Uma Curtíssima Introdução ao Emacs Lisp v.16

Jerônimo C. Pellegrini

8 de julho de 2010

Este tutorial foi escrito para quem já conhece o Emacs (e tem uma mínima expreiência com alguma linguagem de programação) mas não sabe configurá-lo ou programá-lo. O tutorial presume familiaridade com o Emacs e com conceitos básicos de programação.

O Emacs foi projetado para funcionar como editor de textos, mas seu projeto assemelha-se a uma máquina Lisp virtual (e de fato, o Emacs consiste de um pequeno núcleo escrito em cerca de dez mil linhas de C, que implementa uma máquina virtual Lisp – a maior parte do editor é escrita em mais de duzentas mil linhas de Emacs Lisp).

Sendo em sua essência uma máquina virtual, o Emacs pode ser reprogramado para funcionar não apenas como editor, mas também como qualquer outra aplicação que se queira. Existem implementados em Emacs Lisp:

- Leitores de mail e de newsgroups (Wanderlust, Gnus);
- Jogos;
- Acessórios diversos, como calendários, calculadoras e aplicativos para organização pessoal (org-mode, além de acessórios que já vem com o Emacs);
- Ambientes de desenvolvimento muito sofisticados (SLIME);
- Ambientes para edição de texto (AucTeX);
- Navegadores para Internet (w3);
- Clientes de FTP;
- Um editor de vídeo (GNEVE);
- Interface para sintetizador de voz (EmacSpeak), essencial para usuários com deficiência visual.

Este tutorial apresenta conceitos básicos de Emacs Lisp para iniciantes, mas presume alguma familiaridade com o Editor e compreensão de conceitos básicos de programação.

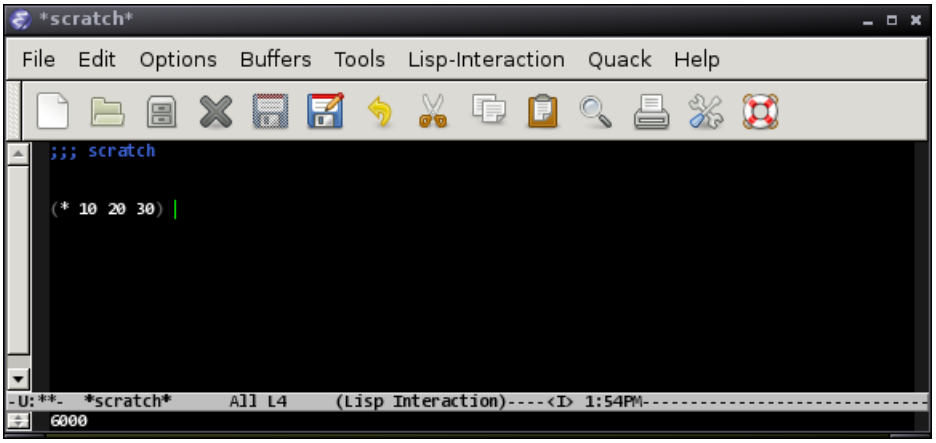


Figura 1: O Emacs apresenta no minibuffer o resultado da avaliação de uma expressão.

1 Programando com Emacs Lisp

Para interagir com o Emacs usando Emacs Lisp, vá ao *buffer* "`scratch`". Digite uma expressão Lisp, como `(* 10 20 30)`, posicione o cursor no final da expressão e digite `C-x C-e`. O Emacs responderá no minibuffer (veja a Figura 1).

Você também pode gravar arquivos com a extensão `.el` e abrí-los no Emacs (ele entrará em lisp-interaction-mode).

Quando uma expressão simbólica como `(* 10 20 30)` é "enviada" ao Emacs, ela é passada para a máquina Lisp interna, que avaliará a expressão.

Exemplos mais interessantes de expressões para avaliar são:

```
(message "Olá, mundo!")

(concat "um" " dois " "três")

(make-string 10 ?z)

(split-string "Andorinha africana ou européia?")
```

O resultado da avaliação é mostrado no minibuffer. Podemos, no entanto, pedir ao Emacs Lisp que insira o resultado diretamente no buffer atual, onde quer que o cursor esteja:

```
(insert  "A word is dead when it's said, they say")

(insert ?a)

(insert (split-string "a b c"))
```

A última expressão resultará em erro, porque só é possível inserir no buffer caracteres e strings.

1.1 *Prompt* para interação com o Lisp

Não é necessário usar apenas um buffer do Emacs e o minibuffer para avaliar expressões. Pode-se abrir uma janela com um prompt para interagir com o Emacs Lisp, enviando expressões. Esta janela pode ser aberta através do comando `M-x ielm`.

1.2 Notação prefixa

As linguagens Lisp usam notação prefixa. Ao invés de "$3 + 4/5 - 2$", em Lisp dizemos `(- (+ 3 (/ 4 5)) 2)`. Isso torna uniforme o tratamento de todo tipo de função – desde os operadores aritméticos até funções definidas pelo usuário.

Embora a notação prefixa pareça estranha em um primeiro contato, ela permite tratar com regularidade todo tipo de função. A notação infixa é normalmente usada apenas para as quatro operações aritméticas, como em $a + b * c - 4$; no entanto, ao usar funções como seno, tangente, logaritmo e outras, usa-se uma notação diferente, que é muito próxima da prefixa usada em Lisp:

```
seno(x)    (seno x)
tan(x)     (tan x)
log(n)     (log n)
f(x, y, z) (f x y z)
```

As duas diferenças são a ausência de vírgulas em Lisp e o nome da função, que em Lisp fica *dentro* dos parênteses.

Além do tratamento uniforme, a notação prefixa elimina a necessidade de definição de precedência de operadores: usando notação infixa, $a + b/c \neq (a + b)/c$. Usando notação prefixa, a expressão `(/ (+ a b) c)` só pode ser interpretada de uma única forma: "divida a soma de a e b por c"[1].

1.3 Símbolos e Variáveis

Um símbolo é semelhante a um nome de variável em outras linguagens de programação, exceto que em Lisp os símbolos são objetos de primeira classe (podem ser usados como valor de variável, podem ser passados como parâmetro e retornados por funções).

