for use in VT52 compatibility mode

Esc<	Enter/exit ANSI mode (VT52)
Esc=	Enter alternate keypad mode
Esc>	Exit alternate keypad mode
Esc**F**	Use special graphics character set
Esc**G**	Use normal US/UK character set
Esc**A**	Move cursor up one line

Esc**B**	Move cursor down one line
Esc**C**	Move cursor right one char
Esc**D**	Move cursor left one char
Esc**H**	Move cursor to upper left corner
Esc**I**	Generate a reverse line-feed
Esc**K**	Erase to end of current line
Esc**J**	Erase to end of screen
Esc**Z**	Identify what the terminal is
Esc/**Z**	Correct response to ident

VT100 Special Key Codes

These are sent from the terminal back to the computer when the particular key is pressed. Note that the numeric keypad keys send different codes in numeric mode than in alternate mode. See escape codes above to change keypad mode.

Function Keys:

Esc**OP** PF1
Esc**OQ** PF2
Esc**OR** PF3
Esc**OS** PF4

Arrow Keys:

	Reset	Set
up	Esc**A**	Esc**OA**
down	Esc**B**	Esc**OB**
right	Esc**C**	Esc**OC**
left	Esc**D**	Esc**OD**

Numeric Keypad Keys:

Esc**Op**	0
Esc**Oq**	1
Esc**Or**	2
Esc**Os**	3
Esc**Ot**	4
Esc**Ou**	5
Esc**Ov**	6
Esc**Ow**	7
Esc**Ox**	8
Esc**Oy**	9

Esc**Om**	-(minus)
Esc**Ol**	,(comma)
Esc**On**	.(period)

Printing:

Esc**[i**	Print Screen
Esc**[1i**	Print Line
Esc**[4i**	Stop Print Log
Esc**[5i**	Start Print Log

Предметный указатель

$ASCII, 88
$BIT, 119, 343
$BITLOGIC, 120
$CASE, 110
$CHAR, 91
$DATA, 74
$ECODE
    Переменная, 375
$ESTACK
    Переменная, 379
$ETRAP
    Обработка ошибок, 371
    Переменная, 371
$EXTRACT, 93
$FIND, 98
$FNUMBER, 104
$GET, 76
$HOROLOG, 478
$INCREMENT, 340
    Атомарность, 341
    Журналирование, 342
$IO, 49
$JOB
    Переменная, 323
$JUSTIFY, 102
$LENGTH, 96
$LISLENGTH, 118
$LIST, 113
$LISTBUILD, 113
$LISTDATA, 118
$LISTFIND, 118
$LISTFROMSTRING, 118
$LISTGET, 118
$LISTNEXT, 118
$LISTSAME, 118
$LISTTOSTRING, 118
$LISTVALID, 118
$NAME, 83
$NEXT, 79
$ORDER, 77
$PIECE, 94
$PRINCIPAL, 49
$QLENGTH, 85
$QSUBSCRIPT, 86
$QUERY, 80
$RANDOM, 108
$REVERSE, 98
$SELECT, 109
$STACK
    Функция, 111
    Переменная, 379
$TEST, 172
$TEXT, 106
$TLEVEL, 333
$TRANSLATE, 99
$TRESTART, 333
$VIEW
    Функция, 109
$X, 484
$Y, 484
$ZEOF, 49
$ZLASCII, 90
$ZLCHAR, 92
$ZQASCII, 90
$ZQCHAR, 92
$ZTIMESTAMP, 482

