Introdução a Linguagem Funcional Haskell

Reescrita, Expressões, Tipos Básicos e Abstrações

M.F. Caetano
mfcaetano@unb.br

Departamento de Ciência da Computação
Universidade de Brasília

Material Base

- Programming in Haskell (G. Hutton);
- Learn You Haskell;
  - `http://learnyouahaskell.com/introduction`
- Material Prof. Rodrigo Bonifácio;
- Material Prof. Marcelo Ladeira.

O que é uma linguagem funcional?

- **Programação Funcional** é um estilo de programação em que o método básico de computação é a aplicação de funções a argumentos;
- Uma linguagem de programação funcional **suporta** e **encoraja** a **aplicação do estilo funcional**, dirigindo a decomposição dos problemas em termos de funções e composição de funções.

Contextualização Histórica

- Alonzo Church desenvolve o λ – *calculus* (1930);
- John McCarthy desenvolve o Lisp (1960), primeira linguagem funcional com influências do λ – *calculus*, mas retendo a atribuição de variáveis;

Contextualização Histórica

- Várias outras linguagens funcionais são desenvolvidas (70s – 80s), sem atribuição de variáveis e com ênfase em **funções de alta ordem** e ricos **sistemas de tipos** (ISWIM, FP e ML).
  - **funções de alta ordem** – podem retornar ou receber outras funções como parâmetros;
  - um **sistema de tipo** permite que o programador indique à priori como devem ser tratados os dados, associando-os a um tipo;
- Um **comitê internacional** de pesquisadores inicia o desenvolvimento de Haskell (1987), uma linguagem funcional padronizada com **recursos de avaliação tardia** (*lazy evaluation*);
  - Expressões não são avaliadas quando são atribuídas a variáveis e sim quando efetivamente são utilizadas! Argumentos de uma função são avaliados, por exemplo, quando efetivamente são usados.
    - **Ponto Negativo**: Uso de memória difícil de prever.
      Exemplo: $2 + 2$ :: *Int* e 4 :: *Int*

Soma de inteiros em uma linguagem imperativa

```
int total = 0;
for(int i = 1; i <= 5; i++){
    total += i;
}
```

- O método de computação é a atribuição de variáveis;

```
total -> 1 ; i = 1
total -> 3 ; i = 2
total -> 6 ; i = 3
total -> 10; i = 4
total -> 15; i = 5
```


Soma de inteiros em Haskell

```
> sum[1..5]
```

- **Método de computação é a aplicação de funções**; no sentido de que a execução de um programa corresponde a uma sequencia de aplicações de funções;
- Neste exemplo, a função *sum* (predefina) é aplicada ao resultado da aplicação da função **geradora** [1..5].

Computação através de reescrita (ou *Calculation*)

```
sum [] = 0
sum (x:xs) = x + sum xs
```

$$
\begin{aligned}
sum[1..5] &= sum[1,2,3,4,5],\ \textit{definição de } [..] && (1)\\
&= 1 + sum[2,3,4,5],\ \textit{definição de sum} && (2)\\
&= ...,\ \textit{definição de sum} && (3)\\
&= 1 + 2 + 3 + 4 + sum[5],\ \textit{definição de sum} && (4)\\
&= 1 + 2 + 3 + 4 + 5 + sum[],\ \textit{definição de sum} && (5)\\
&= 1 + 2 + 3 + 4 + 5 + 0,\ \textit{definição de sum} && (6)\\
&= 15,\ \textit{definição de } (+) && (7)
\end{aligned}
$$

Reescrita como mecanismo de prova

Supondo:

```
f a b c = a * (b + c)
```

Podemos estabelecer que, para qualquer a, b, c a aplicação $f\ a\ b\ c$ deve levar sempre a um resultado igual a aplicação $f\ a\ c\ b$

$$\begin{aligned} f\ a\ b\ c &= a * (b + c),\ \textit{definição de } f && (8)\\ &= a * (c + b),\ \textit{comutatividade da adição} && (9)\\ &= f\ a\ c\ b,\ \textit{definição de } f && (10) \end{aligned}$$

Aplicação de Funções

Em Haskell, **aplicação de função** é representado por **espaço**, assim como **multiplicação** por $*$;

- $f\ a\ b\ +\ c * d \longrightarrow f(a, b) + c \times d$

Aplicação de uma função tem precedência sobre os outros operadores:

- $f\ a\ +\ b \longrightarrow (f\ a) + b$ ao invés de $f(a + b)$

Aplicação de Funções

Mathematics	Haskell
$f(x)$	$f\ x$
$f(x, y)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ x)$
$f(x, g(y))$	$f\ x\ (g\ y)$
$f(x)g(y)$	$f\ x * g\ y$

- Note que ainda é necessário o uso de parenteses no caso: $f\ (g\ x)$;
  - O não uso $(f\ g\ x)$ seria interpretado como a função f recebe dois parâmetros: g e x.