1 Ler as funções e procedimentos usando verbos como "divida:" ou "some:" ajuda a ambientar-se com a notação prefixa.

É importante notar que um símbolo *não* é uma variável. Ele é um *nome* que pode ser vinculado a uma variável (e este nome pode ser manipulado como se fosse um objeto como qualquer outro).

Exemplos de símbolos são `a`, `b`, `uma-palavra`, `+`, `-`, `>`.

Quando o Emacs Lisp avalia um símbolo, imediatamente tentará retornar o *valor* da variável com aquele nome. Se não houver valor, um erro ocorrerá.

```
a         => C-x C-e resulta em erro!
```

Para que o Emacs Lisp retorne o símbolo e não seu valor, é necessário "citá-lo":

```
(quote um-simbolo)
```

A expressão acima retorna o símbolo `um-simbolo`.

Para associar um valor a um símbolo (ou "atribuir valores a variáveis") pode-se usar a forma especial `set`:

```
(set (quote um-simbolo) 1000)

(quote um-simbolo)       => um-simbolo
um-simbolo               => 1000 (o valor)
```

Uma forma curta para `(quote x)` é `'x`:

```
(set 'um-simbolo 1000)
```

Precisamos usar o quote antes de `um-símbolo` porque de outra forma, se usássemos "`(set um-simbolo 1000)`", o Emacs tentaria avaliar `um-simbolo` antes de proceder com o `set`.

```
'um-simbolo              => um-simbolo
um-simbolo               => 1000 (o valor)
```

A forma `set` quase sempre é usada em conjunto com `quote`, por isso há uma forma combinada das duas (`setq` significa "set-quote"):

```
(setq copyright-message "(C) Fulano de Tal, 2010")

(insert copyright-message)
```

Não é possível em Emacs Lisp[2] criar ua variável com nome `T`, que é usado como constante para o valor "verdadeiro". O mesmo vale para `nil`, que é uma constante representando tanto "falso" como "uma lista vazia".

2 Nem em Common Lisp, a não ser com técnicas avançadas (*reader macros*).

2 Funções

A forma especial `lambda` é usada para construir pequenos trechos de código que aceitam parâmetros e devolvem valores – funções! As funções definidas com `lambda` não precisam ter nome.

A seguir há exemplos de funções sem parâmetros:

```
(lambda () "Resultado")
(lambda () 10)

((lambda () "Resultado"))
((lambda () 10))
```

Note que as duas primeiras linhas são a expressão de duas funções; as duas últimas linhas são a expressão da *aplicação* destas funções (o primeiro – e único – item de cada uma destas últimas formas é uma função).

O exemplo a seguir mostra uma função com dois parâmetros:

```
(lambda (a b)
  (if (> a b)
      (- a b)
      (- b a)))
```

Funções são objetos como quaisquer outros em Lisp. Podemos dar-lhes nomes com `fset`, de maneira semelhante ao que fizemos com `set` para valores de variáveis:

```
(fset 'minha-funcao
  (lambda (a b)
    (if (> a b)
        (- a b)
        (- b a))))

(minha-funcao 10 3)       => 7
```

As expressões `lambda` são úteis quando não é necessário dar nome a uma função. Para definir funções já com nome, usa-se a forma especial `defun`:

```
(defun minha-funcao (a b)
  (if (> a b)
      (- a b)
      (- b a)))
```

2.1 Emacs Lisp é um $Lisp_2$

Em diversos dialetos de Lisp (inclusive Emacs Lisp), pode haver mais de um valor associado a um símbolo. No Emacs Lisp, um símbolo pode representar um objeto comum (número, string, outro símbolo, uma lista) e ao mesmo tempo uma função – analogamente a uma caixa com dois compartimentos, um para valores e um para funções.

As funções `set` e `setq` são usadas para mudar o valor associado a um símbolo. A função `fset` muda a função associada a ele:

```
(setq coisa-estranha 5)
(fset 'coisa-estranha '(lambda (a) (* a a)))
```

Não existe `fsetq` para que se possa evitar o `quote` antes de `coisa-estranha`.

Agora o símbolo `coisa-estranha` contém dois objetos: o valor 5 e a função `(lambda (a) (* a a))`. O contexto determinará qual deles será usado:

```
coisa-estranha          => 5
(coisa-estranha 3)      => 9
```

Podemos inclusive aplicar a função `coisa-estranha` (que calcula o quadrado de um número) ao valor `coisa-estranha` (que é igual a 5):

```
(coisa-estranha coisa-estranha) => 25
```

Como um símbolo pode identificar mais de um objeto, diz-se que o Emacs Lisp (assim como Common Lisp) é um "$Lisp_2$". Já em Scheme um símbolo tem um único valor que pode ou não ser uma função – e por isso diz-se que Scheme é um "$Lisp_1$".

3 A avaliação de expressões

A maneira como o Emacs Lisp[3] avalia expressões é:

- Se a expressão for um átomo (não for uma lista), seu valor será imediatamente determinado: para objetos como números, caracteres e strings o valor é o próprio objeto. Para símbolos, o valor retornado será o valor da variável cujo nome é aquele símbolo (se não houver valor associado ao símbolo o Emacs levantará um erro (veja a Figura 2);
- Se a expressão for uma lista, a avaliação dependerá do primeiro elemento da lista: se ele for uma forma especial[4], a avaliação se dará de forma específica para cada forma. Caso contrário, o interpretador avaliará o primeiro elemento, que deve resultar em um procedimento. Em seguida avaliará os outros elementos da lista, e usará os valores retornados como argumentos para o procedimento.

