A linguagem de programação LISP
o dialeto `Common LISP`

Tarcisio Praciano Pereira[1]

Sobral Matematica
Sobral, 25 de fevereiro de 2013

[1] tarcisio@member.ams.org

Sumário

Lista de Figuras

Nem sempre é fácil a leitura da introdução, afinal elas são escritas depois que o livro todo foi escrito! Elas servem como uma apresentação do trabalho, mas o seu local é aqui mesmo, no início. Leia com desconfiança, e volte a reler.

O primeiro capítulo é uma apresentação da linguagem e tenho como objetivo que ao final do mesmo você esteja escrevendo os seus próprios programas em LISP.

O livro se divide em duas partes, a primeira parte é um tutorial sobre a linguagem de modo que o primeiro capítulo sirva como um rápido tutorial que a conduza a iniciar os seus próprios programas com apoio dos demais capítulos, da primeira parte, que representam guia de referência sobre as primitivas do `Common LISP`.

A segunda parte trata da parte avançada da linguagem incluindo programação orientada à objeto em LISP, `CLOS`, que é um dos sistemas OO para LISP.

O livro está baseado GNU CLISP 2.48 (2009-07-28) `http://clisp.cons.org/` de Bruno Haible e outros que pode ser rapidamente instalado se você estiver usando uma distribuição `Linux` que seja baseada em Debian/GNU/Linux, com o comando

```
sudo apt-get install clisp
```

Há uma variedade bem grande de pacotes que rodam `Common LISP`, muitos de natureza comercial e diversos *livres* como `clisp`. Em geral há uma grande compatibilidade entre eles, entretanto eu não posso falar muito a respeito uma vez que somente conheço o `GNU clisp`. Para ser prático, procure um `LISP` que lhe esteja à mão e comece, você vai ver que não importa muito qual seja o pacote e que este livro lhe vai servir de ajuda mesmo que não seja `clisp`. Apenas `clisp` é de domínio público e fácil de instalar, se você estiver usando Debian/GNU/linux ou uma distribuição que seja baseada no Debian.

Deixe-me agora justificar *porque LISP*?

Quem criou LISP foi o matemático, Paul McCarthy que havia se transformado em cientista da computação como a grande maioria dos que criaram a ciência da computação nos anos 40 e 50. É preciso corrigir esta afirmação, ninguém faz uma coisa tão importante sozinho, e McCarthy era o lider do grupo de pesquisas em *inteligência artificial* do MIT e foi este grupo que construiu LISP.

Ao criar a linguagem, McCarthy, e seus colaboradores, deve ter se inspirado em uma construção Matemática do tipo duma estrutura algébrica, ou talvez pensasse na milenar construção lógica a *geometria euclidiana* e de fato foi uma criação muito bem feita. LISP era um *ambiente* em que existem objetos interagindo de modo fechado, cada operação entre os objetos resultaria noutro objeto também reconhecido pelo *ambiente*, exatamente como acontece numa *estrutura algébrica* ou na *geometria euclidiana*.

Na *geometria euclidiana* a operação de interseção de duas retas será uma reta ou um objeto de *categoria inferior*, (dimensão inferior) mas também um objeto geométrico, *um ponto* ou o *vazio*. É assim que funciona LISP de McCarthy, depois ele fui *se humanizando*...e chegou ao `Common LISP`.

A ideia de McCarthy é dificil de ser reproduzida, infelizmente `clisp` já não entende mais a forma primitiva de LISP e não será possível repetir a bela con-

strução inicial.

O risco com esta descrição é de que você conclua que LISP não é para você, até mesmo porque muito provavelmente você já terá ouvido falar desta linguagem como alguma coisa para *gente envolvida com inteligência artificial* e até mesmo é possível que já tenha escutado alguém dizer que LISP é a *linguagem de máquina da inteligência artificial*.

Primeiro que tudo, em parte estas afirmações são verdadeiras, somente em parte, porque *inteligência artificial* é feita com `C++`, `python`, ou pelo menos o que se espera que seja *inteligência artificial*... Um jogo seria *seria um exemplo de construção de inteligência artificial*? certamente sim, e a linguagem de escolha será `C++` ou `python`, mas também pode ser LISP. Depois, se eu não dissesse aqui o que significa LISP entre os que falam mal da linguagem, ficaria um hiato importante na introdução. LISP quer dizer "Lots of Insipid, Stupid Parenthesis- Quantidade de Parentesis Insípidos e Idiotas - porque a marca dos blocos em LISP são parenteses enquanto que em `C++` são chaves, mas ninguém diz que `C++` é um amontoado de chaves idiotas...

A verdade é que você pode conseguir com LISP, em código muito menor, qualquer coisa que você conseguiria fazer em qualquer outra linguagem de programação. Se eu dissesse que é a linguagem superior a qualquer outra linguagem, não estaria dizendo nenhuma mentira, apenas um pequeno exageiro, porque esta linguagem melhor do que todas as outras realmente não existe. Se você for trabalhar com bancos de dados, escolha PHP ou alguma linguagem da família SQL, caso contrário você terá que montar todas as estruturas estatísticas e relacionamento entre dados, mas com um pequeno trabalho você pode montar um SQL em LISP, ou talvez mesmo já exista um SQL feito em LISP e neste caso seria começar a partir dele.

Qualquer linguagem de programação pode crescer e ser transformada na ferramenta que você precisa, apenas LISP cresce de forma natural e se transforma em qualquer ferramenta facilmente!

Se você seguir em frente verá que tudo isto é verdade, mas será preciso investir algumas horas de estudo para superar o primeiro trauma e logo começar a ver que tem em suas mãos um instrumento de primeira qualidade.

Acho que um bom exemplo, para apresentar neste ponto, é a definição do fatorial. Digite no terminal do `clisp`, ou melhor, digite num editor de textos (ou raspe e cole), grave com o nome `fatorial.lsp` o seguinte texto:

```
(defun      fatorial      (n)
     (if (zerop n)     1
          (* n (fatorial (- n 1)))
     )
)
```

Uma função recursiva!

Se você editou o arquivo `fatorial.lsp`, agora execute no terminal do `clisp`
(load "fatorial.lsp")
e `clisp` já terá integrado o código no ambiente do LISP. Você pode agora executar:

- (fatorial 10)
- (fatorial 100)
- (fatorial 1000)

Surpreendente? Em `C++` não daria par ir além de 13! com valor exato. Em `python` em geral é possível chegar a 3000!, com valor exato, e há outras linguagens em que se pode obter o fatorial, exatamente, para grandes números inteiros, com `calc` é possível, também. Em LISP isto somente depende da memória da máquina! e algumas destas linguagens com que se pode calcular fatorial para grandes números inteiros, foram feitas com LISP, `Maxima`, por exemplo.

Aliás, foi por causa do fatorial que comecei a estudar LISP. Um aluno que me havia ouvido falar em aula da impossibilidade de ir além do fatorial de 13, em Pascal, bateu à porta do meu gabinete, e, quando eu apareci ele me exibiu uma folha de papel cheia de números. Quando eu lhe disse: "e daí?", ele respondeu "750!"Eu comecei a estudar LISP! abandonei Pascal e me tornei um LISPnick!

Fatorial é certamente um cálculo importante, está envolvido em combinações e combinatória é uma área que já justificaria o uso de LISP.

Não seria fácil apresentar agora mais exemplos que justificassem o esforço de superar o stress inicial com o aprendizado de uma linguagem que tem uma sintaxe bastante diferente das linguagens de programação usuais. Mas posso lhe garantir que você já estará convencida disto ao final do primeiro capítulo.

Capítulo 1

LISP

LISP foi criada por um matemático, Paul McCarthy e seus estudantes, em 1955 na mesma época em que foram criados a linguagem `C` e o sistema `Unix`. Um pouco antes que LISP fosse cronstruída, havia sido criada a linguagem FORTRAN que alguns ainda tentam reviver... Em 1990 LISP foi oficialmente alterada ganhando o nome de `Common LISP`. É este dialeto do LISP de que este livro trata.

Este primeiro capítulo é uma apresentação da linguagem e tenho como objetivo que ao final do mesmo você esteja escrevendo os seus próprios programas em LISP.

O livro se divide em duas partes, na primeira parte o primeiro capítulo é um tutorial sobre a linguagem de modo que a conduza a iniciar os seus próprios programas.

Os demais capítulos da primeira são um guia de referência da linguagem servindo de apoio ao primeiro capítulo, eles foram redigidos de forma que você os possa ler forma independente para aprofundar o uso das funções básicas da linguagem. Mas eles devem ser lidos depois do capítulo 1 e posteriormente representam um guia de referência sobre as primitivas do `Common LISP`.

A segunda parte é mais avançada, incluindo programação orientada à objeto em LISP, `CLOS`, que é um dos sistemas OO para LISP.

O livro está baseado GNU CLISP 2.48 (2009-07-28) `http://clisp.cons.org/` de Bruno Haible e outros que pode ser rapidamente instalado se você estiver usando uma distribuição `Linux` com o comando

```
sudo apt-get install clisp
```

Há uma variedade bem grande de pacotes que rodam `Common LISP`, muitos de natureza comercial e diversos *livres* como `clisp`. Em geral há uma grande compatibilidade entre eles, entretanto eu não posso falar muito a respeito uma vez que somente conheço o `GNU clisp`.

