

Seminário de Linguagens de Programação

Ramon Bambini
Josias Alexandre

Departamento de Informática
Centro Tecnológico
Universidade Federal do Espírito Santo

06 Nov 2015



Haskell

Sumário

- 01 - Introdução
- 02 - Amarrações
- 03 - Valores e tipos de dados
- 04 - Variáveis e constantes
- 05 - Gerenciamento de Memória
- 06 - Expressões e comandos
- 07 - Modularização
- 08 - Módulos
- 09 - I/O

- 10 - Polimorfismo
- 11 - Exceções
- 12 - Concorrência
- 13 - Haskell Vs OO
- 14 - Avaliação da Linguagem
- 15 - Referências Bibliográficas

1 – INTRODUÇÃO

Histórico

1930 | Alonzo Church → Cálculo Lambda

1950 | John McCarthy → Lisp baseado no Lambda, porém com atribuição de variáveis

1970 | Robin Milner e outros → ML com inferência de tipos e tipos polimórficos.

1987 | Comunidade de Programação Funcional em conferência em Amsterdã - 1ª

1990 a 92 | Haskell versão 1.0/1.1/1.2

1996 e 97 | Haskell versão 1.3/1.4 (Haskell 98)

1999 | Publicação do Haskell 98

2003 | “Sofre” revisão

2006 | Começou o processo de definição de um sucessor do padrão 98 (Haskell')

2010 | Haskell 2010



Haskell Brooks Curry

Primeira linguagem funcional com influência da teoria de lambda

Lisp

Algol 60

Algol 68

Pascal

ML

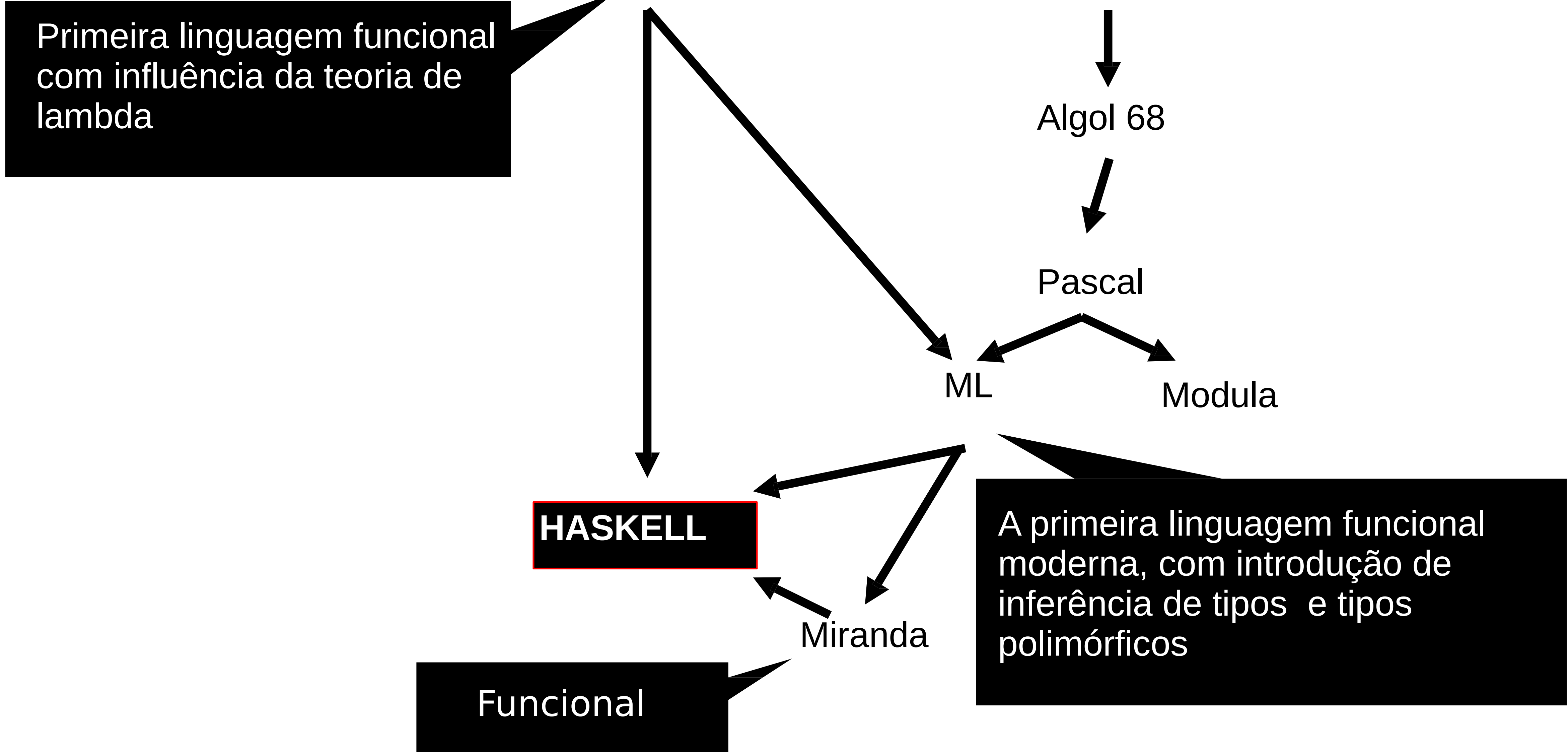
Modula

HASKELL

Miranda

Funcional

A primeira linguagem funcional moderna, com introdução de inferência de tipos e tipos polimórficos



