Tutorial de Programação Lisp

Links Relacionados: Linguagens de Programação, Paradigma de Programação Funcional.

Índice

Referências

Introdução

O Lisp, inventado por J. McCarthy em 1959, é uma linguagem de programação funcional com usos convencionais e uma linguagem para Inteligência Artificial interativo. Programas em Lisp são altamente portáveis entre máquinas e sistemas operacionais. A linguagem se encontra hoje padronizada e possui o nome de Common Lisp.

Essa linguagem tem sido bem aceita nos domínios do processamento simbólico e de conhecimento da Inteligência Artificial, no processamento numérico (MACLISP) e em programas como editores (EMACS) e CAD (AUTOCAD).

O ambiente do LISP é executado de forma interativa. Neste ambiente o usuário entra com um conjunto de expressões, chamadas de ***forms*** e elas são avaliadas. Também é possível inspecionar variáveis, chamar funções com argumentos e definir suas próprias funções.

Características do Common Lisp

- Sintaxe clara;
- **Muitos tipos de dados**: numbers, strings, arrays, lists, characters, symbols, structures, streams etc.;
- **Tipagem em tempo de execução**: o programador geralmente não precisa se preocupar com declarações de tipo, mas ele recebe mensagens de erro caso haja violações de tipo (operações ilegais);
- Funções genéricas;
- Gerenciamento de memória automático (*garbage collection*);
- Empacoteamento (*packaging*) de programas em módulos;
- Um sistema de objetos, funções genéricas com a possibilidade de combinação de métodos;
- **Macros**: todo programador pode realizar suas próprias extensões da linguagem;

A implementação de Common Lisp (CLISP) provê

- Um interpretador;
- Um compilador para executáveis até 5 vezes mais rápidos;
- Todos os tipos de dados com tamanho ilimitado (a precisão e o tamanho de uma variável não necessita de ser declarado, o tamanho de listas e arrays altera-se dinamicamente);
- Inteiros de precisão arbitrária, precisão de ponto flutuante ilimitada.

Tutorial da Linguagem LISP

Neste tutoria estamos utilizando o ambiente CLISP. Este inclui também um dialeto de LISP orientado a objetos, chamado de CLOS.

Símbolos

Um símbolo é somente um string de caracteres. Por exemplo:

```
1 a
2 b
3 c1
4 foo
```

```
5 bar
6 baaz-quux-garply
```

O **prompt** do interpretador do lisp pode ser identificado pelo caractere ">". Ou seja, é a área em que o usuário pode digitar seus comandos. Comentários podem ser adicionados com ";". Tudo o que vier após isso é ignorado pelo interpretador.

```
1 > (setq a 5) ; armazena um numero como valor de um simbolo
2 5
3 > a ; retorna o valor do simbolo a
4 5
5 > (let ((a 6)) a) ; altera o valor de a temporariamente para 6
6 6
7 > a ; sai do let e o valor de a retorna para 5
8 5
9 > (+ a 6) ; utiliza o valor do simbolo a como argumento para uma funcao
10 11
11 > b ; tenta retornar o valor de um simbolo nao definido
12
13 *** - SYSTEM::READ-EVAL-PRINT: variable B has no value
```

Há dois símbolos especiais, t e nil. O valor de t é definido sempre como sendo t. nil é definido como sendo sempre nil.

LISP utiliza t e nil para representar verdadeiro e falso. Por exemplo:

```
1 > (if t 5 6)
2 5
3 > (if nil 5 6)
4 6
5 > (if 4 5 6)
6 5
```

O último exemplo é estranho, mas está correto. nil significa falso e qualquer outra coisa verdadeiro. Usamos t somente para clareza.

Símbolos como nil e t são chamados símbolos auto-avaliantes, porque avaliam para si mesmos.

Há toda uma classe de símbolos auto-avaliantes chamados palavras-chave. Qualquer símbolo cujo nome inicia com dois pontos é uma palavra-chave. Exemplos:

```
1 > :this-is-a-keyword
2 :THIS-IS-A-KEYWORD
3 > :so-is-this
4 :SO-IS-THIS
5 > :me-too
6 :ME-TOO
```

Números

Um **inteiro** é um string de dígitos opcionalmente precedido de um + ou -.

Um **real** parece com um inteiro, só que possui um ponto decimal e pode opcionalmente ser escrito em notação científica.

Um **racional** se parece com dois inteiros com um / entre eles.

LISP suporta números complexos que são escritos #c(r i)

Exemplos:

```
1 17
  -34
2 3 +6
4 3.1415
5 1.722e-15
6 #c(1.722e-15 0.75)
```

As funções aritméticas padrão são todas avaliáveis: *+, -, *, /, floor, ceiling, mod, sin, cos, tan, sqrt, exp, expt*, etc.

Todas elas aceitam qualquer número como argumento:

```
1 > (+ 3 3/4)
2 15/4
3 > (exp 1) ; e
4 2.7182817
5 > (exp 3) ;e*e*e
6 20.085537
7 > (expt 3 4.2) ; expoente com outra base
8 100.90418
9 > (+ 5 6 7 (* 8 9 10))
10 738
```

Não existe limite para o valor absoluto de um inteiro exceto a memória do computador. Evidentemente cálculos com inteiros ou racionais imensos podem ser muito lentos.

Conses - Associações

Um **cons** é somente um registro de dois campos. Os campos são chamados de **car** e **cdr** por razões históricas: na primeira máquina onde LISP foi implementado havia duas instruções assembler CAR (*Contents of Address Register*) e CDR (*Contents of Decrement Register*).

