MC346 - Paradigmas de programação
Lisp

João Meidanis

Conteúdo

Capítulo 1

Introdução

LISP vem de “list processing”. John McCarthy criou LISP em 1960 [4].

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 e sua versão maiúscula são a mesma coisa. Portanto, as palavras ’SEN’, ’sen’ e ’SeN’, por exemplo, são todas equivalentes em LISP.

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 [6]. Vários sites na internet contém cópias deste trabalho seminal.

Algumas aplicações famosas escritas em LISP seguem.

1.1 1960 — Calculando derivadas simbólicas

Marcando o início da computação simbólica, o criador de Lisp escreveu o primeiro 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 — isto será explicado na Seção 4.3) e o interpretador responderá:

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

ou seja, $2x+3$.

1.2 1966 — O psiquiatra

Alan Turing, em meados do século XX, imaginou um teste para decidir se um programa exibia "inteligência" em medida semelhante a um humano. Tratava-se de colocar um ser humano em comunicação privada durante algum tempo, primeiro com um outro humano e depois com o programa (ou vice-versa). Ao final deste tempo, se o humano julgador não conseguisse distinguir com certeza quem era o computador, o programa teria passado no teste.

ELIZA, que pretendia simular um psiquatra, foi talvez o primeiro programa de computador à altura do Teste de Turing. Foi escrito por Joseph Weizenbaum em 1966 [7].

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.

Este programa causou enorme sensação, e até hoje causa. Já na era da Internet, alguém teve a ideia de colocá-lo numa sala de chat e deixar que pessoas ao acaso se plugassem para falar com ele. É interessante notar que as pessoas levam um tempo para perceber que quem está do outro lado da linha não é uma pessoa.

1.3 1976 — MYCIN

MYCIN, um sistema pioneiro para diagnósticos médicos, foi um dos precursores dos sistemas especialistas [5].

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 1995 — Viaweb

Lisp não serve *apenas* para aplicações em inteligência artificial. A liguagem é extremamente expressiva, de forma que programas escritos nela são geralmente mais curtos que programas para o mesmo fim escritos em linguagens imperativas como C++ ou Java.

O software Viaweb foi um dos primeiros a possibilitar aos usuários a criação de lojas online na internet. A companhia homônima criada em 1995 para comercializá-lo foi mais tarde vendida ao grupo Yahoo! por 49 milhões de dólares.

O interessante é que este software foi escrito quase que inteiramente em Common Lisp. Seu principal criador, Paul Graham, tem livros escritos sobre Lisp [1, 2] e também sobre empreendedorismo [3]. Ele e seus colegas da Viaweb formaram um fundo de capital para apoiar jovens empreendedores. Sua página na internet contém inúmeros artigos curtos sobre Lisp, empreendedorismo, e vários outros assuntos, com ideias extremamente interessantes e bem fundamentadas do ponto de vista lógico, embora às vezes controversas.

1.5 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` mesmo, 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`. Indicamos esta avaliação toda escrevendo:

```
(+ 3 (* 4 5)) => 23.
```

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 algum psiquiatra baseado em ELIZA. Traga seus diálogos para a classe. Ou procure na Internet diálogos de outras pessoas com tais programas, e traga alguns diálogos interessantes para a classe.
3. Carregue o interpretador e peça para avaliar: `(+ 1 (* 3 4))`.