Haskell

C#

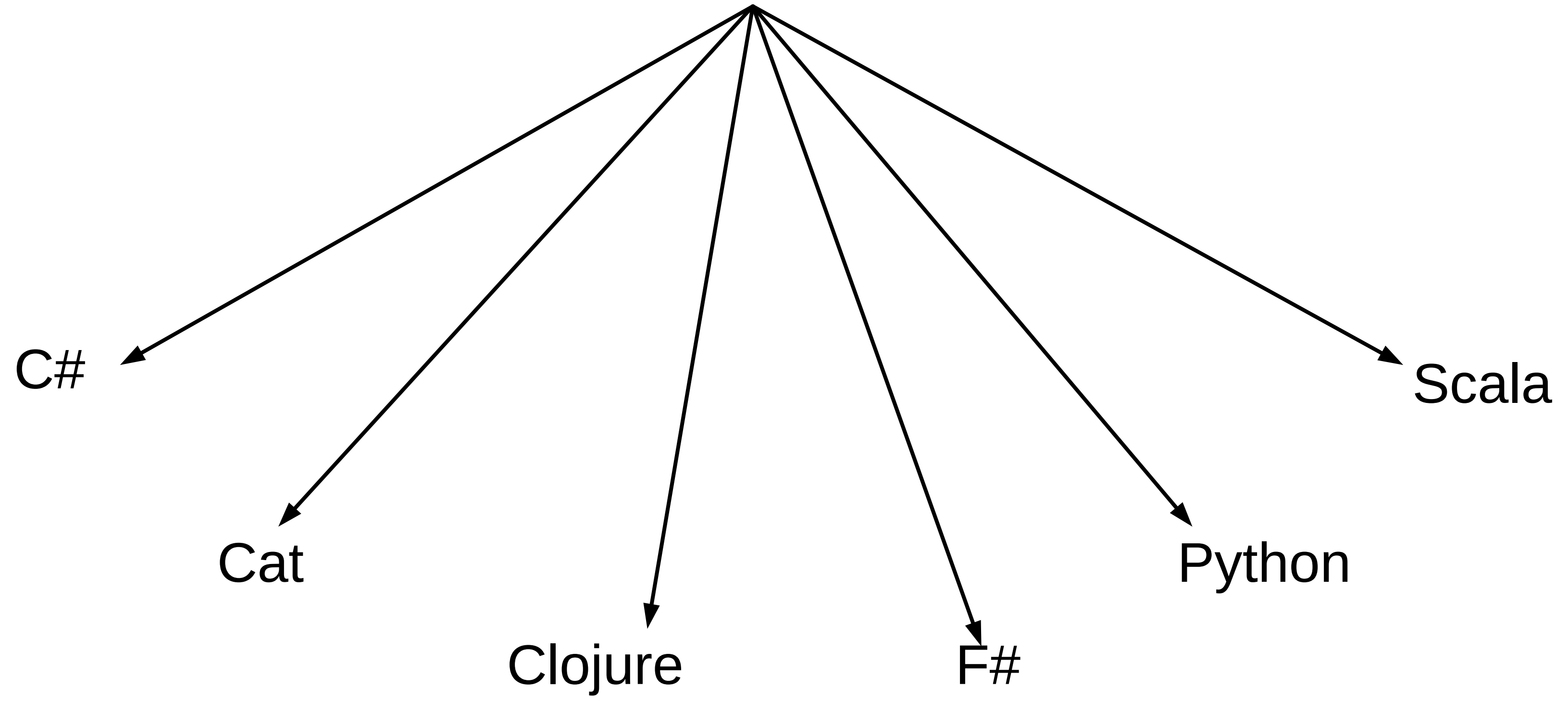
Cat

Clojure

F#

Python

Scala



Até aqui....

- **Haskell** é uma linguagem de programação de propósito geral e adota o paradigma funcional
- É software livre com licença compatível com GPL
- Possui mais de vinte anos de pesquisa e desenvolvimento
- Integra bem com outras linguagens (depuradores, “profilers”, bibliotecas ricas e comunidade ativa)
- Torna fácil produzir software de alta qualidade e de manutenção flexível
- É baseada no cálculo lambda, daí a logo.

Paradigma Funcional e Haskell

$f :: a \rightarrow b$

Ponteiros

Variações

Loops

Paradigma Funcional e Haskell

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

Patterns

Monads

Métodos de Implementação

Pode ser compilado ou interpretado.

- Há vários compiladores à escolha do programador. Os mais usados são o GHC (Glasgow Haskell Compiler) e o Hugs (Haskell User's Gofer System)