3 Todos os Lisps fazem o mesmo.
4 Ou uma "macro"; não trataremos disso neste texto.

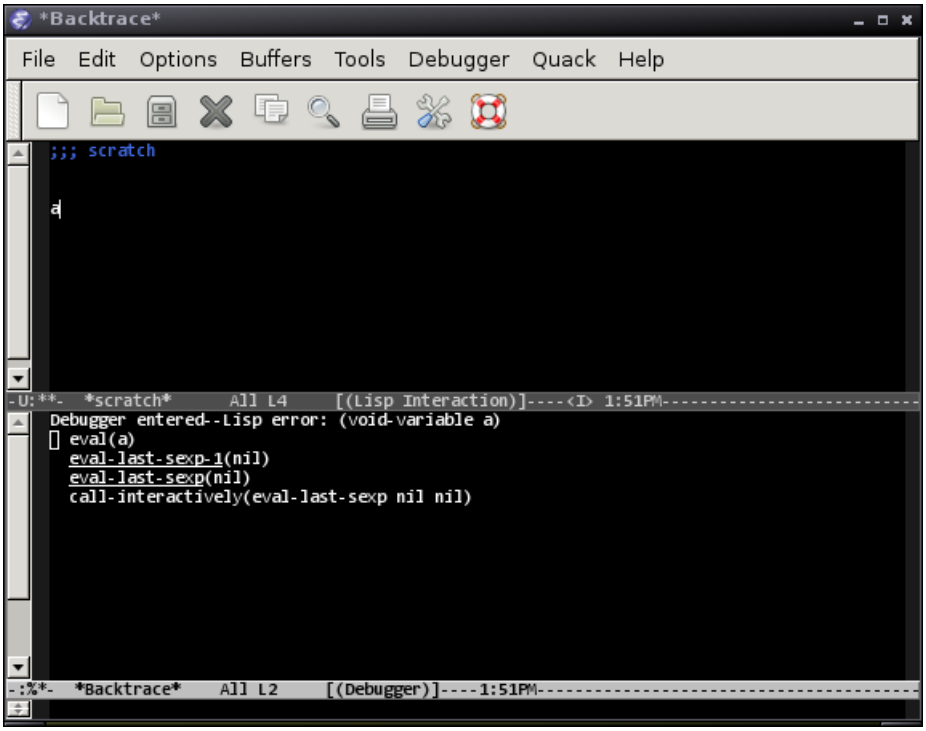


Figura 2: O Emacs apresenta um erro após uma tentativa de avaliar um símbolo sem valor.

4 Listas

Uma lista é formada por vários elementos to tipo *par*. Um par em Lisp é uma estrutura com duas partes. As duas partes de um par são chamadas (por motivos históricos[5]) `car` e `cdr`.

Para criar um par usa-se o procedimento `cons` (o "construtor" de listas).

```
(cons 1 2)
(1 . 2)
```

Pedimos ao REPL para executar o procedimento `cons` com argumentos `1` e `2`, e ele nos enviou o resultado: o par `(1 . 2)`.

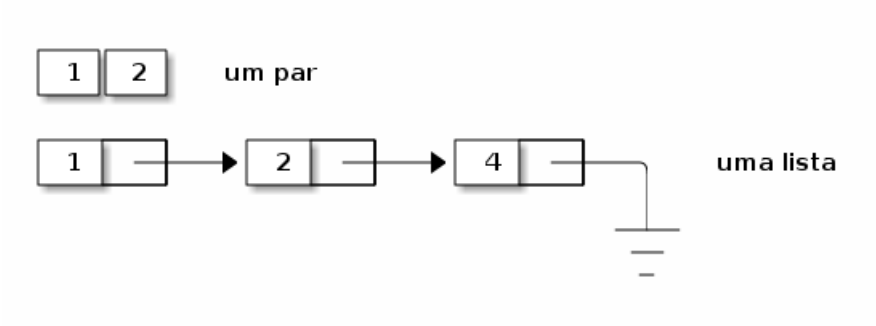
Podemos obter o conteúdo de ambas as partes do par usando os procedimentos `car` e `cdr`:

```
(cons "este é o car" "este é o cdr")
("este é o car" . "este é o cdr")
(car (cons "este é o car" "este é o cdr"))
"este é o car"
(cdr (cons "este é o car" "este é o cdr"))
```

5 Originalmente eram abreviações para *contents of address register* e *contents of data register*.

```
"este é o cdr"
```

Uma lista em Lisp é uma sequência de elementos armazenados na memória usando pares: cada par tem um elemento no `car`, e seu `cdr` tem uma referência ao próximo par:



O sinal de "aterramento" no final da lista normalmente é usado para representar `nil`.

Em uma lista tradicional, cada par contem um elemento no `car` e uma referência ao resto da lista no `cdr`. O `cdr` do último elemento da lista é `nil`.

```
(cons 1 '())
(1)
(cons 1 (cons 2 (cons 3 '())))
(1 2 3)
```

Se o último `cdr` não for `nil`, a lista será mostrada de maneira diferente:

```
(cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
```

O `. 4` no final da lista foi mostrado porque o último `cdr` não era `nil`, e sim 4 – e o Emacs Lisp mostra pares como `(a . b)`.

4.1 Funções que operam em listas

A função `mapcar` aplica uma função (de um argumento) a todos os elementos de uma lista, retornando outra lista:

```
(mapcar '- '(10 20 -30))    => '(-10 -20 30)
```

A função `reduce` aplica uma outra função (com dois argumentos) sucessivamente a todos os elementos de uma lista e acumulando os valores, iniciando com os dois primeiros elementos:

```
(reduce 'fun '(a b c d))
```

é o mesmo que

```
(fun (fun (fun a b) c) d)
```

E quando a função é a soma, `reduce` realiza um somatório de todos os elementos da lista, porque `(reduce '+ '(a b c d))` calculará `(+ (+ (+ a b) c) d)`, que é igual a `(+ a b c d)`.

Um exemplo de função usando `reduce` é mostrado abaixo.

```
(defun media (lista)
  "Calcula a media dos elementos de uma lista"
  (/ (reduce '+ lista) (length lista)))
```

No entanto, ele parece não funcionar:

```
(media '(1 2 3 4))   => 2  Uh?
```

Há algo errado com esta conta. Podemos rapidamente fazer alguns testes enviando expressões ao interpretador:

```
(+ 1 2 3 4)  => 10
(/ 10 4)     => 2   Ahá!
```

Quando os argumentos são inteiros, o Emacs Lisp usa aritmética inteira, e silenciosamente descarta o resto da divisão.

```
(+ 1 0.0)  => 1.0
```

Podemos então iniciar a soma com 0.0, passando o argumento `:initial-value` para a função `reduce`:

```
(defun media (lista)
  "Calcula a media dos elementos de uma lista"
  (/ (reduce '+ lista :initial-value 0.0) (length lista)))

(media '(1 2 3 4))   => 2.5
```

5 Ambientes e Escopo

Os conceitos de ambiente e escopo são fundamentais para a compreensão da maneira como o Emacs Lisp determina os valores associados a símbolos.

