

Seminário

Linguagens de Programação



HASKELL

Pâmella de Oliveira
Vitor de Nardi Moreschi

Departamento de Informática
Centro Tecnológico
Universidade Federal do Espírito Santo

Junho de 2015

Agenda

1. Introdução
2. Amarrações
3. Valores e tipos de dados
4. Variáveis e constantes
5. Gerenciamento de Memória
6. Expressões e comandos
7. Modularização
8. Módulos
9. I/O
10. Polimorfismo

Agenda

11. Exceções
12. Concorrência
13. Orientação a Objetos
14. Avaliação da Linguagem
15. Referências Bibliográficas

1 - INTRODUÇÃO

Histórico

- 1930: Alonzo Church desenvolveu o cálculo do lambda, um simples, mas poderoso teorema de funções.
- 1950: John McCarthy desenvolveu **Lisp**, a primeira linguagem funcional, com influência da teoria do lambda mas aceitando atribuições de variáveis.
- 1970: Robin Milner e outros desenvolveram a **ML**, a primeira linguagem funcional moderna, com introdução de inferência de tipos e tipos polimórficos.
- 1987: conferência realizada em Amsterdã, a comunidade de programação funcional decidiu **implementar uma linguagem puramente funcional**.
- 1990/1991/1992: **Haskell** versão 1.0/1.1/1.2
- 1996/1997: **Haskell** versão 1.3/1.4

Histórico

- 1999: publicação do **Haskell 98**
- 2003: “sofre” revisão
- 2006: começou o processo de definição de um sucessor do padrão 98, conhecido informalmente por Haskell’ (“**Haskell Prime**”).

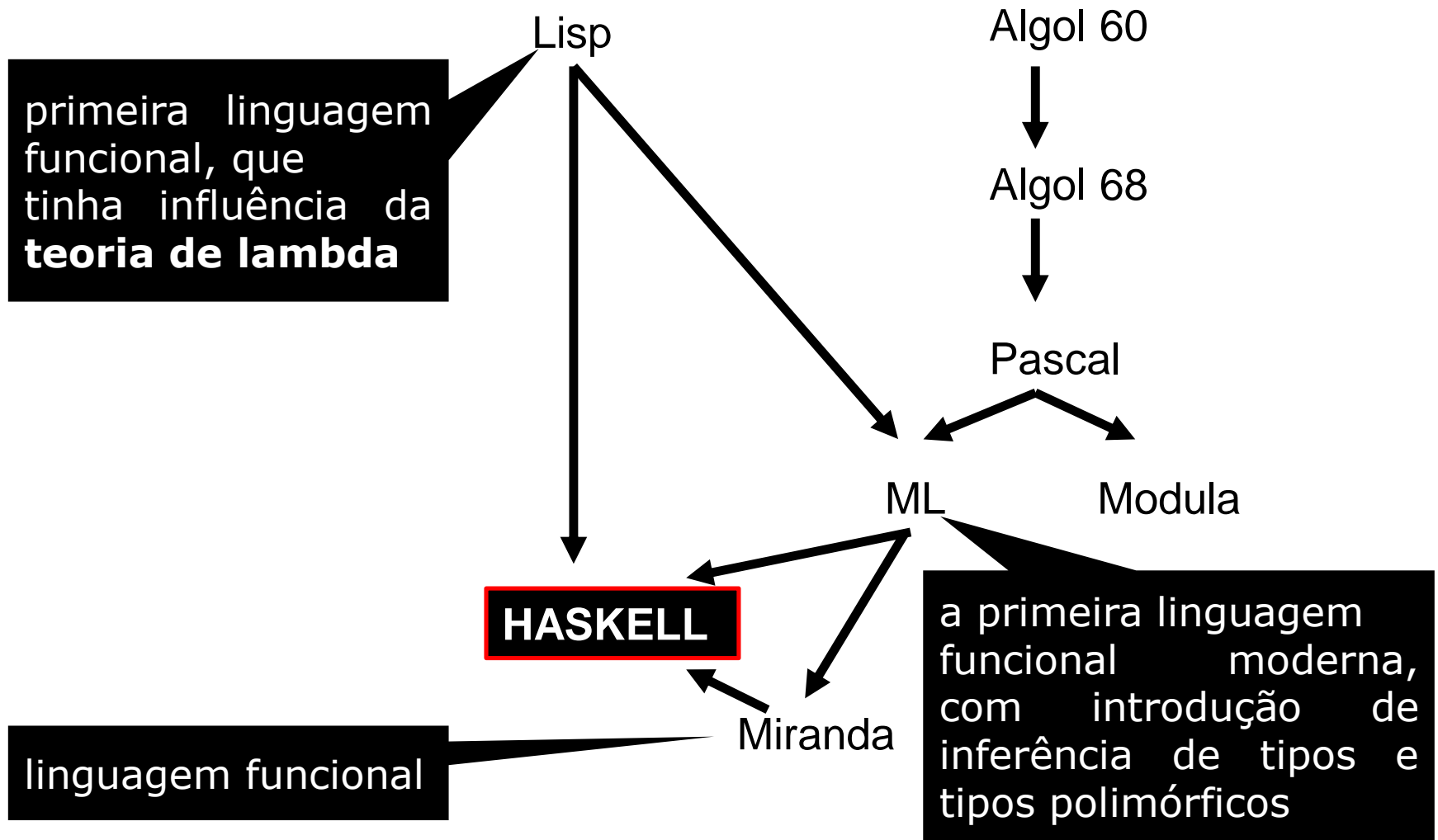
Histórico

- O nome **Haskell** é uma homenagem ao pesquisador **Haskell Brooks Curry**, responsável pela lógica matemática em que a linguagem se baseia.

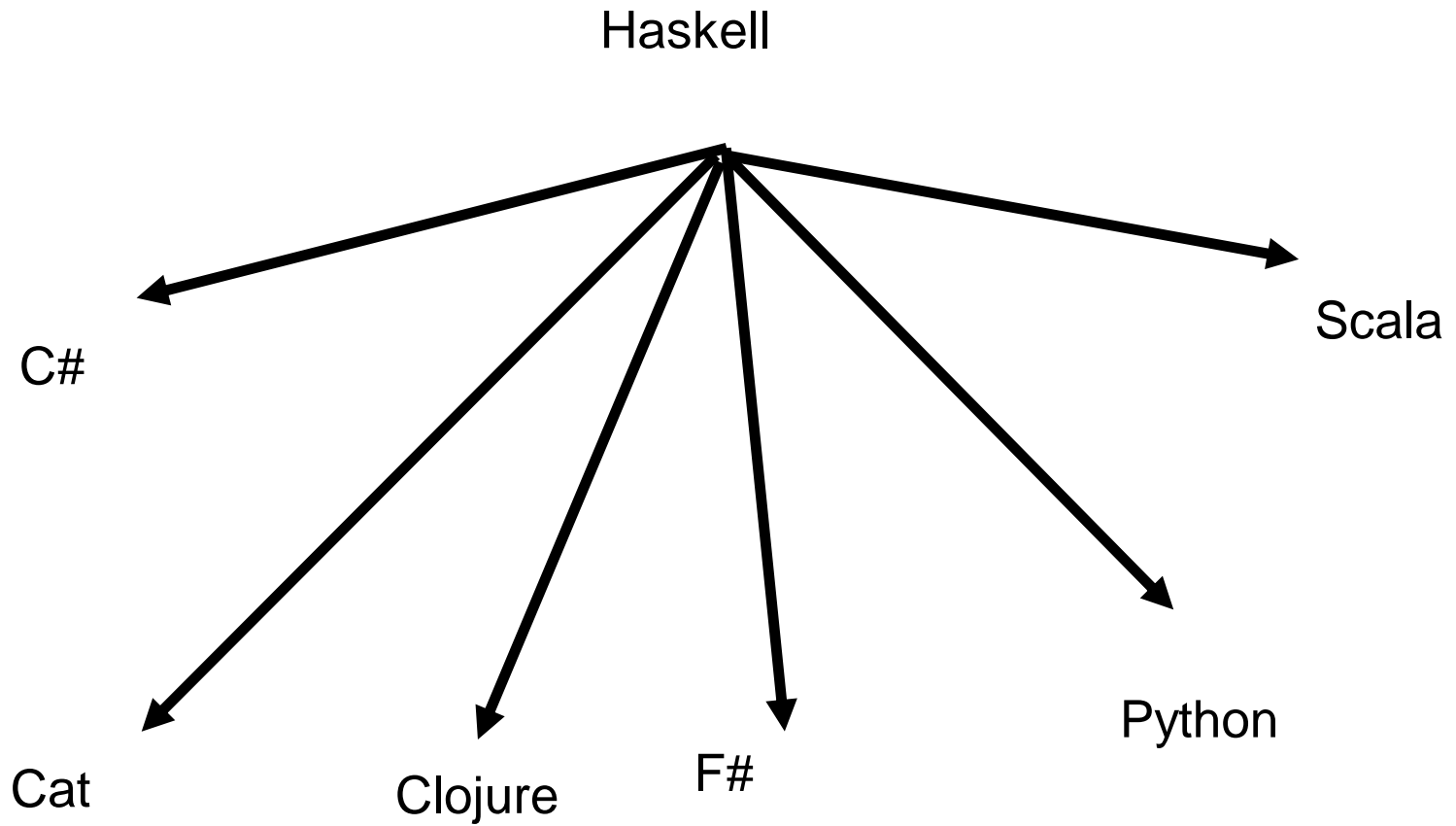


- Haskell é baseada em *cálculo lambda*, por isso o lambda é usado como **logo**.
- Produto de código aberto (*open source*).

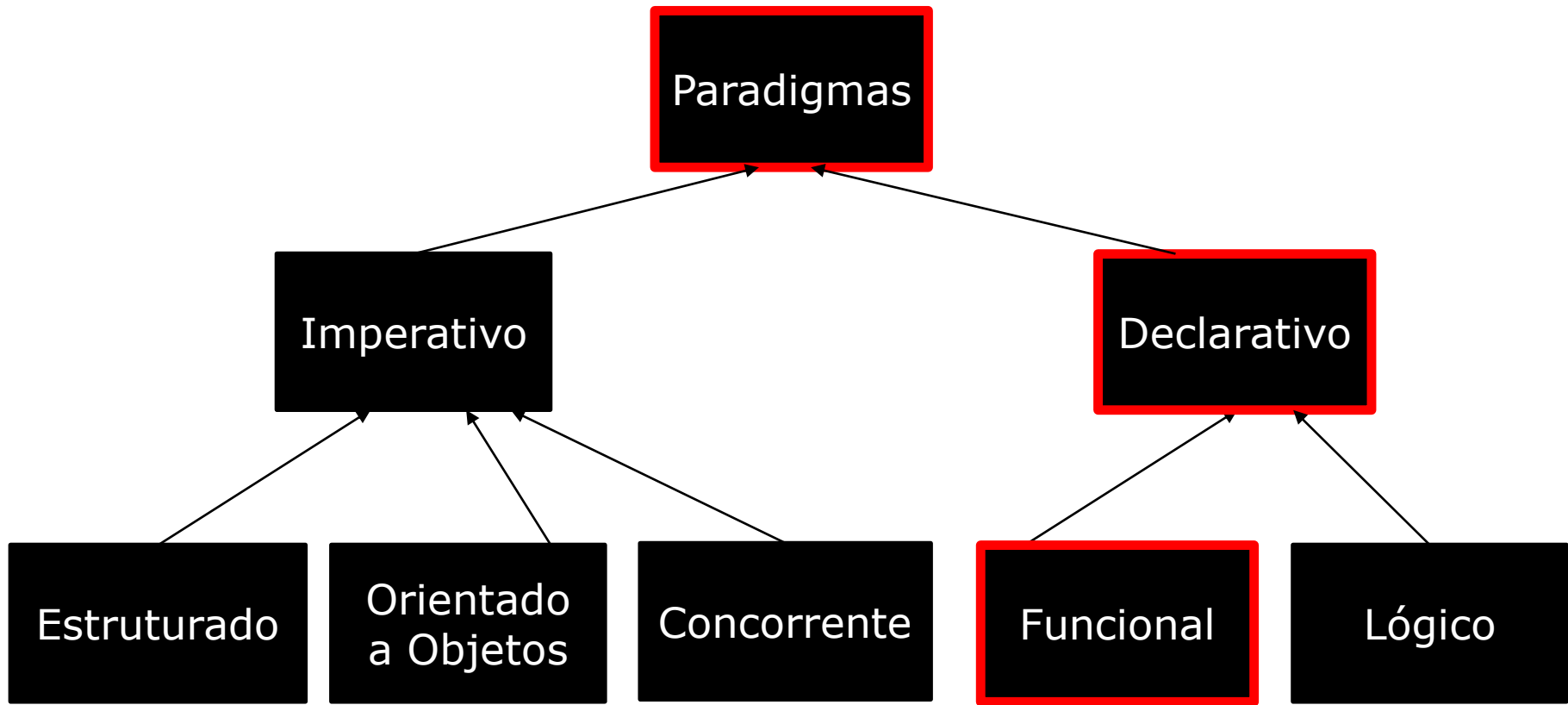
Influenciada por:



Influenciou:



Paradigma



Paradigma Funcional

- Opera apenas sobre funções, as quais recebem listas de valores e retornam um valor.
- **Objetivo da programação funcional:** definir uma função que retorne um valor como a resposta do problema.
- Chamada de função que normalmente chama outras funções para gerar um valor de retorno.
- Características importantes:
 - Principais operações:
 - ✓ Composição de funções;
 - ✓ Chamada recursiva de funções;
 - Funções podem ser passadas como parâmetros a outras funções.

Paradigma Funcional



~~Ponteiros~~

~~Loops~~

~~Variáveis~~

Paradigma Funcional

- **Não existem variáveis**
 - Expressões ✓
- **Não existem comandos**
 - Funções ✓
- **Não existem efeitos colaterais**
 - Declarações ✓
- **Não há armazenamento**
 - Funções de alta ordem ✓
 - Lazy evaluation ✓
 - Recursão ✓
- **Fluxo de controle**
 - Expressões condicionais + Recursão ✓

Por quê Haskell?

- Código menor, mais limpo e mais fácil de dar manutenção;
- Menos erros e maior confiabilidade;
- Um menor **gap semântico** entre o programador e a linguagem;
- Particularmente adequada para programas que precisam ser altamente modicáveis e de fácil manutenção;



Por quê Haskell?

- Grande parte da vida do produto de software é gasto em **especificação, projeto e manutenção** e **não em programação**;
- Programas funcionais são também relativamente fáceis de manter, porque o código é menor.
- Puramente funcional, isto deve-se entre outros ao fato de que a linguagem **não possui efeito colateral**.



Aplicações

- Computação simbólica;
- Processamento de listas;
- Aplicações científicas;
- Aplicações em Inteligência Artificial:
 - Sistemas Especialistas;
 - Representação de conhecimento;
 - Processamento de linguagem natural;
- Jogos;
- Compiladores;

Aplicações

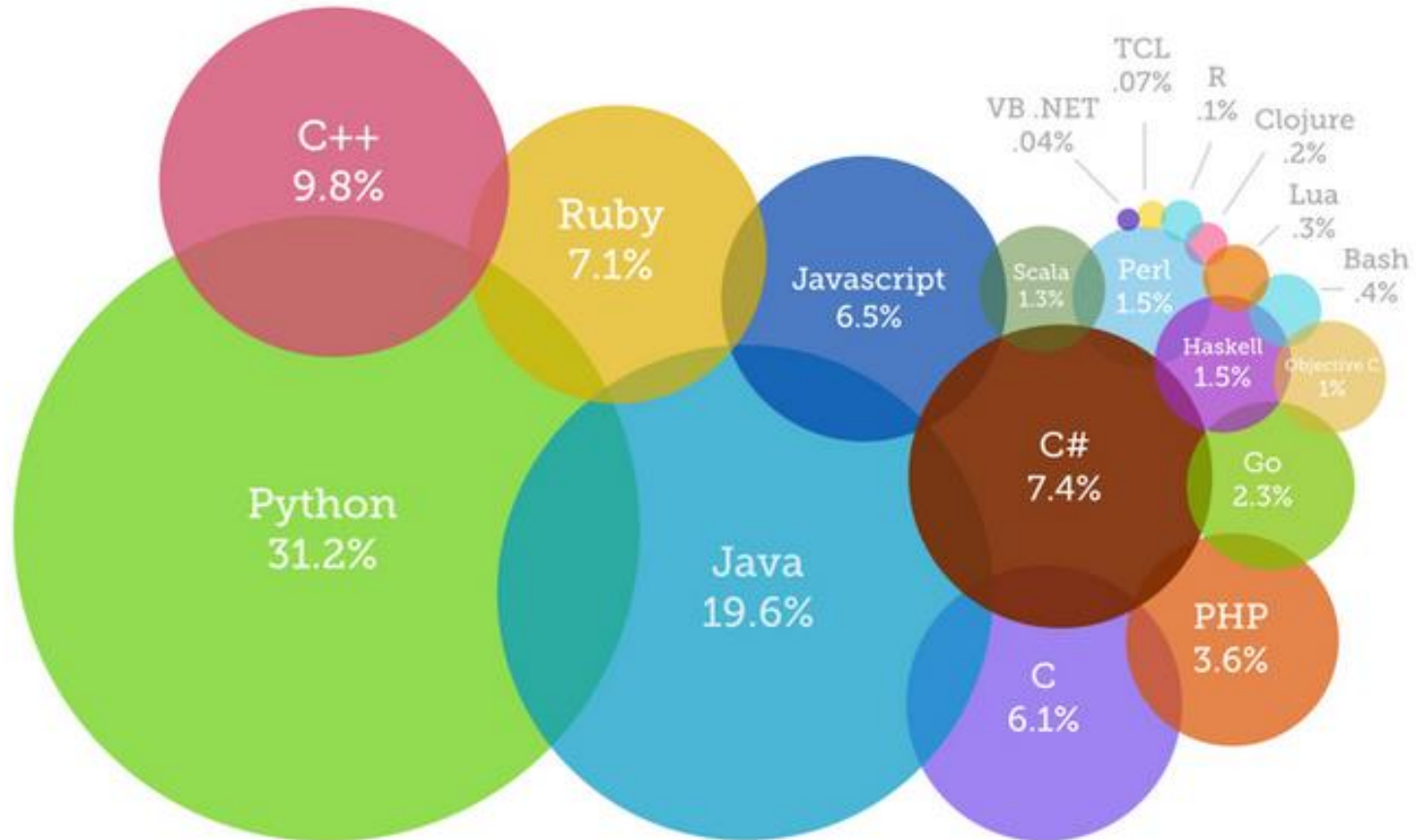
- Functional Reactive Animation (Fran)
- Haskore
 - conjunto de módulos Haskell para a criação , análise e manipulação de música.



