

MC336 - Paradigmas de programação  
Lisp

João Meidanis

©Copyright 2013 J. Meidanis

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>4</b>
1.1	Calculando derivadas simbólicas . . . . .	4
1.2	O psiquiatra . . . . .	5
1.3	MYCIN . . . . .	6
1.4	Interpretador . . . . .	7
<b>2</b>	<b>Elementos da linguagem</b>	<b>10</b>
2.1	Tipos . . . . .	10
2.2	Números . . . . .	10
2.3	Símbolos . . . . .	11
2.4	Pares-com-ponto . . . . .	12
2.5	Representação gráfica . . . . .	12
2.6	Car, cdr e cons . . . . .	13
2.7	Coleta de lixo . . . . .	15
<b>3</b>	<b>Estrutura da linguagem</b>	<b>17</b>

3.1	Definindo funções . . . . .	17
3.2	Condicionais . . . . .	18
3.3	Variáveis locais . . . . .	20
<b>4</b>	<b>Símbolos</b>	<b>22</b>
4.1	Avaliação e valores de um símbolo . . . . .	22
4.2	Atribuindo valores a símbolos . . . . .	23
4.3	Inibindo a avaliação . . . . .	24
<b>5</b>	<b>Recursão</b>	<b>26</b>
5.1	O método do quadrado . . . . .	26
5.2	Recursão e laços . . . . .	29
<b>6</b>	<b>Aritmética</b>	<b>31</b>
6.1	Funções básicas . . . . .	31
6.2	Funções mais sofisticadas . . . . .	33
<b>7</b>	<b>Definição de funções</b>	<b>34</b>
<b>8</b>	<b>Condicionais</b>	<b>36</b>
8.1	A forma especial IF . . . . .	36
8.2	A macro COND . . . . .	37
8.3	Predicados . . . . .	38
<b>9</b>	<b>Funções para listas</b>	<b>39</b>

<b>10 Funções para conjuntos</b>	<b>44</b>
<b>11 Pacotes (módulos)</b>	<b>47</b>
11.1 Acessando símbolos de outros pacotes . . . . .	47
11.1.1 Importação e exportação . . . . .	48
11.2 Pacotes pré-definidos . . . . .	48
11.3 Definindo pacotes . . . . .	49
<b>12 Arrays e Loops</b>	<b>50</b>
12.1 Arrays . . . . .	50
12.1.1 Criando arrays . . . . .	50
12.1.2 Acessando arrays . . . . .	50
12.2 Loops . . . . .	51

# Capítulo 1

## Introdução

LISP vem de “list processing”. John McCarthy criou LISP em 1960 [1].

LISP é uma linguagem tão antiga que na época em que foi criada os computadores SÓ USAVAM LETRAS MAIÚSCULAS. Quando vieram as letras minúsculas, foi adotada a convenção de que em LISP uma letra minúscula ou maiúscula é a mesma coisa. Portanto, as palavras 'SEN', 'sen' e 'SeN' em LISP são todas equivalentes.

O padrão Common Lisp, consolidado em 1990, veio unir os vários dialetos de LISP existentes na época para formar uma linguagem bastante bem especificada. Usaremos o livro *Common Lisp: The Language*, 2a. edição, de Guy Steele, onde este padrão é detalhado com grande cuidado, como referência principal sobre LISP neste curso [3]. Site na internet (um de vários): <http://www.supelec.fr/docs/cltl/cltl2.html>

Algumas aplicações famosas escritas em LISP seguem.

### 1.1 Calculando derivadas simbólicas

Início da computação simbólica: programa que achava a derivada de uma função, mas usando símbolos ( $x, y$ , etc.)

Exemplo:

$$\frac{d}{dx}(x^2 + 3x) = 2x + 3,$$
$$\frac{d}{dx}(\log \sin x) = \frac{\cos x}{\sin x}.$$

Para denotar as funções neste programa é usada uma notação pré-fixa, típica de LISP. A função  $x^2 + 3x$ , por exemplo, é denotada por:

```
(+ (* X X) (* 3 X)).
```

Se for definida uma função chamada DERIV para calcular a derivada, pode-se pedir:

```
(deriv '(+ (* X X) (* 3 X)))
```

(note o apóstrofe antes do argumento) e o interpretador responderá:

```
(+ (* 2 x) 3)
```

ou seja,  $2x + 3$ .

## 1.2 O psiquiatra

ELIZA, que pretendia simular um psiquatra, foi talvez o primeiro programa de computador a passar o Teste de Turing. Foi escrito por Joseph Weizenbaum em 1966 [4].

Para interagir com o programa é preciso usar a língua inglesa. Embora o autor em seu artigo original tenha dito que o programa pode ser portado para outras línguas (inclusive mencionando que já naquele momento existia uma versão em alemão), as versões mais difundidas são em inglês.

O programa utiliza uma estratégia de psiquiatra Rogeriano, na qual o paciente é sempre estimulado a falar, elaborando sobre o que já disse. Eis um fragmento de conversação:

Men are all alike.  
IN WHAT WAY?  
They're always bugging us about something or other.  
CAN YOU THINK OF A SPECIFIC EXAMPLE?  
Well, my boyfriend made me come here.  
YOUR BOYFRIEND MADE YOU COME HERE

E por aí vai. As frases em maiúsculas são as respostas do programa.

Existe uma versão dentro do Emacs: `M-x doctor`.

Este programa causou tanta sensação, e ainda causa! Recentemente, alguém acoplou-o a um AOL Instant Messenger e deixou que pessoas ao acaso se plugassem para falar com ele. Os resultados foram surpreendentes! Confira na Internet.

### 1.3 MYCIN

MYCIN, um sistema pioneiro para diagnósticos médicos, foi um dos precursores dos sistemas especialistas [2].

MYCIN representa seu conhecimento sobre os sintomas e possíveis diagnósticos como um conjunto de regras da seguinte forma:

SE a infecção é bacteremia primária  
E o local da cultura é um dos locais estéreis  
E suspeita-se que a porta de entrada é o trato gastro-intestinal  
ENTÃO há evidência sugerindo (0.7) que a infecção é bacteróide.

As regras são na verdade escritas como expressões LISP. O valor 0.7 é uma estimativa de que a conclusão seja correta dadas as evidências. Se as evidências são também incertas, as suas estimativas serão combinadas com as desta regra para chegar à estimativa de correção da conclusão.

## 1.4 Interpretador

A maneira padrão de interação com uma implementação de Common Lisp é através de um laço ler-avaliar-imprimir (*read-eval-print loop*): o sistema repetidamente lê uma expressão a partir de uma fonte de entrada de dados (geralmente o teclado ou um arquivo), avalia a expressão, e imprime o(s) valor(es) em um destino de saída (geralmente a tela ou um arquivo).

Expressões são também chamadas de formas, especialmente quando são destinadas à avaliação. Uma expressão pode ser simplesmente um símbolo, e neste caso o seu valor é o valor como dado do símbolo.

Quando uma lista é avaliada (exceto nos casos de macros e formas especiais), supõe-se que seja uma chamada de função. O primeiro elemento da lista é tomado como sendo o nome da função. Todos os outros elementos da lista são tratados como expressões a serem avaliadas também; um valor é obtido de cada uma, e estes valores se tornam os argumentos da função. A função é então aplicada aos argumentos, resultando em um ou mais valores (exceto nos casos de retornos não locais). Se e quando a função retornar, os valores retornados tornam-se os valores da lista avaliada.