Capítulo 2: Primeiros Passos

- Navegar nas referências disponíveis em `http://www.haskell.org`;
- Utilizar algumas implementações:
  - Gasgow Haskell Compiler and Interpreter (GHC)[1]

    ```
    :~$> sudo apt-get install ghc ghc-doc
    ```
  - Hugs Interpreter (Hugs)[2]

    ```
    :~$> sudo apt-get install hugs
    ```

[1] `http://www.haskell.org/ghc`
[2] `http://www.haskell.org/hugs`

Iniciando uma sessão com Hugs

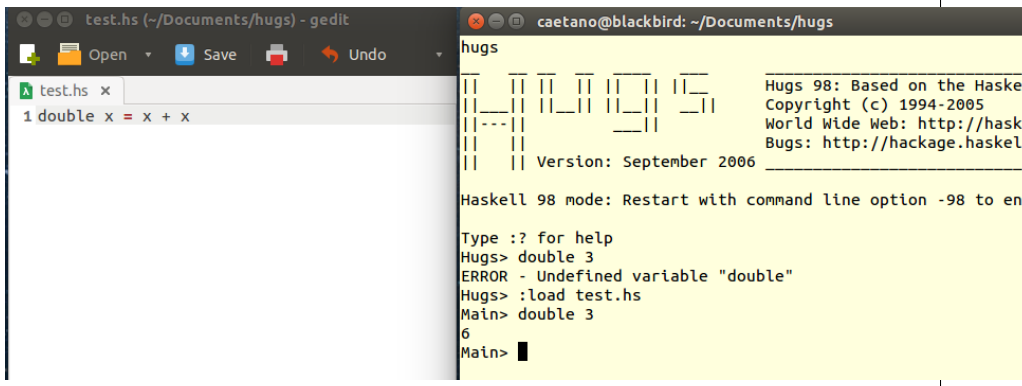
```
caetano@blackbird:~$ hugs
__   __ __  __  ____   ___                          _____________________________________
||   || ||  || ||  || ||__          Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||         Copyright (c) 1994-2005
||---||         ___||               World Wide Web: http://haskell.org/hugs
||   ||                             Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 _____________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> 
```

Interpretador pronto para avaliar expressões

```
Hugs> 2 + 3 * 4
14
Hugs> sqrt (3^2 + 4^2)
5.0
Hugs> 8 `div` 2
4
```


Haskell: Scripts

- Hugs disponibiliza um arcabouço de funções da biblioteca padrão. Também é possível a definição de funções por parte do usuário;
- Novas funções não podem ser definidas no terminal Hugs!
  - são definidas em aquivos texto sem formação e carregadas no terminal.
  - por convenção arquivos possuem extensão *.hs* (**Não é mandatório**);



Haskell: Scripts

Alterações feitas no script precisam ser recarregadas no Hugs.

```
Hugs> :reload
```

Command	Meaning
:*load name*	load script *name*
:*reload*	reload current script
:*edit name*	edit script *name*
:*edit*	edit current script
:*type expr*	show type of *expr*
:?	show all commands
:*quit*	quit Hugs

Figure : Tabela contendo os comandos mais usados em Hugs

Haskell: Funcionalidades

1. Sintaxe elegante, levando a programas concisos;
   - Programas escritos em Haskell são entre duas e dez vezes menores do que programas escritos em outras linguagens;
2. Linguagem funcional pura, com suporte a avaliação tardia (*Lazy Evaluation*);
   - baseado na ideia de que nenhuma computação deva ser realizada até que o resultado seja realmente necessário;
3. Casamento de padrões;
4. Avançado sistema de tipos (polimorfismo, inferência, ...)
5. Funções de alta-ordem;
   - funções podem ser passadas como parâmetros ou retornadas como resultado de funções;
6. Facilidades para "raciocinar" sobre os programas.

