Linguagem LISP

1. Aspectos Gerais

O que é LISP?

LISP é uma linguagem de programação funcional. Foi inventado por J. McCarthy em 1959.

Houve épocas em que havia muitos dialetos de LISP.
Hoje em dia LISP está estandardizado no padrão COMMON LISP.

Há aplicações de LISP nos domínios do
processamento simbólico e de conhecimento (IA),
processamento numérico (MACLISP),
e na confecção de programas muito difundidos como editores (EMACS) e CAD (AUTOCAD).

O texto de referência padrão da linguagem é: Guy L. Steele Jr.: Common Lisp - The Language. Digital Press.
1. edition 1984, 465 pages.
2. edition 1990, 1032 pages.

Este livro está disponível em HTML via FTP de:

ftp.cs.cmu.edu:/user/ai/lang/lisp/doc/cltl/cltl_ht.tgz and

http://www.cs.cmu.edu:8001/Web/Groups/AI/html/cltl/cltl2.html

http://www.cs.cmu.edu:8001/afs/cs/project/ai-repository/ai/html/cltl/cltl2.html

LISP é rodado em um ambiente interativo.

Você entra com "forms" - conjuntos de expressões - e elas são avaliadas de uma só vez.

Você também pode inspecionar variáveis, chamar funções com argumentos dados e definir suas pr'oprias funções.

No decorrer desta disciplina, nos vamos utilizar o CLISP, uma implementação de COMMON LISP.

CLISP está quase completamente de acordo (99%) com a definição de COMMON LISP e inclui ainda CLOS, um dialeto LISP orientado a objetos.

Common Lisp é

- uma linguagem de programação funcional com usos convencionais e uma linguagem para Inteligência Artificial
- interativo

Programas em Common Lisp são altamente portáveis entre máquinas e sistemas operacionais (there is a standard for the language and the library functions)

Common Lisp provê

- sintaxe clara
- muitos tipos de dados: numbers, strings, arrays, lists, characters, symbols, structures, streams etc.
- tipagem em tempo de execução: o programador geralmente não precisa se preocupar com declarações de tipo, mas ele recebe mensagens de erro caso haja violações de tipo (operações ilegais)

.

- Funções genéricas: 88 arithmetic functions for all kinds of numbers (integers, ratios, floating point numbers, complex numbers), 44 search/filter/sort functions para listas, arrays e strings
- gerenciamento de memória automático (garbage collection)
- pacoteamento (packaging) de programas em módulos
- um sistema de objetos, funções genéricas com a possibilidade de combinação de métodos.
- macros: todo programador pode realizar suas próprias extensões da linguagem

A implementação de Common Lisp CLISP provê:

- um interpretador.
- um compilador para executáveis até 5 vezes mais rápidos.
- todos os tipos de dados com tamanho ilimitado (a precisão e o tamanho de uma variável não necessita de ser declarado, o tamanho de listas e arrays altera-se dinamicamente)
- integers de precisão arbitrária, precisão de ponto flutuante ilimitada.

2. Tutorial da Linguagem LISP

Baseado em:

Common LISP Hints

Geoffrey J. Gordon

<ggordon@cs.cmu.edu>

Friday, February 5, 1993

2.1. Symbols

Um símbolo é somente um string de caracteres.
Alguns exemplos:

```
a

b

c1

foo

bar
```

`baaz-quux-garply` Algumas operações com símbolos seguem abaixo. Coisas após o prompt ">" são as que você digita para o interpretador lisp. Tudo após um ";" é um comentário.

```
> (setq a 5) ;store a number as the value of a symbol

5

> a ;take the value of a symbol

5

> (let ((a 6)) a) ;bind the value of a symbol temporarily to 6

6

> a ;the value returns to 5 once the

;let is finished
```

```
5

> (+ a 6) ;use the value of a symbol as an argument to a function

11

> b ;try to take the value of a symbol which has no value

Error: Attempt to take the value of the unbound symbol B
```

Há dois símbolos especiais, t e nil. O valor de t é definido sempre como sendo t. nil é definido como sendo sempre nil.

LISP utiliza t e nil para representar verdadeiro e falso. Exemplo no IF:

```
> (if t 5 6)

5

> (if nil 5 6)

6

> (if 4 5 6)

5
```

O último exemplo é estranho, mas está correto. nil significa falso e qualquer outra coisa verdadeiro. Usamos t somente para clareza.

Símbolos como nil e t são chamados símbolos auto-avaliantes, porque avaliam para si mesmos.

