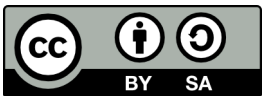
Uma Curtíssima Introdução ao Emacs Lisp

Jerônimo Pellegrini

2025.05.08.09.07
id: 926c9f1e072815ddaa1a8cac4c0b1fc62387009c

Este trabalho está disponível sob a licença *Creative Commons Attribution Share-Alike versão 4.0*.



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 email (Wanderlust, Mew, Gnus, e interfaces para os indexadores de emails notmuch e mu);
- Jogos: Tetris, Pong, Snake, incluídos no Emacs, e vários outros (Tron, Campo minado, e um jogo de ação com gráficos – slime-volleyball);
- Acessórios diversos, como calendários, calculadoras e aplicativos para organização pessoal
- org-mode, um impressionante modo com múltiplos usos: organização pessoal, produção de textos, e várias outras tarefas;
- Ambientes de desenvolvimento muito sofisticados (SLIME);
- Ambientes para edição de texto (AucTeX);
- Navegadores para Internet (w3);
- Testador de APIs (restclient-mode);

- Um editor de legendas de vídeo (subed);
- Interface para sintetizador de voz (EmacSpeak), essencial para usuários com deficiência visual.

O Emacs Lisp conta com um gerenciador de pacotes e repositórios públicos (ELPA, do projeto GNU; MELPA, aberto a contribuições da comunidade).

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

> Este é um tutorial curto! É impossível incluir, neste texto, tudo o que é necessário para construir pacotes Emacs grandes e complexos. A intenção é apenas que o leitor obtenha uma visão clara de como é programar em Emacs Lisp, e que consiga implementar, inicialmente, pequenos programas. Para além disso, poderá tentar algo mais desafiador (e mais útil) posteriormente, tendo como referência o manual do Emacs Lisp.

Os seguintes tópicos deverão ser incluídos em próximas versões:

- Comunicação via rede; acessando APIs RESTful
- Algumas formas de controle ainda não incluidas no tutorial (em particular, geradores)
- Um exemplo de modo para programação com REPL, usando `comint` como ponto de partida
- Desenhos com SVG
- Threads

1 Programando com Emacs Lisp

Há três maneiras de interagir com o Emacs Lisp

- *Em um buffer* – partes deste buffer são comandos que serão enviadas ao Emacs.
- *Em um REPL*, onde haverá um *prompt* onde é possível digitar comandos em Emacs Lisp.
- *Em modo batch*, sem que o Emacs abra um editor. Neste modo, o Emacs se comportará exatamente como um interpretador de linguagem de scripting.

1.1 Usando um *buffer* para interagir

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).

Também é possível 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:

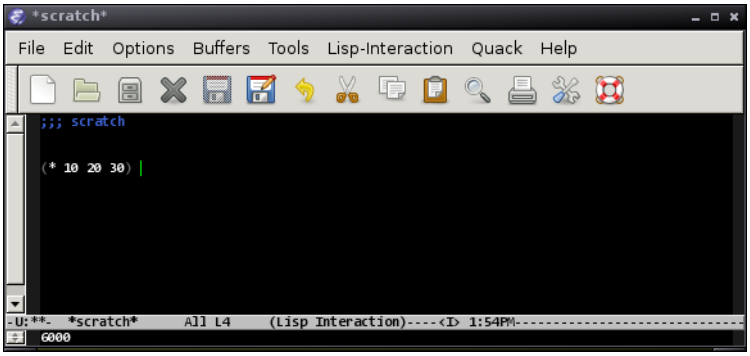


Figura 1: O Emacs apresenta no minibuffer o resultado da avaliação de uma expressão.

```
(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.

1.2 Usando um REPL (obtendo um *prompt* para interação)

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`.

1.3 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

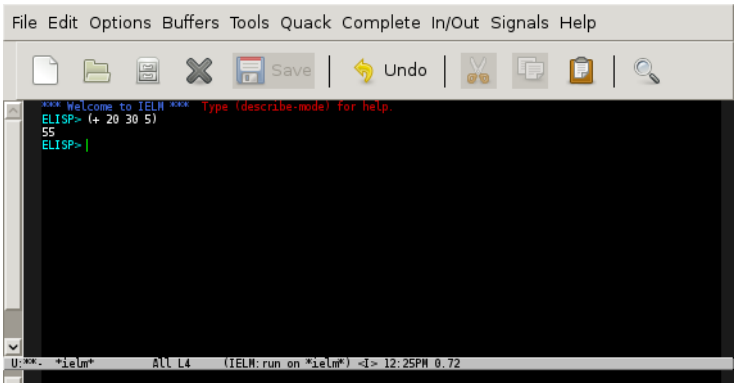


Figura 2: O *prompt* do REPL nativo do Emacs, IELM.

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. Pode haver uma confusão inicial com operações que normalmente usamos com notação infixa, mas basta lembrar que

$$a + 2$$

é o mesmo que

$$\texttt{SOMA}(a, 2), \text{ ou}$$
$$+(a, 2)$$

que em Lisp é denotado

`(+ a 2)`, cuja idéia é a mesma de
`(SOMA a 2)`

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].

1.4 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, 2014. Disponível sob licença GPLv3")

(insert copyright-message)
```

Não é possível em Emacs Lisp[2] criar uma 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".

1.5 `defvar` × `setq`

A forma `setq` modifica o valor de uma variável, criando-a se nào existir.

A forma `defvar` cria a variável e a inicializa com um valor, *desde que não tenha sido criada ainda*.

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

```
ELISP> (setq a 10)
10
ELISP> (defvar a 20)
a
ELISP> (defvar b 30)
b
ELISP> a
10
ELISP> b
30 (#o36, #x1e, ?\C-^)
```

A forma `defvar` também permite incluir uma string de documentação em variáveis. A documentação pode ser verificada com `describe-variable`.

```
ELISP> (defvar flower-color 'blue "Color used when drawing flowers")
color
ELISP> (describe-variable 'flower-color)
"flower-colors value is blue

Documentation:
Color used when drawing flowers"
```

1.6 Variáveis locais

Se uma variável é criada com `defvar` ou `setq` em um buffer, ela será visível em todos os outros buffers, e seu valor não dependerá do buffer ativo. Este comportamento pode ser adequado, se ela representar algo que é global para o Emacs, mas evidentemente há a necessidade de tratar variáveis como locais a um buffer.

`defvar` e `setq` tem variantes que operam somente no buffer corrente: `defvar-local` e `setq-local`.

Crie, no buffer do IELM, a variável `v`, usando `setq`, e depois use `setq-local`:

```
ELISP> (setq v 10)
10
ELISP> (setq-local v 20)
20
ELISP> v
20
```

No buffer do IELM, `v` é local, e tem valor 20.

Nos outros buffers do Emacs, `v` é global e tem valor 10. Para verificar, crie um buffer Emacs Lisp (ou vá ao buffer `*scratch*`) e avalie o símbolo `v`.

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 A lista de argumentos

É possível criar funções com argumentos opcionais, e também com uma lista de argumentos de tamanho variável.

Quando uma função tem argumentos opcionais, eles ficam no final da lista de argumentos da função, separados dos outros pela palavra-chave `&optional`:

```
(defun função (a b &optional c d)
...)
```

No exemplo acima, a função pode ser chamada com no mínimo dois e no máximo quatro argumentos. Quando um argumento não é passado, o valor dele em uma chamada da função é `nil`.

```
(defun f (a &optional b)
  (format "a = %S ... b = %S" a b))

(f 5)    ;; "a = 5 ...  b = nil"
(f 5 6)  ;; "a = 5 ...  b = 6"
```

`&optional` determina que, dali para a frente, os argumentos são opcionais, mas recebem nomes definidos e há um número máximo deles.

```
(defun f (a &rest x)
  (format "a = %S ... x = %S" a x))

(f 5)        ;; "a = 5 ...  x = nil"
(f 5 6)      ;; "a = 5 ...  x = (6)"
(f 5 6 7 8)  ;; "a = 5 ...  x = (6 7 8)"

(defun f (a &rest b c)
        (print (format "a = %S ... b = %S ... c = %S" a b c)))

(f 1 2 3 4) ;; "a = 1 ...  b = (2 3 4) ...  c = nil

(defun f (a b &optional c &rest x)
  (format "a = %S ... b = %S c = %S x = %S" a b c x))

(f 5 6)      ;; "a = 5 ...  b = 6 c = nil x=nil"
(f 5 6 7 8)  ;; "a = 5 ...  b = 6 c = 7 x = (8)"
```

2.2 Inlining

Se a chamada de uma função é muito lenta, e ela é feita muitas vezes, pode ser interessante torná-la uma *função inline*. O compilador do Emacs, quando fizer a *byte compilation* da função, irá tratá-la de forma diferente: ao invés de compilar uma função que precisa ser chamada, ele *expande o código da função* nos lugares onde ela é usada.

Para definir uma função inline, basta usar `defsubst` ao invés de `defun`.

Funções inline (não somente em Emacs Lisp, mas em geral) devem ser usadas somente quando realmente necessárias (otimização prematura pode tornar o programa mais complexo do que o necessário, com ganhos mínimos de eficiência[3]). O manual do Emacs Lisp enuncia alguns pontos que devem ser considerados antes de tornar uma função inline:

- Se você mudar a definição da função, nada acontece com o código que *usa* a função. Para a nova versão ser usada, é necessário compilar todas as funções que usam a função inline.
- Como o código da função é expandido em vários lugares (onde é usada), isso pode tornar o código maior.

[3] Donald Knuth dsse certa vez que "otimização prematura é a raiz de todo mal".

- Funções inline dificultam a depuração, porque os debuggers não sabem a respeito delas.

Como exemplo simples, a função `media` a seguir

```
(defun soma (a b)
  (+ a b))
```

poderia ser trocada pela função inline (ou pela "substituição")

```
(defsubst soma (a b)
  (+ a b))
```

Existe também uma outra forma de definir funções inline, `define-inline`, de que não tratamos.

2.3 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 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[4]) `car` e `cdr`.

[4] Originalmente eram abreviações para *contents of address part of register* e *contents of decrement part of register*, relacionados à arquitetura específica do IBM 704 onde a primeira implementação de Lisp foi feita.

Para criar um par usa-se o procedimento `cons` (o "construtor" de listas). A sintace de `cons` é "`(cons A B)`", onde `A` e `B` são os elementos que comporão o par.