Definindo nomes em Haskell

- Nomes de funções e argumentos devem começar com letra minúsculas.

```
myFun
fun1
arg_2
x'
```

- Por convenção, nome que representa lista de argumentos usualmente tem *s* em seu nome.

```
xs
ns
nss
```


Regras de Layout

Em uma sequencia de definições, cada definição deve começar precisamente na mesma coluna:

```
a = 10          a = 10
b = 20            b = 20
c = 30          c = 30
```

Certo! **Errado!**

Regras de Layout evitam a necessidade de explicitar cada grupo de definição.

```
a = b + c                          {a = b + c
  where                              where
      b = 1                              {b = 1;
      c = 2        Significa ⟶           c = 2}
d = a * 2                          d = a * 2}
```

Grupo Implícito **Grupo Explícito**

Haskell: Palavras reservadas

As seguintes palavras possuem um significado especial e são reservadas a linguagem:

case	**class**	**data**	**default**	**deriving**	**do**	**else**
if	**import**	**in**	**infix**	**infixl**	**infixr**	**instance**
let	**module**	**newtype**	**of**	**then**	**type**	**where**

Haskell: Comentários

O compilador Hugs irá ignorar o simbolo $--$, pois será considerado comentário.

```
--Esta é uma linha de comentario
n = 2 ^ 3
```

Um bloco de comentário pode ser obtido com os símbolos: $\{- \; -\}$.

```
{-
Este é um bloco de comentário
n = 2 ^ 3
-}
```

Haskell: Funções

Definição

Função é um mapeamento que pega um ou mais argumentos e produz um único resultado. É definida através de uma equação que define um nome para a função, um nome para cada dos seus argumentos e um corpo, que especifica como o resultado pode ser calculado em termos dos argumentos.

Exemplo:

```
double x = x + x
```

A função *double* pega o número x como seu argumento e produz como resultado $x + x$.

```
double 3
= {aplicando double}
3 + 3
= {aplicando +}
6
```


Haskell: Funções

Exemplo 2:

```
double (double 2)
= {aplicando double interno}
double (2 + 2)
= {aplicando +}
double 4
= {aplicando double}
4 + 4
= {aplicando +}
8
```


Haskell: Funções

Exemplo 2b: Segunda linha possível de execução.

```
double (double 2)
= {aplicando double externo}
double 2 + double 2
= {aplicando primeiro double}
(2 + 2) + double 2
= {aplicando +}
4 + double 2
= {aplicando double}
4 + (2 + 2)
= {aplicando +}
4 + 4
= {aplicando +}
8
```


Haskell: Funções

Exemplo 2b (6 passos) executado em dois passos a mais que Exemplo 2 (4 passos).

Importante:

A ordem de execução não impacta no resultado final, contudo, pode impactar no tempo final de processamento.

Haskell: Funções

QuickSort: Este método parte do princípio de que é mais rápido ordenar dois vetores com **n/2** elementos cada um, do que um com **n** elementos (**dividir para conquistar**).

Ideia geral:

- O primeiro passo é dividir o vetor original;
- Este procedimento é denominado **particionamento**;
  - Deve-se escolher uma das posições do vetor a qual é denominada **pivô**:

    `V[i]`

Haskell: Funções

- Uma vez escolhido o pivô, os elementos do vetor são movimentados de forma que:
  - O **subvetor à esquerda** do pivô contenha somente os elementos cujos valores são **menores que o pivô**;e
  - O **subvetor da direita** contenha valores **maiores que o valor do pivô**.

    `V[0], ...,V[i-1], V[i], V[i+1], ..., V[n-1]`
  - O procedimento é repetido até que o vetor esteja ordenado;
  - Existem várias formas de se escolher o pivô.

Haskell: Funções

Algoritmo (sugestão):

1. Pivô é escolhido no meio do vetor. O elemento colocado numa variável auxiliar **pivo**;
2. São iniciadas duas variáveis auxiliares $i = inicio$ e $j = fim$;
3. O vetor é percorrido do início até que se encontre um $V[i] \geq pivo$ (valor de i é incrementado no processo);
4. O vetor é percorrido a partir do fim até que se encontre um $V[j] \leq pivo$ (valor de j é decrementado no processo);
5. $V[i]$ e $V[j]$ são trocados; i é incrementado de 1 e j é decrementado de 1;
6. O processo é repetido até que i e j se cruzem em algum ponto do vetor ($i > j$);
7. Quando são obtidos os dois segmentos do vetor por meio do processo de partição, realiza-se a ordenação de cada um deles de forma recursiva.

Haskell: Funções

Algoritmo *qsort* implementado na linguagem C.

```
void quicksort(int vec[], int inicio, int fim) {
   int pivo = vec[ (int)(inicio+fim)/2 ];
   int i = inicio, j = fim, temp;
   while (i < j) {
      while(pivo > vec[i]) i++;
      while(pivo < vec[j]) j--;
      if (i<=j) {
         temp = vec[i];
         vec[i] = vec[j];
         vec[j] = t;
         i++;
         j--;
      }
   }
   if (j > inicio)
      quicksort(vec, inicio, j);
   if (i < fim)
      quicksort(vec, i, fim);
}
```