1.1 Conceitos fundamentais.

Não leia esta seção agora, passe direto para a próxima, ou se insistir em sua leitura, faça-o rapidamente apenas para guardar alguma informações em seu subconsciente... Se encontrar alguma coisa difícil, lembre-se desta observação.

Em LISP o *processamento* gira em torno de dois objetos básicos: listas e átomos. Estas são as duas idéias básicas de LISP e é preferível descrevê-las com um circunlóquio do que tentar uma definição formal. Todas as tentativas que conheço de incluir na teoria estes dois conceitos são falhas, é melhor agir como em Matematica que considera elemento e conjunto dois conceitos básicos que não devem ser definidos. Você sabe o que um *átomo* e o que é uma *lista* e vou partir deste ponto.

Qualquer objeto em LISP é um *átomo* ou uma *lista*. Assim as listas são compostas de *átomos ou de listas* e por sua vez os átomos são os menores componentes duma lista.

Exemplo 1 (lista e átomo) *Átomo e lista*

1. *3,4 são átomos. Teste (atom 3). Teste (atom atom), vai dar erro mas os erros fazem parte do processamento em LISP. Leia as justificativas. Vou corrigir este erro de forma um tanto incompreensível, neste exato momento, mas logo você irá compreender a correção:*

   (setq atom 3)

   (atom atom)

   a resposta agora é `t`*, quer dizer: "*é verdade!*. Eu atribui o valor* 3 *à variável* `atom` *que é o nome de uma função de LISP e isto é possível.*

2. *(3 4 5 6) é uma lista. Teste (listp (3 4 5 6)), vai dar erro que vou discutir logo abaixo.*

3. *Numa lista o primeiro elemento sempre deve ser uma função e você pode corrigir o erro anterior assim: (listp '(3 4 5 6)), teste! A forma clássica para esta expressão é "(listp (quote (3 4 5 6)))", mas como precisamos seguidamente de usar* `quote` *para evitar que uma lista seja evaluada, os planejadores da linguagem inventaram esta forma abreviada de usar* `quote` *e assim as duas expressões:*

   - `(listp '(3 4 5 6))`
   - `(listp (quote (3 4 5 6)))`

   são equivalentes e normalemente todo mundo prefere usar a primeira, a não ser os LISPnick_shiitas, estes geralmente preferem usar a segunda expressão. Aliás, um LISPnick_shiita nunca escreveria como esta acima, iria preferir escrever assim:

4.
```
(listp
     (quote
          (3 4 5 6)
     )
)
```

e teria muita razão porque estaria claramente indicando qual é a função que cada parentesis termina. Mas, como usamos com frequência editores de texto que associam os parenteses que se fecham, não precisamos ter esta grande preocupação hoje. Vou falar a verdade, eu mesmo sou um LISPnick_shiita e vou preferir redigir o texto usando esta forma porque vai ficar melhor para sua leitura, e logo logo você irá perder o receio desta enxurada de parentesis quando se convencer que LISP é uma linguagem muito especial.

A palavra *átomo* foi tomada emprestado da Física quando LISP foi criando na década de 50 quando já se conhecia bem que os *átomos da Física* eram divisíveis mas que é este o significado grego da palavra *átomo*. Em LISP eles são indivisíveis, portanto os átomos são as *unidades minimais* desta linguagem de processamento, mas a menor unidade lógica é uma lista.

Qualquer módulo em LISP ou é uma lista ou uma lista de listas. Por exemplo, quando você ativa uma implementação LISP, por exemplo `clisp`, vem para a memória do computador uma lista de todos os símbolos que estão associados ao segmento de memória com um código executável (no assembler da máquina que você estiver usando), entre eles está `eval` que é logo ativado e passa a avaliar os símbolos que você digitar. Quer dizer que `clisp` é uma "*simples*" implementação da função `eval`, em geral implementada também em LISP.

Desta forma uma lista pode ser avaliada se o seu primeiro átomo for uma função e os demais parâmetros para esta função, ou você pode evitar a avaliação colocando esta lista dentro de outra lista cuja primeira função é "quote"que diz ao `eval` que deve apenas repetir a lista sem avaliar o primeiro símbolo.

Experimente, digite no `clisp`, ou raspe e cole,

`(+ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24)`

se você fizer a conta

`1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20+21+22+23+24`

verá que `clisp` não errou e calculou a soma de uma quantidade arbitrária de números (experimente aumentar a lista)! Experimente agora:

`(* 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24)`

o produto daquela lista anterior de números.

Isto lhe mostra que o primeiro átomo de uma lista deve ser uma função e em LISP é possível fazer funções para receberem uma quantidade arbitrárias de parâmetros, somente tem uma outra linguagem com a qual se pode também fazer isto que é `C`.

Mas para fazer, em `C`, isto aqui:

```
(* (+ 3 4 ) 5)
```

é preciso algumas linhas extras, e vale para `python, Pascal` e todas as linguagens que conheço!

Um dos pontos revolucionários de LISP é que um programa, (leia-se, uma lista) pode receber a sí própria como parâmetro o que permite fazermos programas que se automodifiquem, quer dizer que um programa pode passar a ser o agente de sua própria atualização. Isto na verdade quer dizer "*passar para uma função, uma função como argumento*" e para fazer isto em `C` é preciso avançar alguns kilómetros a dentro da linguagem. Em `python` isto também pode ser feito com facilidade. Na maioria das linguagens não é tão simples como em LISP, (ou `python`), quando for possível.

Exemplo 2 *Atomos e listas.*

Os números são átomos. Os compiladores ou interpretadores LISP foram construidos para reconhecerem qualquer número que possa caber na memória da máquina. Eles serã reconhecidos como átomos.

'(1 2 3 4) é uma lista, você pode digitar esta expressão no `clisp`*, sem receber mensagens de erros, por causa da função* `quote`*.*

As listas são conjuntos ordenados *de átomos ou de outras listas. Assim* `'(1 (1 2 3 40) 3 4 5)` *é uma lista com cinco elementos.*

As listas tem dois aspectos estruturuais que fazem parte integrante da síntaxe da linguagem LISP:

1. Listas são conjuntos ordenados. Assim as duas listas

$$'(123), '(321)$$

   são listas diferentes.

2. O primeiro elemento de uma lista deve ser uma função. As funções são símbolos aos quais se encontra associado um algorítmo legal em LISP.

3. Se o primeiro símbolo de uma lista não for uma função o programa `eval` produzirá um erro indicando isto, a não ser que a lista seja incluida em outra lista cujo primeiro elemento é a função QUOTE que indica ao `eval` que a lista não deve ser avaliada, mas simplesmente reproduzida. Acima , em $'(123)$ temos uma abreviação deste processo, quer dizer que as duas expressões

```
'(1 2 3)  ;  (quote (1 2 3))
```

   são equivalentes.

Vemos assim que existe um terceiro conceito qualificador de objetos em LISP: *funções*. Assim podemos dizer que em LISP existem apenas os seguintes objetos:

1. Átomos.
2. Símbolos.
3. Listas.

4. Funções.

e que a relação entre estes objetos é:

1. Listas são conjuntos ordenados de átomos ou de outras listas.

2. Símbolos são são as palavras da linguagem humana que servem para designar átomos, listas ou funções. Existe uma regra a ser adotada para designar símbolos que irei apresentar depois.

3. As funções são átomos particulares que associados um algorítmo a ser aplicado aos demais elementos de uma lista da qual elas, as funções, são o primeiro átomo. Ver abaixo a estrutura interna de um símbolo.

4. O sinal ' colocado na frente de uma lista indica ao interpretador/compilador LISP que ele deve receber a lista sem tentar *avaliar* o seu primeiro átomo e é uma abreviação da função `quote`.

Foi por esta razão que escrevemos nossos primeiros exemplos de listas com o sinal `<'>` colocado na frente delas. Seria ilegal escrever em LISP `(1 2 3)` uma vez que não há nenhum algoritmo associado ao símbolo "1"e portanto nada fica indicado para ser feito com os demais elementos da lista: um erro de síntaxe teria sido cometido.

Uma característica comum de um *ambiente LISP* é interpretação das listas. Se você estiver ante um terminal rodando `clisp`, escrevendo (+ 1 2) e *dando enter* provocará o imediato aparecimento do número "3"na linha seguinte. Ao receber uma lista, `clisp` a evalua e escreve seu resultado.

O conceito de *programa* das demais linguagens de programação, aqui se resume numa lista. Uma lista pode ser um programa, mas também pode ser conjunto ordenado de dados, se for este o caso deve ser "quotada" para prevenir que LISP tente fazer uma avaliação.

LISP tem duas funções importantes: ATOM, LISTP[1] cujo efeito é identificar se o objeto que logo lhes segue é um átomo ou uma lista.