Haskell no Mundo



Most Popular Coding Languages of 2015



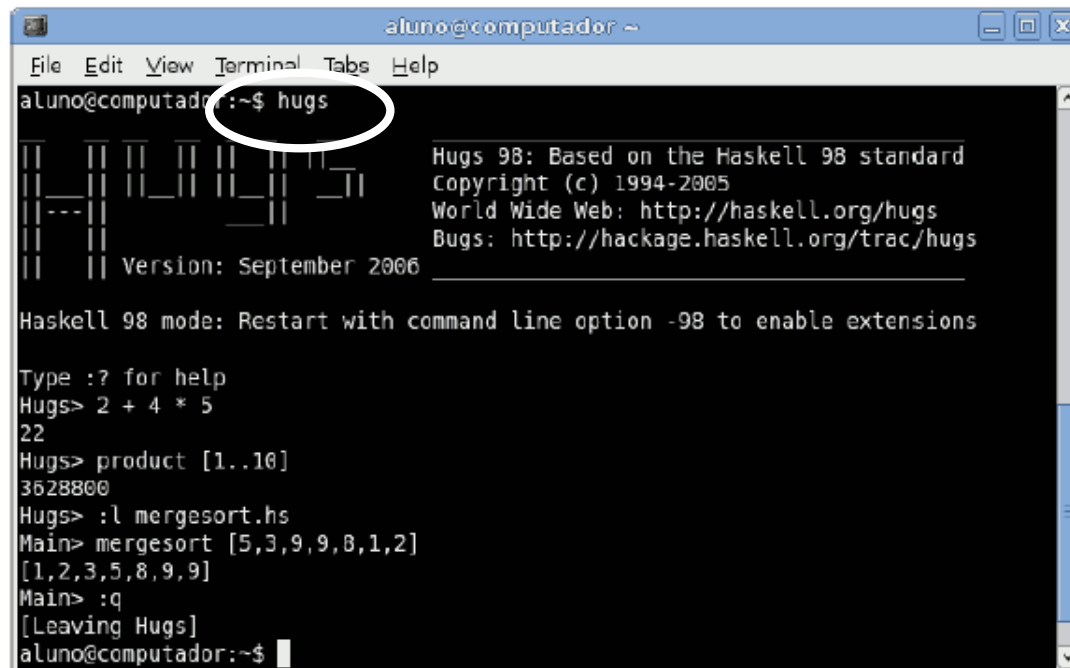
Interessante!

Os programas escritos em Haskell são geralmente chamados de scripts, por isso a extensão normalmente é **"hs"** (Haskell Script).



Método de implementação

- Pode ser **compilado** ou **interpretado**:
 - Hugs: mais usado para fins acadêmicos
 - ✓ Interpretador
 - ✓ Escrito em C
 - ✓ Ideal para iniciantes



```

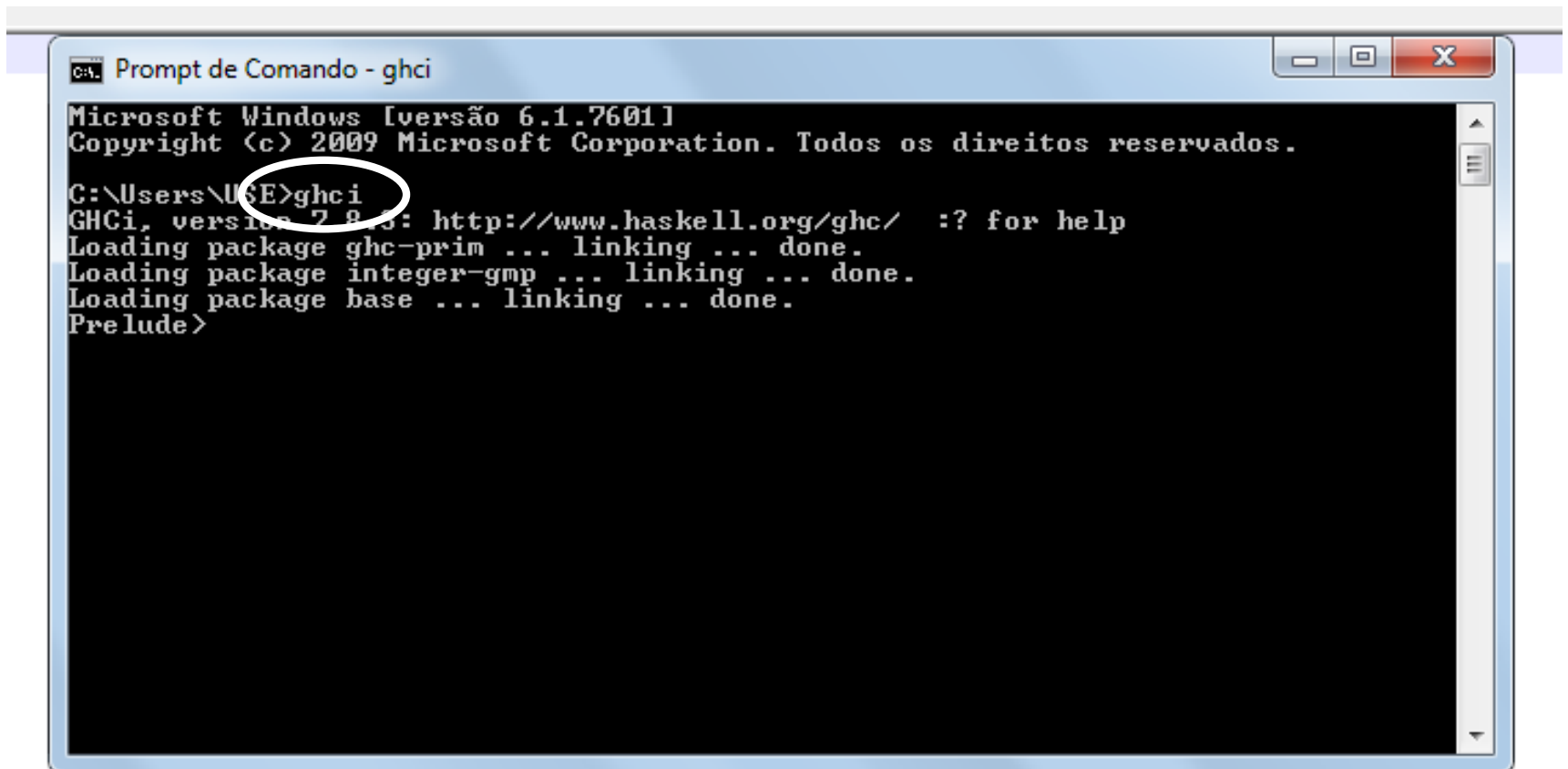
aluno@computador ~
File Edit View Terminal Tabs Help
aluno@computador:~$ hugs
  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
  ||__||  ||__||  ||__||  ||__||  ||__||  ||__||  ||__||  ||__||  ||__||  ||__||
  ||---||  ||---||  ||---||  ||---||  ||---||  ||---||  ||---||  ||---||  ||---||
  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
  ||  ||  Version: September 2006

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

Type :? for help
Hugs> 2 + 4 * 5
22
Hugs> product [1..10]
3628800
Hugs> :l mergesort.hs
Main> mergesort [5,3,9,9,8,1,2]
[1,2,3,5,8,9,9]
Main> :q
[Leaving Hugs]
aluno@computador:~$
  
```

Método de implementação

- GHC: mais usado para produção de software profissional
 - ✓ Interpretador (GHCi) e Compilador
 - ✓ Escrito em Haskell



```
Microsoft Windows [versão 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.

C:\Users\USE>ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

Módulo Prelude

- O arquivo de biblioteca **Prelude.hs** oferece um grande número de funções definidas no padrão da linguagem através do módulo Prelude.
- O módulo **Prelude** é importado automaticamente em todos os módulos de uma aplicação Haskell.
 - Oferece várias funções para manipulação de números
 - Oferece as funções aritméticas familiares
 - ✓ +, -, *, div, mod, /

Módulo Prelude



CA Prompt de Comando - ghci

```
Microsoft Windows [versão 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.

C:\Users\USE>ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> sqrt 2.56
1.6
Prelude> 7*8
56
Prelude> 1+2*3
7
Prelude> (1+2)*3
9
Prelude> div (7*8) 3
18
Prelude> mod (7*8) 3
2
Prelude> (7*8) / 3
18.666666666666668
Prelude> _
```

Módulo Prelude



Além das funções numéricas familiares, o módulo Prelude também oferece muitas funções úteis para a manipulação de listas e outras estruturas de dados.

Funções do módulo Prelude

- O arquivo de biblioteca **Prelude.hs** oferece um grande número de funções definidas no padrão da linguagem através do módulo Prelude.
 - even - verifica se um valor dado é par
 - odd - verifica se um valor dado é impar
 - rem - resto da divisão inteira
 - mod - resto da divisão inteira
 - ceiling - arredondamento para cima
 - floor - arredondamento para baixo
 - round - arredondamento para cima e para baixo
 - truncate - parte inteira do número
 - fromIntegral - converte um inteiro em real
 - sin - seno de ângulo em radianos
 - cos - cosseno
 - tan – tangente

Funções do módulo Prelude

- asin - arco seno
- acos - arco cosseno
- atan - arco tangente
- atan - arco tangente
- abs - valor absoluto
- sqrt - raiz quadrada, valor positivo apenas
- exp - exponencial base e
- log - logaritmo natural (base e)
- logBase - logaritmo na base dada
- min - menor de 2 objetos dados
- max - maior de 2 objetos dados

Primeiro programa



“hs”

“indica que o texto passado como argumento para a função deve ser impresso na tela”

```
1 primeiroprograma = putStrLn "Hello world"
```

len Ln:1 Col:1 Sel:0|0 Dos\Windows ANSI as UTF-8 INS

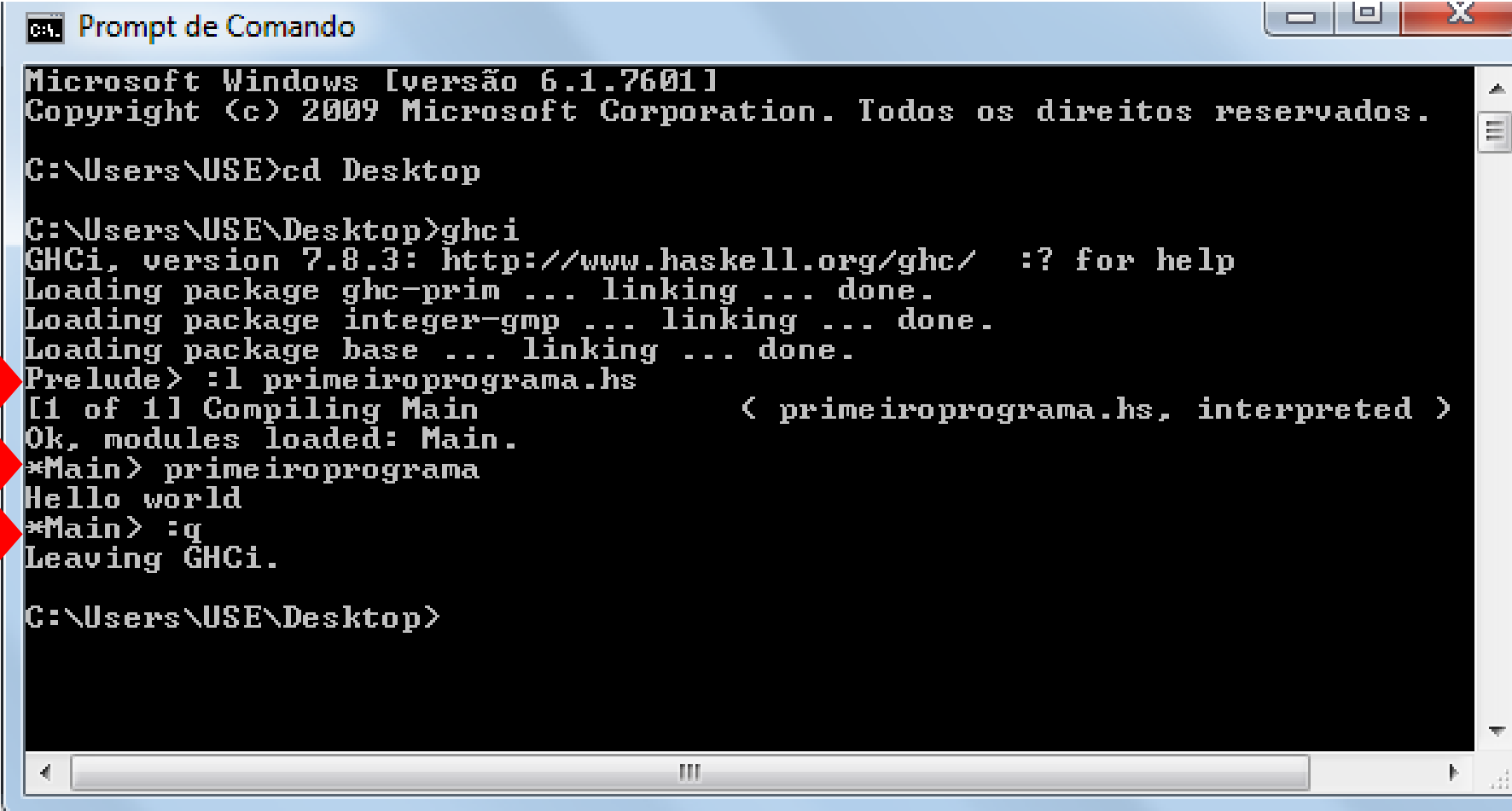
Primeiro programa

:load ou **:l** carrega um programa

:quit ou **:q** ou **Ctrl-D**: sai do interpretador

:reload recarrega um programa

Comandos básicos



```
Microsoft Windows [versão 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.

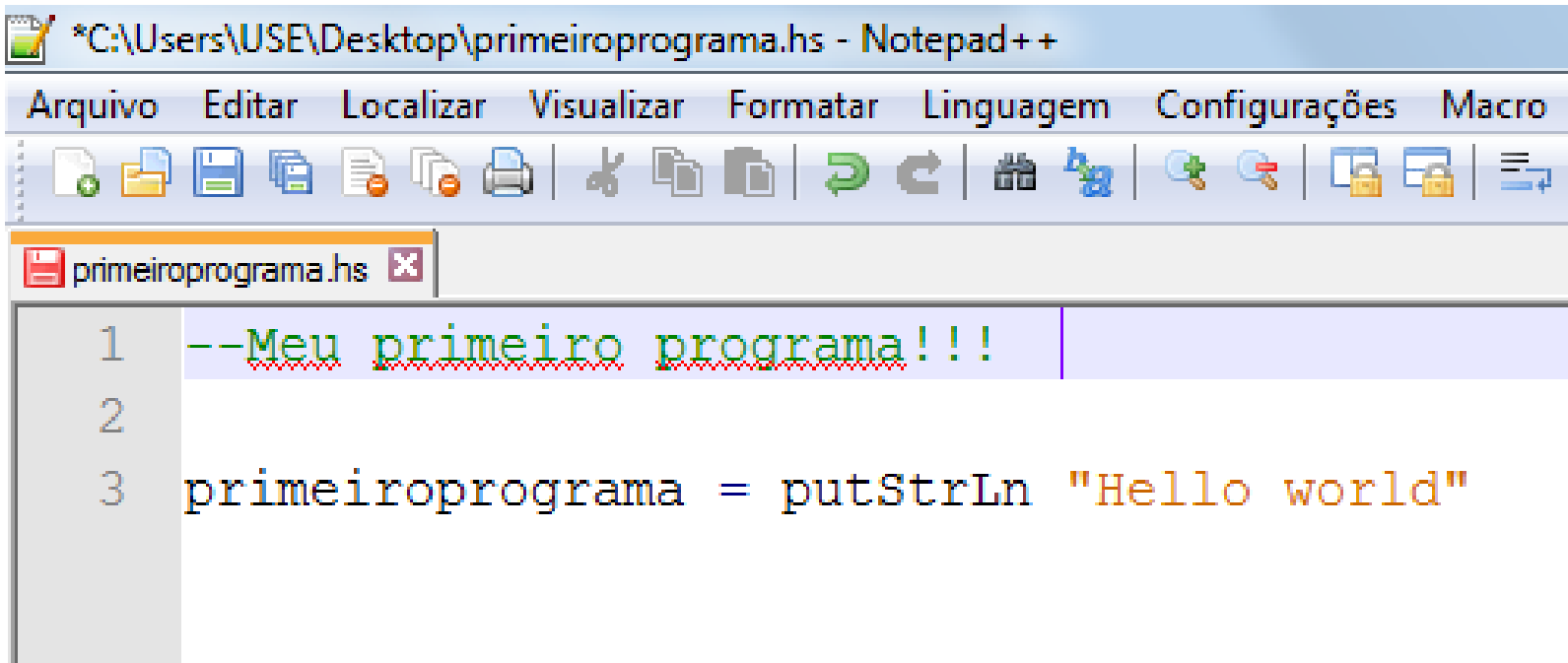
C:\Users\USE>cd Desktop

C:\Users\USE\Desktop>ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l primeiroprograma.hs
[1 of 1] Compiling Main                < primeiroprograma.hs, interpreted >
Ok, modules loaded: Main.
*Main> primeiroprograma
Hello world
*Main> :q
Leaving GHCi.

