Computabilidade

José Carlos Bins Filho

5 de Dezembro de 2013

Conteúdo

1. Funções Recursivas
   1. Funções recursivas de Kleene
   2. Cálculo Lambda
   3. Linguagem Lisp
2. Máquinas Universais
   1. Modelos formais de computação
   2. Máquinas de Turing
   3. Máquinas de Post
   4. Modelo RAM
   5. Equivalência e simulação
   6. Máquina de Turing Universal
   7. Tese de ChurchTuring

Conteúdo (continuação)

1. Indecidibilidade
   1. Classes de problemas
   2. Redução de problemas
   3. Problemas decidíveis, semi-decidíveis e indecidíveis
   4. Teorema da incompletude de Gödel
2. Intratabilidade
   1. Redução polinomial de problemas
   2. Classes de problemas P, NP, NPCompleto e NPDifícil
   3. Teorema de Cook
   4. Problemas intratáveis em grafos
   5. Algoritmos aproximativos e heurísticas

Avaliação

- $MF = \frac{2.5 \times P_1 + 2.5 \times P_2 + 5.0 \times A}{10.0} \geq 6.0$
- Prova Substitutiva no final do semestre para recuperar a nota de 1 prova.
- Datas:
  - Prova 1: 29/01/2014
  - Prova 2: 14/03/2014
  - Prova Sub: 28/03/2014

Bibliografia

- Básica:
  - SIPSER, Michael. Introdução à Teoria da Computação. 2a ed., Thomson, 2007.
  - HOPCROFT, J. E.; ULLMAN, J. D.; MOTWANI, R.. Introdução à Teoria dos Autômatos, Linguagens e Computação. Campus, 2002.
  - CARNIELLI, Walter; EPSTEIN, Richard L.. Computabilidade, Funções Computáveis, Lógica e os Fundamentos da Matemática. São Paulo, Editora Unesp, 2006.

Bibliografia

- Complementar:
  - CORMEN, Thomas H.; MATOS, Jussara Pimenta (Rev.). Algoritmos: teoria e prática. Rio de Janeiro, Campus, 2002.
  - LEWIS, Harry R.. Elementos de Teoria da Computação. 2a ed., Porto Alegre, Bookman, 2000.
  - GAREY, Michael R.; JOHNSON, David S.. Computers and Intractability: a guide to the theory of NP-Completeness. New York, W. H. Freeman, 2003.
  - PAPADIMITRIOU, Christos H.. Computational Complexity. Massachusetts, Addison-Wesley, 1995.
  - VIEIRA, Newton José. Introdução aos Fundamentos da Computação: linguagens e máquinas. São Paulo, Thomson, 2006.

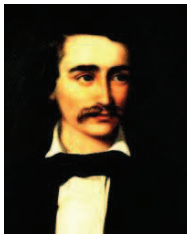
Computabilidade

- O que é computabilidade?
  - Existem dois meios possíveis de definir computabilidade:
    - Intuitivamente: Tudo que pode ser computado (calculado)(por qualquer meio)
    - Formalmente: Tudo que pode ser computado por um método/ou máquina formal.
  - Estes dois conceitos são unidos pela tese de Church que diz que tudo que pode ser computado (definição intuitiva) pode ser computado por uma máquina de Turing (definição formal)
  - Não há como provar a tese de Church (pelo menos ninguém tem idéia de como fazer isto) justamente porque a definição informal é informal e portanto nenhum método de prova (formal) pode trabalhar com ela.
  - A computação em cada um dos modelos formais (Máquina de Turing, Funções recursivas, Gramáticas, etc...) implementa uma noção do que vem a ser um procedimento efetivo, i.e. uma regra mecânica, ou um método automático, ou um programa para executar alguma operação matemática.

Histórico

- As funções recursivas são estudadas a pelo menos 100 anos.
- Elas começaram com estudos dos números inteiros com Peano e Dedekind nas décadas de 1890/90.

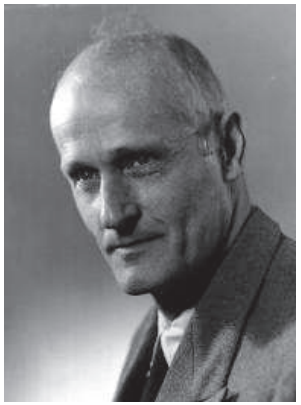
Giuseppe Peano (1858 - 1932)



Richard Dedekind (1831 -1916)

Histórico

- Propôs, em 1936, o uso de funções parciais como forma de formalização de função computavel
- Mais recentemente Kleene em 1938 provou dois importantes teoremas
  - Primeiro teorema da recursão de Kleene: Teorema da recursão fraco
  - Segundo teorema da recursão de Kleene: Teorema da recursão forte
- Ambos são teoremas de ponto fixo para funções recursivas



Stephen Kleene (1909 – 1994)

Tipos de Funções

- **Definição:** Função total: Uma função é dita total se está definida para todos os elementos do domínio.
  Ou seja: $\forall x \in D, \exists y \in I \mid f(x) = y$
  onde D é o Domínio da Função e I é a Imagem da Função
  - Exemplo: $f : \mathbb{R}_+ \to \mathbb{R}_+$ onde $f(x) = \sqrt{x}$
- **Definição:** Função parcial: Uma função é dita parcial se está definida para alguns elementos do domínio.
  Ou seja: $\exists x \in D \mid \exists y \in I \ \wedge \ f(x) = y$
  - Exemplo: $f : \mathbb{Z}_+ \to \mathbb{Z}_+$ onde $f(x) = \sqrt{x}$
- Obs: Toda função total é também função parcial, mas o contrário não é verdadeiro

Tipos de Funções

- A importância da classificação de função em parcial e/ou total é que alguns teoremas e/ou definições se aplicam a apenas um dos dois tipos
- Por Exemplo:
  - Uma função é Turing-computavel ou recursivamente enumerável se, e somente se, ela for uma função recursiva parcial
  - Uma função é Turing-computavel pro uma máquina que sempre pára, ou recursiva, se, e somente se, ela for uma função recursiva total
- As funções recursivas parciais foram introduzidas por Kleene, em 1936, com o objetivo de formalizar a noção intuitiva de função computável.

Introdução

- A teoria de funções recursivas foi criada para capturar a idéia de efetividade, portanto é importante que a classe de funções recursivas seja construída a partir de funções simples.
- Existem varias opções de conjuntos de funções elementares com os quais construir a ideia de funções recursivas.
- Um conjunto possível é:
  - Função constante zero: $Zero(x) = 0$
  - Função sucessor: $Suc(x) =$ sucessor de x
  - Função identidade: $Ident(x) = x$

Introdução

- Agora são necessárias operações para a construção de novas e mais complexas funções.
- Um conjunto mínimo de operações necessárias e suficientes para compor todas as outras funções é:
  - Composição
  - Recursão primitiva
  - Minimização
- **Definição:** Funções recursivas primitivas: Uma função é chamada Função Recursiva Primitiva se ela pode ser obtida a partir das funções primitivas através de um número finito de aplicações de composição e recursão.
- Pode ser bastante trabalhoso verificar se uma função é recursiva primitiva.

Introdução

- Existem funções que embora sejam computáveis (ou seja, sabemos construir um programa para calcular o seu valor para quaisquer números naturais) não são recursivas primitivas.