Há toda uma classe de símbolos auto-avaliantes chamados palavras-chave. Qualquer símbolo cujo nome inicia com dois pontos é uma palavra-chave. Exemplos:

```
> :this-is-a-keyword

:THIS-IS-A-KEYWORD

> :so-is-this

:SO-IS-THIS

> :me-too

:ME-TOO
```

2.2. Números

Um inteiro é um string de dígitos opcionalmente precedido de um + ou -.

Um real parece com um inteiro, só que possui um ponto decimal e pode opcinalmente ser eescrito em notação científica.

Um racional se parece com dois inteiros com um / entre eles.

LISP suporta números complexos que são escritos `#c(r i)`

Exemplos:

```
17

-34

+6

3.1415

1.722e-15

#c(1.722e-15 0.75)
```

As funções aritméticas padrão são todas avaliáveis: `+, -, *, /, floor, ceiling, mod, sin, cos,`

`tan, sqrt, exp, expt` **etc**

Todas elas aceitam qualquer número como argumento: `> (+ 3 3/4) ;type contagion`

```
15/4
```

```
> (exp 1) ;e
```

```
2.7182817
```

```
> (exp 3) ;e*e*e
```

```
20.085537
```

```
> (expt 3 4.2) ;exponent with a base other than e
```

```
100.90418
```

`> (+ 5 6 7 (* 8 9 10)) ;the fns +-*/ all accept multiple arguments` Não existe limite para o valor absoluto de um inteiro exceto a memória do computador. Evidentemente cálculos com inteiros ou racionais imensos podem ser muito lentos.

2.3. Conses - Associações

Um `cons` é somente um registro de dois campos. Os campos são chamados de "car" e "cdr" por razões históricas: na primeira máquina onde LISP foi implementado havia duas instruções assembler CAR e CDR "contents of address register" e "contents of decrement register".

Conses foram implementados utilizando-se esses dois registradores:

```
> (cons 4 5) ;Allocate a cons. Set the car to 4 and the cdr to 5.
```

```
(4 . 5)
```

```
> (cons (cons 4 5) 6)
```

```
((4 . 5) . 6)
```

```
> (car (cons 4 5))
```

```
4
```

```
> (cdr (cons 4 5))
```

```
5
```

2.4. Listas

Você pode construir muitas estruturas de dados a partir de conses. A mais simples com certeza é a lista encadeada:

- o car de cada cons aponta para um dos elementos da lista e
- o cdr aponta ou para outro cons ou para nil.

Uma lista assim pode ser criada com a função de lista:

```
> (list 4 5 6)
```

`(4 5 6)` Observe que LISP imprime listas de uma forma especial: ele omite alguns dos pontos e parênteses.

A regra é: se o cdr de um cons é nil, lisp não se preocupa em imprimir o ponto ou o nil. Se o cdr de cons A é cons B, então lisp não se preocupa em imprimir o ponto para A nem o parênteses para B:

```
> (cons 4 nil)
```

```
(4)

> (cons 4 (cons 5 6))

(4 5 . 6)

> (cons 4 (cons 5 (cons 6 nil)))
```

`(4 5 6)` O último exemplo corresponde ao `(list 4 5 6)` anterior. Note que nil corresponde à **lista vazia**.

- O car e cdr de nil são definidos como nil.
- O car de um átomo é o próprio átomo.
- O cdr de um átomo é nil.

Se você armazena uma lista em uma variável, pode fazê-la funcionar como uma pilha: `> (setq a nil)`

```
NIL

> (push 4 a)

(4)

> (push 5 a)

(5 4)

> (pop a)

5

> a

(4)

> (pop a)

4

> (pop a)

NIL

> a

NIL
```

2.5. Exercícios

2.5.1. Desenhe as representações internas de dados para as listas seguintes:
(A 17 -3)
((A 5 C) %)
((A 5 C) (%))
(NIL 6 A)
((A B))
(* (+ 15 (- 6 4)) -3)

2.5.2. Qual é o CAR de cada uma das listas do exercício anterior?

2.5.3. Qual é o CDR de cada uma das listas do exercício anterior 1?

2.5.4. Escreva as declarações necessárias, usando CAR e CDR, para obter os valores seguintes das listas do exercício 1:

(-3)

(-3 -3)

(C %)

(A C %)

(5 %)

(5 (%))

(6 (6))

(6 (6) 6)

((B) A)

(A ((B) B))

2.5.5. Defina uma representação conveniente na forma de lista para um conjunto de sobrenomes juntamente com os números de telefones de pessoas. O número de telefone deve permitir a inclusão de códigos de DDD e DDI para números não locais. Como resolveria o caso para pesso qu na lista, mas não tivessem telefone ?