Capítulo 2

Elementos da linguagem

2.1 Tipos

Lisp é uma linguagem onde as varáveis não tem tipo, mas os dados sim. Eis uma lista completa dos tipos em Common Lisp, em ordem alfabética:

```
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. Vamos estudá-los nas próximas seções.

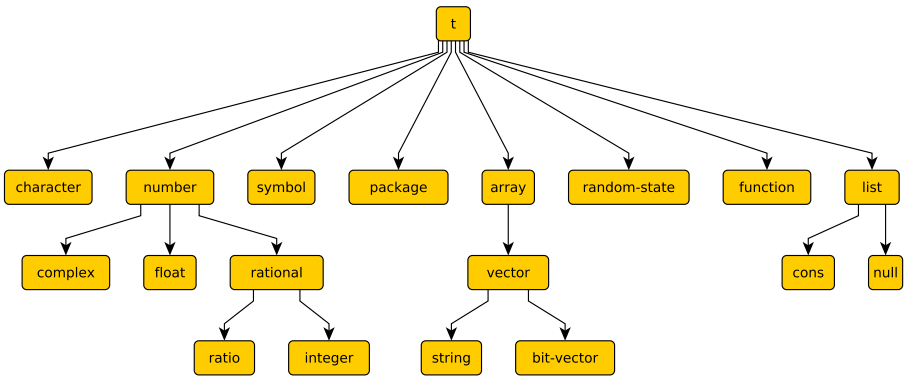

Figura 2.1: Hierarquia parcial de tipos em Lisp.

2.2 Números

Números em Lisp incluem: inteiros, razões, ponto flutuante, complexos.

Para inteiros, temos dois subtipos: fixnum e bignum. Fixnum são inteiros que cabem em 32 ou 64 bits (dependendo da implementação), e são ideais para quando se deseja rapidez nos cálculos. Bignums são inteiros de tamanho ilimitado.

Razões são números racionais, ou seja, que podem ser escritos como o quaociente entre dois inteiros. Lisp aceita que sejam entrados como tais quocientes, por exemplo: `2/3`, `-17/23`, etc. As operações aritméticas feitas com razões procuram manter o formato racional, levando a cálculos de "precisão infinita", ou seja, sem arredondamento.

Números em ponto flutuante são como o tipo `real` em Pascal, ou `float` em C e Java.

Lisp suporta nativamente números complexos. Por exemplo, `#C(0 1)` representa a unidade imaginária *i*. Todas as operações aritméticas podem ser efetuadas sobre tais números. Algumas funções, como raiz quadrada e arco-seno, podem também produzir complexos.

2.3 Símbolos

Símbolos serão tratados com mais detalhes no Capítulo 4. Por enquanto, vamos considerá-los como sequências 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.

Uma lista é definida como sendo a lista vazia ou 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 espaço branco e cercando a lista toda por um par de parênteses, por exemplo `(a b 2 c)`.

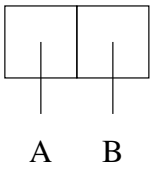

Figura 2.2: Representação gráfica de `(A . B)`.

Para separar os elementos de uma lista podem ser usadas sequências arbitrárias de caracteres espaço, tab ou newline, desde que tenham pelo menos um caractere.

Um cons cujo segundo elemento não é uma lista é denotado de forma semelhante, exceto que 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

Há uma representação gráfica bastante popular para conses, onde usamos uma caixa com 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 podem ser vistos nas 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 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:

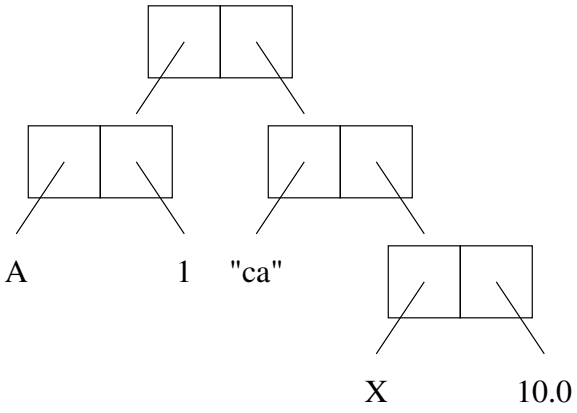

Figura 2.3: Representação gráfica do par-com-ponto cuja representação textual é: `( ( A . 1 ) . ( "ca" . ( x . 10.0 ) ) )`.

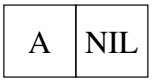

Figura 2.4: Representação gráfica de `(A . NIL)`, ou simplesmente `(A)`.

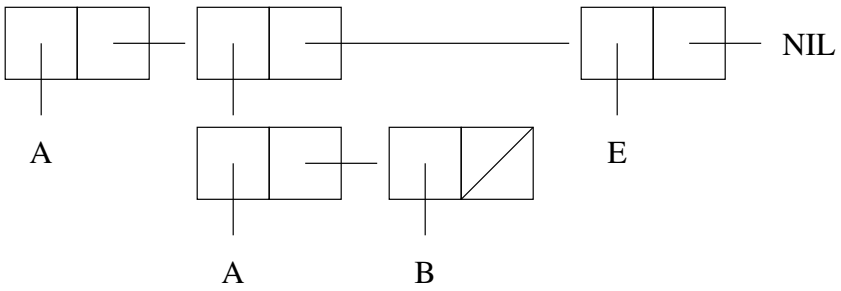

Figura 2.5: Representação gráfica de `( A ( A B ) E )`. Note as várias formas de expressar ponteiros a `NIL`.

```
(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 . 9.0)))) => (("ca" x . 9.0))
```

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 liguagem LISP foi a primeira a adotar um sistema de gerenciamento de memória conhecido como "coleção 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:

```
(car '(A (A B) E)) => 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?
   (a) `1-`
   (b) `9.1`
   (c) `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) `( (A . B) C)`
   (c) `(A . (B . C))`
5. Desenhe a representação gráfica de:
   (a) `((A B)(C D)(E F G) H)`
   (b) `(A 1 2)`
   (c) `(((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 acima.

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: funções puras vs. funções com efeitos colaterais. Uma função tem efeitos colaterais quando modifica seus argumentos, ou modifica variáveis globais. Funções com efeitos colaterais são mais difícies de entender e manter, porém há casos em que temos que apelar a elas por questões de eficiência ou programabilidade. Nesta disciplina, daremos ênfase à programção 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, no Capítulo 7, veremos mais detalhes. Por enquanto, basta saber que podemos definir funções usando a sintaxe básica a seguir:

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

Vejamos alguns exemplos de definição de funções resolvendo os problemas de 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 calcular o quadrado de um número, podemos definir a seguinte função:

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

Para o discriminante da equação $ax^2 + bx + c$, podemos definir a seguinte função:

```
(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 for 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, a convenção é que NIL é falso, e todo os outros valores são verdadeiros.

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 dos testes resulte verdadeiro, o resultado do COND 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. Lembre-se que NIL é falso e qualquer outra coisa é considerada verdadeira, portanto poderíamos usar qualquer outro valor diferente de NIL nesta última cláusula. Mas `t` acabou por atrair a simpatia dos programadores LISP nesta situação, e tornou-se padrão.

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`, que é uma função lógica para a conjunção. Existe também `or`. Usamos 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, não há erro: o resultado é um número complexo.

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 iterpretador 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. Exemplos:

```
(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 5)) =>
            (LIST (+ 1 2) 5 -2)
```

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 portanto 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 quadradão

O método do quadradão é uma maneira de raciocinar que pode ser útil para escrever a definição de funções recursivas. 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 ao diagrama que desenhamos para tirar as conclusões, que pode ser vista na Figura 5.1.

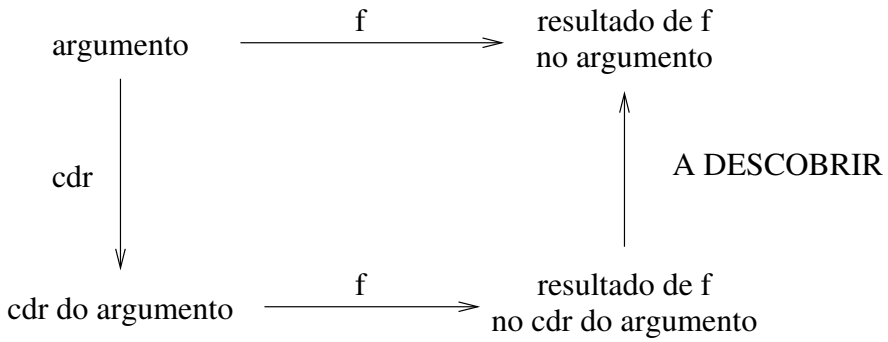

Figura 5.1: O método do quadradão.

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 quandrado, 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.

Escolhemos como argumentos o item `a` e a lista `(b a c d a f)`. O resultado de `count` deve ser 2, que foi colocado no canto superior direito na figura. Aplicando o `cdr` 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`,

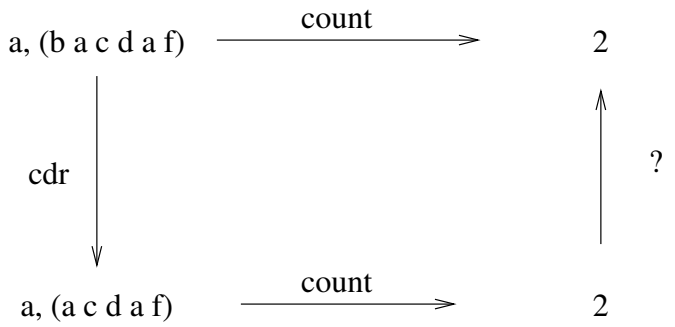

Figura 5.2: O método do quadradão aplicado à função `count`.

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)
       )
    )
  )
```

5.2 Exemplo: *insertion sort*

Vejamos mais um exemplo, no qual vamos escrever uma função recursiva chamada `my-sort` que implementa o *insertion sort*. Neste caso, uma função auxiliar terá que ser criada. Isto é normal em LISP. Acabamos tendo várias funções pequenas em vez de uma grande. É mais fácil de manter.

A função `my-sort` recebe uma lista e retorna uma outra lista com os mesmos elementos, mas ordenados. Vamos imaginar que os elementos são inteiros, para facilitar. Estamos então prontos para desenhar nosso quadradão. Acompanhe pela Figura 5.3.

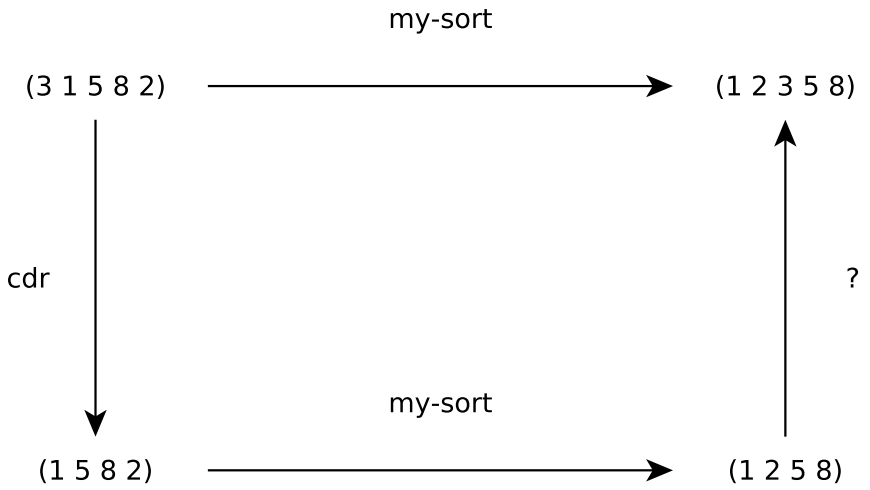

Figura 5.3: O método do quadradão aplicado a uma função que implementa *insertion sort*.

A pergunta que devemos nos fazer agora é: que função deve ser usada para transformar `(1 2 5 8)` em `(1 2 3 5 8)`. A resposta é clara: devemos inserir `3`, que é o car da lista inicial, em sua posição certa na lista ordenada `(1 2 5 8)`. Aí é que entra a função auxiliar. Se a chamarmos de `insert`, o esqueleto da função `my-sort` já está montado:

```
(defun my-sort (lista)
  (if ...
      ...
      (insert (car lista) (my-sort (cdr lista))
    )
  )
```

Falta só escrever a condição inicial e a função auxiliar `insert`, que recebe um número e uma lista ordenada e retorna uma lista um pouco maior, onde seu primeiro parâmetro está colocado no lugar certo dentro da lista dada como segundo parâmetro. Esta função auxiliar pode ela mesma ser projetada com o auxílio do método do quadradão.

A condição inicial é simples. Quando a lista é vazia, sua versão ordenada é vazia também. Deixaremos os detalhes de construir a função auxiliar para o leitor. Uma possível solução final pode ser vista a seguir.

```
(defun my-sort (lista)
  (if (null lista)
      ()
      (insert (car lista) (my-sort (cdr lista))
    )
  )

(defun insert (num lista)
  (if (<= num (car lista))
    (cons num lista)
    (cons (car lista) (insert num (cdr lista)))
  )
)
```

5.3 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 cauda (*tail recursion*), mais eficiente do que outras formas de recursão. Recursão de cauda ocorre quando o resultado das chamadas recursivas é retornado sem modificação pela função.