- Exemplo de como compilar um programa:

```
$ gch lp.hs -o trabl.exe
```

- Exemplo de como rodar um script, sem compilar:

```
$ runhaskell lp.hs
```

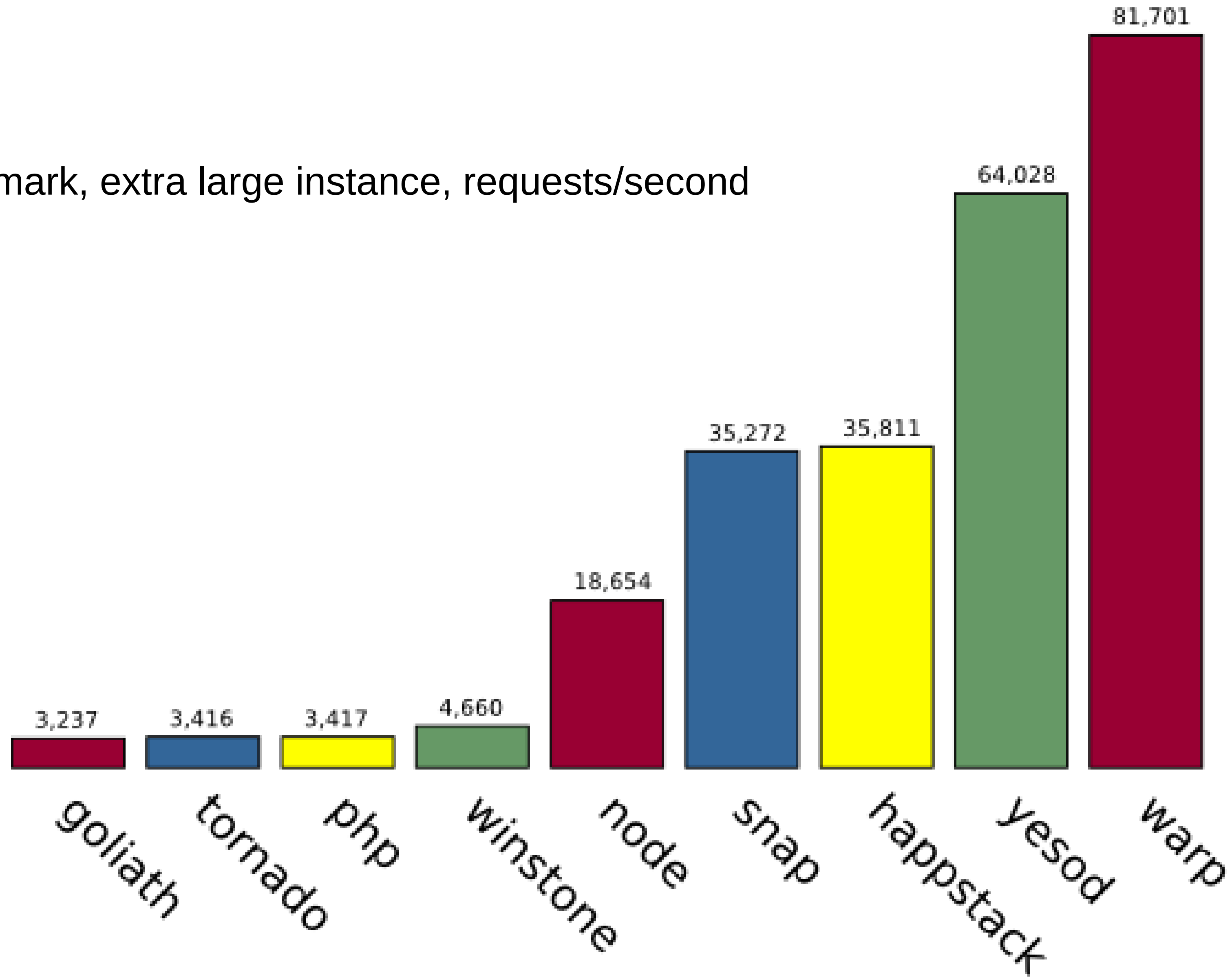
Aplicações

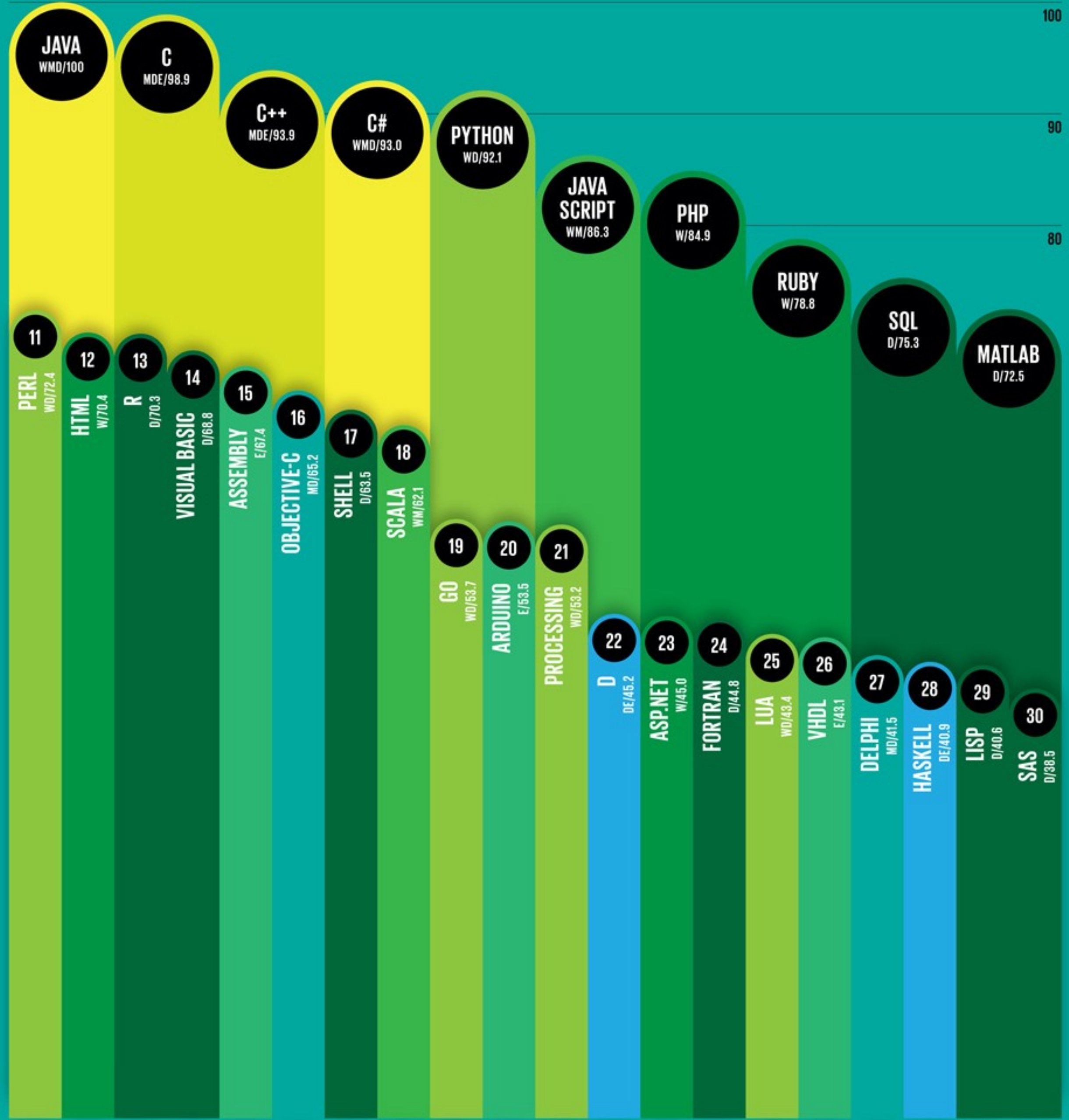
- Computação Simbólica
- Processamento de Listas
- Aplicações científicas
- Aplicações em IA
- Jogos
- Compiladores
- Functional Reactive Animation (FRAN)
- Arte
- Etc...



