3.2.19. Iteração

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

```
> (setq a 4)
4
> (loop
    (setq a (+ a 1))
    (when (> a 7) (return a))
   )
8
> (loop
    (setq a (- a 1))
    (when (< a 3) (return))
   )
NIL
```

O próximo mais simples é a `dolist`: Uma `dolist` ata uma variável aos elementos de uma lista na sua ordem e termina quando encontra o fim da lista:

```
> (dolist (x '(a b c)) (print x))
A
B
C
NIL
```

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

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

```
> (do ((x 1 (+ x 1))  ;variável x, com valor inicial 1
       (y 1 (* y 2))  ;variável y, com valor inicial 1
      )
      ((> x 5) y)     ;retorna valor de y quando x > 5
     (print y)        ;corpo
     (print 'working) ;corpo
  )
1
WORKING
2
WORKING
4
WORKING
8
WORKING
16
WORKING
32
```

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

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

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

3.2.20. Sáidas Não-Locais

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

```
> (defun foo (x)
    (return-from foo 3)
    x
  )
FOO
> (foo 17)
3
```

Na verdade, o `form` `return-from` pode retornar de qualquer bloco nomeado.

Na verdade, funções são os únicos blocos nomeados por default. Você pode criar um bloco nomeado com o `form` especial `block`:

```
> (block foo
    (return-from foo 7)
    3
  )
7
```

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

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

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

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

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

3.2.21. Funcall, Apply, e Mapcar

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

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

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

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

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

3.2.21.1. Utilidade de `Funcall`, `apply` e `Mapcar`

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

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

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

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

3.2.22. Lambda

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

```
> #'(lambda (x) (+ x 3))
(LAMBDA (X) (+ X 3))
> (funcall * 5)   ;observe que neste contexto o * é uma
                  ;variável que representa o último form
                  ;que foi digitado...
8
```

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

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

3.2.23. Sorting - Ordenação

LISP provê duas primitivas para ordenação: `sort` e `stable-sort`.

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

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

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

Seja cuidadoso: sort tem permissão para destruir o seu argumento. Por isso, se você deseja manter a lista original, copie-a com `copy-list` ou `copy-seq`.

3.2.24. Igualdade

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

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

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

```
> (eq 'a 'a)
T
> (eq 'a 'b)
NIL
> (= 3 4)
T
> (eq '(a b c) '(a b c))
NIL
> (equal '(a b c) '(a b c))
T
> (eql 'a 'a)
T
> (eql 3 3)
T
```

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

O predicado `equal` é equivalente a `eql` para símbolos e números.

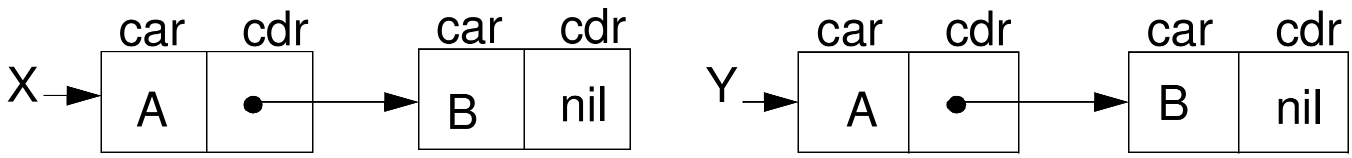
Ele é verdadeiro para dois conses, se e somente se, seus cars são `equal` e seus cdrs são `equal`.

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

3.2.24.1.Exemplos de Fixação: Igualdade e Identidade

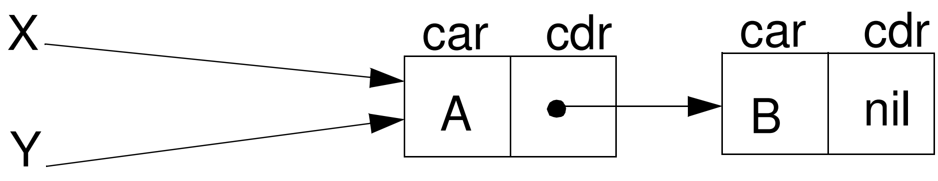
Observe:

```
> (setq X ´(A B))
(A B)
> (setq Y ´(A B))
(A B)
> (equal X Y)      ;X e Y são iguais
T
> (eq X Y)         ;X e Y não são idênticos
NIL
```



X e Y apontam para estruturas diferentes

```
> (setq X ´(A B))
(A B)
> (setq Y X)
(A B)
> (equal X Y)      ;X e Y são iguais
T
> (eq X Y)         ;X e Y agora são idênticos
T
```



X e Y apontam para exatamente a mesma estrutura

3.2.25. Algumas Funções de Lista Úteis

Todas as funções abaixo manipulam listas:

```
> (append '(1 2 3) '(4 5 6))      ;concatena listas
(1 2 3 4 5 6)

> (reverse '(1 2 3))              ;reverte os elementos
(3 2 1)

> (member 'a '(b d a c))          ;pertinência a conjunto
                                  ;retorna primeira cauda
(A C)                             ;cujo car é o elemento
                                  ;desejado

> (find 'a '(b d a c))            ;outro set membership
A

> (find '(a b) '((a d) (a d e) (a b d e) ()) :test
#'subsetp)
(A B D E)                         ;find é mais flexível

> (subsetp '(a b) '(a d e))       ;set containment
NIL
> (intersection '(a b c) '(b))    ;set intersection
(B)
> (union '(a) '(b))               ;set union
(A B)
> (set-difference '(a b) '(a))    ;diferença de conjuntos
(B)
```

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

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

3.2.26. Utilizando Emacs/X-Emacs para programar LISP

Você pode usar o Emacs ou X-Emacs para editar código LISP: a maioria dos Emacses colocam-se automaticamente em modo-LISP quando você carrega um arquivo que termina em `.lisp`, mas se o seu não o faz, você pode digitar `M-x lisp-mode` (ALT-`x lisp-mode` ou ✧-`x lisp-mode`).