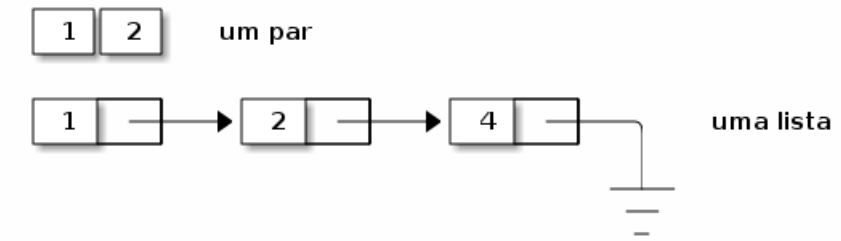
```
(cons 1 2)
```
(1 . 2)

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"))
```
"este é o cdr"

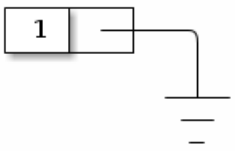
Uma lista em Lisp é uma sequência de elementos armazenados na memória usando pares: cada par tem um elemento no `car`, e seu `cdr` tem uma referência ao próximo par:



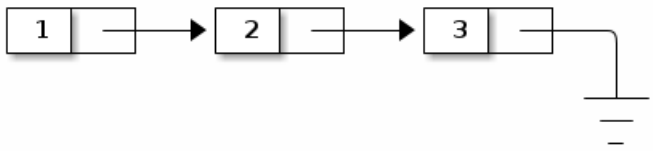
O sinal de "terra" no final da lista normalmente é usado para representar `nil`.

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`.

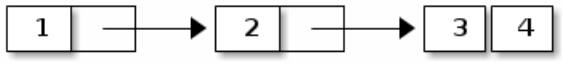
```
(cons 1 '())
```
(1)




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



Se o último `cdr` não for `nil`, não teremos uma lista propriamente falando, mas uma sequencia de pares.




```
(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)`.

3.1 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? Deveria ser 2.5
```

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
```

3.2 Arrays: strings, vetores e vetores booleanos

Listas são, em Emacs Lisp, um subtipo de sequência. A Figura 3 mostra a hierarquia de tipos de sequênci

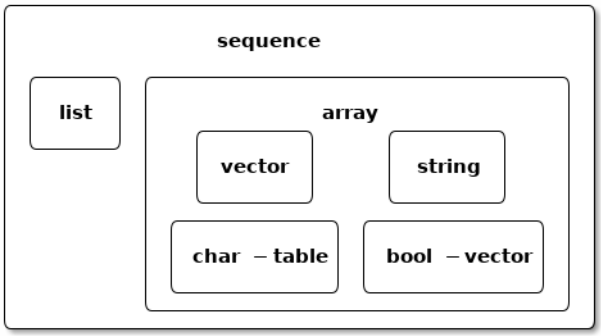


Figura 3: Tipos de sequência em Emacs Lisp.

Vetores, strings e vetores booleanos são todos *arrays*. As funções para criação, referencia e modificação deles são semelhantes. Para criação, as funções são `make-string`, `make-vector` e `make-bool-vector`.

```
(make-string 3 ?z)          ;; "zzz"
(make-vector 4 "abc")       ;; [ "abcabcabcabc"]
(make-bool-vector 3 nil)    ;; #&3"@"
```

A representação de um `bool-vector` começa com `#n`, onde `n` é o tamanho do vetor, seguida de uma string que contém o conteúdo do vetor em binário, mas mostrado como string (o @ significa que o valor binário não corresponde algo prontável em ASCII ou Unicode). O conteúdo do `bool-vector` pode ser mostrado da mesma forma que um vetor comum se usarmos `vconcat`, que concatena vetores e `bool-vectors`, retornando um vetor comum.

```
(vconcat (make-bool-vector 3 nil))  ;; [ nil nil nil ]
```

Os elementos de uma string são caracteres, denotados individualmente por `?a`, `?b`, etc. Um `vector` pode conter elementos de qualquer tipo, inclusive strings, listas ou outros vetores. Um `bool-vector` somente pode conter `t` e `nil`.

Para todos os arrays, `aref` seleciona, e `aset` modifica, um elemento.

```
(aref "abc" 1)                    ;; ?b
(aref [ 'a 'b 'c ] 1)             ;; 'b
(aref (make-bool-vector 10 t) 1)  ;; t

(let ((a "abc")) (aset a 1 ?x) a)      ;; "axc"
(let ((b ['a 'b 'c])) (aset b 1 5) b)  ;; ['a 5 'c]

(let ((c (make-bool-vector 5 t)))
  (aset c 2 nil)
  (aref c 2))                          ;; nil
```

Os predicados para determinar se um objeto é do tipo `sequence`, `array`, `string`, `vector` ou `bool-vector` são

```
sequencep
arrayp
stringp
vectorp
bool-vector-p
```

O nome do último pode parecer em desacordo com o padrão usado nos outros, por terminar em `-p` (com hífen). Na verdade, segue-se aqui um padrão comum em Lisps, onde o `p` (de "Predicado") é adicionado ao final do nome do tipo, e *é usado um hífen antes do `p` se o nome já continha algum hífen*.

```
(sequencep "abc")                  ;; t
(arrayp  "abc")                    ;; t
(stringp "abc")                    ;; t
(vectorp [ 1 2 3 ])                ;; t
(bool-vector-p [ 1 2 3 ])          ;; nil
(vectorp (make-bool-vector 5 nil)) ;; nil
```

A última linha mostra algo importante: *um* `bool-vector` *não é subtipo de* `vector`!

Assim como é possível escrever caracteres entre aspas para construir uma string, e parênteses para construir listas, pode-se usar colchetes para construir vetores. A a notação `#n`, no entanto, é problemática para criar vetores booleanos, porque dificilmente conseguimos incluir, em um arquivo texto com programas Emacs Lisp, uma string contendo uma sequência binária qualquer.

Não tratamos, neste texto, do outro tipo de *array*, o `char-table`.

Ex. 1 — Explique porque a implementação alternativa de `make-matrix` a seguir, menor e mais simples, não é equivalente àquela, e diga o que hás de errado com esta.

```
(defun make-matrix (rows cols init)
  (make-vector rows (make-vector cols init)))
```

3.3 Strings

Strings são semelhante a vetores, mas seus elementos sempre são caracteres.

Há diversas funções para tratamento de strings em Emacs Lisp.

`(concat s1 s2 ...)` concatena sequências e retorna uma string.

```
(concat "abc " "def") ;; "abc def"
```

`(substring s a b)` retorna a parte de `s` iniciando no caracter de índice `a` e terminando no índice anterior a `b`.

```
(substring "Um dois tres quatro" 3 7)   ;; "dois"
```

`clear-string` modifica um astring, preenchendo-a com zeros. A string passa a não usar mais codificação Unicode, e pode mudar de tamanho

```
ELISP> (setq a "ã ç ó")
"ã ç ó"

ELISP> (string-width a)
5 (#o5, #x5, ?\C-e)

ELISP> (clear-string a)
nil

ELISP> (length a)
8 (#o10, #x8, ?\C-h)

ELISP> (string-width a)
16 (#o20, #x10, ?\C-p)
```

`string=` verifica se duas strings são iguais. A função é sensível a caixa alta e baixa, mas ignora propriedades de texto. `string<` verifica se uma string é menor, lexicograficamente, que outra.

```
(string= "abc" (string ?a ?b ?c))  ;; t
(string= "xy" (concat [ ?x ] "y")) ;; t
(string= "AB" "ab")                ;; nil
```

`downcase` e `upcase` mudam todos os caracteres de uma string para caixa baixa ou caixa alta.

3.3.1 Formatando strings

Emacs Lisp tem uma função `format`, semelhante a `FORMAT` de Common Lisp, e `printf` de C.

Em (format string [objetos]), string é uma "srting de controle". O resultado será semelhante a string, substituindo argumentos dentro dela pelos objetos, na ordem em que aparecem. Por exemplo, a marcação %x na string de controle indica que um número é esperado como argumento, e essa parte da string será trocada pela representação do número em hexadecimal.

```
(format "número: %x" 30) ;; "número:  1e"
```

Os possíveis marcadores são mostrados a seguir.

- %s string
- %d número decimal com sinal
- %o número octal
- %x número hexadecimal
- %X mesmo que %x, mas em caixa alta
- %e número em notação exponencial
- %f número em notação de ponto flutuante
- %g número: ponto flutuante se menos que 6 dígitos, exponencial se 6 ou mais
- %c um único caracter
- %S um objeto Lisp, da forma como seria impresso pela função prin1.

```
ELISP> (format "número pequeno: %g. número grande: %g" 100 5000000)
"número pequeno: 100. número grande: 5e+06"

ELISP> (format "Bom dia, %s! Veja uma estrutura Lisp: %S"
        "Fulano"
        '(lista (aninhada)))
"Bom dia, Fulano! Veja uma estrutura Lisp: (lista (aninhada))"
```

Para incluir o caracter % em uma string de controle, use %%.

3.3.2 Conversão de/para srtings

char-to-string, string-to-char convertem entre string e caracter. Se string-to-char for chamada com uma string de comprimento maior que um, converterá somente o primeiro caracter. A string vazia é convertida em zero.

```
(char-to-string ?x)              ;; "x"
(string-to-char "um dois três") ;; ?u
```

number-to-string, string-to-number convertem entre string e número. Um segundo argumento para string-to-number determina a base em que o número será lido (se o argumento não for passado, a base será dez).

```
(number-to-string -25)      ;; "25"
(number-to-string  0)       ;; "0"
(string-to-number "30")     ;; 12
(string-to-number "30" 16) ;; 48
(string-to-number "0A" 16) ;; 10
(string-to-number "11" 2)  ;; 3
```

3.4 Vetores

Em Emacs Lisp, um vetor pode conter elementos de diferentes tipos.

```
ELISP> (setq v (vector 'a 10 "xy" '(a b c)))
[ a 10 "xy" (a b c) ]

ELISP> (aset v 2 ?z)
122 (#o172, #x7a, ?z)

ELISP> v
[ a 10 122 (a b c) ]          ;; não há diferença entre caracter (?z) e número
inteiro (122)

ELISP (aref v 3)
(a b c)
```

A funçãovconcat concatena sequências, retornando um vetor.

```
ELISP> (vconcat [ 10 20 30 ] [ 'a 5 "xyz" ])
[ 10 20 30 'a 5 "xyz" ]
```

Embora não haja suporte nativo a matrizes em Emacs Lisp, é fácil construir as funções necessárias. Podemos implementar matrizes como um vetor de linhas, onde cada linha é uma coluna (a Fig

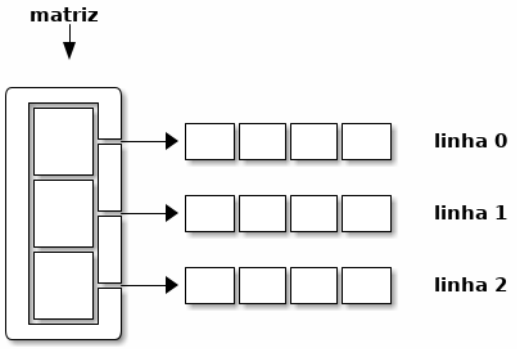


Figura 4: Matriz com 3 linhas e 4 colunas, implementada como vetor de vetores.