Por exemplo, considere a avaliação da expressão  $(+ 3 (* 4 5))$ . O símbolo  $+$  denota a função de adição, que não é uma macro nem uma forma especial. Portanto as duas expressões  $3$  e  $(* 4 5)$  são avaliadas para produzir argumentos. A expressão  $3$  resulta em  $3$ , e a expressão  $(* 4 5)$  é uma chamada de função (a função de multiplicação). Portanto as formas  $4$  e  $5$  são avaliadas, produzindo os argumentos  $4$  e  $5$  para a multiplicação. A função de multiplicação calcula o número  $20$  e retorna-o. Os valores  $3$  e  $20$  são então dados como argumentos à função de adição, que calcula e retorna o número  $23$ . Portanto indicamos  $(+ 3 (* 4 5)) \Rightarrow 23$ .

Usaremos CMUCL, a implementação do Common Lisp feita pela Carnegie Mellon University, EUA.

Onde pegar: <http://www.cons.org/cmucl/credits.html>. Como instalar: faça *download* do site e siga as instruções. Nota: apenas para Unix/Linux. Se você vai usar Windows, há vários outros pacotes na internet. Mas cuidado: antes de entregar qualquer projeto da disciplina, teste seu código no CMUCL!



Em geral, as modificações serão poucas ou nenhuma, especialmente se você usar construções portáteis. Só serão aceitos projetos que rodem em CMUCL.

Como usar (básico):

entrar: `lisp`  
sair: `(quit)`  
usar com arquivo: escreva um arquivo de texto simples com as expressões a avaliar, dê a ele extensão `.lisp`, e chame `lisp <arquivo.lisp`  
usar sem arquivos: simplesmente tecele as expressões (ou formas) seguidas de ENTER  
repetir última expressão: `*`  
repetir penúltima expressão: `**`  
repetir antepenúltima expressão: `***`

Uso dentro do Emacs (básico):

preparar: a variável Emacs de nome `inferior-lisp-program` deve ter valor igual ao comando usado para chamar o interpretador. Se você estiver usando o CMUCL, este valor é `"lisp"`  
iniciar: `M-x run-lisp` cria um buffer Emacs onde você pode interagir com o interpretador. É como se fosse uma shell interna ao Emacs.  
sair: `(quit)`  
usar com arquivo: se você estiver editando um arquivo `.lisp`, o Emacs usará `lisp-mode` para você editá-lo, o que fornece indentação e balanceamento de parênteses, por exemplo  
avaliação imediata: se você estiver editando um arquivo `.lisp`, teclando `C-x C-e` o Emacs avalia a expressão anterior ao cursor no buffer do lisp  
outras facilidades: peça `M-x describe-bindings`

Uma grande vantagem de usar o editor Emacs é que ele balanceia os parênteses para você. E cuida também da indentação.

**Compilação vs. interpretação:** além de interpretador, o Common Lisp prevê que seja compiladas funções ou arquivos LISP, para rodarem mais rápido. Para compilar uma função `deriv`, por exemplo, use:

```
(compile 'deriv).
```

Para compilar um arquivo `deriv.lsp`, por exemplo, use:

```
(compile-file "deriv.lsp").
```

Erros: tecle 0 e volte ao nível inicial.

Depuração: para depurar seus programas, é importante saber que existe um mecanismo de trace: `(trace deriv)`. A partir disso cada chamada a `deriv` imprime o valor dos argumentos passados à função, e cada retorno imprime o valor retornado. Para cancelar o trace, use `(untrace deriv)`. Você pode tracear várias funções ao mesmo tempo: `(trace fft gcd string-upcase)`.

## Exercícios

1. Procure na internet uma versão do programa que calcula derivadas em LISP e use-o para calcular as derivadas das funções dadas como exemplo acima.
2. Converse um pouco com o psiquiatra do Emacs. Traga seus diálogos para a classe.
3. Procure na Internet a experiência feita com o psiquiatra e usuários ao acaso do AOL. Traga alguns diálogos interessantes para a classe.
4. Carregue o interpretador e peça para avaliar: `(+ 1 (* 3 4))`. Avalie diretamente e também a partir de um arquivo dentro do Emacs.

# Capítulo 2

## Elementos da linguagem

### 2.1 Tipos

Tipos em LISP: só para dados, não para variáveis. Lista completa de tipos: array, atom, base-character, bignum, bit, bit-vector, character, compiled-function, complex, cons, double-float, extended-character, fixnum, float, function, hash-table, integer, keyword, list, long-float, nil, null, number, package, pathname, random-state, ratio, rational, readtable, sequence, short-float, signed-byte, simple-array, simple-bit-vector, simple-string, simple-vector, single-float, standard-char, stream, string, symbol, t, unsigned-byte, vector.

Além destes, podem ser criados outros pelo usuário (classes, etc.).

Um desenho hierárquico dos principais tipos, incluindo todos os que usaremos neste texto, está na Figura 2.1.

Os principais são: átomos (números, símbolos, ou strings), e pares-com-ponto.

### 2.2 Números

Números: inteiros, razões, ponto flutuante, complexos.

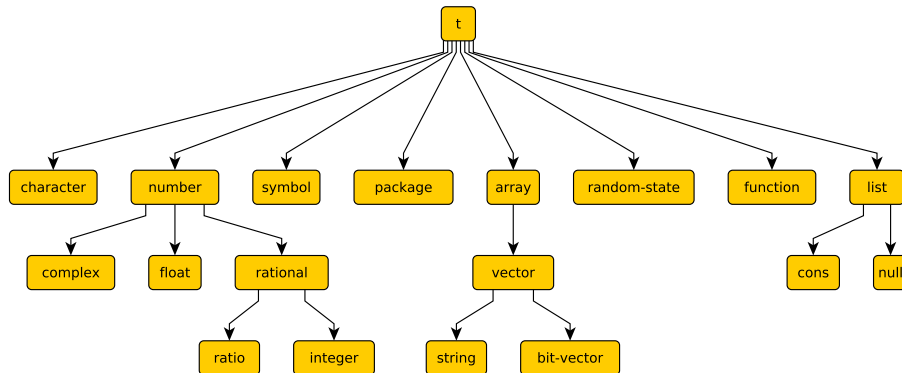


Figura 2.1: Hierarquia parcial de tipos em Lisp.

Inteiros: `fixnum` e `bignum`.

Razões: `2/3`, `-17/23`, etc.

Ponto flutuante: como se fosse `real` em Pascal, ou `float` em C e Java.

Complexos: `#C(0 1)`, a unidade imaginária  $i$ , etc.

## 2.3 Símbolos

Símbolos: serão tratados no Capítulo 4. Por enquanto, vamos considerá-los como seqüências de caracteres que podem incluir letras, números e sinais especiais, exceto brancos e parênteses, e que não possam ser confundidos com números. Além disso, não devem conter apenas pontos.

Exemplos de nomes válidos para símbolos:

```

FROBBOZ
frobboz
fRObBoz
unwind-protect

```

```
+&
1+
pascal_style
b^2-4*a*c
file.rel.43
/usr/games/zork
```

Exemplos de nomes inválidos para símbolos:

```
+1
..
```

## 2.4 Pares-com-ponto

Pares-com-ponto, ou conses: são estruturas, ou registros, com dois componentes: *car* e *cdr*. Dentre os conses, destacamos as listas.

Listas: uma lista é definida como sendo ou bem a lista vazia ou bem um cons cujo *cdr* é uma lista. O símbolo `NIL` é usado para denotar a lista vazia. A expressão `()` é sinônima de `NIL`.

Listas são denotadas escrevendo seus elementos na ordem, separados por espaços em branco (caracteres: espaço, `tab` ou `newline`) e cercados por parênteses.

Um cons cujo segundo elemento não é uma lista é denotado de forma semelhante, exceto há um ponto “.” (cercado de espaço em branco) precedendo o último *cdr*. Exemplo: `(a . 4)` é um cons cujo *car* é um símbolo e cujo *cdr* é um número. Daí o nome par-com-ponto.

