

HASKELL



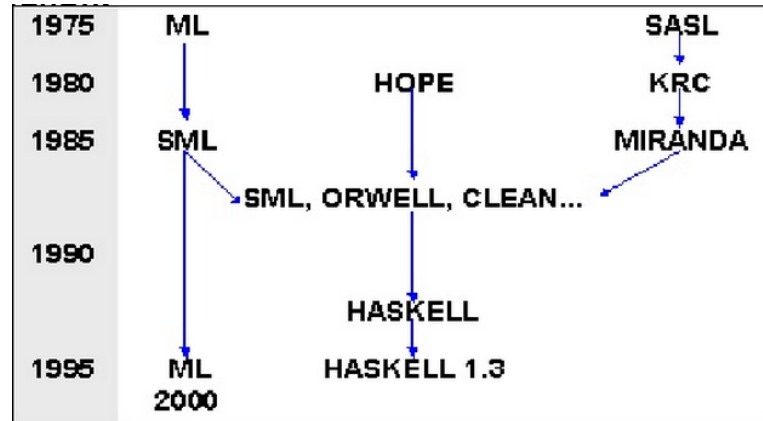
Breno Simão Boscaglia
Pablo Santana Satler
Ramon Pereira de Matos

História

- 1930: Alonzo Church desenvolveu o cálculo de **Lambda**, um simples e poderoso teorema de funções.
- 1950: John McCarthy desenvolveu **Lisp**, a primeira linguagem funcional, que tinha influência da **teoria de 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.

História

- Em 1987, ocorreu a conferência “Functional Programming Languages and Computer Architecture”, consolidação de linguagens existentes e criação de um padrão aberto.
- Em 1990, lançou-se a primeira versão, criada por Simon Peyton-Jones, Paul Hudak e Philip Walder e nomeada em homenagem ao lógico Haskell Curry.
- Surgiram novas versões até que em 1999 foi publicada o Haskell 98, que especifica uma versão mínima, estável e portátil da linguagem.
- Constante evolução: Implementação Hugs e GHC.



O que é Haskell?

Haskell é uma linguagem de paradigma funcional

Mas o que isso quer dizer?

- Em linguagens imperativas, é dado ao computador uma sequência de tarefas, os quais ele executa, podendo inclusive mudar o estado delas.
- Ex.: Atribui-se à variável “a” o valor “5”. Em seguida, são feitas algumas computações e o valor daquela variável se torna “7”. Houve mudança de estado.

Programação Funcional

- Já em programação funcional não fazemos atribuições, mas sim definições.
- Se foi definido que “a” é 5, não se pode depois definir “a” como 7.

Programação Funcional

Portanto,

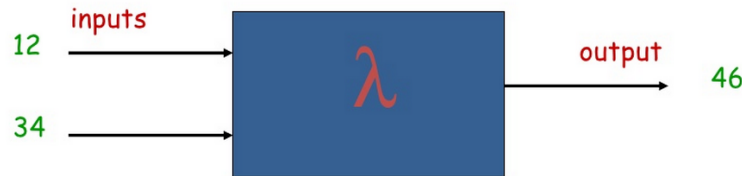
- os dados são imutáveis e são evitados estados.
- Não existe efeitos colaterais.

Conceitos fundamentais da prog. funcional incluem:

- “Funções puras” (sem efeitos colaterais)
- Cálculo Lambda

Paradigma Funcional

- É um paradigma de programação que trata a computação como uma avaliação de funções matemáticas;
- Mapeamento dos valores de entrada nos valores de retorno, por meio de funções



- Um programa funcional é uma única expressão que é executado por meio de avaliação da expressão

Paradigma de Haskell

- Puramente funcional
- Modular
- **Lazy evaluation:** Técnica usada em programação para atrasar a computação até um ponto em que o resultado da computação é considerado necessário. Só se avalia as partes do programa quando é necessário para o cálculo que é efetuado.
- Sintaxe simples, elegante e concisa: Há redução em linhas de códigos numa razão de 2 a 10.