  Exemplo: função de Ackermann : existem várias versões, uma das possíveis é:

$$A(x,y) = \begin{cases} y+1 & se\ x = 0 \\ A(x-1,1) & se\ x > 0 \wedge y = 0 \\ A(x-1,A(x,y-1)) & se\ x > 0 \wedge y > 0 \end{cases}$$

  Função com crescimento muito rápido e extrema recursão.
  Exemplo: $A(1,2) = 4; A(2,2) = 7; A(3,2) = 29; A(4,2) \approx 10^{80}$

Composição

- Compor duas funções é aplicar o resultado de uma como argumento da outra.
- $A \circ B = A(B(x))$
- Exemplo:
  - Começando com a função sucessor e substituindo o argumento pela função zero (assumindo que estejamos trabalhando no conjunto dos inteiros): $Suc(x_1) \circ Zero(x_2)$
    $Suc(Zero(x)) = 1$
  - Compondo com a função sucessor novamente:
    $Suc(x_1) \circ Suc(x_2) \circ Zero(x_3)$
    $Suc(Suc(Zero(x))) = 2$, e assim por diante.
  - Isto nos permite definir a função que devolve os números naturais.
  - Além disto podemos definir outras funções mais complexas. Por exemplo, se quizermos definir uma função $f(x) = x + 2$ podemos definir como $f(x) = Suc(Suc(x))$
  - Exercício: Como representar a função $f(x) = x - 1$

Recursão primitiva

- Quando uma função é definida usando a própria função.

$$\begin{aligned} &h(x,Zero(x)) = f(x) \\ &h(x,Suc(y)) = g(x,h(x,y)) \end{aligned}$$

- Neste caso, quando o segundo argumento da função é zero[1] ele executa a função $f(x)$, quando é diferente de zero ele executa $h(x,h(x,y))$, onde y é o antecessor de $Suc(y)$ (ou seja $Suc(y) = y + 1$)
- Exercício: Defina as funções abaixo:
  - soma(x, y) = x + y
  - dobro(x) = 2x
  - produto(x,y) = x . y
  - fatorial(x) = x!

[1] A partir de agora vamos simplificar a notação e representar números com numerais em vez de funções. Portanto escreveremos o valor zero como '0' em vez de 'zero(x)'.

Recursão primitiva

- soma(x, y) = x + y

  soma(x,0) = Ident(x)
  soma(x,Suc(y)) = g(x,soma(x,y)
  g(x, y) = Suc(y)

- Exemplo de 2 + 3

  $soma(2,3) = g(2, soma(2,2)) = g(2,4) = Suc(4) = 5$
  $soma(2,2) = g(2, soma(2,1)) = g(2,3) = Suc(3) = 4$
  $soma(2,1) = g(2, soma(2,0)) = g(2,2) = Suc(2) = 3$
  $soma(2,0) = Ident(2) = 2$

Recursão primitiva

- dobro(x) = 2x = x + x

  dobro(x) = soma(x,x)

Classes de Funções

- Classe das funções Primitivas fechadas para recursão (PRC: Primitive Recurvively Closed)
- Uma classe de funções totais $\mathcal{C}$ é chamada de PRC se:
  - As funções primitivas pertencem a $\mathcal{C}$
  - Qualquer funções obtida a partir de funções de $\mathcal{C}$ a partir de composição e recursão também pertence a $\mathcal{C}$
- Teorema: A classe das funções computáveis é uma PRC
- Corolário: A classe das Funções recursivas primitivas é uma PRC.

Funções primitivas Recursivas

- Diga se as funções abaixo são primitivas recursivas e caso positivo mostre porquê.
  - $x \dot{-} y = \begin{cases} x - y & if\ x \geq y \\ 0 & if\ x < y \end{cases}$
  - $|x - y|$
  - $\alpha(x) = \begin{cases} 1 & if\ x = 0 \\ 0 & if\ x \neq 0 \end{cases}$
  - $igual(x,y) = \begin{cases} 1 & if\ x = y \\ 0 & if\ x \neq y \end{cases}$
  - $x \leq y$
  - Solução: $x \leq y = \alpha(x \dot{-} y)$

Funções primitivas Recursivas

- Conectivos lógicos são PRC.
- Teorema: Dado $\mathcal{C}$ uma PRC. Se P e Q são predicados que pertencem a $\mathcal{C}$, então $\sim P$, $P \vee Q$, $P \wedge Q$ também pertencem a classe $\mathcal{C}$
- Prova:
  - $\sim P = \alpha(P)$
  - $P \wedge Q = produto(P, Q)$
  - $P \vee Q = \sim (\sim P \wedge \sim Q)$
- Corolário: Se P e Q são predicados computáveis então então $\sim P$, $P \vee Q$, $P \wedge Q$ também são computáveis.
- Exercício: Mostre que $x < y$ é primitivo recursivo.
- Solução: $x < y = \sim (y \leq x)$

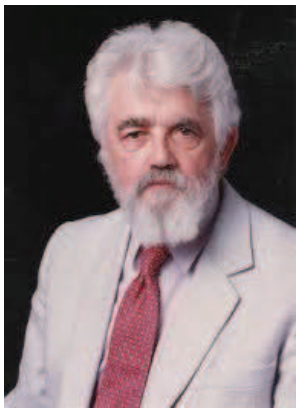
História

- Formalismo proposto por Church na década de 1930
- Introduzido para dar uma fundação funcional para a matemática, mas os matemáticos preferiram usar a teoria de conjuntos como esta base.
- Baseado nos conceitos de
  - definição de função
  - aplicação de função
- Linguagem Universal
- Inspiração para as linguagens funcionais

Alonzo Church (1903-1995)

História

- Lambda Calculus ($\lambda - calculus$) é uma notação para funções arbitrárias
- O cálculo - λ foi redescoberto como uma ferramenta na década de 50 e 60.
  - John Macharty - década de 50: Linguagem Lisp
  - Peter Landin - década de 60: observou que uma linguagem de programação pode ser compreendida formulando-a em um pequeno núcleo (cálculo - λ) capturando suas características essenciais
- O cálculo λ é universal e portanto equivalente a uma Máquina de Turing.



John McCarthy (1927-2011)

Conceitos básicos

- Abstração funcional (ou procedural) é freqüente em praticamente todas as linguagens de programação
  - A seqüência de computação repetitiva abaixo

$$(5 \times 4 \times 3 \times 2 \times 1) + (4 \times 3 \times 2 \times 1) - (3 \times 2 \times 1)$$

pode ser reescrita na forma:

$$fatorial(5) + fatorial(4) - fatorial(3)$$

onde: fatorial(n) = if n = 0 then 1 else n $\times$ fatorial($n - 1$)

- Escrevendo λn no lugar de "A função que, para n, retorna ..." temos: fatorial(n) = λn.if n = 0 then 1 else n $\times$ fatorial($n - 1$)

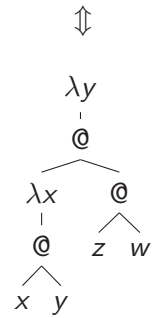
Conceitos básicos

- **Definição:** Variaveis: $x, y, z, \ldots \in Var$ Var é um conjunto finito de nomes
- **Definição:** Aplicação: $M\ N$ significa: chame função M passando N como parâmetro. Tambem escrita como: $@(M, N)$
- **Definição:** Abstração Lambda: $\lambda x.M$ significa: função que recebe um argumento, referenciado como x, e retorna a expressão M.
- As únicos símbolos reservados da linguagem são o λ e o ponto (".").