Conses foram implementados utilizando-se esses dois registradores:

```
1 > (cons 4 5) ; Aloca um cons. Seta car para 4 e cdr para 5.
2 (4 . 5)
3 > (cons (cons 4 5) 6)
4 ((4 . 5) . 6)
5 > (car (cons 4 5))
6 4
7 > (cdr (cons 4 5))
8 5
```

Listas

Você pode construir muitas estruturas de dados a partir de conses. A mais simples com certeza é a lista encadeada:

- o car de cada cons aponta para um dos elementos da lista;
- o cdr aponta ou para outro cons ou para nil.

Uma lista assim pode ser criada com a função de lista:

```
1 > (list 4 5 6)
2 (4 5 6)
```

Observe que Lisp imprime listas de uma forma especial: ele omite alguns dos pontos e parênteses.

A regra é: se o cdr de um cons é nil, Lisp não se preocupa em imprimir o ponto ou o nil. Se o cdr de cons A é cons B, então Lisp não se preocupa em imprimir o ponto para A nem o parênteses para B:

```
1 > (cons 4 nil)
  (4)
2 3 > (cons 4 (cons 5 6))
4 (4 5 . 6)
5 > (cons 4 (cons 5 (cons 6 nil)))
6 (4 5 6)
```

O último exemplo corresponde ao (list 4 5 6) anterior. Note que nil corresponde à lista vazia.

- O car e cdr de nil são definidos como nil.
- O car de um átomo é o próprio átomo.
- O cdr de um átomo é nil.

Se você armazena uma lista em uma variável, pode fazê-la funcionar como uma pilha:

```
 1 > (setq a nil)
 2 NIL
 3 > (push 4 a)
 4 (4)
 5 > (push 5 a)
 6 (5 4)
 7 > (pop a)
 8 5
 9 > a
10 (4)
11 > (pop a)
12 4
13 > (pop a)
14 NIL
15 > a
16 NIL
```

Funções

Vimos exemplos de funções acima. Aqui mais alguns:

```
1 > (+ 3 4 5 6) ; Essa funcao pode ter varios argumentos
2 18
3 > (+ (+ 3 4) (+ (+ 4 5) 6)) ; A saida de uma funcao ja pode servir para entrada em outra
4 22
```

Definindo uma função

```
1 > (defun foo (x y) (+ x y 5))
2 FOO
3 > (foo 5 0) ; chamando a função
4 10
```

Função Recursiva

```
1 > (defun fatorial (x)
2     (if (> x 0)
3        (* x (fatorial (- x 1)))
4         1
5    ))
6 FATORIAL
7 > (fatorial 5)
8 120
```

Funções Mutuamente Recursivas

```
1 > (defun a (x) (if (= x 0) t (b (- x))))
2 A
3 > (defun b (x) (if (> x 0) (a (- x 1)) (a (+ x 1))))
4 B
5 > (a 5)
6 T
```

Função com múltiplos comandos em seu corpo

```
1 > (defun bar (x)
2     (setq x (* x 3))
3     (setq x (/ x 2))
4     (+ x 4)
5   )
6 BAR
7 > (bar 6)
8 13
```

O valor retornado, como em um método em *Smalltalk*, é sempre o valor da última expressão executada.

Escopo de variáveis

Quando nós definimos *foo*, nós lhe atribuímos dois argumentos, x e y. Quando chamamos *foo*, necessitamos prover valores para esses dois argumentos. O primeiro será o valor de x durante a duração da chamada a *foo*, o segundo o valor de y durante a duração da chamada a *foo*. Em LISP, a maioria das variáveis são colocadas no escopo de forma léxica. Isto significa que se *foo* chama *bar* e *bar* tenta referenciar x, *bar* não obterá os valor de x de *foo*.

```
1 > (defun foo (x y) (+ x y 5))
2 FOO
```

O processo de se associar um valor a um símbolo durante um certo escopo léxico é chamado em LISP de ateamento. x estava atado ao escopo de *foo*.

Número Variável de Argumentos para Funções

Você pode especificar também argumentos opcionais para funções. Qualquer argumento após o símbolo &optional é opcional:

```
1 > (defun bar (x &optional y) (if y x 0))
2 BAR
```

É perfeitamente legal chamar a função BAR com um ou dois argumentos. Se for chamada com um argumento, x será atado ao valor deste argumento e y será atado a NIL.

```
1 > (bar 5)
2 0
3 > (bar 5 t)
4 5
```

Se for chamada com dois argumentos, x e y serão atados aos valores do primeiro e segundo argumento respectivamente.

A função BAAZ possui dois argumentos opcionais, porém especifica um valor default para cada um deles.

```
1 > (defun baaz (&optional (x 3) (z 10)) (+ x z))
2 BAAZ
```

```
3 > (baaz 5)
4 15
5 > (baaz 5 6)
6 11
7 > (baaz)
8 13
```

Se quem a chama especificar somente um argumento, z será atado a 10 ao invés de NIL. Se nenhum argumento for especificado, x será atado a 3 e z a 10.