## 2.5 Representação gráfica

Representação gráfica de pares-com-ponto: caixas. Há uma representação gráfica para conses. Nesta representação gráfica, colocamos uma caixa com

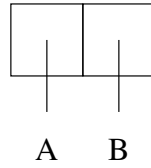


Figura 2.2: Representação gráfica de (A . B).

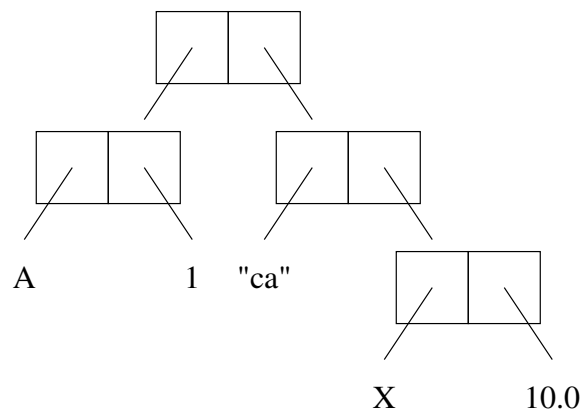


Figura 2.3: Representação gráfica do par-com-ponto cuja representação textual é: ( ( A . 1 ) . ( "ca" . ( x . 10.0 ) ) ).

dois compartimentos, cada um deles com um apontador para uma componente do cons: o compartimento esquerdo aponta para o car e o direito para o cdr.

Exemplos: veja Figuras 2.2, 2.3, 2.4, 2.5.

## 2.6 Car, cdr e cons

Vamos aprender as primeiras funções LISP: CAR, CDR e CONS. São funções pré-definidas, existentes em qualquer ambiente LISP.

A função CAR retorna o primeiro componente de um cons. FIRST é um

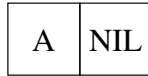


Figura 2.4: Representação gráfica de (A . NIL), ou simplesmente (A).

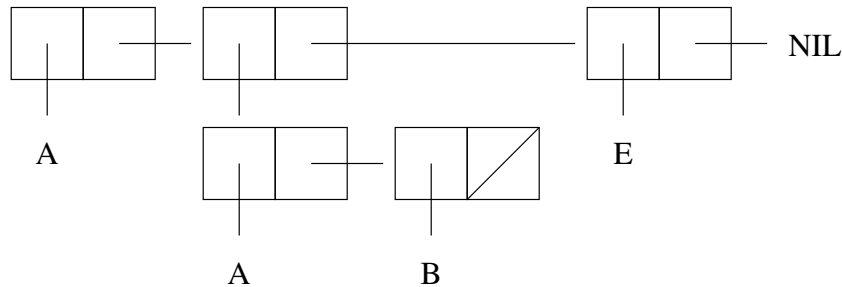


Figura 2.5: Representação gráfica de (A (A B) E). Note as várias formas de expressar ponteiros a NIL.

sinônimo de CAR. A função CDR retorna o segundo componente de um cons. REST é um sinônimo de CDR. Se CAR ou CDR forem aplicadas a (), o resultado é (). Dá erro se CAR ou CDR forem aplicadas a algo que não é um cons ou (). Exemplos:

```
(car '(A (A B) E)) => A
(car '((A . 1).("ca".(x . 10.0)))) => (A . 1)
(cdr '(A (A B) E)) => ((A B) E)
(cdr '((A . 1).("ca".(x . 10.0)))) => (("ca" x . 10.0))
```

Lembramos que a notação => indica o resultado da avaliação de uma expressão LISP.

A função CONS recebe dois argumentos e retorna uma nova caixa contendo estes argumentos como componentes, na ordem dada. Não há sinônimos para CONS. Exemplos:

```
(cons 'a 'b) => (A . B)
(cons '1 '(a b)) => (1 A B)
```

## 2.7 Coleta de lixo

A linguagem LISP foi a primeira a adotar um sistema de gerenciamento de memória conhecido como “coleta de lixo” (*garbage collection*). Neste sistema, o programador não precisa se preocupar em alocar ou desalocar memória explicitamente. Ao invés disso, o próprio interpretador da linguagem se encarrega de alocar e desalocar memória quando necessário. A alocação de memória ocorre sempre que CONS é chamada, direta ou indiretamente. Quanto à desalocação, ela ocorre quando é necessário alocar mais memória mas acabou a memória do sistema. Neste momento o interpretador sai “coletando lixo”, que são caixas que não estão sendo mais apontadas por nenhuma estrutura do sistema, e que podem portanto ser liberadas. Por exemplo, na avaliação da expressão:

$$(\text{car } '(A (A B) E)) \Rightarrow A$$

o interpretador teve que construir 5 caixas para calcular o CAR da expressão. Depois deste cálculo as caixas ficaram “soltas” no sistema, e serão recolhidas na próxima coleta de lixo.

Hoje em dia há várias linguagens que fazem coleta de lixo, entre elas Perl, Python e Java.

### Exercícios

1. Quais dos seguintes símbolos são válidos: 1-, 9.1, max-value?
2. LISP sabe mexer com frações: avalie (+ 2/3 4/5).
3. Não há limite para os inteiros em LISP: avalie  $2^{1000}$  usando a expressão (expt 2 1000).
4. Quais das seguintes expressões denotam listas: (), ( (A . B) C), (A . (B . C)).
5. Desenhe a representação gráfica de: ((A B)(C D)(E F G) H), (A 1 2), (((A 1) 2)).



6. Calcule o CAR e o CDR das expressões dos dois últimos exercícios.
7. Para formar (A B C) a partir de átomos é necessário fazer: (cons 'a (cons 'b (cons 'c ())))). Monte expressões semelhantes para os conses dos exercícios 4 e 5.

# Capítulo 3

## Estrutura da linguagem

Um programa em LISP define uma ou mais funções. As funções então são chamadas para executar o que se deseja. Por isto LISP é um exemplo de linguagem que segue a “programação funcional”.

É importante distinguir entre dois estilos de escrever código em LISP: LISP puro vs. funções com efeitos colaterais. Uma função tem efeitos colaterais quando modifica seus argumentos, ou modifica variáveis globais. Funções com efeitos colaterais são mais difíceis de entender e manter, por isso devem ser evitadas. Nesta disciplina, daremos ênfase à programação em LISP puro, sem efeitos colaterais, ou reduzindo-os ao mínimo.

### 3.1 Definindo funções

A macro DEFUN é usada para definir funções em LISP. Mais tarde veremos a sintaxe completa. Por enquanto usaremos a sintaxe básica a seguir:

```
defun nome lista-de-args corpo
```

Vejam alguns exemplos de definição de funções: calcular o quadrado de um número e calcular o discriminante de uma equação do segundo grau.

Antes, porém, vamos relembrar como efetuar operações aritméticas em LISP. As quatro operações básicas da aritmética são implementadas pelas funções pré-definidas indicadas por +, -, \*, / e a notação é prefixa.

Para o quadrado de um número:

```
(defun quadrado (x) (* x x))
```

Para o discriminante da equação  $ax^2 + bx + c$ :

```
(defun discr (a b c) (- (* b b) (* 4 a c)))
```

Note que a função \* admite mais de dois argumentos. Neste exemplo, usamos três argumentos. Ela simplesmente multiplica todos e retorna o resultado. A função + também aceita um número variável de argumentos.

## 3.2 Condicionais

Para prosseguir no nosso estudo de LISP, precisamos conhecer os condicionais, que permitem seleção de fluxos alternativos de código. Em LISP, os mais usados são IF e COND.