Haskell: Funções

Seja o Algoritmo *qsort* definido pela função:

```
qsort [ ] = [ ]
qsort (x : xs) = qsort smaller ++[x]++ qsort larger
    where
        smaller = [a| a <- xs, a <=x]
        larger  = [b| b <- xs, b > x]
```


Haskell: Funções

Qual é o efeito em se aplicar o algoritmo *qsort* em uma lista de um elemento?

```
qsort [x]
= {aplicando qsort}
qsort[ ] ++[x]++ qsort[ ]
= {aplicando qsort}
[ ] ++[x]++ [ ]
= {aplicando ++}
[x]
```


Haskell: Funções

Aplicando *qsort* na lista $[3, 5, 1, 4, 2]$:

```
qsort [3,5,1,4,2]
= {aplicando qsort}
qsort [1,2] ++[3]++ qsort[5,4]
= {aplicando qsort}
(qsort [ ] ++[1]++ qsort [2]) ++[3]++
(qsort [4] ++[5]++ qsort [ ])
= {aplicando qsort}
([ ] ++[1]++ [2]) ++[3]++ ([4] ++[5]++ [ ])
= {aplicando ++}
[1,2] ++[3]++ [4,5]
= {aplicando ++}
[1,2,3,4,5]
```


Haskell: Funções

Modifique o programa anterior para que a ordenação seja feita de forma decrescente.

```
qsortd [ ] = [ ]
qsortd (x : xs) = qsortd larger ++[x]++ qsortd smaller
    where
        smaller = [a| a <- xs, a <=x]
        larger  = [b| b <- xs, b > x]
```

Haskell: funcionalidades comuns

- **Selecionar o primeiro elemento de uma lista não vazia:**

```
> head [1,2,3,4,5]
1
```

- **Remover o primeiro elemento de uma lista não vazia:**

```
> tail [1,2,3,4,5]
[2,3,4,5]
```

- **Selecionar n – *elemento* da lista (índice inicia a partir de zero):**

```
> [1,2,3,4,5] !! 2
3
```

- **Selecionar os n primeiros elementos de uma lista:**

```
> take 3 [1,2,3,4,5]
[1,2,3]
```

- **Remover os n primeiros elementos de uma lista:**

```
> drop 3 [1,2,3,4,5]
[4,5]
```


Haskell: funcionalidades comuns

- **Calcular o tamanho de uma lista:**

```
> length [1,2,3,4,5]
5
```

- **Calcular a soma dos elementos de uma lista:**

```
> sum [1,2,3,4,5]
15
```

- **Calcular o produto dos elementos de uma lista:**

```
> product [1,2,3,4,5]
120
```

- **Unir duas listas;**

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

- **Reverter a ordem dos elementos de uma lista:**

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```


Haskell: funcionalidades comuns

Utilizando a linguagem Haskell, implemente a função fatorial.

Arquivo: *fatorial.hs*

```
fatorial n = product [1..n]
```

Ambiente Hugs:

```
Hugs> :load fatorial.hs
Main> fatorial 8
40320
```


Capítulo 3: Tipos e Classes

- **Expressões:** Objetos nos quais aplicamos os cálculos (simplificações, substituições ou reescritas de termos).
- **Valores:** Objetos resultantes dos cálculos realizados. É interessante pensar sobre *valores* como expressões que não podem ser ainda mais simplificadas.

Exemplos

- Valores atômicos: 42, 'a', ...
- Valores estruturados: $[1, 2, 3]$, $('b', 4)$, ...
- Expressões: $1 + 2$, *head* $[1, 2, 3]$, *fst* $('b', 4)$, ...

Tipos

Toda expressão e, por consequência, todo valor possui um tipo associado. Tipos podem ser vistos como conjuntos de expressões (ou valores) nos quais os membros possuem características em comum.

- $42 :: \mathit{Integer}$
- $'a' :: \mathit{Char}$
- $[1, 2, 3] :: [\mathit{Integer}]$
- $('b', 4) :: (\mathit{Char}, \mathit{Integer})$

Tipo *Bool* contem os dois valores lógicos *True* e *False*. Enquanto o tipo $\mathit{Bool} \to \mathit{Bool}$ contém todas as funções que mapeiam os argumentos de *Bool* para resultados tipo *Bool*. Exemplo:

- $\mathit{False} :: \mathit{Bool}$
- $\mathit{True} :: \mathit{Bool}$
- $\neg :: \mathit{Bool} \to \mathit{Bool}$

Sistema de tipos Haskell

- Implementa um algoritmo que infere os tipos das expressões;
- Garante que os programas em Haskell sejam bem tipados (**não faz sentido a expressão como 'a' + 'b'**), ou seja, Haskell é uma linguagem estaticamente tipada (problemas de tipos são identificados antes da execução).