Um *ambiente* é um conjunto de associações de símbolos (ou "nomes") para valores (e funções) – este conjunto muda à medida que o programa é executado e acontecem chamadas e retornos de funções.

O *escopo* de uma variável define como seu valor é determinado.

Veja este exemplo:

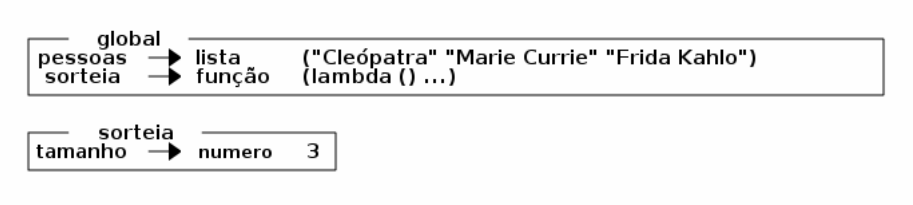
```
(setq pessoas '("Cleopatra" "Marie Currie" "Frida Kahlo"))

(setf 'sorteia
      '(lambda ()
        (nth (random (length pessoas))
             pessoas)))
```

A função `random` escolhe um número aleatóreo: `(random n)` será algum número x tal que $0 \leqslant x < n$.

A função `nth` retorna o n-ésimo elemento de uma lista: `(nth 1 '(a b c))` é o símbolo `b` (o primeiro elemento tem índice zero).

No exemplo anterior, o ambiente global tem duas vinculações definidas pelo usuário:



além, é claro, daquelas já definidas "de fábrica" pelo Emacs Lisp (por exemplo, os símbolos `random`, `nth`, `insert` e muitos outros tem vinculações definidas), mas não nos preocuparemos com estas.

Este conjunto de vinculações é chamado de *ambiente*. Assim, logo após enviarmos as duas formas (o `setq` e o `setf`) ao Emacs, o ambiente será aquele definido na tabela acima. Se depois ainda enviarmos a seguinte forma ao Emacs:

```
(sorteia)
```

O Emacs verá que o símbolo `sorteia` está no início de uma lista, e que portanto deve obter seu valor de função (aquele que foi definido com `fset`) – ele procurará este valor *no abiente corrente*, que neste caso é aquele já mostrado (e que inclui um valor de função para `sorteio`). Ao aplicar a função, precisará de um valor para o símbolo `pessoas`. Novamente, procurará este valor no ambiente corrente.

Se tentarmos usar a forma `(sorteia)` sem antes definir valores para estes dois símbolos, nos depararemos com um erro, obviamente.

A forma especial `let` permite definir valores temporários para símbolos, extendendo temporariamente o ambiente. O `let` tem este nome por ser análogo ao "let" ("seja") usado em prosa Matemática: considere a afirmação "Seja x igual a zero. A expressão $1/x$ não tem valor definido.". A palavra *"seja"* é usada para determinar um valor para x, não em todo o texto, mas apenas na afimração seguinte. Da mesma forma, podemos escrever, em Lisp:

```
(let ((x 2))
  (* x x))        => 4
```

O símbolo x passou a ter valor 2, mas apenas temporariamente:

```
(let ((x 2))
  (* x x))        => 4

(+ x 1)           => resulta em erro!
```

No exemplo acima, a expressão `(+ x 1)` está fora do *escopo* do `(let ((x 2)) ...)`.

5.1 Escopo dinâmico

No Emacs Lisp o escopo das variáveis é dinâmico. Veja a seguinte função C:

```
int x = 2;

void mostra () {
  int y = 3;
  printf ("%d %d\n", x, y);
}
```

As seguintes observações a respeito da função `mostra` são relevantes para a compreensão da diferença entre escopo léxico e dinâmico:

- Ela não pode ser compilada sem que `x` tenha sido declarada como variável global;
- Dentro de `mostra`, a variável `x` sempre se referirá a um local de memória previamente definido (o local da variável global `x`);
- Não é possível mudar a vinculação de `x` antes de chamar a função `mostra`: não podemos fazer algo como "chame `mostra` com `x=10`" sem ter que explicitamente modificar o valor da variável global.

Isso acontece porque o ambiente ao qual `mostra` tem acesso consiste de:

- Seu ambiente local, onde `y` está definido;
- O ambiente global, onde `x` está definido.

Este ambiente não pode ser modificado depois que o programa tiver sido compilado. As variáveis a que `mostra` tem acessso são aquelas alcançáveis visualmente através de inspeção do código. Dizemos que C tem variáveis com *escopo léxico* ou *estático*.

Em Emacs Lisp pode-se modificar o ambiente em tempo de execução.

A função abaixo toma uma lista e divide cada elemento pela variável *total*:

```
(defun proporcoes (valores)
  (mapcar '(lambda (v) (/ (+ 0.0 v) total))
          valores))
```

Seu uso resultará em erro, porque a variável `total` não tem valor definido. Há duas maneiras de usar esta função. A primeira é familiar a quem conhece linguagens como C: basta criar uma variável global com o nome de total:

```
(setq total 30)
(proporcoes '(2 3 4)) => '(0.0666666 0.1 0.1333333)
```

No entanto, tendo ou não definido um valor global para `total`, pode-se criar temporariamente um valor para `total` usando `let`:

```
total => 30

(let ((total 20))
  (proporcoes  '(2 3 4)))   => '(0.1 0.15 0.2)

total => 30
```

O valor de `total` permaneceu inalterado quando o avaliamos fora do `let`, mas valia 20 quando da chamada a `proporcoes` dentro do `let`.

Com *escopo dinâmico* o valor de uma variável é determinado através de uma busca na função em que ela foi referenciada e, caso necessário, recursivamente nas funções que foram chamadas até chegar na função atualmente sendo executada.

Com *escopo léxico*, o valor de uma variável é determinado através de uma busca na função onde ela foi definida e, caso necessário, recursivamente nas funções onde esta função foi definida.