CLISP para PC (DOS, Windows NT, Windows 95 e Windows 3.11)

2.6. Funções

Vimos exemplos de funções acima. Aqui mais alguns:

```
> (+ 3 4 5 6)                   ;this function takes any
                        ;number of arguments
18
> (+ (+ 3 4) (+ (+ 4 5) 6)) ;isn't prefix notation fun?
22
```

2.6.1. Definindo uma função:

```
> (defun foo (x y) (+ x y 5))
FOO
> (foo 5 0)                     ;chamando a função
10
```

2.6.2. Função Recursiva

```
> (defun fact (x)
    (if (> x 0)
      (* x (fact (- x 1)))
      1
  ) )

FACT
> (fact 5)
120
```

2.6.3. Funções Mutuamente Recursivas

```
> (defun a (x) (if (= x 0) t (b (- x))))
A
> (defun b (x) (if (> x 0) (a (- x 1)) (a (+ x 1))))
B
> (a 5)
```

```
T
```

2.6.4. Função com múltiplos comandos em seu corpo

```
> (defun bar (x)
    (setq x (* x 3))
    (setq x (/ x 2))
    (+ x 4)
  )
BAR
> (bar 6)
13
```

- O valor retornado, como em um método em Smalltalk, é sempre o valor da última expressão executada.

2.6.5. Escopo de variáveis

Quando nós definimos `foo`, nós lhe atribuímos dois argumentos, x e y. Quando chamamos `foo`, necessitamos prover valores para esses dois argumentos. O primeiro será o valor de x durante a duração da chamada a `foo`, o segundo o valor de y durante a duração da chamada a `foo`. Em LISP, a maioria das variáveis são escopadas lexicamente. Isto significa que se `foo` chama `bar` e `bar` tenta referenciar x, bar não obterá os valor de x de `foo`.

```
> (defun foo (x y) (+ x y 5))
FOO
```

O processo de se associar um valor a um símbolo durante um certo escopo léxico é chamado em LISP de ateamento. x estava atado ao escopo de `foo`.

2.6.6. Número Variável de Argumentos para Funções

Você pode especificar também argumentos opcionais para funções. Qualquer argumento após o símbolo `&optional` é opcional:

```
> (defun bar (x &optional y) (if y x 0))
BAR
```

É perfeitamente legal chamar a função BAR com um ou dois argumentos. Se for chamada com um argumento, x será atado ao valor deste argumento e y será atado a NIL.

```
> (bar 5)
0
> (bar 5 t)
5
```

Se for chamada com dois argumentos, x e y serão atados aos valores do primeiro e segundo argumento respectivamente.

A função BAAZ possui dois argumentos opcionais, porém especifica um valor default para cada um deles.

```
> (defun baaz (&optional (x 3) (z 10)) (+ x z))
BAAZ
> (baaz 5)
15
> (baaz 5 6)
11
> (baaz)
13
```

Se quem a chama especificar somente um argumento, z será atado a 10 ao invés de NIL. Se nenhum argumento for especificado, x será atado a 3 e z a 10.

2.6.7. Número Indefinido de Parâmetros

Você pode fazer a sua função aceitar um número indefinido de parâmetros terminando a sua lista de parâmetros com o parâmetro `&rest`. `LISP` vai coletar todos os argumentos que não sejam contabilizados para algum argumento formal em uma lista a atá-la ao parâmetro `&rest` :

```
> (defun foo (x &rest y) y)
FOO
> (foo 3)
NIL
> (foo 4 5 6)
(5 6)
```

2.6.8. Passagem de Parâmetros por Nome

Existe ainda um tipo de parâmetro opcional chamado de parâmetro de palavra-chave. São parâmtros que quem chama pode passar em qualquer ordem, pois os valores são passados precedidos pelo nome do parâmetro formal a que se referem:

```
> (defun foo (&key x y) (cons x y))
FOO

> (foo :x 5 :y 3)
(5 . 3)

> (foo :y 3 :x 5)
(5 . 3)

> (foo :y 3)
(NIL . 3)

> (foo)
(NIL)
```

Um parâmetro `&key` pode ter um valor default também:

```
> (defun foo (&key (x 5)) x)
FOO

> (foo :x 7)
7

> (foo)
5
```

2.7. Impressão

Algumas funções podem provocar uma sáida. A mais simples é print, que imprime o seu argumento e então o retorna.

```
> (print 3)
3
3
```

O primeiro 3 acima foi impresso, o segundo retornado.