A função a seguir cria uma matriz usando essa representação.

```
(defun make-matrix (rows cols init)
  (let ((m (make-vector rows nil)))
    (dotimes (r rows)
      (aset m r (make-vector cols init)))
      m))
```

Para acessar e modificar os elementos da matriz, as duas função abaixo podem ser usadas.

```
(defun matref (m i j)
  (aref (aref m i) j))
```

```
(defun matset (m i j x)
  (aset (aref m i) j x))
```

Também é possível usar um único vetor para representar matrizes, armazenando consecutivamente as linhas, como ilustrado na Figura 5.

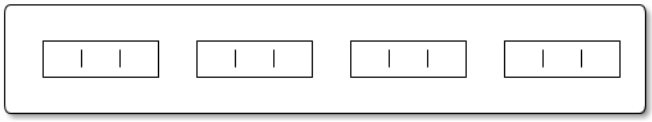


Figura 5: Matriz com 3 linhas e 4 colunas, implementada como único vetor.

3.5 bool-vectors

Vetores booleanos podem ser usados para representar conjuntos.

Suponha que tenhamos uma quantidade de possíveis elementos que podem estar em conjuntos. Se o conjunto Universo (contendo todos os elementos possíveis) tem tamanho n, um vetor booleano de tamanho n representa um subconjunto: cada posição do vetor indica se um certo element oestá presente ou não. As funções para isso são `bool-vector-union`, `bool-vector-intersection`, `bool-vector-set-difference`, `bool-vector-subsetp`.

É evidente que também seria interessante poder realizar operações lógicas com vetores booleanos. As funções Emacs Lisp para realizar operações lógicas em vetores booleanos são:

```
bool-vector-not             ;; negação de cada elemento
bool-vector-exclusive-or    ;; ou exclusivo, em cada posição dos vetores

ELISP> (let ((a (bool-vector t t   nil nil))
             (b (bool-vector t nil t   nil)))
         (print (vconcat (bool-vector-not a)))
         (print (vconcat (bool-vector-exclusive-or a b)))
         nil)

[nil nil t t]          ;; negação de a = [ t t nil nil]
[nil t t nil]          ;; [ t t nil nil] ⊕ [ t nil t nil ]
nil
```

Denotamos (fora do Emacs Lisp) a operação de ou-exclusivo por $\oplus$, e a segunda linha da resposta do REPL é `[nil t t nil]`, porque

```
  [ t   t   nil nil ]
⊕[ t   nil t   nil ]
=[ nil t   t   nil ]
```

Aparentemente faltam funções para *and* e *or*. Uma inspeção mais cuidadosa mostra que podemos usar funções já definidas para conjuntos:

```
(fset 'bool-vector-and 'bool-vector-intersection)
(fset 'bool-vector-or  'bool-vector-union)
```

`bool-vector-count-population` calcula a quantidade de elementos `t` em um bool-vector (ou seja, calcula o peso de Hamming do vetor).

```
(bool-vector-count-population (bool-vector t t nil t)) ;; 3
```

4 A avaliação de expressões

A maneira como o Emacs Lisp[5] 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 6);
- Se a expressão for uma lista, a avaliação dependerá do primeiro elemento da lista: se ele for uma forma especial[6], a avaliação se dará de forma específica para cada forma. Caso contrário, o interpretador avaliará o primeiro elemento, que deve resultar em uma função. Em seguida avaliará os outros elementos da lista, e usará os valores retornados como argumentos para o procedimento.

O Emacs Lisp – como toda linguagem Lisp – tem uma função que aceita uma expressão (que é uma lista), avalia como se fosse código Emacs Lisp, e retorna seu resultado. O nome da função é `eval`. A seguir damos alguns exemplos.

```
(eval  '(+ 6 5))                ;; 11

(eval '(if (< 10 5) 'broken
                    'ok))        ;; ok

(let ((a 10))
  (eval 'a))                     ;; 10

(let ((a 10))
  (eval '(+ a 20)))              ;; 30
```

[5] Todos os Lisps fazem o mesmo.
[6] Ou uma "macro".

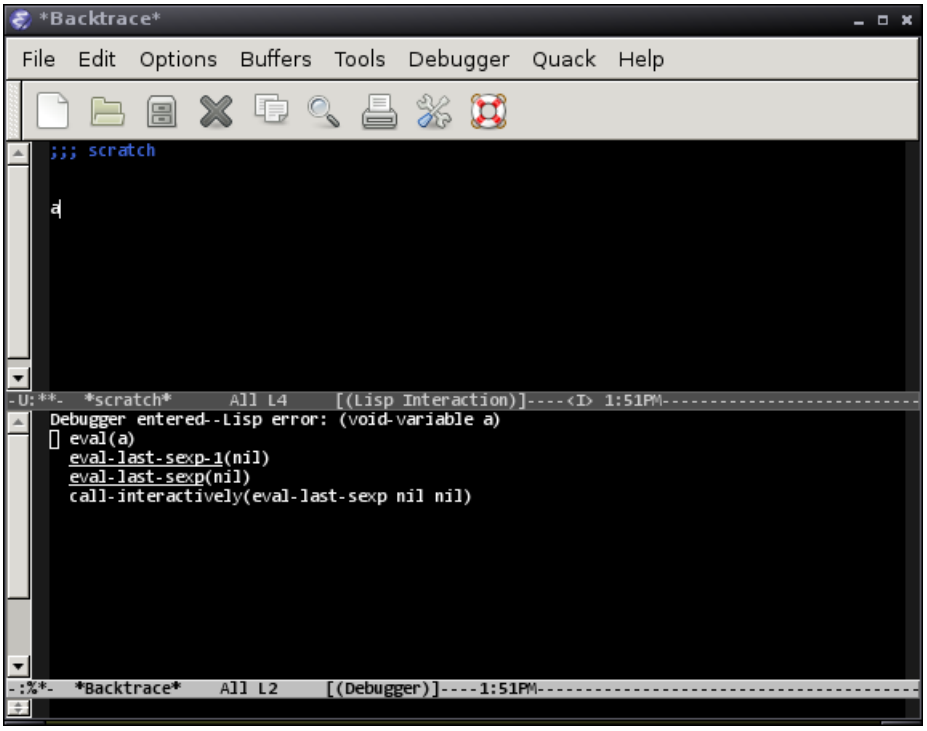


Figura 6: O Emacs apresenta um erro após uma tentativa de avaliar um símbolo sem valor.

`eval` pode aceitar, opcionalmente, um segundo argumento. Se este for `nil` (ou se não for passado), a avaliação será feita usando escopo dinâmico; se for `t`, será feita usando escopo léxico.

O uso de `eval`, no entanto, não é encorajado, porque pode tornar o código obscuro, evitar que trechos sejam compilados e até inserir falhas de segurança.

O Emacs também oferece funções `eval-region`, que toma tudo o que estiver na região e envia para `eval`; e `eval-buffer`, que avalia um buffer inteiro.

4.1 Formas especiais e Macros

Porque `defvar` não é uma função? Se fosse função, quando avaliássemos

```
(defvar x 10)
```

o Emacs Lisp começa a avaliar a lista. Voê que o primeiro símbolo é o nome de uma função. Então, avalia `todos` os seus argumentos – inclusiv eo `x`. Mas se `x` for avaliado, o efeito não será o que queremos! Se `x` não tiver sido definido antes, um erro será sinalizado. Por isso `defvar` é tratado de maneira especial pelo Emacs Lisp. De fato, dizemos que é uma *forma especial*: seus argumentos não sào avaliados por padrão.

`and` e `or` também são formas especiais. O código a seguir exemplifica isso.

```
(defun inverse (x)
  (and (not (zerop x)) (/ 1.0 x)))
```

É um código um tanto obscuro – seria melhor ser mais explícito, mas ele nos servirá de exemplo. Quando todos os argumentos avaliam para algo diferente de `nil`, o `and` retorna

seu último argumento, que neste caso será o inverso de `x`. Mas isto só funciona se `and` não for uma função. Se fosse, os *dois* parametros, `(not (zerop x))` e `(/ 1.0 x)` seriam avaliados, e um erro seria sinalizado se `x` fosse zero.

Da mesma forma que `defvar`, `and` e `or`, em toda linguagem do tipo Lisp, o controle de fluxo que implementa escolha (`if`, `cond`) é implementado como forma especial. Antes de uma forma Lisp ser avaliada, as formas especiais e *macros* nela são expandidas.

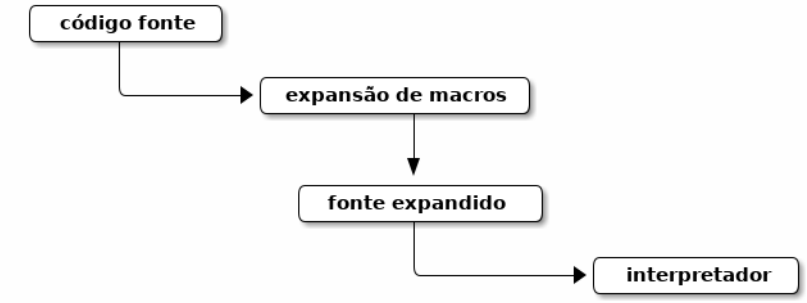


Figura 7: Macros são expandidas antes da avaliação de formas.

Por exemplo, a

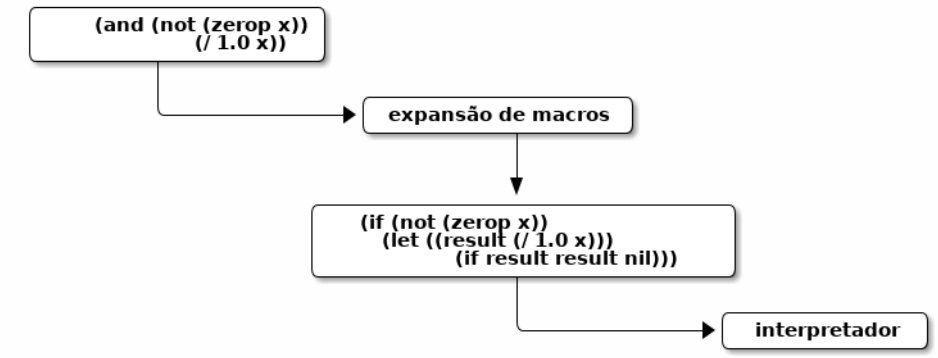


Figura 8: Expansão de uma expressão contendo `and`.

É possível ao programador construir algo parecido com formais especiais – as *macros*. Uma macro é semelhante a uma função, exceto que ela retorna código.

Definir uma macro `foo` com argumentos `a, b` é o mesmo que definir uma função que aceita os *nomes* dos argumentos `a, b` e devolve o código – uma lista Lisp – que é como gostaríamos que a macro `(foo a b)`fosse expanida

`defmacro` é usada para definir novas macros. Como exemplo inicial (e bobo), faremos uma versão piorada do `and`, que só aceita três argumentos.