$ZTIMEZONE, 482
$ZTRAP
  Обработка ошибок, 358
  Переменная, 358
$ZWASCII, 90
$ZWCHAR, 92
Блок строк, 129
Блокировка, 327
  В транзакции, 336
Число, 67
Функция, 72
  $ASCII, 88
  $BIT, 119, 343
  $BITLOGIC, 120
  $CASE, 110
  $CHAR, 91
  $DATA, 74
  $EXTRACT, 93
  $FIND, 98
  $FNUMBER, 104
  $GET, 76
  $INCREMENT, 340
  $JUSTIFY, 102
  $LENGTH, 96
  $LISLENGTH, 118
  $LIST, 113
  $LISTBUILD, 113
  $LISTDATA, 118
  $LISTFIND, 118
  $LISTFROMSTRING, 118
  $LISTGET, 118
  $LISTNEXT, 118
  $LISTSAME, 118
  $LISTTOSTRING, 118
  $LISTVALID, 118
  $NAME, 83
  $NEXT, 79
  $ORDER, 77
  $PIECE, 94
  $QLENGTH, 85
  $QSUBSCRIPT, 86
  $QUERY, 80
  $RANDOM, 108
  $REVERSE, 98
  $SELECT, 109
  $STACK, 111, 384
  $TEXT, 106
  $TRANSLATE, 99
  $VIEW, 109
  $ZLASCII, 90
  $ZLCHAR, 92
  $ZQASCII, 90
  $ZQCHAR, 92
  $ZWASCII, 90
  $ZWCHAR, 92
Глобал, 183
  Индексация, 207
  Кеширование, 198
  Кодирование, 187
  Компрессия, 186
  Лексикографическая сортировка, 190
  Маппинг, 224
  Полное имя, 184
Группировка, 210
Индексация, 207, 226
Каноничность, 220
Команда, 13
  Передача управления, 33
  Постусловие, 53
  Присваивание, 18
  Служебная, 50
  Условная, 26
  Ввод-вывод, 41
  BREAK, 396
  CLOSE, 45
  DO, 35
  ELSE, 26
  FOR, 28
  GOTO, 39
  HALT, 35
  HANG, 50

IF, 26
JOB, 51
KILL, 23
LOCK, 52
MERGE, 21
NEW, 24
OPEN, 44
PRINT, 48
READ, 46
SET, 18
TCOMMIT, 333
TROLLBACK, 333
TSTART, 333
USE, 44
VIEW, 53
WRITE, 45
XECUTE, 40
ZWRITE, 48
ZZDUMP, 49
Комментарий, 175
Косвенность, 143
аргумента $TEXT, 149
аргумента команды, 151
имени, 146
индексов, 147
метки, 149
шаблона, 153
Лексикографическая сортировка, 190
Метка, 127
Параметры, 131
Неопределенное значение, 135
Оператор, 55
Арифметический, 60
Логический, 61
Строковый, 61
Отладчик
BREAK, 396
Caché, 404
MiniM, 401
Serenji, 406
Переменная, 62
$ECODE, 375
$ESTACK, 379
$ETRAP, 371
$HOROLOG, 478
$IO, 49
$JOB, 323
$PRINCIPAL, 49
$STACK, 379
$TEST, 172
$TLEVEL, 333
$TRESTART, 333
$X, 484
$Y, 484
$ZEOF, 49
$ZTIMESTAMP, 482
$ZTIMEZONE, 482
$ZTRAP, 358
Глобальная, 64
Индексация, 207
Локальная, 63
Неопределенная, 135
Пользовательская, 66
Системная, 64
Структурирование, 200
Структурная, 64
Naked indicator, 163
Переносимость, 505
Препроцессор, 471
Директива, 472
Рутина, 125
Блок, 129
Метка, 127
Вызов, 128
Строка, 67
Шаблон, 139
Альтернатива, 142
Число повторов, 140
Код символа, 140
Транзакция, 333
$TLEVEL, 333

TCOMMIT, 333
TROLLBACK, 333
TSTART, 333

B-Tree, 183
BREAK
Команда, 396

CLOSE, 45

DO, 35

ELSE, 26

FOR, 28

GOTO, 39

HALT, 35
HANG, 50

IF, 26

JOB
Команда, 51
Параллельность, 323

KILL, 23

LOCK, 327
Дедлок, 345
Инкрементная, 329
Команда, 52
Правила, 328
Таймаут, 331
В транзакции, 336
LRU, 199

MDC, 177
Требования переносимости, 506
MERGE, 21
MUMPS, 177

Naked indicator, 163
NEW, 24

OPEN, 44

PRINT, 48

READ, 46

SET, 18
Subscript, 62

TCOMMIT, 333
TROLLBACK, 333
TSTART, 333

USE, 44

VIEW
Команда, 53
VLDB, 337

WRITE, 45

XECUTE, 40

ZWRITE, 48
ZZDUMP, 49

Издательство СОЛОН-ПРЕСС представляет

Непрерывное ТРИЗ-образование

Нетрудная ТРИЗ. Универсальный практический курс

Автор: Орлов М. А.
Серия: Библиотека создания инноваций
ISBN: 978-5-91359-089-3
Формат: 70x100/16