Número Indefinido de Parâmetros

Você pode fazer a sua função aceitar um número indefinido de parâmetros terminando a sua lista de parâmetros com o parâmetro &rest. LISP vai coletar todos os argumentos que não sejam contabilizados para algum argumento formal em uma lista a atá-la ao parâmetro &rest :

```
1 > (defun foo (x &rest y) y)
2 FOO
3 > (foo 3)
4 NIL
5 > (foo 4 5 6)
6 (5 6)
```

Passagem de Parâmetros por Nome

Existe ainda um tipo de parâmetro opcional chamado de parâmetro de palavra-chave. São parâmetros que quem chama pode passar em qualquer ordem, pois os valores são passados precedidos pelo nome do parâmetro formal a que se referem:

```
 1 > (defun foo (&key x y) (cons x y))
 2 FOO
 3 > (foo :x 5 :y 3)
 4 (5 . 3)
 5 > (foo :y 3 :x 5)
 6 (5 . 3)
 7 > (foo :y 3)
 8 (NIL . 3)
 9 > (foo)
10 (NIL)
```

Um parâmetro **&key** pode ter um valor default também:

```
1 > (defun foo (&key (x 5)) x)
2 FOO
3 > (foo :x 7)
4 7
5 > (foo)
6 5
```

Impressão

Algumas funções podem provocar uma saída. A mais simples é print, que imprime o seu argumento e então o retorna.

```
1 > (print 3)
2 3
3 3
```

O primeiro 3 foi impresso, o segundo retornado.

Se você deseja uma saída mais complexa, você necessita utilizar *format*:

```
1 > (format t "An atom: ~S~%and a list: ~S~%and an integer:~D~%"
2           nil (list 5) 6)
3 An atom: NIL
4 and a list: (5)
5 and an integer: 6
```

- O primeiro argumento a format é ou *t*, ou *NIL* ou um arquivo;
- *T* especifica que a saída deve ser dirigida para o terminal;
- *NIL* especifica que não deve ser impresso nada, mas que *format* deve retornar um string com o conteúdo ao invés,

uma referência a um arquivo especifica o arquivo para onde a saída vai ser mostrada;

- O segundo argumento é um padrão de formatação, o qual é um string contendo opcionalmente diretivas de formatação, de forma similar à Linguagem *C*: "An atom: ~S~%and a list: ~S~%and an integer:~D~%".

Todos os argumentos restantes devem ser referenciados a partir do string de formatação.

As diretivas de formatação do string serão repostas por LISP por caracteres apropriados com base nos valores dos outros parâmetros a que eles se referem e então imprimir o string resultante.

Format sempre retorna NIL, a não ser que seu primeiro argumento seja NIL, caso em que não imprime nada e retorna o string resultante.

No exemplo acima, há três diretivas de formatação: ~S, ~D e ~%.:

- A primeira, *~S*, aceita qualquer objeto Lisp e é substituída por uma representação passível de ser impressa deste objeto (a mesma produzida por print);
- A segunda, *~D*, só aceita inteiros;
- A terceira, *~%*, não cita nada. Sempre é reposta por uma quebra de linha;
- Outra diretiva útil é ~~, que é substituída por um simples ~.

Forms e o Laço Top-Level

O que você digita no *prompt* do interpretador Lisp são chamadas *forms*. O interpretador repetidamente lê um form, o avalia e mostra o resultado. Este procedimento é chamado *read-eval-print loop*.

Alguns *forms* provocam erros. Dessa forma, o Lisp entr ano ambiente de *debug*.

Os *Debuggers* de interpretadores LISP são muito diferentes entre si. A maioria aceita um *help* ou *:help* para auxiliar no *debug*.

No *debugger* do CLISP você pode sair dando um *Control-Z* (no CMD do Windows) ou *Control-D* (em Unix/Linux).

Em geral, um *form* é ou um átomo (um símbolo, um inteiro ou um string) ou uma lista.

Se o *form* for um átomo, o Lisp o avalia imediatamente. Símbolos avaliam para seu valor, inteiros e strings avaliam para si mesmos.

Se o form for uma lista, o Lisp trata o seu primeiro elemento como o nome da função, avaliando os elementos restantes de forma recursiva. Então chama a função com com os valores dos elementos restantes como argumentos.

Por exemplo, se LISP vê o form:

```
1 > (+ 3 4)
2 7
```

Ele trata + como o nome da função, avaliando 3 para obter 3, avaliando 4 para obter 4 e finamente chamando a função + com 3 e 4 como argumentos, retornando 7 e imprimindo na tela.

O *top-level loop* provê algumas outras conveniências. Uma particularmente interessante é a habilidade de falar a respeito dos resultados de *forms* previamente digitados: o Lisp sempre salva os seus três resultados mais recentes. Ele os armazena sob os símbolos *, ** e ***.

```
1 > 3
2 3
3 > 4
4 4
5 > 5
6 5
7 > ***
8 3
9 > ***
10 4
11 > ***
12 5
13 > **
14 4
15 > *
16 4
```

Forms especiais

Há um número de *forms* especiais que se parecem com chamadas a funções mas não o são. Um form muito útil é o form aspas. As aspes prevem um argumento de ser avaliado.

```
1 > (setq a 3)
2 3
3 > a
4 3
5 > (quote a)
6 A
7 > 'a ;'a eh uma abreviacao para (quote a)
8 A
```

Outro form especial similar é o *form function*.

Function faz com que seu argumento seja interpretado como uma função ao invés de ser avaliado:

```
1 > (setq + 3)
2 3
3 > +
4 3
5 > '+
6 +
7 > (function +)
8 #<Function + @ #x-fbef9de>
9 > #'+                    ; #'+ eh uma abreviacao para (function +)
10 #<Function + @ #x-fbef9de>
```

O *form* especial function é útil quando você deseja passar uma função como parâmetro para outra função. Mais tarde apresentaremos alguns exemplos de funções que aceitam outras funções como parâmetros.