Se você deseja um output mais complexo, você necessita utilizar format:

```
>(format t "An atom: ~S~%and a list: ~S~%and an integer:~D~%"
          nil (list 5) 6)
An atom: NIL
and a list: (5)
and an integer: 6
```

O primeiro argumento a format é ou t, ou NIL ou um arquivo.

- T especifica que a saída deve ser dirigida para o terminal,
- NIL especifica que não deve ser impresso nada, mas que fermat deve retornar um string com o conteùdo ao invés,
- uma referência a um arquivo especifica o arquivo para onde a saída vai ser redirigida.

O segundo argumento é um template de formatação, o qual é um string contendo opcionalmente diretivas de formatação, de forma similar à Linguagem "C": `"An atom: ~S~%and a list: ~S~%and an integer:~D~%"`

Todos os argumentos restantes devem ser referenciados a partir do string de formatação.

- As diretivas de formatação do string serão repostas por LISP por caracteres apropriados com base nos valores dos outros parâmetros a que eles se referem e então imprimir o string resultante.
- Format sempre retorna NIL, a não ser que seu primeiro argumento seja NIL, caso em que não imprime nada e retorna o string resultante.

No exemplo acima, há três diretivas de formatação: `~S, ~D` e `~%.`:

- A primeira,`~S`, aceita qualquer objeto LISP e é substituída por uma representação passível de ser impressa deste objeto (a mesma produzida por print).
- A segunda, `~D`, só aceita inteiros.
- A terceira, `~%`, não acita nada. Sempre é reposta por uma quebra de linha.

- Outra diretiva útil é ~~, que é substituída por um simples ~.

- Veja no manual do LISP muitas, muitas outras diretivas de formatação...

2.8. Forms e o Laço Top-Level

As coisas que você digita para o interpretador LISP são chamadas forms. O interpretador repetidamente lê um form, o avalia e imprime o resultado.

- Este procedimento é chamado read-eval-print loop.

Alguns forms vão provocar erros. Após um erro, LISP vai pô-lo/la no ambiente do debugger.

Debuggers de interpretadores LISP são muito diferentes entre si. A maioria aceita um "help" ou ":help" para ajudá-lo/la na debugação.

No debugador do CLISP você pode sair dando um Control-Z (em DOS) ou Control-D (em Unix).

Em geral, um form é ou um átomo (p.ex.: um símbolo, um inteiro ou um string) ou uma lista.

Se o form for um átomo, LISP o avalia imediatamente. Símbolos avaliam para seu valor, inteiros e strings avaliam para si mesmos.

Se o form for uma lista, LISP trata o seu primeiro elemento como o nome da função, avaliando os elementos restantes de forma recursiva. Então chama a função com com os valores dos elementos restantes como argumentos.

Por exemplo, se LISP vê o form (+ 3 4):

Irá tratar + como o nome da função.

Ele então avaliará 3 para obter 3, avaliará 4 para obter 4 e finalmente chamará + com 3 e 4 como argumentos.

A função + retornará 7, que LISP então imprime.

O top-level loop provê algumas outras conveniências.

Uma particularmente interessante é a habilidade de falar a respeito dos resultados de forms previamente digitados: LISP sempre salva os seus três resultados mais recentes. Ele os armazena sob os símbolos *, ** e ***.

```
> 3
3
> 4
4
> 5
5
> ***
3
> ***
4
> ***
5
> **
4
> *
4
```

2.9. Forms especiais

Há um número de forms especiais que se parecem com chamadas a funções mas não o são. Um form muito útil é o form aspas. As aspes prevêm um argumento de ser avaliado.

```
> (setq a 3)
3
> a
3
> (quote a)
A
> 'a                    ;'a is an abbreviation for (quote a)
A
```

Outro form especial similar é o form function.

Function provoca que seu argumento seja interpretado como uma função ao invés de ser avaliado:

```
> (setq + 3)
3
> +
3
> '+
+
> (function +)
#<Function + @ #x-fbef9de>
> #'+                   ;#'+ is an abbreviation for (function +)
#<Function + @ #x-fbef9de>
```

O form especial function é útil quando você deseja passar uma função como parâmetro para outra função. Mais tarde apresentaremos alguns exemplos de funções que aceitam outras funções como parâmetros.

2.10. Binding - Atamento/Amarração

Binding é uma atribuição escopada lexicamente. Ela ocorre com as variáveis de uma lista de parâmetros de uma função sempre que a função é chamada: os parâmetros formais são atados aos parâmetros reais pela duração da chamada à função.