Finalmente existe uma lista especial em LISP, '(), a lista vazia que é também considerada um átomo. É a única exceção. Esta lista, ou átomo, tem um símbolo a ela associado: NIL. Aqui temos uma outra característica particular de LISP associado ao conceito de VERDADEIRO e FALSO. Tudo que não for NIL é verdadeiro. Se você escrever:

```
(equal '(1 2 3) '(3 2 1))
```

a resposta do terminal `clisp` será NIL, porque listas são na verdade sucessões (lista ordenadas). Se você escrever:

```
(equal '(1 2 3) '(1 2 3))
```

[1]LISP guarda um hábito das primeiras linguagens de programação, usar letras maiúsculas para os nomes dos símbolos da linguagem. O pacote que uso, `clisp`, admite, na linha de comando, `clisp -modern` quando então passa a reconhecer a diferença entre maiúsculas e minúsculas com os símbolos da linguagem todos em minúsculas. Em LINUX coloque `alias clisp='clisp -modern'` no arquivo `/.bash_aliases`, no diretório `home` de sua área, depois basta chamar `clisp`.

a resposta do terminal será T. Esta é uma forma de comunição com os humanos. NIL ou T indicam verdadeiro ou falso. É bom evitar de atribuir algum valor ao símbolo T, porque, primeiro que tudo LISP pode aceitar [2] sem que isto provoque nenhum problema lógico maior, (desde que o valor não seja NIL, obviamente). Mas o resultado pode ser algumas vezes desconcertante...

1.2 Programas, algorítmos, funções.

Uma das dificuldades básicas de quem aprende LISP se encontra no fato de que já andou pelo menos rodando programas em outras linguagens de processamento. Vamos tentar tirar partido desta dificuldade.

Na grande maioria das linguagens de processamento existe uma diferença entre *programas* e as outras informações. Em LISP tudo que existe é uma síntaxe ligando alguns símbolos a outros símbolos de modo que numa crescente complexidade se pode executar qualquer algorítmo.

Desta forma em LISP há dois tipos de símbolos:

1. aqueles que identificam um algorítmo que deve ser aplicado a outros símbolos: as funções.

2. os outros.

e obviamente há uma função em LISP para identificar se um símbolo é uma função, ou não. Junto com os identificadores de listas e de átomos, esta função é indispensável na organização da lógica dos algorítmos.

Aquilo que nas demais linguagens de processamento se caracteriza como um programa, em LISP será uma lista cujo `car` é uma função. Desta forma o fatorial definido na introdução é um programa:

```
(defun fatorial (n)
      (if (zerop n) 1
            (* n (fatorial (- n 1)))
      )
)
```

Na verdade eu aqui estou criando uma confusão e vou piorá-la. A `macro defun` cria um programa além de ser também um programa, o programa criado é

```
(if (zerop n) 1
      (* n (fatorial (- n 1)))
)
```

e como esta lista está associada ao símbolo `fatorial` ela volta a ser chamada com `(- n 1)` até que `(zerop n)` seja verdade quando se dá a última multiplicação por 1.

[2] `clisp` não vai aceitar!

Os idealizadores de LISP, na década de 50, já se adiantavam na concepção de modularização que só ficaria clara cerca de 20 anos depois. LISP é por natureza uma linguagem de processamento modularizada. Um programa em LISP é uma lista em que alguns dos seus elementos podem ser outras listas, sempre obedecida a regra sintática de que o primeiro elemento de uma lista deve ser uma função ou a lista deve ser precedida do síbolo `<'>` caso ela deve ser considerada como uma lista de dados ou um símbolo que não deva ser inicialmente avaliado.

É comum observar-se que em LISP um programa, de forma natural pode ser um dado passado a outro programa, em particular um programa pode receber a si mesmo como dado e assim se auto-modificar [3]. Este é o aspecto que faz com que LISP seja considerada como a *linguagem de máquina da Inteligência Artificial*.

Por sua natureza modularizada, programar em LISP significa essencialmmente descobrir os menores módulos do problema e escrever as funções que se ocupam deles e assim num planejamento de *baixo para cima*, ou equivalentemente, *de cima para baixo*, criar a função que irá gerenciar a solução do problema. Talvez melhor fosse dizer *criar a função que irá gerenciar a evolução do problema*, pois exatamente se pode pensar que um problema evolue e sua solução não pode residir num *programa estatítico*.

Entretanto, aprendendo a programa em LISP, pelo menos uma grande vantagem pode ser obtida: *se treinar em modularizar os problemas, aprender a escrever pequenas funções que executem tarefas bem genéricas (que possam ser reutilizadas) e depois chamar estas funções com uma estrutura adequada para resolver um problema.*

1.3 Símbolos, nomes, expressões.

Vamos aprofundar um pouco mais a análise da linguagem LISP e dos seus elementos. As linguagens de processamento são assim chamadas porque elas representam uma forma de comunicação *homem- máquina*. Elas são exemplos de "linguagens"artificiais como também é o caso do *esperanto*. Depois do advendo dos computadores, em forma resumida, podemos descrever a história evolutiva deste processo de comunicação homem-máquina, com as seguintes etapas:

listagem sequencial de operações.

listagem sequencial de operações com interrupçêos rudimentares de fluxo: surgimento de condicionais: década de 50.

Fluxo repetitivo controlado. Lógica avançada.

Uso de símbolos: década de 60.

Uso de macros: década de 80.

[3] Um pequeno exemplo disto é o fatorial que recebe o próprio fatorial com o valor do parâmetro alterado: `(- n 1)`.

Observe que LISP se originou por volta de 1954 no MIT este início de história estando habitualmente centrado no nome de Paul McCarthy.

A ideia de algorítmos sequenciais dominou a ciência da computação até recentemente sendo o paradigma sobre o qual foram construidas a maioria das linguagens populares de computação como FORTRAN, C, Pascal, Algol, Cobol. LISP se caracteriza por ser uma *linguagem funcional* o que só seria compreendido inteiramente bem depois de sua criação.

Modernamente, depois e inclusive ao longo dos anos 80, se intensificou o conceito de abstração em programação. Sublinarmente a ideia é que se tem em Matemática. Abstração consiste em definir uma estrutura da qual objetos são representações, uma classificação. Nas linguagens modernas de programação, um programa não deve resolver um problema, mas sim se aplicar *numa classe de problemas*.

Também as linguagens de programação se constituem de *expressões*, como a linguagens humanas. Um algorítmo é uma expressão que pode ou não resolver algum problema. Em LISP as expressões se apresentam sob forma de listas, estas constituidas de átomos ou outras listas.

Como um dos objetivos de qualquer linguagem de processamento se encontra o de se comunicar com humanos, em LISP existe um tipo especial de átomo que são *cadeias de caracteres* e que se caracterizam por estarem encerrados entre aspas.

```
"Este objeto é  uma cadeia de caracteres."
```

Agora temos uma visão total de todos objetos da linguagen LISP, ou suas expressões:

Átomos:

números, um tipo especial de átomo que é formado segundo as regras da Aritmética.

simbólicos, representados pelo que n'outras linguagens de programação são *nome de variáveis*. Estes átomos contém uma informação que aponta para um endereço de memória que lhes caracteriza o tipo: função, macro ou forma especial. Teste

(atom '+)

Cadeias de caracteres, um tipo especial de átomo formados pela presença de um par de aspas abrindo e fechando. Teste

```
(atom "werq qewrqerq er")
```

Listas que se caracterizam pela presença de parenteses abrindo-fechando. Um exemplo particular de lista é `()`, a lista vazia que também é um átomo sendo o único átomo que também é uma lista. Experimente:

```
(atom ())
(listp ())
```

Um átomo é uma expressão, uma lista é uma expressão. Estas são as possiveis expressões em LISP.

Se você digitar em `clisp`

qwerqer

a função `eval` que está rodando como `clisp`, irá responder que este símbolo não existe mas você pode criar este símbolo usando `setq`

`(setq qwerqer 35)`

depois do que é legal digitar

qwerqer

experimente!

Os símbolos não númericos, nem do tipo "cadeias de caracteres", tem duas células para guardar informações sobre eles. Uma memoriza o endereço na memória onde fica guardado o seu valor. A outra célula se chama de "célula funcional"onde fica guardada endereço da memória onde está algorítmo, como função, se esta definição for feita. Ou seja, você pode guardar um valor de uma função no simbolo que designa esta função, experimente o seguinte exemplo, e ignore por um momento o significado do código, mas deixe-me explicar que a última linha contém a função `format` que serve para *formatar os dados para impressão* em que você pode ver "`soma =` dentro do texto entre aspas (*texto mesmo*), e ao final o símbolo `soma` onde estão guardados os valores da soma em cada etapa da iteração. É o nome da função `soma` sendo usado como uma variável.

```
(defun soma (n)
      (setq soma 0)
      (do* ((k 0 (+ k 1)) (soma (+ soma k) (+ soma k)))
            ((> k n) soma) ; quando verdadeiro, para
            (format t "k= ~d soma =  d ~%"k soma)
      )
)
```

Procure no índice remissivo *propriedades* e leia agora, rapidamente sobre *lista de propriedades*. Rapidamente! não se perca!

Observe este peculiaridade de LISP que mesmo um símbolo que seja o nome de uma função, pode ser usado para guardar valores como uma variável comum, entretanto esta é uma prática que deve ser usada com cuidado. Observe que neste código `soma` é uma variável local e neste caso não usei a célula de valor para guardar o resultado desta função. Isto poderia ser feito assim:

`(setq soma (soma 10))`

a função `setq` tem acesso a célula de valor de um símbolo ou cria um simbolo colocando um valor em sua célula de valor.