A forma especial IF recebe três argumentos. Após a avaliação do primeiro, apenas um dos outros dois é avaliado. Se o valor do primeiro argumento seja diferente de NIL, o segundo argumento é escolhido; caso contrário, o terceiro argumento é escolhido. Em qualquer caso, o argumento escolhido é avaliado e tem seu valor retornado pela forma IF.

Para quem está acostumado com programação imperativa, leva um tempo para se acostumar com esta maneira nova de lidar com um IF. Ele parece mais uma função agora, e não mais um comando. Observe também que LISP não tem um tipo booleano. Para fins de verdadeiro ou falso, podemos dizer que NIL é falso, e todo o resto é verdadeiro.

Vejamos alguns exemplos. A seguinte função retorna o maior entre dois números:

```
(defun maior (x y) (if (> x y) x y))
```

Note o uso de uma nova função pré-definida: `>`. Podemos também definir uma função para retornar o maior entre três números:

```
(defun maior3 (x y z)
  (if (> x y)
      (if (> x z) x z)
      (if (> y z) y z)
  )
)
```

A forma especial `COND` tem um número qualquer de argumentos. Cada um deles é uma lista. Em cada um deles, o primeiro elemento da lista é um teste. Se o teste for verdadeiro, o segundo elemento da lista é avaliado e retornado. Os testes são avaliados em ordem, e o primeiro que der certo determina a resposta. Caso nenhum teste seja verificado, o resultado é `NIL`. É comum o uso do símbolo especial `T` no último teste, para garantir que “dali não passa” e deixar o programa mais claro. Seria como uma condição “default” dentro do `COND`. Como sempre, `NIL` é falso e qualquer outra coisa é considerada verdadeira.

Aqui também os argumentos são avaliados apenas se necessário.

Para os acostumados com programação imperativa, pode-se dizer que o `COND` está para o comando `case` (ou `switch` em C e Java) assim como o `IF` está para o comando `if`.

Vejamos os mesmos exemplos anteriores feitos com `COND`:

```
(defun maior (x y)
  (cond
    ((> x y) x)
    (t y)
  )
)
```

Observe o uso de `T` para garantir a captura do controle caso o primeiro teste falhe.

```
(defun maior3 (x y z)
  (cond
    ((and (>= x y) (>= x z)) x)
    ((and (>= y x) (>= y z)) y)
    ((and (>= z x) (>= z y)) z)
  )
)
```

Observe o uso de uma nova função: `and`, uma função lógica para a conjunção. Existe também `or`. Tivemos que usar desigualdades não estritas para garantir que algum dos números seja o vencedor.

### 3.3 Variáveis locais

Funções via de regra precisam de variáveis locais. A maneira correta de definir e utilizar variáveis locais de funções em LISP é através da forma `LET`. Esta palavra pode ser traduzida como “seja” em português, e o seu significado é exatamente este. Numa função, às vezes você gostaria de dizer: “seja  $x$  igual a ...” e continuar um cálculo baseado em  $x$ . Pois `LET` permite que façamos este tipo de construção.

Por exemplo, ao calcular as raízes de uma equação do segundo grau, precisaremos do discriminante. A função a seguir recebe os coeficientes de uma equação do segundo grau e retorna uma de suas raízes.

```
(defun raiz (a b c)
  (let ((disc (discr a b c)))
    (/ (+ (- b) (sqrt disc)) (* 2 a))
  )
)
```

Observe o uso de `sqrt`, que calcula a raiz quadrada de um número. Se ele for negativo, já sai o complexo corretamente.

Mais de uma variável local pode ser definida no mesmo bloco LET. Existe também uma variante do LET, chamada LET\*, que permite que cada variável dependa da anterior. A função a seguir utiliza este recurso e também a função pré-definida LIST para a formação de uma lista com as duas raízes da equação:

```
(defun raizes (a b c)
  (let* ( (disc (discr a b c))
          (r1 (/ (+ (- b) (sqrt disc)) (* 2 a)))
          (r2 (/ (- (- b) (sqrt disc)) (* 2 a)))
        )
    (list r1 r2)
  )
)
```

A sintaxe completa das formas LET pode ser encontrada no manual online.

## Exercícios

1. Escreva uma função para calcular a média entre dois números.
2. Modifique a função que retorna as raízes de uma equação do segundo grau para que retorne as partes real e imaginária das raízes, no caso de elas serem complexas. Suponha que os coeficientes sejam reais.

# Capítulo 4

## Símbolos

Símbolos são objetos em LISP que aparecem em vários contextos: nomes de variáveis e de funções, por exemplo. Todo símbolo tem um nome (*print name* ou *pname*). Dado um símbolo, é possível obter seu nome como uma string, e vice-versa.

Símbolos não precisam ser explicitamente criados. A primeira vez que o interpretador vê um símbolo, ele o cria automaticamente.

### 4.1 Avaliação e valores de um símbolo

Todo símbolo tem uma lista de propriedades (*property list* ou *plist*). Duas destas propriedades que nos interessam muito são o **valor como dado** e o **valor como função** de um símbolo. Estes valores são usados para avaliar expressões em LISP.

Uma expressão em LISP é um átomo ou uma lista. Se for um átomo, temos três possibilidades: número, string ou símbolo. A avaliação de um número resulta no próprio número, e o mesmo ocorre com strings. Para um símbolo, a coisa é diferente. A avaliação de um símbolo resulta no seu valor como dado. Caso o símbolo não tenha valor como dado, ocorre um erro.

Isto encerra a questão de avaliar átomos. Mas, e quanto a avaliar listas? Na

hora de avaliar, uma lista é sempre vista como a chamada de uma função. O primeiro elemento da lista é o nome da função e os elementos restantes são os argumentos. Para avaliar uma lista, LISP procura o valor como função do primeiro elemento, depois avalia os argumentos, e finalmente aplica a função nos valores dos argumentos.

Por exemplo, ao avaliar a lista

```
(+ 1 2 3)
```

LISP busca o valor como função do símbolo `+`, que resulta numa função pré-definida que soma números. Depois, LISP avalia os argumentos, que no caso são números e portanto correspondem a seus próprios valores. Finalmente, LISP aplica a função pré-definida nos valores 1, 2 e 3, resultando em 6.

## 4.2 Atribuindo valores a símbolos

Já vimos a macro `DEFUN`. Ela atribui um valor como função ao símbolo que é o nome da função. Existe uma macro análoga para valor como dado. A macro `SETF` atribui um valor como dado a um símbolo. A sintaxe de `SETF` é:

```
setf nome valor
```

Por exemplo, para armazenar 8 na variável `X`, fazemos:

```
(setf x 8)
```

Observe que esta é uma forma “impura” de modificar o valor como dado de um símbolo, pois resulta de um efeito colateral, e não guarda o valor anterior para se desfazer a ação, caso necessário. Existem outras formas de modificar o valor como dado de um símbolo que são mais “puras”, ou seja, tem efeito temporário, e o símbolo volta ao valor anterior ao final de um certo processo. São elas: o bloco `LET` e a passagem de parâmetros a uma função. Note



que tanto o LET como a chamada de uma função fazem uma “atribuição” de valores a certos símbolos, e esta atribuição é desfeita após o fim do processo correspondente (fim do bloco LET no primeiro caso; fim da execução da função, no segundo caso).

Mesmo assim, SETF é bastante usado para variáveis globais ou vetores. SETF é muito útil para armazenar expressões grandes para testar funções junto ao interpretador. No entanto, é importante evitar seu uso indiscriminado, por exemplo, em simples “tradução” de construções imperativas (de Pascal, C ou Java) para LISP.

### 4.3 Inibindo a avaliação

O interpretador LISP existe para avaliar expressões. É isso que ele foi programado para fazer. No entanto, é possível inibir este comportamento.