Binding - Atamento/Amarração

Binding é uma atribuição escopada lexicamente. Ela ocorre com as variáveis de uma lista de parâmetros de uma função sempre que a função é chamada: os parâmetros formais são atados aos parâmetros reais pela duração da chamada à função.

Você pode também amarrar variáveis em qualquer parte de um programa com o form especial let:

```
1 (let ((var1 val1)
2       (var2 val2)
```

```
3        ...
4      )
5 body)
```

Let ata var1 a val1, var2 a val2, e assim por diante; então executa os comandos de seu corpo. O corpo de um let segue as mesmas regras de um corpo de função:

```
1 > (let ((a 3)) (+ a 1))
2 4
3 > (let ((a 2)
4         (b 3)
5         (c 0))
6     (setq c (+ a b))
7     c
8   )
9 5
10 > (setq c 4)
11 4
12 > (let ((c 5)) c)
13 5
14 > c
15 4
```

Ao invés de *(let ((a nil) (b nil)) ...)* você pode escrever *(let (a b) ...)*.

Os valores val1, val2, etc. dentro de um *let* não podem referenciar as variáveis *var1*, *var2*, etc. que o *let* está atando:

```
1 > (let ((x 1)
2         (y (+ x 1)))
3     y
4   )
5 Error: Attempt to take the value of the unbound symbol X
```

Se o símbolo x já possui um valor, coisas estranhas podem acontecer:

```
1 > (setq x 7)
2 7
3 > (let ((x 1)
4         (y (+ x 1)))
5     y
6   )
7 8
```

O form especial *let** é semelhante, só que permite que sejam referenciadas variáveis definidas anteriormente:

```
1 > (setq x 7)
2 7
3 > (let* ((x 1)
4          (y (+ x 1)))
5     y
6   )
7 2
```

O form:

```
1 (let* ((x a)
2        (y b))
3         ...
4 )
```

é equivalente a:

```
1 (let ((x a))
2    (let ((y b))
```

```
3    ...
4 ))
```

Dynamic Scoping - Escopo Dinâmico

Os *forms let* e *let** proveem escopo léxico, que é o que você está acostumado quando programa em "C" ou PASCAL.

Escopo dinâmico é o que você tem em BASIC, ou seja, se você atribui um valor a uma variável com escopo dinâmico, toda menção desta variável retorna aquele valor até que você atribua outro valor à mesma variável.

Em LISP variáveis com referenciadas escopo dinâmico são variáveis especiais. Você pode criar uma variável especial através do *form defvar*.

Abaixo seguem alguns exemplos de variáveis com escopo léxico e dinâmico.

Neste exemplo a função *check-regular* referencia uma variável regular (escopo léxico). Como *check-regular* está em um contexto léxico fora do *let* que ata regular, *check-regular* retorna o valor global da variável:

```
1 > (setq regular 5)
2 5
3 > (defun check-regular () regular)
4 CHECK-REGULAR
5 > (check-regular)
6 5
7 > (let ((regular 6)) (check-regular))
8 5
```

Neste exemplo, a função *check-special* referencia uma variável especial (escopo dinâmico). Uma vez que a chamada a *check-special* é temporariamente dentro do *let* que amarra special, *check-special* retorna o valor local da variável:

```
1 > (defvar *special* 5)
2 *SPECIAL*
3 > (defun check-special () *special*)
4 CHECK-SPECIAL
5 > (check-special)
6 5
7 > (let ((*special* 6)) (check-special))
8 6
```

Por uma questão de convenção, o nome de uma variável especial começa e termina com *. Variáveis especiais são principalmente usadas como variáveis globais.

Arrays

A função *make-array* cria um array e para acessar seus elementos, usa-se a função *aref*. Todos os elementos de um array são inicialmente setados para nil:

```
1 > (make-array '(3 3))
2 #2a((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))
3 > (aref * 1 1)
4 NIL
5 > (make-array 4) ; arrays de uma dimensao nao precisam de parametros extras.
6 #(NIL NIL NIL NIL)
```

Índices de um array sempre começam em 0, tal como em C ou Java.

Strings

Um string é uma seqüência de caracteres entre aspas duplas. Lisp representa um string como um array de tamanho variável de caracteres.

Você pode escrever um string que contém a aspa dupla, precedendo a de uma barra invertida \ ("\""). Se na string você quiser representar uma barra invertida, basta utilizar duas barras invertidas juntas \\.

Segue algumas funções para manipulação de strings:

```
1 > (concatenate 'string "abcd" "efg")
2 "abcdefg"
3 > (char "abc" 1) ; caracteres em Lisp são precedidos de #\
4 #\b
5 > (aref "abc" 1) ; em Lisp, uma string e uma lista de caracteres
6 #\b
```

A função *concatenate* pode trabalhar sobre qualquer tipo de seqüência:

(concatenate 'string '(#\a #\b) '(#\c)) "abc" > (concatenate 'list "abc" "de") (#\a #\b #\c #\d #\e) > (concatenate 'vector '#(3 3 3) '#(3 3 3))

1. (3 3 3 3 3 3)

</syntaxhighlight>

Estruturas

Estruturas Lisp são análogas a *structs* em "C" ou *records* em PASCAL:

```
1 > (defstruct foo
2     bar
3     baaz
4     quux
5   )
6 FOO
```

Este exemplo define um tipo de dado chamado *FOO* contendo 3 campos.