1.4 Como tudo começa.

Procurar o início é uma caracteristica sequencial e possivelmente um vício de pensamento. Os fatos *e as* coisas *não têm um início, nem existem con-*

sequências, premissas ou causas a não ser nos casos mais simples. Procurar pelo início *representa uma simplificação ingênua do Universo que é extremamente complexo.*

Entretanto existe um momento em que ligamos o computador e LISP indica sua presença com um sinal qualquer na tela. Estamos no início[4]... e LISP começa por aplicar uma função chamada EVAL a qualquer coisa que digitarmos. LISP é ele mesmo o parentesis inicial e a tecla RETURN o final. O resultado desta lista é o valor que ela tem: se digitarmos um *nome*, LISP examinará em sua memória, se o conhecer e responderá com o seu valor. Se o estiver encontrando pela primeira vez, o repetirá indicando que o *evalou*[5] como sendo seu próprio nome. É exatamente o que vai acontecer com um número. Se reconhecer como sendo uma lista, marcada pelo QUOTE, fará o mesmo que com números ou nomes desconhecidos, a repetirá. Se for uma lista não QUOTada, verificará se o seu primeiro elemento é uma função, lhe examinará o algorítmo e a executará. Se não for uma função emitirá um laudo: erro de sintaxe, o símbolo "nome" tem sua definição funcional *vazia*.

Isto é tudo, mas que pode atingir um nível de complexidade muito elevado, porque dentro de uma lista pode estar outra lista, dentro da qual pode estar outra lista...

Neste lista de complicações se encontram funções que produzem efeitos colaterais em vez de produzir um valor apenas. O exemplo mais simples é a função DEFUN que serve para definir novas funções. Em algumas implementações o seu resultado consiste em repetir o nome da função que está sendo definida, apenas para indicar ao usuário que a análise sintática foi positiva. `clisp` faz isto!

Para que tudo isto funcione bem uma regra lógica de funcionamento tem que existir: o processo de evaluação das listas de se faz de dentro para fora. LISP segue verificando as listas e ao encontrar listas internas começa por evaluar estas inicialmente e assim sucessivamente até atingir o fundo do poço, depois é que sairá calculando valores de volta. Isto cria um conceito bem natural na análise de *programas* LISP, o conceito de profundidade de uma lista, consiste na determinação do número inteiro que corresponde a grau mais interno de todas as listas e sub-listas.

1.5 A estrutura interna de um símbolo

O conteúdo desta seção deve ser considerado como uma curiosidade que pode eventualmente ser usada, mas sob o rótulo, *somente faça se você souber o que está fazendo*. É difícil de se preverem os danos que podem ser causados por um uso incorreto das informações contidas nesta seção.

[4] Ou algo mais simples, você tem nas mãos um quantidade de fio emaranhado e aí tem um `início` que é preciso encontrar para obter um fim útil para alguma coisa,... às vezes precisamos procurar o `início`.

[5] `clisp` não faz isto, se não reconhecer o símbolo vai dizer que este símbolo não pertence a sua tabela de símbolos, emite uma mensagem de erro. Outras implementação de LISP fariam isto.

A única razão para que eu apresente aqui esta informação é a de transmitir para o programador algumas razões do que acontece em LISP e naturalmente permitir que ele use, se *souber o que está fazendo*, algumas técnicas avançadas de uso da linguagem.

Quando você ativa o interpretador LISP e que o sistema operacional reserva para a linguagem um segmento de memória acontecem as seguintes ações:

- É criado um *espaço de nomes* inicialmente ocupado com os símbolos da linguagem;
- A cada símbolo é associado um conjunto de 5 bytes de mémoria designados como (no jargão computacional do LISP)
    1. célula do nome, que contém um ponteiro para nome, no espaço de nomes, (e vice versa, naturalmente);
    2. célula de valor;
    3.
    4. célula funcional, que contém um ponteiro para o segmento inicial da memória onde começa o código, se este "nome"estiver associado a algum algoritmo, caso contrário o ponteiro é NIL.
    5. lista de propriedades.

1.6 As funções primitivas

Nesta segunda seção eu vou mostrar-lhe a ideia de McCarthy, infelizmente `clisp` já não entende mais a forma primitiva de LISP e não será possível repetir a bela construção inicial. Observe também que lendo diversos textos sobre a história de `LISP` você vai encontrar também *distintas formas de ver a ideia inicial de McCarthy*, esta é a minha, como um matemático estudando lisp.
A primeira ideia de McCarthy foi escolher a estrutura de dados da linguagem: **lista**. Os números naturais são uma lista, com eles se constroi o resto dos números, planos para a geometria, placas gráficas e finalmente se modela o Universo nos computadores... Claro, tem muita coisa nesta história, mas tudo parte da lista dos números naturais!
E assim McCarthy criou uma linguagem para fazer processamento de listas, *lists processing*, LISP.
palavras chave: = atom bloco car cdr cons equal list listp nil nth progn setq symbolp

Não leia esta seção agora, passe para a próxima! Mas se você insistir, for desobediente, meus sinceros cumprimentos: somente as pessoas desobedientes é que podem construir algo interessante! e LISP, positivamente é para os desobedientes!

Nesta seção vou mostrar-lhe um pouco da história desta incrível linguagem de processamento. Tem que ser um pouquinho, ela é das primeiras linguagens, foi produzida em 1956 e como sua sucessora, `Common LISP`, está viva e se encontra entre as mais bem sucedidas linguagens, então ela tem uma história para ser contada que não caberia nas quatro ou cinco páginas de uma seção do primeiro

capítulo. Então você pode facilmente encontrar outro livro que conte uma outra história, vai depender de quem escreve esta história!

Há algumas funções que vou apresentar aqui que não pertencem ao conjunto inicial da linguagem construída por McCarty, como `nth`. O objetivo desta primeira seção é quase que histórico, mostrar como começou LISP. A próxima seção vai conduzí-la a *programas* mais efetivos.

Coloque `clisp` no ar, e se você não o tiver ainda instalado, faça-o agora, ou alternativamente use o sistema `Common LISP` que seja de sua preferência ou que esteja à sua disposição, mas comece, desde o começo a experimentar.

Exemplo 3 (algumas expressões) *LISP, algumas expressões*

1. *3 resulta em 3. Porque 3 é um átomo e* `clisp` *avalia os átomos resultando no próprio átomo. Todos os números são átomos.*

2. *(+ 3 4) é uma lista, e em LISP existe apenas dois tipos de objetos: átomos ou listas. Numa lista o primeiro elemento deve ser uma função e os demais elementos são dados passados para esta função. Neste caso passei 3,4 para a função "somar" que vai resultar em 7.*

3. *(+ 1 2 3 4 5 6 7 8 9 10) vai resultar na soma dos termos desta* **p.a.** *e se eu conseguisse escrever (+ 1 2 3 . . . 1000) o resultado seria 500500. Mas na introdução, se você não tiver lido, leia agora, você viu que calcular o fatorial de um número qualquer é coisa simples para LISP. Agora você vê que podemos lhe passar um número indefinido de valores para uma função. Porque LISP opera com listas!*

4. *(* (+ 4 5) 3) é um exemplo complicado! As funções, em LISP são o primeiro elemento de uma lista, caso contrário dá erro. Experimente*

   ```
   (1 2 3 4 5 6 7 8 9 10)
   ```

   que vai resultar em uma longa lista de linhas de reclamação somente porque LISP não reconheceu uma função no primeiro elemento desta lista. Mas deixe-me quebrar ((+ 4 5) 3) em pedaços:* `clisp` *avalia resultando em 9 portanto na verdade o que você lhe passou foi (* 9 3) cujo resultado é 27.*

   (a) *Primeiro tem a função "multiplicar" depois vem (+ 4 5) que* `clisp` *avalia como 9.*

   (b) *Portanto você passou (* 9 3) ao clisp.*

5. *(> 3 4) significa que você passou ao* `clisp` *a função lógica > e está perguntando se 3 é maior do que 4. Verifique o resultado no clisp. Tente também (< 3 4). Ou então tente (< 1 2 3 4 5 6 7 8 9 10) e você está perguntando se sucessão 1 2 3 4 5 6 7 8 9 10 é crescente. Experimente permutar alguns dos elementos e a sucessão não seria mais crescente, e nem decrescente!*

   ```
   (> 1 2 3 4 5 7 6 8 9 10)
   (< 1 2 3 4 5 7 6 8 9 10)
   ```

6. *E como seria um programa em LISP? Um programa, em* `clisp`*, é uma lista cujo primeiro elemento seja uma função. Observe que não estou dizendo que será um* programa correto*. E pode ser até correto mas que* `clisp` *não entenda porque pode falta alguma coisa:*

   `(power 2 3)`

   vai encher a tela de reclamações porque a função `power` *não foi ainda definida, mas eu não vou defini-la agora para não cortar a linha do pensamento.*

   Portanto (+ 1 2 3 4 5 6 7 8 9 10) é um programa e você pode passar um programa para outro programa:

   ((+ 1 2 3 4 5 6 7 8 9 10) 100)*

   ou

   (+ 1 2 3 4 5 6 7 8 9 10 (+ 1 2 3 4 5 6 7 8 9 10))

   LISP avalia os elementos de uma lista da direita para esquerda para finalmente chegar ao primeiro elemento que deve ser uma função para avaliar os dados que lhe forem passados.