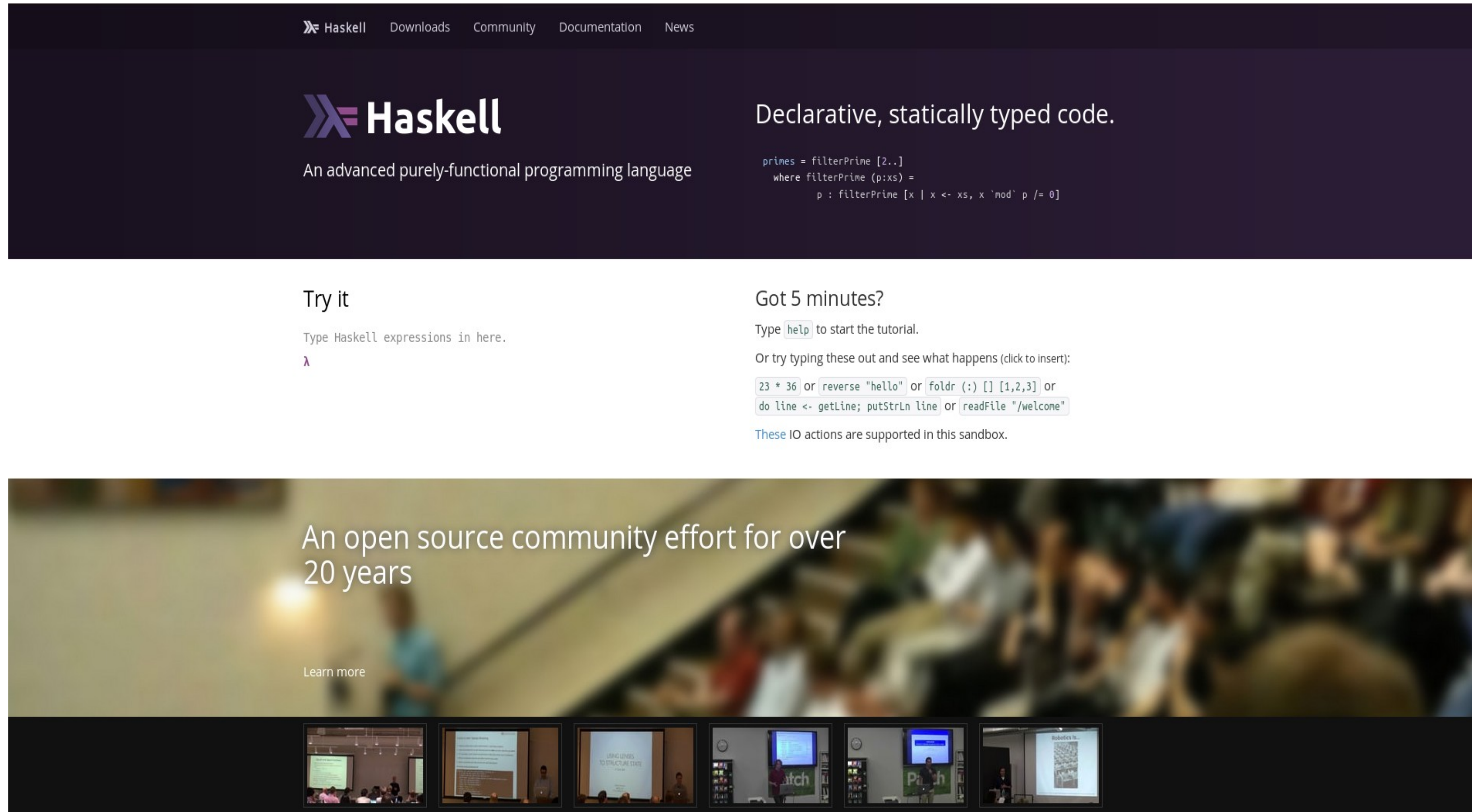
ABN·AMRO

Pong benchmark, extra large instance, requests/second





Como obter



The image is a screenshot of the Haskell website homepage. At the top, there is a navigation bar with links for "Haskell", "Downloads", "Community", "Documentation", and "News". The main header features the Haskell logo and the text "Declarative, statically typed code." Below this, there is a code snippet for a prime number filter. The page is divided into two columns: "Try it" and "Got 5 minutes?". The "Try it" section includes a text input field and a prompt to type Haskell expressions. The "Got 5 minutes?" section provides a tutorial link and several code snippets for arithmetic, string reversal, list folding, and file I/O. A large banner below the main content reads "An open source community effort for over 20 years" with a "Learn more" link. At the bottom, there is a row of six small thumbnail images showing various Haskell community events and presentations.

Haskell Downloads Community Documentation News

Haskell

An advanced purely-functional programming language

Declarative, statically typed code.

```
primes = filterPrime [2..]
where filterPrime (p:xs) =
      p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

Try it

Type Haskell expressions in here.

λ

Got 5 minutes?

Type `help` to start the tutorial.

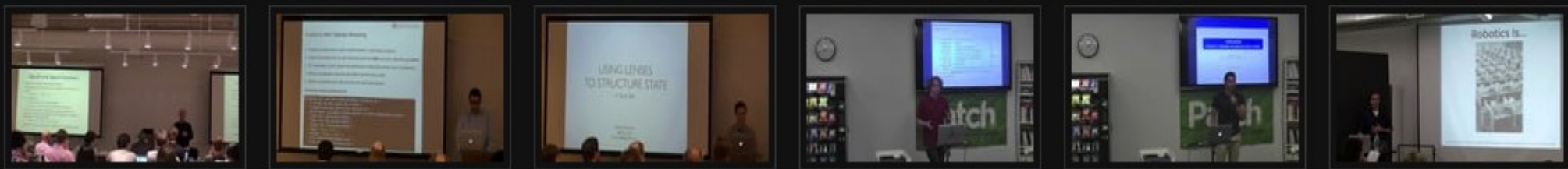
Or try typing these out and see what happens (click to insert):

`23 * 36` or `reverse "hello"` or `foldr (:) [] [1,2,3]` or
`do line <- getLine; putStrLn line` or `readFile "/welcome"`

[These IO actions are supported in this sandbox.](#)

An open source community effort for over 20 years

[Learn more](#)



Como obter

Três sabores:

- Mínimo: só instalar o GHC e o Cabal pelo terminal de qualquer distribuição decente. As dependências de cada projeto são instaladas globalmente no sistema.
- Stack: instala o stack globalmente no sistema e ele gerencia os projetos e dependências individualmente = -)
- Haskell Platform: instala outras ferramentas e bibliotecas extras. Também no sistema e globalmente.

Experimentem o Stack!

2 – Amarrações

"é uma associação entre entidades de programação, como entre uma variável e seu valor, ou entre um **identificador** e um **tipo**."

(Varejão, Flávio)

Identificador é: "uma cadeia de caracteres definidos pelos programadores para servirem de referências às entidades de computação".

Identificadores

- Sem limites de tamanho
- Devem necessariamente começar com letras e em seguida, opcionalmente, dígitos, sublinhas ou apóstrofes.
- Case-sensitive
- Funções devem sempre começar com letra minúscula
- Convenção de nomes

Palavras reservadas

case

class

data

deriving

do

else

if

import

in

infix

infixl

infixr

instance

let

of

module

newtype

then

type

then