Define também 4 funções que operam neste tipo de dado: *make-foo, foo-bar, foo-baaz, and foo-quux*.

A primeira cria um novo objeto do tipo *FOO*.

As outras acessam os campos de um objeto do tipo *FOO*.

```
1 > (make-foo)
2 #s(FOO :BAR NIL :BAAZ NIL :QUUX NIL)
3 > (make-foo :baaz 3)
4 #s(FOO :BAR NIL :BAAZ 3 :QUUX NIL)
5 > (foo-bar *)
6 NIL
7 > (foo-baaz **)
8 3
```

A função *make-foo* pode tomar um argumento para cada um dos campos da estrutura do tipo. As funções de acesso a campo tomam cada uma um argumento.

Setf

Alguns *forms* em Lisp naturalmente definem uma locação na memória.

Por exemplo, se x é uma estrutura do tipo *FOO*, então *(foo-bar x)* define o campo *BAR* do valor de x.

Ou, se o valor de y é um array unidimensional, então *(aref y 2)* define o terceiro elemento de y.

O form especial *setf* usa seu primeiro argumento para definir um lugar na memória, avalia o seu segundo argumento e armazena o valor resultante na locação de memória resultante:

```
1 > (setq a (make-array 3))
2 #(NIL NIL NIL)
3 > (aref a 1)
4 NIL
5 > (setf (aref a 1) 3)
6 3
7 > a
8 #(NIL 3 NIL)
9 > (aref a 1)
10 3
11 > (defstruct foo bar)
12 FOO
13 > (setq a (make-foo))
14 #s(FOO :BAR NIL)
15 > (foo-bar a)
16 NIL
17 > (setf (foo-bar a) 3)
18 3
19 > a
20 #s(FOO :BAR 3)
21 > (foo-bar a)
22 3
```

Setf é a única maneira de se setar os valores de um array ou os campos de uma estrutura.

Alguns exemplos de *setf* e funções relacionadas:

```
1 > (setf a (make-array 1))       ; setf em uma variavel eh equivalente a setq
2 #(NIL)
3 > (push 5 (aref a 1))           ; push pode agir como setf
4 (5)
5 > (pop (aref a 1))              ; pop insere um elemento
6 5
7 > (setf (aref a 1) 5)
8 5
9 > (incf (aref a 1))             ; incf le de um local, incrementa e altera o valor
10 6
11 > (aref a 1)
12 6
```

Booleanos e Condicionais

Lisp usa o símbolo auto-avaliante *NIL* para significar *FALSO*. Qualquer outra coisa significa *VERDADEIRO*.

Nós usualmente utilizaremos o símbolo auto-avaliante *t* para significar *TRUE*.

Lisp provê uma série de operadores booleanos como *and*, *or* e *not*. Os operadores *and* e *or' são curto-circuitantes. AND não vai avaliar quaisquer argumentos à direita daquele que faz a função avaliar para* NIL*, enquanto* OR *não avalia nenhum à direita do primeiro verdadeiro.*

Lisp também provê uma série de *forms* para execução condicional. O mais simples é o *IF*, onde o primeiro argumento determina se o segundo ou o terceiro será avaliado.

Exemplos:

```
1 > (if t 5 6)
2 5
3 > (if nil 5 6)
4 6
5 > (if 4 5 6)
6 5
```

Se você necessita colocar mais de um comando em uma das cláusulas, então use o *form progn*. *Progn* executa cada comando em seu corpo e retorna o valor do último.

```
 1 > (setq a 7)
 2 7
 3 > (setq b 0)
 4 0
 5 > (setq c 5)
 6 5
 7 > (if (> a 5)
 8     (progn
 9       (setq a (+ b 7))
10       (setq b (+ c 8)))
11     (setq b 4)
12   )
13 13
```

Um *if* que não possui uma clausula *then* ou uma cláusula *else* pode ser escrito utilizando-se *when* ou *unless*:

```
1 > (when t 3)
2 3
3 > (when nil 3)
4 NIL
5 > (unless t 3)
6 NIL
7 > (unless nil 3)
8 3
```

When e *unless*, ao contrário de *if*, aceitam qualquer número de comandos em seus corpos. O *form (when x a b c)* é equivalente a *(if x (progn a b c))*.

```
1 > (when t
2     (setq a 5)
3     (+ a 6)
4   )
5 11
```

Condicionais mais complexos podem ser construídos através do *form cond*, que é equivalente a *if ... else if ... fi*.

Um *cond* consiste de símbolo *con* seguido por um número de cláusulas-cond, cada qual é uma lista. O primeiro elemento de uma cláusula-cond é a condição, os lementos restantes são a ação.

O *cond* encontra a primeira cláusula que avalia para true, executando a ação respectiva e retornando o valor resultante. Nenhuma das restantes é avaliada.

```
1 > (setq a 3)
2 3
3 > (cond
4    ((evenp a) a)                ;if a is even return a
5    ((> a 7) (/ a 2));else if a is bigger than 7 return a/2
6    ((< a 5) (- a 1));else if a is smaller than 5 return a-1
7    (t 17)               ;else return 17
8   )
9 2
```

Se não há nenhuma ação na cláusula *cond* selecionada, *cond* retorna o valor verdadeiro:

```
1 > (cond ((+ 3 4)))
2 7
```

O comando Lisp case é semelhante a um *"C" switch statement*:

```
1 > (setq x 'b)
2 B
3 > (case x
4    (a 5)
5    ((d e) 7)
6    ((b f) 3)
7    (otherwise 9)
8   )
9 3
```

A cláusula *otherwise* significa que se x não for nem a, b, d, e, ou f, o *case* vai retornar 9.