Você pode também amarrar variáveis em qualquer parte de um programa com o form especial let:

```
(let ((var1 val1)
      (var2 val2)
      ...
     )
  body
)
```

Let ata var1 a val1, var2 a val2, e assim por diante; então executa os comandos de seu corpo. O corpo de um let segue as mesmas regras de um corpo de função:

```
> (let ((a 3)) (+ a 1))
4

> (let ((a 2)
        (b 3)
        (c 0))
    (setq c (+ a b))
    c
  )
5

> (setq c 4)
4

> (let ((c 5)) c)
5

> c
4
```

A invés de `(let ((a nil) (b nil)) ...)` você pode escrever (let (a b) ...).

Os valores val1, val2, etc. dentro de um let não podem referenciar as variáveis var1, var2, etc. que o let está atando:

```
> (let ((x 1)
        (y (+ x 1)))
    y
  )
Error: Attempt to take the value of the unbound symbol X
```

Se o símbolo x já possui um valor, coisas estranhas podem acontecer:

```
> (setq x 7)
7
> (let ((x 1)
        (y (+ x 1)))
    y
  )
8
```

O form especial let* é semelhante, só que permite que sejam referenciadas variáveis definidas anteriormente:

```
> (setq x 7)
7
> (let* ((x 1)
         (y (+ x 1)))
    y
  )
2
```

O form

```
        (let* ((x a)
               (y b))
          ...
        )
```

é equivalente a:

```
        (let ((x a))
          (let ((y b))
            ...
        ) )
```

2.11. Dynamic Scoping - Escopo Dinâmico

Os forms let e let* provêem escop léxico, que é o que você está acostumado quando programa em "C" ou PASCAL.

Escopo dinâmico é o que você tem em BASIC: se você atribui um valor a uma variável escopada dinamicamente, TODA menção desta variável vai retornar aquele valor até que você atribua outro valor à mesma variável.

Em LISP variáveis dinamicamente escopadas são variáveis especiais. Você pode criar uma variável especial através do form defvar.

Abaixo seguem alguns exemplos de variáveis escopadas lexicamente e dinamicamente.

1. Neste exemplo a função check-regular referencia uma variável regular (ie, lexicamente escopada). Como check-regular está lexicamente fora do let que ata regular, check-regular retorna o valor global da variável:

```
> (setq regular 5)
5

> (defun check-regular () regular)
CHECK-REGULAR
> (check-regular)
5

> (let ((regular 6)) (check-regular))
5
```

1. Neste exemplo, a função check-special referencia uma variável especial (ie, escopada dinamicamente). Uma vez que a chamada a check-special é temporariamente dentro do let que amarra special, check-special retorna o valor local da variável:

> (defvar *special* 5) *SPECIAL* > (defun check-special () *special*) CHECK-SPECIAL > (check-special) 5 > (let ((*special* 6)) (check-special)) 6 Por uma questão de convenção, o nome de uma variável especial começa e termina com *.

Variáveis especial são principalmente usadas como variáveis globais.

2.12. Arrays

A função make-array faz um array. A função aref acessa seus elementos. Todos os elementos de um array são inicialmente setados para nil:

```
> (make-array '(3 3))
#2a((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))

> (aref * 1 1)
NIL
> (make-array 4) ;1D arrays don't need the extra parens
```

```
#(NIL NIL NIL NIL)
```

Índices de um array sempre começam em 0

Veja abaixo como setar os elementos de um array.

2.13. Strings

Um string é uma seqüência de caracteres entre aspas duplas. LISP representa um string como um array de tamanho variável de caracteres.

Você pode escrever um string que contém a aspa dupla, precedendo a de um backslash \ . Um backslash duplo \\ está para um \ :

```
        "abcd" has 4 characters
        "\"" has 1 character
        "\\" has 1 character
```

Algumas funções para manipulação de strings:

```
> (concatenate 'string "abcd" "efg")
"abcdefg"
> (char "abc" 1)
#\b             ;LISP writes characters preceded by #\
> (aref "abc" 1)
#\b             ;remember, strings are really arrays
```

A função concatenate pode trabalhar sobre qquer tipo de seqüência:

```
> (concatenate 'string '(#\a #\b) '(#\c))
"abc"
> (concatenate 'list "abc" "de")
(#\a #\b #\c #\d #\e)
> (concatenate 'vector '#(3 3 3) '#(3 3 3))
#(3 3 3 3 3 3)
```

2.14. Estruturas

Estruturas LISP são análogas a structs em "C" ou records em PASCAL:

```
> (defstruct foo
    bar
    baaz
    quux
  )
FOO
```

Este exemplo define um tipo de dado chamado FOO contendo 3 campos.