Conceitos básicos

- Escopo: O escopo de uma abstração lambda extende-se para a direita
  Ex: $\lambda x.xy = \lambda x.(xy) \neq (\lambda x.x)y$
- Operador de aplicação '@' é associativo à esquerda
  Ex: $M\ N\ P$ significa $(MN)P \neq M(NP)$ ou usando o operador de aplicação explicitamente $@(@(M, N), P)$
- Abreviatura
  $\lambda x.\lambda y.\lambda z.M$ pode ser abreviado para $\lambda xyz.M$
  Observação: a ordem de substituição é $x \to y \to z$

$\lambda y.(\lambda x.x\ y)(z\ w)$



Conceitos básicos

- Variável livre x variável dependente (bounded)
  - No cálculo λ os nomes são locais à definição.
  - Nomes não precedidos do λ são chamadas de variáveis livres e os precedidos são chamadas de variáveis dependentes ("bounded").
    Ex: $\lambda x.xy$
    x é a variável dependente e y é a variável livre
  - Uma variável pode ser dependente e livre na mesma expressão
    Ex: $(\lambda x.x)(\lambda y.xy)$
    x é dependente na primeira sub-expressão e livre na segunda sub-expressão

Conceitos básicos

- Variáveis livres e dependentes tem significados distintos.
  - Variáveis livres são nomes globais, são importantes

$$\text{expressões distintas} \begin{cases} \sin(\pi) - 42 + \pi^2 \\ \sin(e) - 42 + e^2 \end{cases}$$

  - Variáveis dependentes são apenas nomes usados para substituição, não são importantes

$$\text{mesma expressão} \begin{cases} f(x) = \sin(x) - 42 + x^2 \\ f(e) = \sin(e) - 42 + e^2 \end{cases}$$

Conceitos básicos

- No calculo lambda a maneira de efetuar cálculos é através da operação de redução
- Uma redução nada mais é que a substituição da variável dependente por uma expressão
- **Exemplo:** $\lambda n.n$ é a função identidade
  - $(\lambda n.n)\ 5 = 5$
  - $(\lambda n.n)\ (x+1) = x+1$
  - $(\lambda n.n)\ (n+1) = n+1$
- Seria de esperar que após um determinado número de reduções se chegaria a uma forma para a qual não fossem possíveis realizar mais reduções. No entanto isto não é verdadeiro. Um exemplo é dado abaixo.
  $(\lambda x.xx)(\lambda x.xx)$
- Quando uma sequência de reduções foram efetuadas e nenhuma redução é mais possível então a expressão restante é dita na sua forma normal. Nem toda expressão tem forma normal.

Conceitos básicos

- Para simplificar os termos lambda muitas vezes usamos nomes representando expressões lambda
- **Exemplo:** 1. $\mathbf{I} = \lambda x.x$ para a Identidade
  Com isto podemos usar: $\mathbf{I}\ 5$ em vez de $(\lambda x.x)5$
- Por enquanto usaremos algumas funções sem definição, mais tarde estas funções serão definidas
- **Exemplo:** 2. $(add\ n\ m)$ é a função que soma dois números
- Com isto podemos definir termos lambda como:
  - $(\lambda n.\ (add\ n\ 1))\ 5 = 6$
  - $(\lambda x.\ (add\ x\ 2))\ ((\lambda n.\ (add\ n\ 1))\ 5) = (\lambda x.\ (add\ x\ 2))\ 6 = 8$
  - $(\lambda x.\ (add\ n\ 2))\ ((\lambda n.\ (add\ n\ 1))\ 5) = (\lambda x.\ (add\ n\ 2))\ 6 = add\ n\ 2$
- Outro exemplo usando *sqr* (raiz quadrada):
- **Exemplo:** 3. $\lambda\ f.(\lambda\ x.(f(f\ x)))$ é uma função com dois argumentos. Uma função e um valor.
  - $(\lambda\ f.(\lambda\ x.(f(f\ x))))sqr\ 3 = (\lambda\ x.(sqr(sqr\ x)))3 = (sqr(sqr\ 3)) = (sqr\ 9) = 81$

Conceitos básicos

- Os dois mais confusos aspectos do cálculo λ são que os nomes das variáveis dependentes não tem significado e que a ordem de aplicação pode ser confusa
- **Exemplo:** 1.
  - Aplicando a função identidade sobre ela mesma
  - $(\lambda x.x)(\lambda x.x)$
    Observação: a variável x da primeira sub-expressão não tem nada a ver com a variável x da segunda subexpressão portanto podemos reescrever a expressão

    $$(\lambda x.x)(\lambda x.x) = (\lambda x.x)(\lambda y.y) = \lambda y.y$$

    Ou seja a função identidade aplicada sobre ela mesma dá a própria identidade.
  - $\lambda x.x = \lambda y.y = \lambda t.t = \lambda u.u$

Conceitos básicos

- **Exemplo:** 2.
  - Definindo: $Q = \lambda x.*\ x\ x$ função que faz quadrado de um termo usando a função produto
  - O que é : $\lambda x.Q(Q(Q\ x))$: função que calcula x^8
  - redução:

$$\begin{aligned}(\lambda x.Q(Q(Qx)))2 &= Q(Q(Q\ 2) = (\lambda x.*\ x\ x)((\lambda y.*\ y\ y)((\lambda z.*\ z\ z)\ 2))\\ &= (\lambda x.*\ x\ x)((\lambda y.*\ y\ y)(*2\ 2)) = (\lambda x.*\ x\ x)((\lambda y.*\ y\ y)4)\\ &= (\lambda x.*\ x\ x)(*4\ 4) = (\lambda x.*\ x\ x)16 = (*16\ 16) = 256\end{aligned}$$

  - **Observação:** : Notem que a ordem das reduções poderia ser diferente:

$$\begin{aligned}(\lambda x.Q(Q(Qx)))2 &= Q(Q(Q\ 2) = (\lambda x.*\ x\ x)((\lambda y.*\ y\ y)((\lambda z.*\ z\ z)\ 2))\\ &= *\ ((\lambda y.*\ y\ y)((\lambda z.*\ z\ z)\ 2))\ ((\lambda y.*\ y\ y)((\lambda z.*\ z\ z)\ 2))\\ &= *\ ((\lambda y.*\ y\ y)(*2\ 2))((\lambda y.*\ y\ y)(*2\ 2))\\ &= *(*(*2\ 2)(*2\ 2))(*(*2\ 2)(*2\ 2))\\ &= *(*4\ 4)(*4\ 4) = *16\ 16 = 256\end{aligned}$$

Conceitos básicos

- **Exercícios:**
  1. Coloque os parenteses nas expressões abaixo
     1. $\lambda x.x\lambda y.yx$
     2. $(\lambda x.x)(\lambda y.y)\lambda x.x(\lambda y.y)z$
     3. $(\lambda f.\lambda y.\lambda z.fz\ y\ z)px$
     4. $\lambda x.x\lambda y.y\ \lambda z.z\lambda w.w\ z\ y\ x$
  2. Reduza as expressões a sua forma normal
     1. $(\lambda x.(x+3))4$
     2. $(\lambda fx.f(fx))(\lambda y.*\ y\ y)5$
  3. Dado $T = \lambda x.xxx$ Aplique a redução para TT.