O uso de escopo dinâmico era comum em variantes antigas de Lisp, mas as formas mais modernas (como Scheme e Common Lisp) implementam escopo estático, que oferece menos oportunidade ao programador para cometer erros.

O Emacs Lisp permite usar escopo léxico também, usando o `lexical-let`:

```
(setq total 10)
```

```
(lexical-let ((total 10000))
  (proporcoes  '(2 3 4)))  => '(0.2 0.3 0.4)
```

O `lexical-let` não alterou o ambiente usado por `proporcoes`.

```
(lexical-let ((total 100))
  (defun lex-prop (valores)
    (mapcar (lambda (v) (/ (+ 0.0 v) total))
            valores)))

(setq total 10)

(lex-prop '(2 3 4))  => '(0.02 0.03 0.04)
```

O `lexical-let` definiu `total` de forma que `lex-prop` não consiga mais usar a variável global `total` – agora só a variável local `total` é usada, e sua vinculação não pode ser mudada por `let` ou `lexical-let`:

```
(let ((total 2))
  (lex-prop '(2 3 4)))  => '(0.02 0.03 0.04)

(lexical-let ((total 2))
  (lex-prop '(2 3 4)))  => '(0.02 0.03 0.04)
```

6 Guardando informações: *hashtables* e listas de associação

É muito comum que implementações de Lisp tragam funções que implementam tabeas de *hashing*. Emacs Lisp não é exceção, e esta seção mostra como usar *hashtables* em Emacs Lisp.

Muitas funções Emacs Lisp usam uma forma diferente de mapeamento entre chaves e valor, chamada de "listas de associação", que também são descritas logo após as tabelas de *hashing*.

6.1 *Hashtables*

A função `make-hash-table` cria *hashtables*:

```
(make-hash-table)    => #<hash-table 'eql nil 0/65 0x4950480>
```

Esta função aceita alguns parâmetros do tipo *keyword*, dentre os quais os mais úteis são `:test`, que determina qual função será usada para determinar igualdade de chaves na tabela de *hashing*, e `:size`, que é uma *dica* ao Emacs a respeito do número esperado de elementos. O parâmetro `:test` deve necessariamente ser um dentre `eq`, `eql` e `equal`.

```
(make-hash-table :test 'equal
                 :size 10000)

 => #<hash-table 'equal nil 0/10000 0x4673d60>


(setq h (make-hash-table))

(gethash 1 h)     => nil
(puthash 1 'x h)
(gethash 1 h)     => 'x

(puthash "chave" 'valor h)
(gethash "chave" h)         => nil
```

O segundo exemplo não funciona porque a tabela `h` usa `eql` para comparar chaves, e strings não devem ser comparadas com `eql`, e sim com `equal`:

```
(setq h2 (make-hash-table :test 'equal))

(puthash "chave" 'valor h2)
(gethash "chave" h2)         => 'valor
```

6.2 Listas de associação

Quando o número de informações a guardar é pequeno pode-se usar *listas de associação* ("assoc lists").

Uma lista de associações é uma lista de pares, onde o primeiro faz papel de chave e o segundo de valor armazenado:

```
(setq cineastas '((polanski "Roman Polanski")
                  (stan  "Stanley Kubrick")
                  (al       "Alfred Hitchcock")
                  (akira    "Akira Kurosawa")
                  (dave     "David Lynch"")))
```

A função `assoc` retorna o primeiro par da lista cujo `car` é `equal` à uma chave:

```
(assoc 'al cineastas)  => '(al "Alfred Hitchcock")
```

Note que o "par" retornado neste caso é uma lista, porque os pares armazenados eram também listas. Se os pares armazenados não forem lisas (se tiverem átomos tanto no `car` como no `cdr`), o valor retornado não será uma lista terminando em `nil`:

```
(setq escritores '((ernie . "Ernest Hemingway")
                   (em    . "Emily Dickinson")))

(assoc 'em escritores) => (em . "Emily Dickinson")
```

7 Expressões Regulares

O Emacs Lisp tem diversas funções relacionadas a expressões regulares. A sintaxe de expressões regulares no Emacs é:

- `.` casa com qualquer caracter exceto *newline*
- `*` permite que o padrão imediatamente anterior seja repetido zero ou mais vezes
- `+` permite que o padrão imediatamente anterior seja repetido uma ou mais vezes
- `?` o padrão anterior pode ser casado uma ou zero vezes
- `[` inicia uma lista de caracteres; qualquer um deles casará
- `^` string vazia no início de linha em um buffer
- `$` string vazia no final de um buffer

Dentro de listas de caracteres delimitadas com `[...]` pode-se usar especificadores de classes de caracter:

```
[:ascii:] caracter ASCII
[:alnum:] caracter alfanumérico
[:alpha:] caracter alfabético
[:blank:] espaços e tabs
[:digit:] dígitos numéricos
[:lower:] letras em minúsculas ([:upper:] também existe)
```

A contrabarra antes de um caracter em uma expressão regular pode ser usada da seguinte forma:

- `\|` alternativa A expressão `a|b` casa com o caracter `a` ou com o caracter `b`.
- `\{n\}` repete o padrão anterior `n` vezes: `a\{3\}` é o mesmo que `aaa`.
- `\( ... \)` realiza agrupamento de padrões. `abc|def` casa com `abc` ou com `def`.
- `\n` refere-se ao `n`-ésimo padrão. Em `\(abc\|def\)ghi\(jkl|mno\), \2` refere-se ao padrão `jkl|mno`

Quando uma expressão regular é descrita dentro de uma string, é necessário escapar também as contrabarras. Assim, a expressão regular `\(abc\|def\)*` dentro de uma string é `"\\(abc\\|def\\)*"`

A função `string-match` retorna a primeira posição em uma string onde há o casameto de uma dada expressão regular:

```
(let ((expressao "\\(John\\|Mary\\)[[:blank:]]\\(Smith\\|Doe\\)"))
  (string-match expressao "E assim John Doe conheceu Mary Smith"))

 ==> 8
```

A função *re-search-forward* tenta casar uma dada expressão regular do ponto e para a frente no buffer. Quando há o casamento, o ponto é modificado para a posição *final* onde houve o casamento. A função também retornará o valor do ponto.