```
module Module1Name where
```

```
f :: a → b
```

```
g :: a → b
```

```
module Module2Name (f, g) where
```

```
h :: a → b
```

```
g' :: a → b
```

```
f :: a → b
```

Programa em Haskell → Estrutura Léxica

Escopo de visibilidade de uma amarração

- Estático - Bloco
- Identificadores, dentro de um escopo, assumem um valor na criação e nunca mudam
- Imensa maioria das vezes “early binding” mas suporta “late binding” de forma poderosa. (+)

Definições e Declarações

“frases de programa elaboradas para produzir amarrações”

- Definições → amarra identificadores e entidades criadas na própria definição
- Declarações → amarra identificadores e entidades já criadas ou ainda por criar
- Não existe atribuição em **Haskell**, e sim definição!
- O comando de definição é o sinal “=”.

Ex: $x = 2$

Ex: $y = \text{“string”}$

3 – Valores e Tipos de Dados

Valor e Tipo

"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)

"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)

- Tipos numéricos
 - Inteiros: `Int`, `Integer`
 - Reais: `Float`, `Double`
- Character: `Char` (`'c'`, `'/'`, `'j'`, etc)
- Lógico: `Bool` (**`True`**, **`False`**)
- Vazio: `Void`

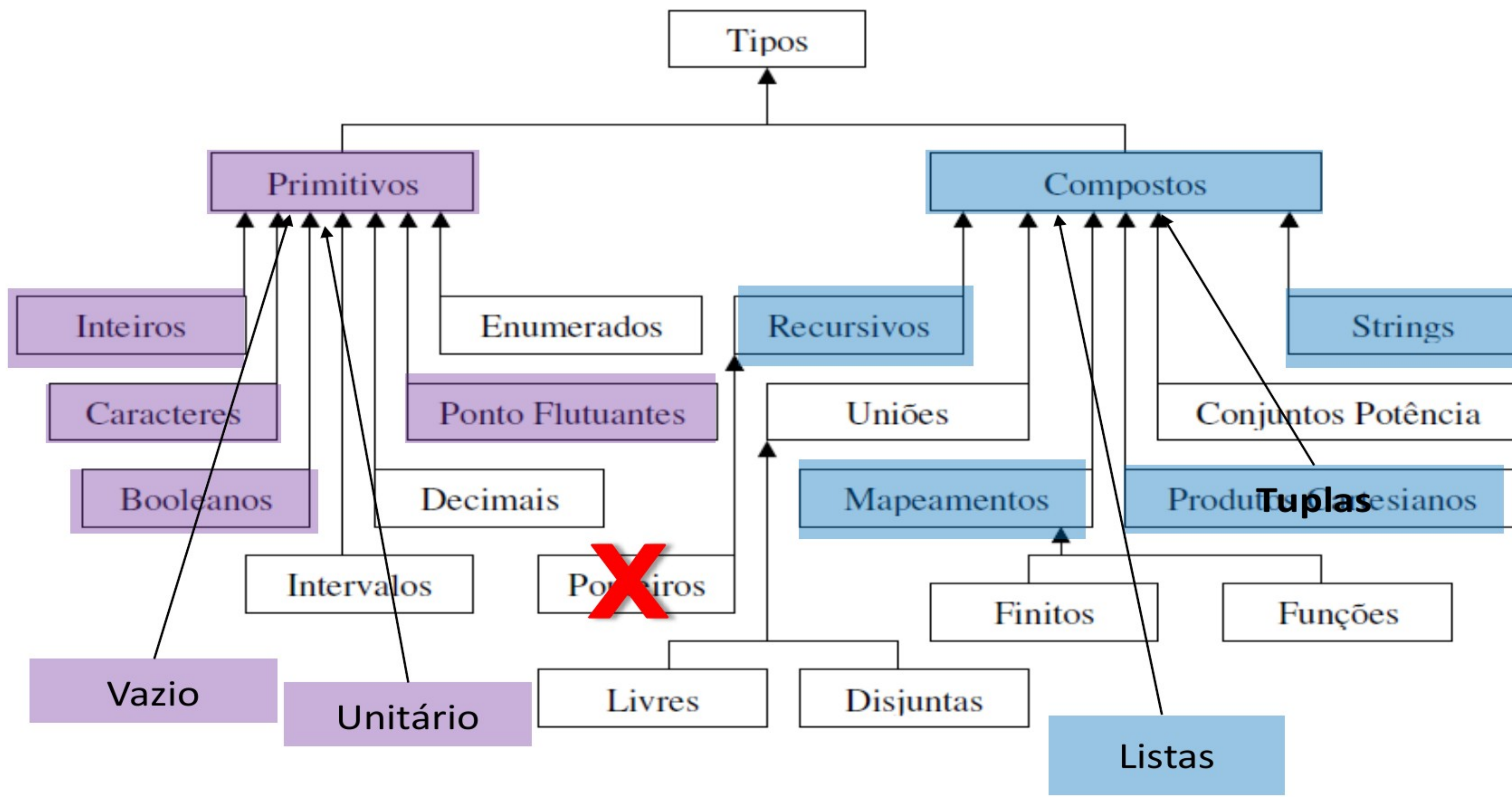
Tipos Compostos

“aqueles criados a partir de tipos mais simples.” (VAREJÃO, Flávio)

- Strings → “Ramon”
- Tuplas (1, 3.0, “Ramon”) == Produto cartesiano
- Listas → [1, 2, 3], [“Josias”, “LP”], [[(“LP”, 'c', True), (“Josias”, 's', False)]]

Mapeamentos