Existem construções LISP para loops, que serão vistas mais adiante, no Capítulo 12.

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. Este exercício destina-se a testar se você aprendeu a transformar laços em recursão. Escreva uma função `contpos` para contar o número de positivos numa lista.

4. Escreva uma função `(comb n m)` que retorna o número de combinações de *n* elementos tomados *m* a *m*.

Capítulo 6

Aritmética

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

6.1 Funções básicas

Comecemos pelas quatro operações básicas (+, -, *, /). Todas elas aceitam múltiplos argumentos. Vejamos cada uma delas em mais detalhe.

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, quando 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, quando 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.

Abaixo vemos exemplos relevantes sobre as quatro operações aritméticas básicas.

```
(+) => 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
```

Há funções para 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.

As funções `1+` e `1-` são usadas para incremento e decremento, respectivamente.

As funções `gcd` e `lcm` calculam o máximo divisor comum (*greatest common divisor*) e o mínimo múltiplo comum (*least common multiple*), respectivamente. 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.

A função `abs` retorna o valor absoluto de seu argumento.

6.2 Funções mais sofisticadas

Há funções para calcular quadrados e raízes quadradas: `sqr`, `sqrt`.

Há funções exponenciais: `(exp x)`, que calcula e^x; `(expt x y)`, que calcula x^y.

Há funções logarítmicas: `(log x y)`, que calcula $\log_y x$. O segundo argumento é opcional, e, caso seja omitido, a base default é $e = 2.7182817...$, a base dos logaritmos neperianos ou naturais.

Há funções trigonométricas e suas inversas: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`. Há também funções hiperbólicas e suas inversas: `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`. Há também a constante `pi`.

Para números complexos, temos as função `conjugate`, `realpart`, `imagpart`, que na verdade podem ser aplicadas também a números que não sejam complexos. Todas as funções anteriores que faça sentido aplicar em complexos aceitam complexos e produzem as respostas corretas. Algumas delas não têm sentido quando aplicadas a complexos: arredondar e truncar, por exemplo.

Exercícios

1. O cálculo da seguinte expressão poderia aparecer como exercício num livro do ensino fundamental sobre frações e números mistos:

$$\frac{1}{2} - \frac{4}{9} + 2 - 1\frac{3}{4} + 4\frac{6}{7} - 1 + 11\frac{1}{2}.$$

   Use LISP para resolver este. A resposta deve ser 3947/252.

2. O cálculo da seguinte expressão poderia aparecer como exercício num livro do ensino médio sobre números complexos:

$$\frac{(5+7i)(3-2i)}{8+5i}.$$

   Use LISP para resolver este. A resposta deve ser $(287-57i)/89$.

Capítulo 7

Definição de funções

Como vimos na Seção 3.1, usamos a forma especial `DEFUN` para definir uma função, informando seu nome, seus argumentos e seu corpo, que é o que será avaliado quando a função for chamada. O nome da nova função é um símbolo qualquer. Os argumentos são dados numa lista e há várias modalidades de argumentos, como veremos a seguir. Por fim, o corpo é formado por usa série de formas que são avaliadas em sequência, e o valor da ultima delas é retornado. Se não houver corpo, a função sempre retorna `NIL`.

Os argumentos de uma função são sempre avaliados antes de serem passados para ela. O padrão Common Lisp prevê também a existência de macros, que são semelhantes a funções, mas são apenas regras sofisticadas para expansão de expressões, e por isso não avaliam seus argumentos. Porém, não aprofundaremos a questão das macros neste texto. Algumas formas especiais também não avaliam seus argumentos (por exemplo, os condicionais).

7.1 Expressões lambda

Funções são definidas em Lisp através da forma especial `DEFUN`. Porém, tudo o que esta forma especial faz é associar um *nome* a uma expressão lambda que determina o que a função faz. Esta expressão lambda deriva do cálculo lambda, criado por Alonzo Church na década de 1930 para claramente separar uma função

do seu valor, algo que realmente causa confusão e precisa ser bem especificado.

Para exemplificar, considere uma função que leva cada número real x em seu quadrado x^2. Como expressar esta função? Se escrevermos simplesmente x^2 estamos confundindo a função com o valor dela no ponto x. Church resolveu o problema indicando a função por $\lambda x.(x^2)$. Note o uso da letra grega λ para indicar claramente quais são os argumentos da função e qual é o resultado dela. A função $\lambda x.(x^2)$ tem um argumento, x, e o seu valor é a expressão x^2, que está entre parênteses.

Em Lisp, temos as expressões lambda que servem para indicar explicitamente os argumentos de uma função e o que ela deve calcular. A forma mais simples de uma expressão lambda em Lisp é:

```
lambda (arg1 arg2 ...) corpo
```

onde são dados uma lista de argumentos e um corpo que é uma série de formas. As formas do corpo são avaliadas uma a uma, e o valor da última é retornado como resultado da função. Para definir uma função que calcula e retorna o quadrado de seu argumento, podemos escrever:

```
(lambda (x) (* x x))
```

Esta expressão pode então ser usada para calular quadrados. Para maior conveniência, em geral preferimos atribuir um nome às funções para uso repetido, fazendo

```
(defun f (x) (* x x))
```

Porém, se, após esta definição, olharmos o valor de `f` como função, veremos algo como:

```
>#'f
(lambda (x) (* x x))
```

que é a expressão lambda associada ao símbolo `f`.