Aritmética

- **Definição:** Define-se a constante zero como sendo a função:
  define Zero $= \lambda f.(\lambda z.z) = \lambda fz.z$
- Com isto podemos definir os próximos números naturais como:
  - $\lambda fz.f(z) = 1$
  - $\lambda fz.f(f(z)) = 2$
  - $\lambda fz.f(f(f(z))) = 3$
- Com isto a função sucessor pode ser definida como:
  define Suc $= \lambda nfx.f(nfx)$
- Quanto dá Suc Zero?

$$\begin{aligned} \textit{Suc Zero} &= (\lambda nfx.f(nfx))(\lambda fz.z) \\ &= (\lambda fx.f((\lambda yz.z)fx)) = (\lambda fx.f((\lambda z.z)x)) = \\ &= (\lambda fx.f(x)) \\ &\quad \text{como em cálculo lambda as variáveis não tem nome} \\ &= \lambda fz.f(z)) = 1 \end{aligned}$$

- Quanto dá Suc Suc Zero?

Aritmética

- Soma: define Add $= \lambda mnfx.\ m\ f(n\ f\ x)$
- Multiplicação: define Mult $= \lambda nmf.n(m\ f)$
- Exercício: Avalie:
  - Add 2 3
  - Mult 2 3

Simulador de Cálculo Lambda

- No Moodle foi colocado um simulador (apenas o executável) de cálculo Lambda.
- O simulador foi feito pelo Prof. Rodrigo Machado
- Para rodar o simulador primeiro vocês tem que instalar o Haskel e o Gtk
  - Link para Haskel: http://www.haskell.org/platform/
  - Link para Gtk: http://www.gtk.org/download/index.php
    - use all-in-one-bundle
    - abra num diretório onde não haja nenhum nome no caminho com espaço em branco
    - coloque o diretório bin do gtk no Path
- Abra o executável

Motivação

- Linguagens Predominantes
  - Começando nos anos 60: Linguagens imperativas procedurais
    - Cobol, Fortran, C, ...
  - Depois: Linguagens imperativas orientadas à objetos
    - Smaltalk, C++, Java, ...
- Linguagens Funcionais:
  - Basicamente na área acadêmica
    - Lisp, Haskel, ...
  - Mas de repente:
    - 1998 - Erlang (Ericsson Language) - Banida e reintroduzida em 2004
    - 2002 - Microsoft F# - Parte integrante do Visual Studio 2010
    - 2007- Clojure - Dialeto Moderno de Lisp - Roda na JVM
    - 2003 - Scala - Funcional + Orientação à objetos - Backend do twitter

Motivação

- O que houve ?
  - O software esta ficando mais lento mais rápido que o hardware está ficando rápido (Nickaus Wirth)
  - Temos que escrever código paralelo
  - Temos que escrever código rápido
  - É mais fácil escrever código paralelo usando linguagens funcionais.
    - Funções são de alta ordem
    - Valores são imutáveis (uma vez atribuídos)
    - Não tem efeitos colaterais
    - Casamento de padrões é simples
    - Recursão
    - Composição de Funções
    - Coleta de lixo
    - Outros
- Então usar apenas linguagens funcionais é a solução?
- Não. É necessário uma mistura de tecnologias, que inclui linguagens funcionais, procedurais, orientadas a objetos, etc...

CLisp

- A linguagem escolhida é o CLisp (Common Lisp)
  - Baseada no Cálculo Lambda
  - Simples
  - Padronizada
  - Base para as que vieram depois
- Vamos apenas dar uma introdução de Lisp, como aplicação do cálculo lambda.

Números e aritmética

- 3 tipos de Números:
  - Inteiros: 3, 6, 473, -5
  - Ponto Flutuante: 3.56, -8.34
    Observação: Trabalha com números de tamanho grande. Não tem tamanho máximo, exceto o tamanho da memória.
  - Razão: $\frac{4}{5}$
    Observação: As razões são sempre representadas com os menores denominadores possíveis. **Exemplo:** 6 dividido por 8 resulta em $\frac{3}{4}$ e não em $\frac{6}{8}$
- Operações aritméticas:
  - Exceto operadores unários, os outros podem ter mais de 2 operandos
  - A ordem dos operandos é importante
  - Pode usar todos os tipos de números
  - Faz a conversão necessária Inteiro $\rightarrow$ razão $\rightarrow$ Ponto flutuante

Números e aritmética

- Operações aritméticas Básicas:
  - Soma: **Exemplo:** $(+\ 3\ 5\ 7) = 15$
  - Subtração: **Exemplo:** $(-\ 3\ 5\ 7) = -9$
  - Multiplicação: **Exemplo:** $(*\ 6\ 8) = 48$
  - Divisão: **Exemplo:** $(/\ 3\ 5\ 7) = \frac{3}{35}$
  - Absoluto: **Exemplo:** $(abs\ -4) = 4$
  - Raiz quadrada: **Exemplo:** $(sqrt\ 37) = 6.0827627$

Símbolos

- Símbolo:
  - Tipo de dado muito importante em Lisp
  - Não confundir com string
  - Conjunto de letras, números e alguns caracteres
    Exemplo:
    - ano-atual
    - R2D2
    - +
    - 7-11

    Observação: +4 é um número não um símbolo, mas '+' e '7-11' são símbolos.
- Símbolos Especiais:
  - T : verdadeiro (true)
  - *NIL*: falso, vazio, não

Predicados

- Um predicado é uma pergunta cuja resposta pode ser sim (T) ou não (*NIL*).
- Os predicados são tão importantes em Lisp que pode-se pensar que o Lisp foi feito para eles.
- Os predicados normalmente são terminados pela letra p, embora isto seja só convenção.
- Alguns predicados básicos:
  - numberp: retorna T se a entrada for um número e *NIL* caso contrário
  - symbolp: retorna T se a entrada for um símbolo e *NIL* caso contrário
  - zerop: retorna T se a entrada for um zero e *NIL* caso contrário
  - oddp: retorna T se a entrada for impar e *NIL* caso contrário
  - evenp: retorna T se a entrada for par e *NIL* caso contrário
  - $<$: retorna T se a primeira entrada for menor que a segunda entrada e *NIL* caso contrário
  - $>$: retorna T se a primeira entrada for maior que a segunda entrada e *NIL* caso contrário

Predicados

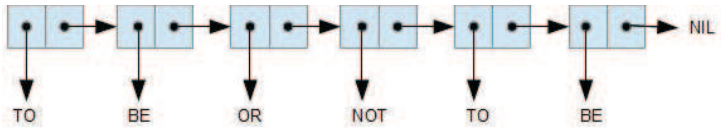
- Predicado de igualdade: Em Lisp existem 5 predicados de igualdade
  - $=$: compara apenas o valor dos números; Só aceita números; Não leva em conta tipo do número.
  - eq: são o mesmo objeto; tem o mesmo endereço
  - eql: similar ao *eq* mas para números não compara o endereço mas o conteúdo
  - equal: similar ao *eql* mas compara listas elemento a elemento
  - equalp: similar ao *equall* mas aceita algumas variações em strings. Por exemplo desconsidera se a letra é maiúscula ou minúscula.
- **Exercícios:**
  - $(=\ 3\ 3) = T$ e $(equal\ 3\ 3) = T$ e
  - $(setf\ a\ 3)$ e depois $(=\ a\ 3) = T$ e $(equal\ a\ 3) = T$ mas
  - $(=\ 3\ 3.0) = T$ e $(equal\ 3\ 3.0) = NIL$
    Observação: : Não se deve usar *eq* para comparar números pois mesmo duas constantes iguais podem ter endereços diferentes dependendo da implementação do Lisp.