```
parseMembers :: [[String]] -> [UniversityMember]
ParseMembers [] = []
parseMembers (m:ms) =
  case buildMember m of
    Just member -> member : parseMembers ms
    Nothing -> parseMembers ms
```



Conversão entre tipos

- Em **Haskell** não há coerção.
- O usuário deve converter explicitamente um dado tipo em outro.
- **Haskell**, então, apresenta um série de funções (biblioteca padrão **Prelude**) que podem “tentar” fazer as devidas conversões.

Qualquer tipo para `String` ou `[Char]`

`show` :: **Show** a => a -> `String`

`String` ou `[Char]` para qualquer tipo

`read :: Read a => String -> a`

4 – Variáveis e constantes

■ ■ ■

5 – Gerenciamento de Memória

Gerenciamento de Memória

- O GHC utiliza um GC (Garbage Colector) para cada geração
- Novos dados são alocados num berçário de 512k
- Quando o berçário estiver cheio, apenas os valores utilizados na iteração atual sobrevivem.
- Pela imutabilidade dos dados, temos garantia que não existe um “ponteiro” apontando pra dados de outras iterações.
- Quanto maior o percentual de valores lixo, mais rápido funciona!!

6 – Expressões e Comandos

Expressão vs Comando

“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)

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

Construção de expressões

Em **Haskell**, construímos expressões a partir de:

- Aplicação de **funções** (algumas até disfarçadas de operadores)
- Utilização de parênteses
- Alguns poucos operadores
- Valores literais, ou constantes

Operadores básicos (007)

>

>=

==

/=

<

<=

&&

||

not

+

-

*

/

^

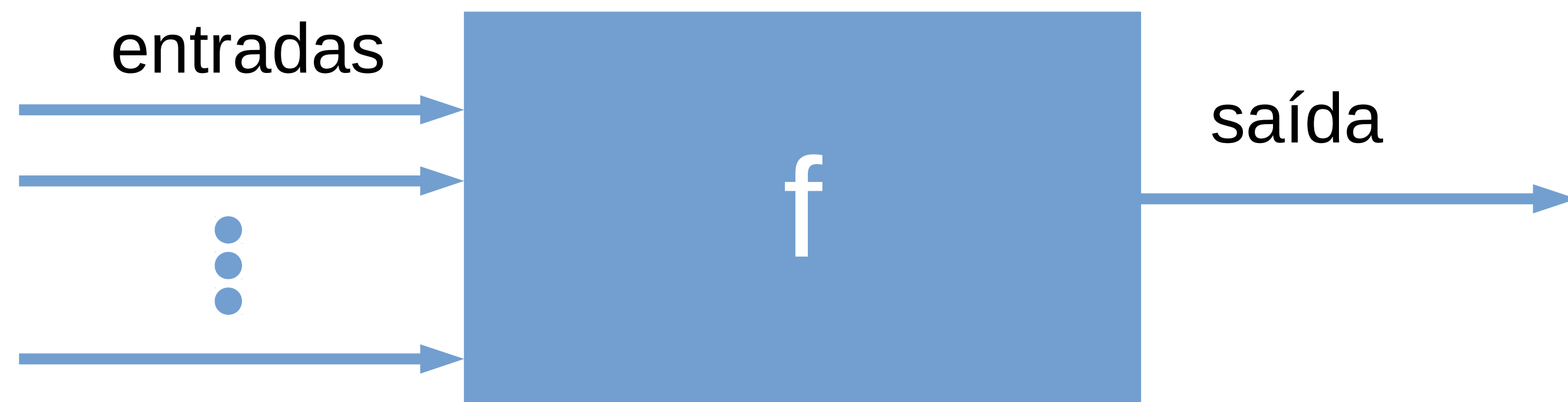
div

mod

abs

negate

Função



Definição de Função

$$f = 2$$



Definição de Função

$g = \text{"string"}$



Inferência de tipos

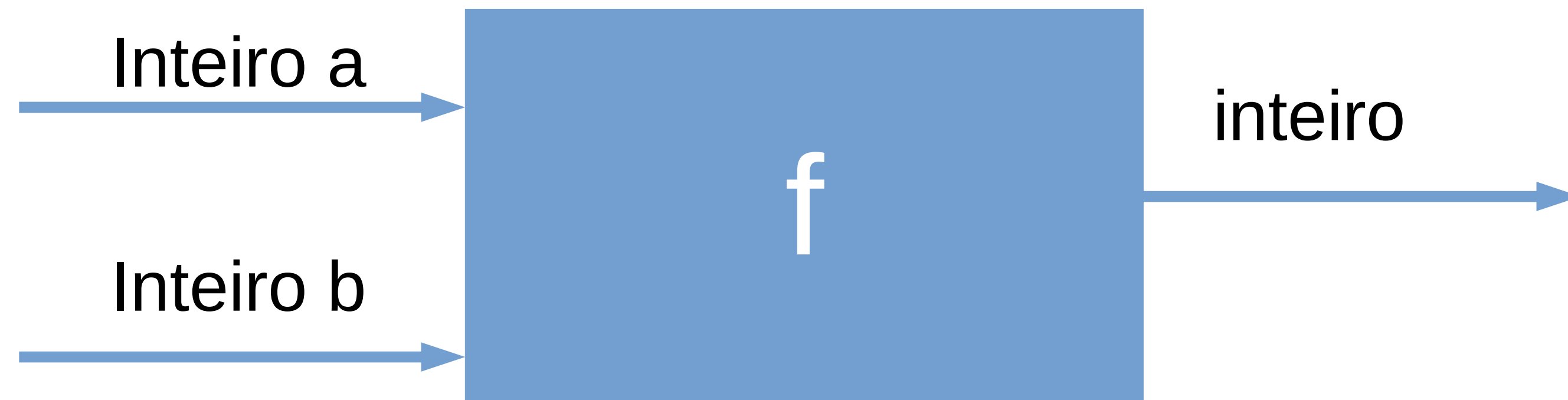


$f :: \text{Num } a \Rightarrow a$
 $f = 2$



$g :: [\text{Char}]$
 $g = \text{"string"}$

Parâmetros ou Entradas



$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$f \ a \ b = 2*a + 2*b$

Parâmetros ou entradas

`g :: [Char] → [Char]`

`g name = "User name: " ++ name`

`h :: ([Char] → [Char]) → Int → Char`

`h func index = func !! index`

Definições Locais a uma Função - **where**

```
-- verify if the registration are the same
sameReg :: UniversityMember -> UniversityMember -> Bool
sameReg a b =
    not $ null $ intersect regA regB
  where
    regA = [fstReg a, sndReg a]
    regB = [fstReg b, sndReg b]
```

Definições Locais a uma Função - **let**

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$f\ a\ b = a + b * c$

where

$c = \mathbf{let\ } x = a \mathbf{\ in\ } x^2 + 2 * x - 1$

Expressões condicionais

Temos as seguintes possibilidades:

- **If then else** para um caso binário
- As guardas que escolhem o fluxo na definição da função: “|”
- A expressão **case of ->** (Similar a um switch)
- Métodos mais sofisticados através de Functors, Applicatives e Monads.

```

removeMemberByReg :: Integer -> [UniversityMember] -> [UniversityMember]
removeMemberByReg _ [] = []
removeMemberByReg reg' (um:ms) =
    if hasTwoRegistrations um
    then
        -- the member has 2 registrations
        if e_reg um == reg'
        then
            (removeEmployeeData um) : ms
        else
            if s_reg um == reg'
            then
                removeStudentData um : ms
            else
                um : removeMemberByReg reg' ms
    else
        -- the member has only 1 registration
        if isStudent um
        then
            if s_reg um == reg'
            then
                ms
            else
                um : removeMemberByReg reg' ms
        else
            -- it's an employee
            if e_reg um == reg'
            then
                ms
            else
                um : removeMemberByReg reg' ms

```

Expressões condicionais - Guardas

```
f :: Int -> Int -> Int
f a b
  | a < b = a + b*c
  | 0 == b = a + c
  | otherwise = a*c + b
where
  c = let x = a in x^2 + 2*x - 1
```

Expressões condicionais – **case**

```
parseMembers :: [[String]] -> [UniversityMember]
ParseMembers [] = []
parseMembers (m:ms) =
  case buildMember m of
    Just member -> member : parseMembers ms
    Nothing -> parseMembers ms
```

Recursão

```
parseMembers :: [[String]] -> [UniversityMember]
ParseMembers [] = []
parseMembers (m:ms) =
  case buildMember m of
    Just member -> m : parseMembers ms
    Nothing -> parseMembers ms
```

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

```
parseMembers :: [[String]] -> [UniversityMember]
ParseMembers [] = []
parseMembers (m:ms) =
  case buildMember m of
    Just member -> m : parseMembers ms
    Nothing -> parseMembers ms
```

7 – Modularização

Mecanismo de passagem de parâmetros

- Sem efeitos colaterais
- Parâmetros são sempre passados por valor
- Se uma função pura é chamada duas vezes com os mesmos parâmetros, o resultado retornado será sempre o mesmo.
- Mudar o identificador dentro de um escopo só tem validade dentro desse escopo.

TADs

- Criação de sinônimos através da palavra reservada **type**
- Criação de novos tipos de dados através da palavra reservada **data**