Em `clisp` existe uma função para construir listas:
`(list 1 2 3 4 5 6 7 8 9 10 11 12 13)`
produz o objeto
`(1 2 3 4 5 6 7 8 9 10 11 12 13)`

Se você digitar isto no terminal do `clisp`[6] terá feito uma *operação legal*, porém em vão, porque em seguida se evapora a lista `(1 2 3 4 5 6 7 8 9 10 11 12 13)`.

Execute
`(setq L (list 1 2 3 4 5 6 7 8 9 10 11 12 13))`
e agora você terá criado um símbolo, `L` associado a lista dos números naturais menores do que 14. Você pode digitar `L` e `clisp` vai imprimir
`(1 2 3 4 5 6 7 8 9 10 11 12 13)`

Em `LISP` existem dois tipos de objetos: átomos e listas e duas funções primitivas, chamadas *predicados*, porque são lógicas, para testar se um objeto é um símbolo ou uma lista:
`(listp L)`
vai responder `T`, se você tiver criado a lista `L`. No caso contrário vai responder `NIL` que também é a lista vazia, mas representa o falso.

Teste:
`(equal NIL ())`
e `LISP` vai responder `t`. `equal` é uma função mais abstrata do que a função `=` que é uma função lógica aritmética. A expressão `(equal 1 1)` é válida e a expressão `(= '(1 2) '(1 2))` não é válida. Teste! Teria que usar `(equal '(1 2) '(1 2))`.

[6] De agora em diante em vez de escrever "no terminal do `clisp`, vou passar a escrever apenas " no `clisp`".

Em LISP o falso é () que é idêntico a nil e qualquer símbolo, diferente da lista vazia representa o verdadeiro. A lista vazia é um símbolo e o teste:

```
(atom ()).
```

vai indicá-lo. Teste também `(equal () nil)`.

Os números são átomos

A linguagem C++ copiou isto, *zero é o falso* e *qualquer coisa diferente de zero é o verdadeiro.*

Agora vem duas primitivas muito especiais: car, cdr, experimente no clisp

```
(car L) --> 1
(cdr L) --> (2 3 4 5 6 7 8 9 10 11 12 13)
```

a função car indica qual é primeiro elemento de uma lista, e cdr indica a lista[7] que sobra tirando o primeiro elemento. Nenhuma desta funções é *destrutiva*, depois de executar estas duas funções execute

```
L
```

para ver que L não foi alterada. Mas se você executar

```
(setq L (cdr L))
```

então sim, você terá alterado L.

Experimente agora:

```
(car (cdr L))
(car (cdr (cdr L)))
(car (cdr (cdr (cdr L))))
(setq L (cdr (cdr (cdr L))))
```

A última operação é destrutiva, você terá alterado L.

E agora você certamente balançou a cabeça e concluiu que LISP é uma linguagem complicada, porque para obter o 50º elemento de uma lista com 1500 elementos precisariamos de meia folha de texto de aplicação da função cdr...

Não é assim porque LISP cresce conforme seja necessário e podemos criar, uma função que selecione o *enésimo* elemento de uma lista, mas esta função já existe, a função nth que recebe dois parâmetros:

```
(nth 3 L) --> 4
(nth 0 L) --> 1
(nth 12 L) --> 13
```

e você pode ver que as listas são indexadas a partir de zero, C++ copiou isto do lisp.

Não sabendo que nth existia, você poderia construir a uma e chamá-la enesimo. Mas já existem em Common LISP funções para fazer praticamente todas as operações que precisarmos fazer com listas, e se você precisar de alguma muito esquisita, você logo vai aprender a construí-la.

Vou terminar com uma função que é das primitivos, mas um pouco extranha: quote, experimente:

```
(quote L)
```

lembre-se que fiz anteriormente

[7] Logo você irá ver que esta frase está errada, que cdr lhe fornce o cons de um objeto. As listas, em LISP, são uma sucessão de objetos com um ponteiro para outro objeto e cdr mostra o que tem depois do ponteiro que é (cons (car L)) *esta função existe*, e logo você vai ver como é usada.

`(setq L '(1 2 3 4 5 6 7 8 9 10 11 12 13) )`
e observe que
`(1 2 3 4 5 6 7 8 9 10 11 12 13)`
não foi avaliado, caso contrário daria num erro. Então `quote` evita a avaliação e produz como resultado o proprio objeto que recebeu como parâmetro. Isto pode lhe parecer meio rídiculo, agora! Depois você vai entender que faz parte essencial da lógica da coisa. É tão importante o `quote` que desde o princípio ganhou um símbolo próprio:

`(quote L)` e `'L` são equivalentes, experimente! E como brincadeira, execute e procure entender a seguinte sequência de operações:

```
(setq L '(1 2 3 4 5 6 7 8 9 10 11 12 13) )
L
(cons '+ L)
(eval (cons '+ L))
```

A função `cons` serve para adicionar um elemento como `car` de uma lista. Isto é importante na *construção automática de programas* quando vamos precisar de acrescentar uma função como primeiro elemento de uma certa lista.

1.6.1 cons car cdr

Estas seriam as três mais primitivas funções de LISP. Uma lista é um sistema de pares de *ponteiros*

- o primeiro ponteiro idenfica, *aponta*, o `car` da lista, que pode ser uma lista!
- e o segundo ponteiro identifica uma lista, que pode ser a lista nula indicando que a lista terminou.

A função `cons` serve para construir listas:
`(cons '1 '(2 3)) --> (1 2 3)`
Como no caso dos conjuntos, partindo de um conjunto tão simples como o conjunto dos números naturais, podemos chegar, usando o `operador` "partes de" em conjuntos extremamente complexos, com listas podemos construir dados, também, extremamente complexos: programas muito complexos e, se não tivermos cuidado, podem também *não processar* nada interessante!

1.6.2 progn

Esta função serve para definir um bloco, quer dizer, juntar duas ou mais funções para serem executadas sequencialmente, experimente:

```
(progn      (+ 1 2 3)
     (* 1 2 3 4 5)
     (setq a (+ 1 2 3) b (* 1 2 3 4 5))
)
```

e para entender *o que aconteceu* observe que toda função em LISP tem como resultado o último valor calculado (ou função executada). Neste caso é o valor da função `setq` cujo último valor calculado é a atribuição ao símbolo `b` do produto `(* 1 2 3 4 5)`, o fatorial de 5.

Uso de blocos no condicional `if` que admite, como em todas as linguagens de programação apenas uma expressão *do tipo então* e também apenas uma expressão *do tipo ou_então*. Usando `progn` você pode criar um bloco de funções para serem executadas em cada uma dessas clausulas do `if`.

Outro uso de blocos vai ser em *laços* para tornam mais visível um certo grupo de funções a serem executadas, mas o que é importante, para definir o valor de retorno do bloco, observe o exemplo acima.

Nesta seção vou mostrar-lhe as funções da *segunda geração*, elas não existiam na construção inicial de McCarthy, mas foram logo em seguida criadas e fazem parte do `Common LISP`. Com elas é possível, *rapidamente*, construir qualquer programa semelhante aos que você pode construir com as *outras linguagens* de programação.
palavras chave: do if let

1.7 O próximo passo na construção

Não tenha dúvida que dificilmente encontrará algum livro fazendo referência às funções de *segunda geração* de LISP, mas é o que representam as funções listadas nas *palavras chave*. Elas foram logo incorporadas aos diversos dialetos LISP e finalmente incluídas na especificação que instituiu `Common LISP`.

`do` é o verbo `fazer`

```
(defun   ListaQuadrados (inicio fim)
         (do ( (i inicio (+ i 1 ) ) )
                                        ((> i fim) 'encerrado )
                                        (format t "~A ~A ~%"i (* i i ) )
         )
)
```

`if` É o condicional `se`. Originalmente não havia esta função e você pode se perguntar como se usava o condicional em LISP. Havia uma forma interessante de fazê-lo, que acho que não interessa mais. `if` funciona semelhantemente ao `C++` (ou vice-versa ...), por exemplo:

`(if 4 5 6) --> 5`

porque `4` é verdadeiro,

`(if () 5 6) --> 6`

porque `()` é falso. Se a seguir ao `if` vier "alguma coisa" que `clisp` avalie como verdadeiro, então o valor será o próximo item. Caso contrário será o terceiro item avaliado. Esta funcionalidade falta ao `C++` que necessita um `else` explicito.

`let` A palavra "let", em inglês, significa "*seja*" que também usamos em Matemática para introduzir símbolos e dar-lhes ao mesmo tempo valores:

Seja $a = 7$ e considere a expressão $f(x) = ax + 5$

é frase comum em qualquer livro de Matemática. Suponha agora que o símbolo x já estivesse associado ao valor 1, em `clisp` teriamos:

```
(setq a 1000 x 10000)
(let ((a 7) (x 3))
     (+ (* a x) 5)
)
--> 26
```

porque dentro do `let` foram criadas variáveis locais `a,x` e o resultado da operação foi aplicado a estas variáveis locais. Verifique que os valores *originais* dados a estes símbolos permanecem.