Listas

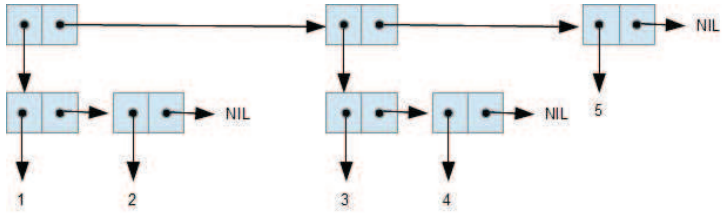
- Lisp é uma abreviatura de List Processor
- Listas são o tipo de dado mais importante de Lisp
- Listas são primitivas em Lisp
- Listas são usadas em Lisp para representar praticamente tudo:
  - Conjuntos
  - Tabelas
  - Gráficos
  - Funções
  - Sentenças numa Linguagem Natural
  - etc...
- **Exemplo:**
  - (TO BE OR NOT TO BE)
  - (1 2 3 4 5)
  - ((Joao das Neves) (Marcia Taborda))

Listas

- Uma Lista sempre termina por NIL
- A lista vazia é só o NIL: ()
- **Exemplo:**
  - (TO BE OR NOT TO BE)



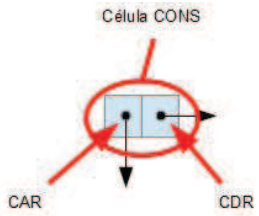
  - ((1 2) (3 4) 5)



Listas

- Célula CONS e CAR e CDR
  - A célula que aponta para um elemento de uma lista é chamado de célula CONS
  - A primeira parte da célula CONS é chamada de CAR
  - A segunda parte é chamada de CDR

    Observação: : Estes nomes são mantidos por questões históricas e não tem mais nenhum significado atual.



Listas

- Função CONS: A função cons constrói células cons (e portanto cria uma lista).
- Ela recebe 2 entradas:
  - A primeira será usada para montar o CAR da célula e será tratada como um objeto único
  - A segunda será usada para montar o CDR da célula e deve ser uma lista ou NIL
- Ela pode criar listas do zero ou acrescentar elementos numa lista

  Observação: : Cuidado veja o resultado da aplicação dos cons abaixo.

  Exemplo: :
  - $(cons\ '(joao\ antonio)\ '(maria\ silva)) = ((joao\ antonio)\ maria\ silva)$
  - $(cons\ 'joao\ '(maria\ silva)) = (joao\ maria\ silva)$
  - $(cons\ 'joao\ '(maria)) = (joao\ maria)$
  - $(cons\ 'joao\ 'maria) = (joao \bullet maria)$: Um tipo de lista que não vamos ver por enquanto.
  - $(cons\ 'joao\ NIL) = (joao)$
  - $(cons\ NIL\ NIL) = (NIL)$: que não é a lista vazia

Listas

- Funções primitivas para trabalhar com Listas
  - length: retorna o tamanho (número de elementos) da lista de entrada ou erro caso a entrada não seja uma lista
    Observação: : (*length* (1 2 (3 4))) não funciona porque o interpretador vai esperar uma função no lugar do 1. Caso se queira que o interpretador não avalie a expressão como uma função ela deve ser precedida por '.
    Exemplo: : (*length* '(1 2 (3 4))) = 3
  - list:Criar listas é tão comum em Lisp que existe uma função para criar listas diretamente (sem ter que acrescentar um a um)
    Exemplo:
    - (list 'bom 'dia) = (bom dia)
    - (list 1 3 4 ' (4 5 6)) = (1 3 4 (4 5 6))

Listas

- Funções primitivas para trabalhar com Listas
  - first: retorna o primeiro elemento de uma lista de entrada ou erro caso a entrada não seja uma lista
  - second: retorna o segundo elemento de uma lista de entrada ou erro caso a entrada não seja uma lista
  - third: retorna o terceiro elemento de uma lista de entrada ou erro caso a entrada não seja uma lista
  - rest: retorna o endereço do primeiro elemento ou erro caso a entrada não seja uma lista
  - **Observação:** : Podemos dizer que a função *first* retorna o CAR do nodo e a função *rest* o CDR do nodo
- Predicados primitivos para trabalhar com Listas
  - listp: retorna *T* se a entrada for uma lista e *NIL* caso contrário
  - consp: retorna *T* se a entrada for uma célula cons e *NIL* caso contrário
  - atom: retorna *T* se a entrada não for uma célula cons e *NIL* caso contrário

Avaliação

- Expressões em Lisp são avaliadas (executadas) antes de serem repassadas.
- Neste processo uma série de regras de avaliação são seguidas
- Regras:
  - Números, T e NIL: são avaliados para eles mesmos
  - Símbolos: são avaliados para o conteúdo deles (são considerados variáveis)
  - Listas: O primeiro elemento da lista especifica uma função a ser chamada, e esta função será chamada para a avaliação dos argumentos (proximos elementos na lista)
  - Aspas simples: A aspa simples ´ ou a função **quote** forçam a avaliação da expressão para ela mesma.

Declaração de Funções

- Em Lisp declara-se funções usando uma macro
- Sintaxe:
  - (*defun* < *name* > (< *par*1 >< *par*2 > ... < *parN* >) < *body*1 > ... < *bodyM* >)
  - O nome da função é dado pelo segundo elemento da macro: < *name* >
  - Os parametros da função são dados pelo terceiro elemento da macro, que por sua vez deve ser uma lista: (< *par*1 >< *par*2 > ... < *parN* >)
  - O corpo da função é dado pelo quarto elemento, ou elementos a seguir: < *body*1 > ... < *bodyM* >
  - O segundo e terceiro elementos de uma macro defun não são avaliados.
  - Apenas a avaliação do ultimo elemento da macro é retornado
- **Exemplo:** :
  - (*defun f*1 (*x y*) (∗ *x y*))

Declaração de Funções

- **Exercícios:** : Quais os erros, se houverem, na declaração das funções abaixo
  - (*defun f3* ('*x* '*y*) (*list x* '*gosta* '*de y*)) :
  - *x* e *y* em ('*x* '*y*) vão ser constantes e não parametros, portanto as variáveis *x* e *y* em (*list x* '*gosta* '*de y*)) não terão conteúdo.
  - (*defun f3* (*x y*) (*list* '*x* '*gosta* '*de y*)) :
  - *x* em (*list* '*x* '*gosta* '*de y*)) será uma constante e portanto o resultado não será o esperado.
  - (*defun f3* (*x y*) (*list x gosta de y*)) :
  - **gosta** e **de** em (*list x gosta de y*)) serão avaliados e portanto serão esperadas variáveis, como nenhuma variável **gosta** ou **de** foi instanciada, será gerado um erro.
  - (*defun f3* (*x y*) (*list x* '*gosta* '*de y*) *x*) :
  - o conteúdo da variável *x* (último corpo) será retornado e não a lista esperada.
  - (*defun f3* (*x y*) (*list x* '*gosta* '*de y*)) :
  - Correto