Queremos que `(silly-and a b c)` expanda para

```
(if a
    (if b
```

```
        (if c t)
        nil)
    nil))
```

Usaremos quasiquote para construir o resultado. Sabemos que

```
`(if ,x (if ,y (if ,z t nil) nil) nil)
```

é o mesmo que

```
(list 'if ('if y ('if z t nil) nil) nil)
```

portanto podemos definir a macro aceitando parametros `x, y, z` e retornando a forma construída com quasiquotes:

```
(defmacro silly-and (x y z)
  `(if ,x
       (if ,y
           (if ,z t nil)
           nil)
       nil))

(silly-and 1 'a "huh?") ;; t
(silly-and 2 nil 'a)    ;; nil

(silly-and (print "Passei por aqui!") 2 nil) ;; nil

(silly-and 1 nil (print "Passei por aqui!")) ;; nil
```

Para testar macros, é útil usar `macroexpand`, que mostrará como uma forma seria expandida:

```
(macroexpand
 '(silly-and 1 2 3))

;; (if 1 (if 2 (if 3 t) nil) nil)
```

Há também variantes, `macroexpand-all` e `macroexpand-1`.

Para ver e usar a expansão de uma macro em um buffer, e não no REPL, pode ser interessante usar `insert` e `format`:

```
(insert (format "%S" (macroexpand (quote
  (macro args)
))))
```

4.2 Uso de macros

4.2.1 Formas de controle

```
(let ((a (busca-valor-de-a)))
  (if (> a 0)
      faz-algo
      não-faz-algo))
```

```
(if-let (a (> a 0)) (busca-valor-de-a)))
        faz-algo
        não-faz-algo)

(defmacro if-let (var-and-test then &optional else)
  (let ((var  (nth 0 var-and-test))
        (test (nth 1 var-and-test)))
    `(let ((,var ,test))
       (if ,var ,then ,else))))
```

4.2.2 Macros anafóricas

A forma especial `lambda` constrói funções anônimas, mas não conseguimos com ela criar funções recursivas, porque uma função anônima não tem nome:

```
(fset 'fat (alambda (n)
               (if (< n 2)
                   1
                   (* n (??? (- n 1)))))) ;; 'fat ainda não foi definido!
```

Seria interessante podermos usar o símbolo `self` para nos referirmos a "esta função".

```
(defmacro alambda (parms &rest body)
  `(let (self)
     (fset 'self (lambda ,parms ,@body))
     #'self))
```

- `(let (self)` é o mesmo que `(let ((self nil))`. Não queremos mudar o valor de alguma variável "self" que havia fora da nossa definição.
- `(fset 'self (λ,parms ,@body))` cria a função, usando a forma `lambda`, e armazena a função no símbolo `self`.
- finalmente, o valor função `#'self` é retornado.

```
(fset 'f (alambda (n)
           (if (< n 2)
               1
               (* n (self (- n 1))))))

(f 4) ;; 24
```

4.2.3 Macros `with-`

```
(defmacro with-positive (x &optional default)
  (let ((value (gensym 'value)))
    `(let ((,value ,x))
```

```
    (if (>= ,value 0)
        ,value
        ,default))))
```

4.2.4 Linguagens de domínio específico (DSLs)

Macros Lips são conhecidas por seu uso em *linguagens de domínio específico*[7]. Aqui, "domínio" se refere a algum domínio de aplicação. Damos um exemplo muito breve nesta seção[8].

As grandezas físicas que normalmente usamos podem ser descritas por algumas dimensões básicas. Por exemplo, para descrever fenômenos mecânicos, precisamos apenas de três dimensões: espaço (ou comprimento, *length*), que denotamos por L; massa, que denotamos M; e tempo, T.

- área é medida em L^2, comprimento ao quadrado:
- velocidade é medida em L/T^2, ou ainda, LT^{-1} – eslocamento por tempo: km/h, m/s, etc.
- pressão

Ao escolhermos unidades básicas para L, M, e T, determinamos unidades para todas as grandezas:

- no sistema métrico, *metro* é a unidade báscia de L; *grama* é a unidade básica de massa; e *segundo* é a unidade básica de T.
- no sistema imperial, não há uma unidade básica de L, mas poderíamos usar, por exemplo, a *polegada* (2.54cm); para massa, a *libra internacional* (453.59237g); para tempo, usa-se também segundos.

4.3 Armadilhas de macros

Há alguns problemas comuns no uso de macros do tipo que Emacs Lisp oferece[9]:

- **efeitos colaterais:** uma macro é uma função que escreve código. Sendo uma função, ela poderia causar efeitos colaterais. Idealmente, uma macro não deve usar nada além de seus argumentos, e não deve causar qualquer efeito colateral – deve apenas produzir código.

```
(defmacro call-it (name &rest args)
  (if (member name forbidden)
        '(progn)
        `(,name ,@args)))
```

[7] Domain-specific languages
[8] Inspirado no exemplo de Doug Hoyte, em "Let Over Lambda"
[9] As macros de Scheme são diferentes, e tentam proteger o programador de alguns desses problemas.

Esta macro depende da variável `forbidden`, e não somente de seus argumentos – mais ainda, ela depende do valor dessa variável *no momento em que a macro foi expandida*, que é algo difícil de determinar.

- **múltiplas avaliações:** os argumentos de uma função sempre são avaliados, e por isso, quando uma função é desenvolvida, sempre podemos contar com argumento sendo valores. Os argumentos de uma macro são *formas Lisp*, e serão expandidos em todos os lugares do código onde forem mencionados. Isso pode fazer com que sejam avaliados mais de uma vez.

  Para exemplo, construiremos uma *macro anafórica*.

```
(defmacro aif (test then else)
  `(if ,test
       (let ((it ,test))
            ,then)
       ,else))

(aif (progn (print 'OH) 10)
     (print it)
     (print "no x!"))

OH
OH
10
```

  Não é o que queremos! A forma `test` foi avaliada duas vezes. Se ali houvesse o incremento de uma variável com `incf`, por exemplo, isso também ocorreria duas vezes!

  O que queremos é uma expansão diferente, criando uma variável temporária para armazenar o resultado do teste.

```
(defmacro aif (test then else)
  (let ((test-var (gensym 'testvalue)))
    `(let ((,test-var ,test))
       (if ,test-var
         (let ((it ,test-var))
           ,then)
         ,else))))
```

- **ordem de avaliação:** os argumentos de funções em Emacs Lisp são avaliados *da esquerda para a direita*[10]. Ao escrever uma macro, o programador pode escolher a ordem em que quer que os argumentos sejam avaliados. Se uma macro avaliar seus argumentos em ordem diferente da que é usada para funções, poderá causar estranheza em quem a usa.

[10] Scheme, por exemplo, não especifica ordem de avaliação de argumentos.

- **captura de símbolo:** uma macro pode expandir código que inclui algum símbolo já em uso.

  Para exemplificar, criaremos uma macro parecida com `and`, mas que retorna `t` somente quando a maioria de seus argumentos avaliar para algo não-nulo. A macro não deve avaliar mais argumentos que necessário (se chegar a um quórum, para de avaliar e retorna `t`).

  Primeiro construímos uma função que aceita uma lista de argumentos, um nome de símbolo, e uma quantidade (o "quórum" necessário para retornar `t`):

```
(defun vote-aux (args counter quorum)
  (if (null args)
      nil
      `(progn (when ,(car args) (incf ,counter))
              (if (> ,counter ,quorum)
                  t
                  ,(vote-aux (cdr args) counter quorum)))))
```

  A expansão de `(vote-aux '(a b c) 'i 2)` é

```
(progn  (when a (incf i))
        (if (>= i 2)
            t
            (progn (when b (incf i))
                   (if (>= i 2)
                       t
                       (progn (when c (incf i))
                              (if (>= i 2)
                                  t
                                  nil))))))
```

  A macro simplesmente calcula o tamanho da lista de argumentos, o quórum, e gera código que inicializa a variável `quorum` para zero. Depois concatena isso com a saída de `(vote-aux args 'counter quorum)`.

```
(defmacro vote (&rest args)
  (let ((size (length args)))
    (let ((quorum (/ size 2.0)))
          (append '(let ((counter 0)))
                  (list (vote-aux args 'counter quorum))))))
```

  Usando `(macroexpand '(vote a b))`, verificamos a expansão:

```
(let ((counter 0))
     (progn (when a (incf counter))
            (if (> counter 1.0)
                t
                (progn (when b (incf counter))
                       (if (> counter 1.0)
```

```
                        t
                        nil)))))
```

Aparentemente, tudo está bem, e construímos nossa macro. No entanto, a macro expande para uma forma uqe usa o símbolo `counter`. Se o usarmos como argumento, a macro deixa de funcionar como esperado: `(macroexpand '(vote (decf counter) b)`:

```
(let ((counter 0))
  (progn (when (decf counter) (incf counter))
         (if (> counter 1.0)
             t
             (progn (when b (incf counter))
                    (if (> counter 1.0)
                        t
                        nil)))))
```

Logo na segunda linha vemos que a expressão `(decf counter)` interferirá no funcionamento da macro.

Há diferentes maneiras de lidar com este problema. Um deles é usar símbolos que certamente nunca foram usados. Como o programador não tem como saber que símbolo nunca foi usado antes, a linguagem nos oferece uma função que faz exatamente isso:

`gensym` cria um novo símbolo, que não foi ainda internalizado.

```
ELISP> (gensym)
g2066
ELISP> g2066
*** Eval error ***  Symbols value as variable is void: g2066
```

`gensym` aceita um argumento, que será o prefixo do símbolo gerado. Este argumento pode ser *qualquer* objeto Lisp!

```
ELISP> (gensym 'whatever)
whatever982
ELISP> (gensym "hello")
hello983
ELISP> (gensym 10)
\10984
ELISP> (gensym '(a b c))
\(a\ b\ c\)985
ELISP> (gensym '[a b c])
\[a\ b\ c\]986
```

Reconstruímos nossa macro, desta vez usando `gensym`.

```
(defmacro vote (&rest args)
  (let ((size (length args)))
    (let ((quorum (/ size 2.0))
          (counter (gensym 'counter)))
```

```
      (append `(let ((,counter 0)))
              (list (vote-aux args counter quorum))))))
```

Desta vez a expansão mostra que não haverá conflitos entre nomes de variáveis: `(macroexpand '(vote (decf counter) b))`

```
(let ((counter981 0))
  (progn (when (decf counter) (infc counter981))
         (if (> counter981 1.0)
             t
             (progn (when b (infc counter981))
                    (if (> counter981 1.0)
                        t
                        nil)))))
```

- **recursão:** embora seja possível construir macros recursivas, é necessário cuidado especial para garantir que a recursão termine.