QUOTE é uma forma especial que retorna seu argumento sem avaliação. É utilizada para introduzir expressões constantes. É tão usada que inventaram uma abreviatura: apóstrofe precedendo uma expressão significa QUOTE aplicada a esta expressão. Exemplo:

```
(quote a) => A
'(cons 'a 'a) => (CONS (QUOTE A) (QUOTE A))
```

Existe também uma forma de especificar expressões quase constantes, isto é, expressões nas quais grande parte é constante exceto alguns pontos onde se deseja avaliação. O acento grave é usado no lugar do apóstrofe, e as sub-expressões que se deseja avaliar são precedidas de vírgula:

```
'(list (+ 1 2) ,(+ 2 3) ,(+ 3 4)) => (LIST (+ 1 2) 5 7)
```

#### Exercícios

1. Suponha que foram definidos:

```
(defun xxx (x)
  (+ 1 x))
```

```
(setf xxx 5)
```

Qual o valor das seguintes expressões?

- (a) (xxx 2)
- (b) (xxx (+ (xxx 5) 3))
- (c) (+ 4 xxx)
- (d) (xxx xxx)

2. Qual o valor das expressões:

- (a) (car '((a b c d)))
- (b) (cdr '((a b c d)))
- (c) (car (cdr (car (cdr '((((a b) (c d)) (e f)) (g h))))))
- (d) (cons (car '(a b f)) (cons (cons 'c '(x)) nil))

# Capítulo 5

## Recursão

Recursão ocorre quando uma função chama a si própria, direta ou indiretamente. Por exemplo, considere a seguinte definição de fatorial.

```
(defun fatorial (n)
  (if (= n 0)
      1
      (* n (fatorial (1- n)))
  )
)
```

Observe como existe uma chamada da própria função em seu corpo. Trata-se de uma função recursiva.

Entre as vantagens da recursão podemos citar a produção de código mais compacto, e, portanto, mais fácil de manter.

### 5.1 O método do quadrado

O método do quadrado é uma maneira de raciocinar que pode ser útil para escrever a definição de funções recursivas em LISP. A idéia do método é

usar um argumento específico para determinar como o resultado da chamada recursiva deve ser trabalhado para obter o resultado a retornar. O método tem este nome devido à figura que desenhamos para tirar as conclusões, que pode ser vista na Figura 5.1.

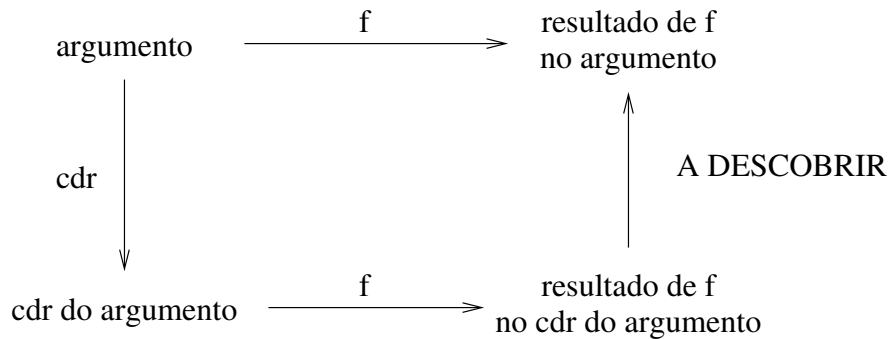
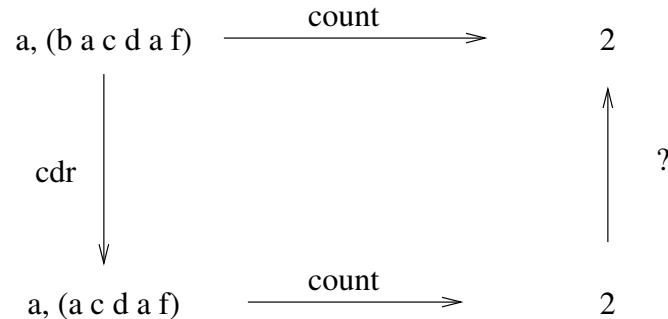


Figura 5.1: O método do quadrado.

Começamos escolhendo um argumento específico para o cálculo da função. Este argumento é colocado no canto superior esquerdo do quadrado. A partir dele, traçamos uma seta para a direita, que simboliza a aplicação da função  $f$  a definir. No canto superior direito colocamos o resultado de  $f$  aplicada ao argumento.

A seguir, descendo a partir do argumento escolhido, escrevemos o `cdr` deste argumento no canto inferior do quadrado. Traçamos uma seta de cima para baixo simbolizando a aplicação da função `cdr`. Na parte inferior do quadrado, traçamos outra seta simbolizando a aplicação de  $f$ , levando ao valor de  $f$  aplicada ao `cdr` do argumento. Finalmente, olhamos para o quadrado e tentamos descobrir como é que se obtém o valor no canto superior direito a partir do valor no canto inferior direito. É a seta que está indicada por ? na figura.

Vamos a um exemplo. Suponha que desejemos escrever a função `count` que recebe um item e uma lista e conta o número de ocorrências deste item na lista. Acompanhe pela Figura 5.2. Em nossa definição usaremos o nome `my-count` para a função para que, caso seja fornecida a um interpretador LISP, a função `count` do sistema não seja alterada.



? = somar 1 ou 0, conforme o car da lista  
seja ou não igual ao primeiro argumento

Figura 5.2: O método do quadrado aplicado à função count.

Escolhemos como argumentos o item `a` e a lista `(b a c d a f)`. O resultado de `count` deve ser 2, conforme ilustrado na figura. Aplicando o `cdr` ao argumento à lista dada, temos `(a c d a f)`. Mantemos o primeiro argumento inalterado, ou seja, continua sendo `a`. A aplicação de `count` na parte inferior da figura resulta em 2 também. Como fazemos para obter 2 a partir de 2? Neste caso, não é necessário fazer nada, porém, se o primeiro elemento da lista fosse igual a `a`, teríamos que somar uma unidade. Assim, temos a seguinte expressão para obter o resultado a partir da chamada recursiva:

```
(+ (my-count item (cdr lista))
   (if (equal item (car lista)) 1 0))
```

Agora falta apenas acrescentar a condição de parada: se a lista for vazia, o contador dá zero. A definição completa fica assim:

```
(defun my-count (item lista)
  (if (null lista)
      0
      (+ (my-count item (cdr lista))
         (if (equal item (car lista)) 1 0))))
```

```
)  
)  
)
```

Fazer o exemplo da função `my-sort`, utilizando *insertion sort*. Neste caso, uma nova função terá que ser criada. Isto é normal em LISP. Acabamos tendo várias funções pequenas em vez de uma grande. É mais fácil de manter.

## 5.2 Recursão e laços

A recursão pode substituir laços de repetição. Loops simples em linguagens imperativas podem ser transformados em funções LISP. Vejamos um exemplo: somar os números de 1 a  $n$ . Numa linguagem imperativa, isto poderia ser feito da seguinte forma.

```
soma ← 0  
for i ← 1 to n  
    soma ← soma + i
```

Para transformar isto em recursão em LISP, as atribuições, que são impuras, se transformam em outra forma de colocar valores em variáveis: chamadas de função, onde os parâmetros recebem valores. Para tal, precisamos que uma função auxiliar cujos parâmetros sejam as variáveis locais `soma` e `i`, além do próprio parâmetro `n` da função inicial.

```
(defun soma (n) (somaux n 0 1))  
  
(defun somaux (n soma i)  
  (if (> i n)  
      soma  
      (somaux n (+ soma i) (1+ i))  
  )  
)
```

Loops transformados em funções recursivas assim resultam numa forma de recursão especial, chamada de recursão de rabo (*tail recursion*), mais eficiente do que outras formas de recursão. Recursão de rabo ocorre quando o resultado das chamadas recursivas é retornado sem modificação pela função.