Comparação quicksort C - Haskell

```
void qsort(int a[], int lo, int hi)
{
    int h, l, p, t;

    if (lo < hi) {
        l = lo;
        h = hi;
        p = a[hi];

        do {
            while ((l < h) && (a[l] <= p))
                l = l+1;
            while ((h > l) && (a[h] >= p))
                h = h-1;
            if (l < h) {
                t = a[l];
                a[l] = a[h];
                a[h] = t;
            }
        } while (l < h);

        a[hi] = a[l];
        a[l] = p;

        qsort( a, lo, l-1 );
        qsort( a, l+1, hi );
    }
}
```

```
quicksort :: Ord a => [a] -> [a]
quicksort []      = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser = filter (< p) xs
        greater = filter (>= p) xs
```

Propriedades de Haskell

Possui boa:

- Redigibilidade
- Confiabilidade
- Reusabilidade
- Modificabilidade
- Alta Portabilidade

Métodos de Implementação

- Pode ser compilado ou interpretado.
- Há vários compiladores à escolha do programador. Um dos mais usados é GHC (Glasgow Haskell Compiler) e o Hugs (Haskell User's Gofer System)
- Exemplo de como compilar um programa.

```
breno@vingador:~/Área de Trabalho$ ghc --make lp.hs -o trabl.exe  
Linking trabl.exe ...  
breno@vingador:~/Área de Trabalho$ ./trabl.exe
```

Métodos de Implementação

Fazendo interpretação do código

- Digitar runhaskell nomeDoPrograma.hs

```
pablo@pablo-Dell:~/Desktop/haskell_Trab_LP$ runhaskell lp.hs
```

- Os programas em Haskell são normalmente chamados de script, por isso a extensão “**hs**” (Haskell Script).

Métodos de Implementação

Implementadores:

Hugs

- Interpretador;
- Escrito em C;
- Portável;
- Leve;
- Ideal para iniciantes;

GHC

- Interpretador (GHCi) e Compilador;
- Escrito em Haskell;
- Menos portável;
- Mais lento;
- Exige mais memória;
- Produz programas mais rápidos;

Métodos de Implementação

```
pablo@pablo-Dell:~$ hugs
```

```
  _ _ _ _ _  
  ||  ||  ||  ||  ||  ||  || | |
  ||__||  ||__||  ||__||  ||__||  
  ||--||      ||__||  
  ||  ||  
  ||  ||  
  ||  || Version: September 2006
```

```
Hugs 98: Based on the Haskell 98 standard  
Copyright (c) 1994-2005  
World Wide Web: http://haskell.org/hugs  
Bugs: http://hackage.haskell.org/trac/hugs
```

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

```
Type :? for help
```

```
Hugs> :q
```

```
[Leaving Hugs]
```

```
pablo@pablo-Dell:~$
```

```
pablo@pablo-Dell:~$ ghci
```

```
GHCI, version 7.4.1: 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> :q
```

```
Leaving GHCi.
```

```
pablo@pablo-Dell:~$
```

Amarrações

- Todos os tipos são conhecidos em tempo de compilação.
- Apresenta associação estática.
- Existe inferências de tipos. Portanto, não é preciso explicitar de que tipo é uma certo identificador.
- Haskell permite que um mesmo identificador seja declarado em diferentes partes do programa, possivelmente representando diferentes entidades.

Amarrações

Amarração estática. Consequências:

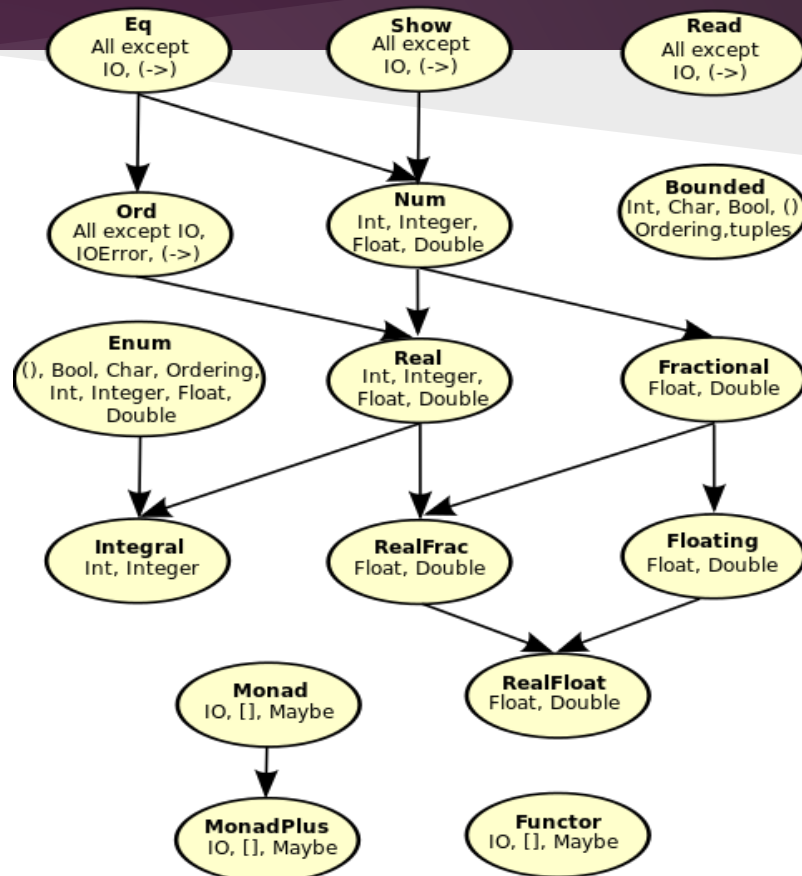
- As conversões de dados são explícitas

```
precoAux2 = read precoAux :: Float
```

