Linguagem de Programação
Introdução à Linguagem Hugs98

Turma A

Prof. Marcelo Ladeira
CIC/UnB

Baseado em slides preparados pelo ex-professor do CIC. Wagner Teixeira, atualmente aposentado.

Introdução à Linguagem Hugs98

- Referência
  - *Hugs98 User Manual*
- Autores
  - Mark P. Jones
  - John C. Peterson
- Distribuição e informações
  - http://haskell.org/hugs/
  - http://www.haskell.org/hugs/

Visão de Mundo

- Programar em uma LP exige pensar com os significados das suas construções.
- Cada paradigma (visão) tem construções que lhe são peculiares:
  - Procedural (ou imperativo)
    - Solução algorítmica (passo a passo).
  - Funcional
    - Declara a solução como valores a retornar.
  - Relacional (ou lógico)
    - Declara a solução como relação entre entidades do discurso.
  - Orientado a objetos
    - Descreve o problema em termos do próprio problema, ao invés de descrevê-lo em termos de um algoritmo que o computador vai rodar.

3

MDC

- Funcional

  mdc a 0 = a

  mdc a b = mdc b (a ´mod´ b)

- **Imperativo**

```
function mdc(a,b:integer):integer
      var t:integer;
       begin
         while b<> 0 do begin
             t := b;  b:=a mod b;  a:= t;
        end;
          mdc :=a ;
       end;
```

- **Lógico**

```
mdc(A,0,A).
mdc(A,B,X) :-
    BB is A mod B, mdc(B, BB, X).
```

Linguagens Declarativas

- Realizam o processamento simbólico
- Processam listas (ES)
- Bloco básico de construção: funções
- Declaram o problema fazendo sua especificação (não resolvendo passo a passo)
- implementam funções complexas,
- implementam metas programas
- Tratam símbolos e operações da lógica matemática com relativa facilidade.

Ambiente HUGS

- **É um interpretador com o ciclo lê-avalia-exibe resultados.**
- **Funções primitivas:**
  - **lidas do** Prelude
- **Funções definidas pelo usuário:**
  - **usando expressões** let **ou** where
  - **carregadas de um arquivo de texto.**

Expressões let e where

- let <definição> in <expressão>

  let {d_1; d_2; ....; d_n} in expressão

  ```
  Prelude> let soma a b = a+b in soma 12 15
  27 :: Integer
  ```

- <expressão> where <definição>

  expressão where {d_1; d_2; ....; d_n}

  ```
  Prelude> fat 5 where fat n = product [1..n]
  120 :: Integer
  ```

Comandos no Prelude

```
Prelude> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.
:load <filenames>
                load modules from specified files
:load           clear all files except prelude
:also <filenames>   read additional modules
:reload         repeat last load command
:project <filename>       use project file
:edit <filename>          edit file
:edit                     edit last module
:module <module>
        set module for evaluating expressions
<expr>          evaluate expression
:type <expr>              print type of expression
:?                  display this list of commands
:set <options>      set command line options
:set help on command line options
:names [pat]        list names currently in scope
:info <names>             describe named objects
:browse <modules>
        browse names defined in <modules>
:find <name>        edit module containing
        definition of name
:! command                shell escape
:cd dir                   change directory
:gc                       force garbage collection
:version                  print Hugs version
:quit                     exit Hugs interpreter
```

Re-escrita

- Permite transformar (re-escrever) termos em outros
- É a base do processo de avaliação de expressões
- Programas funcionais são executados usando a redução ou re-escrita de termos.
- Exemplo:

```
append   [ ]     ys  =  ys
append (x:xs)  ys   = x:append xs ys

append [1,3,5] [4,6] =1:append [3,5] [4,6]
                     = 1:3:append [5] [4,6]
                     = 1:3:5:append [ ] [4,6]
                     = 1:3:5:[4,6]
                     = [1,3,5,4,6]
```

Funções e Operadores

- São estruturas da LF que aplicadas a parâmetros e operandos retornam valores.
- Funções são em geral prefixadas, associativas à esquerda e com prioridade máxima (10).
  - Exemplo: f a b = (f a) b
- Operadores são em geral infixados e podem ter suas prioridades e associatividades declaradas.
  - Exemplo: x^y^2 = x^(y^2), mas
    x+y+2 = (x+y)+2

Funções

- Tem nomes com letras e dígitos, começando com letra. São prefixadas:

  a) fatorial n = product [1..n]

  b) mdc a b
  ```
          | b == 0  = a
          | otherwise  = mdc b (a `mod` b)
  ```


Uma função pode ser tratada de forma infixada se for envolvida por apóstrofes.