  Como exemplo tentaremos construir uma versão macro da função fatorial (somente para este exemplo – este é um caso em que uma função funciona melhor que uma macro)

  ```
  (defmacro fatorial (n)
    `(if (<= ,n 1)
         1
         (* ,n (fatorial (- ,n 1)))))
  ```

  A macro acima não funciona: `(fatorial 3)` resulta em um erro. O debugger do Emacs nos diz

  ```
  Debugger entered--Lisp error: (error "Lisp nesting exceeds max-lisp-eval-d
  ```

  O nível máximo de recursão foi atingido ao tentar avaliar a forma. Isso porque a expansão dessa macro continua infinitamente, e só poderia parar se `(- n 1)` fosse avaliado, mas isso não ocorre.

  ```
  ELISP> (macroexpand-1 '(fatorial 3))

  (if (<= 3 1)
      1
      (* 3 (fatorial (- 3 1))))

  ELISP> `(if (<= 3 1) 1 (* 3 ,(macroexpand-1 '(fatorial (- 3 1)))))

  (if (<= 3 1)
      1
      (* 3 (if (<= (- 3 1) 1)
               1
               (* (- 3 1)
                  (fatorial (- (- 3 1) 1))))))
  ```

```
ELISP> `(if (<= 3 1) 1 (* 3 (if (<= (- 3 1) 1) 1 (* (- 3 1)
       ,(macroexpand-1 '(fatorial (- (- 3 1) 1)))))))

(if (<= 3 1)
    1
    (* 3 (if (<= (- 3 1) 1)
             1
             (* (- 3 1)
                (if (<= (- (- 3 1) 1) 1)
                    1
                    (* (- (- 3 1) 1)
                       (fatorial (- (- (- 3 1) 1) 1)))))))))
```

4.4 Estudo avançado de macros

Embora muito poderosas, macros não são comuns em código Lisp.

Usualmente, tenta-se usar funções sempre que possível, sendo o recurso das macros reservado para abstrações onde elas realmente fazem sentido – *em geral, abstrações de sintaxe, e não de semântica.* Em particular, *é sempre desaconselhado usar macros para ganhar eficiência – ao invés disso, eve-se melhorar algoritmos, realizar profiling e usar, se necessário, funções inline.*

Ainda assim, é interessante estudá-las, e há uma grande quantidade de material escrito sobre elas.

As macros de Emacs Lisp são muito parecidas com as de Common Lisp, e o nível de semelhança é tal que é possível usar material instrucional de Common Lisp para aprender mais sobre macros de Emacs Lisp.

No entanto, nem todo programador Emacs Lisp usa o estilo de programação usado em livros sobre Common Lisp. Ao tentar contribuir código para um projeto existente, é sempre bom estudar o estilo usado no projeto antes de tentar contribuir.

Além de livros comuns de Common Lisp (“Land of Lisp”, de Conrad Barski, e “Practical Comomn Lisp”, de Peter Seibel são duas publicações recentes muito boas), há dois que se destacam por tratarem especialmente de macris: *On Lisp*, de Paul Graham, e *Let Over Lambda*, de Doug Hoyte. As técnicas descritas nestes dois últimos dois livros, no entanto, são avançadas e acabam sendo raramente usadas na prática. A leitura vale, ainda assim.

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 $0 \leq x < n$.

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 por default. 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.

5.2 Escopo léxico

O uso de escopo dinâmico era comum em variantes antigas de Lisp, mas as formas mais modernas (como Scheme e Common Lisp) implementam escopo léxico, que oferece menos oportunidade ao programador para cometer erros, além de permitir mais possibilidades de otimização do código quando este é compilado.

O Emacs Lisp permite usar escopo léxico também, de duas maneiras:

- usando `lexical-let` ao invés de `let`, ou
- modificando a variável `lexical-binding` para `t` no buffer em que o `let` estiver sendo executado.

A seguir temos um exemplo 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)
```

Teríamos conseguido o mesmo usando a variável `lexical-binding`:

```
(setq lexical-binding t)

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

(setq total 10000)

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

Ao invés de usar código Elisp para modificar o valor de `lexical-binding`, também podemos usar o cabeçalho do arquivo: se ele contiver a linha `;;; -*- lexical-binding: t -*-`, o efeito será o mesmo, *desde que o buffer seja compilado ou enviado inteiro para* `eval`:

```
;;; -*- lexical-binding:  t -*-
(let ((total 100))
  (defun lex-prop (valores)
    (mapcar (lambda (v) (/ (+ 0.0 v) total))
            valores)))

(setq total 10000)

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

Mesmo quando `lexical-binding` é `t` em um *buffer*, as variáveis criadas com `defvar` sempre terão escopo dinâmico.

O uso de escopo dinâmico traz dificuldades para a implementação eficiente de *threads*. Aos poucos, o código interno do Emacs tem migrado de escopo dinâmico para léxico. Sempre que possível, escolha escopo léxico!

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 Decisões

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.

A forma `case` pode ser usada para escolher um dentre vários valores.

```
(setq opcao 2)

(insert
  (case opcao
    (1 "laranja")
    (2 "abacaxi")
    (t "água"))))
```

A sintaxe para `cond` é

```
(case <exp>
      (<exp1> <corpo1>)
```

```
      (<exp2> <corpo2>)
       ...)
```

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

9 Repetições

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, ..., $max - 1$. Exemplo:

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

10 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`.

11 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 b)`: seleciona e ativa o *buffer* cujo nome é `b`. O *buffer* será ativado na janela corrente.

`(set-buffer b)` Muda o foco do Emacs para o buffer no qual programas rodarão para aquele cujo nome é `b`. 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 obtido, e pode ser, entre outros:

- b: O nome de um buffer;
- f: O nome de um arquivo;
- n: um número, lido no minibuffer;
- 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 em que a função é
  ;; chamada
  (message "Posição = %d" pos))
```

12 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
```

13 Buffers

Buffers são áreas na memória onde o Emacs armazena dados (usualmente texto, para exibição em janelas). Os buffers do Emacs são, claro, objetos de prieira classe – podem ser armazenados em variáveis, e é possível verificar se um objeto é um buffer com a função `bufferp`.

Há funções Emacs para inserir mais texo em um buffer, para apagar um buffer inteiro, e, claro, várias outras utilidades – afinal de contas, o Emacs nasceu como um "editor de textos programável". Algumas destas funções (há muitas outras – consulte o manual de referência do Emacs Lisp) são listadas a seguir.

- `(current-buffer)` retorna o objeto do buffer atual
- `(buffer-name buf)` retorna o nome do buffer `buf`
- `(rename-buffer nome)` renomeia o buffer atual para `nome`
- `(get-buffer nome)` retorna o objeto buffer cujo nome é `nome`

- `(generate-new-buffer-name nome)` gera um novo nome de buffer, único, começando com `nome`
- `(buffer-file-name buf)` retorna o nome do arquivo que está sendo visitado pelo buffer `buf`
- `(kill-buffer buf)` elimina o buffer `buf`
- `(buffer-list)` retorna a lista de buffers

A função interativa a seguir cria um buffer e mostra nele uma mensagem. O buffer ficará aberto em uma nova janela, em modo somente de leitura, e poderá ser fechado com a tecla `q`. Nesta função, usamos:

- `(get-buffer-create buf)`: se `buf` é uma string, a função retornará um buffer com o nome dado. Um que já exista com este nome, se houver, ou um novo.
- `(erase-buffer)`
- `(with-current-buffer buf ...)` muda o buffer atual pata `buf` durante a execução do código dentro de seu escopo.
- `(pop-to-buffer buf)` vai para uma nova janela, e seleciona, nesta nova janela, o buffer identificado por `buf`, que pode ser um objeto buffer ou uma string com o nome de um buffer. A janela poderá ser criada, se o buffer rquisitado não estiver já aberto em alguma janela do Emacs.
- `(view-mode)` entra no modo *view mode*: o buffer passa para o estado de somente leitura, e algumas teclas passam a ser vinculadas a ações relacionadas à visualização apenas – por exemplo, `s` e `r` fazem busca para frente e para trás; espaço e `Del` avançam e retrocedem página; `q` sai do *view mode* e restaura a janela ao seu estado anterior.

```
(defun show-message (msg)
  (interactive "smensagem? ")
  (let ((buffer (get-buffer-create "*saida*")))
    (with-current-buffer buffer
      (erase-buffer)
      (insert (concat "IMPORTANT MESSAGE:\n\n" msg)))
    (pop-to-buffer buffer)
    (view-mode)))
```

14 Debuggers

O Emacs tem dois debuggers: `debug`, que permite observar a pilha de avaliação (quais S-expressões foram sendo avaliadas até chegar à situação atual); e o `edebug`, que tem como objetivo acompanhar a execução de programas elisp, passo a passo.

14.1 debug

O debug mostra uma pilha de execução e permite navegar por ela, examinando-a, verificando valores de variáveis, e eventualmente prosseguindo com o programa. Por default, sempre que um erro em Emacs Lisp acontece, o debug é chamado. Uma janela é aberta, com um *backtrace*, e vários comandos ficam disponíveis para examiná-lo. Alguns deles são

- e avalia uma expressão, usando os valores das variáveis no stack frame escolhido.
- v mostra as variáveis locais em cada nível do *backtrace* – mas só está disponível se o *backtrace* envolver ao menos uma chamada de função.
- c sai do edebug e continua o programa, como se nada tivesse acontecido.
- q fecha o *backtrace* e sai do debug

Note que ao avaliar uma expressão com e, fará diferença se o Emacs estiver usando escopo léxico ou dinâmixco no buffer atual.

Há algumas variáveis que determinam quando o debug deve ser chamado. Quando diferentes de nil, o debug é chamado em situações específicas:

- debug-on-error determina que, sempre que um erro acontecer, debug será chamado
- debug-on-quit determina que o Emacs chame o debug em um evento *quit* – que é vinculado normalmente ao C-g. Se você usa C-g para interromper um processo, mas não sabe porque o processo estava em loop, pode usar (setq debug-on-quit t), reproduzir o problema, e usar C-g, que imediatamente o levará ao stacktrace do debug.

O debug pode ser chamado na entrada de uma função. Para isto, basta usar (debug-on-entry nome). Para cancelar, (cancel-debug-on-entre nome).

Da mesma forma que na entrada de funções, o debug pode ser chamado sempre que o valor de uma variável mudar, usando (debug-on-variable-change nome). Para cancelar, (cancel-debug-on-variable-change nome).

Também é possível chamar debug explicitamente em qualquer lugar onde uma expressão Emacs Lisp é avaliada (em seu programa elisp, em um bufer que esteja no modo elisp, no REPL, ou mesmo interativamente, com M-x debug.

Além disso, quando o Emacs é chamado com a opção --debug-init, ele liga debug-on-error antes de ler seu arquivo de configuração.

14.2 edebug

O edebug é um debugador interativo semelhante aos tradicionais debuggers em ambientes de programação, que permitem executar cada linha de um programa, verificando o estado de objetos na memória. Em Lisp, no entanto, faz mais sentido *avaliar uma expressão* do que "executar uma linha".

Para marcar uma função, indicando que ela deve ser debugada, vá até sua definição e useC-u C-M-x, ou chame interativamente a função edebug-defun. Quando a função for chamada, será possível acompanhar, passo a passo, sua execução, usando várias diferentes teclas associadas a comandos do edebug. Alguns destes comandos são:

- `n` segue para a avaliação da próxima expressão
- `o` sai da forma atual – o que pode ser muito útil para sair de um loop
- `i` "entra" debugando na chamada de função a ser avaliada, ao invés de simplesmente chamá-la
- `e` avalia uma expressão dada
- `q` termina a execução

14.3 trace

O Emacs vem com um rastreador (tracer) simples de funções.

- `(trace-function fun)` marca a função de nome `fun` para ser rastreada (`fun` deve ser um *símbolo*).
- `(trace-function-background fun)` marca a função de nome `fun` (símbolo) para ser rastreada, mas o traço será gravado em um buffer separado.
- `(untrace-function fun)` desmarca a função `fun` (símbolo), que não será mais rastreada.
- `(untrace-all)` desmarca todas as funções, e nenhuma delas será mais rastreada.