Exemplo:

- $\neg\ \mathit{False} :: \mathit{Bool}$ (CORRETO)
  - $\neg :: \mathit{Bool} \to \mathit{Bool}$
  - $\mathit{False} :: \mathit{Bool}$
- $\neg\ 3$ (ERRADO)
  - $\neg :: \mathit{Bool} \to \mathit{Bool}$
  - $3 :: \mathit{Int}$

Sistema de tipos Haskell

Sistema de tipos de Haskell é muito importante e consegue identificar uma grande classe de erros de programa. Nem todo erro é detectado:

- $1\ `\mathit{div}'\ 0$ – é livre de erros de tipo. Contudo, produz um erro de avaliação pois não existe divisão por zero.

Tipos Básicos

Haskell possui os seguintes tipos básicos disponíveis:

- *Bool* – Valores lógicos;
- *Char* – Único Caracter;
- *String* – Uma string de caracteres;
- *Int* – Inteiro com precisão fixa (-2^{31} até 2^{31-1});
- *Integer* – Número inteiros com precisão arbitrária;
- *Float* – Números de ponto flutuante;

Lista

Uma lista é uma sequencia de valores do mesmo tipo.

```
[False,True,False] :: [Bool]
['a','b','c','d'] :: [Char]
[['a','b'],['c','d']] :: [Char]
```

Generalizando:
$[t]$ é o tipo da lista com elementos com tipo t;

- Número de elementos de uma lista é chamado de *length*;
  - O tipo de uma lista não diz nada sobre o seu tamanho;
- A lista [] tem *length* igual a zero e é uma lista vazia;
- As listas [[]] e [] são diferentes;

Tuplas

Uma tupla é uma sequencia de valores PODENDO ter tipos diferentes:

```
(True,False) :: (Bool, Bool)
(False,'b',True) :: (Bool,Char,Bool)
```

Generalizando:
$(T_1, T_2, \ldots, T_n)$ é o tipo de n-tuplas onde o i elemento possui o tipo T_i, para qualquer i entre $1..n$;

Tuplas

Note que o tipo de uma tupla associa o seu tamanho:

```
(True,False) :: (Bool, Bool)
(False,'b',True) :: (Bool,Char,Bool)
```

O tipo de componentes é irrestrito:

```
('a',(False,'b')) :: (Char,(Bool,Char))
(True,['a','b']) :: (Bool,[Char])
```


Funções

Uma função é o mapeamento de um valor(es) de um tipo para valor(es) de outro tipo:

$$\neg :: Bool \rightarrow Bool \quad (11)$$

$$even :: Int \rightarrow Bool \quad (12)$$

Generalizando:
$T_1 \rightarrow T_2$ é o tipo de uma função que mapeia valores do tipo T_1 para o tipo T_2.

Funções

- A seta $\rightarrow$ é representada no teclado por $-$ $>$ (sem espaço);
- O tipo dos argumentos e do resultado são irrestritos. Por exemplo, funções com múltiplos argumentos ou resultados são possíveis utilizando estrutura do tipo **lista** ou **tupla**:

```
add :: (Int,Int) -> Int
add(x,y) = x + y

zeroto :: Int -> [Int]
zeroto n = [0..n]
```


Funções Curried

Funções com múltiplos argumentos também são possíveis de retornar uma função como resultado:

```
add' :: Int -> (Int -> Int)
add' x y = x + y
```

- A função *add′* pega um inteiro x e retorna a função $add'x$. Por sua vez, esta função pega o inteiro y e retorna como resultado $x + y$;
- As funções *add* e *add′* produzem o mesmo resultado final, mas *add* pega dois argumentos ao mesmo tempo e *add′* pega um argumento por vez:

```
add :: (Int,Int) -> Int
add' :: Int -> (Int -> Int)
```

  - Funções que pegam o seu argumento um de cada vez são chamadas de **curried functions**. Homenagem a Haskkel Curry que trabalhou neste tipo de função.

Funções Curried

Funções com mais de um argumento seguem a mesma ideia:

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x*y*z
```

O que está acontecendo?

A função *mult* pega o inteiro x e retorna a função *mult* x. Esta pega o inteiro y e retorna a função *mult* x y que finalmente pega o inteiro z e retorna $x * y * z$

Porque Funções Currying são importantes?

Funções *Curried* são mais flexíveis que funções usando **tuplas**, basicamente pois funções podem ser feitas a partir da aplicação parcial de funções *curried*.