No padrão Common Lisp as definições de funções e as expressões lambda adquiriram mais funcionalidades. Foram introduzidos argumentos opcionais, argumentos restantes, argumentos por palavra-chave e argumento que são na verdade variáveis auxiliares. Vejamos a seguir exemplos destas funcionalidades.

7.2 Argumentos opcionais

É possível especificar argumentos opcionais numa expressão lambda ou numa definição de função através da chave `&optional`. Por exemplo, suponhamos que a função `f` deva receber dois parâmetros obrigatórios `a` e `b` e um opcional `c` e deva retornar a soma de todos eles. A definição poderia ser:

```
(defun f (a b &optional c)
  (if c
    (+ a b c)
    (+ a b)
  )
)
```

Nesta definição estamos usando o fato de que quando um argumento opcional não é dado, seu valor é `NIL` dentro da função. Temos então:

```
(f 5 8) => 13
(f 5 8 10) => 23
```

É permitido também especificar um valor *default* para argumentos opcionais, que será usado quando não for passado um valor para este argumento. A definição de `f` poderia então ser:

```
(defun f (a b &optional (c 0))
  (+ a b c)
)
```

com o mesmo efeito.

7.3 Argumentos restantes

É possível especificar que todos os argumentos a partir de um ponto sejam colocados numa lista, sejam quantos forem, e que esta lista seja passada como o valor de um único argumento que representa todos os "restantes". A sintace para tal construção é baseada na chave `&rest`

```
(defun f (a b &rest c)
  (list 'recebi a b 'e 'mais (length c) 'argumentos)
)
```

Observe o funcionamento desta função:

```
(f 1 2) => (RECEBI 1 2 E MAIS 0 ARGUMENTOS)
(f 'x 'y 'a 'b 'c) => (RECEBI X Y E MAIS 3 ARGUMENTOS)
```

Com este recurso. o usuário pode imitar o que fazem algumas funções pré-definidas que recebem um número arbitrário de argumentos. A função `list`, por exemplo, recebe um número arbitrário de argumentos e retorna uma lista com todos eles. Lançando mão do `&rest`, podemos imitar esta função:

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

A função + em Lisp recebe um número arbitrário de argumentos e retorna sua soma. Poderia ser definida assim:

```
(defun my+ (&rest args)
  (sum-all args)
)

(defun sum-all (args)
  (if (null args)
    0
```

```
    (+ (car args) (sum-all (cdr args)))
  )
)
```

Alguém poderia argumentar que estamos usando + na definição de `my+`, mas observe que usamos apenas a funcionalidade dela para dois argumentos.

7.4 Argumentos por palavra-chave

Normalmente, a ordem dos argumentos deve ser a mesma na definição e na chamada de uma função. Algumas pessoas acham isto muito restritivo, e preferem especificar os valores dos argumentos pelo *nome* ao invés de pela posição dos argumentos. Por exemplo, suponha que uma função receba como argumentos o raio da base e a altura de um cone e retorne o volume deste cone.

A fórmula para o volume de um cone com raio da base r e altura h é:

$$V = \frac{1}{3}\pi r^2 h.$$

Mas suponha que, para que o usuário não tenha que lembrar se o primeiro argumento é o raio da base ou a altura, usemos argumento por palavra-chave para definir esta função. Ficaria assim:

```
(defun volume-cone (&key raio altura)
  (/ (* pi raio raio altura) 3)
)
```

Na hora da chamada, não importa a ordem, pois os arugmentos são dados através das palavras-chave:

```
(volume-cone :raio 1 :altura 2) => 2.0943951023931953
(volume-cone :altura 2 :raio 1) => 2.0943951023931953
```

7.5 Argumentos como variáveis auxiliares

Common Lisp permite que argumentos sejam usados como variáveis locais de funções. Estes argumentos são introduzidos através da chave `&aux` e devem ser os últimos da lista de argumentos. Podem receber valores iniciais também.

Exercícios

1. Invente e escreva uma função que aceite argumentos opcionais.
2. Invente e escreva uma função que aceite um número qualquer de argumentos.
3. Invente e 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`

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

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.

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 availado 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.

8.3 As macros `OR` e `AND`

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 availa 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, para 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 prefeitamente ser usados como funções booleanas. Além disto existe a função NOT, que retorna T se o argumento é NIl e retorna NIL caso contrário.

8.4 Predicados

Predicados são funções LISP que retornam um valor booleano. Alguns dos predicados mais importantes são:

- `null`: retorna `T` se o argumento é NIL; retorna NIL caso contrário.
- `atom`: retorna `T` se o argumento é átomo; retorna NIL caso contrário.
- `consp`: retorna `T` se o argumento é caixa; retorna NIL caso contrário.
- `listp`: retorna `T` se o argumento é caixa ou NIL; retorna NIL caso contrário.
- `numberp`: retorna `T` se o argumento é número; retorna NIL caso contrário.
- `integerp`: retorna `T` se o argumento é inteiro; retorna NIL caso contrário.

Há muitos outros predicados que testam o tipo de seus argumentos. Podemos citar `realp`, `complexp`, `floatp`, `stringp`, dentre outros. Uma lista completa pode ser econtrada no Manual de Common Lisp [6, Sec.6.2.2].

Há ainda o predicado `typep` para saber se um valor em Lisp é de um certo tipo e também o `subtypep` para saber se um tipo é subtipo de outro.