- Muitos bugs/erros de códigos são “pegos” em tempo de compilação

Valores e tipos de dados

- Cada expressão bem formada tem um tipo que é calculado automaticamente em tempo de compilação usando **inferência de tipos**;



Valores e tipos de dados

Para verificar de que tipo é um identificador/expressão, basta digitar

`:t identificador/expressão`

Onde `::` significa “**tem tipo de**”

```
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 primitivos

Alguns tipos são:

- **Int:** Inteiros com precisão limitada. Em máquinas 32 bits, por exemplo, com valores entre 2147483647 a -2147483648
- **Integer:** Também inteiros, mas sem limite de precisão, usados para representar números muito grandes.

```
*Main> :t fatorial
fatorial :: Integer -> Integer
*Main> fatorial 50
30414093201713378043612608166064768844377641568960512000000000000
```

Tipos de dados compostos

- **Tuplas:** É uma agregação de um ou mais componentes, que podem ser de tipos diferentes. Elas são denotadas por parenteses e seus componentes separados por vírgulas.

ex.: (“Fulano de Tal”,3337721,”M”)

- **Listas:** Agregação de um ou mais componentes do mesmo tipo.

ex.: [1,2,3,4,5] -> lista de inteiros

[“fulano”, “ciclano”, “beltrano”] -> lista de strings

Tipos de dados primitivos

- **Float:** Ponto flutuante precisão simples. Ex.: 3.0
- **Double:** Ponto flutuante precisão dupla. Ex.: 3.23433
- **Char:** Character. Ex.: 'a'
- **String:** Cadeia de caracteres representado sob a forma de lista de char.

```
Prelude> "Texto" == ['T','e','x','t','o']  
True
```

Tipos de dados compostos

- **Tipos recursivos:** Grande parte das definições em Haskell serão recursivas, principalmente as que precisam de algum tipo de repetição.

```
fatorial :: Integer -> Integer  
fatorial n = n * fatorial(n-1)
```

Tipos de dados compostos

- Algumas 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.

Tipos de dados compostos

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.

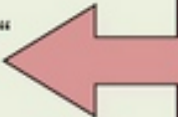
```
ghci> 10 `elem` [3,4,5,6]  
False
```

Tipos de dados compostos

```
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"
```

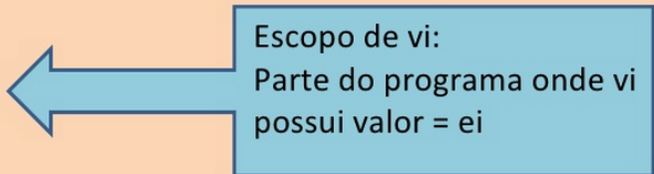


Selecionando intervalo de
elementos em listas.

Regras de Escopo

- As regras de escopo em **Haskell** ditam que se uma expressão está na forma apresentada abaixo, o escopo do identificador vi é o mesmo de e , assim como todos do lado direito de todas as definições de *let*.

```
let     $v1 = e1$   
       $v2 = e2$   
      ...  
       $vi = ei$   
      ...  
in  $e$ 
```



Escopo de vi :
Parte do programa onde vi
possui valor = ei

Variáveis e Constantes

- Em Haskell as variáveis não variam.
- Tudo é constante: Variáveis assumem um valor quando são criadas e este valor nunca muda dentro do escopo da mesma.