8 Mais Emacs Lisp

A forma especial `if` é usada para tomar decisões:

```
(if (> 10 5)
    (message "Maior")
    (message "Menor"))
```

A forma geral do `if` é

```
(if <teste> <forma1> <forma2>)
```

Primeiro a expressão `<teste>` é avaliada; se ela resultar em `nil`, então a `<forma2>` é avaliada. Caso contrário, a `<forma1>` é avaliada.

O `if` aceita *exatamente* duas formas. Quando é necessário incluir mais de uma forma dentro de um dos casos, é necessário agrupá-los em um `progn`:

```
(if (< i max)
    (progn
      (processa i)
      (setq i (+ i 1)))
    (finaliza i))
```

Qualquer objeto diferente de `nil` será tratado como valor "verdadeiro", mesmo que diferente de `T`.

A forma `if` só é interessante quando há exatamente duas formas a escolher. Quando é necessário escolher entre muitos casos, a forma `cond` é preferível:

```
(setq saldo 10)

(insert
  (concat "Saldo "
    (cond ((> n 0)
           "credor")
          ((< n 0)
           "devedor")
          (T
           "zero"))))
```

A sintaxe para `cond` é

```
(cond (<teste1>
       <corpo1>)
      (<teste2>
       <corpo2>)
       ...)
```

Cada `corpo*` pode conter várias formas, que serão executadas sequencialmente.

As funções para comparação de strings são `string<` ou `string-lessp`; e `string=` ou `string-equal`.

Uma forma de iterar é o `while`:

`(while teste f1 f2 ...)`: executa as formas f1, f2, ... enquanto `teste` for verdadeira. O exemplo a seguir conta de um a dez (usando a variável `n`), somando os valores de `n` em outra variável, *s*:

```
(let ((n 1)
      (s 0))
  (while (<= n 10)
    (setq s (+ s n))
    (setq n (+ 1 n)))
  s)
```

Este outro exemplo insere novas linhas no buffer, cada uma com uma mensagem:

```
(let ((n 0))
  (insert "\nUma linha nova!\n")
  (while (< n 10)
    (insert "Mais uma linha!\n")
    (setq n (+ 1 n))))
```

Note que estes exemplos são um tanto artificiais, porque há outras maneiras mais simples de executar formas um número predeterminado de vezes (veja a forma especial `dotimes`).

`(dolist (e lista resultado) f1 f2 ...)`: para cada membro da lista, vincula *e* ao valor do membro e executa as formas. Ao final retorna `resultado`. Exemplo:

```
(let ((v 0))
  (dolist (n '(1 2 3 4 5 6 7 8 9 10) v)
    (setq v (+ v n))))

(let ((palavras "Uma frase qualquer"))
  (dolist (p palavras #t)
    (insert (concat p "\n"))))
```

`(dotimes (v max resultado) f1 f2 ...)`: executa as formas `max` vezes, vinculando *v* a $0, 1, 2, \ldots, max - 1$. Exemplo:

```
(dotimes (i 10)
  (insert (format "i = %d\n" i)))
```

9 Exceções

Para tratamento de exceções pode-se usar `throw` e `catch`. A versão a seguir da função `sorteia` levanta uma exceção se um nome de pessoa não for uma string:

```
;; A função sorteia presume a existência de uma variável pessoas,
;; que deve ser uma lista de strings.
(defun sorteia ()
  (let ((uma-pessoa
          (nth (random (length pessoas)) pessoas)))
    (if (stringp uma-pessoa)
        uma-pessoa
        (throw 'pessoa-nao-string "Ninguém")))))
```

A forma `(catch local corpo` executa `corpo`. Se o corpo levanter uma exceção com "`(throw local valor)`", o controle será transferido de volta imediatamente, e o valor de corpo passará a ser o `valor` determinado no `throw`:

```
(setq pessoas '(cleopatra marie-currie frida-kahlo))

(sorteia)                       => erro

(catch 'pessoa-nao-string
  (sorteia))                    => "Ninguém"
```

Veja também no manual do Emacs Lisp `unwind-protect`, `error` e `signal`.

10 Usando procedimentos e macros do Emacs

A seguir há uma lista de procedimentos úteis do Emacs Lisp. Não se trata de uma lista exaustiva, mas deverá permitir ao leitor começar a escrever funções minimamente úteis. Par auma referência mais completa, consulte o manual do Emacs Lisp.

`(message fmt v1 v2 ...)`: mostra a string `fmt` no minibuffer. Exemplos: `(message "Olá!")` `(message "Olá, %s! Tenho %d coisas para falar!" "meu caro" 1000)`

`(insert str)`: insere a string `str` no buffer, na posição em que o cursos estiver no momento. Exemplo: `(insert "Olá")`

`(format fmt v1 v2 ...)`: retorna a string formatada, de maneira semehante ao `format` de Scheme e o `printf` de C.

`(buffer-size)`: retorna o número de caracteres no *buffer* atual.

`(what-line)`: retorna o número da linha onde o cursor está.

`(point)`: retorna a posição do cursor, contada em número de caracteres desde o início do *buffer*.

`(point-min)` Returna o menor valor possível para o ponto no *buffer* atual (normalmente 1).

`(point-max)` Retorna o maior valor permisível para o ponto no buffer atual (é o fim do buffer, a não ser em alguns casos especiais).

`(buffer-name)`: retorna o nome do buffer como uma string.

`(buffer-file-name)`: retorna o nome do arquivo associado ao buffer atual.

`(current-buffer)`: o *buffer* ativo no momento.

`(switch-to-buffer)`: seleciona e ativa um *buffer*.

`(set-buffer b)` Muda o foco do Emacs para o buffer no qual programas rodarão. Não altera que janela é mostrada.

`(goto-char n)`: posiciona o cursor no n-ésimo caracter do *buffer*.

`(save-excursion f1 f2 f3)`: executa a lista de formas, guardando antes a posição do cursor. Ao terminar, a posição antiga será restaurada. Exemplo:

```
(progn
  (save-excursion
    (goto-char 1)
    (insert ";; Esta foi inserida no inicio do buffer\n"))
  (insert ";; Mas esta foi para onde o cursor estava antes!"))
```

`(interactive str)`: usada no início do corpo de uma função, declara que a função pode ser chamada co `M-x`. Se a função não tem argumentos, basta usar `(interactive)`, e a string não é necessária.