Existem construções LISP para loops, que serão tratadas mais adiante.

### Exercícios

1. Escreva a função `last`, que recebe uma lista e retorna a última caixa desta lista, ou `NIL` se a lista for vazia.
2. Escreva a função `member`, que recebe um item e uma lista e retorna o sufixo da lista a partir do ponto onde ocorre o item pela primeira vez nela, ou `NIL` caso o item não ocorra na lista.
3. Exemplo de laço para transformar em recursão: contar o número de positivos numa lista.
4. Combinação  $n$  tomados  $m$  a  $m$ .

# Capítulo 6

## Aritmética

LISP possui um repertório notável de funções aritméticas. Neste capítulo veremos algumas delas.

### 6.1 Funções básicas

Começamos pelas quatro operações básicas (+, -, \*, /) com 0, 1, 2, 3 ou mais argumentos.

A função + recebe um número qualquer de argumentos e retorna a soma de todos eles. Se houver zero argumentos, retorna 0 que é o elemento neutro para a adição.

A função - subtrai do primeiro argumento todos os outros, exceto quando há só um argumento: daí ela retorna o oposto dele. É um erro chamá-la com zero argumentos.

A função \* recebe um número qualquer de argumentos e retorna o produto de todos eles. Se houver zero argumentos, retorna 1 que é o elemento neutro para a multiplicação.

A função / divide o primeiro argumento sucessivamente por cada um dos outros, exceto quando há só um argumento: daí ela retorna o inverso dele.



É um erro chamá-la com zero argumentos. Dá erro também se houver só um argumento e ele for zero, ou se houver mais de um argumento e algum dos divisores for zero. Esta função produz razões se os argumentos são inteiros mas o resultado não é inteiro.

```
(+) => 0  
(+ 3) => 3  
(+ 3 5) => 8  
(+ 3 5 6) => 14
```

```
(-) => ERRO  
(- 3) => -3  
(- 3 5) => -2  
(- 3 5 6) => -8
```

```
(*) => 1  
(* 3) => 3  
(* 3 5) => 15  
(* 3 5 6) => 90
```

```
(/) => ERRO  
(/ 3) => 1/3  
(/ 3 5) => 3/5  
(/ 3 5 6) => 1/10
```

Arredondamento e truncamento: `floor` arredonda para baixo; `ceiling` arredonda para cima; `truncate` arredonda em direção ao zero; `round` arredonda para o inteiro mais próximo, e escolhe o par se houver dois inteiros mais próximos.

Funções `1+` e `1-`: incremento e decremento.

Funções `gcd` e `lcm`: máximo divisor comum (*greatest common divisor*) e mínimo múltiplo comum (*least common multiple*). Aceitam um número qualquer de argumentos inteiros, positivos ou negativos, retornando sempre um número positivo ou nulo. Com zero argumentos, retornam os elementos neutros para as respectivas operações: 0 e 1.

Função `abs`: retorna o valor absoluto de seu argumento.

## 6.2 Funções mais sofisticadas

Quadrado e raiz quadrada: `sqr`, `sqrt`.

Funções exponenciais: `exp`, calcula  $e^x$ ; `expt`, calcula  $x^y$ .

Funções logarítmicas: `log`, calcula  $\log_y x$ . O segundo argumento é opcional, e caso seja omitido a base default é  $e = 2.7182817\dots$ , a base dos logaritmos neperianos ou naturais.

Funções trigonométricas: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`. Há também a constante `pi`.

Números complexos; função `conjugate`, `realpart`, `imagart`. Todas as funções acima que faça sentido aplicar em complexos aceitam complexos e produzem as respostas corretas. Algumas delas não faz sentido aplicar em complexos: arredondar e truncar, por exemplo.

### Exercícios

1. Abra um livro do ensino médio sobre frações e use LISP para resolver expressões complicadas envolvendo frações. Confira as respostas com as do livro.
2. Faça o mesmo para números complexos.

# Capítulo 7

## Definição de funções

Função vs. macro (DEFUN e DEFMACRO): macro apenas expande e não avalia nada.

Argumentos: avaliados e não avaliados. Os argumentos de uma função são sempre avaliados antes de serem passados para ela. Numa macro, não são, mas macro é só para expandir. É só uma conveniência de notação. O próprio DEFUN é uma macro. Formas especiais também não avaliam seus argumentos (exemplo: condicionais).

Argumentos opcionais: têm que ser sempre os últimos, e são introduzidos através da chave `&optional`.

Número variável de argumentos: a chave `&rest` junta todos os argumentos restantes numa lista e passa esta lista à função.

PROGN implícito no corpo de uma função.

### Exercícios

1. Escreva uma função que aceite argumentos opcionais.
2. Escreva uma função `mais` que aceite um número qualquer de argumentos numéricos e retorne a soma de todos eles.

3. Escreva uma função que aceite argumentos através de chaves e não de posição.

# Capítulo 8

## Condicionais

Neste capítulo veremos certas expressões condicionais que servem para testar condições que selecionarão uma entre várias expressões a avaliar.

### 8.1 A forma especial IF

A forma especial IF recebe em geral três argumentos. Inicialmente, o primeiro argumento é avaliado. Caso seja verdadeiro (isto é, seu valor é diferente de NIL), o segundo argumento é avaliado e seu valor retornado pelo IF. Se a avaliação do primeiro argumento resultar em falso (isto é, NIL), então o terceiro argumento é avaliado e seu valor retornado pelo IF. Assim, o primeiro argumento funciona como uma condição que determina quem será escolhido como valor final: o segundo argumento ou o terceiro argumento. O terceiro argumento de IF pode ser omitido, caso em que é considerado igual a NIL.

IF é chamada de *forma especial* porque não necessariamente avalia todos os seus argumentos, ao contrário das funções.

Embora qualquer valor diferente de NIL seja considerado verdadeiro, existe o valor especial T que é geralmente usado quando uma função precisa retornar um valor verdadeiro específico (por exemplo, as funções de comparação <, >, etc.).

Os símbolos NIL e T são constantes em Common LISP. Seus valores são NIL e T, respectivamente, e não podem ser modificados.

## 8.2 A macro COND

A macro COND implementa uma espécie de “case”, ou seja, uma seleção entre múltiplas alternativas. O formato geral de um COND é a seguinte:

```
cond {(teste {forma}*)}* 
```

onde as estelas indicam repetição zero ou mais vezes. Assim, a macro COND tem um número qualquer de *cláusulas*, sendo cada uma delas uma lista de formas (expressões a avaliar). Uma cláusula consiste de um *teste* seguido de zero ou mais *conseqüentes*.

A avaliação de um COND processa as cláusulas da primeira à última. Para cada cláusula, o teste é avaliado. Se o resultado é NIL, o processamento passa para a próxima cláusula. Caso contrário, cada um dos conseqüentes desta cláusula é avaliado e o valor do último é retornado pelo COND. Se não houver conseqüentes, o valor do teste (que é necessariamente diferente de NIL) é retornado.

Se as cláusulas acabarem sem que nenhum teste seja verdadeiro, o valor retornado pelo COND é NIL. É comum colocar-se T como teste da última cláusula, para evitar que isto aconteça.

A macro OR recebe zero ou mais argumentos e fornece uma espécie de “ou” lógico. Porém, ao invés de retornar somente T ou NIL, a forma OR retorna algo que pode ser mais útil em determinadas situações.