Exercícios

- Para podermos fazer algumas funções mais interessantes vamos mostrar a função especial IF
  - (*if* < *condition* > <body-true> <body-false>)
  - É uma função especial pois apenas um dentre os parâmetros 3 ou 4 (< *body* − *true* > *ou* < *body* − *false* >) será avaliado e seu resultado retornado.
- **Exemplo:** :
  - (*if* (*oddp* 1) '*odd* '*even*) = *odd*
  - (*if* (*oddp* 2) '*odd* '*even*) = *even*
- **Exercícios:** :
  - Crie uma função que recebe um número e retorna se ele é par ou impar: (*defun par*/*impar* (*x*) (*if* (*oddp x*) '*impar* '*par*))
  - Crie uma função que calcula o fatorial de um número:

```
(defun factorial(n)
   (if (= n 1)
      1
      (* n (factorial (- n 1)))
   )
)
```


Exercícios

- **Exercícios:** :
  - Dada a função abaixo
    1. Diga o que será impresso se for chamada por (teste 5)
    2. Diga a finalidade do símbolo **numero** em cada parte da função.

```
(defun teste(numero)
   (if (numberp numero)
      (list numero 'eh 'um 'numero)
      (list numero 'nao 'eh 'um 'numero)
   )
)
```

  - Crie uma função que recebe dois números e os ordena (ordem crescente):
  - Crie uma função que recebe dois números e a ordem desejada (crescente ou decrescente) e os ordena na ordem solicitada:

Exercícios

- **Exercícios:** :
  - Dada a função abaixo
    1. Diga o que será impresso se for chamada por (teste 5)
    2. Diga a finalidade do símbolo **palavra** em cada parte da função.

```
(defun teste(numero)
   (if (numberp numero)
      (list numero 'eh 'um 'numero)
      (list numero 'nao 'eh 'um 'numero)
   )
)
```

  - Crie uma função que recebe dois números e os ordena (ordem crescente):
  - Crie uma função que recebe dois números e a ordem desejada (crescente ou decrescente) e os ordena na ordem solicitada:

Notação Lambda

- Em Lisp toda as funções são funções lambda para as quais foi dado um nome.
- Mas Lisp permite que se use funções lambda sem nome.
- **Exemplo:** :
  - Função do Cálculo Lambda: $(\lambda x.\ (soma\ x\ 1))$
  - Função do Lisp: $(lambda\ (x)\ (+\ x\ 1))$
- Qual a vantagem de se usar uma função sem nome?
  - Quando é uma função simples que não vai ser mais usada e portanto não vale a pena nomear.
  - Quando a função é passada como parâmetro para outras funções.
  - Quando se quer aplicar uma função a uma lista inteira e cuja função não vai ser mais usada e portanto não vale a pena nomear.

Notação Lambda

- Mais importante é que funções em Lisp são expressões lambda para as quais foi associado um símbolo que a nomeia.
- No entanto este símbolo é só um apontador, a função independe dele e pode ser associada a outros símbolos e portanto podemos ter dois símbolos que chamam a mesma função.

  Observação: : na prática existe uma pequena diferença, o novo apontador deve ser chamado usando uma macro.

Eval e Apply

- Eval é uma função primitiva do Lisp que avalia 1 nível da expressão.
- **Exemplo:** :
  - $(eval\ '(list\ *\ 9\ 6)) = (*\ 9\ 6)$
  - $(eval\ (eval\ '(list\ *\ 9\ 6))) = 54$
- **Exercícios:** : O que resulta das expressões abaixo:
  - $(eval\ '(list\ (*\ 9\ 6))) = ((*\ 9\ 6))$
  - $(eval\ (list\ (*\ 9\ 6))) = Erro$
  - $(eval\ (eval\ '(list\ (*\ 9\ 6)))) = Erro$
  - $(eval\ '(*\ 9\ 6)) = 54$
  - $(eval\ (*\ 9\ 6)) = Erro$

Eval e Apply

- Apply é uma função primitiva do Lisp que recebe uma função e uma lista de objetos e chama a função usando os objetos da lista como parâmetros.
- A função deve ser precedido por $\#'$
- **Exemplo:** :
  - $(apply\ \#'+\ '(2\ 3\ 4)) = 9$
  - $(apply\ \#'equal\ '(15\ 20)) = NIL$

Cond

- Cond é uma macro que recebe um conjunto de clausulas, onde cada clausula é uma lista com um teste e um comando.
- Ele funciona assim: o teste da primeira clausula é executado, se for verdadeiro a macro retorna o resultado do comando associado; caso o teste seja falso, o controle passa para a segunda clausula e assim por diante até que algum teste seja verdadeiro ou que não haja mais clausulas. Neste caso a macro retorna NIL.
- **Exemplo:** :

```
(defun compare (xy)
  (cond
    ((equal x y) (x 'e y 'sao 'iguais))
    ((> x y) (x 'eh 'maior 'que y))
    ((< x y) (x eh 'menor 'que y))
  )
)
```


Cond

Observação: Caso se deseje uma clausula default, como em C, é só usar uma condição que sempre dá verdadeiro como T.

Exemplo: Função que devolve a capital de um pais.

```
(defun capital (x)
  (cond
    ((equal x 'Brasil) 'Brasilia)
    ((equal x 'Franca) 'Paris)
    ((equal x 'Italia) 'Roma)
    ((equal x 'Portugal) 'Lisboa)
    (T 'Desconhecido)
  )
)
```


And e Or

- E (And) e Ou (Or) são um pouco diferentes do costume em outras linguagens de programação.
- E (And)
  - (*and* $<$ *clause*1 $>$ $<$ *clause*2 $>$ ... $<$ *clauseN* $>$)
  - *And* avalia cada clausula, uma por uma e se alguma retorna NIL então o *And* também retorna NIL, caso contrário, o *And* retorna o valor da avaliação da última clausula.
- Ou (Or)
  - (*or* $<$ *clause*1 $>$ $<$ *clause*2 $>$ ... $<$ *clauseN* $>$)
  - *Or* avalia cada clausula, uma por uma e retorna a avaliação da primeira clausula que não retornar NIL. Caso todas retornem NIL, então o *Or* também retorna NIL.

And e Or

Exemplo:

- (*and* '*jorge* (= *prof* '2) (get-number-of-children *L*))
  - para *prof* = 3 vai retornar NIL
  - para *prof* = 2 vai retornar o resultado da função get-number-of-children chamada com o parametro *L*.
- (*or* '*jorge* (= *prof* '2) (get-number-of-children *L*))
  - vai retornar *jorge* independente dos valores de *prof* e *L*.

Exercícios: 1. Faça uma função que julgue o resultado de um jogo de papel, pedra e tesoura. A função recebe as escolhas dos 2 jogadores e retorna o resultado da jogada.