Macros

Além de *progn* existem MACROS em Lisp para definir blocos. Uma muito usada é *prog*, que permite, entre outras coisas, a declaração explícita de variáveis locais, além de retorno explícito:

```
1 (defun F2 nil
2    (prog (i j) ; define i e j como variáveis locais inicializadas com nil
3       (setq i (read))
4       (setq j (read)
5       (return (print (- i j)))
6    )
7 )
```

Iteração

A construção de iteração mais simples em LISP é loop: um loop repetidamente executa seu corpo até que ele encontre um *form* especial do tipo return:

```
 1 > (setq a 4)
 2 4
 3 > (loop
 4    (setq a (+ a 1))
 5    (when (> a 7) (return a))
 6   )
 7 8
 8 > (loop
 9    (setq a (- a 1))
10    (when (< a 3) (return))
11   )
12 NIL
```

O próximo mais simples é a *dolist*, que tem o papel de *atar* uma variável aos elementos de uma lista na sua ordem e termina quando encontra o fim da lista:

```
1 > (dolist (x '(a b c)) (print x))
2 A
3 B
4 C
5 NIL
```

Dolist sempre retorna *NIL* como valor. Observe que o valor de X no exemplo acima nunca é *NIL*, o valor *NIL* abaixo do C é o *NIL* retornado por *dolist*, impresso pelo *read-eval-print loop*.

A primitiva de iteração mais complicada é o do. Um comando do tem a seguinte forma:

```
1 > (do ((x 1 (+ x 1)) ; variavel x, com valor inicial 1
2        (y 1 (* y 2)) ; variavel y, com valor inicial 1
3       )
4       ((> x 5) y)    ; retorna valor de y quando x > 5
5     (print y)        ; corpo
6     (print 'working) ; corpo
7   )
8 1
9 WORKING
10 2
11 WORKING
12 4
13 WORKING
14 8
15 WORKING
16 16
17 WORKING
18 32
```

- A primeira parte do do (sublinhada) especifica quais variáveis que devem ser atadas, quais são os seus valores iniciais e como eles devem ser atualizados.
- A segunda parte especifica uma condição de término (em itálico) e um valor de retorno.
- A última parte é o corpo.

Um *form* do ata suas variáveis aos seus valores iniciais da mesma forma como um *let* e então checa a condição de término. Enquanto a condição for falsa, executa o corpo repetidamente. Quando a condição se torna verdadeira, ele retorna o valor do *form* de valor-de-retorno.

O *form do** é para o do o que *let** é para *let*.

Saídas Não-Locais

O *form* especial return mencionado na seção de iteração é um exemplo de um return não-local. Outro exemplo é o *form* return-from, o qual retorna um valor da função que o envolve:

```
1 > (defun foo (x)
2     (return-from foo 3)
3     x
4   )
5 FOO
6 > (foo 17)
7 3
```

O *form return-from* pode retornar de qualquer bloco nomeado.

Funções são os únicos blocos nomeados por padrão. Você pode criar um bloco nomeado com o *form* especial *block*:

```
1 > (block foo
2     (return-from foo 7)
3     3
4   )
5 7
```

O *form* especial return pode retornar de qualquer bloco nomeado *NIL*. laços são por default nomeados *NIL*, mas você pode fazer também seus próprios blocos *NIL-nomeados*:

```
> (block nil
    (return 7)
    3
  )
7
```

Outro *form* que causa uma saída não-local é o *form* error:

```
> (error "This is an error")
Error: This is an error
```

O *form* error aplica format aos seus argumentos e então coloca você no ambiente do debugador.

2.21. Funcall, Apply, e Mapcar

Como foi dito antes, em Lisp também funções podem ser argumentos para funções. Aqui algumas funções que pedem como argumento uma função:

```
> (funcall #'+ 3 4)
7
> (apply #'+ 3 4 '(3 4))
14
> (mapcar #'not '(t nil t nil t nil))
(NIL T NIL T NIL T)
```

Funcall chama seu primeiro argumento com os argumentos restantes como argumentos deste.

Apply é semelhante a *Funcall*, exceto que seu argumento final deverá ser uma lista. Os elementos desta lista são tratados como se fossem argumentos adicionais ao *Funcall*.

O primeiro argumento a *mapcar* deve ser uma função de um argumento. *mapcar* aplica esta função a cada elemento de uma lista dada e coleta os resultados em uma outra lista.

Utilidade de *funcall*, *apply* e *mapcar*

Funcall e *apply* são principalmente úteis quando o seu primeiro argumento é uma variável.

Por exemplo, uma máquina de inferência poderia tomar uma função heurística e utilizar *funcall* ou *apply* para chamar esta função sobre uma descrição de um estado.

As funções de ordenação a serem descritas mais tarde utilizam *funcall* para chamar as suas funções de comparação.

Mapcar, juntamente com funções sem nome (veja abaixo) pode substituir muitos laços.

Lambda

Se você somente deseja criar uma função temporária e não deseja perder tempo dando-lhe um nome, lambda é justamente o que você precisa.

```
> #'(lambda (x) (+ x 3))
(LAMBDA (X) (+ X 3))
> (funcall * 5)  ; * variavel que representa o ultimo form digitado.
8
```

Observe que no contexto acima, o * é uma variável que representa o último form que foi digitado.