```
(defun dentro (x y)
  (< (sqrt (+ (* x x)
              (* y y)))
     1.0))

(defun random-float ()
  (/ (float (random 1000000)) 1000000.0))

(defun aprox-pi (n)
  (let ((pontos-dentro 0.0))
    (dotimes (i n)
      (let ((x (random-float))
            (y (random-float)))
        (when (dentro x y)
          (setq pontos-dentro (+ 1.0 pontos-dentro)))))
    (* 4 (/ pontos-dentro (float n)))))

(mapcar #'trace-function '(dentro random-float aprox-pi))
(aprox-pi 2)

======================================================================
1 -> (aprox-pi 2)
| 2 -> (random-float)
```

```
| 2 <- random-float: 0.416228
| 2 -> (random-float)
| 2 <- random-float: 0.846292
| 2 -> (dentro 0.416228 0.846292)
| 2 <- dentro: t
| 2 -> (random-float)
| 2 <- random-float: 0.362399
| 2 -> (random-float)
| 2 <- random-float: 0.986528
| 2 -> (dentro 0.362399 0.986528)
| 2 <- dentro: nil
1 <- aprox-pi: 2.0
```

Para tentar uma aproximação mais realista, mais iterações são necessárias:

```
(mapcar #'byte-compile '(dentro random-float aprox-pi))
(aprox-pi 1000000)

3.14108
```

15 Exemplos

Esta seção mostra alguns exemplos de código em Emacs Lisp.

15.1 Buscando parênteses pelo buffer - sem expressões regulares

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.

Ex. 2 — As funções `acha-par-*` 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.

15.2 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.

Ex. 3 — Crie uma função para desfazer o destaque feito por `destaca-s-expressao`.

15.3 Mudando o navegador

Há vários pacotes Emacs que podem abrir URLs em um navegador. Dependendo da situação, navegadores diferentes podem ser preferíveis. A função a seguir pergunta qual navegador o usuário quer usar, dando duas opções:

1. Default. O Emacs poderá chamar algum navegador externo, dependendo de quais estiverem disponíveis no sistema;
2. `w3m`. Neste caso, o pacote Emacs para interface com o navegador `w3m` deve estar disponível.

Nesta função, usamos:

- `browse-url-browser-function` determina que navegador o Emacs usa;
- `concat` concatena sequências, e strings são sequências de caracteres;
- `symbol-name` retorna a string com o nome de um símbolo (não podemos concatenar uma string com um símbolo).

A função `change-browser`, que desenvolvemos, muda o valor de `browse-url-browser-function` de acordo com a opção escolhida.

```
(defun change-browser (choice)
  "Muda o navegador"
  (interactive "nQual navegador quer usar? (1) default, (2) w3m ")
  (setq browse-url-browser-function
        (case choice
          (1 'browse-url-default-browser)
          (2 'w3m-browse-url)
          (t browse-url-browser-function)))
  (message (concat "browser mudado para "
           (symbol-name browse-url-browser-function))))
```

15.4 Abrindo e fechando tags XML – com expressões regulares

No ponto em que estiver no buffer, `insert-xml-tag` só precisa inserir <, a tag desejada, e >, e depois avançar até o final da tag.

```
(defun insert-xml-tag (tag)
  (let ((start (point)))
    (let ((text (concat "<" tag ">")))
      (insert text)
      (goto-char (+ start (length text))))))
```

`close-xml-tag` já é mais interessante: fechará a última tag aberta, qualquer que seja.

- usa `save-excursion` para lembrar onde estava

- procura pela expressão regular "<.*>", para trás, e grava o ponto em que começa o nome da tag em uma variável x
- avança até a expressão "[^\\]>" – que casa com >, mas não depois de \, e guarda o ponto do final do nome em y
- obtém o nome da tag usando buffer-substring
- sai do save-excursion e insere o fechamento da tag

```
(defun close-xml-tag ()
  (interactive)
  (let ((tag-name ""))
    (save-excursion
      (let ((x (+ (re-search-backward "<.*>") 1)))
        (goto-char x)
        (let ((y (- (re-search-forward "[^\\]>") 1)))
          (setq tag-name (buffer-substring x y)))))
    (insert (concat "</" tag-name ">"))))
```

Podemos criar funções específicas para inserir tags de interesse, e vinculá-las a atalhos de teclado.

```
(defun insert-xml-xyz ()
  (interactive)
  (insert-xml-tag "xyz"))

(global-set-key (kbd "C-M-s") insert-xml-xyz)
```

O defun para inserir uma tag específica não é completamente necessário; poderíamos ter resumido o defun e o global-set-key da seguinte maneira,

```
(global-set-key (kbd "C-M-s")
                (lambda () (interactive) (insert-xml-tag "xyz")))
```

Esta versão, embora mais compacta, é menos legível e menos elegante.

Ex. 4 — A função close-xml-tag não usa o resultado do casamento da expressão regular; ao invés disso, usa buffer-substring para obter o nome da tag. Tente remover buffer-substring, e usar o resultado de re-search-backward.

Ex. 5 — close-xml-tag retornará um erro quando o usuário tentar fechar uma tag sem ter nenhuma aberta. No entanto:

1. O erro retornado é um erro de re-search-backward, que não é claro nesta situação. Faça com que a função retorne outro erro, xml-no-open-tag.
2. Se o usuário tentar fechar uma tag mais de uma vez, o comportamento será incorreto: após abrir a tag <minha-tag> e chamar close-xml-tag três vezes, o resultado é

   <minha-tag></minha-tag><//minha-tag><///minha-tag>

   Conserte isto.

16 Framework de testes

O Emacs vem com um framework para testes – o ERT (Emacs-Lisp Regression Testing). O ERT é semelhante em espírito a outras ferramentas de testes, mas pode ser usado de forma muito dinâmica, devido à natureza do Emacs Lisp.

Um teste é definido pela macro `ert-deftest`:

```
(ert-deftest nome ()
  docstring
  :tags tags
  :expected-result res
  corpo)
```

No corpo do teste,

- `(should expr)` afirma que a expressão `expr` deveria ser diferente de `nil`
- `(should-error expr)` afirma que a expressão `expr` deveria resultar em erro

Definimos a seguir dois testes, para as funções que abrem e fecham tags XML. O primeiro insere e fecha a tag sem nenhum conteúdo dentro.

```
(ert-deftest insert-xml-tag-empty ()
  "test XML open and close with nothing in between"
  :tags '(open-close-xml)
  (with-current-buffer (get-test-buffer)
    (goto-char 20)
    (let ((x (point)))
      (po-insert-xml-tag "some-tag")
      (should (= (point) (+ x 2 (length "some-tag"))))
      (po-close-xml-tag)
      (should
       (string-match "<some-tag></some-tag>"
                     (buffer-substring
                       x
                       (+ x (length "<some-tag></some-tag>")))))))))
```

O segundo tenta fechar uma tag sem tê-la aberto.

```
(ert-deftest insert-xml-tag-unopened ()
  ""
  :tags '(open-close-xml)
  (with-current-buffer (get-test-buffer)
    (goto-char 20)
    (let ((x (point)))
      (should-error (po-close-xml-tag)))))
```

- `(ert-run-tests-interactively seletor)` executa os testes especificados pelo seletor, interativamente
- `(ert-run-tests-batch seletor)` executa os testes especificados pelo seletor

```
(ert-run-tests-interactively t)
```

O seletor, que no exemplo era simplesmente `t`, determina quais testes devem ser executados. Há várias possibilidades para o seletor:

- `t` seleciona todos os testes definidos
- uma string contendo uma expressão regular – neste caso somente os testes cujos nomes casam com a expressão serão selecionados.
- um símbolo com o nome de um único teste
- um objeto teste, retornado por `ert-deftest`
- `:failed` `:passed` `:expected` `:unexpected`

Para executar os testes cujo nome casem com a expressão `"insert-xml.*"`,
`(ert-run-tests-interactively "insert-xml.*")`
Para executar os que tem a tag `open-close-xml`,
`(ert-run-tests-interactively '(tag open-close-xml))`
E é possível combinar critérios com operadores booleanos (veja o manual info do ERT):
`(ert-run-tests-interactively '(and "insert-xml.*" :passed)`

16.1 Modo batch

O ERT também pode realizar o testes em modo *batch*. Isto é comum em pacotes distribuídos em repositórios, para permitir a automação de testes. A função `ert-run-tests-batch` recebe um seletor, realiza os testes selecionados, e escreve o resultado no terminal. Para os testes que desenvolvemos, ela pode ser chamada da seguinte forma:

```
(ert-run-tests-batch "insert-xml.*")
```

A função `ert-run-tests-batch-and-exit` é semelhante, mas sai do Emacs depois dos testes, retornando zero se nenhum teste falhou, e algo diferente de zero se algum teste falhou

```
emacs -batch -l ert -l testes.el -eval "(ert-run-tests-batch-and-exit '(tag open-
close-xml))"

Running 2 tests (2020-05-10 19:26:26-0300, selector (tag open-close-xml))
   passed  1/2  insert-xml-tag-empty (0.000202 sec)
   passed  2/2  insert-xml-tag-unopened (0.000107 sec)

Ran 2 tests, 2 results as expected, 0 unexpected (2020-05-10 19:26:26-0300, 0.000600 sec)
```

Ex. 6 — Escreva outros testes: um para o caso em que a tag não é vazia, e um para o caso em que o argumento de `insert-xml-tag` não é string.

Ex. 7 — Escreva testes para `destaca-s-expressao`.

17 Configuração (.init.el, antigo .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 "`.config/emacs/init.el`", dentro de seu diretório (em sistemas BSD ou GNU/Linux, `~/.config/emacs/init.el`; em Windows, `.config\emacs\init.el`, dentro da pasta do usuário). Antigamente, o arquivo lido era `~/.emacs`. Se não encontrar estes, o Emacs procurará outras possibilidades – consulte o manual.

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`.

18 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)
```

19 Chamando scripts Emacs na linha de comando

O Emacs Lisp é uma ótima linguagem de *scripting*. Usualmente, scripts em diferentes linguagens iniciam com uma linha contendo o par de caracteres `#!` (em Inglês, *"shebang"*). Isto acontece porque o caracter `#` é normalmente tratado como delimitador de início de comentário na maioria das linguagens usadas para scripting (Bash, Zsh, Python, Ruby, Perl, etc). O Emacs, desde a versão 22, aceita o parametro `-script`.