Entre os predicados que causam mais confusão ao programador iniciante são os três predicadfos de igualdade: `eq`, `eql`, `equal`. Vamos tentar elucidar esta questão aqui.

O predicado `equal` é o mais confiável e o mais demorado, pois ele compara as duas S-expressões recursivamente, até chegar em átomos, que são então comparados com `eql`. Por sua vez, `eq` apenas compara os endereços dos argumentos, produzindo resultados inesperados em algumas situações, como:

```
> (eq (cons 'a nil) (cons 'a nil))
NIL
```

Portanto, `eq` não deve ser usado para comparar pares-com-ponto. Mesmo com átomos, a veracidade de uma expressão como:

```
> (eq 3 3)
```

vai depender da implementação. Existem implementações que adotam a convenção de que constantes com o mesmo valor devem compartilhar o mesmo endereço, por motivos de economia de memória. Nestas implementação, o valor da expressão acima é T. Mas esta convenção não é universal entre as implementações de Common Lisp.

O predicado `eql` é como `eq`, exceto que garante que constantes de mesmo valor e mesmo tipo serão consideradas iguais. Ou seja, temos:

```
> (eql 3 3)
T
```

Porém, note como o tipo do argumento é relevante:

```
> (eql 3 3.0)
NIL
```

Desta forma, para os novatos, a seguintes diretrizes podem ser usadas para garantir sucesso em grande parte das situações. Se estiver comparando S-expressões arbitrárias, use `equal`. Se tiver certeza que seus argumentos são átomos, use `eql`. Evite `eq`.

Exercícios

1. Escreva uma função `abs` que retorna o valor absoluto de um argumento inteiro que receba.

2. Escreva uma função `ccomp` que receba dois argumentos numéricos `a` e `b` e retorne:

   - -1, se `a` for menor que `b`
   - 0, se `a` for igual a `b`
   - -1, se `a` for maior que `b`

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

4. Escreva um predicado `xor` que retorne o ou-exclusivo entre dois argumentos.

5. Imagine que `equal` não fosse pré-definido. Escreva sua própria versão para este predicado, usando como auxliares os predicados pré-definidos `eql`, `consp` ou `atom`.

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 indice 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 indice`: 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 distingüir 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))
    )
  )
```

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

```
(defun my-append (&rest listas)
  (my-append-aux listas)
  )

(defun my-append-aux (listas)
  (if (null listas)
      ()
    (my-append2 (car listas)
                (my-append-aux (cdr listas))
                )
    )
  )

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

`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 subseqüência de uma lista, a partir de uma posição dada, e opcionalmente indo até uma outra posição dada (sem incluí-la). As posições são numeradas a partir de zero.

```
(defun my-subseq (lista comeco &optional final)
  (if final
      (my-subseq-aux lista comeco (- final comeco))
    (my-subseq-aux lista comeco nil)
    )
  )

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

(defun primeiros (lista compr)
  (if (= compr 0)
      ()
    (cons (car lista) (primeiros (cdr lista) (1- compr)))
    )
  )
```

`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 os resultados das expressões a seguir:

   - `(third '(a b c))`
   - `(sixth '(a b c d e))`
   - `(nth 0 '(a b c))`
   - `(nth 2 '(a b c))`
   - `(elt '(a b c) 1)`
   - `(last '(a b c))`
   - `(cadar '((a b) c))`
   - `(cdddadr '(a b c (e f)))`
   - `(cdadddr '(a b c (e f)))`
   - `(length '(a b c))`
   - `(member 'b '(a b c))`
   - `(member 'x '(a b c))`
   - `(reverse '(a b c))`
   - `(append '(a b c) '(d e) '(f g h))`
   - `(list 'a 'b 'c '(d e) 'f 'g 'h)`
   - `(subst 'a 'b '(b o m (b o n) a))`
   - `(position 'b '(a b c))`
   - `(position 'x '(a b c))`

- `(count 'b '(b o m (b o n) a))`
- `(subseq '(a b c) 0 1)`
- `(subseq '(a b c) 1)`
- `(remove 'a '(b o m (b o n) a))`
- `(remove 'b '(b o m (b o n) a))`
- `(mapcar #'1+ '(1 2 3))`
- `(mapcar #'+ '(1 2 3) '(4 5))`

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 comjuntos: 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.

2. Escreva os resultados das expressões a seguir:

   - `(union '(a b c) '(c d e))`
   - `(intersection '(a b c) '(c d e))`
   - `(set-difference '(a b c) '(c d e))`
   - `(subsetp '(a b c) '(c d e))`
   - `(subsetp '(a b c) '(d e c b a g h))`

Capítulo 11

Pacotes (módulos)

Os objetivos do sistema de pacotes em Lisp são: oferecer modularidade e divisâo 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 têm 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 chamamdo `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 vísivel pelo pacote B, deve ser exportado por A e importando 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-definidfos 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; `keyword`, 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`.

2. Crie um novo pacote, defina nele uma função e exporte esta função. Depois vá para o pacote `common-lisp-user` e importe esta função. A seguir, chame-a sem qualificação, ou seja, apenas como `funcao`.

3. Repita os exercícios anteriores criando dois pacotes diferentes mas com o mesmo nome de função. É possível realizar o que pedem os dois exercícios sem provocar erro?

Capítulo 12

Arrays e Loops

Arrays e loops são construções que não existiam na concepção original de Lisp. Foram adicionadas mais tarde, possivelmente para conveniência de programadores acostumados com este tipo de construção em outras linguagens. Mas arrays e loops, em algumas situações, realmente tornam o código mais enxuto. Foram inclusive incluídas no padrão Common Lisp. Contudo, a forma de pensar quando se usam arrays e loops é diferente da forma tradicional de pensar em Lisp, e mais se aproxima da forma de pensar quando se programa em linguagens imperativas.