Especificamente, OR avalia os argumentos da esquerda para a direita. Se algum deles der verdadeiro, OR retorna este valor e não avalia os demais. Se nenhum der verdadeiro, OR retorna NIL. A forma especial OR retorna portanto o valor do primeiro argumento que for diferente de NIL, não avaliando os demais.

A macro AND funciona de maneira similar: ela avalia os argumentos um a um, pára e retorna NIL ao achar o primeiro cujo valor seja NIL, e retorna o valor do último se todos forem diferentes de NIL. Um caso especial é o de zero argumentos, quando retorna T.

Note que OR a AND foram definidos de tal forma que podem preferitamente ser usados como funções booleanas. Além disto existe a função NOT, que retorna T se o argumento é Nil e retorna NIL caso contrário.

### 8.3 Predicados

Predicados são funções LISP que retornam um valor booleano. Alguns dos predicados mais importantes: `null`, `atom`, `consp`, `listp`, `numberp`. Eles testam se seus argumentos são, respectivamente: nulo, átomo, cons (par-com-ponto), lista, e número.

#### Exercícios

1. Quais das seguintes funções têm comportamento igual quando aplicadas a um único argumento: `null`, `not`, `and`, `or`.

# Capítulo 9

## Funções para listas

Algumas das funções que mencionamos abaixo vêm acompanhadas de uma definição. Nestes casos, precedemos o nome da função com `my-` para que, caso seja fornecida a um interpretador LISP, não altere a função original do sistema.

Existem as funções `first`, `second`, etc., até `tenth`: retorna o primeiro, segundo, etc. elemento da lista (há funções até para o décimo).

`nth` índice lista: retorna o n-ésimo elemento de uma lista. Os índices começam de zero.

```
(defun my-nth (indice lista)
  (if (= indice 0)
      (car lista)
      (my-nth (1- indice) (cdr lista)))
  )
)
```

`elt` lista índice: mesma coisa, só que a ordem dos argumentos é trocada.

`last` lista: retorna uma lista com o último elemento da lista dada. Se a lista dada for vazia, retorna `NIL`. Observe que esta função não retorna o último elemento, mas a última caixa. Porém, se for desejado o último elemento,



basta aplicar CAR ao resultado desta função. Foi feito desta forma para poder distinguir uma lista vazia de uma lista tendo NIL como último elemento.

```
(defun my-last (lista)
  (if (null lista)
      ()
      (if (null (cdr lista))
          lista
          (my-last (cdr lista)))
      )
  )
)
```

caar, cdar, etc. (até 6 letras a e c no meio, entre c e r): retornam a composição de até 6 aplicações de CAR e CDR. Por exemplo: (caddr x) equivale a (car (cdr (cdr x))).

length lista: retorna o comprimento (número de elementos no nível de topo) da lista.

```
(defun my-length (lista)
  (if (null lista)
      0
      (1+ (my-length (cdr lista)))
  )
)
```

member item lista: se item pertence à lista, retorna a parte final da lista começando na primeira ocorrência de item. Se item não pertence à lista, retorna NIL. Pode ser usado como predicado para saber se um certo item pertence a uma lista.

```
(defun my-member (item lista)
  (cond ((null lista) nil)
        ((equal item (car lista)) lista)
        (t (my-member item (cdr lista))))
)
```

```
)  
)
```

reverse lista: retorna uma lista com os elementos em ordem inversa relativa à ordem dada.

```
(defun my-reverse (lista)  
  (my-revappend lista ())  
)
```

```
(defun my-revappend (lista1 lista2)  
  (if (null lista1)  
      lista2  
      (my-revappend (cdr lista1) (cons (car lista1) lista2)))  
)  
)
```

list &rest args: constrói e retorna uma lista com os argumentos dados. Aceita um número qualquer de argumentos. Com zero argumentos, retorna NIL.

```
(defun my-list (&rest args)  
  args  
)
```

append &rest lists: retorna a concatenação de uma quantidade qualquer de listas.

```
(defun my-append2 (lista1 lista2)  
  (if (null lista1)  
      lista2  
      (cons (car lista1) (my-append2 (cdr lista1) lista2)))  
)  
)
```

subst novo velho arvore: substitui uma expressão por outra numa lista, em todos os níveis (por isto chamamos de árvore).

```

(defun my-subst (novo velho arvore)
  (cond ((equal velho arvore) novo)
        ((atom arvore) arvore)
        (t (cons (my-subst novo velho (car arvore))
                  (my-subst novo velho (cdr arvore))
                  )
          )
        )
  )
)

```

position item lista: retorna a primeira posição em que item aparece na lista. As posições são numeradas a partir de zero. Se o item não está na lista, retorna NIL.

```

(defun my-position (item lista)
  (cond ((null lista) nil)
        ((equal item (car lista)) 0)
        (t (let ((pos (my-position item (cdr lista))))
              (and pos (1+ pos))
            )
          )
        )
  )
)

```

count item lista: retorna o número de ocorrências do item na lista dada.

subseq lista comeco &optional final: retorna a subsequência de uma lista, a partir de uma posição dada, e opcionalmente terminando numa outra posição dada. As posições são numeradas a partir de zero.

```

(defun my-subseq (lista comeco)
  (if (= comeco 0)
      lista
      (my-subseq (cdr lista) (1- comeco))
  )
)

```

remove item lista: retorna uma nova lista obtida da lista dada pela remoção dos elementos iguais a item. A ordem relativa dos elementos restantes não é alterada.

```
(defun my-remove (item lista)
  (cond ((null lista) nil)
        ((equal item (car lista)) (my-remove item (cdr lista)))
        (t (cons (car lista) (my-remove item (cdr lista))))
  )
)
```

mapcar funcao lista: retorna uma lista composta do resultado de aplicar a função dada a cada elemento da lista dada. Nota: funções com mais de um argumento podem ser usadas, desde que sejam fornecidas tantas listas quantos argumentos a função requer. Se as listas não forem todas do mesmo tamanho, o resultado terá o comprimento da menor.

```
(defun my-mapcar1 (funcao lista)
  (if (null lista)
      nil
      (cons (funcall funcao (car lista))
            (my-mapcar1 funcao (cdr lista))
            )
  )
)
```

## Exercícios

1. Escreva a função `append` para um número qualquer de argumentos. Pode usar a função `append2` definida anteriormente.

# Capítulo 10

## Funções para conjuntos

Às vezes queremos usar listas para representar conjuntos de objetos, sem nos importarmos com a ordem deles na lista. LISP oferece suporte com várias funções que procuram imitar as operações mais comuns entre conjuntos: união, intersecção, etc.

Observe que conjuntos não têm elementos repetidos, enquanto que listas podem ter. Em cada operação, indicaremos o que ocorre quando há repetições nas listas dadas como argumentos.

As definições serão novamente com `my-` precedendo o nome da função, para permitir testes sem comprometer a função original definida no Common LISP.