Você pode rodar LISP sob Emacs tambèm, usando-o como um ambiente de programação: certifique-se de que há um comando chamado "`lisp`" que roda seu LISP favorito. Por exemplo, você poderia digitar:

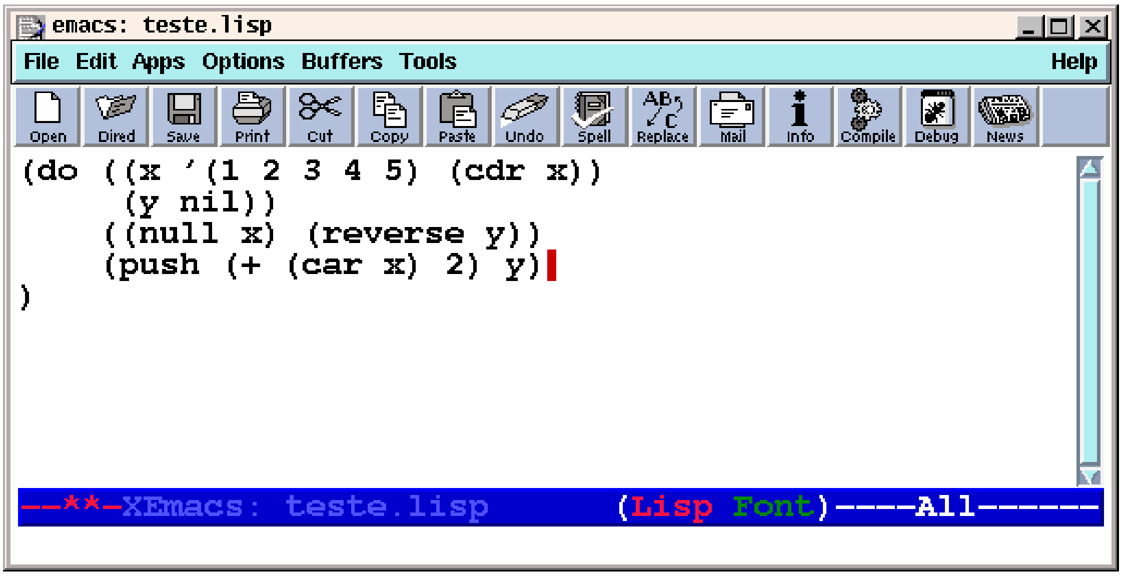
```
ln -s /usr/local/bin/clisp ~/bin/lisp
```

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

Na verdade, você nem precisa fazer o atalho (simblic link) acima. Emacs possui uma variável chamada inferior-lisp-program; assim você só precisa adicionar a linha:

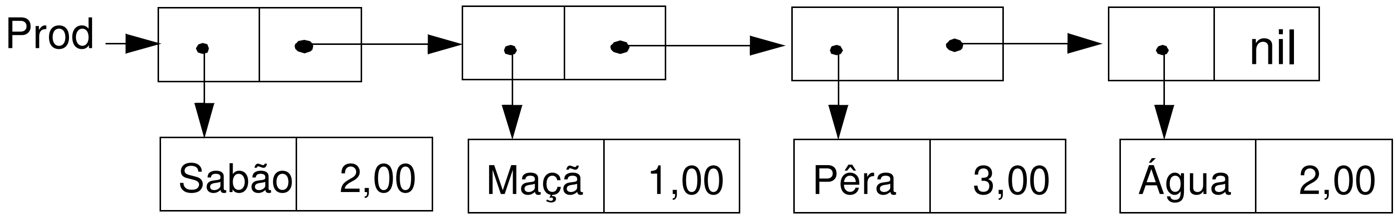
```
(setq inferior-lisp-program "/usr/local/bin/clisp")
```

ao seu arquivo `.emacs`, Emacs vai saber encontrar CLISP quando você digita `M-x run-lisp`.



3.2.27. Lista de Exercícios Nº 4:

1. Escreva uma função que leia do usuário uma lista de produtos e seus respectivos preços, colocando-os em uma lista organizada por pares produto-preço. A entrada de dados é finalizada digitando-se a palavra ‘fim ao invés de um nome de produto.
   Utilize o comando `loop` para implementar o laço de leitura e defina uma variável global onde a lista ficará armazenada ao fim da leitura.



   Os pares produto-preço você pode organizar tanto como um cons, uma sublista ou uma estrutura com campos produto e preço. A `list` tem a vantagem de ser extremamente flexível: você pode extender a sua estrutura de dados sem necessitar entrar com os dados de novo. O `cons` é a forma mais econômica em termos de memória. A estrutura permite uma modelagem elegante. Fica a seu critério.
2. Escreva uma função ou conjunto de funções, que, através de um **menu de opções**, realizem as seguintes tarefas:
   a) **Pesquisar preço** de um produto: Um ambiente onde o usuário entra com o nome de um produto e o programa ou diz que não encontrou o produto ou devolve o preço.
   b) **Mostrar em ordem alfabética** toda a lista de produtos disponíveis com os respectivos preços, formatada na tela. A cada 20 produtos o programa deve fazer uma pausa e esperar o usuário teclar alguma coisa para continuar.
   c) **Fazer compras**: Um ambiente onde o usuário pode entrar com nomes de produtos e quantidades que deseja comprar. Ao final o programa emite uma lista com todos os produtos comprados, total parcial e total final das compras.