A combinação de *lambda* e *mapcar* pode substituir muitos laços. Por exemplo, os dois *forms* seguintes são equivalentes:

```
1 > (do ((x '(1 2 3 4 5) (cdr x))
2        (y nil))
3       ((null x) (reverse y))
4     (push (+ (car x) 2) y)
5   )
6 (3 4 5 6 7)
7 > (mapcar #'(lambda (x) (+ x 2)) '(1 2 3 4 5))
8 (3 4 5 6 7)
```

Ordenação

Lisp provê duas primitivas para ordenação: *sort* e *stable-sort*.

```
1 > (sort '(2 1 5 4 6) #'<)
2 (1 2 4 5 6)
3 > (sort '(2 1 5 4 6) #'>)
4 (6 5 4 2 1)
```

O primeiro argumento para *sort* é uma lista, o segundo é a função de comparação. A função de comparação não garante estabilidade: se há dois elementos a e b tais que *(and (not (< a b)) (not (< b a)))*, *sort* vai arranjá-los de qualquer maneira.

A função *stable-sort* é exatamente como *sort*, só que ela garante que dois elementos equivalentes vão aparecer na lista ordenada exatamente na mesma ordem em que aparecem lista original.

É importante tomar cuidado ao utilizar o *sort', pois ele tem permissão para destruir o seu argumento. Por isso, se você deseja manter a lista original, copie-a com* copy-list *ou* copy-seq.

Igualdade

Lisp tem muitos conceitos diferentes de igualdade. Igualdade numérica é denotada por =.

Como em Smalltalk ou em Prolog, existem os conceitos de identidade (mesmo objeto) e igualdade (objetos distintos, porém iguais).

Dois símbolos são *eq* se e somente se eles forem idênticos (identidade). Duas cópias da mesma lista não são *eq* (são dois objetos diferentes) mas são *equal* (iguais).

```
 1 > (eq 'a 'a)
 2 T
 3 > (eq 'a 'b)
 4 NIL
 5 > (= 3 4)
 6 T
 7 > (eq '(a b c) '(a b c))
 8 NIL
 9 > (equal '(a b c) '(a b c))
10 T
11 > (eql 'a 'a)
12 T
13 > (eql 3 3)
14 T
```

O predicado *eql* é equivalente a *eq* para símbolos e a = para números. É a identidade que serve tanto para números como para símbolos.

O predicado *equal* é equivalente *eql* para símbolos e números.

Ele é verdadeiro para dois *conses*, se e somente se, seus *cars* são *equal* e seus *cdrs* são *equal*.

Ele é verdadeiro para duas estruturas se e somente se as estruturas forem do mesmo tipo e seus campos correspondentes forem *equal*.

Exemplos de Fixação: Igualdade e Identidade

```
1 > (setq X '(A B))
2 (A B)
3 > (setq Y '(A B))
4 (A B)
5 > (equal X Y)           ;X e Y são iguais
6 T
7 > (eq X Y)              ;X e Y não são idênticos
8 NIL
```

```
1 > (setq X '(A B))
2 (A B)
3 > (setq Y X)
4 (A B)
5 > (equal X Y)           ;X e Y são iguais
6 T
7 > (eq X Y)              ;X e Y agora são idênticos
8 T
```

Algumas Funções de Lista Úteis

Todas as funções abaixo manipulam listas:

```
 1 > (append '(1 2 3) '(4 5 6)) ; concatena listas
 2 (1 2 3 4 5 6)
 3 > (reverse '(1 2 3)) ; reverte os elementos
 4 (3 2 1)
 5 > (member 'a '(b d a c)) ; pertinência a conjunto retorna primeira cauda cujo car é o elemento desejado
 6 (A C)
 7 > (find 'a '(b d a c)) ; outro set membership
 8 A
 9 > (find '(a b) '((a d) (a d e) (a b d e) ()) :test #'subsetp) ; find é mais flexível
10 (A B D E)
11 > (subsetp '(a b) '(a d e)) ; set containment
12 NIL
13 > (intersection '(a b c) '(b)) ; set intersection
14 (B)
15 > (union '(a) '(b)) ; set union
16 (A B)
17 > (set-difference '(a b) '(a)) ; diferença de conjuntos
18 (B)
```

Subsetp, *intersection*, *union*, e *set-difference* todos assumem que cada argumento não contém elementos duplicados. *(subsetp '(a a) '(a b b))* é permitido falhar, por exemplo.

Find, *subsetp*, *intersection*, *union*, e *set-difference* podem todos tomar um argumento *:test*. Por padrão, todos usam *eql*.

Utilizando *Emacs/X-Emacs* para programar Lisp

Você pode usar o *Emacs* ou *X-Emacs* para editar código Lisp. Esses editores colocam-se automaticamente em modo-Lisp quando você carrega um arquivo que termina em .lisp, mas se o seu não o faz, você pode digitar M-x lisp-mode (ALT-x lisp-mode ou META-x lisp-mode).