```
add' 1 :: Int -> Int
take 5 :: [Int] -> [Int]
drop 5 :: [Int] -> [Int]
```

Para evitar o uso excessivo de parênteses, são adotados duas convenções:

- A seta $\rightarrow$ se associa com a **direta**:
  - $Int \rightarrow Int \rightarrow Int \rightarrow Int$
  - Significa: $Int \rightarrow (Int \rightarrow (Int \rightarrow Int))$
- Natural aplicação de uma função é a **esquerda**:
  - *mult* x y z
  - Significa: $((mult\ x)\ y)\ z$

Porque Funções Currying são importantes?

Qual é o tipo da função *sqr*? Definida como:

```
sqr x = x * x
```

Recebe um valor tipo *Integer* e retorna um outro valor também do tipo *Integer*. Dizemos que a função *sqr* é do tipo:
Integer $\rightarrow$ *Integer*

- Se uma função recebe um parâmetro de um tipo T_1 qualquer e retorna um valor do tipo T_2 qualquer, dizemos:
  - tal função mapea valores do tipo T_1 em valores do tipo T_2
  - tal função é associada ao tipo: $T_1 \rightarrow T_2$

Generalizando

- Se existirem $n-1$ argumentos $\{A_1, A_2, \ldots, A_{n-1}\}$ para uma função f, tal que $A_1 :: T_1, A_2 :: T_2, \ldots, A_{n-1} :: T_{n-1}$ e o tipo de retorno de f for T_n indicamos que:

$$f :: T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_{n-1} \rightarrow T_n$$

E podemos escrever:

f :: *Integer* $\rightarrow$ *Integer* $\rightarrow$ *Integer* $\rightarrow$ *Integer*

```
f a b c = a * (a + b)
```


Inferência de tipos

A chave para este processo é uma regra de tipagem para a *aplicação de funções*, indicando que se f é uma função que mapeia argumentos do tipo T_1 em um resultado do tipo T_2, e *exp* é uma expressão do tipo T_1, então a aplicação f *exp* tem tipo T_2.

$$\frac{f :: T_1 \rightarrow T_2 \;\; exp :: T_1}{f \;\; exp :: T_2} \qquad (13)$$

Funções Polimórficas

Uma função é chamada de *polimórfica* (*polymorphic function*) (muitas formas) se o seu tipo contem um ou mais *variável de tipos*.

$$length \;\; :: \;\; [a] \;\; \rightarrow Int$$

Para qualquer tipo ***a***, **length** pega a lista de valores do tipo ***a*** e retorna um **inteiro**

Note:

- **Variável de tipo** pode ser instanciada em diferentes tipos e em diferentes circunstâncias:

```
> length [False,True]        a = Bool
2
> length [1,2,3,4]           a = Int
4
```

Variável de tipo deve iniciar com letra minúscula como: a,b,c, etc...

Funções Polimórficas

Muitas funções definidas pela linguagem são polimórficas. Por exemplo:

```
fst :: (a,b) -> a

head :: [a] -> a

take :: Int -> [a] -> [a]

zip :: [a] -> [b] -> [(a,b)]

id :: a -> a
```


Funções Sobrecarregadas

Uma **função polimórfica** é chamada de **sobrecarregada** (*overloaded*) se seu tipo contem um ou mais **restrições de classes** (*class constraints*)

Restrições de classe são escritas na forma *C a*, onde *C* é o nome da classe e *a* é uma **variável de tipo**.

```
(+) :: Num a => a -> a -> a
```

Para qualquer tipo numérico **a**, que é uma instância da classe *Num* dos **tipos numéricos**, o operador/função **(+)** tem o tipo $a \to a \to a$ pega dois valores tipo **a** e retorna um valor do tipo **a**.

- $Num\ a \Rightarrow a \to a \to a$ é um tipo sobrecarregado (*overloaded type*)
- $(+)$ é uma função sobrecarregada (*overloaded function*)

Funções Sobrecarregadas

Variáveis de tipos restritos (*Constrained type variables*) podem ser instanciadas para qualquer tipo que satisfaça as restrições:

```
> 1 + 2
3
> 1.0 + 2.0
3.0
> 'a' + 'b'
ERROR
```

a = Int

a = Float

Char não é do tipo numérico

Funções Sobrecarregadas

Haskell tem um número de **classes de tipos** (*type classes*). Exemplos:

```
(+) :: Num a => a -> a -> a
(==) :: Eq a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
```

Alguns tipos de classes suportados pela linguagem Haskell:

Classe Eq: tipo igualdade; são valores que podem ser comparados.

```
(==) :: a -> a -> Bool
(\=) :: a -> a -> Bool
```

Os tipos básicos: *Bool*, *Char*, *String*, *Int*, *Integer* e *Float* são instâncias de **Eq**.

Funções Sobrecarregadas

Classe Ord: – tipo ordenado. Contém tipos que são instâncias da classe **Eq** e podem ser ordenados com os seguintes métodos.