Exercícios: 2. Faça uma função que recebe uma lista com um conjunto de jogadas para os dois jogadores e retorne qual jogador venceu o maior numero de jogadas . (**Dica:** use a função do exercício anterior ou similar)

Step

- Step é uma ferramenta que permite executar passo a passo uma expressão.
- È usada principalmente para debugar o código.
  - $(step\ <clause>)$
  - Cada implementação de Lisp possui comandos diferentes internos ao step. No GNU-COMMON LISP 2.6.1 os comandos sâo mostrados abaixo:
    - n (or N or Newline): advances to the next form.
    - s (or S): skips the form.
    - p (or P): pretty-prints the form.
    - f (or F) FUNCTION: skips until the FUNCTION is called.
    - q (or Q): quits.
    - u (or U): goes up to the enclosing form.
    - e (or E) FORM: evaluates the FORM and prints the value(s).
    - r (or R) FORM: evaluates the FORM and returns the value(s).
    - b (or B): prints backtrace.
    - ?: prints this.

Trace

- Trace é uma ferramenta que sleleciona funções que serão monitoradas (entrada e saida) cada vez que forem chamadas.
- É usada principalmente para debugar o código.
  - $(trace\ <function1><function2>\ ...\ <functionN>)$
  - A partir do comando trace a função monitorada exibirá a sua entrada e saída sempre que for chamada.
  - **Exemplo:**

```
> (fat 5)
 1 > (FAT 5)
  2 > (FAT 4)
   3 > (FAT 3)
    4 > (FAT 2)
     5 > (FAT 1)
     < 5(FAT 1)
    < 4(FAT 2)
   < 3(FAT 6)
  < 2(FAT 24)
 < 1(FAT 120)
120
```


Trace

- O trace usado sem parâmetros mostra todas as funções que estão sendo monitoradas
- Para parar o monitoramento da função usa-se o untrace.
  - $(untrace\ <function1><function2>\ ...\ <functionN>)$
  - Caso se deseje parar de monitorar todas as funções usa-se o untrace sem parâmetros.

Atribuição

- Setf: Setf atribui um valor para uma variável. Setf é normalmente usado para variáveis globais.
  - $(setf\ <name><value>)$

    Observação: : Embora *setf* possa também ser usado para atribuir valores para variáveis locais isto é considerado um estilo ruim de programação, visto que em Lisp evita-se alterar valores de variáveis locais.
- Let: Let atribui valores para variáveis locais e executa um corpo para aqueles valores.
  - $(let\ ((<name1><value1>)\ (<name2><value2>)\ ...\ (<nameN><valueN>))\ <body>)$

    Primeiro as atribuições são feitas em paralelo e após o corpo é executado.

    Exemplo:

```
(defun media (x y z)
  (let ((soma (+ x y z)))
    (list'a 'media'de x y 'e z'eh (/ soma 2.0))
  )
```

Atribuição

- Let*: Let* atribui valores para variáveis locais e executa um corpo para aqueles valores.
  - (let* ((< *name*1 > < *value*1 >) (< *name*2 > < *value*2 >) ... (< *nameN* > < *valueN* >)) < *body* >)

    A diferença para *let* é que as atribuições são feitas em ordem e portanto uma atribuição pode usar os valores das atribuições anteriores.

Load

- Load: A função Load permite carregar um arquivo contendo um programa (declarações de funções, macros, variáveis globais, etc...)
  - (*load* <string-name>)

    Carrega o arquivo designado pelo string.
    Existem várias opções que podem ser definidas neste momento, mas nós não as veremos.
    Quem tiver interesse em se aprofundar pode acessar o livro on-line do Guy L. Steele Jr no site
    http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/clm.html

Comentários

- Em Lisp existem 2 maneiras de se colocar comentário num código.
- Comentário em arquivos:
  - Quando Lisp encontra um ponto e virgula numa linha ele desconsidera a linha até o próximo retorno de carro.
  - Por convenção existem 3 tipos de comentários de linha
    - ;;;: Três ponto-virgulas é usado para comentários fora da função
    - ;;: Dois ponto-virgulas é usado para comentários que ocupem toda a linha mas dentro da declaração da função
    - ;: Um ponto-virgula é usado para comentários que ocupem apenas a parte final de uma linha.
- Comentário que faz parte do código:
  - Uma função pode conter comentários que fazem parte do código e que podem ser acessados por comandos dentro do interpretador do Lisp.
  - Estes comentários são compostos por um string que deve ser colocado logo após a lista de parâmetros da função.

Comentários

Exemplo:

```
;;; Função Auxiliar
(defun media (x y z)
  "Função que calcula a média de 3 valores."
  (let ((soma (+ x y z))) ; Declaração da variável soma
    ;; A lista abaixo é devolvida com a média dos valores
    (list 'a 'media 'de x y 'e z 'eh (/ soma 2.0))
  )
)
```

- A função usada para visualizar o comentário interno ao código é documentation
  - (*documentation* < *name* > ' < *option* >)

    As opções mais usadas sâo function e variable.
    Existem outras opções, mas elas não serão mostradas aqui.

Apropos

- Apropos: É uma função que lista todas as funções atualmente declaradas que contenham um determinado string.
  - $(apropos \ < string > < package >)$

    O nome do pacote permite limitar a busca consultando apenas algum pacote incluído.
    Caso o nome do pacote seja "user"a busca se dará apenas entre as funçoes declaradas pelo usuário.

Listas

- Outras funções para trabalhar com Listas
  - append: une duas listas
  - reverse: inverte uma lista
  - nthcdr: retorna o n-ésimo cdr da lista
  - remove: remove um item de uma lista
  - member: testa se um item é membro da lista
  - intersection: acha a interseção entre duas listas
  - union: une duas listas, a diferença de append é que não duplica itens
  - set-difference: retorna a diferença entre duas listas $(L_1 - L_2)$
  - set-exclusive-or: retorna a diferença entre duas listas $((L_1 - L_2) + (L_2 - L_1))$
  - subsetp: testa se uma lista é subset de outra
  - assoc: percorre uma lista de listas e retorna a lista cujo primeiro elemento coincide com o elemento procurado.
  - subst: substitui um item por outro numa lista

Aplicando funções à listas

- Muitas vezes queremos aplicar uma mesma função a todos os elementos de uma lista.
- Para fazer isto o Lisp provê uma série de macros.
- A forma geral destas macros é:
  - $(< macro > \ \#' < function > < list >)$

    O nome da macro especifica como será aplicado a função e o que será coletado.
- As principais macros são:
  - mapcar: aplica a função a todos os elementos da lista e retorna uma lista com os resultados de cada aplicação.
  - find-if: aplica a função a todos os elementos da lista e retorna o primeiro para o qual a aplicação deu verdadeiro (não retornou NIL).
  - remove-if: aplica a função a todos os elementos da lista e retorna aqueles para os quais a aplicação deu falso (retornou NIL).
  - remove-if-not: aplica a função a todos os elementos da lista e retorna aqueles para os quais a aplicação deu verdadeiro (não retornou NIL).

Aplicando funções à listas

  - reduce: aplica a função a todos os elementos da lista 2 a 2. Após a primeira aplicação a operação é sempre entre o resultado anterior e o próximo elemento da lista.
  - every: aplica a função a todos os elementos da lista e retorna verdadeiro se todas as aplicação forem verdadeiras (não retornam NIL). Retorna NIL caso contrário.
- **Observação:** Mapcar pode ser usado para funções com mais de um parâmetro, neste caso deverão ser dadas tantas listas quantos parâmetros a função requerer. Caso as listas possuam tamanhos diferentes, o mapcar fará a aplicação para o menor tamanho de lista.