Como a função não será chamada por outra, se ela tiver argumentos, será necessário determinar como estes argumentos serão passados para a função. A string `str` contém as especificações de cada argumento separadas por `\n`. O primeiro caracter determina como o argumento será obtidos, e pode ser, entre outros:

- b: O nome de um buffer;
- f: O nome de um arquivo;
- p: um número;
- s: uma string, lida no minibuffer;
- d: a posição atual do cursor.

O resto da string `str`, até o próximo `\n`, é a pergunta que será feita n ominibuffer (veja os exemplos).

A função descrita abaixo pergunta ao usuário dois números e em seguida retorna a média harmônica deles, $2/(\frac{1}{a}+\frac{1}{b})$:

```
(defun media-har-2 (a b)
  (interactive "nDigite o valor de a:\nnDigite o valor de b:")
  ;; Acima definimos que a função aceitará dois argumentos via
  ;; minubuffer:  ambos numericos e serao perguntados ao usuario
  ;; (as perguntas sao "Digite o valor de a:"e
  ;; "Digite o valor de b:").

  ;; a seguir, calculamos a media e mostramos no minibuffer:
  (message
    (format "A media harmonica e %g" (/ 2
                                        (+ (/ 1.0 a)
                                           (/ 1.0 b))))))
```

A função a seguir não pergunta nada ao usuário, mas reporta a posição atual do cursor:

```
(defun onde-estou? (pos)
  (interactive "d")
  ;; A declaracao acima define que esta funcao recebe um argumento,
  ;; que é a posicao do cursor no momento emque a funcao é
  ;; chamada
  (message "Posicao = %d" pos))
```

11 Exemplos

Esta seção mostra alguns exemplos de código em Emacs Lisp.

11.1 Buscando parênteses pelo buffer

As funções a seguir acham os parênteses que delimitam a s-expressão onde o cursor está, e inserem asteriscos para mostrá-los:

```
(defun acha-par-esquerdo ()
  (interactive)
  (save-excursion
    (let ((nivel 1))
      (while (> nivel 0)
        (when (= (char-before) ?( )
          (setq nivel (- nivel 1)))
        (when (= (char-before) ?) )
          (setq nivel (+ nivel 1)))
          (backward-char 1))
      (insert ?*)
      (point))))


(defun acha-par-direito ()
  (interactive)
  (save-excursion
    (let ((nivel 1))
      (while (> nivel 0)
        (when (= (char-after) ?( )
          (setq nivel (+ nivel 1)))
        (when (= (char-after) ?) )
          (setq nivel (- nivel 1)))
                (forward-char 1))
      (insert ?*)
      (point))))
```

Estas funções usam `save-excursion` para que o Emacs se lembre de onde estava o cursor (porque elas usam `backward-char` e `forward-char`, que andam para trás e para a frente no buffer). Além disso, usam as funções `char-before` e `char-after` para verificar qual caracter está antes (ou depois) do cursor.

Para ver a função funcionando, digite uma expressão:

```
(abc def (ghi
```

```
(jkl (mno pqr (stu) vw)
  ((x (yz))))))
```

Posicione o cursor sobre o `n` e digite `M-x comenta-exp`. Tente posicionar o cursor em outras partes da expressão e verifique o resultado.

11.2 Fechos

É possível criar fechos (*closures*) usando escopo léxico em Emacs Lisp usando o `lexical-let`:

```
(defun get-counter ()
  (lexical-let ((cont 0))
    (lambda ()
      (setq cont (1+ cont))
      cont)))

(fset 'a (get-counter))
(fset 'b (get-counter))

(a)  ==> 1
(a)  ==> 2
(b)  ==> 1
(a)  ==> 3
```

11.3 Sublinhando s-expressões

O exemplo a seguir cria uma função que sublinha em vermelho a s-expressão onde o cursor estiver.

```
(defun destaca-s-expressao ()
  (interactive)
  (let ((over (make-overlay (acha-par-esquerdo)
                            (acha-par-direito))))
    (overlay-put over 'face
                 '(:underline "red"))))

(global-set-key (kbd "M-<f3>") 'destaca-s-expressao)
```

Um *overlay* é uma região do buffer que pode ter propriedades diferentes do resto dele. Este exemplo cria um overlay no espaço onde estiver a s-expressão atual, e depois usa `overlay-put` para dar a este overlay uma propriedade nova. A propriedade que foi incluída é "face" (que no Emacs corresponde à forma como o texto é mostrado), adicionando "`:underline "red"`", que sublinha o texto em vermelho.

12 .emacs

Ao inicializar, o Emacs "lê suas configurações de um arquivo" (em termos familiares para quem conhece outras aplicações) – ou, de maneira mais precisa, "lê um arquivo com formas Lisp e as avalia". O arquivo lido pelo Emacs é o "`.emacs`"

Após baixar algum pacote interessante feito em Emacs Lisp (leitor de mails, ambiente de programação diferente, ou algum programa Emacs Lisp que você tenha feito), você pode querer carregá-lo toda vez que o Emacs iniciar. Para isto, você pode alterar a variável `load-path`, que contém uma lista de srtings onde o Emacs procura arquivos:

```
(add-to-list 'load-path "~/.emacs.d")
```

E adicionar outra linha pedindo para carregar o arquivo:

```
(load "meu-programa.el")
```

Outras formas Lisp podem ser incluídas livremente no arquivo `.emacs` – elas serão processadas uma a uma. Por exemplo:

```
;; Função que comenta s-expressões:

(defun comenta-s-exp ()
  (interactive)
  (comment-region (acha-par-esquerdo)
                  (acha-par-direito)))

;; Definimos que Meta - F2 ativará a função que comenta
;; s-expressões:

(global-set-key (kbd "M-<f2>") 'comenta-exp)
```

A função acima usa `acha-par-direito` e `acha-par-esquerdo`, que definimos anteriormente – mas ela funcionará melhor se removermos as linhas que imprimem asteriscos naquelas duas funções (execute `comenta-s-exp` e você perceberá o motivo).