12.1 Arrays

Nesta seção veremos algumas das mais importantes funcionalidades relativas a arrays em Lisp.

12.1.1 Criando arrays

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

Ao serem criadas, as arrays têm todos os seus elementos indefinidos, a não ser que seja especificado na criação um valor default através de algum parâmetro palavra-chave, como `:initial-element`. Há diversos parâmetros opcionais na criação de uma array, que não serão abordados aqui. O leitor interessado pode procurar a documentação de Common Lisp.

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 contruções 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. Loops são particulamente interessantes para manipular arrays.

12.2.1 Dolist

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`.

Se *resultform* é omitida, o resultado é NIL.

Como exemplo, oferecemos a seguinte função para reverter uma lista usando `dolist`:

```
(defun rev-loop (lista)
  (let ((result '()))
    (dolist (x lista result)
      (setf result (cons x result))
    )
  )
)
```

12.2.2 Dotimes

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 de *countform* 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`.

Se *resultform* é omitida, o resultado é NIL.

Como exemplo, oferecemos a seguinte função que remove os `k` primeiros elementos de uma lista: usando `dolist`:

```
(defun without-first-k-loop (k lista)
  (let ((result lista))
    (dotimes (i k result)
      (setf result (cdr result))
    )
  )
)
```

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`.
3. Escreva uma função que receba uma array unidimensional de números e a retorne ordenada usando as funcionalidades vistas neste capítulo e o algoritmo quicksort.

Apêndice A

Dicas sobre os interpretadores Lisp

A.1 Editor Emacs

Sei que o editor Emacs é meio velha-guarda, mas fui ensinado nele e tenho carinho por ele até hoje. Para quem gosta de Emacs, aqui vão algumas dicas para rodar Lisp dentro do editor. Uma vantagem de usar o editor Emacs é que ele balanceia os parênteses para você, e ajuda também da indentação. Mas grande parte dos editores mais modernos também faz isto. Poucos editores têm as macros do Emacs, porém, e menos ainda deixam você criar suas próprias macros de edição.

Antes de mais nada, é preciso preparar o editor. Para tanto, 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 a string `"lisp"`. Se for usar o CLISP, este valor é a string `"clisp"`.

Para chamar o interpretador, use `M-x run-lisp`. Este comando cria um buffer Emacs onde você pode interagir com o interpretador. É como se fosse uma shell interna ao Emacs. Para sair, use o comando `(quit)`.

Se você estiver editando um arquivo `xxx.lsp`, o Emacs usará `lisp-mode` para você editá-lo, o que fornece indentação e balanceamento de parênteses, por exemplo. É possível realizar uma avaliação imediata de expressões enquanto edita: ao teclar `C-x C-e`, o Emacs avalia a expressão anterior ao cursor. Outras facilidades

existem: consulte `M-x describe-bindings`.

A.2 CLISP

CLISP é a implementação do Common Lisp feita por Bruno Haible, e atualmente mantida por Sam Steingold. É distribuída com licença GNU-GPL, e pode ser obtida no seguinte endereço (consultado em julho/2014):

`http://clisp.cons.org`

Para instalar, faça *download* do site e siga as instruções. Existem versões para plataformas das famílias Unix e Windows.

Eis as receitas para algumas operações básicas:

entrar no interpretador: `clisp`
sair do interpretador: `(quit)`
usar o interpretador: tecle expressões seguidas de `ENTER`
repetir a última expressão: `*`
repetir a **penúltima** expressão: `**`
repetir a **antepenúltima** expressão: `***`
se ocorrer um **erro**: tecle `:rN` e volte ao nível inicial
(onde `:rN` é a opção listada como `abort`)
usar com **arquivo**: `clisp < arquivo.lsp`
(`arquivo.lsp` é um arquivo com expressões a avaliar)
(o arquivo deve ser do tipo texto simples)

A.3 CMUCL

CMUCL é a implementação do Common Lisp feita pela Carnegie Mellon University, EUA, *para plataformas da família Unix*. Pode ser obtida no seguinte endereço (consultado em julho/2014):

```
http://www.cons.org/cmucl
```

Para instalar, faça *download* do site e siga as instruções. Eis as receitas para algumas operações básicas:

entrar no interpretador: `lisp`
sair do interpretador: `(quit)`
usar o interpretador: tecle expressões seguidas de `ENTER`
repetir a última expressão: `*`
repetir a **penúltima** expressão: `**`
repetir a **antepenúltima** expressão: `***`
se ocorrer um **erro**: tecle `0` e volte ao nível inicial
usar com **arquivo**: `lisp < arquivo.lsp`
(`arquivo.lsp` é um arquivo com expressões a avaliar)
(o arquivo deve ser do tipo texto simples)

A.4 Compilação

Além de servir como interpretador, o Common Lisp permite que funções ou arquivos Lisp sejam compilados, 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").
```

A.5 Depuração

Para depurar seus programas, é importante saber que existe um mecanismo de trace, acionado como segue:

```
(trace deriv)
```

A partir deste comando, 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 acompanhar várias funções ao mesmo tempo:

```
(trace fft gcd string-upcase)
```

Apêndice B

Cálculo Lambda

B.1 Introdução

Motivação:

- funções anônimas
- retornando funções, basta 1 argumento (currying)
- nome do argumento pode mudar sem afetar função

Definição formal:

- termo: var
- termo: λ var $\cdot$ termo
- termo: termo termo (nota: associa à esquerda)

Isto implementa funções anônimas e currying. Para a 3a. motivação, temos a definição de α-equivalência, verificável em tempo linear. Conceito de variável livre e variável ligada.

B.2 Reduções

β-redução é aplicar funções. Vai aplicando até não poder mais: termo *irredutível* ou *normal*. Há vários caminhos de redução. Os que param chegam ao mesmo resultado. Mas pode haver alguns que não param.

Duas estratégias interessantes:

- reduzir sempre o mais à esquerda: se algum caminho parar, este para, mas pode demorar mais.
- reduzir sempre o mais interno: mais rápido, mas pode não parar mesmo existindo outro caminho que para.

η-redução: simplificar $(\lambda \text{var} \cdot M\text{var}) \longrightarrow M$. Diminui tamanho.

B.3 Exemplos

Função Identidade:

$$(\lambda x \cdot x)y \longrightarrow y$$

$$(\lambda y \cdot y)y \longrightarrow y$$

$$(\lambda z \cdot z)y \longrightarrow y$$

$$(\lambda z \cdot z)x \longrightarrow x$$

Reduções múltiplas:

$$(\lambda x \cdot (\lambda y \cdot y)y)z \longrightarrow (\lambda y \cdot y)y \longrightarrow y$$

$$(\lambda x \cdot (\lambda y \cdot y)x)z \longrightarrow (\lambda y \cdot y)z \longrightarrow z$$

$$(\lambda x \cdot (\lambda x \cdot x)y)z \longrightarrow (\lambda x \cdot x)y \longrightarrow y$$

Na β-redução, substitui só as ocorrências livres.

$$(\lambda x \cdot (\lambda x \cdot x)x)z \longrightarrow (\lambda x \cdot x)z \longrightarrow z$$

Loop infinito:

$$(\lambda x \cdot xx)(\lambda x \cdot xx) \longrightarrow (\lambda x \cdot xx)(\lambda x \cdot xx)$$

Associatividade à esquerda:

$$(\lambda x \cdot xy)(\lambda x \cdot yx)z \longrightarrow (\lambda x \cdot yx)yz \longrightarrow yyz$$

$$(\lambda x \cdot xy)((\lambda x \cdot yx)z) \longrightarrow (\lambda x \cdot xy)(yz) \longrightarrow yzy$$

Vários caminhos de redução: partindo do mesmo termo, chega no mesmo resultado.

Por dentro e por fora:

$$(\lambda x \cdot (\lambda y \cdot y)y)z \longrightarrow (\lambda x \cdot y)z \longrightarrow y$$
$$(\lambda x \cdot (\lambda y \cdot y)y)z \longrightarrow (\lambda y \cdot y)y \longrightarrow y$$

Por dentro e por fora:

$$(\lambda x \cdot (\lambda y \cdot y)x)z \longrightarrow (\lambda x \cdot x)z \longrightarrow z$$
$$(\lambda x \cdot (\lambda y \cdot y)x)z \longrightarrow (\lambda y \cdot y)z \longrightarrow z$$

Às vezes tem que mudar nome de argumento para não "prender" variável indevidamente:

$$(\lambda y \cdot (\lambda x \cdot yx))x \longrightarrow \lambda z \cdot xz$$

- Aritmética
- Booleanos
- If-then-else
- Pair
- Recursão
- Minimização
- Todas as funções computáveis (tese de Church-Turing)

Exercícios

1. Escreva um predicato `termop` em LISP que reconheça S-expressões que representem termos do cálculo lambda, de acordo com a seguinte definição:

   - todo símbolo (exceto `l`) é um termo
   - se `s` é um símbolo (exceto `l`) e `u` é um termo, então `(l s u)` é um termo
   - se `u` e `v` são termos, então `(u v)` é um termo

2. Escreva uma função `livres` que receba um termo lambda como definido acxima e retorne a lista dos símbolos livres no termo.

3. Escreva uma função `subst-livre` que receba um símbolo `s` e dois termos lambda `u` e `v` e retone um novo termo lambda obtido a partir de `u` substituindo todas as ocorrências livres de `s` em `u` por `v`. Pode-se dizer que este novo termo é uma β-redução de `((l s u) v)`?

4. Modifique sua função `subst-livre` acima para que ela evite que variáveis fiquem ligadas indevidamente após a substituição.

5. Escreva uma função `leftmost` que recebe um termo lambda e retorna o resultado de aplicar a β-redução mais à esquerda neste termo lambda, ou `nil` se não for possível aplicar nenhuma β-redução no termo dado.

6. Escreva uma função `normal` que receba um termo lambda e retorne a sua forma irredutível (também chamada de normal), obtida efetuando sucessivas β-reduções, sempre escolhendo a redução mais à esquerda.

7. Escreva um predicado `equiv` em LISP que decida se dois termos lambda são α-equivalentes.

Bibliografia

[1] Paul Graham. *On Lisp*. Prentice Hall, 1993.

[2] Paul Graham. *ANSI Common Lisp*. Prentice Hall, 1995.

[3] Paul Graham. *Hackers & Painters*. O'Reilly, 2004.

[4] John L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[5] Edward H. Shortliffe. *Computer-Based Medical Consultation: MYCIN*. Elsevier North Holland, 1976.

[6] Guy L. Steele. *Common Lisp the Language*. Digital Press, second edition, 1990. ISBN 1-55558-041-6.

[7] 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.