```
(<)  :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>)  :: a -> a -> Bool
(>=) :: a -> a -> Bool
min  :: a -> a -> a
max  :: a -> a -> a
```


Funções Sobrecarregadas

Classe Num: – tipo numérico. Podem ser processados usando os seguintes métodos.

```
(+)    :: a -> a -> a
(-)    :: a -> a -> a
(*)    :: a -> a -> a
negate :: a -> a
abs    :: a -> a
signum :: a -> a
```


Funções Sobrecarregadas

Classe Integral: – tipo integral. Essa classe contem tipos que são instâncias da **class Num** e adicional valores que são Inteiros e que suportam divisão:

```
div :: a -> a -> a
mod :: a -> a -> a
```

Classe Fractional: – Esta classe contém tipos que são da **classe Num** mas adicionalmente valores não-inteiros, bem como métodos para suporte de divisão facionado.

```
(/)   :: a -> a -> a
recip :: a -> a
```


Exercícios

Qual é o tipo dos seguintes valores?

a. ['a','b','c','d']
b. ('a','b','c')
c. [(*False*, '0'),(*True*,'1')]
d. ([*False*, *True*],['0','1'])
e. [*tail*,*init*,*reverse*]

Respostas:

a. [*Char*]
b. (*Char*, *Char*, *Char*)
c. [(*Bool*, *Char*)]
d. ([*Bool*], [*Char*])
e. $[[a] \to [a]]$

Verifique suas respostas utilizando o Hugs!

```
> :type ['a','b','c','d']
[Char]
```

Capítulo 4: Definindo Funções

Assim como em outras linguagens, em Haskell é possível definir novas funções em função de funções já existentes.

- **Decide se um caracter é um dígito:**

```
isDigit :: Char -> Bool
isDigit c = ((c >= '0') && (c <= '9'))
```

- **Decide se um inteiro é par:**

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0
```

- **Divide uma lista na n-enésima posição:**

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)
```


Capítulo 4: Definindo Funções

- **Remove um elemento da lista na n-enésima posição:**

```
removeAt :: Int -> [a] -> [a]
\begin{verbatim}
removeAt x xs = (take (x-1) xs) ++ (drop x xs)
```

- **Revezamento:**

```
rev :: Fractional a => a -> a
rev n = 1/n
```

Note que o uso de *class constraints* nos tipos de ***even*** e ***rev*** acima torna a ideia de que as funções são aplicadas a funções do tipo *integral* e *fracionário*.

Conditional Expression

Assim como na maioria das linguagens, funções podem ser definidas utilizando **expressões condicionais**.

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

abs pega um inteiro **n** e retorna **n** se ele for um valor não negativo e **-n** caso contrário.

Conditional Expression

Expressões condicionais podem ser alinhadas:

```
signum :: Int -> Int
signum n = if n < 0 then -1 else
if n == 0 then 0 else 1
```

Problema de Ambiguidade

Em Haskell, expressões condicionais **SEMPRE** devem ter o **else**, pois evitam problemas de ambiguidade (problema do **else pendente)**.

```
if True then if False then 1 else 2
```

- pode retornar tanto o **valor 2** quanto ocorrer um erro. Irá depender da interpretação de quem o **else** está associado (primeiro ou segundo if).

Guarded Equations

Uma alternativa para estrutura de controle condicional vista.

```
abs n | n >= 0    = 0
      | otherwise = -n
```

Utilizando múltiplos condicionais:

```
signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

O uso de **otherwise** não é obrigatório. Ele significa todos os outros casos!

Pattern Matching

Muitas funções são definidas utilizando **pattern matching** em seus argumentos

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


Pattern Matching

Funções podem ser definidas de maneiras diferentes utilizando **pattern matching**.

```
(&&) :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

Uma forma mais compacta usando o coringa "_":

```
True && True = True
_ && _       = False
```


Pattern Matching

Versão mais eficiente pois não precisa verificar o segundo argumento caso haja um **match** no primeiro argumento:

```
True  && True = True
False && _    = False
```

Padrões são aplicados em ordem. Ou seja, a definição abaixo **SEMPRE** retorna **False**:

```
_ && _       = False
True && True = True
```

Pattern Matching

Por problemas de ordem internas, Padrões não podem repetir variáveis:

```
b && b = b
_ && _ = False
```

Maneira correta de representar o código anterior:

```
b && c | c == b    = b
       | otherwise = False
```


List Patterns

A função ***teste*** abaixo decide se a lista contem precisamente três caracteres iniciando com a letra *a*:

```
teste :: [Char] -> Bool
teste ['a',_,_] = True
teste _         = False
```

Como as listas são construídas?

Internamente, cada lista **não vazia** é construída utilizando operador **(:)** chamado de **cons**, que adiciona um elemento ao começo da lista.