C:\Users\USE\Desktop>
```

Comentários

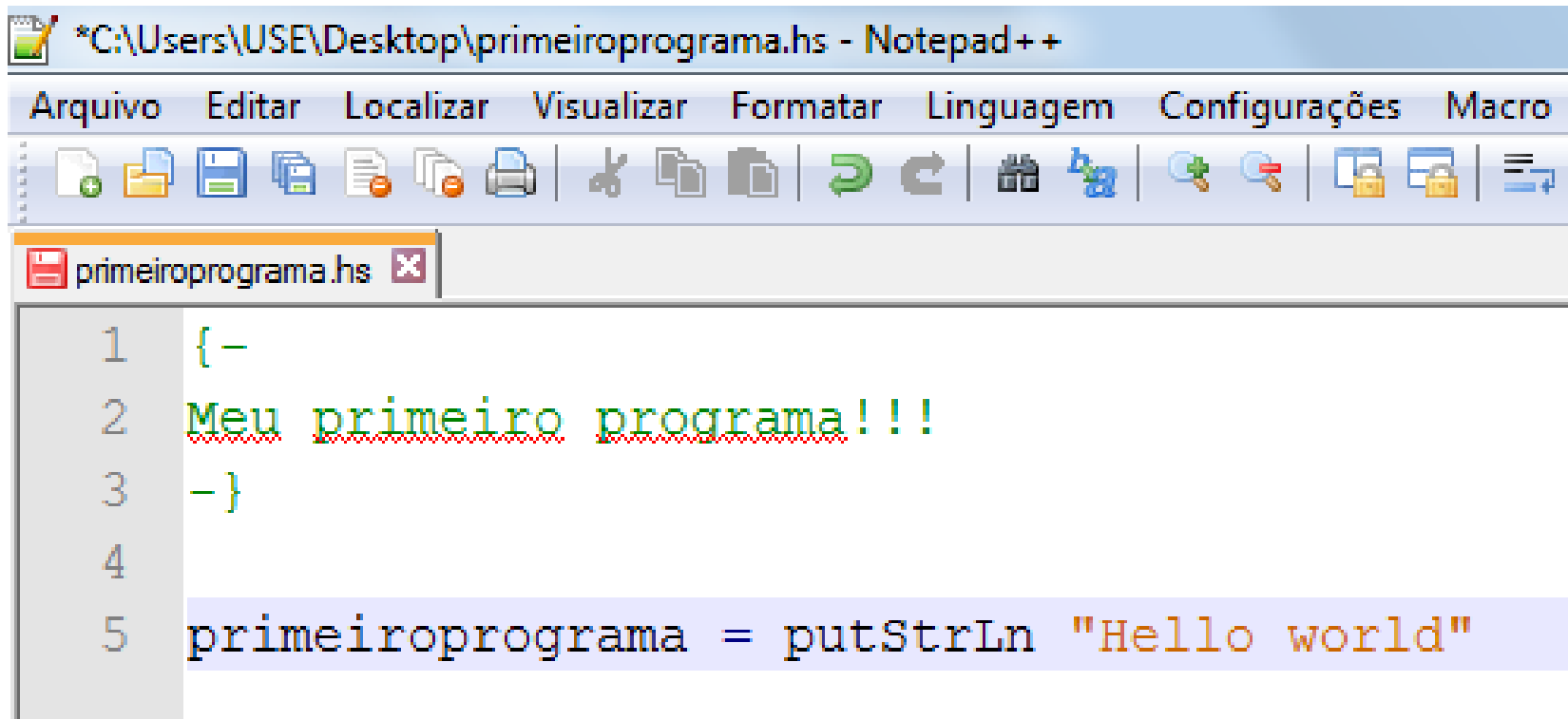
- **Comentário de linha:** introduzido por "--" e se estende até o final da linha.



```
*C:\Users\USE\Desktop\primeiroprograma.hs - Notepad++
Arquivo  Editar  Localizar  Visualizar  Formatar  Linguagem  Configurações  Macro
primeiroprograma.hs
1  --Meu primeiro programa!!!
2
3  primeiroprograma = putStrLn "Hello world"
```

Comentários

- **Comentário de bloco:** delimitado por {- e -}.



The image shows a Notepad++ window titled "*C:\Users\USE\Desktop\primeiroprograma.hs - Notepad++". The menu bar includes "Arquivo", "Editar", "Localizar", "Visualizar", "Formatar", "Linguagem", "Configurações", and "Macro". The toolbar contains various icons for file operations and editing. The active document is "primeiroprograma.hs". The code content is as follows:

```
1 {-  
2 Meu primeiro programa!!!  
3 -}  
4  
5 primeiroprograma = putStrLn "Hello world"
```

Regra de Layout

- Em uma sequência de definições, cada definição deve começar precisamente na **mesma coluna**.

```

C:\Users\USE\Desktop\layout
Arquivo  Editar  Localizar  V
layout.hs x
1 a = 10
2 b = 20
3 c = 30

```

```

Prelude> :l layout
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main>

```

```

C:\Users\USE\Desktop\layout.hs -
Arquivo  Editar  Localizar  Visualiza
layout.hs x
1 a = 10
2   b = 20
3 c = 30
4

```

```

Ok, modules loaded: Main.
*Main> :l layout
[1 of 1] Compiling Main
layout.hs:2:11: parse error on input '='
Failed, modules loaded: none.
Prelude>

```

```

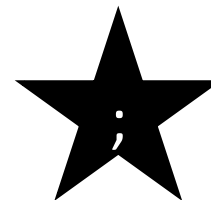
C:\Users\USE\Desktop\layout.hs - N
Arquivo  Editar  Localizar  Visualiza
Executar  Plugins  Janela  ?
layout.hs x
1   a = 10
2 b = 20
3   c = 30
4

```

```

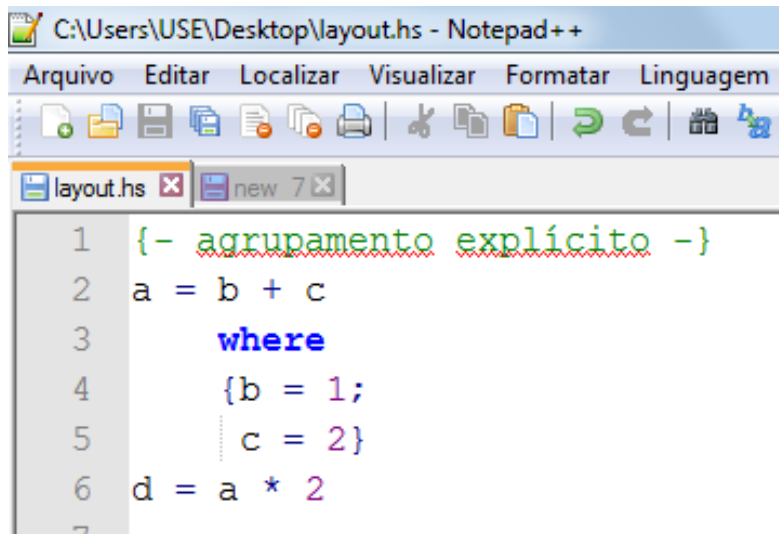
Prelude> :l layout
[1 of 1] Compiling Main
layout.hs:2:1: parse error on input 'b'
Failed, modules loaded: none.
Prelude>

```



Regra de Layout

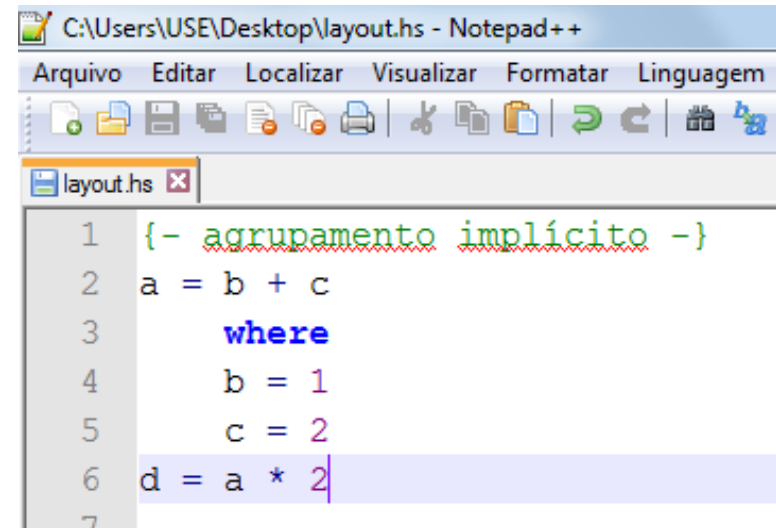
- A regra de layout evita a necessidade de uma sintaxe explícita para indicar o agrupamento de definições



```

1  {- agrupamento explícito -}
2  a = b + c
3      where
4      {b = 1;
5      .....
6      c = 2}
7  d = a * 2

```



```

1  {- agrupamento implícito -}
2  a = b + c
3      where
4      b = 1
5      c = 2
6  d = a * 2
7

```


2 - AMARRAÇÕES

Conceito

“é uma associação entre entidades de programação, como entre uma variável e seu valor, ou entre um **identificador** e um **tipo**.” (VAREJÃO, Flávio)

- Cada declaração produz uma ou mais amarrações.
- Um ambiente de referenciamento ou **namespaces** é um conjunto de amarrações.



Falaremos
depois

Identificadores



“Cadeias de caracteres definidos pelos programadores para servirem de referências às entidades de computação.”

Identificadores em Haskell

- Sem limites para identificadores
- Os identificadores necessariamente devem começar com uma letra maiúscula ou minúscula, seguida por uma sequência opcional de letras, dígitos, sublinhas ou apóstrofes.
- **É case-sensitive;**
- Nomes de função devem começar com uma letra minúscula.
 - Exemplos: `myFun`, `fun1`, `arg_2`, `x'`
- Por convenção, **uma lista de elementos** normalmente têm um sufixo **s** em seu nome, que indica plural.
 - Exemplos: **`xs`**, **`ns`**, **`nss`**

Palavras reservadas

- Identificadores com vocábulo especial para a LP, não podem ser usados pelo programador para criação de identificadores de entidades.

case

class

data

deriving

do

else

if

import

in

infix

infixl

infixr

instance

let

of

module

newtype

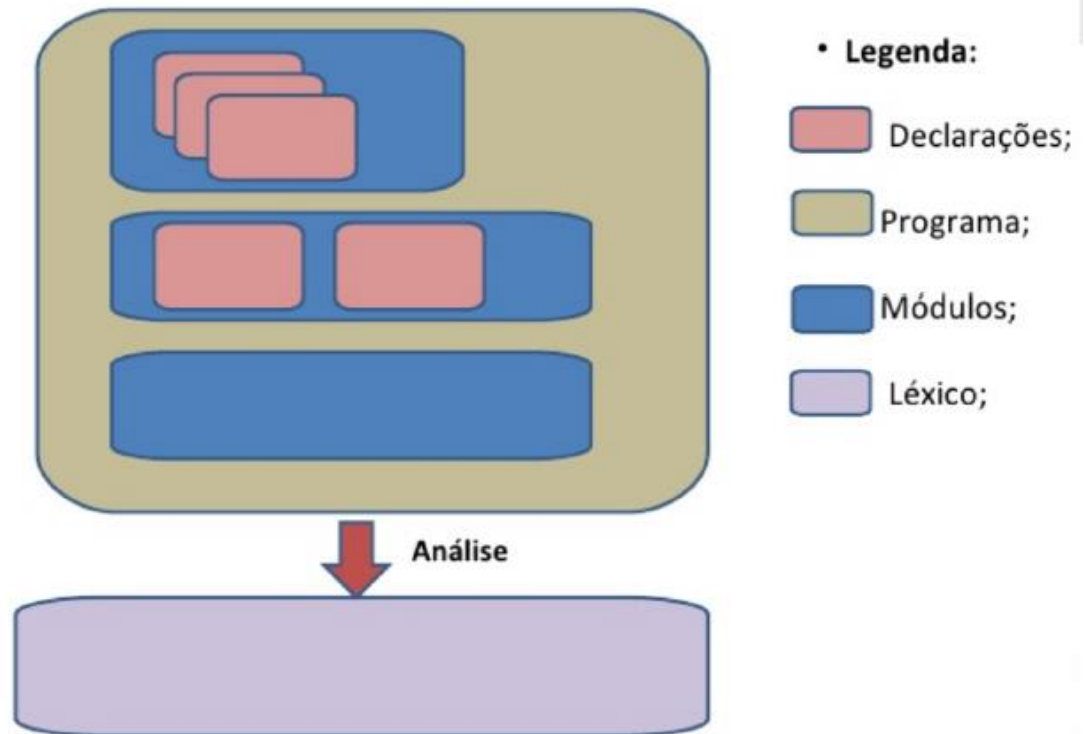
then

type

where

Estrutura abstrata em Haskell

- 4 níveis
 - Conjunto de **módulos**: Os módulos oferecem uma maneira de controlar os namespaces;
 - Um módulo é composto de: Uma coleção de **declarações** (definições de tipos de dados, classes e tipos de informação);
 - Uma **expressão** denota um valor e um tipo; Haskell é composto de expressões;
 - **Estrutura léxica**: capta a representação concreta dos programas Haskell em arquivos texto.



Escopo de visibilidade de uma amarração

- Estático:
 - O conceito de **BLOCO** é fundamental para o entendimento. Um bloco delimita o escopo de qualquer amarração que ele pode conter.
- Variáveis assumem um valor quando são criadas e este valor nunca muda dentro do escopo da mesma.




Definições e Declarações

- “frases de programa elaboradas para produzir amarrações.”
 - Definições -> produzem amarrações entre identificadores e entidades criadas na própria definição;
 - Declarações -> produzem amarrações entre identificadores e entidades já criadas ou ainda por criar;
 - Não existe atribuição em Haskell, e sim **definição**. Uma variável é definida.
 - O comando de definição é o sinal “ = ”.
 - ✓ **Ex.:** $x = 2$
 - ✓ **Ex.:** $y = \text{“string”}$

Definindo funções

- Além de poder usar as funções das bibliotecas, o programador também pode definir e usar suas próprias funções. Novas **funções são definidas dentro de um script**, um arquivo texto contendo **definições** (de variáveis, funções, tipos, etc.).



Veremos
depois...

Definições locais a uma equação

- Em Haskell equações são usadas para definir variáveis e funções, como discutido anteriormente. Em muitas situações é desejável poder definir valores e funções auxiliares em uma definição principal. Isto pode ser feito escrevendo-se uma cláusula **where** ao final da equação. Uma cláusula **where** é formada pela palavra chave **where** seguida das definições auxiliares.
- A cláusula where **faz definições locais à equação**, ou seja o escopo dos nomes definidos em uma cláusula where restringe-se à equação contendo a cláusula where, podendo ser usados:
 - nas guardas da equação principal (quando houver)
 - nos resultados (expressões que ocorrem no lado direito) da equação principal
 - nas próprias definições locais da cláusula where

Definições locais a uma equação

```

C:\Users\USE\Desktop\areaTriangulo.hs - Notepad++
Arquivo  Editar  Localizar  Visualizar  Formatar  Linguagem  Configurações  Macro  Executar  Plugins  Janela
?
[Icons]
areaTriangulo.hs x
1 areaTriangulo a b c = sqrt (s * (s-a) * (s-b) * (s-c))
2   where
3     s = (a + b + c)/2
4
length: 94  lines: 4  Ln: 3  Col: 9  Sel: 0 | 0  Dos\Windows  ANSI as UTF-8  INS

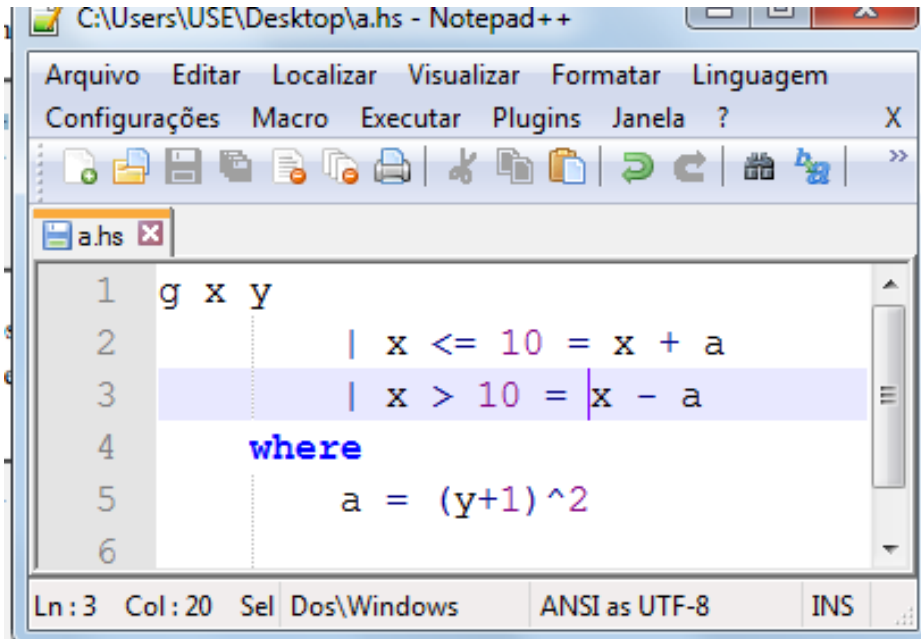
```

```

areaTriangulo 5 6 8
~> sqrt (s * (s-5) * (s-6) * (s-8))
  where
    s = (5 + 6 + 8)/2 = 9.5
~> sqrt (9.5 * (9.5-5) * (9.5-6) * (9.5-8))
~> sqrt 224.4375
~> 14.981238266578634

```

Definições locais a uma equação