O programa a seguir joga dados. Se um parâmetro for dado, ele será usado como quantidade de dados a jogar. Se não for passado qualquer parâmetro, somente um dado será jogado.

```
#!emacs --script

(let ((dice-number
          (if (null command-line-args-left)
              1
              (string-to-number (nth 0 command-line-args-left)))))
  (dotimes (i dice-number nil)
    (princ (+ 1 (random 6)))
    (princ "\n")))
```

```
$ ./dado
3
$./dado 3
2
3
5
```

20 JSON

Há, incluso no Emacs, um codificador e decodificador para o formato JSON. As principais funções que podem ser usadas são:

- `json-encode` codifica um objeto Emacs Lisp, devolvendo uma string com sua representação em JSON.
- `json-read` lê um objeto JSON do buffer atual, e retorna sua representação em Emacs Lisp.
- `json-read-from-string` semelhante a `json-read`, mas lê de uma string ao invés de ler do buffer.
- `json-read-file` semelhante a `json-read`, mas lê de um arquivo.

Antes de usá-las, é necessário pedir ao Emacs que carregue a biblioteca `json`.
A seguir há um exemplo simples.

```
ELISP> (require 'json)
json
ELISP> (json-read-from-string  "{ \"a\":  1, \"b\": 2 }")
((a . 1)
 (b . 2))
ELISP> (json-encode (list (cons 'x 2)
                          (cons 'y "uma string qualquer")))
"{\"x\":2,\"y\":\"uma string qualquer\"}"
```

Quando a variável `json-encoding-pretty-print` é diferente de `nil`, o codificador produz uma string mais legível.

```
ELISP> (let ((json-encoding-pretty-print t))
             (json-encode (list (cons 'x 2)
                                (cons 'y "uma string qualquer"))))
"{
  \"x\": 2,
  \"y\": \"uma string qualquer\"
}"
```

- `json-object-type` determina como objetos JSON são representados ao serem *decodificados* (são usados por `json-read` e variantes).
  - `alist` (default)

    ```
    ELISP> (let ((json-object-type 'alist))
                 (json-read-from-string
                    "{
                       \"x\": 2,
                       \"y\": \"uma string qualquer\"
                     }"))
    ((x . 2)
     (y . "uma string qualquer"))
    ```

  - `plist`: como *property lists*.

```
ELISP> (let ((json-object-type 'plist))
             (json-read-from-string
               "{
                  \"x\": 2,
                  \"y\": \"uma string qualquer\"
                }"))

(:x 2
 :y "uma string qualquer")
```

  - `hash-table`: como tabelas de hashing. As chaves são, por default, strings.

```
ELISP> (setq h (let ((json-object-type 'hash-table))
                     (json-read-from-string
                       "{
                          \"x\": 2,
                          \"y\": \"uma string qualquer\"
                        }")))
#<hash-table equal 2/65 0x158b8b6df239>
ELISP> (gethash "x" h)
2 (#o2, #x2, ?\C-b)
```

- `json-array-type` determina como vetores JSON são representados.
  - `vector` (default): como vetores de Emacs Lisp.
  - `list`: como listas de Emacs Lisp.
- `json-key-type` determina como as chaves JSON são representadas em Emacs Lisp.
  - `nil` (default). Neste caso, a biblioteca `json` tentará adivinhar qual tipo usar.
  - `string`
  - `symbol`
  - `keyword`
- `json-false` é a representação que deve ser usada, em Emacs Lisp, para o valor JSON `false` quando traduzido para Emacs Lisp. O default é `:json-false`.
- `json-null` é a representação que deve ser usada, em Emacs Lisp, para o valor JSON `null`. O default é `nil`

Há outras variáveis de configuração que podem ser úteis:

- `json-encoding-separator` (default: `","`)
- `json-encoding-default-indentation` (default:)
- `json-encoding-pretty-print` (default: `nil or t`)
- `json-encoding-lisp-style-closings` (default: `nil or t`)

A seguir há alguns exemplos.

```
ELISP> (let* ((json-object-type 'alist)
              (json-encoding-pretty-print t)
              (json-array-type 'list)
              (json-null 'NADA)
              (json-false 'NO!)
              (json-key-type 'keyword))
         (json-read-from-string
           "{ \"a\": \"x\",
              \"b\": null,
              \"c\": false,
              \"e\": [ 1, 2, 3 ]  }"))

((:a . "x")
 (:b . NADA)
 (:c . NO!)
 (:e 1 2 3))
```

As chaves foram codificadas como `plists`. O vetor na chave `e` é codificado como lista, porque usamos `(json-array-type 'list)`.

Se trocarmos `(json-array-type 'list)` para `(json-array-type 'list)`, o resultado será o mesmo, exceto que o vetor JSON será decodificado em um vetor Emacs Lisp:

```
((:a . "x")
 (:b . NADA)
 (:c . NO!)
 (:e .
   [1 2 3]))

ELISP> (let* ((json-encoding-pretty-print t)
              (json-object-type 'alist)
              (json-array-type 'vector)
              (json-key-type 'symbol)
              (v (make-vector 3 "a")))
         (aset v 1 "b")
         (aset v 2 -1)
         (json-encode (list (cons 'x 2)
                            (cons 'y "xyz")
                            (cons 'z v))))

"{
  \"x\": 2,
  \"y\": \"xyz\",
  \"z\": [
    \"a\",
    \"b\",
    -1
```

```
  ]
}"
```

21 XML

21.1 Parsing

O Emacs tem com um parser para XML, acessível através de algumas funções.

A função `libxml-parse-xml-region` recebe dois parâmetros – início e final da região no buffer corrente – e faz o parsing do XML que estiver nessa região.

Será mais útil se pudermos fazer parsing de XML em um arquivo ou em uma string. Construímos, portanto, uma função para ler XML de um arquivo.

```
(defun xml-parse-file (file-path)
  (interactive "fQue arquivo XML será lido? ")
   (with-temp-buffer
    (insert-file-contents file-path)
    (let ((result (libxml-parse-xml-region 1 (point-max))))
         (print result))))
```

Esta função lê o conteúdo de um arquivo XML e chama `libxml-parse-xml-region` para fazer parsing e retornar uma estrutura Lisp.

Grave, em um arquivo `test.xml`, o seguinte conteúdo:

```
<div id="like">Things I like:<br/>
<ul>
<li><a href="http://nowhere/">Nowhere Land</a>,</li>
<li><a href="http://whatever/">Whatever</a>.</li>
</ul>
</div>
```

Use a função que acabou de escrever:

```
ELISP> (print (xml-parse-file "~/text.xml"))
```

O resultado será a estrutura a seguir.

```
(div
 ((id . "like"))
 "Things I like:"
 (br nil)
 "
"
 (ul nil
     (li nil
  (a
   ((href . "http://nowhere/"))
   "Nowhere Land")
```

```
  ",")
     (li nil
  (a
   ((href . "http://whatever/"))
   "Whatever")
  "."))
 "
")
```

A função `xml-parse-file` pode ser facilmente modificada para ler de uma string ao invés de um arquivo.

Ex. 8 — Mude a função `xml-parse-file`, que desenvolvemos na seção sobre XML, para que faça parsing do conteúdo de uma string, e não de um arquivo.

21.2 Renderização do DOM como Markdown

A função `shr-insert-document` renderiza o DOM obtido por `libxml-parse-xml-region` no buffer atual.

Em um buffer, avalie

```
(shr-insert-document (xml-parse-file "~/text.xml"))
```

Logo após o cursor, o texto a seguir será inserido:

```
Things I like:

* Nowhere Land,
* Whatever.
```

21.3 Geração de XML

Para a geração de XML a partir de estruturas Lisp, há duas opções: criar suas próprias funções ou usar uma biblioteca externa.

Uma biblioteca que gera XML é a `xmlgen`. Após instalar esta biblioteca, é necessário `(require 'xmlgen)` para poder usá-la.

```
(xmlgen '(div nil
              (ul nil
                  (li nil "Item 1")
                  (li nil "Item 2"))))

"<div><nil/><ul><nil/><li><nil/>Item 1</li><li><nil/>Item 2</li></ul></div>"
```

O XML gerado pela biblioteca, no entanto, é incompatível com o lido pelo parser do Emacs.

22 Modos

Um *modo* no Emacs é um conjunto de configurações feitas especificamente para facilitar alguma tarefa. Por exemplo,

- o *modo C* inclui configurações de highlighting de sintaxe para a linguagem C; atalhos de teclado e menus para programar em C; configuração para que comando `comment-region` saiba como comentar a região (usando delimitadores de comentário da linguagem C, `/*` e `*/`);
- o modo `display-line-number-mode` adiciona números de linha ao buffer;
- o jogo `5x5`, que é incluído no Emacs, é implementado como um modo (o `5x5-mode`);
- o `SLIME` é um modo que transforma o Emacs em um ambiente de desenvolvimento para Common Lisp.

Normalmente, em cada momento, o Emacs usa exatamente um *major mode* e zero ou mais *minor modes*. Os *minor modes*, em geral, podem ser usados com qualquer *major mode* – embora isso nem sempre faça sentido.

22.1 Criando minor modes

Construiremos um minor mode para inserir e remover, no buffer atual, lembretes com a cadeia "`FIXME!`".

Usaremos três funções:

- `within-fixme` determina se o cursor (ou, na terminologia do Emacs, o *ponto* está sobre a cadeia "`FIXME!`";
- `fixme-add` insere uma cadeia "`FIXME!`", desde que o ponto já não esteja em uma;
- `fixme-remove` remove uma cadeia "`FIXME!`", se houver uma sob o ponto.

```
(defun within-fixme ()
  (save-excursion
    (goto-char (- (point) 5))
    (search-forward "FIXME!" (+ (point) 12) t)))

(defun fixme-add ()
  (interactive)
  (insert "FIXME!"))

(defun fixme-remove ()
  (interactive)
  (let ((pos (within-fixme)))
    (if (not (null pos))
        (delete-region (- pos 6) pos)
        (message "Não há FIXME aqui para remover!"))))
```

Para testá-las, avalie o código e depois tente chamar `M-x fixme-add`. A cadeia "`FIXME!`" será inserida no buffer, exatamente onde estava o cursor. Da mesma forma, se o cursor estiver sobre a cadeia, chamar `M-x fixme-remove` apagará a cadeia.

```
(define-minor-mode NOME
  docstring
  :lighter <a string a mostrar na modeline>
  :keymap <o mapa de teclado>
  :global <se o modo é global ou não> )
```

- `:lighter` é a cadeia que será mostrada na *modeline* quando o modo estiver ativado – desde que o usuário não tenha modificado sua modeline de forma a não mostrar os minor modes, claro.
- `:keymap` é o mapa de teclado a ser usado.
- `:global` determina se o modo é local (pode ser ativado para buffers isolados) ou global (para todos os buffers). Se um modo é global, o Emacs adiciona, na interface de configuração, a opção de ligá-lo ou desligá-lo (passa a existir uma variável de configuração para o modo).