Define também 4 funções que operam neste tipo de dado:

```
make-foo, foo-bar, foo-baaz, and foo-quux.
```

A primeira cria um novo objeto do tipo FOO.

As outras acessam os campos de um objeto do tipo FOO.

```
> (make-foo)
#s(FOO :BAR NIL :BAAZ NIL :QUUX NIL)
> (make-foo :baaz 3)
#s(FOO :BAR NIL :BAAZ 3 :QUUX NIL)
> (foo-bar *)
NIL
> (foo-baaz **)
3
```

A função `make-foo` pode tomar um argumento para cada um dos campos da estrutura do tipo. As funções de acesso a campo tomam cada uma um argumento.

Veja abaixo como setar os campos de uma estrutura.

2.15. Setf

Alguns forms em LISP naturalmente definem uma locação na memória.

Por exemplo, se x é uma estrutura do tipo FOO, então (foo-bar x) define o campo BAR do valor de x.

Ou, se o valor de y é um array unidimensional, então (aref y 2) define o terceiro elemento de y.

O form especial setf usa seu primeiro argumento para definir um lugar na memória, avalia o seu segundo argumento e armazena o valor resultante na locação de memória resultante:

```
> (setq a (make-array 3))
#(NIL NIL NIL)
> (aref a 1)
NIL
> (setf (aref a 1) 3)
3
> a
#(NIL 3 NIL)
> (aref a 1)
3
> (defstruct foo bar)
FOO
> (setq a (make-foo))
#s(FOO :BAR NIL)
> (foo-bar a)
NIL
> (setf (foo-bar a) 3)
3
> a
#s(FOO :BAR 3)
> (foo-bar a)
3
```

Setf é a única maneira de se setar os valores de um array ou os campos de uma estrutura.

Alguns exemplos de setf e funções relacionadas:

```
> (setf a (make-array 1))       ;setf on a variable is equivalent to setq
#(NIL)
> (push 5 (aref a 1))           ;push can act like setf
(5)
> (pop (aref a 1))              ;so can pop
5
> (setf (aref a 1) 5)
5
> (incf (aref a 1))             ;incf reads from a place, increments,
6                               ;and writes back
> (aref a 1)
6
```

2.16. Booleanos e Condicionais

LISP usa o símbolo auto-avaliante NIL para significar FALSO. Qualquer outra coisa significa VERDADEIRO.

Nós usualmente utilizaremos o símbolo auto-avaliante t para significar TRUE.

LISP provê uma série de funções booleanas-padrão, como and, or e not. Os conetivos and e or são curto-circuitantes. AND não vai avaliar quaisquer argumentos à direita daquele que faz a função avaliar para NIL, enquanto OR não avalia nenhum à direita do primeiro verdadeiro.

LISP também provê uma série de forms para execução condicional. O mais simples é o IF, onde o primeiro argumento determina se o segundo ou o terceiro será avaliado.

Exemplos:

```
> (if t 5 6)
5
> (if nil 5 6)
6
> (if 4 5 6)
5
```

Se você necessita colocar mais de um comando em uma das cláusulas, então use o form progn. Progn executa cada comando em seu corpo e retorna o valor do último.

```
> (setq a 7)
7
> (setq b 0)
0
> (setq c 5)
5
> (if (> a 5)
    (progn
      (setq a (+ b 7))
      (setq b (+ c 8)))
    (setq b 4)
  )
13
```

Um if statement que não possui uma clausula then ou uma cláusula else pode ser escrito utilizando-se when ou unless:

```
> (when t 3)
3
> (when nil 3)
NIL
> (unless t 3)
NIL
> (unless nil 3)
3
```

When e unless, ao contrário de if, aceitam qquer número de comandos em seus corpos:

```
(when x a b c) é equivalente a(if x (progn a b c)).
```

```
> (when t
    (setq a 5)
    (+ a 6)
  )
11
```

Condicionais mais complexos podem ser construídos através do form cond, que é equivalente a `if ... else if ... fi`.

Um cond consiste de símbolo con seguido por um número de cláusulas-cond, cada qual é uma lista. O primeiro elemento de uma cláusula-cond é a condição, os lementos restantes são a ação.

O cond encontra a primeira cláusula que avalia para true, executando a ação respectiva e retornando o valor

resultante. Nenhuma das restantes é avaliada.

```
> (setq a 3)
3
> (cond
   ((evenp a) a)                ;if a is even return a
   ((> a 7) (/ a 2));else if a is bigger than 7 return a/2
   ((< a 5) (- a 1));else if a is smaller than 5 return a-1
   (t 17)               ;else return 17
  )
2
Se não há nenhuma ação na cláusula cond selecionada, cond retorna o valor verdadeiro:

> (cond ((+ 3 4)))
7
```

O comando LISP case é semelhante a um "C" switch statement:

```
> (setq x 'b)
B
> (case x
   (a 5)
   ((d e) 7)
   ((b f) 3)
   (otherwise 9)
  )
3
```

A cláusula otherwise significa que se x não for nem a, b, d, e, ou f, o case statement vai retornar 9.