```

1 g x y
2     | x <= 10 = x + a
3     | x > 10 = x - a
4     where
5         a = (y+1)^2
6
Ln: 3 Col: 20 Sel Dos\Windows ANSI as UTF-8 INS

```

- O escopo de **a** inclui os dois possíveis resultados determinados pelas guardas.

Definições locais a uma expressão

- Também é possível fazer definições locais a uma expressão escrevendo-se uma expressão **let**. Uma expressão **let** é formada por uma lista de definições mutuamente recursivas, e por um corpo (que é uma expressão), introduzidos pelas palavras chave **let** e **in**:

let *definições* **in** *expressão*

- O escopo dos nomes definidos em uma expressão **let** restringe-se à própria expressão **let**, podendo ser usados:
 - no corpo da expressão **let**
 - nas próprias definições locais da expressão **let**
- O tipo de uma expressão **let** é o tipo do seu corpo. O valor de uma expressão **let** é o valor do seu corpo, calculado em um contexto que inclui os nomes introduzidos nas definições locais.

Definições locais a uma expressão

```

let x = 3+2 in x^2 + 2*x - 4
let x = 3+2 ; y = 5-1 in x^2 + 2*x - y
let quadrado x = x*x in quadrado 5 + quadrado 3
  
```

```

Loading package base ... linking ... done.
Prelude> let x = 3+2 in x^2 + 2*x-4
31
Prelude> let x = 3+2; y = 5-1 in x^2 + 2*x -y
31
Prelude> let quadrado x = x*x in quadrado 5 + quadrado 3
34
Prelude>
  
```

Diferenças entre **let** e **where**

- Com **where** as definições são colocadas no final, e com **let** elas são colocadas no início.
- **let** é uma expressão e pode ser usada em qualquer lugar onde se espera uma expressão.
- Já **where** não é uma expressão, podendo ser usada apenas para fazer definições locais em uma definição de função.

Declaração de tipo algébrico

- Como definir?
 - Dias da semana
 - Estações do ano
- “Um tipo algébrico é um tipo onde são especificados a forma de cada um dos seus elementos.”

Declaração de tipo algébrico

```

data cx => T u1 ... uk = C1 t11 ... t1n1
      | Cm tm2 ... tmnm
  
```

- cx é um contexto
- $u_1 \dots u_k$ são variáveis de tipo
- T é o **construtor de tipo**
- $T u_1 \dots u_k$ é um novo tipo introduzido pela declaração data
- C_1, \dots, C_m são **construtores de dados**
- t_{ij} são tipos
- Construtores de tipo e construtores de dados são identificadores alfanuméricos começando com letra maiúscula, ou identificadores simbólicos.

Declaração de tipo algébrico

- Vamos ver como o tipo Bool é definido na biblioteca padrão:
 - **data** Bool = False | True
- data significa que estamos definindo um novo tipo de dados.
- A parte anterior a = diz o tipo, que é Bool.
- As partes depois são **value constructors** (*construtores de valores*). Elas especificam os diferentes valores que o tipo pode assumir. O | pode ser lido como *ou* (OR). Então você pode ler tudo como: “o tipo Bool pode ter valores True ou False. Ambos, *nome do tipo* e *construtores de valores* devem começar com *letras maiúsculas*.”

Declaração de tipo algébrico

- Definição de um novo tipo para representar cores:
 - data Cor = Azul | Amarelo | Verde | Vermelho
 - ✓ Construtor de tipo: Cor
 - ✓ Construtores de dados deste tipo são:
 - Azul :: Cor
 - Amarelo :: Cor
 - Verde :: Cor
 - Vermelho :: Cor

- Definição de um novo tipo para representar horários:
 - data Horario = AM Int Int Int | PM Int Int Int
 - ✓ Construtor de tipo: Horário
 - ✓ Construtores de dados deste tipo são:
 - AM :: Int -> Int -> Int -> Horario
 - PM :: Int -> Int -> Int -> Horario

3 - VALORES E TIPOS DE DADOS

Valor

“Um valor é qualquer entidade que existe durante uma computação, isto é, tudo que pode ser avaliado, armazenado, incorporado numa estrutura de dados, passado como argumento para um procedimento ou função, retornado como resultado de funções, etc.” (VAREJÃO, Flávio)

Tipo

“Um tipo de dado é um conjunto cujos valores exibem comportamento uniforme nas operações associadas com o tipo.” (VAREJÃO, Flávio)

Tipos Primitivos

- “A partir deles é que todos os demais tipos podem ser construídos.” (VAREJÃO, Flávio)
1. Tipos Numéricos:
 - Inteiros:
 - ✓ Int, Integer;
 - Reais:
 - ✓ Float, Double;
 2. Caracter: Char
 3. Lógico: Bool
 4. Vazio: Void
 5. Unitário: Unit (implementação do conjunto 1)

Tipos Numéricos

- Inteiros:
 - Int:
 - ✓ trabalha com um intervalo de valores fixo e reduzido e tem como vantagem economizar memória do computador e tempo de processamento.
 - ✓ **Ex.:** Em máquinas 32 bits, por exemplo, o valores variam entre **2147483647 a -2147483648**
 - Integer:
 - ✓ pode produzir números que podem ter uma quantidade ilimitada de algarismos, entretanto, como a memória do computador é finita, qualquer que seja a maquina real que estivermos usando, inevitavelmente esbarraremos em limites.
 - ✓ **Ex.:** **28904374642004**

Tipos Numéricos

- Reais:
 - Float:
 - ✓ Ponto flutuante de precisão simples
 - ✓ Ex.: **3.0**
 - Double:
 - ✓ Ponto flutuante de precisão dupla
 - ✓ Ex.: **3.23433**

Caracter: Char / Lógico: Bool

- Char -> Caracter
 - **Ex.:** 'a'
- Bool -> Booleano
 - **Ex.:** True / False

Tipos compostos

- “aqueles criados a partir de tipos mais simples.” (VAREJÃO, Flávio)
1. Strings
 2. Tuplas
 3. Listas

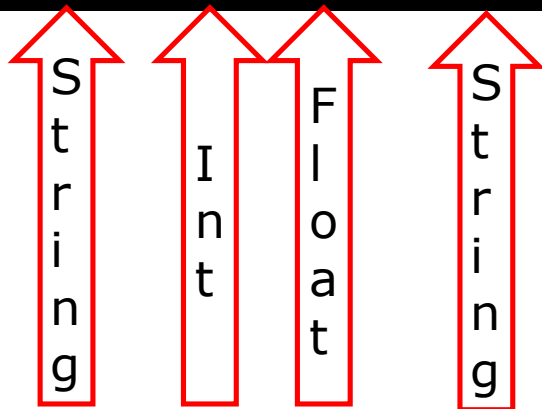
Strings

- Cadeia de caracteres, representada sob forma de lista de Char. A sintaxe tanto pode ser de lista quanto a abreviação, entre aspas.
- Definido como um conjunto sequencial de caracteres textuais da tabela ASCII.
 - **Ex.:** “Pamella”
 - **Ex.:** ['P','a','m','e','l','l','a']

Tuplas – Produto cartesiano em Haskell

- Permite a definição e o uso de tipos de dados heterogêneos sob uma estrutura relacional.
- **Objetivo:** definir uma função que receba ou retorne mais de um valor e, geralmente, de tipo heterogêneo.
- Estrutura estática: Uma vez criada, não pode ser modificada. Representada entre ().

```
Prelude> (1,3)
(1,3)
Prelude> ("Vitor", 21, 65.23, "Azul")
("Vitor", 21, 65.23, "Azul")
```



A função **fst** extrai o primeiro elemento da tupla.
A função **snd** extrai o segundo elemento da tupla.

Obs.: somente no caso das tuplas-2

Mapeamento

- Funções genéricas sobre listas aplicando alguma regra geral sobre os elementos de uma lista.
- Num mapeamento, uma função é aplicada a cada elemento de uma lista, de modo que uma nova lista modificada é retornada.
- Por exemplo: Dobrar uma lista de inteiros.

Mapeamento



```
dobro.hs x
1  dobrar :: [Int] -> [Int]
2  dobrar [] = []
3  dobrar (x:xs) = x*2 : dobrar xs
```



```
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l dobro.hs
[1 of 1] Compiling Main                ( dobro.hs, interpreted )
Ok, modules loaded: Main.
*Main> dobrar [1,3,5]
[2,6,10]
*Main> _
```

Listas

- Uma lista é uma estrutura de dados que representa uma coleção de objetos homogêneos em sequência.
- Permite que seus elementos sejam consultados recuperando todos os elementos anteriores a ele.
- Estas não permitem modificar um elemento no meio da lista sem que um acesso sequencial seja realizado até tal elemento.
- O último “nó” da lista sempre contém uma lista vazia. Em Haskell, uma lista vazia representada pelo símbolo `[]` é a estrutura base da existência de uma lista.

Listas

- Uma lista é composta sempre de dois segmentos: **cabeça** (head) e **corpo** (tail). A cabeça da lista é sempre o primeiro elemento.
- Representada entre [].

['a', 'b', 'c', 'd'] → 'a': ['b', 'c', 'd']

primeiro elemento *corpo da lista*

Listas – Interessante!!!

```

c:\ Prompt de Comando - ghci
Microsoft Windows [versão 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.

C:\Users\USE>ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> [1,2,3] == (1:2:3)_

```

```

c:\ Prompt de Comando - ghci

Prelude> [1,2,3] == (1:2:3:[1])
True
Prelude>

```

Criando Lista



```
Prompt de Comando - ghci
Microsoft Windows [versão 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.

C:\Users\USE>ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> let lista = [4,8,15,16,23,48]
Prelude> lista
[4,8,15,16,23,48]
Prelude> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
Prelude> ['o','i','t'] ++ ['o','r']
"oitor"
Prelude> "Seminarario" ++ " " ++ "LP"
"Seminarario LP"
Prelude> 5 : [1,2,3,4,5]
[5,1,2,3,4,5]
Prelude>
```

Cria Lista

Concatena

Operações com Listas

```
Prompt de Comando - ghci
Microsoft Windows [versão 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.

C:\Users\USE>ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> "Steve Buscemi" !! 6
'B'
Prelude> [9.4,33.2,96.2,11.2,23.25] !! 4
23.25
Prelude> _
```

Obter elemento por índice

Operações com Listas

```

Prompt de Comando - ghci
Microsoft Windows [versão 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.

C:\Users\USE>ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> let b = [[1,2,3,4],[5,6,7],[8,9,10]]
Prelude> b
[[1,2,3,4],[5,6,7],[8,9,10]]
Prelude> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,6,7],[8,9,10],[1,1,1,1]]
Prelude> [6,6,6,6] : b
[[6,6,6,6],[1,2,3,4],[5,6,7],[8,9,10]]
Prelude> b !! 2
[8,9,10]
Prelude> _

```

Lista de Lista

Operações com Listas



▪ Funções de operações de listas:

head []

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

Retorna o primeiro elemento da lista.

tail

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

Retorna a calda da lista, ou, não exibe o primeiro elemento.

last

```
ghci> last [5,4,3,2,1]  
1
```

Retorna o último elemento da lista.

init

```
ghci> init [5,4,3,2,1]  
[5,4,3,2]
```

Retorna todos os elementos da lista menos o último.

Operações com Listas



length

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

Retorna o comprimento da lista.

reverse

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

Inverte os elementos da lista.

elem

```
ghci> 4 `elem` [3,4,5,6]  
True
```

Verifica se um elemento está presente na lista.

```
ghci> 10 `elem` [3,4,5,6]  
False
```

Operações com Listas



```
null  
ghci> null [1,2,3]  
False  
ghci> null []  
True
```

**Retorna True se a
lista for vazia**

```
maximum  
ghci> maximum [1,9,2,3,4]  
9
```

**Retorna maior
elemento da lista**

```
minimum  
ghci> minimum [8,4,2,1,5,6]  
1
```

**Retorna menos
elemento da lista**

Operações com Listas



```
sum  
ghci> sum [5,2,1,6,3,2,5,7]  
31
```

**Retorna a soma dos
elementos da lista**

```
product  
ghci> product [6,2,1,2]  
24
```

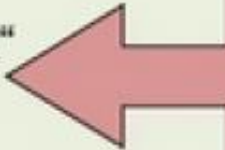
**Retorna o produto
dos elementos da
lista**

Operações com Listas - Rangers

```
ghci> [1..20]  
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

```
ghci> ['a'..'z']  
"abcdefghijklmnopqrstuvwxyz"
```

```
ghci> ['K'..'Z']  
"KLMNOPQRSTUVWXYZ"
```



Selecionando intervalo de elementos em listas.

Operações com Listas – Compreensão de listas (List Comprehension)



- Uma lista pode ser especificada pela definição de seus elementos. Uma maneira alternativa para construir e manipular listas é a definição por compreensão de listas. A compreensão de listas é feita com um construtor de listas que utiliza conceitos e notações da teoria dos conjuntos.
- Por exemplo, seja o conjunto A definido por:
 - $A = \{x^2 \mid x \in \mathbb{N} \wedge x \text{ é par}\}$
 - ✓ Este exemplo define o conjunto dos quadrados dos números pares

**Em Haskell pode ser representado por uma lista
definida da seguinte maneira ...**

Operações com Listas – Compreensão de listas (List Comprehension)

```

C:\Users\USE\Desktop\listaPares.hs - Notepad++
Arquivo  Editar  Localizar  Visualizar  Formatar  Linguagem  Configurações  Macro  Executa
listaPares.hs x
1  constroi_lista = [x*x | x <- [9 .. 39], par x]
2  par :: Int -> Bool
3  par x = mod x 2 == 0
  
```

gerador de números de 39 a 99,
em um intervalo default de 1

```

In an equation for `it': it = par [1, 2, 3]
*Main> constroi_lista
[100,144,196,256,324,400,484,576,676,784,900,1024,1156,1296,1444]
*Main>
  
```

Operações com Listas – Compreensão de listas (List Comprehension)



```
Prelude> [x | x <- [1,4 .. 10]]  
[1,4,7,10]
```

- É possível existir mais de um gerador em uma mesma compreensão de lista:

```
Prelude> [(x,y) | x <- [1 .. 3], y <- ['a','b','c']  
[(1,'a'), (1,'b'), (1,'c'), (2,'a'), (2,'b'), (2,'c'), (3,'a'), (3,'b'), (3,'c')]
```

Enumeração



```
1 data Dia = Domingo
2           | Segunda
3           | Terça
4           | Quarta
5           | Quinta
6           | Sexta
7           | Sábado
8           deriving (Eq, Show)
9
10 dia :: Int -> Dia
11 dia num | (num == 1) = Domingo
12          | (num == 2) = Segunda
13          | (num == 3) = Terça
14          | (num == 4) = Quarta
15          | (num == 5) = Quinta
16          | (num == 6) = Sexta
17          | (num == 7) = Sábado
```

```
Prompt de Comando - ghci
*Main> dia 1
Domingo
*Main> dia 7
Sábado
*Main> :t Domingo
Domingo :: Dia
*Main> _
```

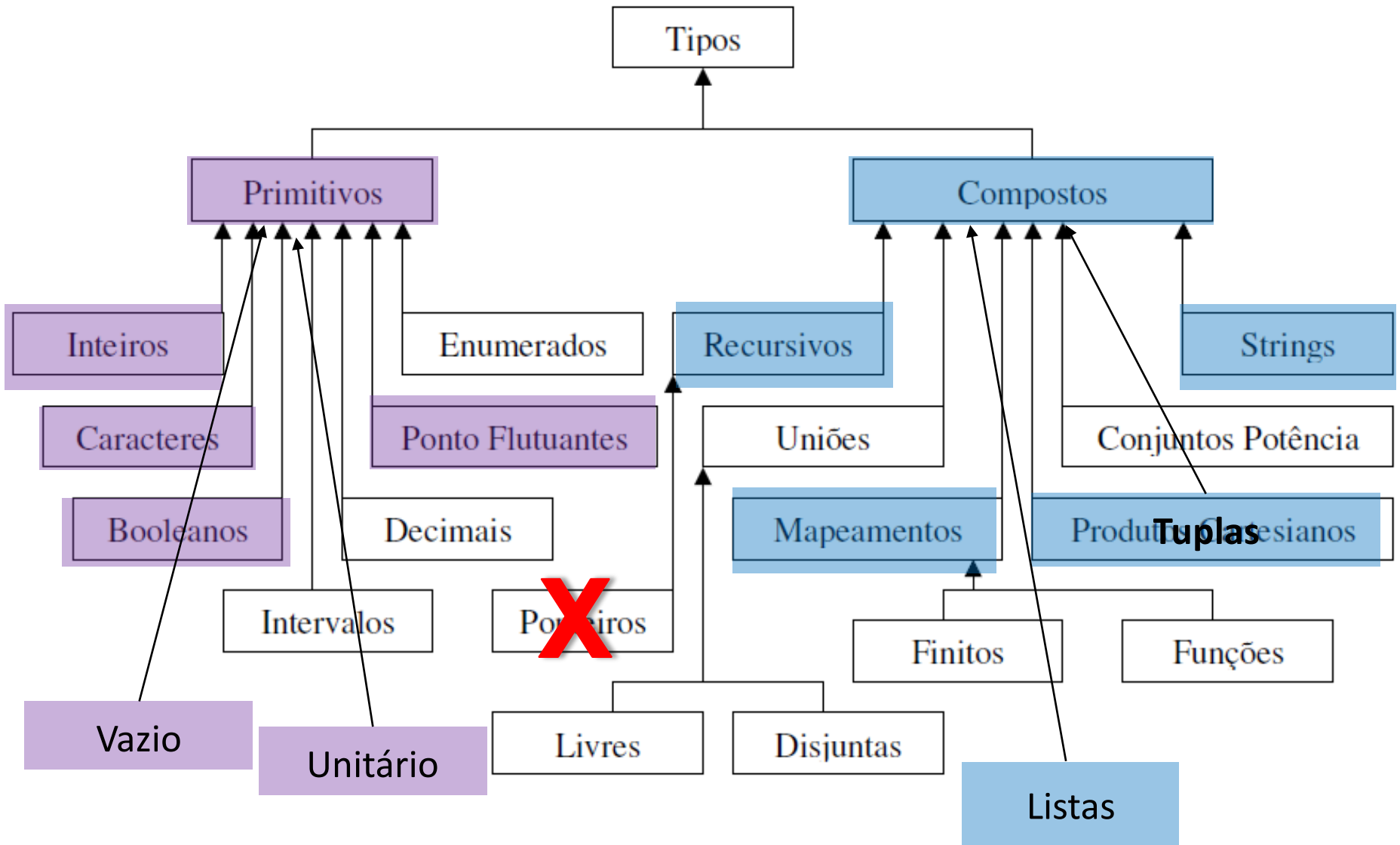
A diretiva deriving (Eq, Show) faz com que o compilador derive automaticamente definições dos operadores == e /= para valores desse tipo

