Common Lisp. Введение

Кальянов Д.В.
Kalyanov.Dmitry@gmail.com

10 апреля 2009 года

Содержание

Одна цитата

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

— Eric Raymond, "How to Become a Hacker"

Парадигмы Лиспа

Common Lisp — мультипарадигменный язык, поддерживает:

- Функциональное программирование
- Императивное программирование
- Структурное программирование
- Объектно-ориентированное программирование
- Обобщенное программирование

Содержание

Выражения

S-Expressions:

- Атом
  - 123456789, 1/3, 0.132
  - A−SYMBOL, +
  - "A␣string"

  Два специальных значения:
  - T
  - NIL
- Список $(E_0\ E_1\ E_2\ \ldots\ E_n)$
  - Применение функции. (+ 1 2 3), (**sin** pi).
  - Блокирование вычисления. (quote (1 2 3)) = '(1 2 3)
  - Функция. (function +) = #'+
  - Макрос
  - Специальный оператор

Префиксная запись

- Инфиксная запись: $x + y$
- Префиксная запись: $sin(x)$
- S-expression: (**sin** x)

Специальные операторы

- Условный оператор
  $$(\text{if a b c}) = \begin{cases} b, & \text{если } a \\ c, & \text{иначе} \end{cases}$$
- Оператор присваивания
  (**setf** var new−value)
- Последовательное вычисление
  (progn E_1 E_2 ... E_n)

Содержание

Символы

Символ — это объект, идентифицирующий что-либо. Имена переменных, функций, типов, классов являются символами.

- (symbol−name symbol) — имя символа
- (**intern** name) — возвращает символ с указанным именем
- (**gensym**) — создает уникальный символ

Числа

- Целые числа неограниченного размера
  119911638487169062970727 21
- Рациональные числа неограниченной точности
  211308/359465
- Вещественные числа с плавающей запятой
  3.141592653589793
- Комплексные числа
  (**sqrt** -1) = #C(0.0 1.0)

Списки и массивы

- Список — простейшая структура. Имеет голову и хвост.
  (**first** '(1 2 3 4)) = 1
  (**rest** '(1 2 3 4)) = (2 3 4)
- Массив — набор элементов, индексированный одним или несколькими числами
  (**aref** #(1 2 3) 0) = 1

Структуры

- (**defstruct** point x y z)
- (make−point :x 1 :y 2 :z 3)
- (point−x (make−point :x 1 :y 2 :z 3)) = 1
- (**setf** (point−x p) 1)
- (slot−value p 'x)
- (**setf** (slot−value p 'x) 1)

Функции

Функции — объекты, которые могут быть «применены» к другим объектам и предоставить некоторый результат этого применения в виде одного или нескольких значений.

- #'+, #'**sin**, #'**first**
- (symbol−function '+)
- (**truncate** 10 3) => 3, 1
- **funcall** применяет функцию к аргументам:
  (**funcall** #'+ 1 2 3) => 6
- **apply** применяет функцию к списку аргументов:
  (**apply** #'+ '(1 2 3)) => 6

Содержание

Определение пользовательских функций

(**defun** name (args) body)

- name — имя функции
- args — список аргументов функции
- body — тело функции — последовательность выражений
- результат функции — результат последнего выражения

lambda-list

λ-list — список аргументов функции
a b c &optional d (e 0.0) &key f (g 1) &**rest** args

- a, b, c — обязательные параметры
- d, e — необязательные параметры. e по умолчанию имеет значение 0.0, d — NIL
- f, g — именованные параметры. g по умолчанию имеет значение 1, f — NIL
- args — остаток списка аргументов

Вызов функции:
(f 1 2 3 T 13 :f 10 1 3 4 'A)

lambda

λ, или безымянная функция — функция, которая не имеет имени.
(lambda (args) body) — функция, которая возвращает результат применения последовательности выражений body к аргументам args.

