Curso de LaTeX- SEMAC 2012/2

Marcelo Garlet Millani

Universidade Federal do Rio Grande do Sul
Instituto de Informática
Grupo PET Computação

7 de maio de 2013

O Que É

- Linguagem puramente funcional (não permite efeitos colaterais)
- Possui um sistema de mônadas para isolar partes impuras do código
- Usa avaliação preguiçosa (só calcula uma expressão se ela for necessária)

História

- Proposta na conferência FPCA '87 (Functional Programming Languages and Computer Architecture)
- Havia diversas linguagens funcionais não-estritas, todas similares
- Foi criada com o objetivo de propor uma linguagem funcional padrão
- Nome em homenagem a Haskell Curry

História

A linguagem proposta deveria satisfazer o seguinte:

- Deveria ser adequada para o ensino, pesquisa e aplicações (incluindo sistemas grandes)
- Deveria ser descrita completamente através da publicação de sintaxe e semânticas formais
- Deveria estar disponível livremente, de forma que qualquer um pudesse implementar e distribuir a linguagem
- Deveria ser baseada em ideias amplamente aceitas
- Deveria reduzir diversidade desnecessária em linguagens funcionais

Interpretadores e Compiladores

Hugs Apenas um interpretador. É mais rápido que o interpretador do GHC;

- http://haskell.org/hugs
- Pacote hugs no linux

GHC Compilador e interpretador. É o mais usado dos três;

- http://haskell.org/ghc
- Pacote ghc no linux

NHC Apenas um compilador. Arquivos produzidos geralmente melhores que o do GHC.

-

Base

- Declarações começam com *let*

Sintaxe

let <identificador> [variáveis...] = <expressão>

Exemplos

- let a = 12
- let soma x y = x+y

Operadores

operador	operação
+	soma
-	subtração
*	multiplicação
/	divisão (float)
++	concatenação de listas
:	inserção em uma lista
!!	acesso a uma posição da lista
^	potenciação

Operadores Lógicos

operador	operação
&&	$\wedge$
||	$\vee$
not	$\neg$

operador	operação
==	igual
/=	diferente
>	maior que
>=	maior ou igual
<	menor que
<=	menor ou igual

Pattern Matching

Exemplo

```
Prelude> let a = (\(x,y) -> x+y)
Prelude> let b = (\(x:y:_) -> x*y)
Prelude> let (p,k) = (\x y -> (x,y)) 3 4
```

Declaração

Listas

```
Prelude> [1,2,3,4]
Prelude> []
Prelude> [2,4..16]
Prelude> [1,3..] !! 5
```

Tuplas

```
Prelude> ( 1 )
Prelude> (1,2,3)
Prelude> (1,"hello",[1,2])
Prelude> let tuple = (1 , (+) , 2)
```

Listas

Operações

```
Prelude> let duplica x = x*x
Prelude> map duplica [1,3..11]
Prelude> map duplica [1,3..] !! 7
Prelude> foldr (+) 0 [1,3,5,7]
Prelude> foldr (-) 0 [1,3,5,7]
Prelude> foldl (-) 0 [1,3,5,7]
Prelude> foldr (-) 2 [1,3,5,7]
Prelude> foldl (-) 2 [1,3,5,7]
Prelude> filter (>3) [1,3,5,7]
```

Folding

foldr

Associativo a direita

```
Prelude> foldr (-) 0 [1,2,3,4]
Prelude> 1 - (2 - (3 - (4 - 0)))
```

foldl

Associativo a esquerda

```
Prelude> foldl (-) 0 [1,2,3,4]
Prelude> (((0 - 1) - 2) - 3) - 4
```

Determinando Tipos

Exemplo

```
Prelude> :t 5
 Num a => a
Prelude> :t "Hello"
 "Hello" :: [Char]
Prelude> :t (1,'a')
 Num t => (t, Char)
Prelude> :t (\x y -> x+y)
 Num a => a -> a -> a
```

Data

- Definição de estruturas
- Muito útil com pattern matching

Definindo

```
Prelude> data Tree a = Leaf
         | Node (Tree a) a (Tree a)
```

Construindo

```
Prelude> let n2 = Node Leaf 10 Leaf
Prelude> let n1 = Node Leaf 6 Leaf
Prelude> let n0 = Node n1 7 n2
```

Funções Lambda

- Funções sem nome
- São avaliadas através de substituição de símbolos
- Usa-se redução β e α para fazer a avaliação

Exemplo

$$(\lambda x.\lambda y.x + y)\ 5\ 4$$

$$(\lambda y.5 + y)\ 4$$

$$5 + 4$$

Funções Lambda

Em Haskell

```
Prelude> (\x -> x*x) 5
Prelude> (\x y ->x+y) 4 5
```

Currying

- Avaliação parcial de funções
- Consequência direta do cálculo lambda

Exemplo

```
Prelude> let a = (\x y -> x+y)
Prelude> let b = a 5
```

Funções de Saída

- Não são funções de fato
- Permitem efeitos colaterais
- São consideradas "IO Actions"
- Devem ser executadas no ambiente *do* (ambiente padrão do *ghci*)

Definição

```
print :: Show a => a -> IO ()
putStr :: String -> IO ()
putStrLn :: String -> IO ()
```

Funções de Saída

printing

```
Prelude> putStrLn "Nova linha"
Prelude> putStr "Mesma linha"
Prelude> print 42
```

Funções de Entrada

Definição

```
readFile :: FilePath -> IO String
getLine :: IO String
```

```
Prelude> file <- readFile "arquivo"
Prelude> line <- getLine
```

Funções

```
Prelude> show 32
Prelude> read "32"::Int
Prelude> read "32"::Float
```

Arquivos

- Código pode ser dividido em diversos arquivos
- Permite a criação de módulos para facilitar reúso
- Usa-se a extensão .hs

Exemplo

```
Module Main where

main = do
     putStrLn "Hello World"
```

Definição de Funções

Formato

```
<identificador> [variáveis...] = <expressão>
```

Exemplo

```
fatorial 0 = 1
fatorial 1 = 1
fatorial n = n * fatorial (n-1)
```

Referências

 Hal Daumé III, *Yet Another Haskell Tutorial*