Verificação de tipos

- Para definir que uma expressão **E** tem o tipo **T**:
 - E é do tipo T escreve-se:
 - ✓ $E :: T \rightarrow \text{Saída}$
- Para verificar o tipo de um identificador/expressão, basta usar o **:t**

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```

Tipos de dados: Resumo



Conversões entre Tipos

- Algumas funções embutidas para entrada e saída de caracteres convertem automaticamente os tipos caracteres para valores numéricos e vice-versa.
 - **read**: lê um conteúdo e o converte a numérico
 - **show**: sempre apresenta um tipo String na saída, embora internamente possa manipular outros tipos de dados.

```
cmd Prompt de Comando - ghci
Prelude> show (log 4567)
"8.426611813185"
Prelude> read "77" + 3
80
Prelude> read "77 + 56" + 33
*** Exception: Prelude.read: no parse
Prelude> show (read "77" + log 100)
"81.60517018598809"
Prelude> _
```

Avaliação Preguiçosa em Listas - *Lazy evaluation*



```
C:\Users\USE\Desktop\listaInfinita.hs - Notepad++
Arquivo  Editar  Localizar  Visualizar  Formatar  Linguagem
listaInfinita.hs x
1 lista_infinita :: Int -> [Int]
2 lista_infinita n = n : lista_infinita (n+1)
```

A função retorna uma lista de inteiros a partir do número passado como parâmetro.

```
ca: Prompt de Comando - ghci
72, 20373, 20374, 20375, 20376, 20377, 20378, 20379, 20380, 20381, 20382, 20383, 20384, 20385,
20386, 20387, 20388, 20389, 20390, 20391, 20392, 20393, 20394, 20395, 20396, 20397, 20398, 2
0399, 20400, 20401, 20402, 20403, 20404, 20405, 20406, 20407, 20408, 20409, 20410, 20411, 204
12, 20413, 20414, 20415, 20416, 20417, 20418, 20419, 20420, 20421, 20422, 20423, 20424, 20425
, 20426, 20427, 20428, 20429, 20430, 20431, 20432, 20433, 20434, 20435, 20436, 20437, 20438, 2
0439, 20440, 20441, 20442, 20443, 20444, 20445, 20446, 20447, 20448, 20449, 20450, 20451, 204
52, 20453, 20454, 20455, 20456, 20457, 20458, 20459, 20460, 20461, 20462, 20463, 20464, 20465
, 20466, 20467, 20468, 20469, 20470, 20471, 20472, 20473, 20474, 20475, 20476, 20477, 20478, 2
0479, 20480, 20481, 20482, 20483, 20484, 20485, 20486, 20487, 20488, 20489, 20490, 20491, 204
92, 20493, 20494, 20495, 20496, 20497, 20498, 20499, 20500, 20501, 20502, 20503, 20504, 20505
, 20506, 20507, 20508, 20509, 20510, 20511, 20512, 20513, 20514, 20515, 20516, 20517, 20518, 2
0519, 20520, 20521, 20522, 20523, 20524, 20525, 20526, 20527, 20528, 20529, 20530, 20531, 205
32, 20533, 20534, 20535, 20536, 20537, 20538, 20539, 20540, 20541, 20542, 20543, 20544, 20545
, 20546, 20547, 20548, 20549, 20550, 20551, 20552, 20553, 20554, 20555, 20556, 20557, 20558, 2
0559, 20560, 20561, 20562, 20563, 20564, 20565, 20566, 20567, 20568, 20569, 20570, 20571, 205
72, 20573, 20574, 20575, 20576, 20577, 20578, 20579, 20580, 20581, 20582, 20583, 20584, 20585
, 20586, 20587, 20588, 20589, 20590, 20591, 20592, 20593, 20594, 20595, 20596, 20597, 20598, 2
0599, 20600, 20601, 20602, 20603, 20604, 20605, 20606, 20607, 20608, 20609, 20610, 20611, 206
12, 20613, 20614, 20615, 20616, 20617, 20618, 20619, 20620, 20621, 20622, 20623, 20624, 20625
, 20626, 20627, 20628, 20629, 20630, 20631, 20632, 20633, 20634, 20635, 20636, 20637, 20638, 2
0639, 20640, 20641, 20642, 20643, 20644, 20645, 20646, 20647, 20648, 20649, 20650, 20651, 206
52, 20653, 20654, 20655, 20656, 20657, 20658, 20659, 20660, 20661, 20662, 20663, 20664, 20665
, 20666, 20667, 20668, 20669, 20670, 20671, 20672, 20673, 20674, 20675, 20676, 20677, 20678, 2
0679, 20680, 20681, 20682, 20683, 20684, 20685, 20686, 20687, 20688, 20689, 20690, 20691, 206
92, 20693, 20694, 20695, 20696, 20697, 20698, 20699, 20700, 20701
```

Avaliação Preguiçosa em Listas - *Lazy evaluation*

```

C:\Users\USE\Desktop\listaInfinita.hs - Notepad++
Arquivo  Editar  Localizar  Visualizar  Formatar  Linguagem
listaInfinita.hs x
1 lista_infinita :: Int -> [Int]
2 lista_infinita n = n : lista_infinita (n+1)
  
```

A função retorna uma lista de inteiros a partir do número passado como parâmetro.

Prelude> lista_infinita 3

```


3 : lista_infinita (3+1)
3 : 4 : lista_infinita (4+1)
3 : 4 : 5 : lista_infinita (5+1)
...
...
...
  
```

A lista não é determinada, até que necessária. Como não há critério de parada, esta, cresce indefinidamente.

IO



- Tipo abstrato para operações de E/S, como **putStr**, **print**, **getChar**, **getLine** e **readIO**.




Falaremos
mais tarde

IOError



- Tipo abstrato para erros nas operações de E/S com IO.

A light blue thought bubble with a dark blue outline, containing the text 'Falaremos mais tarde'. The bubble has several smaller circles leading to it from the bottom left.

Falaremos
mais tarde

4 - VARIÁVEIS E CONSTANTES

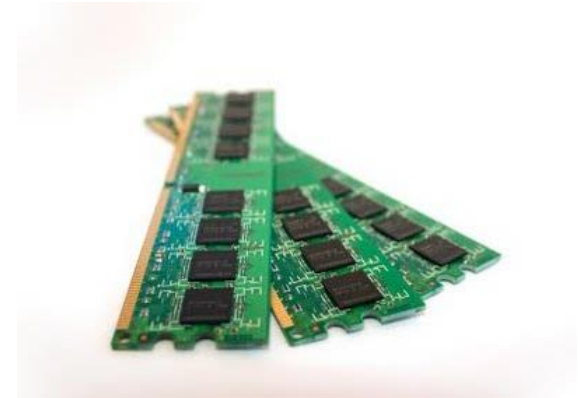
Variáveis e Constantes

- Em Haskell, *as variáveis não variam*. Ou seja, tecnicamente, são constantes. Ou, equivalentemente, não existem variáveis como nas outras linguagens de programação.
- **Tudo é constante**: Variáveis assumem um valor quando são criadas e este valor nunca muda dentro do escopo da mesma.
- Ausência de variáveis globais
- Uma variável é definida na forma:
 - `let x = 3.14159265358979323846264338327950`

5 - GERENCIAMENTO DE MEMÓRIA

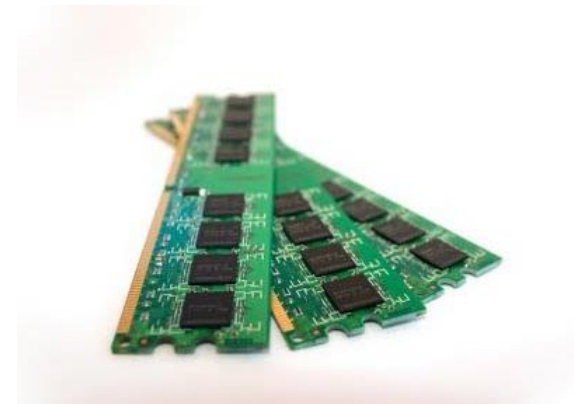
Gerenciamento de memória

- Cálculos em Haskell produzem uma grande quantidade de lixo na memória.
- Como os dados são imutáveis, a única forma de guardar o resultado de cada operação é a criação de novos valores.
 - Ex.: cada iteração recursiva cria um novo valor.
 - Isso obriga a produção de uma grande quantidade de dados temporários, o que também ajudará o recolhimento deste lixo rapidamente.



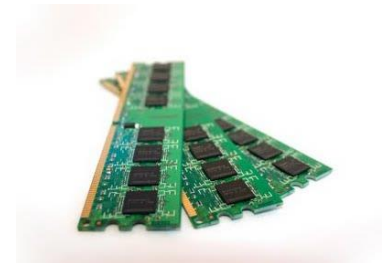
Gerenciamento de memória

- O truque é que dados imutáveis **nunca** apontam para valores novos.
- Os valores novos ainda não existem no momento em que um antigo é criado, por isso, não podem ser apontados do início.
- E uma vez que os valores nunca podem ser modificados, nenhum pode ser apontados mais tarde.



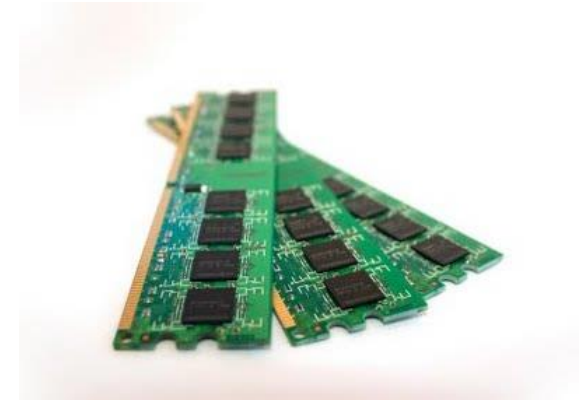
Gerenciamento de memória

- A qualquer momento podemos verificar os últimos valores criados e liberar aqueles que não são apontados a partir do mesmo conjunto.
- Novos dados são alocados em um “berçário” de 512 kb.
- Por padrão, o GHC usa um Garbage Collector por geração.
- Uma vez que este berçário está esgotado, ele é verificado e valores não usados são liberados.
- Os valores sobreviventes são então copiados para a memória principal
- Quanto menos valores alocados, menos trabalho a fazer.



Gerenciamento de memória

- Por exemplo:
 - Se você tem um algoritmo recursivo que rapidamente encheu o berçário com gerações de suas variáveis de indução, apenas a última geração das variáveis vai sobreviver e ser copiada para a memória principal. O resto não será sequer tocado.
- Por isso, Haskell tem um comportamento contra-intuitivo: Quanto maior o percentual de seus valores são lixo, mais rápido ele funciona.



6 - EXPRESSÕES E COMANDOS

Conceito

“frase do programa que necessita ser avaliada e produz como resultado um valor. **Expressões** são caracterizadas pelo uso de operadores, pelos tipos de operandos e pelo tipo de resultado que produzem.” (VAREJÃO, Flávio)

Conceito

Importante

“Expressão é um conceito chave, pois seu propósito é computar novos valores a partir de valores antigos, que é a essência da programação funcional.”

Por ser uma linguagem funcional, Haskell enfatiza a avaliação de expressões ao invés da execução de comandos

- Em Haskell, construímos expressões a partir de:
 1. Constantes
 2. Operadores
 3. Aplicações de funções
 4. Parênteses

Operadores básicos

>	maior
>=	maior ou igual
==	igual
/=	diferente
<	menor
<=	menor ou igual

&&	e
	ou
not	negação

+	Soma
-	Subtração
*	Multiplicação
^	Potência
div	Divisão inteira
mod	resto da divisão
abs	valor absoluto de um inteiro
negate	troca o sinal do valor

/ divisão

++ Concatenação : Composição de lista

Precedência dos Operadores

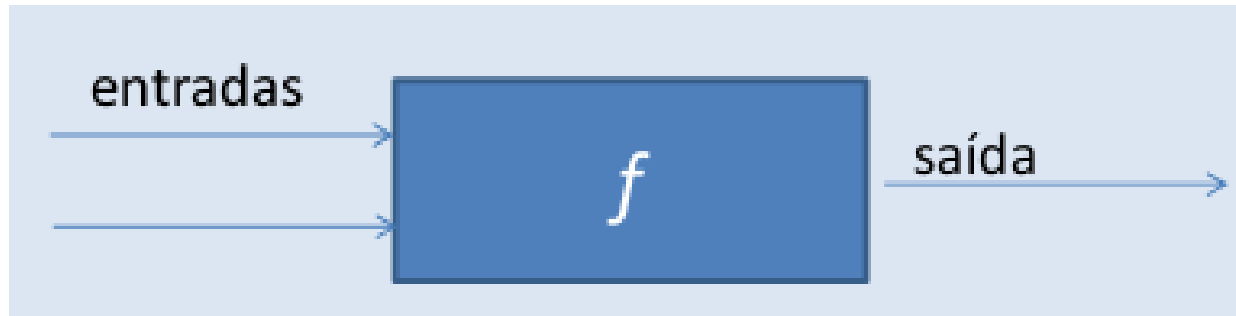
- 1ª) *div, mod, abs, sqrt* e qualquer outra função
 - 2ª) \wedge
 - 3ª) $*$ /
 - 4ª) $+$, $-$
-
- **#Obs.:** Da mesma forma que na matemática usual podemos usar os parênteses para forçar uma ordem diferente de avaliação.

Haskell ↔ Matemática

Haskell		Matemática
$f\ x$	↔	$f\ (x)$
$f\ x\ y$	↔	$f\ (x,y)$
$f\ (g\ x)$	↔	$f\ (g(x))$
$f\ x\ (g\ y)$	↔	$f\ (x,g(y))$
$f\ x\ * \ g\ y$	↔	$f\ (x)\ g(y)$

Funções

- Uma função pode ser representada da seguinte maneira:



- A partir dos valores de entrada é obtido o valor de saída

Funções

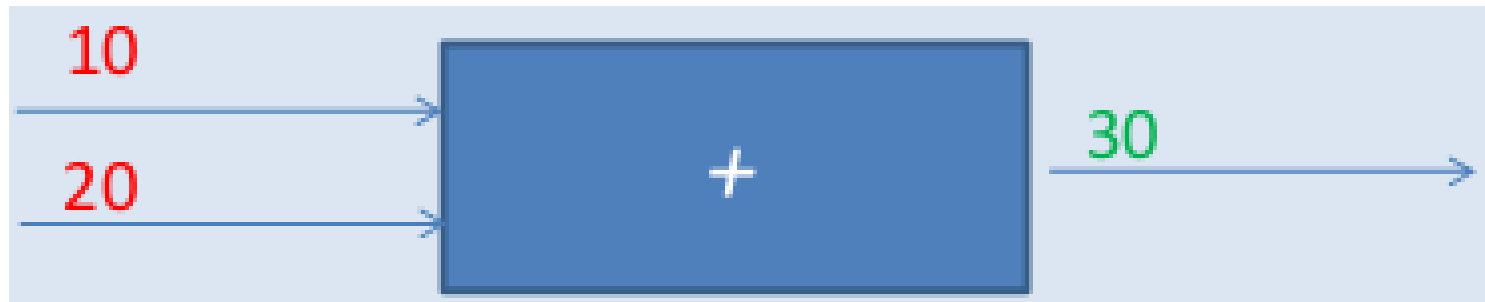
- Funções em Haskell são normalmente definidas pelo uso de equações.
- Por exemplo, a função soma pode ser escrita:

soma x y = x + y

- Chamada da seguinte maneira:

>soma 10 20

30



Funções

- As definições de funções em Haskell podem ser feitas por meio de uma sequência de equações.
- *<nomedafuncao><corpo>*
- **`x :: Int -> Int -> Float -> Bol -> Int`**
 - “x” é o nome da função;
 - O operador `->` indica o **tipo função**;
 - ✓ o último tipo especificado identifica o tipo de dado a ser retornado;
 - ✓ Os quatro tipos do “meio” são argumentos da função.
- Ex.:
 - `Bool -> Bool -> Bool`
 - ✓ Tipo das funções com dois argumentos do tipo `Bool`, e resultado do tipo `Bool`
 - `add :: Int -> Int -> Int`
 - `add x y = x + y`

prototipação de tipos
sequência dos
argumentos da função

Expressões e Comandos

- Trabalha-se somente com funções, seus parâmetros e seus retornos;
- Não possui comando de repetição como **while** e **for**;
- Não possui desvios incondicionais

Expressões e Comandos

- Expressões aritméticas
- Expressões relacionais
- Expressões booleanas

 Prompt de Comando - ghci

```

Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 25+25
50
Prelude> 5 == 5
True
Prelude> True && True
True
Prelude> True && False
False
Prelude> _

```

Expressões e Comandos – if then else

- Expressões condicionais

```
8  -- Aumenta um numero se ele eh pequeno
9  superSizeMe x = if x < 100
10     then x^2
11     else x
12
```


Expressões e Comandos - Guards

- É uma característica na definição de funções, que exploram a capacidade de se inserir condições (*expressão condicional*) que são utilizadas como alternativas para a função.
- Usada para escolher entre vários possíveis resultados.

```

13  -- verifica se eh par
14  par :: Int -> Bool
15  par x
16      | mod x 2 == 0 = True
17      | otherwise = False
18

```

Guard

Se resto de x por
2 for 0 retorne
Verdade

Caso contrário,
retorne falso

Expressões e Comandos - do

- Como Haskell é uma linguagem “preguiçosa”, a ordem em que ocorrem as expressões não é especificada. No entanto, as vezes faz-se necessário ao programador definir a ordem de execução das expressões, o que pode ser feito através do comando **do**.
- O comando **do** tem uma grande aplicação em programas interativos, ou seja, quando é necessário processar alguma informação inserida pelo usuário. Assim, é essencial que a instrução de entrada de dados seja executada antes de todas as outras que irão utilizar essa informação.