```
let quant = read quantidade
    precoAux = replace preco "," "."
    precoAux2 = read precoAux :: Float
    soma = quant * precoAux2
```

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

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 contraintuitivo: Quanto maior o percentual de seus valores são lixo, mais rápido ele funciona.

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.

Expressões e Comandos

- Uma expressão é avaliada como um **valor** e tem um **tipo estático**.
- Valores e tipos **não** são misturados em Haskell.
- Haskell utiliza a ordem normal de avaliação (*leftmost-outertmost*).
- Semelhante a avaliação curto-circuito de linguagens tradicionais
- Haskell permite:
 - Definir tipos de dados pelo usuário;
 - Polimorfismo paramétrico;
 - Sobrecarga (usando tipos classes);
- Erros em Haskell são tratados por exceções.

Expressões e Comandos: Sintaxe

- Trabalha-se somente com funções, seus parâmetros e seus retornos.
- É case-sensitive.
- Não possui comandos de repetição como *While* e *For*.
- Não possui comando de *goto*.
- Sem limites para identificadores.
- Variáveis devem começar com letras minúsculas ou sublinhado
- Nomes que denotam tipos de classes, módulos, classes de tipos, começam com letra maiúscula.

Expressões e Comandos

- Palavras reservadas :

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

Operadores Básicos

>	Maior
>=	Maior ou igual
==	Igual
/=	Diferente
<	Menor
<=	Menor ou igual
&&	e
	ou
not	Negação
++	Concatenação

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

Expressões e Comandos

- Equivalência entre matemática e Haskell

Matematicamente

$f(x)$

$f(x, y)$

$f(g(x))$

$f(x, g(y))$

$f(x)g(y)$

Haskell

`f x`

`f x y`

`f (g x)`

`f x (g y)`

`f x * g y`

Expressões e comandos: Regras de Layout

- Haskell permite a omissão de “{ }” e “;”
- A regra de layout entra em vigor quando existe a omissão de chaves antes de:
 - **where**
 - **let**
 - **do**
 - **of**

Expressões e comandos: Regras de Layout

```
a = 10
```

```
b = 20
```

```
c = 30
```



```
a = 10
```

```
  b = 20
```

```
c = 30
```



```
a = 10
```

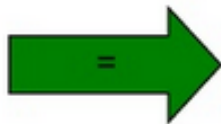
```
b = 20
```

```
c = 30
```



Expressões e comandos: Regras de Layout

```
a = b + c
  where
    b = 1
    c = 2
d = a * 2
```



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

Implícito

Explícito

Comandos

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

Comandos

- **Atribuição:** Não existe atribuição, e sim definição. Uma variável é uma coisa. O comando de definição é o sinal “ = ”.

ex.: `x = 2`

`y = “string”`

- **Iterativos:** Apresenta iterações com auxílio da recursão.

```
-- faz a soma do valor gasto pelo cliente
soma [] = 0
soma (primeiro:resto) = do
    let prim = read primeiro :: Float
    prim + soma resto
```

Funções

- As definições de funções em Haskell podem ser feitas por meio de uma sequencia de equações.
- A estrutura básica é:
 < nome da função > = < corpo >
- **x :: Int -> Int -> Float -> Bool -> Int**
 - “x” é o nome da função
 - o último tipo especificado identifica o tipo de dado a ser retornado
 - os quatro tipos do meio são argumentos da função

Funções

- Exemplo

```
300 --Funcao para retirar um elemento da lista, dado o numero do telefone
301 retiraDaLista :: [Char] -> [String] -> [String] -> [String]
302 retiraDaLista num listaAux lista = do
303     let (primeiro:resto) = listaAux
304         aux = splitOn ";" primeiro
305         teste = (" " ++ num) `elem` aux
306     if teste
307     then do
308         let novaLista = delete primeiro lista -- deleta a entrada da lista original
309             novaLista
310     else do
311         retiraDaLista num resto lista --Passa o numero do telefone, o resto da lista e a lista original
312     ...
```

Funções

- Do exemplo anterior temos exemplos de:
 - “**combinação por padrões**” - linha 303
 - uso de função de forma
 - **pré-fixada** - linha 304
 - **infixada** - linha 305
 - **concatenação de strings** - linha 305
 - e **recursão.** - linha 311

Funções

- Outro exemplo: Combinação por padrões