A sintaxe do `let` é

`(let lista_de_pares expressão_final)` sendo o valor de `expressão_final)` o resultado da avaliação.

Suponha ainda que o símbolo x já estivesse associado ao valor 1 e considere:

`(let ((a 7) (x 9)) (+ (* a x) 5) )` $\longrightarrow 68$

porque agora, dentro do let a variável x foi iniciada com o valor 9. Execute a sequência para compreender como funciona:

```
(setq x 1)

(let ((a 7) ) (+ (* a x) 5) )

(let ((a 7) (x 9)) (+ (* a x) 5) )

x
```

Neste ‘‘ambiente’’ a variável global x está associada ao valor 1, dentro do segundo let x tem um valor local 9 que é o que está sendo usado no cálculo do segundo let.

1.8 Dados, entrada e saída

Nesta seção vou mostrar-lhe como guardar e recuperar dados, a entrada e saída de dados. O objetivo nesta seção será modesto e você poderá aprofundar a questão com auxílio de capítulo a parte que será indicado aqui posteriormente.

Capítulo 2

Controle lógico

2.1 Funções e macros

palavras chave: do, do* dolist, if, let, let* loop,

do

do, que significa faça, em inglês e corresponde ao while de C, Pascal ou python. Mas a sintaxe é bem diferente das funções para laços destas linguagens. Compare com let, e se você cometer um erro provavelmente o interpretador vai lhe dizer que houve um erro de sintaxe do let porque do tem um let implícito (está incluído um let dentro da função do).

Compare com let para melhor entender tanto do como let.

A sintaxe.

(do (lista1 ...listaN) ExpressaoLISP) em que (lista1 ...listaN) é uma lista de listas, podendo ter um único elemento, e cada lista-elemento tem o formato

(variavel valor alteração)

entretanto alteração é opcional, ou seja do é uma generalização de let na macro let as listas-elemento têm apenas dois elementos (variavel valor). A macro do faz um setq de valor em variavel e executa a alteração, se for fornecida. A macro do* executa psetq nas atribuições , ou seja faz as atribuições em paralelo. Os dois exemplos abaixo mostram a diferença entre do, do* que são as mesma que entre let, let*.

Dentro da ExpressaoLISP deve haver um teste obtido por alteração de alguma das variáveis, quando este for teste for T o processamento desta função para. O teste pode ser independente das variáveis sendo processadas, um exemplo abaixo ilustra isto.

Exemplo Execute para ver o resultado:

```
(do
```

```
        ((k 0 (+ k 1)) (soma 0 (+ soma k)))
        ((> k 10) 'done)
        (format t "k=  d → soma = ~d ~%"k soma)
    )

    (defun soma (n)
        (setq Soma 0)
        (do* ((k 0 (+ k 1)) (Soma (+ Soma k) (+ Soma k)))
             ((> k n) 'done) ; quando verdadeiro, para
             ;((> Soma 10) 'done) ; quando verdadeiro, para
             ;((> 1 10) 'done) ; falso, nunca vai parar
             (format t "k=  d soma =  d  %"k Soma)
        )
    )
```

Resulta no erro: variable soma has no value porque a expressão (soma 0 (+ soma k)) foi executada num momento em que soma ainda não tinha recebido um valor. Correção:

```
    (setq soma 0)
    (do
        ((k 0 (+ k 1)) (soma 0 (+ soma k)))
        ((> k 10) 'done)
        (format t "k=  d → soma = ~d ~%"k soma)
    )
```

Observe que foi necessário apenas fazer (setq soma 0) porque do, inicialmente atribui valor às variáveis k, soma, mas, na primeira rodada do laço, soma ainda não tem valor e a expressão (+ soma k) fica sem sentido.

Na próxima seção vou apresentar o do em paralelo, do* que é outra forma de corrigir este processo, porque ele atribui valores, paralamente, as variáveis.

Experimente:

1. pedir os valores de soma, k depois de rodar o código (supondo que estas variáveis ainda não foram definidas anteriormente). Melhor:
   (a) saia de clisp, retorne, execute este código,
   (b) peça os valores de soma, k, antes e depois de rodar o código.
   (c) Entenda o resultado sob o ângulo de *variável local* e *variável global*, k é uma *variável global*.
2. Substitua ((> k n) por ((> 10 1) e rode o código, procure entender o que ocorreu, ou, teste

3. Substitua ((> k n) por ((< 10 1) e rode o código, procure entender o que ocorreu. Se o processamento não parar, use CTRL-C...

Observe a execução passo a passo do código corrigido

1. do executa, inicialmente, (setq k 0 soma 0) e atualiza k soma que passam a valer, respectivamente, 1 , 0. fomat imprime os valores de k, soma;

2. do nas subsequentes ações de do apenas se atualizam k soma que passam a valer, respectivamente, 2,1; 3,2; ...n, n-1;

3. Até que seja verdadeiro (> k 10) do apenas atualiza k soma que passam a valer, k (+ soma k), format imprime os valores de k, soma;

Mas o resultado não é a soma dos números 0, ..., 10 e sim dos números 0, ..., 9. Antes de refazer este exemplo para obter a soma dos números 0, ..., 10, vou analisar a sintaxe da função do agora à luz deste exemplo.

```
(do (       (variável valor atualização ))       (k 0 (+ k
   1) )
      (variável valor atualização )       (soma (+ soma k)
   (+ soma k)
     ExpressaoLISP      o restante
)
```

Um outro exemplo bem simples:

```
(setq x '(1 2 3))
(do ( (n (length x) (- n 1) ) )
     ((= n 1) (format t "~a ~a  % "x n) )
     (format t "~a ~a ~%"x n)
     )
)
```

O (length x) é 3 que é atriuído à n, mas ao mesmo tempo esta definida a condição de atualização de n que é (- n 1) portanto n vai decrescer.
A condição de parada é
((= n 1) (format t "~a ~a % "x n))
e enquanto ela não for atingida a próxima expressão será executada
(format t "~a ~a ~% "x n)
rode para ver ver que a lista (1 2 3) será impressa três vezes e pense por que, compare agora com

```
(setq x '(1 2 3)) - é desnecessário, apague!
(do ( (n (length x) (- n 1) ) )
     ((= n 1) )
     (format t "~a ~a ~%"x n)
     )
)
```

está executando apenas o format que segue à condição de parada enquanto ela não for atingida.

Exemplo O fatorial não recursivo. São definidas duas variáveis

```
(defun fact (n)
      (do
            (
            (i n (- i 1)) ; atribuição e atualização
            (resultado 1 (* resultado i)) ; atribuição e atualização
            ((zerop i) resultado) ; se verdadeiro, para
      )
)
```

do em paralelo

Exemplo Reinicialize clisp e execute para ver o resultado:

```
(setq soma 0)
(do*
      ((k 0 (+ k 1)) (soma (+ soma k) (+ soma k)))
      ((> k 10) 'done)
      (format t "k = ~d soma = ~d ~%"k soma)
)
(format t "~d ~d "soma k)
```

Esta é a função paralell do, ela executa um let paralelo, quer dizer, atribui os valores às variáveis ao mesmo tempo assim como faz a atualização simultaneamente. Quer dizer que k, soma estão sendo atualizadas simultaneamente. No caso anterior, o valor de soma atribuido na posição k do laço, somente seria usado na posição k + 1 do laço.

dotimes

```
(dotimes (var valor-maximo
      valor-de-retorno-opcional)
            expr1
            expr2
      ...)
```

if

Sintaxe

```
(if (predicado)
primeira alternativa      segunda alternativa
)
```

No exemplo primeiro criei uma variável, (setq ava "Ava Garden") para ser usada ao final: (PedeNumero ava), do contrário haveria um erro. Experimente fazer diferente e tente entender o que acontece.