2.17. Além de progn...

...existem **MACROS** em LISP para definir blocos. Uma muito usada é prog. Prog permite, entre outras coisas, a DECLARAÇ[Atilde]O explícita de variáveis locais, além de **retorno explícito**:

```
(defun F2.17a nil
        (prog (i j)             ;define i e j como variáveis
                        ;locais inicializadas com nil
                (setq i (read))
                (setq j (read)
                (return (print (- i j)))
        )
)
```

2.18. Lista de Exercícios Nº 2:

2.18.1. Escreva uma declaração em LISP para executar cada uma das operações abaixo:

- Ler dois números, imprimir sua soma e acrescentar 3 ao resultado. Assim 5 e 11 devem produzir 16 e 19 na tela.
- Ler um único valor e imprimí-lo como uma lista. Assim o valor 6 deve produzir (6).
- Ler dois valores e imprimir sua soma como uma lista. Deste modo 6 e 7 devem produzir a lista (13).
- Ler três números e imprimí-los como uma lista.
- Ler três números e imprimir a soma dos dois primeiros e o produto desta pelo terceiro como uma lista.

2.18.2. Escreva uma função que:

- Devolva o valor 1 se seu parâmetro for maior que zero, -1 se for negativo, 0 se for zero.
- Leia um nome. Se este for o mesmo nome que o dado como parâmetro, a função deve imprimir uma saudação simplese devolver o valor **t**. Se for diferente, nao deve imprimir nada e devolver **nil**.
- Dados três parâmetros, se o primeiro for um asterisco, os outros dois serão multiplicados; se for uma barra, o segundo deve ser dividido pelo terceiro; se não for nenhum dos dois, imprima uma mensagem

de erro e assuma o valor zero. A função deve devolver como valor o resultado da operação aritmética.

- Devolva **t** se seu primeiro parâmetro estiver no conjunto de valores especificado pelo seu segundo e terceiro parâmetros e nil se não estiver. Asim: **(func-4 5 5 7)** = t e **(func-4 6 5 7) = nil**.
- Aceite um valor simples e uma lista como parâmetros. Devolva **t** se o valor estiver na lista, **nil** caso nao esteja (este exercício pode ser resolvido de forma recursiva - pense um pouco...).

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.

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.

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.

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.