Função Lambda

- Função anônima. É definida como: (\args -> corpo)

  Exemplo: (\x y -> x^2 + y^2)

  Prelude> (\x -> x^2) 3

  9 :: Integer

  Prelude> map (\x ->x^2) [1,2,3,4]

  [1,4,9,16] :: [Integer]

Funções com Guardas

- Uma função é definida como:

eq_1	**mdc a 0 = a**
eq_2	**mdc a b = mdc b (mod a b)**
...	**fat 0 = 1**
eq_n	**fat (n+1) = (n+1) * fat n**

- Onde cada equação tem uma das formas:

a) f p_1 p_2 ... p_k | <guarda> = expressão

b) f p_1 p_2 ... p_k = expressão

```
mdc a b
      | b == 0   = a
      | otherwise = mdc b (mod a b)
```

Expressões Case

- Uma expressão *case* tem a forma geral:
  case <exp> of { p_1 $match_1$; ... ; p_n $match_n$ }

```
merge xs ys = case (xs,ys) of
   (z:zs,w:ws) | z<=w   -> z:merge zs ys
               | z>w    -> w:merge xs  ws
   ([],ws)              -> ys
   (zs,[])              -> xs
Main> merge [1,3..9] [2,4..10]
[1,2,3,4,5,6,7,8,9,10] :: [Integer]
```

Expressões Case

- fat n = case n of
  0 -> 1
  1 -> 1
  (k+1) -> (k+1)*fat k
- if e_1 then e_2 else e_3
  = case e_1 of { True -> e_2 ; False -> e_3 }

  e2 e e3 podem ser if também!

Expressões Case

```
qsort ls =
   case ls of
          []          -> []
          [x]         -> [x]
          otherwise -> qsort ys ++ [x] ++ qsort zs
              where
                   (x:xs) = ls
                   ys = [y | y <- xs, y <  x]
                   zs = [z | z <- xs, z >= x]
Main> qsort [8,5,7,3,4]
   [3,4,5,7,8] :: [Integer]
```

Expressões Case

```
pega n ys = case (n,ys) of
      (0,_)     ->  []
      (_,[])    ->  []
      (n,x:xs) ->  x : pega (n-1)  xs
```

```
Main> pega 3 [8,5,7,3,4]
  [8,5,7] :: [Integer]
Main> pega 7 [8,5,7,3,4]
  [8,5,7,3,4] :: [Integer]
```

Operadores

- **Operador tem nome formado por símbolo especial (não letras ou dígito) e é infixado:**

```
Prelude> map (\x ->x^2) [1,2,3,4]
[1,4,9,16] :: [Integer]
Prelude> 5 == 9
False :: Bool
Prelude> [1,2,3] ++ [5,6]        ou        (++) [1,2,3] [5,6]
[1,2,3,5,6] :: [Integer]
```

Operadores

- **Operadores são definidos de forma similar a funções:**

```
[] \/ ys = ys
(x:xs) \/ ys | membro x ys  = xs \/ ys
             | otherwise    = x: xs \/ ys
membro z [] = False
membro z (w:ws) = z==w || membro z ws
```

- **Funções e operadores podem fazer uso de definições locais com let ou where**

```
xs \/ ys      | xs == [] = ys
              |  membro x ys = resto
              | otherwise = x:resto where (x:t)=xs;resto = t \/ ys
```


\/ é a união de listas.