Depois defini a função PedeNumero. Se a execução for (PedeNumero 3) a primeira alternativa do if será executada, experimente. Se você não fornecer um número será executada segunda alternativa apenas informando que um número será pedido. Raspe e cole o texto do programa no terminal do clisp que ele irá funcionar. Altere-o para fazer suas experiências e entender como funciona.

```
(setq ava "Ava Garden") ; atribuição global
(defun PedeNumero (valor)
     (if (numberp valor)
               valor
               (print "Vou pedir-lhe um número ")
     )
     (print "Um número ")(setq valor (read))
     (format t "o número é  d "valor)
)
(PedeNumero ava)
```

Ver let, read

let

let é uma *macro* cuja sintaxe é:

```
(let      ((      variavel      valor))
     (funcao )
)
```

Ela lhe permite definir diversos pares (variavel valor) e depois executa alguma expressão associada à variavel.

```
(defmacro seja let);; traduzindo let
(defun teste ()
     (seja ((variavel (read)))
          (if (numberp variavel)
               variavel
          (ask-number)
          )
     )
)
```

Ver if, read

print

Para imprimir texto. Alternativas princ, format, terpri, prin1

read

```
(defun PedeNumero ()
      (let ((variavel (read)))
      (if (numberp variavel)
            variavel
            (PedeNumero)
      )
      )
)
```

Ver if, let

Pequenos programas exemplos

Primeiro substitua let por seja:
(defmacro seja let)
Na verdade não é uma substituição e sim uma nova definição, você pode usar seja em lugar de let mas pode continuar usando let.

```
(defun PedeNumero ()
      (let ((variavel 0))
            (format t "Forneça-me um número ")
            (setq variavel (read))
            (if (numberp variavel) variavel (PedeNumero))
      )
)
```

Observe que variavel é uma *variável local* da função PedeNumero isto significa que ao terminar a execução desta função não haverá rastro desta variável no *sistema* clisp. Se você precisar de criar uma variável usando este programa deverá executar:
(setq UmValor (PedeNumero))
Não se esqueça de executar a definição PedeNumero, caso contrário LISP vai lhe dizer que não existe nada com este nome na memória.
Depois do que UmValor é um novo símbolo do LISP contendo o símbolo, variavel, que você tiver fornecido. Experimente!

2.2

dolist
É uma função que itera usando uma lista como um stack, sintaxe:
tt (dolist (var lista resultado) expressão)
dolist evalua lista cujo resultado deve ser uma lista, então, sequencialmente

- var assume o valor (car lista);

- expressão é executada;
- lista é substituída por (cdr lista) até que (cdr lista) se torne nil quando resultado é devolvido como valor de dolist

Por exemplo,
(setq x '(1 2 3))
(dolist (y x x) (print x))
irá imprimir a lista (1 2 3) três vezes usando-a como stack de onde saem os valores para y, quando o stack ficar vazio ira devolver o resultado x. Experimente! e verique que internamente o que acontece é que clisp cria um stack temporário igual a lista x sendo este stack que vai ser usado para controlar a iteração portanto x fica intacto e pode ser usado na expressão, aqui eu usei a expressão (print x) .

Capítulo 3

Função, macros, operadores

palavras chave: atom, cond, declare, defun,, floatp, integerp, length, list, listp, member, not, numberp, parametros inicializdos, setq, sort, stringp, symbolp symbol-plist type-of,

3.1 funções de Common LISP

Estou apresentando na próxima seção as funções aritméticas, separadamente, até porque elas são aquilo que esperamos apenas dentro da sintaxe de LISP.

3.1.1 atom

É uma *função predicado* retornando t, NIL se o parâmetro que lhe for for passado for um átomo ou não.
(atom 3) *rightarrow* t
(setq L '(1 2 3 4 5 6)
(atom L) *rightarrow* NIL
Observe que a função-predicado para testar se um objeto é uma lista é LISP e não list. Estas duas funções estão descritas em seguida.

3.1.2 cond

Esta função corresponde ao case de C++, sintaxe:

```
(cond
     sentença1
     sentença2
     sentença3
)
```

sendo sentença1, sentença2, sentença3 ... um número arbitrário de expressões LISP da forma (obj_j L_j). cond seleciona a primeira expressão cujo car seja verdadeiro devolvendo o valor desta sentença:

```
(setq R ())
(cond
      (R "não vazia")
      (t "vazia")
      (t "será ignorada")
)
```

resulta em ‘‘vazia’’.
ou

```
(setq R '(1 2 3 4 5 6 7 ))
(cond
      (R "não vazia")
      (t "vazia")
      (t "será ignorada")
)
```

resulta em ‘‘não vazia’’.

Nos dois exemplos acima, como uma das duas primeiras expressões tem car verdadeiro, então a terceira expressão será ignorada, como seria o caso de mais expressões que houvessem para ser avaliadas. Será avaliada apaenas a primeira expressão verdadeira.

Também cond aceita um *valor default*, se uma lista de setenças lhe forem passadas e todas forem falsas, a última, será avaliada e o seu valor será retornado. Dois exemplos:

```
(cond
      (() "não vazia")
      (() "vazia")
      (() "será ignorada")
      (3)
)
```

resulta em 3 porque a última expressão, dentro da lista, tem valor 3.

```
(cond
      (() "não vazia")
      (() "vazia")
      (() "será ignorada")
      (() 3)
)
```

resulta NIL, porque o car da última expressão foi avaliado tendo como resultado NIL.

3.1.3 defun

Esta macro define uma função. Sintaxe

```
(defun nome (ListaParametros)
     CorpoDaFunção
)
```

Um exemplo:

```
(defun f (x y)
     (+ (* x x) (* y y) (* 3 x y) )
)
```

define a função $f(x,y) = x^2 + 3xy + y^2$

Podemos definir esta função *anonimamente* ou, com uma *expressão-lambda* o que é algumas vezes é prático fazer mas é também um indicativo de que o código poderia ser quebrado em pedaços menores. De qualquer forma as *expressões-lambda* são históricamente muito importantes e servem de comparativo para entender melhor a sintaxe de uma função:

```
(lambda (x y)
     (+ (* x x) (* y y) (* 3 x y) )
)
```

Experimente raspar e colar no clisp

```
((lambda (x y) (+ (* x x) (* y y) (* 3 x y) )) 3 4)
```

Observe que os valores $3, 4$ estão sendo passados para a *expressão-lambda* envoltos em novo parêntesis. Isto seria o equivalente a (f 3 4), em que f é a *expressão-lambda*.

Uma forma muito comum e prática de uso para *expressão-lambda* ocorre quando precisamos trabalhar com uma função-parcial - eliminando uma das variáveis de uma função multivariada. Acima você tem o exemplo de uma função bivariada e dentro de um programa poderia ser necessário calcular (f x 4), em que a outra variiável está recebendo o valor constante y = 4, e *expressão-lambda* entra bem neste caso:

```
(lambda (x) (f x 4) )
((lambda (x) (f x 4) ) 3)
```

e a *expressão-lambda* definiu uma função-parcial de f em que o parâmetro y é mantido constante valendo 4.

```
(loop
```

errado!

```
(do* (
(x 0) (+ x 1)
(soma ((lambda (z) (f z 4)) x) )
(if (> x 10) soma)
)
)

(defun f (x y) (+ x y))

(do* (
      (x 0) (+ x 1)
            (soma ((lambda (z) (f z 4)) x) )
      )
      (if (> x 10) soma)
      )
)
```

3.1.4 length

Se L for uma lista, (length L) retorna o número de elementos de L.

3.1.5 list

Constrói uma lista usando o seus parâmetros. Observe que será preciso setq em algum símbolo.
   (list 1 2 3 4 5 6) *rightarrow* (1 2 3 4 5 6)
porém nada foi criado, entretanto
   (setq L (list 1 2 3 4 5 6)) *rightarrow* (1 2 3 4 5 6)
   L *rightarrow* (1 2 3 4 5 6)

3.1.6 listp

É a ***função predicado*** que testa se algum objeto é (ou não) uma lista.
   (setq L (list 1 2 3 4 5 6))
   (listp L) *rightarrow* t

3.1.7 not

not recebe um argumento e inverte a verdade deste argumento.
   (setq L '(1 2 3 4 5)
   (not L) *rightarrow* NIL
porque L não é NIL.
   (not ()) *rightarrow* t

3.1.8 numberp

É um predicado e testa se o argumento é um número:
(numberp 3.1415) *rightarrow* t
(setq L '(1 2 3 4 5 6 7 8)
(numberp L) *rightarrow* NIL
(setq um 1 dois 2 tres 3 quatro 4 cinco 5 seis 6 sete 7 oito 8)
(numberp tres) *rightarrow* t

3.1.9 member

Esta função recebe dois parâmetros
(member x y)
verifica se y não for uma lista se produz um erro, e se x for um elemento da lista y devolve o (cdr y) iniciando com x.
Desta forma (car (member x L)) sendo verdadeiro vale x.

3.1.10 setq

Esta função atribui valores a um símbolo e também cria o simbolo. Se o símbolo qwert não exitir, isto é (symbolp qwert) for NIL então (setq qwert 'algo) coloca o valor algo em qwert. A partir de então
(symbolp qwert) *rightarrow* t
Se o símbolo qwert exitir, então (setq qwert 'algo) destrutivamente coloca o valor algo em qwert, ou ainda, substitui o valor antigo.

3.1.11 sort

Esta função recebe dois parâmetros, uma lista L e um atributo de *desigualdade* e arranja os elementos da lista L obedecendo a ordem definida pelo atributo.
(setq L '(7 1 10 5 3 2 4 6 8 9))
(sort L '<) *rightarrow* (1 2 3 4 5 6 7 8 9 10)
e L terá sido alterada ficando ordenada de acordo com o atributo de ordem fornecido.

3.1.12 symbolp

Esta função testa se o seu argumento é um símbolo da linguagem (se é um nome legal para guardar uma valor).