Expressões e Comandos - do

```

C:\Users\USE\Desktop\do.hs - Notepad++
Arquivo  Editar  Localizar  Visualizar  Formatar  Linguagem  Configurações  Macro  Executar  Plugins  Janela  ?
do.hs
1  main:: IO ()
2  main = do
3      putStrLn "Por favor, entre com seu nome: "
4      nome <- getLine
5      putStrLn ("Oi, " ++ nome ++ ", prazer em conhecê-lo(a)")

```

```

C:\Users\USE\Desktop>ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l do.hs
[1 of 1] Compiling Main                ( do.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
Por favor, entre com seu nome:
Pamella de Oliveira
Oi, Pamella de Oliveira, prazer em conhecê-lo(a)
*Main> _

```

Expressões e Comandos – case – seleção múltipla



- Expressão **case** é uma forma de expressão que permite selecionar um entre vários resultados possíveis baseando-se no casamento de padrões. Uma expressão case é formada por:
 - uma *expressão de controle*, cujo valor é usado para escolher uma das alternativas
 - uma sequência de alternativas, onde cada alternativa é formada por:
 - ✓ um padrão, usado para decidir se a alternativa será escolhida
 - ✓ uma expressão, usada para dar o resultado caso a alternativa seja escolhida
 - É feito o casamento de padrão do valor da expressão com os padrões, na sequência em que foram escritos, até que se obtenha sucesso ou se esgotem os padrões

Expressões e Comandos – case - seleção múltipla



```
C:\Users\USE\Desktop\case.hs - Notepad++
Arquivo  Editar  Localizar  Visualizar  Formatar  Linguagem  Configurações  Macro  Executar  Plugins  Janela  ?
X
case.hs X
1 case 3-2+1 of
2   0 -> "zero"
3   1 -> "um"
4   2 -> "dois"
5   3 -> "tres"
```

- resulta em **"dois"**, pois o valor da expressão $3-2+1$ é 2, que casa com o terceiro padrão 2, selecionando "dois" como resultado.
- **Obs.:** A regra de layout pode ser aplicada para uma expressão case, permitindo a omissão ou não dos sinais de pontuação { , ; e } .

Expressões e Comandos – Laços de Repetição



- Em Haskell, não existem laços. As repetições são executadas através da chamada recursiva de funções.

Namespaces

- Existem **seis** tipos de nomes em Haskell:
 - Os de **variáveis** e **constructores** denotam valores;
 - **Tipo de Variáveis, tipo de constructores e tipo de classes** referem-se a entidades ligadas ao sistema de tipos;
 - **Nomes de módulos** referem-se a módulos;
- Existem duas restrições à nomeação:
 - Os nomes de variáveis e variáveis de tipo são identificadores que começam com letras minúsculas ou sublinhado;
 - Os outros quatro tipos de nomes são identificadores que começam com letras maiúsculas;
- Um identificador não deve ser usado como o nome de um construtor de tipo e uma classe no mesmo escopo;

Ordenação de Listas - QuickSort



```
C:\Users\USE\Desktop\qSort.hs - Notepad++
Arquivo  Editar  Localizar  Visualizar  Formatar  Linguagem  Configurações  Macro  Executar  Plugins  Janela  ?
qSort.hs x
1  qsort [] = []
2  qsort (x:xs) = qsort elts_esquerdo_x ++ [x] ++ qsort elts_direito_x
3      where
4          elts_esquerdo_x = [y | y <- xs, y < x]
5          elts_direito_x  = [y | y <- xs, y >= x]
```

C:\> Prompt de Comando - ghci

```
C:\Users\USE>cd Desktop
C:\Users\USE\Desktop>ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l qSort
[1 of 1] Compiling Main                < qSort.hs, interpreted >
Ok, modules loaded: Main.
*Main> qsort [15,14,13,12,11,10,9,8,1,2,3,5,7,6]
[1,2,3,5,6,7,8,9,10,11,12,13,14,15]
*Main> _
```


Recursão

- Recursividade é um campo da ciência da computação que estuda funções recursivas. Funções recursivas são todas aquelas que apresentam recursividade.



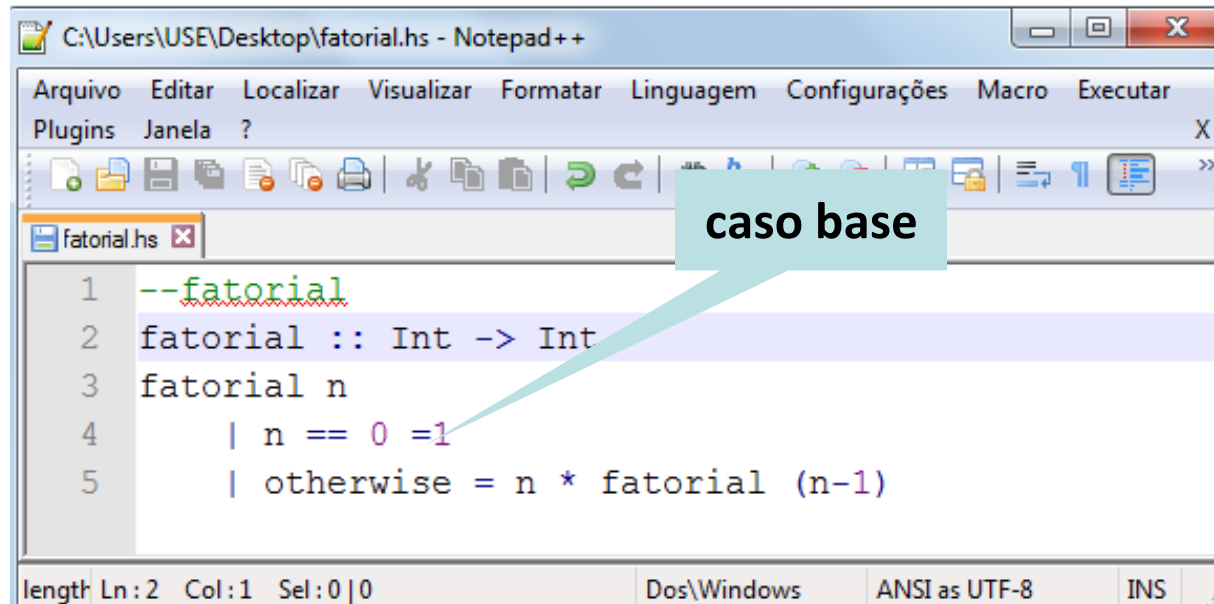
Falando sério...

- Uma função é dita recursiva se ela for definida em termos de si mesma.
- Ideia do algoritmo recursivo:
 - Se a instância do problema é pequena, resolva-a diretamente.
 - Se a instância é grande, reduza-a a uma instância menor do mesmo problema.
- Funções recursivas são construídas a partir de **relações de recorrência**.

Recursão

- Fatorial de um número:

$$- n! = \begin{cases} 1, & \text{se } n = 0 \\ n \cdot (n - 1)!, & \text{se } n > 0 \end{cases}$$



```

1  --fatorial
2  fatorial :: Int -> Int
3  fatorial n
4      | n == 0 = 1
5      | otherwise = n * fatorial (n-1)

```

caso base

length Ln: 2 Col: 1 Sel: 0 | 0 Dos\Windows ANSI as UTF-8 INS

Recursão - *Lazy evaluation*

- A linguagem Haskell utiliza uma estratégia na avaliação das funções denominada avaliação preguiçosa, cujo fundamento é que não avalia nenhuma subexpressão ou função até que seu valor seja reconhecido como necessário. Este conceito é ilustrado pelo exemplo a seguir:

```

C:\Users\USE\Desktop\lazy.hs - Notepad++
Arquivo  Editar  Localizar  Visualizar  Formatar  Linguagem  Configurações
lazy.hs x
1 dobro, triplo :: Int -> Int
2 menor, f :: Int -> Int -> Int
3
4 menor x y
5     | x < y = x
6     | otherwise = y
7
8 dobro x = x + x
9 triplo x = 3 * x
10 f a b = (dobro(triplo(menor a b)))
  
```

$$\begin{aligned}
 f\ 9\ 8 &= (\text{dobro}(\text{triplo}(\text{menor}\ 9\ 8))) \\
 &= (\text{triplo}(\text{menor}\ 9\ 8)) + (\text{triplo}(\text{menor}\ 9\ 8)) \\
 &= (3 * (\text{menor}\ 9\ 8)) + (\text{triplo}(\text{menor}\ 9\ 8)) \\
 &= (3 * (\text{menor}\ 9\ 8)) + (3 * (\text{menor}\ 9\ 8)) \\
 &= (3 * 8) + (3 * (\text{menor}\ 9\ 8)) \\
 &= (3 * 8) + (3 * 8) \\
 &= 24 + (3 * 8) \\
 &= 24 + 24 \\
 &= 48
 \end{aligned}$$

Casamento de Padrões

- Linguagens funcionais modernas usam casamento de padrão em várias situações, como por exemplo para selecionar componentes de estruturas de dados, para selecionar alternativas em expressões **case**, e em aplicações de funções.
- **Padrão** é uma construção da linguagem de programação que permite analisar um valor e associar variáveis aos componentes do valor.
- **Casamento de padrão** é uma operação envolvendo um padrão e uma expressão que faz a correspondência (casamento) entre o padrão e o valor da expressão.

Casamento de Padrões

- Padrão constante
 - O padrão constante é simplesmente uma constante. O casamento sucede se e somente se o padrão for idêntico ao valor. Nenhuma associação de variável é produzida.
 - ✓ Ex.: padrão (10) valor (10) há o casamento
- Padrão variável
 - O padrão variável é simplesmente um identificador de variável de valor (e como tal deve começar com letra minúscula). O casamento sucede sempre.
 - ✓ Ex.: padrão (x) valor (10) há o casamento $x \mapsto 10$
- Padrão curinga
- Padrão tupla
- Padrões lista

7 - MODULARIZAÇÃO

Mecanismo de implementação da passagem de parâmetro

- Haskell é uma linguagem puramente funcional, o que significa que não há efeitos colaterais para funções;
- Parâmetros serão sempre passados por valor.
- Haskell é uma linguagem puramente funcional, o que significa que não há efeitos colaterais para as funções.
- Se uma função é chamada duas vezes com os mesmos parâmetros, o resultado retornado por ela será o mesmo.
- Os parâmetros são sempre passados **por cópia**. Qualquer modificação ou definição feita nesse parâmetro ocorrerá apenas dentro daquele escopo.

Mecanismo de implementação da passagem de parâmetro

Por referência



Por cópia
"criação de uma
cópia do parâmetro
real no ambiente
local."

Tipos abstratos de dados - TADs

- Uma declaração de tipo é feita usando a palavra reservada **type**.
- A partir dos tipos pré-definidos, novos tipos podem ser construídos pelo programador, denominados classes, que funcionam como tipo abstrato de dados.

Tipos abstratos de dados - TADs

- Haskell permite que o usuário crie alguns outros tipos denominados classes, que funcionam como tipos abstratos de dados.
- As definições de uma classe consistem em:
 - `class <nome da classe> <tipo de entrada> where`
 `<assinatura da classe>`

TAD Fila



```
tad.hs x
1  module Fila where
2
3  data Fila t = F [t]
4             deriving (Show)
5
6  novaFila :: Fila t
7  novaFila = F []
8
9  inserirFila :: Fila t -> t -> Fila t
10 inserirFila (F lista) n = F (lista ++ [n])
11
12 removerFila :: Fila t -> Fila t
13 removerFila (F []) = error "fila vazia"
14 removerFila (F (x:xs)) = F xs
15
16 frente :: Fila t -> t
17 frente (F []) = error "fila vazia"
18 frente (F (x:xs)) = x
19
20 filaVazia :: Fila t -> Bool
21 filaVazia (F []) = True
22 filaVazia _ = False
```

8 - MÓDULOS

Módulos

- Um módulo Haskell é uma coleção de funções, tipos e typeclasses. Um programa Haskell é uma coleção de módulos, onde o módulo principal carrega outros e usa suas funções para fazer algo de útil.
- A sintaxe de importação módulos em um script Haskell é **import <nome do módulo>**. Isso deve ser feito antes de definir qualquer função.
- **Exemplos de módulos:**
 - **Data.List** (possui várias funções úteis para se trabalhar com listas)
 - **Data.Map** (provê formas de pesquisar valores por chaves em estruturas de dados)
 - **Control.Exception** (oferece suporte para levantar exceções definidas pelo usuário)
 - **System.IO.Error** (trata erros de entrada e saída)

Módulos



```
1
2
3
4 import System.Environment
5 import System.Directory
6 import System.IO
7 import Data.List
8 import Data.List.Split -- para poder usar a função splitOn. Para
9 import System.IO.Error
10 import System.Exit
11
```

Módulos ... Criando um

- No topo ficam as declarações das funções exportadas pelo módulo, e abaixo suas implementações.
- O módulo deve ter o mesmo nome do script.
- Caso você queira utilizar um módulo feito por outra pessoa e que não esteja instalado ainda na máquina, deve-se fazer o seguinte:
 - Primeiro é necessário instalar um programa chamado “cabal”
 - ✓ `sudo apt-get install cabal-install`
 - ✓ `cabal install nomeDoMódulo`

Criando um



```
modulo.hs x new 5 x
1 module Funcoes where
2
3 ehPar :: Int -> Bool
4 ehPar n | (mod n 2 == 0) = True
5         | otherwise = False
6
7 ehImpar :: Int -> Bool
8 ehImpar n | (mod n 2 == 0) = False
9           | otherwise = True
```

```
CA. Prompt de Comando - ghci
Loading package base ... linking ... done.
Prelude> :l modulo.hs
[1 of 1] Compiling Funcoes          < modulo.hs, interpreted >
Ok, modules loaded: Funcoes.
*Funcoes> ehPar 5
False
*Funcoes> ehImpar 5
True
*Funcoes> _
```

Criando um



```
modulo.hs | teste2.hs |
1 module Teste where
2
3 import Funcoes
4
5 testePar n = ehPar n
6
7 testeImpar n = ehImpar n
```

```
*Funcoes> :l teste2.hs modulo.hs
[1 of 2] Compiling Funcoes      ( modulo.hs, interpreted )
[2 of 2] Compiling Teste        ( teste2.hs, interpreted )
Ok, modules loaded: Funcoes, Teste.
*Teste> testePar 2
True
*Teste> testeImpar 2
False
*Teste>
```

9 - 10

IO

- Uma ação de entrada e saída (E/S) é um valor que representa uma interação com o mundo. Uma ação de E/S pode ser executada para interagir com o mundo e retornar um valor obtido através desta interação.
- Em Haskell IO a é o tipo das ações de entrada e saída que interagem com o mundo e retornam um valor do tipo a . IO a é um tipo abstrato, logo sua representação não está disponível nos programas.
- Haskell provê algumas ações de entrada e saída primitivas, e um mecanismo para combinar ações de entrada e saída.

IO

- Ações de saída padrão
 - putChar
 - putStr
 - putStrLn
 - print
- Ações de entrada padrão
 - getChar
 - getLine
 - getContents
 - readLn

```

C:\ Prompt de Comando - ghci
Loading package ghc-prim ... linking ... done
Loading package integer-gmp ... linking ... done
Loading package base ... linking ... done.
Prelude> :t putChar
putChar :: Char -> IO ()
Prelude> :t putStr
putStr :: String -> IO ()
Prelude> :t putStrLn
putStrLn :: String -> IO ()
Prelude> :t print
print :: Show a => a -> IO ()
Prelude> :t getChar
getChar :: IO Char
Prelude> :t getLine
getLine :: IO String
Prelude> :t getContents
getContents :: IO String
Prelude> :t readLn
readLn :: Read a => IO a
Prelude>
  
```

IO



```
C:\Users\USE\Desktop\Haskell\io.hs - Notepad++
Arquivo  Editar  Localizar  Visualizar  Formatar  Linguagem  Configurações  Macro  Executar  Plugins  Janela  ?  X
io.hs
1  main :: IO () --funcao que retorna IO
2                    --para fazer ação de e/s
3                    --produzirá um valor ou um resultado
4  main = do
5      putStr "Digite o primeiro número: "
6      n1 <- getLine
7      putStr "Digite o segundo número: "
8      n2 <- getLine
9      putStrLn ("Soma: " ++ show (read n1 + read n2) )

C:\> Prompt de Comando - ghci
Prelude> :l io.hs
[1 of 1] Compiling Main                < io.hs, interpreted >
Ok, modules loaded: Main.
*Main> main
Digite o primeiro número: 12
Digite o segundo número: 8
Soma: 20
*Main> _
```

10 - POLIMORFISMO

Sistemas de tipos

- Verificação de tipos



- **Fortemente tipadas:** possui uma disciplina rigorosa de tipos de dados.
- Tanto os compiladores quanto os interpretadores implementam o mecanismo de **checagem forte de tipo de dados**, devido a tamanha variedade de tipos.