`union lista1 lista2`: retorna uma lista contendo todos os elementos que estão em uma das listas dadas. Se cada uma das listas dadas não contém elementos repetidos, garante-se que o resultado não contém repetições. Contudo, se as listas de entrada tiverem elementos repetidos, o resultado pode ou não conter repetições.

```
(defun my-union (lista1 lista2)
  (cond ((null lista1) lista2)
        ((member (car lista1) lista2)
         (my-union (cdr lista1) lista2))
        (t (cons (car lista1) (my-union (cdr lista1) lista2))))
```

```
)  
)
```

`intersection lista1 lista2`: retorna uma lista com os elementos comuns a `lista1` e `lista2`. Se cada uma das listas dadas não contém elementos repetidos, garante-se que o resultado não contém repetições. Contudo, se as listas de entrada tiverem elementos repetidos, o resultado pode ou não conter repetições.

```
(defun my-intersection (lista1 lista2)  
  (cond ((null lista1) nil)  
        ((member (car lista1) lista2)  
         (cons (car lista1) (my-intersection (cdr lista1) lista2)))  
        (t (my-intersection (cdr lista1) lista2)))  
  )  
)
```

`set-difference lista1 lista2`: retorna uma lista com os elementos de `lista1` que não estão em `lista2`.

```
(defun my-set-difference (lista1 lista2)  
  (cond ((null lista1) nil)  
        ((member (car lista1) lista2)  
         (my-set-difference (cdr lista1) lista2))  
        (t (cons (car lista1) (my-set-difference (cdr lista1) lista2))))  
  )  
)
```

`subsetp lista1 lista2`: retorna verdadeiro quando cada elemento de `lista1` aparece na `lista2`.

```
(defun my-subsetp (lista1 lista2)  
  (cond ((null lista1) t)  
        ((member (car lista1) lista2)  
         (my-subsetp (cdr lista1) lista2))  
        (t nil))  
  )  
)
```

Todas estas funções admitem outras variações através de parâmetros adicionais ou maneiras diferentes de testar igualdade. Consulte a documentação oficial do Common Lisp para mais detalhes.

## Exercícios

1. Escreva uma função `prod-cart` que retorne o produto cartesiano de duas listas dadas, onde os pares ordenados são representados por listas de dois elementos. Exemplo:

```
> (prod-cart '(a b) '(1 2 3))  
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3))
```

A ordem dos pares no resultado não é relevante.

# Capítulo 11

## Pacotes (módulos)

Os objetivos do sistema de pacotes em Lisp são: oferecer modularidade e divisão do espaço de nomes.

Um pacote é um mapeamento de nomes para símbolos. A cada momento, apenas um pacote é usado para este mapeamento, o pacote corrente. A variável global `*package*` contém como valor o pacote corrente. Cada pacote tem seu nome. A lista de todos os pacotes existentes no sistema num certo momento pode ser obtida chamando-se a função pré-definida `list-all-packages`. Quando um argumento para uma função ou macro é um pacote, ele geralmente é dado como um símbolo cujo nome é o nome do pacote.

Um pacote tem símbolos internos e externos. Os símbolos externos de um pacote são a sua interface com o mundo exterior, por isso devem ter nomes bem escolhidos e anunciados a todos os usuários do sistema. Símbolos internos são para uso interno apenas, e não devem ser acessados por outros pacotes.

### 11.1 Acessando símbolos de outros pacotes

Freqüentemente, é necessário fazer referências a símbolos de outros pacotes que não o corrente. Para acessar tais símbolos, é usado o mecanismo de



*qualificação*. Isto é feito usando nomes *qualificados*, que são formados por um nome de pacote, dois pontos (“:”), e o nome do símbolo. Por exemplo, `jogador:iniciar` refere-se a um símbolo de nome `iniciar` num pacote chamando `jogador`. Para isto, o símbolo deve ser um símbolo externo do referido pacote.

### 11.1.1 Importação e exportação

Por vezes, é importante referir-se a símbolos de outros pacotes sem qualificação. Para este fim, existe a exportação e a importação. Um símbolo de um pacote A, para ser visível pelo pacote B, deve ser exportado por A e importado por B. Vejamos como fazer isto.

A função `export` recebe um símbolo e o torna externo a um pacote. Ele pode então ser importado por outros pacotes.

Para importar símbolos, existe a função `import`, que recebe uma lista de símbolos e os importa no pacote corrente. A partir daí, eles podem ser usados sem qualificação. Outra possibilidade é utilizar a função `use-package`, que internaliza no pacote corrente todos os símbolos externos do pacote usado como argumento. A partir daí, eles podem ser usados sem qualificação.

## 11.2 Pacotes pré-definidos

Alguns pacotes já vem pré-definidos no Common Lisp: `common-lisp`, um pacote que contém as funções básicas do Common Lisp; `common-lisp-user`, o pacote default onde as definições do usuário são colocadas; `keywords`, pacote das palavras-chave (*keywords*), que são símbolos iniciados em dois pontos (“:”), cujo valor como dado são eles mesmos e são constantes (não podem ter seu valor modificado).

## 11.3 Definindo pacotes

A função `make-package` é usada para criar novos pacotes. Tipicamente, ela recebe como argumento um símbolo, cujo nome será o nome do novo pacote. A função `delete-package` remove do sistema o pacote especificado. A macro `in-package` faz com que o seu argumento passe a ser o pacote corrente.

Para colocar num novo pacote um lote de definições, basta preceder as definições pelas linhas de código abaixo:

```
(make-package 'pacote)
(in-package pacote)
(use-package 'common-lisp)
```

A seguir, pode-se colocar uma chamada de `export` para exportar os símbolos que se deseja exportar.

### Exercícios

1. Crie um novo pacote, defina nele uma função e exporte esta função. Depois vá para o pacote `common-lisp-user` e chame esta função usando a notação `pacote:funcao`.

# Capítulo 12

## Arrays e Loops

### 12.1 Arrays

#### 12.1.1 Criando arrays

A função `make-array` retorna uma nova array. Seu parâmetro é uma lista de dimensões, por exemplo, `(make-array '(4 3 7))` retorna uma array tridimensional onde o primeiro índice vai de 0 a 3, o segundo vai de 0 a 2, e o terceiro vai de 0 a 6.

#### 12.1.2 Acessando arrays

A função `aref` recebe uma array e índices e retorna o elemento da array na posição especificada pelos índices. Por exemplo, se a variável `mat` contiver a array criada no parágrafo anterior, a chamada `(aref mat 2 1 6)` retorna o elemento indexado por 2, 1 e 6 (equivalente a algo como `mat[2][1][6]` em C ou `mat[2,1,6]` em Pascal).

A macro `setf` pode ser usada com `aref` para modificar o elemento numa dada posição de uma array.

## 12.2 Loops

Há várias formas de fazer construções iterativas em Lisp, mas aqui vamos apenas dar dois exemplos: as macros `dolist` e `dotimes`. O leitor interessado poderá encontrar outras construções mais complexas no manual de Common Lisp.

A forma `dolist` executa iteração sobre os elementos de uma lista. O formato geral é:

```
dolist (var listform [resultform])
      {declaration}* {tag | statement}*
```

Primeiramente, a forma *listform* é avaliada e deve produzir uma lista. Em seguida o corpo é executado uma vez para cada elemento da lista, com a variável *var* tendo como valor este elemento. Então a forma *resultform* é avaliada e seu valor é retornado pela macro `dolist`.

A forma `dotimes` executa iteração sobre uma seqüência de inteiros. O formato geral é:

```
dotimes (var countform [resultform])
      {declaration}* {tag | statement}*
```

Primeiramente, a forma *countform* é avaliada e deve produzir um inteiro. Em seguida o corpo é executado uma vez para cada inteiro de zero ao valor deste inteiro menos um, com a variável *var* tendo como valor este inteiro. Então a forma *resultform* é avaliada e seu valor é retornado pela macro `dotimes`.

Em ambos os casos, se *resultform* é omitida, o resultado é `NIL`.

### Exercícios

1. Escreva uma expressão que imprime os elementos de uma lista usando `dolist`.
2. Escreva uma expressão que some os elementos de 0 a *n* usando `dotimes`.

# Bibliografia

- [1] John L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.
- [2] Edward H. Shortliffe. *Computer-Based Medical Consultation: MYCIN*. Elsevier North Holland, 1976.
- [3] Guy L. Steele. *Common Lisp the Language*. Digital Press, second edition, 1990. ISBN 1-55558-041-6.
- [4] Joseph Weizenbaum. ELIZA - a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, January 1966.