Operadores

- A definição de um operador leva em conta:
  - prioridade
  - associatividade
  - comportamento
- Prioridade (inteiro entre 1 e 9)
  2*3+4 = (2*3) +4 =10 ?
  = 2*(3 + 4) =14 ?
- Associatividade
  1 - 2 - 3 = (1-2)-3 = -4 ?
  = 1-(2-3) = 2
- x⊕y⊕z = (x ⊕ y) ⊕ z - à esquerda infixl
  = x ⊕ (y ⊕ z) - à direita infixr
  = erro! - não associado. infix

Tipo Booleano

```
(&&), (||)      :: Bool -> Bool -> Bool

True  && x      =  x
False && _      =  False

True  || _      =  True
False || x      =  x

not             :: Bool -> Bool
not True        =  False
not False       =  True

otherwise       :: Bool
otherwise       =  True
```

Tipo Char

- É uma enumeração e consiste de valores de 16 bits, conforme o padrão unicode.
- É representado pelo caractere entre aspas simples: ‘a’, ‘b’, ‘A’, ‘0’, ‘1’, ..., ‘9’, etc.
- Cada um dos caracteres de controle ascii tem mais de uma representação possível:
  - Ex.:
    ‘\7’, ‘\a’ e ‘\BEL’, ‘\b’ e ‘\BS’, ‘\f’ e ‘\FF’, ‘\r’ e ‘ \CR’, ‘\t’ e ‘\HT’, ‘\v’ e ‘\VT’, ‘\n’ e ‘\LF’
- Funções de conversão: chr e ord
  - disponível após “import Data.Char” ou “:l Data.Char”

    chr :: Int → Char
    ord :: Char → Int

Tipo String

- É uma lista de caracteres

  type String = [Char] é um tipo sinônimo

- Strings podem ser abreviadas envolvendo os caracteres por aspas

  " string” abrevia a notação [‘ ‘,’s’,’t’,’r’, ‘i’,’n’,’g’]

- Todas as operações para lista se aplicam a strings

Strings e I/O

- String são objetos visíveis: podem ser lidos ou impressos.
- Exemplo:

```
Prelude> let { leia = do
                        putStr "informe uma string >";
                            str <- getLine; putStr str}
             in leia
informe uma string > lah vai a string, pega!
 lah vai a string, pega! :: IO ()
Prelude> putStr "Isto eh uma string"
Isto eh uma string :: IO ()
```

Strings e I/O

- Qualquer objeto para ser impresso deve antes ser convertido em string

Exemplo:

```
Prelude> putStr (show [1,2,3,4,5])
    [1,2,3,4,5] :: IO ()
Prelude> read "[1,2,3,4]" :: [Int] -- converte string em objeto
    [1,2,3,4] :: [Int]
Prelude> show (23,5.4) -- converte objeto em string
    "(23,5.4)" :: [Char]
Prelude> read "[('a','b',4.5),('c','d',6.0)]" ::[(Char,Char,Float)]
    [('a','b',4.5),('c','d',6.0)] :: [(Char,Char,Float)]
```

Tipos Numéricos

- **Int e Integer**
  Int tem valores limitados
  Integer tem valores ilimitado
  - **Operadores para inteiros**
    - **+, -, *, /, ^, negate, div, rem, mod, odd, even, abs, etc**
  - **Funções diversas existem no prelude para inteiros**
- **Float**
- **Complex**

Tipo Listas

- É um tipo algébrico de dois construtores: ‘[]’ e ‘:’
- Há muitas funções no Prelude para manipular listas
- Exemplos:

```
[ ]                              lista vazia ou nula
[3]   == 3:[]                    lista com 1 elemento
[1,2,3,4,5] == 1:2:3:4:5:[]      lista com 5 elementos numéricos
Prelude> [1,3..15]
   [1,3,5,7,9,11,13,15] :: [Integer]
Prelude> ['a'..'z']
   "abcdefghijklmnopqrstuvwxyz" :: [Char]
Prelude> [0.1,0.3 .. 2.0]
   [0.1,0.3,0.5,0.7,0.9,1.1,1.3,1.5,1.7,1.9] :: [Double]
Prelude> length [1..10]
   10 :: Int
```

Tipo Listas

- É um conjunto ordenado e **homogêneo** de elementos com repetição permitida.
- Se t é um tipo, então [t] é uma lista de elementos do tipo t.
- Operadores e funções sobre listas

  length, ++, :, concat, filter, head, tail, last, take, drop, replicate, reverse, zip, unzip, elem, and, or, foldl1, foldl, etc.