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)
```

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 exatamento 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`.

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 `eql` para símbolos e números.

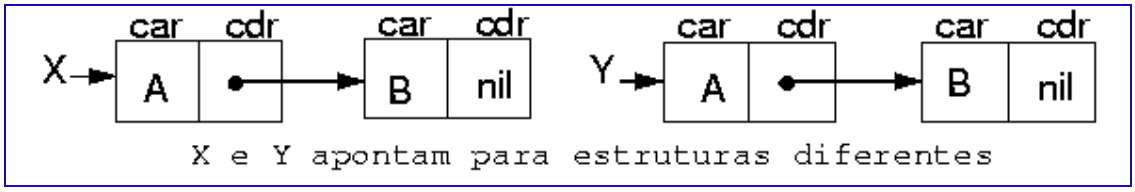
Ele é verdadeiro para dois `cons`es, se e somente se, seus cars são equal e seus cdrs são `equal`.

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

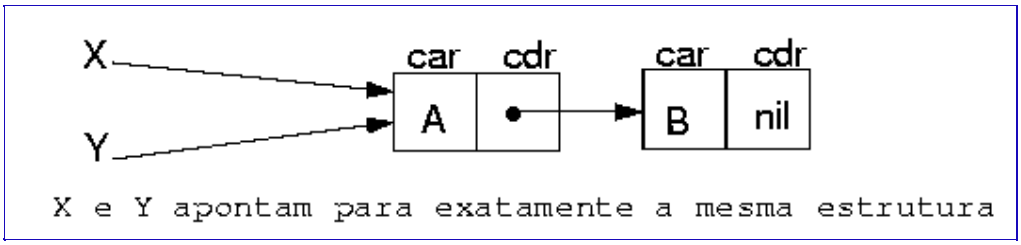
2.24.1. Exemplos de Fixação: Igualdade e Identidade

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
```




```
> (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

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.

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 META-x lisp-mode)`.

Você pode rodar LISP sob Xemacs também, 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 o seguinte atalho:

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

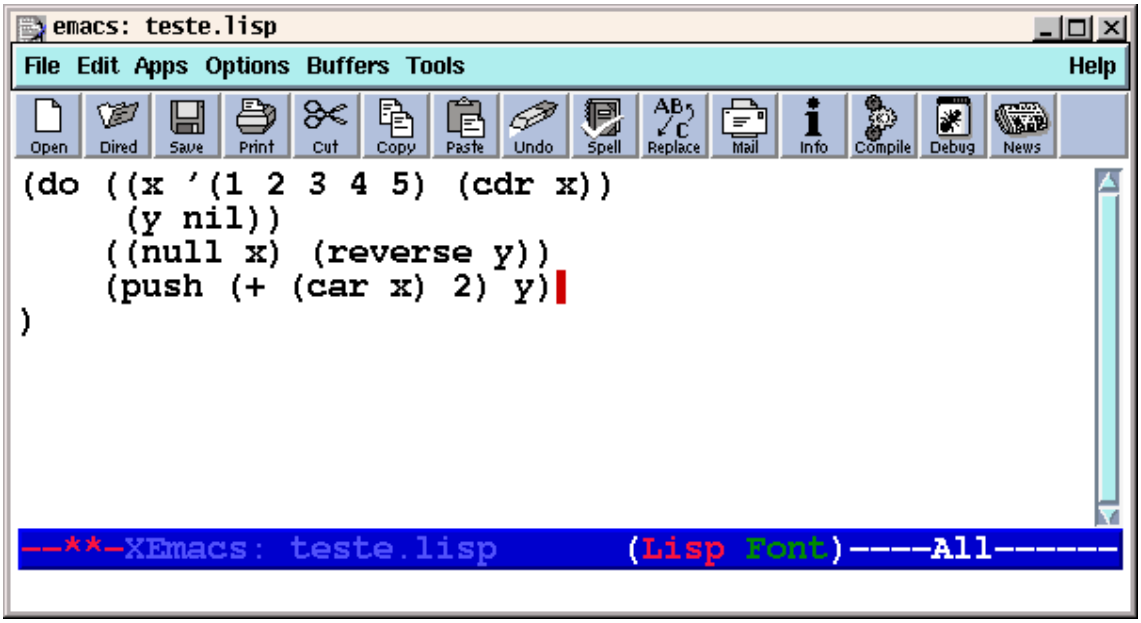
Então, uma vez no Xemacs, digite `META-x run-lisp (ALT-x run-lisp)`. Você pode enviar código LISP para o LISP que você acabou de invocar e fazer toda uma série de outras coisas. Para maiores informações, digite `C-h m (Control-h m)` de qualquer buffer que esteja em modo-LISP.

Outra opcao mais simples e utilizar o **Ponto de Menu CLISP** que aparece no menu **Tools**.

Na verdade, você nem precisa fazer o atalho (symbolic link) acima. Emacs possui uma variável chamada `inferior-lisp-program`; assim você só precisa adicionar algumas linhas de codigo ao seu arquivo .emacs, Emacs vai saber encontrar CLISP quando você digita ALT-x run-lisp.

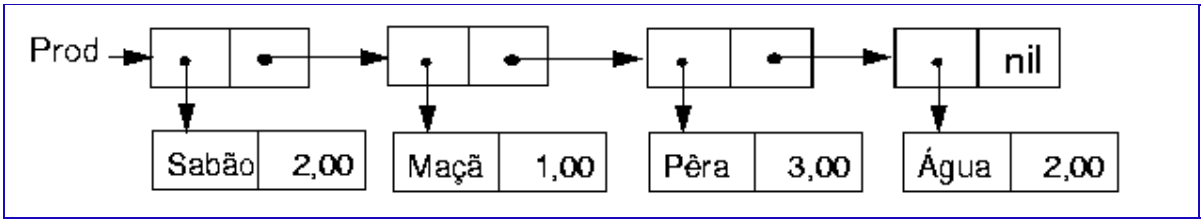
IMPORTANTE1: Voce tem de configurar o seu Xemacs para trabalhar com o pacote Ilisp, para que possa executar o CLISP sob o Xemacs. Para isto voce pode copiar o arquivo .emacs disponivel aqui e coloca-lo no seu diretorio-raiz.

IMPORTANTE2: Se voce **nao esta no INE**, onde este pacote e acessivel a todos, voce **necessita instalar localmente** o pacote ILISP, que permite ao emacs controlar um subprocesso LISP. Este pacote esta disponivel aqui.



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.