- args — λ-list
- body — последовательность выражений

(lambda (x y) (+ x y 10))
(**funcall** (lambda (x y) (+ x y 10)) 10 15) => 35

Содержание

places

Обобщенная ссылка — некое указание на «место», из которого можно читать и писать.
(**setf** x new−value) присваивает значение new-value по ссылке x
(**setf** (point−x p) 1)
Примеры стандартных ссылок:

- (**nth** n **list**) — *n*-й элемент списка
- (**aref** array index) — значение в массиве
- (symbol−name symbol) — имя символа

Примеры возможных ссылок:

- (point−x p) — x-координата точки
- (edge−weight vertex−1 vertex−2) — вес ребра между двумя вершинами графа

Передача управления

- Возврат из функции: (return−from name x) осуществляет возврат значения x из функции name
- GOTO:

```
(tagbody
   label-1  expr-1
   label-2  (go  label-1))
```

- Система обработки событий и сигналов
(if (**not** (data−good)) (**error** "Error␣in␣data"))!

синтаксис цикла

(**loop** clauses)

- for X from A [to B|below B]
- for X in LIST
- for X initially Y then Z
- for X = EXPR
- while CONDITION
- **unless** CONDITION
- **do** EXPR
- **collect** EXPR
- sum EXPR

(**loop** for x from 0 to 10 for y = (**expt** x 2) **collect** y)
=> (0 1 4 9 16 25 36 49 64 81 100)

Содержание

Окружение

- Окружение (environment) — отображение из имени переменной в ее значение.
- Каждое выражение вычисляется в каком-то окружении, и значения переменных берутся из этого окружения.
- `(defun foo (x y) (+ x y))`

  В выражении (+ x y) значения x и y берутся из окружения, созданного вызовом функции foo.

Лексическое окружение

- Лексическое окружение — окружение, действие которого распространяется на выражения, расположенных в определенной части программы, и ни для каких других.
- (**defun** foo (x y) (+ x y)) — переменные x и y находятся в лексическом окружении
- let, let* создают лексическое окружение:

```
(let ((x 1) (y 2))
  (+ x y))
```

Динамическое окружение

- Динамическое окружение — окружение, действие которого распространяется на выражения, выполняющиеся во время действия этого выражения.
- Глобальные переменные находятся в динамическом окружении.

Замыкание

- Замыкание — функция, тело которой вычисляется при действии внешнего лексического окружения.
- ```
  (let ((x 10))
    (lambda (y) (+ x y)))
  ```
- Переменная x отсутствует в лексическом окружении, создаваемом при входе в λ. Ее значение берется из лексического окружения, создаваемого let, которое действовало в момент создания λ.
  ```

Функции высокого порядка

ФВП — функции, возвращающие или принимающие в качестве аргументов другие функции

- (**defun** make−adder (x) (lambda (y) (+ x y)))
- (make−adder x) возвращает функцию от одного аргумента, которая увеличивает свой аргумент на X и возвращает увеличенное значение.
- (**funcall** (make−adder 10) 20) => 30

eval

- (**eval** expr)
- Функция eval принимает выражение expr (заданное в виде списка) и возвращает его значение. Это — наиболее мощный примитив вычисления выражений.
- (**eval** '(+ 10 20)) => 30

compile

- (**compile** name &optional definition)
- Функция compile компилирует функцию с именем name (или безымянную функию, если name = NIL) с лямбдой definition и возвращает скомпилированную функцию.
- (**compile** nil '(lambda (x y) (+ x y 10))) возвращает функцию от двух аргументов, которая возвращает сумму этих аргументов и числа 10.
- Функция compile позволяет «на лету» создавать новые функции произвольного вида. Например, в математическом пакете для формул, введенных пользователем.

defmacro

- Макрос — какое-либо преобразование исходного кода программы.
- Макрос задается в виде функции, принимающей на вход выражение, и возвращающий новое выражение.
- При чтении компиляции исходного текста программы компилятор рекурсивно применяет макросы к коду программы, до тех пор, пока это возможно.
- Макрос может совершать произвольные действия, в том числе читать и писать файлы.

Пример макроса

Пример:

```
(defmacro assert (expr)
  `(unless ,expr
     (error "Assertion ~A failed" ',expr)))
```

assert не может быть функцией, т.к. должен иметь доступ к исходному тексту выражения expr.
` и , — синтаксис для управления блокированием вычисления.

Более интересный пример: CL-YACC

```
(define-parser *expression-parser*
  (:start-symbol expression)
  (:terminals (int id + - * / |(| |)|))
  (:precedence ((:left * /) (:left + -)))

  (expression
   (expression + expression #'i2p)
   (expression - expression #'i2p)
   (expression * expression #'i2p)
   (expression / expression #'i2p)
   term)
  (term id int (- term) (|(| expression |)| #'k-2-3)))
```