```
-- Função de verificação da existência dos arquivos de entrada
verifica :: [FilePath] -> IO ()
verifica [] = do
    putStrLn "Arquivos de entrada nao especificados. Use: -e <entregadores> -c <cardapio>"
    exitFailure --
verifica [e1] = do
    putStrLn "Arquivos de entrada nao especificados. Use: -e <entregadores> -c <cardapio>"
    exitFailure
verifica [e1,e2] = do
    putStrLn "Arquivos de entrada nao especificados. Use: -e <entregadores> -c <cardapio>"
    exitFailure
verifica [e1,e2,e3] = do
    putStrLn "Arquivos de entrada nao especificados. Use: -e <entregadores> -c <cardapio>"
    exitFailure
verifica [e1,e2,e3,e4] = do
    readFile e2
    readFile e4
    putStr ""
```

- Sem tal funcionalidade, talvez teríamos de fazer vários “if-else” para tratar a entrada

Funções

- **Uso de guards:** Guards são uma característica na definição de funções, que exploram a capacidade de se inserir condições que são utilizadas como alternativas para a função.

▪ A função max retorna o maior valor entre dois números.

```
max :: Int -> Int -> Int
```

```
max x y
```

```
  | x >= y      = x
```

```
  | otherwise   = y
```

se $x \geq y$ então retorne x

caso contrário, retorne y

guard.

Funções

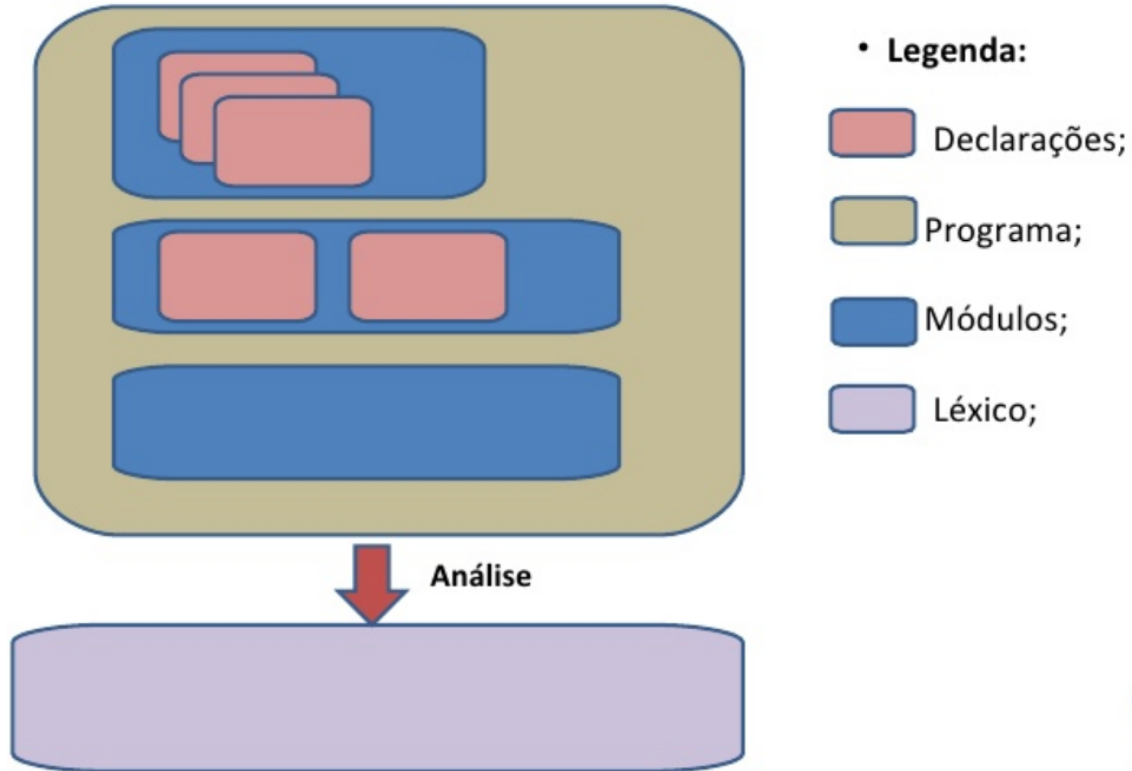
- **passagens de parâmetros:** Haskell é uma linguagem puramente funcional, o que significa que não há efeitos colaterais para as funções.
- Se uma função é chamada duas vezes com os mesmos parâmetros, o resultado retornado por ela será o mesmo.
- Os parâmetros são sempre **passados por cópia**. Qualquer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

Estrutura do Programa

A estrutura sintática e semântica abstrata em Haskell, assim como a forma como se relaciona pode ser dividida em 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);
- Expressões: Uma expressão denota um valor e tem um tipo estático. Haskell é composto de expressões;
- Estrutura Léxica: Capta a representação concreta dos programas Haskell em arquivos de texto.

Estrutura do Programa



Módulos

- Para usar um módulo, basta importá-lo usando a palavra reservada **import**

