3.2.6. 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
```

3.2.6.1. Definindo uma função:

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

3.2.6.2. Função Recursiva

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

FACT
> (fact 5)
120
```

3.2.6.3. 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
```

3.2.6.4. 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.

3.2.6.5. 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`.

3.2.6.6. 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.

3.2.6.7. 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)
```

3.2.6.8. 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
```

3.2.7. 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...

3.2.8. 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
```

3.2.9. 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.

3.2.10. 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))
    ...
) )
```

3.2.11. 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
```

2. 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.

3.2.12. 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.

3.2.13. 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)
```

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.

3.2.15. 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
```

3.2.16. 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.

3.2.17. Além de progn...

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

```
(defun F3.2.17a nil
         (prog (i j)        ;define i e j como variáveis
                            ;locais inicializadas com nil
                  (setq i (read))
                  (setq j (read)
                  (return (print (- i j)))
         )
)
```

3.2.18. Lista de Exercícios Nº 3:

3.2.18.1. 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.

3.2.18.2. 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...).