Este minor mode terá o nome `fixme-mode`. Na versão inicial, o mapa de teclado dele definirá dois *bindings*:

- `"TAB"` (tab) insere a cadeia;
- `"SPC"` (espaço) remove a cadeia.

```
(define-minor-mode fixme-mode
  "Mode for inserting and removing FIXME notes."
  :lighter " FIXME"
  :keymap (let ((map (make-sparse-keymap)))
              (define-key map (kbd "TAB") 'fixme-add)
              (define-key map (kbd "SPC") 'fixme-remove)
              map)
  :global nil)
```

`:keymap` é definido como o valor retornaro por uma expressão `let`. Ali, cria-se um mapa de declado vazio com `make-sparse-keymap`, e depois duas teclas dele são modificadas com `define-key`; depois o mapa é retornado.

- `(make-sparse-keymap)` cria um mapa de teclado vazio, que em seguida será modificado com os *bindings* que queremos.
- `(kbd s)` transforma uma string em uma sequência de teclas.
- `(define-key MAP KEY FUNCTION)` modifica o MAPA, vinculando KEY à função FUNCTION.

Para testar o modo, use `M-x fixme-mode` em algum buffer. A partir de agora, a tecla TAB incluirá a cadeia `FIXME!` onde o cursor estiver, e o espaço removerá a cadeia, se o cursor estiver sobre ela.

Seria interessante se pudéssemos usar uma tecla para desativar o modo `fixme`. Podemos fazer isso criando mais um *keybinding*, para uma função `fixme-disable`, que desativa o

modo. Quando um modo é criado, `define-minor-mode` cria uma função com o nome do modo. Esta função pode ser chamada sem argumentos, "`(fixme-mode)`", para mudar o estado do modo (ativado/desativado), ou com um argumento para ativar ou desativar (0 desativa, 1 ativa). A função `fixme-disable`, portanto, só precisa chamar `(fixme-mode 0)` – mas será interessante também enviar uma mensagem no minibuffer avisando o usuário que o modo foi desativado.

```
(defun fixme-disable ()
  (interactive)
  (fixme-mode 0)
  (message "fixme-mode desativado"))

(define-minor-mode fixme-mode
  "Mode for inserting and removing FIXME notes."
  :lighter " FIXME"
  :keymap (let ((map (make-sparse-keymap)))
              (define-key map (kbd "TAB") 'fixme-add)
              (define-key map (kbd "SPC") 'fixme-remove)
              (define-key map (kbd "q")   'fixme-disable)
              map)
  :global nil)
```

Para tornar o modo um pouco mais interessante, adicionaremos algo mais: uma tecla fará o Emacs mostrar, no minibuffer, a quantidade de cadeias `FIXME!` no buffer.

Criaremos uma variável local (visível somente no buffer corrente), e uma função que usa `count-matches` para contabilizar a quantidade de cadeias.

`count-matches` inicia a busca a partir do ponto atual, por isso faremos `(goto-char 0)` antes. Como não queremos confundir o usuário mudando a posição do cursor, tudo isso em um `save-excursion`.

```
(defvar-local num-fixmes 0
  "Number of FIXME! messages in the current buffer.")

(defun fixme-count ()
  (interactive)
  (save-excursion
   (goto-char 0)
   (setq num-fixmes (count-matches "FIXME!"))
   (message (concat "Ocurrences of FIXME!s: "
                    (number-to-string num-fixmes)))))

(define-minor-mode fixme-mode
  "Mode for inserting and removing FIXME notes."
  :lighter " FIXME"
  :keymap (let ((map (make-sparse-keymap)))
              (define-key map (kbd "TAB") 'fixme-add)
              (define-key map (kbd "SPC") 'fixme-remove)
              (define-key map (kbd "#")   'fixme-count)
```

```
              (define-key map (kbd "q")   'fixme-disable)
              map)
  :global nil)
```

A variável `num-fixes` foi definida usando `defvar-local`, para que tenha valores diferentes em buffers diferentes.

23 Menus

Criaremos duas funções, que exibem uma saudação. Uma delas mostra a mensagem no minibuffer, e a outra insere no buffer corrente.

```
(defun hello-message ()
  (interactive)
  (message "Hello, Wild Weird World!"))


(defun hello-insert ()
  (interactive)
  (insert "Hello, Wild Weird World!"))
```

É fácil testá-las com `M-x hello-insert` e `M-x hello-message`
Para adicionar um menu, primeiro criamos um mapa de teclado vazio.

> Um mapa de teclado funciona como a descrição de um menu, se contém uma string, chamada de *prompt srting*. Assim, para definir uma entrada em um menu, usaremos funções que definem keybindings em mapas de teclado.

A *prompt string* de nosso menu será "Hello". Damos à variável o nome `hello-menu`, e não `hello-keymap`, somente para que fique claro que será usada para descrever as opções do menu.

```
(defvar hello-menu (make-sparse-keymap "Hello"))
```

É possível consultar a *prompt string* de um keymap usando a função `keymap-prompt`.

```
IELM> (keymap-prompt hello-menu)
Hello
```

Agora, a função `define-key-after` insere um menu exatamente após algum outro, na barra de menus do Emacs. Um dos menus que vem pré-configurados é o "Tools", cuja identificação é `'tools`.

```
(define-key-after
  global-map
  [menu-bar hello-menu]
  (cons "Hello" hello-menu)
  'tools)
```

- `global-map` é o mapa (menu) onde modificaremos um *binding*. Aqui parece que estamos selecionado o mapa global, mas o próximo item mostra que não...
- `[menu-bar hello-menu]` é a identificação de nosso menu. Teremos que usá-la quando nos referirmos a ele. *Como este vetor tem mais de um elemento*, a modificação não será feita em `global-map`, mas em `global-map` → `menu-bar` → `hello-menu`.
- `(cons "Hello"hello-menu)` contém o nome do menu, seguido do "mapa de teclado" (na verdade, a descrição das entradas).
- `'tools` é a identificação do menu após o qual incluiremos o nosso.

Depois de avaliar `define-key-after`, como acima, omenu já existirá. No entanto, pode ser necessário usar o mouse (clicar em algum lugar dentro do Emacs) para que ele mostre o menu – que por enquanto estará vazio.

```
(define-key
  global-map
  [menu-bar hello-menu msg]
  '("Message" . hello-message))

(define-key
  global-map
  [menu-bar hello-menu ins]
  '("Insert" . hello-insert))
```

Nestes dois exemplo, usamos `global-map` seguido de `[menu-bar hello-menu ALGO]`. Poderíamos também ter usado, ao invés de `global-map`, nosso menu:

```
(define-key
  hello-menu
  [hello-message]
  '(menu-item "Message" hello-message :help "Hello message"))

(define-key
  hello-menu
  [hello-insert]
  '(menu-item "Insert" hello-insert :help "Insert Hello"))
```

Ex. 9 — Mude o `fixme-mode`:

- para que a mensagem seja customizável por uma variável (`fixme-mode-string`, por exemplo)
- para que use apenas uma tecla para inserir ou remover a cadeia `FIXME!` (se estiver sobre a cadeia, apaga; se não estiver, insere)
- para suportar mais de uma mensagem (por exemplo, fazendo a função `fixme-add` mostrar um menu)
- mude o item anterior de forma que os primeiros itens do menu sejam os mais usados (voce terá que guardar a frequência de cada string inserida)
- para funcionar em modos de programação: o `FIXME!` é inserido como comentário (dica: selecione, em Emacs Lisp, a região do `FIXME!` e chame `comment-region`)

24 Mais sobre Emacs Lisp

Este tutorial não cobre diversos pontos interessantes do Emacs Lisp: expressões regulares, major modes, 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. Este é, no entanto, um projeto difícil – e o software já escrito em Emacs Lisp (como o SLIME) dificilmente seria portado para Scheme. De qualquer forma, a idéia será permitir executar Emacs Lisp sobre Scheme – e portanto o aprendizado de Emacs Lisp continua sendo válido por muito tempo.

Sobre este texto

Este tutorial foi preparado em LaTeX, tendo sido digitado em Emacs[11] (!) com o modo AUCTeX (https://www.gnu.org/software/auctex/, https://www.emacswiki.org/emacs/AUCTeX). Os diagramas foram produzidos com a ferramenta ditaa (http://ditaa.sourceforge.net/). Alguns dos pacotes LaTeXusados são `scrartcl`, `listings`, `exercise` e `graphicx`.

Exercícios

Ex. 10 — 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. 11 — 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. 12 — Faça diagramas iguais àqueles na seção que discorre sobre listas, representando as seguintes estruturas:

[11] Inicialmente Emacs versão 24; nos estágios finais, checkouts frequentes do repositório git do Emacs foram usados.

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. 13 — 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. 14 — 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 topo do *buffer* o texto resumido de uma licença (GNU GPL, MIT, ou o que você preferir);

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

f) Inserir no ponto onde estiver o cursor, um comentário, devidamente delimitado de acordo com a linguagem (`;;` para Lisps, `/* */` para C, `#` para Python, etc) com sua assinatura e data.

g) Inserir um comentário ou marca `TODO`, ou `FIXME`, na linha em branco imediatamente antes do trecho onde está o cursor.

h) Percorrer o buffer e eliminar espaços horizontais desnecessários: quando houver duas ou mais linhas em branco consecutivas, deixe apenas uma.

i) 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 argumentos não usados, como o `y` e o `z` neste caso).

j) Codificar e decodificar o *buffer* atual usando base 64.

k) Encriptar o *buffer* atual usando:

- A cifra de César;
- A cifra de Vigenère;
- Uma cifra de bloco comum (por exemplo, AES ou *twofish*). Neste caso, use um programa externo – mande o *buffer* para o programa, leia de volta, e faça codificação em base 64.

Ex. 15 — Construa um ambiente de desenvolvimento C para o Emacs usando o modo C que já existe no Emacs, mas usando algum *interpretador*[12] C. O ambiente deve funcionar de maneira semelhante ao Geiser.
Examine e descreva as diferenças entre seu ambiente para C e o Geiser para Scheme (até que ponto eles podem ter as mesmas funcionalidades, e por que motivos?)

Ex. 16 — Descubra algum jogo que ainda não tenha sido escrito em Emacs Lisp e o desenvolva.

[12] Há diversos deles!