```
import System.Environment
import System.Directory
import System.IO
import Data.List
import Data.List.Split --Biblioteca para usar a função splitOn.
import System.IO.Error
import System.Exit
```

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. No caso ao lado, o nome do script é **Geometry.hs**

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side
```

Módulos: instalando

- 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**”
 - Para isso, no terminal digite:

sudo apt-get install cabal-install

- Em seguida, ainda no terminal digite:

cabal install nomeDoMódulo

- Na resolução do trabalho, utilizamos dois módulos não préinstalados no ghc, os quais foram o **split** (para usar a função `splitOn`) e o **exit** (para usar a função `exitFailure`).

```
18 import Data.List.Split
19 import System.IO.Error
20 import System.Exit
21
```

Classes

- São usadas em Haskell para realizar sobrecarga. Uma classe define uma ou mais funções que podem ser aplicadas para qualquer tipo que seja membro desta classe.
- Uma classe é análoga a uma interface em Java ou C#
- Uma classe deve ser declarada com um ou mais tipos de variáveis.

ex.: Class Eq a where

```
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    (/=) a b = not (a == b)
```

Orientação a objetos

- 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 normal de linguagens orientada a objeto.
- Porém, há algumas linguagens, variações de haskell, que são orientadas a objetos.
 - Haskell++
 - O'Haskell
 - Mondrian

Polimorfismo

- Em Haskell há o polimorfismo universal.
- A função possui um tipo genérico e a mesma definição é usada para vários tipos.
- Para conseguir uma função polimórfica, basta declarar e definir as funções com o mesmo nome, porém com argumentos próprios para cada objetivo.

ex.: `length :: String -> Int`
 `length :: [a] -> Int`

- No exemplo acima, a função `length` pode retornar tanto o tamanho de uma string quanto o de uma lista.

Verificação de Tipos

Tanto os compiladores quanto os interpretadores implementam o mecanismo de **checagem forte de tipo de dados**, devido a tamanha variedade de tipos.

Em Haskell, valores podem pertencer a um tipo de dado que pode ser:

- Monomórfico: apenas uma instância com mesmo nome.
- Polimórfico: mais de uma instância com mesmo nome.

Verificação de Tipos: Monomórficos

A checagem de tipos sem polimorfismo:

$f :: s \rightarrow t$ - Onde a função f deve ser aplicada sobre um argumento de tipo s e resultar numa expressão do tipo t .

O compilador/interpretador deve descobrir o tipo mais simples utilizado e comparar com a definição da própria função.

Verificação de Tipos: Polimórficos

`soma :: a -> b -> c`

`soma a b = a+b`

A função soma recebe dois argumentos que podem ser de tipos quaisquer.

Coerção

- Apresenta-se em Haskell.

Coerção Implícita:

```
ghci> 2 + 13.2  
15.2
```

Coerção Explícita:

```
ghci> (2 :: Float) + 10.0  
12.0
```

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"
```

Sobrecarga

- Sobrecarga sobre métodos: NÃO existe em Haskell.

incrementa :: Int -> Int

incrementa x = x + 1

incrementa :: String -> String

incrementa lista = lista ++ "italo"

Hugs> :load exemplo.hs ERROR file:exemplo.hs:19 -"incrementa" multiply definedHugs>

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

```
30 leituraArqEntrada :: IO [String]
31 leituraArqEntrada = do
32     entrada <- getArgs
33     verifica entrada `catch` handler
34     return entrada
--

    verifica [e1,e2,e3,e4] = do
        readFile e2 -- se um dos arquivos não existe, então é lançada uma exceção
        readFile e4 -- o qual é pega pela função handler definida abaixo
        putStr ""

handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e = do
        putStrLn $ "Nao foi possível processar os arquivos especificados:"
        exitFailure
    | otherwise = do
        ioError e
        exitFailure
```


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 (próx. slide).

Concorrência