```
-- new type
type Day = Int

-- new data type – Enumerate like
data Month = Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec
           deriving (Show, Eq, Ord)

-- new data type – now with constructors and values
data Shape = Circle Double Double Double | Rectangle Double Double Double Double
```

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)

Como lidar com módulos

```
module Date (Month (..), Date (..), readMaybeMonth, readMaybeDate) where

import Data.Text as T
import Prelude hiding (split)
import Text.Read (readMaybe)
import Lib (readMaybeInt)

-- new type
type Day = Int

-- new data type - Enumerate
data Month = Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec
deriving (Show, Eq, Ord)

. . .
```


E se eu não tiver o módulo?

- A comunidade provê um repositório chamado Hackage contendo uma miríade de pacotes/módulos pra todo tipo de projeto (Bitcoin a openCV e por aí vai)
- Para obter o módulo, pode-se fazer o download diretamente ou utilizar um gerenciador de pacotes como o cabal. No ubuntu, por exemplo:

```
$ sudo apt-get install cabal-install
```

- Para instalar um pacote com o cabal:

```
$ cabal install nomeDoModulo
```

- Stack

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 básicos, e um mecanismo para combinar ações de entrada e saída.

Ações de saída padrão

putChar

putStr

putStrLn

print

Ações de entrada padrão

getChar

getLine

getContents

readLn

```
C:\Users\USE\Desktop\Haskell\io.hs - Notepad++
Arquivo  Editar  Localizar  Visualizar  Formatar  Linguagem  Configurações  Macro  Executar  Plugins  Janela  ?  X
io.hs x
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) )

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 – Polimorfimos

Sistema de Tipos

- Fortemente tipado: disciplina rigorosa com os tipos.
- Tanto os compiladores quanto os interpretadores utilizam a checagem de tipos.
- Tudo tem um tipo. Todas as expressões, funções e até os operadores possuem tipos: ou seja, o compilador sempre vai determinar com rigor qual o tipo de dada expressão: quais os tipos dos dados que estão envolvidos.
- Apesar de todo rigor, o sistema infere tudo. Não é requerido do programador que ele defina o tipo de cada dado utilizado.
- Grande parte dos erros são verificados em tempo de compilação:

`not 'True'`

- Para verificar o tipo de qualquer coisa em Haskell pode-se usar o GHCi e diretamente na linha de comando utilizar o comando `:type` ou `:t`

```
$ :t read
```

```
>> read:: Read a => String -> a
```

Sistema de Tipos

- Literais inteiros são do tipo: Num a => a
- Literais fracionários são do tipo Fractional a => a
- Literais characters são do tipo Char
- Literais strings são do tipo String que é apenas um sinônimo para [Char]
- Construtores constantes são idênticos ao texto: True e False

Ad-hoc

- Sem Coerção! Haskell não faz coerção implícita de tipos. Isso não é coerção:

```
$ ghci >> 14 + 1.25
```

```
$ ghci >> 15.25
```

- Pode se sobrecarregar as funções através de classes de tipo. É o que aconteceu acima no fundo.

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.
- 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

- Criação de classes de tipo através da palavra reservada **class**
- Implementação de uma dada classe em um tipo de dados através das palavras **deriving** (automático) e **instance** (manualmente)

```
data Genre = Female | Male deriving (Read, Show, Eq)
```

```
class Person a where  
  name :: a -> Text  
  age  :: a -> Int  
  genre :: a -> Genre
```

```
-- new data type - Enumerate
data Month = Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec
deriving (Show, Eq, Ord)

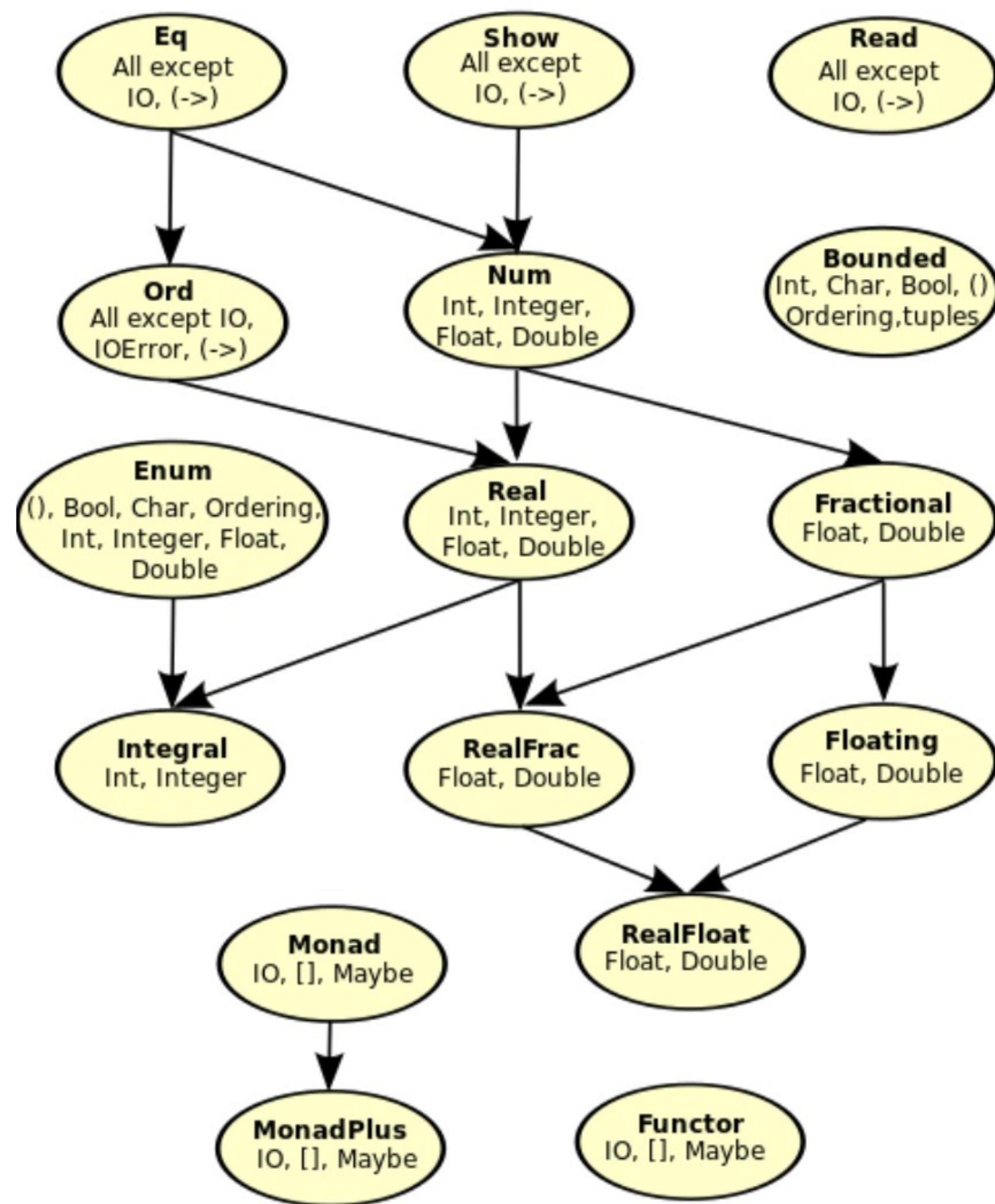
-- new data type - Record Syntax
data Date =
  Date {
    day :: Int
    , month :: Month
    , year :: Int
  } deriving (Eq)

-- the Show class implementation
instance Show Date where
  show (Date d m y) = show d ++ " " ++ show m ++ " " ++ show y
```

```
data Teacher =  
  Teacher {  
    t_name :: String  
    , t_age :: Int  
    , t_genre :: Genre  
    , t_reg :: Integer  
    , t_adm :: Date  
    , t_salary :: Float  
    , t_regime :: Regime  
  } deriving (Show, Eq)
```

```
-- the Person class implementation  
instance Person Teacher where  
  name = t_name  
  age = t_age  
  genre = t_genre
```

```
class Person a where  
  name :: a -> String  
  age :: a -> Int  
  genre :: a -> Genre
```