Operadores e Funções em Tipos Listas Exemplos

```
length [1..1000]
    1000
[1,2]:[2,3]:[]
    [[1,2],[2,3]]
concat [['1'],['2'],['3']," abc"," efg"]
    "123 abc efg"
head (tail [1,5..200])
    5
last (reverse [1,10..200])
    1
drop 2 (take 4 ["jan","fev","mar","abr","mai","jun"])
replicate 5 ‘a’
    “aaaaa”
replicate 3 “a”
    [“a”, “a”, “a”]
elem 13 [1,3..20]
    True
[1,3..10]++[2,4..8]
    [1,3,5,7,9,2,4,6,8]
filter even [1..10]
    [2, 4, 6, 8, 10]
```

Operadores e Funções em Tipos Listas Exemplos

```
zip [1,3..10] [0,2..15]
    [(1,0),(3,2),(5,4),(7,6),(9,8)]
unzip (zip [1,3..10] [0,2..15])
    ([1,3,5,7,9],[0,2,4,6,8])
and (map even [1,2,3,4])
    False
foldl (+) 2 [-2..2]
    2
foldr (-) 2 [-2..2]
    -2
foldl (-) 2 [-2..2]
    2
or (map even [1,2,3,4])
        True
foldl1 (+) [-2..2]
        0
foldr1 (-) [-2..2]
        0
foldl1 (-) [-2..2]
        4
```


Continuação.

Exemplos de Fold

foldl (op) arg0 [a,b,c] = ((arg0 op a) op b) op c

foldl1 (op) [a,b,c] = (a op b) op c

foldr (op) arg0 [a,b,c] = a op (b op (arg0 op c))

foldr1 (op) [a,b,c] = a op (b op c)

```
foldr (-) 2 [-2..2]
   -2
foldl (-) 2 [-2..2]
   2
```

```
foldr1 (-) [-2..2]
        0
foldl1 (-) [-2..2]
        -4
```

Tipo Tuplas

- É uma estrutura do tipo registro.
- Uma relação fixa de campos de tipos quaisquer.
- Se t_1, t_2, ..., t_n são tipos, então o tipo da n-tupla é (t_1, t_2, ..., t_n).
- Exemplo:

  (("Nome","Gisele"),"mulher","casada",("idade", 28))

  (5, [1,2,3], "Brasília")

Entrada/Saída

```
entrada = do  putStr "\n dados> "
              dados <- getLine
              putStr “Digitado: "
              putStr (concat [dados, "\n"])
              putStr "continua, (s/n)?"
              carac <- getChar
              if carac=='S' || carac=='s' then entrada
              else return ()
Main> entrada
 dados> primeira
Digitado: primeira
continua, (s/n)?s
 dados> Segunda
Resultado: Segunda
continua, (s/n)?T :: IO [Char]
```

Entrada/Saída

```
import Data.Char
tecla = do putStr "\n Qual a tecla? > "
        carac <- getChar
        putStr (show (ord carac))
        if carac == '\ESC' then (return ())
                           else tecla
Main> tecla
 Qual a tecla? > a97
 Qual a tecla? > b98
 Qual a tecla? > B66
 Qual a tecla? > 7 :: IO ()
```

Paradigma Funcional

- Uma função é definida como um conjunto de equações.
- Cada equação é uma regra de reescrita (redução).
- Exemplo:

```
filtro p (x:xs)
                | p x       = x:filtro p xs
                | otherwise = filtro p xs
filtro _ [] = []
Main> filtro even [1..50]
   [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,4
   4,46,48,50] :: [Int]
```

Casamento de Padrões

- É o processo de avaliar um ou mais dos argumentos para determinar qual a expressão do lado direito que se aplica.
- É o processo de identificar a regra de reescrita que se aplica.
- Reescrita (redução)
  substituir a expressão corrente pela expressão do lado direito, conforme os padrões casados com a instância de argumentos.