3.1.13 symbol-plist

Esta função descreve o conteúdo das células de um símbolo.
Em LISP a todo símbolo está associado uma lista de propriedades que uma lista de associações com elementos no formato
chave valor

Experimente
(symbol-plist car)
que vai dar erro... experimente então
(symbol-plist 'car)
e você vai poder ver os registros de mémória associados a este função da linguagem. Em LISP isto é muito extenso do que a função que pode examinar o registro na memória associado a um símbolo de uma linguagem de programação tradicional (todas tem esta associação, faz parte dos dados da tabela de alocação de memória que a linguagem recebe do sistema operacional quando é posta em uso). Mas LISP cria a sua própria tabela e a coloca como valor de cada símbolo. Se você tiver definido fatorial, experimente
(symbol-plist 'fatorial)
e verá a definição colocada como uma das propriedades deste símbolo.
Se você tiver definido fatorial experimente: (ou primeiro defina fatorial)

```
(setq teste (symbol-plist 'fatorial)
(car teste)
(nth 0 teste)
(nth 1 teste)
(car (nth 1 teste))
(nth 0 (nth 1 teste))
```

3.2 As funções aritméticas

palavras chave: +, *, /, -, >, <, and, complexp, evenp, minusp, oddp, or, zerop,

3.3 Parâmetros inicializados

Um exemplo não funciona como eu esperava!

```
(defun f ((x 2) (y 3) (z 4))
     (+ (* x y z) 1)
)
```

Esta função tem três variáveis (parâmetros) que tem um valor inicial, respectivamente, 2,3,4. Não funciona!

Capítulo 4

Entrada e saída de dados

Entrada e saída de dados é um assunto muito extenso, para começar depende do que você precisa como programa. Você vai encontrar aqui a informação básica que você precisará completar frente às suas necessidades.
palavras chave: format, load, read, read-line,

4.1 A função `format`

Se você usou fprintf() da linguagem C vai achar que a semelhança é grande entre a sintaxe da função fprintf() e format em clisp. Mas você não precisará aprender C para entender LISP, o contrário poderia ser verdade... estou apenas comparando funções parecidas em duas linguagens diferentes.

A forma mais simples de usar é

```
(format t "Escrevendo alguma coisa")
```

a função format serve para formatar a saída de dados, e o primeiro parâmetro, t indica onde os demais parâmetros será impressos.

format recebe um número qualquer de parâmetros divididos em três classes: (format t máscara p1 p2 ...) assim descritas:

1. para onde vão os dados, o primeiro parâmetro deve ser t caso você deseje imprimir no dispositivo padrão que em geral é tela do computador. Não sendo este o caso você deve indicar para onde deve ir a impressão, por exemplo, um arquivo anteriormente definido, exemplos vão lhe mostrar como fazer.

2. O que vai ser impresso

   - A máscara é semelhante ao caso de C, ou python, um texto que será impresso, observe, entre aspas. Dentro *deste texto*, que estou chamando de ‘‘*mascara*’’, você pode escrever uma frase ou várias frases, dentro do qual se encontram macros

que serão substituídas, na mesma ordem em que ocorram, pelos parâmetros, p1 p2 ..., que representam a *terceira classe de parâmetros*.
Experimente o exemplo, raspe e cole no terminal do clisp,

```
(format t
"Vai imprimir ~A linhas de texto, ~% trocar de linha aqui,
mas observe que neste caso haverá ~A, apenas dois elementos,
na lista de parâmetros, ~% (outra mundança de linha) as mudanças
de linha não contam."
2 2)
```

O resultado será:

```
Vai imprimir 2 linhas de texto,
trocar de linha aqui, mas observe que neste caso haverá 2, apenas dois elementos,
na lista de parâmetros,
(outra mundança de linha) as mudanças de linha não contam.
```

Possivelmente clisp imprime o texto em dobro, depois você irá aprender como evitar isto.

- Em C a mudança de linha ocorre quando houver a presença do símbolo \n na máscara, aqui é o símbolo ~%, observe a diferença com os outros símbolos, este não exige nenhum parâmetro na lista de parâmetro e não será contado, mas será executado. Altere o exemplo e volte a rodá-lo para entender como funciona.
- Use ~d para que indicar um inteiro e ~f para indicar *ponto flutuante*.

3. a lista final de parâmetros é uma lista ordenada[1] que será utilizada para preencher as macros usadas na ‘‘máscara’’ descrita acima: p1 p2 .... Se não houver dados suficientes, clisp irá reclamar.

4.1.1 Função imprimir alguma coisa

Vou dar-lhe dois exemplos de função que você precisará com frequência:

1. **Uma função que escreva “qualquer coisa”.**

```
(defun Escreva (UmTexto)
     (format t UmTexto)
)
```

Experimente: (Escreva "Eu escrevi este livro") , obviamente, primeiro defina a função no terminal do clisp.

Depois, dentro de um programa, quando quiser escrever qualquer texto, use (Escreva) fornecendo-lhe o texto adequado.

[1] Pleonasmo, toda *lista* em LISP é uma lista ordenada...

2. **Uma função *boba* de grande utilidade**. Você vai precisar de usar o texto

   (Escreva "Aperte uma tecla para continuar").

   Claro, como esta expressão é muito importante em programas, é interessante ter uma função mais adequada para escrevê-la:

```
(defun Apeteco ()
     (format t "Aperte uma tecla para continuar ~% ")
)
```

   e para chamá-la basta escrever (Apeteco), sem nenhum parãmetro. Observe a diferença com C em que seria preciso chamar a função Apeteco(), em LISP, você não pode fazer assim, experimente e verá que o interpretador reclama que há parãmetro em excesso, porque () é um objeto em LISP, portanto a expressão (Apeteco ()) significa que você está passando o parâmetro () para a função (Apeteco) e neste caso eu defini uma função sem parâmetros usando a *lista vazia* de parâmetros.

4.1.2 Carregando funções definidas

Duas operações são muito importantes: gravar e recuperar.

Gravar os dados, as funções, as listas, que você tiver construído durante o seu trabalho. e recuperar este trabalho gravado em disco para colocá-lo na memória do clisp.

Se você tiver produzido o arquivo (êle existe) com funções para usar gnuplo, por exemplo, gnuplot.lsp execute no terminal do clisp

```
(load "gnuplot.lsp")
```

e todo o seu trabalho contido neste arquivo estará de volta dentro da memória do clisp.

4.2 A função `read`

Uma função para *entrada de dados* muito genêrica é (read) que serve para tudo, desde leitura de caracteres pelo teclado até leitura de arquivos. Mas ela tem variantes que serve para usos específicos que vou discutir aqui.

Antes de mais nada observar um erro que você pode cometer numa função que chame (read), usá-la sem indicação de que espera um dado. É preciso sempre usar (read) com format dizendo o que se espera que seja fornecido, caso contrário a função em execução vai parecer travada...mas não estará, apenas (read) esta esperando dados. Experimente o exemplo, mas não fique molestada com o resultado, logo vou explicar-lhe como é interessante este exemplo. Digite (ou raspe e copie) no terminal do clisp e depois rode (PedeNumero).

```
(defun PedeNumero ()
  (format t "Um número, por favor. ")
  (let ((val (read)))
    (if (numberp val)
        val
        (PedeNumero))))
```

Se você tiver fornecido qualquer outra coisa diferente de número, a função (PedeNumero) insiste: Um número, por favor. Se fornecer um número ela para, fez o que era para fazer.

Apenas você terá feito trabalho inútil, use um setq para guardar o resultado na variavel que desejar:

```
(setq a (PedeNumero))
```

e depois verifique

```
a
```

e vai ver que clisp guardou o resultado na variável a. Aqui você tem um exemplo de função lhe será muito útil, você poderá com ela preencher toda uma matriz de números, apenas deverá aprender a chamá-la de dentro de outra função que irá lançar as leituras na matriz...

Mas o que eu queria chamar sua atenção é para o seguinte exemplo:

```
(defun PedeNumero ()
  (let ((val (read)))
    (if (numberp val)
        val
        (PedeNumero))))
```

e você verá o terminal do clisp mudo como se estivesse travado. Evite este erro.

```
(defun PedeNome ()
  (format t "Escreva-me o seu nome, por favor. ")
  (let ((val (read)))
    (if (not (numberp val))
        val
        (PedeNome))))
```

Novamente, se você não tiver usado um setq terá sido trabalho em vão.

Também é possível que você cometa outro tipo de erro: fornecer todo o seu nome sem o colocar entre aspas. (PedeNome) vai ler o primeiro átomo do seu nome e depois clisp irá reclamar que encontrou variáveis que não estão definidas. Experimente para entender como funciona. Forneça um número, em vez de um nome.

Aqui entra a função (read-line) que é uma variante de (read) para ler frases, irá ler todo o conteúdo de uma linha até o sinal de fim de linha. Novamente, se você não tiver usado um setq terá sido trabalho em vão.

```
(defun PedeNome ()
  (format t "Escreva-me o seu nome, por favor. ")
  (let ((val (read-line)))
    (if (not (numberp val))
        val
        (PedeNome))))
```

Referências Bibliográficas

[1] Foundation for Free Software. Gpl - general public license. Technical report, http://www.FSF.org, 2011.

[2] Paul Graham. *ANSI Common LISP*. Prentice Hall, 1996.

[3] Colin Kelley Thomas Williams and many others. gnuplot, software to make graphics. Technical report, http://www.gnuplot.info, 2010.

Índice Remissivo