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


Universal

- Haskell não é OO, então sem polimorfismo de inclusão.
- Haskell possui poliformismo paramétrico, muito similar aos templates em C++
- Algumas funções podem ser definidas para serem totalmente genéricas como a seguinte função identidade ou a função que retorna o tamanho de uma lista:

```
f :: a -> b  
f a = a
```

```
length :: [a] -> Int  
length [] = 0  
length (_:t) = 1 + length t
```

11 – Exceções

Excessões

- 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.
- 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.

Excessões

```
import Control.Exception
```

```
getLines = liftM lines . readFile
```

```
main :: IO ()
```

```
main = do
```

```
  master <- try (getLines "static/mestre.txt") :: IO (Either IOException [String])
```

```
  case master of
```

```
    Right strings -> fmapM_ putStrLn master
```

```
    Left ioexc -> putStrLn "Error: " ++ (show ioexc)
```

Excessões - error

```
myDiv1 :: Float -> Float -> Float
myDiv1 x 0 = error "Division by zero"
myDiv1 x y = x / y
```

```
import qualified Control.Exception as E

example1 :: Float -> Float -> IO ()
example1 x y =
  E.catch (putStrLn (show (myDiv1 x y)))
    (\err -> putStrLn (show err))
```

Excessões - Maybe

```
myDiv2 :: Float -> Float -> Maybe Float
myDiv2 x 0 = Nothing
myDiv2 x y = Just (x / y)

example2 x y =
  case myDiv2 x y of
    Nothing -> putStrLn "Division by zero"
    Just q   -> putStrLn (show q)
```

```
divSum2 :: Float -> Float -> Float ->
         Maybe Float
divSum2 x y z = do
  xdy <- myDiv2 x y
  xdz <- myDiv2 x z
  return (xdy + xdz)
```

Excessões - Custom

```
import Control.Monad.Error

data CustomError = DivByZero
                 | OutOfCheese
                 | MiscError String

instance Show CustomError where
  show DivByZero = "Division by zero"
  show OutOfCheese = "Out of cheese"
  show (MiscError str) = str

instance Error CustomError where
  noMsg = MiscError "Unknown error"
  strMsg str = MiscError str

myDiv5 :: (MonadError CustomError m) =>
         Float -> Float -> m Float
myDiv5 x 0 = throwError DivByZero
myDiv5 x y = return (x / y)

example5 :: Float -> Float ->
          Either CustomError String
example5 x y =
  catchError (do q <- myDiv5 x y
                return (show q))
             (\err -> return (show err))
```

12 – Concorrência

Concorrência

- Suporte a concorrência por padrão. Basta incluir o módulo `Control.Concurrent` e utilizar as funções que criam novas threads.
- Em Haskell, `thread` é uma IO action.
- Para comunicação entre threads utiliza-se os tipos `MVar`
- A linguagem e o compilador estão bem maduros
- Há formas bem simples de se programar implicitamente a concorrência e outras formas mais avançadas.

13 – Haskell Vs OO

■ ■ ■

14 – Avaliação da Linguagem

Critérios gerais

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

Critérios Gerais	C	JAVA	Haskell
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Não	Sim	Sim
Facilidade de Aprendizado	Não	Não	Não
Eficiência	Sim	Parcial	Entre sim e parcial
Portabilidade	Não	Sim	Parcial
Suporte ao método de projeto	Estruturado	OO	Funcional
Evolutibilidade	Não	Sim	Sim
Reusabilidade	Parcial	Sim	Sim
Integração	Sim	Parcial	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

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

Cr�terios Gerais	C	JAVA	Haskell
Escopo	Sim	Sim	Sim
Express�es e comandos	Sim	Sim	Sim
Tipos prim. e compostos	Sim	Sim	N�o
Gerenciamento de mem�ria	Programador	Sistema	Sistema
Passagem de par�metros	Lista vari�vel e por valor	Valor e c�pia de refer�ncia	C�pia
Persist�ncia de dados	Biblioteca de fun�es	Biblioteca de classes, serializa�o e JDBC	Classe mon�dica de IO
Encapsulamento e prote�o	Parcial	Sim	Parcial
Sistema de tipos	N�o	Sim	Sim
Verifica�o de tipos	Est�tica	Est�tica e Din�mica	Fortemente tipada
Polimorfismo	Coer�o e sobrecarga	Coer�o, sobrecarga, inclus�o e param�trico	Sobrecarga e param�trico
Exce�es	N�o	Sim	Sim
Concorr�ncia	N�o	Sim	Sim

Vantagens

- Suporte à simultaneidade e paralelismo;
- Apoiado por uma grande biblioteca de módulos de pacotes;
- Fornecido com depuradores e profilers;
- Livrementemente disponível
(código-fonte aberto, pacote de desenvolvimento completo);
- Fortemente Tipada e Estática;
- Avaliação Lazy;
- Polimorfismo Universal Paramétrico;
- Função (superior e parcial);
- Ausência de desvios incondicionais;

Desvantagens

- Apesar de poderoso, o paradigma é de difícil aprendizagem
- Algumas áreas ainda estão em pesquisa e não 100% resolvidas (poucas)

15 – Referências

Referências

- <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
- Apresentação Seminário Pâmela e Vitor (2015/1)
- <http://spectrum.ieee.org/computing/software/top-10-programming-languages>