Guardas

- Cada uma das equações na definição pode conter guardas.
- Uma guarda é uma função predicativa.
- Se a guarda for verdadeira, a expressão correspondente na função é executada.
- Exemplo:

```
fat n | n == 0  || n==1  = 1
      | n > 1              = n* fat (n-1)
Main> fat 5
   120 :: Integer
```

Recursão

- É a única estrutura de controle entre comandos em uma linguagem funcional pura.
- O controle de seqüência intra-comando, em expressões, é definido pelas prioridades e associatividades das funções e operadores.
- É o processo da função chamar a si mesma, direta ou indiretamente.
- Exemplo:

```
fat 0 = 1
fat 1 = 1
fat n = n * fat (n-1)                -- e se n < 0 ?
```

Polimorfismo

- Tipos polimórficos descrevem famílias de tipos:
  [a] é uma família de listas para diferentes instâncias de tipos:
  a = Int, tem-se [Int]
  a = Char, “ [Char]
  a = [Float], “ [[Float]]
  a - é uma variável de tipo
  Nesse sentido a é mais geral que as instâncias de tipos Int, Char, [Float], etc.

Polimorfismo

- Funções Polimórficas

  Uma função que se aplica a qualquer tipo de parâmetro é uma função polimórfica.

- Exemplos.:

```
size (x:xs) = 1+size xs
size [] = 0
Main> size [1,3,15]
3 :: Integer
Main> size ['a'..'z']
26 :: Integer
```

Polimorfismo

```
somamenor m xs
   | length xs <= 1 = []
   | z < m        = (x,y,z):somamenor m (y:t)
   | otherwise  = somamenor m (y:t) where
   (x:y:t) = xs; z = x+y
Main> somamenor 10 [1,2,3,4,5,6,7,8,9]
   [(1,2,3),(2,3,5),(3,4,7),(4,5,9)]
Main> somamenor 10
   [1.5,2.5,3.5,4.5,5.5,6.5,7.5,8.5,9.5]
   [(1.5,2.5,4.0),(2.5,3.5,6.0),(3.5,4.5,8.0)]
```

Avaliação Preguiçosa

- Uma expressão só é avaliada quando seu valor é requerido.
- Uma expressão cotizada, aparecendo em vários lugares, é avaliada uma única vez
- Exemplos:

```
zero x = 0
Main> zero 10
    0 :: Integer
Main> zero (1/0)
    0 :: Integer
Main> zero $! (1/0)  -- Força avaliação argumento
    Program execution error: {primDivDouble 1.0 0.0}
```

Avaliação Preguiçosa: qd x =x*x

- Passagem de parâmetro por referência:

  qd(qd (qd 2)) = (qd (qd 2)) * (qd (qd 2))
  = (qd 2) * (qd 2) * (qd (qd 2))
  = 2 * 2 * (qd 2) * (qd (qd 2))
  = 4 * (qd 2) * (qd (qd 2))
  = 4 * 4 * (qd (qd 2))
  = 16 * (qd (qd 2))
  = 16 * 16 = 256

- Passagem de parâmetro por valor:

  qd(qd(qd 2)) = qd(qd (2*2)) = qd(qd(4))=qd(4*4)
  = qd(16) = 16*16

- Ambas apresentam esforço próximo a 3 multiplicações

Objetos Infinitos

- Graças a avaliação preguiçosa é possível lidar com lista infinitas.
- Exemplos:

```
Main> let naturais n = take n [0..] in naturais 8
[0,1,2,3,4,5,6,7] :: [Integer]

Main> let impar n = take n [1,3..] in impar 10
[1,3,5,7,9,11,13,15,17,19] :: [Integer]
```