```
import Control.Parallel

main = a `par` b `par` c `pseq` print (a + b + c)
  where
    a = ack 3 10
    b = fac 42
    c = fib 34

fac 0 = 1
fac n = n * fac (n-1)

ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))

fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Haskell no mundo



Banco - Amsterdam, Holanda



Banco - Alemanha



Japão



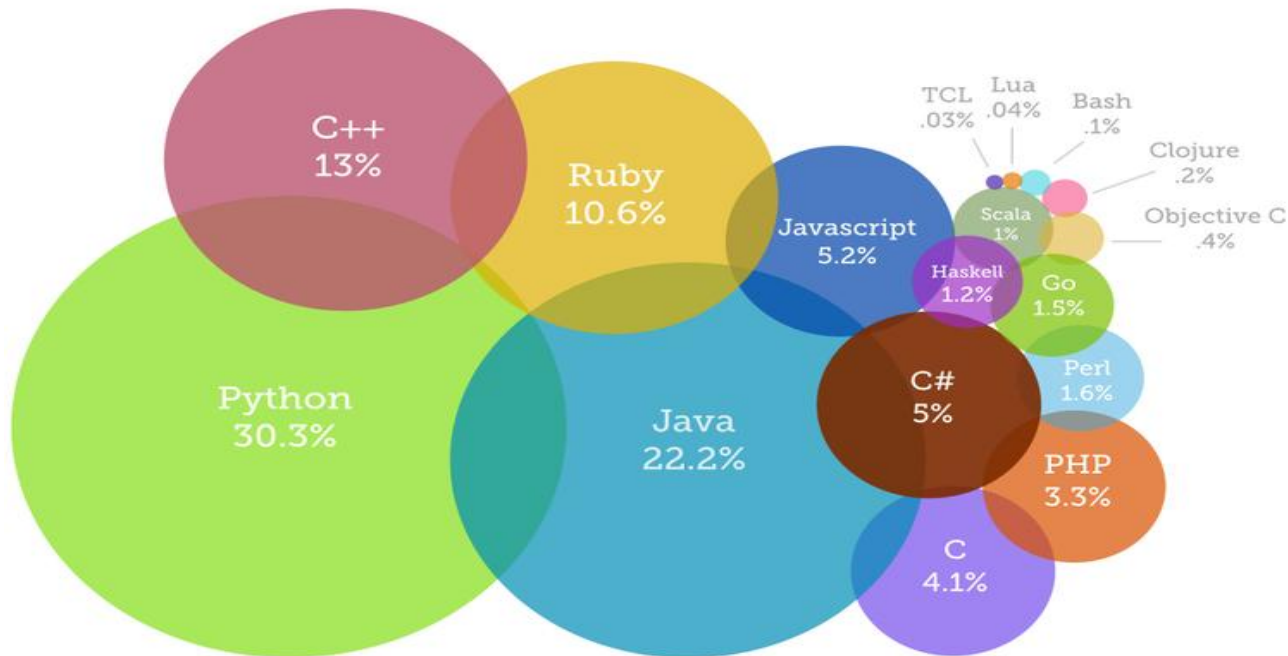
Telecomunicações - EUA

The Facebook logo, which is a solid blue rectangle with the word "facebook" in white lowercase letters and a registered trademark symbol.

facebook®

Haskell no mundo

Most Popular Coding Languages of 2014



Avaliação

- Alta flexibilidade;
- Extremamente tipada;
- Alta portabilidade;
- Grande reusabilidade;
- Dificuldade para quem está aprendendo programação funcional;
- Falta de comandos de repetição;
- Pouca interatividade com o usuário;

Referências

<http://learnyouahaskell.com/>

<http://pt.wikibooks.org/wiki/Haskell>

<http://claudiaboeres.pbworks.com/w/page/66262150/Programa%C3%A7%C3%A3o%20I%20-%202013>

<http://www.pergamum.udesc.br/dados-bu/000000/0000000000008/000008A3.pdf>

[http://www.haskell.org/haskellwiki/Haskell em 10 minutos#Sintaxe conveniente](http://www.haskell.org/haskellwiki/Haskell_em_10_minutos#Sintaxe_conveniente)

http://www.haskell.org/haskellwiki/GHC/Memory_Management