```
[1,2,3,4]
```

Significa:

```
1:(2:(3:(4:[])))
```


List Patterns

Podemos generalizar a função ***teste***:

```
teste :: [Char] -> Bool
teste ('a':_)   = True
teste _         = False
```

Funções de lista podem ser definidas usando o padrão **x:xs**

```
null :: [a] -> Bool
null []   = True
null(_:_) = False

head :: [a] -> a
head (x:_)  = x

tail :: [a] -> [a]
tail (_:xs) = xs
```


List Patterns

Importante:

O padrão **x:xs** deve ser sempre utilizado com parênteses, pois funções de aplicações tem maior prioridade do que todos os outros operadores.

Exemplo, seja a definição:

```
tail _ : xs = xs
```

Ela será interpretada de forma errada como:

```
(tail _) : xs = xs
```

Lambda Expressions

Funções podem ser construídas sem um nome utilizando para isso **expressões lambda**:

$$\lambda x \to x + x$$

Função sem nome que pega o número x (argumento) e retorna como resultado $x + x$.

- A letra grega Lambda λ é representada pela barra \

Expressões Lambda são úteis?

Expressões Lambda podem ser utilizadas para dar um significado formal para funções definidas por *currying*.

Exemplo:

```
add x y = x + y
```

Significado:
$add \ = \ \lambda x \to (\lambda y \to x + y)$

```
add = \x -> (\y -> x + y)
```


Expressões Lambda são úteis?

Expressões lambda são úteis para definição de funções que retornam funções.

Exemplo:

```
const :: a -> b -> a
const x _ = x
```

Definindo de uma forma mais "natural":

```
const :: a -> (b -> a)
const x = \_ -> x
```

Exemplo de uso:

```
> map (const 42) [1..5]
[42,42,42,42,42]
```


Expressões Lambda são úteis?

Expressões lambda podem ser utilizada para evitar nomes de funções que são referenciadas apenas uma vez. Desta forma, a definição torna-se mais enxuta.

```
odds n = map f[0..n-1]
    where
        f x = x*2 + 1
```

Podendo ser simplificado, da seguinte forma:

```
odds n = map(\x -> x*2 +1)[0..n-1]
```

Sections

Um operador escrito entre dois argumentos pode ser convertido para *curried function* escrita antes dos argumentos utilizando parênteses.

```
> 1+2
3
> (+) 1 2
3
```

Outra forma possível:

```
> (1+) 2
3
> (+2) 1
3
```

Definição

De forma geral, se $\oplus$ é um operador, então a função formada por $(\oplus)$, $(x \oplus)$ e $(y \oplus)$ são chamados de *sections*.

Seções são úteis?

Funções úteis podem ser construídas de uma forma fácil utilizando seções.

- $(1+)$ – *successor function*;
- $(1/)$ – *reciprocation function*
- $(*2)$ – *doubling function*;
- $(/2)$ – *halving function*.

Exercícios

Utilizando a biblioteca de funções, defina a função *halve* :: $[a] \rightarrow ([a], [a])$ que divide listas de tamanho par em duas partes.

```
> halve [1,2,3,4,5,6]
([1,2,3],[4,5,6])
```

Resposta A:

```
halve xs = splitAt (length xs `div` 2) xs
```

Resposta B:

```
halve xs = (take n xs, drop n xs)
    where
        n = length xs `div` 2
```


Exercícios

Resposta C (testa tamanho lista):

```
halve xs | (m == 0) = (first, second)
         | otherwise = (xs, [])
    where
        first  = take n xs
        second = drop n xs
        n      = ((length xs) `div` 2)
        m      = ((length xs) `mod` 2)
```

Exercícios

Considerando a função **safetail** que se comporta que nem a função **tail**, exceto que a função **safetail** mapeia uma lista vazia para uma lista vazia, enquanto a função **tail** retorna um erro. Defina **safetail** utilizando:

- *Conditional Expression*;
- *Guarded Equations*;
- *Pattern Matching*.

Exemplo:

```
Hugs> tail [1,2,3,4,5]
[2,3,4,5]
Hugs> tail []
Program error: pattern match failure: tail []
```


Exercícios

Algumas formas de se resolver este problema.

Resposta A:

```
safetail xs = if null xs then [] else tail xs
```

Resposta B:

```
safetail xs | null xs   = []
            | otherwise = tail xs
```

Resposta C:

```
safetail [] = []
safetail xs = tail xs
```

Resposta D:

```
safetail []     = []
safetail (_:xs) = xs
```