Avaliação Preguiçosa e Objetos Infinitos

```
fiblst =1:1:[x+y|(x,y) <- (zip fiblst (tail fiblst))]
Main> take 10 fiblst
   [1,1,2,3,5,8,13,21,34,55] :: [Integer]
```

		fiblst anterior	tail fiblst anterior	
1	**[1]**	**[]**	**não utilizada**	
2	**[1,1]**	**[1]**	**a partir daqui calcule (x,y)**	
3	**[1,1,2]**	**[1,1]**	**[1]**	**(1,1)**
4	**[1,1,2,3]**	**[1,1,2]**	**[1,2]**	**(1,1), (1,2)**
5	**[1,1,2,3,5]**	**[1,1,2,3]**	**[1,2,3]**	**(1,1), (1,2), (2,3)**
6	**[1,1,2,3,5,8]**	**[1,1,2,3,5]**	**[1,2,3,5]**	**(1,1), (1,2), (2,3), (3,5)**

Funções de Ordem alta

- Funções que manuseiam outras funções:
  - composição de funções.
  - passadas como parâmetros.
  - resultado da chamada de função.
  - aplicação parcialmente aos seus argumentos.
  - usadas em estrutura de dados.

Composição de Funções

- f . g x = f (g x)

```
var xs = (sum . map (^2)) xs / n - (sum xs / n)^2
        where n = length xs
Main> var [1.0,1.5..5.0]
    1.66667 :: Double
Main> ((^3) . (/2)) 10
    125.0 :: Double
Main> (sum . take 10) [1..]
    55 :: Integer
Main> let uns = 1:uns in (sum . take 20) uns
    20 :: Integer
Main> take 20 uns where uns = 1:uns
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1] :: [Integer]
Main> let uns n | n==1 = [1] | n/=1 = 1:uns (n-1) in
    (sum.uns) 30
```

Funções como parâmetros

```
filtro p (x:xs)
                | p x       = x:filtro p xs
                | otherwise = filtro p xs
filtro _ [] = []
par x = x `mod` 2 == 0
```

```
Main> filtro par [1..21]
    [2,4,6,8,10,12,14,16,18,20] :: [Integer]
Main> filtro odd [1..20]
    [1,3,5,7,9,11,13,15,17,19] :: [Integer]
Main> map par [1..10]
    [False,True,False,True,False,True,False,True,False,True] :: [Bool]
Main> foldr (+) 0 [1..10]
    55 :: Integer
```

Função como resultado

```
somat  = foldl (+) 0
produt = foldl (*) 1
append = foldr (++) []
quadlista = map (^2)
```

```
somat :: [Integer] -> Integer

produt :: [Integer] -> Integer

append :: [[a]] -> [a]

quadlista :: [Integer] -> [Integer]
```

Exs.

```
Main> somat [1..10]
   55 :: Integer
Main> produt [1..10]
   3628800 :: Integer
Main> append [[1,2], [3,4],[5,6,7],[8,9,10] ]
   [1,2,3,4,5,6,7,8,9,10] :: [Integer]
Main> quadlista [1,3,5,7,9]
   [1,9,25,49,81] :: [Integer]
```

Aplicação parcial

```
Main> suc 4 where suc = (+1)
    5 :: Integer
Main> somatorio [1..10]
                        where somatorio = foldr (+) 0
    55 :: Integer
Main> ola "Joao"
    where ola = ("Oi " ++) . (++ " como vai voce?")
    "Oi Joao como vai voce?" :: [Char]
Main> soma 3 5 where soma = (+)
    8 :: Integer
```

Funções em Estrutura de dados

```
Main>  let operacao (f, x, y) = f x y in operacao ((+), 4, 6)
    10 :: Integer
Main> let operacao (f, x, y) = f x y
                              in map operacao [((+),4,6),((*),3,4)]
    [10,12] :: [Integer]
```

Map

```
map f xs          = [f x | x <- xs]

map2 f xs ys      = [f x y | (x,y) <- zip xs ys]
```

- representação f x y versus f(x,y)

  f(x,y) - uncurry f x y