Sistemas de tipos

- Fortemente tipada indica que toda função, “variável”, constante, etc., tem um só tipo de dado, e é sempre possível determinar o seu tipo.
- Embora fortemente tipada, a linguagem Haskell possui um sistema de dedução automática de tipos para funções cujos tipos não foram definidos.

Sistemas de tipos – Checagem de tipos

- Toda função, variável, constante tem um tipo de dado, que sempre pode ser determinado;
- $E :: T$ (como vimos anteriormente)
 - Usamos a notação para indicar que a avaliação produz um valor do tipo T.
- Usando **:type** <nome> ou **:t** <nome> você verifica o tipo.

```
Prompt de Comando - ghci
Microsoft Windows [versão 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.

C:\Users\USE>ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :type False
False :: Bool
Prelude> :t 'z'
'z' :: Char
Prelude> :t 2*(5-8) <= 6+1
2*(5-8) <= 6+1 :: Bool
Prelude> let listaLetra = ['v','a','s','c','o']
Prelude> :t listaLetra
listaLetra :: [Char]
Prelude>
```

Sistemas de tipos – Checagem de tipos

- Toda expressão sintaticamente correta tem o seu tipo calculado em *tempo de compilação*. Se não for possível determinar o tipo de uma expressão ocorre um erro de tipo.
- A aplicação de uma função a um ou mais argumentos de tipo inadequado constitui um erro de tipo. Por exemplo:



Sistemas de tipos – Checagem de tipos

```

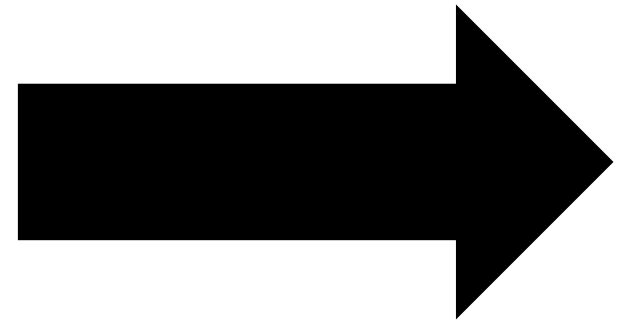
Prompt de Comando - ghci
Prelude>
Prelude> not 'A'

<interactive>:8:5:
  Couldn't match expected type `Bool' with actual type `Char'
  In the first argument of `not', namely `A'
  In the expression: not 'A'
  In an equation for `it': it = not 'A'
Prelude> _
  
```

- A função **not** requer um valor booleano, porém foi aplicada ao argumento 'A', que é um character. Haskell é uma linguagem **fortemente tipada**, com um sistema de tipos muito avançado. Todos os possíveis erros de tipo são *encontrados em tempo de compilação (tipagem estática)*. Isto torna os programas mais seguros e mais rápidos, eliminando a necessidade de verificações de tipo em tempo de execução.

Sistemas de tipos

- Toda expressão bem formada tem um tipo mais geral, que pode ser calculado automaticamente em tempo de compilação usando um processo chamado **inferência de tipos**.
- A capacidade de inferir tipos automaticamente *facilita a programação*, deixando o programador livre para omitir anotações de tipo ao mesmo tempo que permite a verificação de tipos.
- A inferência de tipo é feita usando as **regras de tipagem** de cada forma de expressão.



Sistemas de tipos

- Literais inteiros
 - Os literais inteiros são do tipo **Num** $a \Rightarrow a$
- Literais fracionários
 - Os literais fracionários são do tipo **Fractional** $a \Rightarrow a$
- Literais caracteres
 - Os literais caracteres são do tipo **Char**
- Literais strings
 - Os literais strings são do tipo **String**.
- Construtores constantes
 - Os construtores constantes de um tipo são do tipo associado. Assim: os construtores constantes booleanos **True** e **False** são do tipo **Bool**.

Sistemas de tipos

- *Em uma aplicação de função:*
 - o tipo dos argumentos deve ser compatível com os domínios da função.
 - o tipo do resultado deve ser compatível com o contra-domínio da função

Polimorfismo genérico

- **length**: calcula o tamanho (quantidade de elementos) de uma lista - **length :: [a] -> Int**
 - ✓ length [1,2,3,4]
 - ✓ 4
 - ✓ length ['a','b','c','d']
 - ✓ 4
- Você não precisará criar uma função pra cada tipo que você for utilizar.
- Independente do tipo que passarmos sempre retornará o tamanho da lista.

Tipos de polimorfismo

Polimorfismo	Ad-hoc	Coerção
	Universal	Sobrecarga
		Paramétrico
		Inclusão

Polimorfismo Ad-hoc

- Ocorre quando um mesmo símbolo ou identificador é associado a diferentes trechos de código que atuam sobre diferentes tipos.

- Ex.: função que aceita inteiros e admite somente inteiros

```

1 my_length :: [Int] -> Int
2 my_length [] = 0
3 my_length (x:xs) = 1+ my_length xs

```

- No exemplo anterior , trabalharia apenas com uma lista de inteiros, porém utilizando o polimorfismo paramétrico em Haskell poderíamos fazer esta função receber qualquer tipo de lista

```

1 my_length :: [a] -> Int
2 my_length [] = 0
3 my_length (x:xs) = 1+ my_length xs

```

Ad-hoc: Coerção

- **Coerção Implícita:**
- `ghci> 2 + 13.2`
- `15.2`

- **Coerção Explícita:**
- `ghci> (2 :: Float) + 10.0`
- `12.0`

Ad-hoc: Sobrecarga

- Sobrecarga sobre operadores: Ocorre em Haskell e o mais comum é o “+”.

```
ghci>1.4 +1
```

```
2.4
```

```
ghci>1 + 1
```

```
2
```

```
ghci>1.666 + 1.2
```

```
2.866
```

```
ghci>"italo" ++ " a"
```

```
"italoa"
```

Ad-hoc: Sobrecarga

- O módulo Prelude apresenta algumas sobrecargas:
 - o identificador **pi** é sobrecarregado e denota variáveis dos tipos numéricos com representação em ponto flutuante cujo valor é uma aproximação de π
 - o identificador **abs** é sobrecarregada e denota funções que calculam o valor absoluto, cujo argumento pode ser de qualquer tipo numérico, e cujo resultado é do mesmo tipo que o argumento
 - o operador (/) é sobrecarregado e denota funções de divisão fracionária com dois argumentos de qualquer tipo numérico fracionário, e resultado do mesmo tipo dos argumentos

Classes de tipo

- Para expressar a sobrecarga, Haskell usa classes de tipo. Uma classe de tipo é uma coleção de tipos (chamados de instâncias da classe) para os quais é definido um conjunto de funções (aqui chamadas de métodos) que podem ter diferentes implementações, de acordo com o tipo considerado.
- Uma classe especifica uma interface indicando o nome e a assinatura de tipo de cada função. Cada tipo que é instância (faz parte) da classe define (implementa) as funções especificadas pela classe.

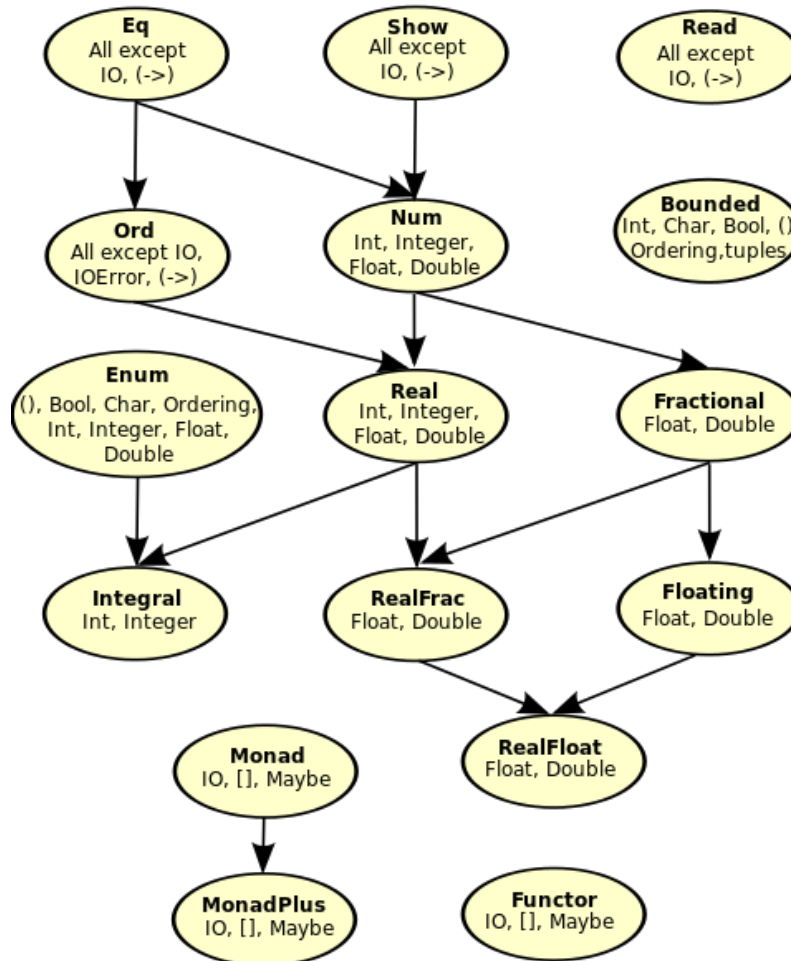


Classes de tipo

- Por exemplo:
- A classe **Num** é formada por todos os tipos numéricos e sobrecarrega algumas operações aritméticas básicas, como adição. Os tipos **Int** e **Double** são instâncias da classe **Num**. Logo existe uma definição da adição para o tipo **Int** e outra para o tipo **Double**, usando algoritmos diferentes.
- A classe **Eq** é formada por todos os tipos cujos valores podem ser verificados se são iguais ou diferentes, e sobrecarrega os operadores (**==**) e (**/=**). Logo para cada instância desta classe existe uma definição destes operadores. Todos os tipos básicos apresentados anteriormente são instâncias de **Eq**. Nenhum tipo função é instância de **Eq**, pois de forma geral não é possível comparar duas funções.

Classes de tipo pré-definidas

- Haskell tem várias classes predefinidas e o programador pode definir suas próprias classes.



Herança

- O sistema de classes de Haskell também suporta a noção de herança, onde uma classe pode herdar todos os métodos de uma outra classe, e ao mesmo tempo ter seus próprios métodos.

- Exemplo: a classe **Ord**:

```
class (Eq a) => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max           :: a -> a -> a
```

- **Eq** é uma superclasse de **Ord**.
 - **Ord** é uma subclasse de **Eq**.
 - **Ord** herda todos os métodos de **Eq**.
 - Todo tipo que é instância de **Ord** tem que ser necessariamente instância de **Eq**.
- Haskell suporta herança múltipla: uma classe pode ter mais do que uma superclasse.

Ad-hoc: Sobrecarga

- Sobrecarga sobre métodos: NÃO existe em Haskell.

Universal: Paramétrico

- Operação sobre vários tipos de dados
- Algumas funções podem operar sobre vários tipos de dados.
Ex.: a função **head** recebe uma lista e retorna o primeiro elemento da lista:
 - head ['b','a','n','a','n','a']
 ✓ 'b'
 - head ["maria","paula","peixoto"]
 ✓ "maria"
 - head [True,False,True,True]
 ✓ True
 - head [("ana",2.8),("pedro",4.3)]
 ✓ ("ana",2.8)
- Não importa qual é o tipo dos elementos da lista.

11 - EXCEÇÕES

Exceções

- Em Haskell as exceções podem ser geradas a partir de qualquer local do programa. No entanto, devido à ordem de avaliação especificada, elas só podem ser capturadas na IO.
- Em Haskell a manipulação de exceção não envolve sintaxe especial como faz em Python ou Java. Pelo contrário, os mecanismos para capturar e tratar exceções são funções.

Exceções



```
1 import Control.Exception
2 import System.IO.Error
3
4 ler_arq :: IO ()
5 ler_arq = do
6     {catch (ler_arquivo) tratar_erro;}
7     where
8         ler_arquivo = do
9             {
10                 conteudo <- readFile "teste.txt";
11                 return (read conteudo);
12             }
13         tratar_erro erro = if isDoesNotExistError erro
14             then do
15                 {
16                     putStrLn "Exceção: arquivo inexistente";
17                     putStrLn ""
18                     --trate a exceção
19                 }
20             else ioError erro
21
```

Funções para tratar exceções

- Handle
 - objetivo capturar exceções geradas no programa.
- Throw
 - objetivo levantar exceções
 - retorna um valor de qualquer tipo
- Error
 - recebe uma String como parâmetro e levanta exceção

12 - CONCORRÊNCIA

Concorrência

- Bastante usado em Haskell por meio de threads.
- Thread em Haskell é mais eficiente em tempo e em espaço em relação ao SO
- Podemos escrever um programa para agir em paralelo bastando adicionar a palavra ``par`` entre as expressões, como no exemplo.

Threads



```
15 main = do
16     -- forkIO recebe um IO () e retorna um IO ThreadId
17     -- inicia uma nova thread
18     forkIO $ do {
19         putStrLn ("Obtendo todos os números perfeitos até 10000...");
20         putStrLn ("Números perfeitos: " ++ (show (obter_perfeitos 10000)));
21         putStrLn ("Acabou de obter os números perfeitos!");
22     }
23     forkIO $ do {
24         -- suspende a execução da thread por 3 segundos (3000000 microssegundos)
25         threadDelay 3000000;
26         putStrLn ("\nExecutando outra thread...");
27     }
28     threadDelay 4000000 -- suspende a execução por 40 segundos
29     putStrLn ("Fim!")
```

Concorrência

- Faremos um exemplo utilizando threads sem usar a pilha mVar e com a pilha

```

1  ex1 = do
2
3      let msg1 = "Não gosto"
4          let msg2 = "de Haskell"
5
6          forkIO $ print msg1
7          forkIO $ print msg2
8          print "Fim :)"
9

```

- Este exemplo executará tudo junto.
- Para resolver este problema , utilizaremos mVar



Concorrência

```

1 ex2 = do
2     m <- newEmptyMVAR
3
4     forkIO $ putMVAR m "Não gosto"
5     forkIO $ putMVAR m "de Haskell"
6
7     r <- takeMVAR m
8     print r
9     r <- takeMVAR m
10    print r
11    print "Fim :)"

```

- Agora, o programa irá retirar de m e executará cada um de uma vez.

13 - ORIENTAÇÃO A OBJETOS

00

- Em Haskell não há uma estrutura orientada a objetos propriamente dita, por se tratar de uma linguagem puramente funcional;
- Nem mesmo o tipo class (o nome pode enganar) é parecido com uma classe formal de linguagens orientadas a objeto;
- Porém, há algumas linguagens, variações de Haskell, que são orientadas a objetos:
 - Haskell++
 - O'Haskell
 - Mondrian

14 - AVALIAÇÃO DA LINGUAGEM

Avaliação da Linguagem

- **Critérios gerais:**
 - Aplicabilidade
 - Confiabilidade
 - Facilidade de Aprendizado
 - Eficiência
 - Portabilidade
 - Suporte ao método de projeto
 - Evolutibilidade
 - Reusabilidade
 - Integração com outros softwares
 - Custo

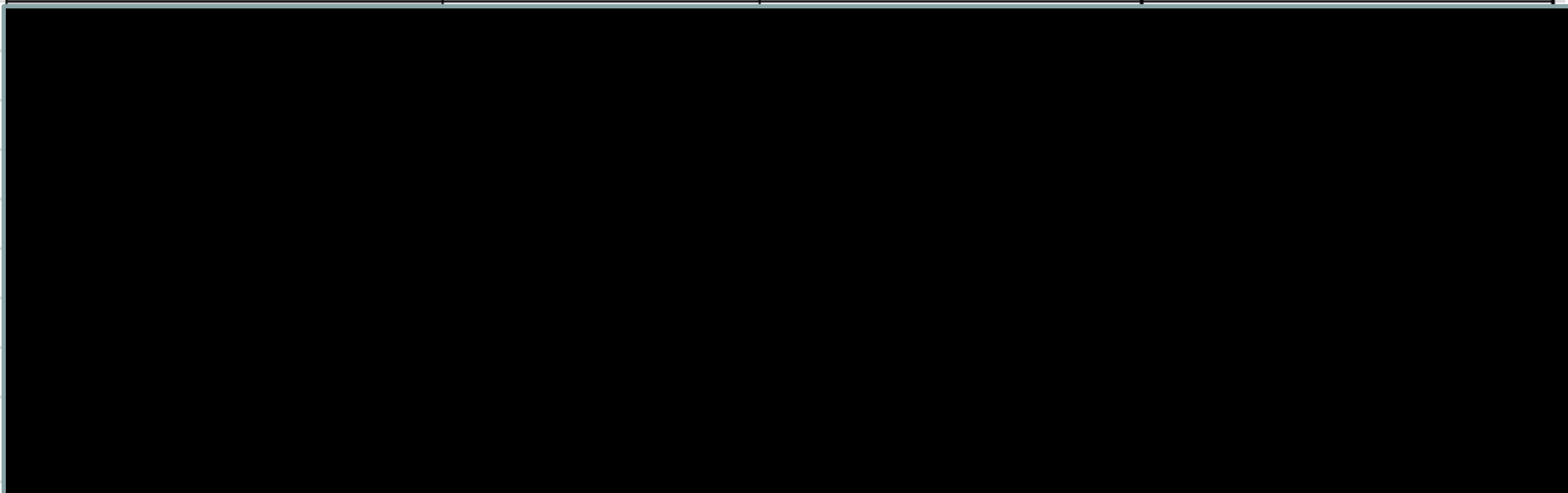
Avaliação da Linguagem

Critérios Gerais:	C	JAVA	Haskell
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Não	Sim	Sim
Facilidade de Aprendizado	Não	Não	Sim
Eficiência	Sim	Parcial	Parcial
Portabilidade	Não	Sim	Parcial
Suporte ao método de projeto	Estruturado	Orientado a Objetos	Funcional
Evolutibilidade	Não	Sim	Sim
Reusabilidade	Parcial	Sim	Sim
Integração	Sim	Parcial	Não
Custo	Depende da ferramenta de desenvolvimento	Depende da ferramenta de desenvolvimento	Depende da ferramenta de desenvolvimento