A função `comenta-s-exp` usa `comment-region`, que quando chamada interativamente, usará a marca e ponto correntes e não precisa fazer perguntas ao usuário. No exemplo acima `comment-region` *não* é chamada interativamente, por isso é necessário passar a marca e o ponto.

Veja também na documentação do Emacs Lisp `provide` e `require` para outras maneiras de carregar código no `.emacs`.

13 Hooks

Os *hooks* (ganchos) são pontos onde se podem inserir funções a serem executadas em certos momentos.

Por exemplo, se quisermos que nossa função `text-customize` (mostrada abaixo) seja sempre executada quando o Emacs entrar no modo texto, basta adicioná-la ao `text-mode-hook`:

```
;;; Text-customize
;;;
(defun text-customise ()
  "text mode customiser - sets auto-fill and binds M-i to ispell"
  (turn-on-auto-fill)
  (local-set-key "\M-i" 'ispell-paragraph))

(add-hook 'text-mode-hook 'text-customise)
```

14 Mais sobre Emacs Lisp

Este tutorial não cobre diversos pontos interessantes do Emacs Lisp: expressões regulares, modos, funções para acesso à rede, modificação de propriedades do texto e uso de gráficos. Há mais informações sobre Emacs Lisp no Emacs Wiki (http://www.emacswiki.org/).

É iportante notar que há a intenção de alguns desenvolvedores de trocar o Emacs Lisp por Scheme (Guile) em algum momento no futuro, embora não haja ainda consenso quanto a isso. Este é, no ententao, um projeto difícil – e o software já escrito em Emacs Lisp (como o SLIME) dificilmente seria portado rapidamente para Scheme.

Sobre este texto

Este tutorial foi preparado em LaTeX, os diagramas foram produzidos com a ferramenta ditaa (http://ditaa.sourceforge.net/). Alguns dos pacotes LaTeX usados são scrartcl, listings, exercise e graphicx.

Exercícios

Ex. 1 — Converta as seguintes expressões para a notação prefixa:

a) $a + b + c + d + e + f$

b) $a + b - \frac{1}{(c-b)}$

c) $a + 1 - b - 2 + c + 3$

d) $\frac{(a+b)(c+d)}{e+1}$

e) $\frac{(a+b+c)(d+e+f)}{g+1}$

f) $\frac{2a}{b-c}$

Ex. 2 — Converta as seguintes expressões para a notação infixa:

a)`(+ (* a b c) d e f)`

b)`(+ (- a b) c d)`

c)`(* a b (+ c d) (- e f) g)`

d)`(* 2 a (/ b 4) (+ c 10))`

e)`(/ 1 (+ x y z))`

f)`(* (+ a b c) (* d e (+ f g) (- h i)))`

Ex. 3 — Faça diagramas iguais àqueles na seção que discorre sobre listas, representando as seguintes estruturas:

a)`(cons 1 (cons 2 (cons 3 4)))`

b)`(cons (cons 'a 'b) (cons 1 nil))`

c)`(cons (cons nil nil) T)`

d)`(cons 1 (cons (cons 2 t) 3))`

Ex. 4 — Explique cuidadosamente o que significam as expressões e quais seus valores:

a)`(let ((let 'let)) let)`

b)`(let ((let (let let))) let (let (let) 'let))`

c)`(let (let) let (let (let) 'let))`

d)`(let ((let 'let)) (let ((lambda let)) ((lambda nil lambda))))`

e)
```
(let ((print 'let))
    ((lambda (x y)
      (apply (symbol-function x)
             (list y)))
      'print print))
```

Ex. 5 — As funções `acha-par-*` definidas neste tutorial tem um problema: se o cursor não estiver dentro de uma s-expressão, elas terminam em erro. Conserte-as para que neste caso retornem -1. Depois, mude a função que comenta s-expressões para não fazer nada se alguma das duas funções retornar -1.

Ex. 6 — Crie uma função para desfazer o destaque feito por `destaca-s-expressao`.

Ex. 7 — Escreva funções Emacs Lisp para:

a)Lembrar de um ponto no buffer. Quando o usuário estiver editando um buffer muito grande, ele poderá acionar a função que guarda a posição atual. Assim, poderá ir a outra posição, editar, e depois pedir para voltar ao ponto guardado.

b)Guardar bookmarks (usando as funções do exercício anterior).

c)Dentro da região selecionada, mudar para maiúsculas apenas as letras que iniciam período.

d)Incluir no top do *buffer* o texto resumido de uma licença (GNU GPL, MIT, "(C) Fulano" ou o que voce preferir);

e)Modifique a função anterior para que ela pergunte ao usuário que licença quer incluir (deve haver um conjunto pré-definido de funções);

f)Inserir no ponto onde estiver o cursor, um comentário Scheme com sua assinatura e data.

g)Percorrer o buffer e eliminar espaços horizontais desnecessários: quando houver mais de uma linha em branco consecutivas, deixe apenas uma.

h)Remover uma aplicação de função em Scheme/Lisp. Por exemplo, se há no *buffer* a expressão `(sqrt (+ (exp (abs x)) y z))` e o cursor está sobre o símbolo `exp`, sua função deve remover o `(exp` *e também* o fechamento dos parênteses à direita, resultando em `(sqrt (+ (abs x) y z))`. Se o cursor estivesse sobre o sinal de `+`, seriam removidos o `(+` e o resultado seria `(sqrt (exp (abs x) y z))` (não se preocupe se a expressão resultante contiver rgumentos não usados, como o `y` e o `z` neste caso).

Ex. 8 — O `c-repl` (http://neugierig.org/software/c-repl/) é um REPL para a linguagem C. Por estranho que o conceito possa parecer, existe e funciona. Construa um ambiente de desenvolvimento C para o Emacs usando o modo C que já existe no Emacs e o `c-repl`. O ambiente deve funcionar de maneira semelhante ao Quack.
Examine e descreva as diferenças entre seu ambiente para C e o Quack para Scheme (até que ponto eles podem ter as mesmas funcionalidades, e por que motivos?)

Ex. 9 — Descubra algum jogo que ainda não tenha sido escrito em Emacs Lisp e o desenvolva