Exercícios

- **Exercícios:**
  1. Faça uma função que recebe um número *n* e retorna uma lista com *n* números aleatórios entre 0.0 e 1.0. **Dica:** Use recursão e a função (random <upper limit>).
  2. Faça uma função que recebe uma lista e retorna a quantidade de números multiplos de 3 e 5 na lista.
  3. Faça uma função que recebe uma lista e retorna a quantidade de números primos na lista.
  4. Crie um banco de dados (próxima página), formado por uma lista de atributos para objetos e faça os exercícios pedidos usando o banco de dados.
     - 4.1 Faça uma função que liste a placa de todos os veículos.
     - 4.2 Faça uma função que conte o número de veículos de um determinado ano.
     - 4.3 Faça uma função que liste o nome dos donos de veículos com cor azul.
     - 4.4 Faça uma função que liste o nome dos donos de veículos de uma determinada cor.
     - 4.5 Faça uma função que liste a placa de todos os veículos anteriores a uma determinada data que sejam carros de passeio.

Exercícios

```
(setf bd (list   '(veiculo1 placa ibc4076)
                 '(veiculo1 tipo carro-passeio)
                 '(veiculo1 cor vermelha)
                 '(veiculo1 ano 2010)
                 '(veiculo1 dono "Joao da Silva")
                 '(veiculo2 placa baf1800)
                 '(veiculo2 tipo carro-passeio)
                 '(veiculo2 cor azul)
                 '(veiculo2 ano 2009)
                 '(veiculo2 dono "Mauro Amaral")
                 '(veiculo3 placa ghi3456)
                 '(veiculo3 tipo utilitario)
                 '(veiculo3 cor azul)
                 '(veiculo3 ano 2009)
                 '(veiculo3 dono "Carlos Gomes")
))
```


Strings

- String em Lisp é um conjunto de caracteres entre aspas duplas.
- Os strings são um subtipo dos vetores (que não serão estudados neste curso).
- **Exemplo:**
  - "este é um string"
  - "1234"
  - "Este string contem letras e outros caracteres tais como -, /, *"
- Para acessar um caracter de um string usa-se a função char
  - (*char* < *string* > < *index* >)
- **Exemplo:**
  - > (setf st "este é um string")
    > (char st 3) = #\e
- **Observação:**
  1. Strings são indexados à partir de 0 (zero)
  2. Existem várias outras funções para criar, comparar e manipular strings, mas elas não serão estudadas neste curso.

Entrada e Saída

- Lisp possui um número muito grande de funções de entrada e saída.
- No entanto estas funções não são muito padronizadas e podem variar de implementação para implementação.
- Neste curso estudaremos apenas um pequeno número de funções de entrada e saída. Mais informações podem ser acessadas no site http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/clm.html
- Format: função que faz a impressão formatada
  - (*format* < *file* − *variable* > < *string* > < *exp*1 > < *exp*2 > ... < *expN* >)

    Caso a saída desejada seja a saída padrão, usa-se T no < *file* − *variable* >
    O string contem caracteres de controle que dizem como as expressões a seguir devem ser impressas.
    Estes caracteres de controle controlam não apenas o tipo de expressão mas como estas expressões são impressas.

Entrada e Saída

- Principais caracteres de controle
  - ∼A: Ascii - Qualquer expressão lisp será impressa. Mais amigavel para leitura por humanos.
  - ∼S: S-expression: Qualquer expressão lisp será impressa, mas num formato que poderá ser lido depois por um read
  - **Exemplo:** Usando o banco de dados criado anteriormente:
    - (format T "− ∼A-"bd) imprime:
      -((VEICULO1 PLACA IBC4076) (VEICULO1 TIPO CARRO-PASSEIO)
      (VEICULO1 COR VERMELHA) (VEICULO1 ANO 2010)
      (VEICULO1 DONO Joao da Silva) (VEICULO2 PLACA BAF1800)
      (VEICULO2 TIPO CARRO-PASSEIO) (VEICULO2 COR AZUL)
      (VEICULO2 ANO 2009) (VEICULO2 DONO Mauro Amaral)
      (VEICULO3 PLACA GHI3456) (VEICULO3 TIPO UTILITARIO)
      (VEICULO3 COR VERDE-MUSGO) (VEICULO3 ANO 2009)
      (VEICULO3 DONO Carlos Gomes))-

Entrada e Saída

- (format T "− ∼S-"bd) imprime:
  -((VEICULO1 PLACA IBC4076) (VEICULO1 TIPO CARRO-PASSEIO)
  (VEICULO1 COR VERMELHA) (VEICULO1 ANO 2010)
  (VEICULO1 DONO "Joao da Silva") (VEICULO2 PLACA BAF1800)
  (VEICULO2 TIPO CARRO-PASSEIO) (VEICULO2 COR AZUL)
  (VEICULO2 ANO 2009) (VEICULO2 DONO "Mauro Amaral")
  (VEICULO3 PLACA GHI3456) (VEICULO3 TIPO UTILITARIO)
  (VEICULO3 COR VERDE-MUSGO) (VEICULO3 ANO 2009)
  (VEICULO3 DONO "Carlos Gomes"))-

Entrada e Saída

- **Observação:** : De forma geral existem muitas maneiras diferentes de escrever um objeto em lisp. Por exemplo, o inteiro 27 pode ser escrito como: 27, 27., #o33, #x1B, #b11011, #.(* 3 3 3), e 81/3.
- Outros caracteres de controle:
  - ∼D: Decimal
  - ∼B: Binário
  - ∼O: Octal
  - ∼X: Hexadecimal
  - ∼R: Racional
  - ∼C: Caracter
  - ∼F: Ponto flutuante
  - ∼E: Exponencial
  - ∼%: Pula linha
  - ∼∼: Imprime ∼
- **Exemplo:**
  - ∼D: imprime um número decimal
  - ∼3D: imprime um número decimal usando pelo menos 3 casas.
  - ∼3,'0D: imprime um número decimal usando pelo menos 3 casas e preenchendo com 0 em vez de espaço.

Entrada e Saída

- Read: A função read lê um objeto da entrada.
  - $(read \ < file - variable >)$

    Caso a entrada desejada seja a entrada padrão, omite-se o $< file - variable >$
- Quando se deseja usar um arquivo que não seja a entrada/saída padrão, deve-se primeiramente abrir o arquivo.
  - $(whith - open - file \ (< file - variable > < file - name > < options >) \ < body >)$

    Caso a entrada desejada seja a entrada padrão, omite-se o $< file - variable >$
    :DIRECTION :OUTPUT são as opções usadas quando se desejar abrir o arquivo para escrita.

Outros

- Esta introdução dá uma ideia do que o Lisp pode fazer e qual é a ideia de programação em Lisp. No entanto deixa de fora muitas características e funções do Lisp.
- **Exemplo:** Lisp possui ainda:
  - Tipos estruturados de dados:
    - Sequencias
    - Vetor
    - Matrizes
    - Tabelas Hash
    - Pilhas
    - Listas de propriedades
    - Estruturas com e sem herança de atributos, com e sem declaração de métodos
  - Declarações de Macros

Outros

- **Exemplo:** Lisp possui ainda:
  - Comandos iterativos:
    - Do
    - Do*
    - Dotimes
    - Dolist
  - Comandos controle:
    - Return
    - Return-from
    - Error
    - Debug
  - E muitas outras funções pré-definidas