Данная книга является составной частью учебно-методического комплекса, базирующегося на трех учебниках, справочнике, софтвере и соответствующих дистанционных курсах автора:

1. «Азбука ТРИЗ. Основы изобретательного мышления»
2. «Первичные инструменты ТРИЗ. Справочник практика»
3. «Нетрудная ТРИЗ. Универсальный практический курс» – данная книга
4. «Основы классической ТРИЗ»

Книга, как и весь учебный комплекс под брендом «EASyTRIZ», базируется на новой технологии, разработанной автором и не имеющей аналога в мире. Эта технология прошла многолетнюю отработку во многих странах с участием тысяч обучаемых. Учебник снабжен многочисленными примерами, задачами и ответами – всего более 250, что гарантирует полноту пояснения методических рекомендаций для практического применения.

Книга рекомендуется специалистам, не знакомым и знакомым с ТРИЗ, а также студентам по любым специальностям – инженерным, педагогическим, дизайнерским, менеджерским, маркетинговым, практически без ограничений.

Основы классической ТРИЗ. Практическое руководство для изобретательного мышления. Изд. 4., дополненное

Автор: Михаил Орлов
Серия: Библиотека создания инноваций
ISBN: 5-98003-191-X
Формат: 70x100/16

Русское издание уникальной книги для менеджеров, бизнесменов, аналитиков, инженеров, экономистов, преподавателей, психологов, студентов и школьников. Рожденная в России Теория Решения Изобретательских Задач (ТРИЗ) сегодня быстро распространяется в мире. Все ведущие компании применяют ТРИЗ, например, Mitsubishi, Samsung, Hewlett Packard, General Electric, Siemens. Книга профессора Михаила Орлова, впервые выпущенная известным издательством Springer Verlag на немецком (Берлин, Германия, 2002) и английском (Нью-Йорк, США, 2003) языках, представляет основные принципы и модели ТРИЗ. Автор книги обладает не только отечественным опытом, но и более чем 10-летним опытом ТРИЗ-консалтинга и чтения лекций во многих странах.

Книга может быть рекомендована для самостоятельного развития систематического, направленного, изобретательного мышления как инженерами, так и инновативными менеджерами, бизнесменами, аналитиками, экономистами, психологами, на применение преподавателями высших и средних школ в любых читаемых дисциплинах. Книга вполне доступна и, несомненно, полезна студентам и старшим школьникам.

Литература

[1] Гессе С., Кирстен В. Введение в язык программирования М. – СПб: СП. АРМ, 1996 – 280с.

[2] Кирстен В. От ANS MUMPS к ISO M. – СПб: СП. АРМ, 1995 – 277с.

[3] Кирстен В., Иринger М., Кюн М., Рериг Б. Постреляционная СУБД Caché 5. Объектно-ориентированная разработка приложений. – М: Бином пресс, 2005 – 416с.

[4] Lewkowicz, J.M. (1989) The Complete MUMPS. Prentice Hall.

[5] Cache Online Documentation
http://docs.intersystems.com/cache20121/csp/docbook/
DocBook.UI.HomePageZen.cls

[6] GT.M Programmer's Guide.
http://tinco.pair.com/bhaskar/gtm/doc/books/pg/
UNIX_manual/index.html

[7] MiniM Online Documentation
http://www.minimdb.com/doc.html

Евгений Анатольевич Каратаев

MUMPS СУБД

Ответственный за выпуск
В. Митин

Обложка
СОЛОН-ПРЕСС

ООО «СОЛОН-ПРЕСС»
123001, г. Москва, а/я 82
Телефоны:
(495) 617-39-64, (495) 617-39-65
E-mail: avtor@solon-press.ru,
www.solon-press.ru

По вопросам приобретения обращаться:
ООО «ПЛАНЕТА АЛЬЯНС»
Тел: (499) 782-38-89, www.alians-kniga.ru

По вопросам подписки на журнал «Ремонт & Сервис» обращаться:
ООО «СОЛОН-ПРЕСС»
тел.: (495) 617-39-64
www.remserv.ru

ООО «СОЛОН-ПРЕСС»
115142, г. Москва, Кавказский бульвар, д. 50
Формат 60×88/16. Объем 34,375 п. л. Тираж 50 экз.

Заказ №