Você pode rodar Lisp no *Xemacs* também, usando-o como um ambiente de programação: certifique-se de que há um comando chamado "lisp" que roda seu Lisp favorito. Por exemplo, você poderia digitar o seguinte atalho:

ln -s /usr/local/bin/clisp ~/bin/lisp </syntaxhighlight>

Então, uma vez no *Xemacs*, digite META-x run-lisp (ALT-x run-lisp). Você pode enviar código Lisp para o Lisp que você acabou de invocar e fazer toda uma série de outras coisas. Para maiores informações, digite C-h m (Control-h m) de qualquer *buffer* que esteja em modo-LISP.

Outra opção mais simples e utilizar o Ponto de Menu clisp que aparece no menu Tools.

Na verdade, você nem precisa criar o link simbólico (atalho) acima. O *Emacs* possui uma variável chamada inferior-lisp-program; assim você só precisa adicionar algumas linhas de código ao seu arquivo .emacs, Emacs vai saber encontrar CLISP quando você digita ALT-x run-lisp.

Importante: Você precisa de configurar o seu *Xemacs* para trabalhar com o pacote *Ilisp*, para que possa executar o *CLISP* sob o *Xemacs*.

Exercícios

Parte 1 (até listas)

1. Desenhe as representações internas de dados para as listas seguintes:

```
1 (A 17 -3)
2 ((A 5 C) %)
3 ((A 5 C) (%))
4 (NIL 6 A)
5 ((A B))
6 (* ( + 15 (- 6 4)) -3)
```

2. Qual é o CAR de cada uma das listas do exercício anterior?

3. Qual é o CDR de cada uma das listas do exercício anterior 1?

4. Escreva as declarações necessárias, usando CAR e CDR, para obter os valores seguintes das listas do exercício 1:

```
 1 (-3)
 2 (-3 -3)
 3 (C %)
 4 (A C %)
 5 (5 %)
 6 (5 (%))
 7 (6 (6))
 8 (6 (6) 6)
 9 ((B) A)
10 (A ((B) B))
```

5. Defina uma representação conveniente na forma de lista para um conjunto de sobrenomes juntamente com os números de telefones de pessoas. O número de telefone deve permitir a inclusão de códigos de DDD e DDI para números não locais. Como resolveria o caso para as pessoas que estão na lista, mas não têm telefone?

Parte 2 (até macros)

1. Escreva uma declaração em LISP para executar cada uma das operações abaixo:

a) Ler dois números, imprimir sua soma e acrescentar 3 ao resultado. Assim 5 e 11 devem produzir 16 e 19 na tela.

b) Ler um único valor e imprimi-lo como uma lista. Assim o valor 6 deve produzir (6).

c) Ler dois valores e imprimir sua soma como uma lista. Deste modo 6 e 7 devem produzir a lista (13).

d) Ler três números e imprimi-los como uma lista.

e) Ler três números e imprimir a soma dos dois primeiros e o produto desta pelo terceiro como uma lista.

2. Escreva uma função que:

a) Devolva o valor 1 se seu parâmetro for maior que zero, -1 se for negativo, 0 se for zero.

b) Leia um nome. Se este for o mesmo nome que o dado como parâmetro, a função deve imprimir uma saudação simples e devolver o valor t. Se for diferente, não deve imprimir nada e devolver nil.

c) Dados três parâmetros, se o primeiro for um asterisco, os outros dois serão multiplicados; se for uma barra, o segundo deve ser dividido pelo terceiro; se não for nenhum dos dois, imprima uma mensagem de erro e assuma o valor zero.

d) A função deve devolver como valor o resultado da operação aritmética.

e) Devolva t se seu primeiro parâmetro estiver no conjunto de valores especificado pelo seu segundo e terceiro parâmetros e nil se não estiver. Asim: (func-4 5 5 7) = t e (func-4 6 5 7) = nil.

f) Aceite um valor simples e uma lista como parâmetros. Devolva t se o valor estiver na lista, nil caso nao esteja (este exercício pode ser resolvido de forma recursiva - pense um pouco...).

Parte 3

1. Escreva uma função que leia do usuário uma lista de produtos e seus respectivos preços, colocando-os em uma lista organizada por pares produto-preço. A entrada de dados é finalizada digitando-se a palavra `fim ao invés de um nome de produto.

2. Utilize o comando loop para implementar o laço de leitura e defina uma variável global onde a lista ficará armazenada ao fim da leitura.

3. Os pares produto-preço você pode organizar tanto como um cons, uma sublista ou uma estrutura com campos produto e preço. A list tem a vantagem de ser extremamente flexível: você pode extender a sua estrutura de dados sem necessitar entrar com os dados de novo. O cons é a forma mais econômica em termos de memória. A estrutura permite uma modelagem elegante. Fica a seu critério.

4. Escreva uma função ou conjunto de funções, que, através de um menu de opções, realizem as seguintes tarefas:

a) Pesquisar preço de um produto: Um ambiente onde o usuário entra com o nome de um produto e o programa ou diz que não encontrou o produto ou devolve o preço.

b) Mostrar em ordem alfabética toda a lista de produtos disponíveis com os respectivos preços, formatada na tela. A cada 20 produtos o programa deve fazer uma pausa e esperar o usuário teclar alguma coisa para continuar.

c) Fazer compras: Um ambiente onde o usuário pode entrar com nomes de produtos e quantidades que deseja comprar. Ao final o programa emite uma lista com todos os produtos comprados, total parcial e total final das compras.

Referências

- Tutorial de Lisp (http://www.dca.ufrn.br/~adelardo/lisp/): O conteúdo aqui apresentado foi extraído do tutorial de Lisp do professor Adelardo A Dantas de Medeiros da UFRN e adaptado ao formato wiki.