Avaliação da Linguagem



Crítérios Gerais:	C	JAVA	Haskell
Aplicabilidade	Sim	Parcial	Sim



Avaliação da Linguagem



Critérios Gerais:	C	JAVA	Haskell
Confiabilidade	Não	Sim	Sim

Avaliação da Linguagem



Critérios Gerais:	C	JAVA	Haskell
Facilidade de Aprendizado	Não	Não	Sim

Avaliação da Linguagem



Critérios Gerais:	C	JAVA	Haskell
Eficiência	Sim	Parcial	Parcial

Avaliação da Linguagem



Critérios Gerais:	C	JAVA	Haskell
Portabilidade	Não	Sim	Parcial

Avaliação da Linguagem



Critérios Gerais:	C	JAVA	Haskell
[Redacted]			
Suporte ao método de projeto	Estruturado	Orientado a Objetos	Funcional
[Redacted]			

Avaliação da Linguagem



Critérios Gerais:	C	JAVA	Haskell
Evolutibilidade	Não	Sim	Sim

Avaliação da Linguagem



Critérios Gerais:	C	JAVA	Haskell
Reusabilidade	Parcial	Sim	Sim

Avaliação da Linguagem



Critérios Gerais:	C	JAVA	Haskell
Integração	Sim	Parcial	Não

Avaliação da Linguagem



Critérios Gerais:	C	JAVA	Haskell
Custo	Depende da ferramenta de desenvolvimento	Depende da ferramenta de desenvolvimento	Depende da ferramenta de desenvolvimento

Avaliação da Linguagem

- **Critérios específicos:**
 - Escopo
 - Expressões e comandos
 - Tipos primitivos e compostos
 - Gerenciamento de memória
 - Persistência de dados
 - Passagem de parâmetros
 - Encapsulamento e proteção
 - Sistema de tipos
 - Verificação de tipos
 - Polimorfismo
 - Exceções
 - Concorrência

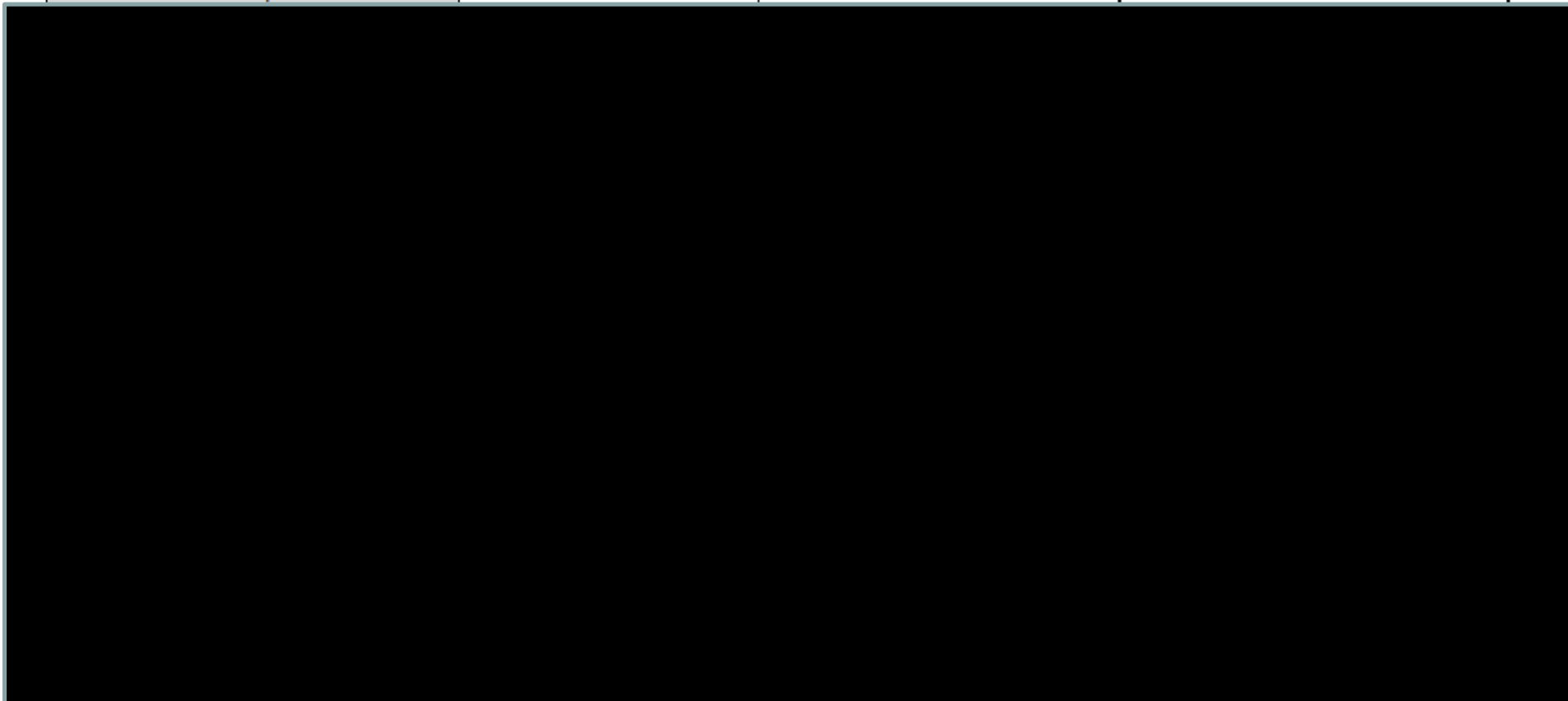
Avaliação da Linguagem

Crítérios Específicos:	C	JAVA	Haskell
Escopo	Sim	Sim	Sim
Expressões e comandos	Sim	Sim	Sim
Tipos primitivos e compostos	Sim	Sim	Sim
Gerenciamento de memória	Programador	Sistema	Sistema
Persistência de dados	Biblioteca de funções	Biblioteca de classes, serialização e JDBC	Classe monádica de I/O
Passagem de parâmetros	Lista variável e passagem por valor	Passagem por valor e por cópia de referência	Passagem por cópia
Encapsulamento e proteção	Parcial	Sim	Parcial
Sistema de tipos	Não	Sim	Sim
Verificação de tipos	Estática	Estática e dinâmica	Fortemente tipada
Polimorfismo	Coerção e sobrecarga	Coerção, sobrecarga e inclusão	Sobrecarga, Coerção, Paramétrico
Exceções	Não	Sim	Parcial
Concorrência	Não	Sim	Sim

Avaliação da Linguagem



CrITÉrios EspecÍficos:	C	JAVA	Haskell
Escopo	Sim	Sim	Sim



Avaliação da Linguagem



Critérios Específicos:	C	JAVA	Haskell
Expressões e comandos	Sim	Sim	Sim

Avaliação da Linguagem



Critérios Específicos:

C

JAVA

Haskell

Tipos primitivos e compostos

Sim

Sim

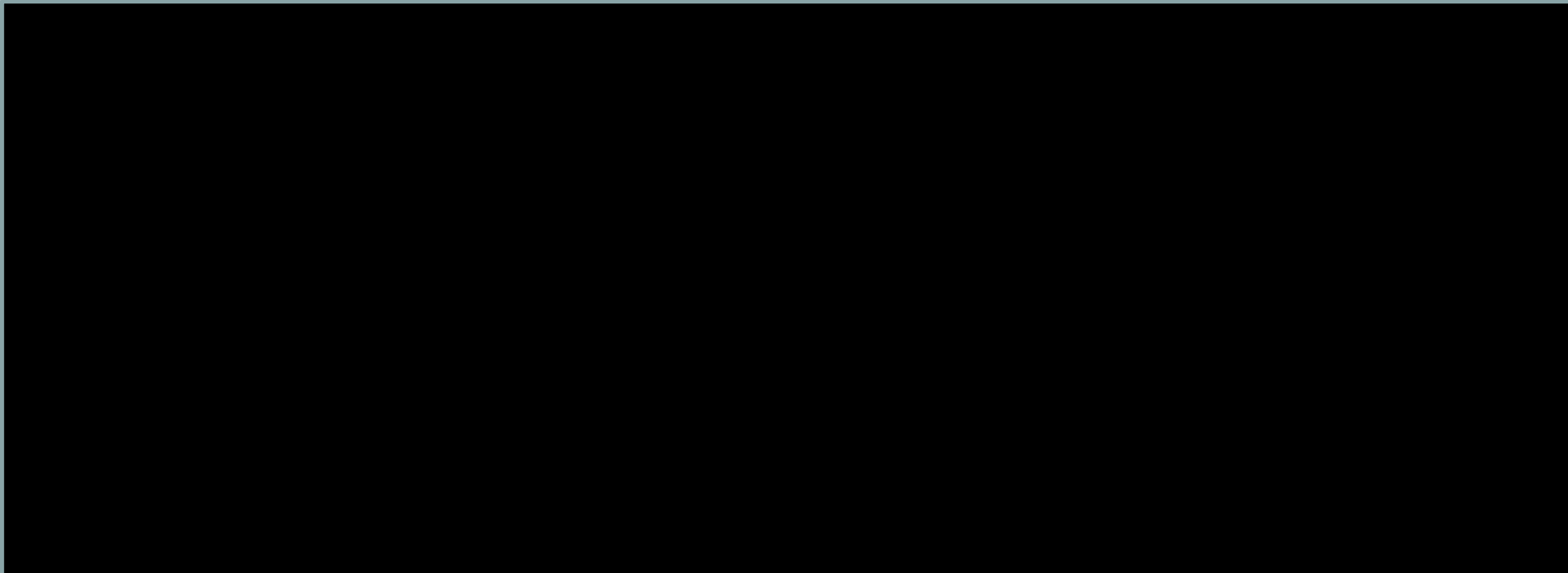
Sim

Avaliação da Linguagem

Critérios Específicos:	C	JAVA	Haskell
------------------------	---	------	---------

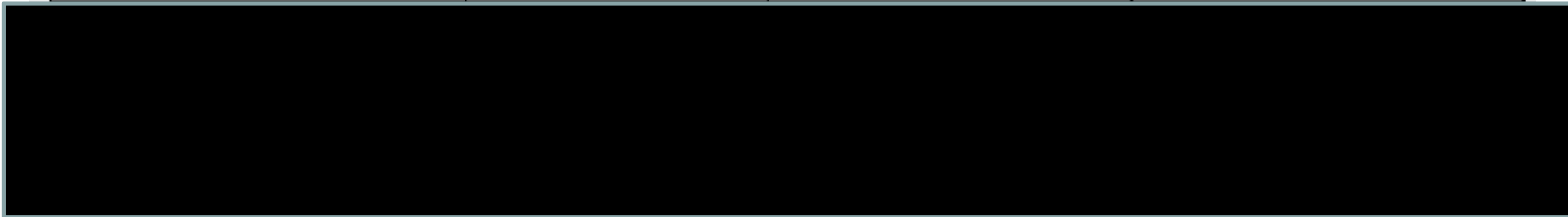


Gerenciamento de memória	Programador	Sistema	Sistema
--------------------------	-------------	---------	---------



Avaliação da Linguagem

Critérios Específicos:	C	JAVA	Haskell
------------------------	---	------	---------

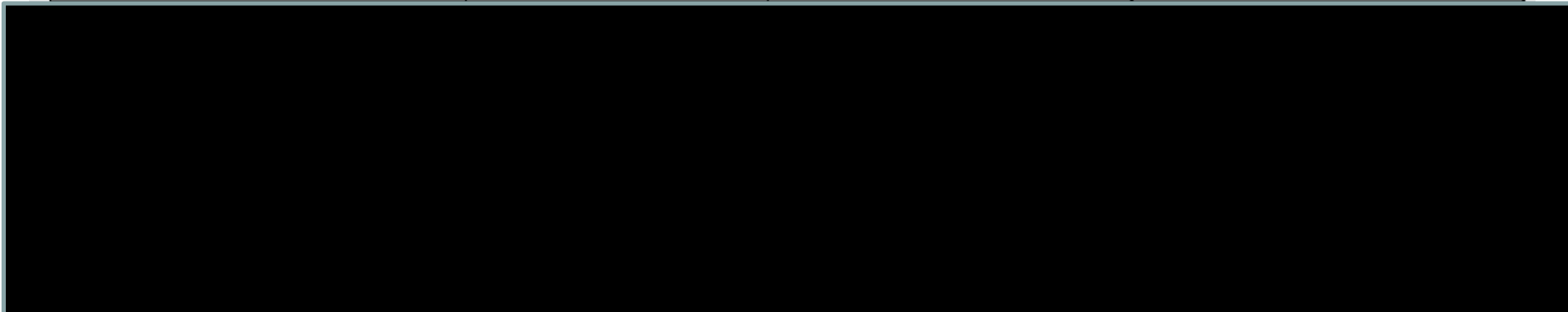


Persistência de dados	Biblioteca de funções	Biblioteca de classes, serialização e JDBC	Classe monádica de I/O
-----------------------	-----------------------	--	------------------------

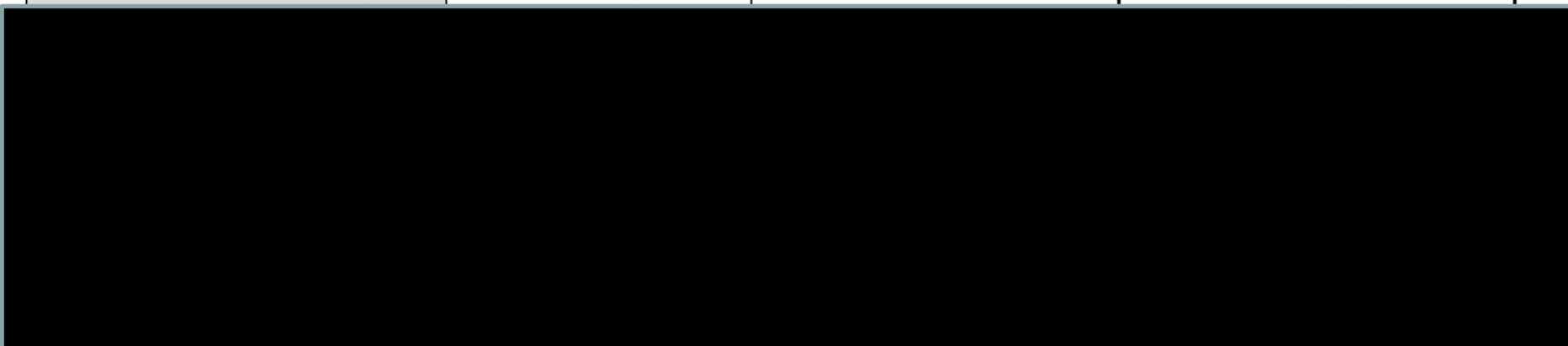


Avaliação da Linguagem

Critérios Específicos:	C	JAVA	Haskell
------------------------	---	------	---------



Passagem de parâmetros	Lista variável e passagem por valor	Passagem por valor e por cópia de referência	Passagem por cópia
------------------------	-------------------------------------	--	--------------------



Avaliação da Linguagem



Critérios Específicos:

C

JAVA

Haskell

Encapsulamento e proteção

Parcial

Sim

Parcial

Avaliação da Linguagem



Critérios Específicos:	C	JAVA	Haskell
Sistema de tipos	Não	Sim	Sim

Avaliação da Linguagem



Critérios Específicos:

C

JAVA

Haskell

Verificação de tipos

Estática

Estática e dinâmica

Fortemente tipada

Avaliação da Linguagem

Critérios Específicos:	C	JAVA	Haskell
------------------------	---	------	---------

--	--	--	--

Polimorfismo	Coerção e sobrecarga	Coerção, sobrecarga e inclusão	Sobrecarga, Coerção, Paramétrico
--------------	----------------------	--------------------------------	----------------------------------

--	--	--	--

Avaliação da Linguagem



Critérios Específicos:	C	JAVA	Haskell
[Redacted Content]			
Exceções	Não	Sim	Parcial

Avaliação da Linguagem



Critérios Específicos:

C

JAVA

Haskell

Concorrência

Não

Sim

Sim

15 - REFERÊNCIAS BIBLIOGRÁFICAS

Referências Bibliográficas

- <https://wiki.haskell.org/Pt/>
- **Livro Haskell :Uma Abordagem Prática**
 - Claudio Cesar de Sá
 - Márcio Ferreira da Silva
- **Livro Linguagens de Programação**
 - Flávio Varejão
- **Livro Introdução à Programação: uma Abordagem Funcional**
 - Alberto Nogueira de Castro Júnior
 - Cláudia Galarda Varassin
 - Crediné Silva de Menezes
 - Maria Christina Valle Rauber
 - Maria Cláudia Silva Boeres
 - Thais Helena Castro