

# *Estruturas de Linguagem*

## *Interpretação de Programas* *(com programação funcional)*

**Francisco Sant'Anna**

`francisco@ime.uerj.br`

<http://github.com/fsantanna-uerj/EDL>

# Provas

- T3: 16/06 – Linguagem
- P2: 28/06 – Tudo
- PF: 05/07 – Tudo

# Programa em C

- Como executar?
- Como representar?
- Como interpretar?

```
#include <stdio.h>

int main()
{
    int c = 1;

    while (c <= 10) {
        printf("%d ", c);
        c++;
    }

    return 0;
}
```

bibliotecas

funções

variáveis

loops

chamadas

tipos

expressões



# Expressões

- Combinação de constantes, variáveis, operadores, etc, que pode ser avaliada (reduzida) a um valor.
  - $1 + 10$
  - $x * y$
  - $(x > 10) \ \&\& \ (x < 100)$
- Expressões envolvendo constantes e operações aritméticas:
- Como representá-las usando Haskell?

# Expressões Aritméticas

- Expressões envolvendo números e operações aritméticas
- Como representá-las em Haskell?
  - somente números inteiros, subtração, adição
  - sem variáveis
- Como avaliá-las em Haskell?
  - `avalía :: Exp -> Int`

```
data Exp = Num Int
         | Add Exp Exp
         | Sub Exp Exp
deriving Show

e1 = Num 10
e2 = Add e1 e1
e3 = Sub (Num 100) e2

main = print e3
```

```
...

avalía :: Exp -> Int
avalía (Num v)      = v
avalía (Add e1 e2) = (avalía e1) + (avalía e2)
avalía (Sub e1 e2) = (avalía e1) - (avalía e2)

main = print (avalía e3)
```

# Variáveis

```
data Exp = Num Int
         | Add Exp Exp
         | Sub Exp Exp
         | Var String
  deriving Show

-- 5 + i
e1 = Add (Num 5) (Var "i")

main = print e1
```

```
ambiente :: String -> Int
ambiente id = <...> -- retorna o valor de ID

avalialia :: Exp -> Int
avalialia (Num v)      = v
avalialia (Add e1 e2)  = (avalialia e1) + (avalialia e2)
avalialia (Sub e1 e2)  = (avalialia e1) - (avalialia e2)
avalialia (Var id)    = ambiente id

main = print (avalialia e1)
```



# Comandos (Statements)

- Unidade sintática que descreve uma ação em um programa imperativo
- Atribuição, Controle de Fluxo (sequência, condicional, repetição), Chamadas, etc
- Como representá-los em Haskell?

- **atribuição**
- **sequência**
- **condicional**
- **repetição**

```
data Cmd = Atr String Exp
          | Seq [Cmd]
          | Cnd Exp Cmd Cmd
          | Rep Exp Cmd
deriving Show

c1 = Seq [ Atr "x" (Num 10),
          Atr "y" (Var "x") ]

main = print c1
```



# Comandos (Statements)

- Como avaliá-los em Haskell?
  - `avalialCmd :: Cmd -> ???`
  - `avalialCmd :: Tela -> Cmd -> Tela`
  - `avalialCmd :: Teclado -> Tela -> Cmd -> Tela`
  - **`avalialCmd :: Amb -> Cmd -> Amb`**
- 2 questões
  - o que um programa faz efetivamente?
  - como manter o ambiente?

```
c1 = Seq [ Atr "x" (Num 10),  
          Atr "x" (Num 20),  
          Atr "y" (Var "x") ]  
  
main = avalialCmd c1
```

```
c2 = Atr "x"  
      (Add (Var "x") (Num 1))  
  
main = avalialCmd c2
```

# Comandos (Statements)

- Como manter o ambiente?

- `avalialCmd :: Amb -> Cmd -> Amb`

amb0 "x"

x = 10

amb1 "x"

x = 20

amb2 "x"

y = x

amb3 "x"

```
amb0 :: Amb
amb0 id = 0

prog = Seq [ Atr "x" (Num 10),
             Atr "x" (Num 20),
             Atr "y" (Var "x") ]

amb3 = avalialCmd amb0 prog
```

# Comandos (Statements)

- Como manter o ambiente?
  - `avaliaCmd :: Amb -> Cmd -> Amb`

```
type Amb = String -> Int

data Cmd = Atr String Exp
          | Seq Cmd Cmd

avaliaCmd :: Amb -> Cmd -> Amb
avaliaCmd amb (Atr id exp) =
    (\x -> if x==id then v
          else amb id)
    where v = avaliaExp amb exp
avaliaCmd amb (Seq c1 c2) =
    avaliaCmd amb' c2
    where amb' = avaliaCmd amb c1

avaliaExp :: Amb -> Exp -> Int
...
avaliaExp amb (Var id) = amb id
```

# Comandos (Statements)

- Condicional

```
type Amb = String -> Int

data Cmd = Atr String Exp
          | Seq Cmd Cmd
          | Cnd Exp Cmd Cmd

avaliaExp :: Amb -> Exp -> Int
...

avaliaCmd :: Amb -> Cmd -> Amb
avaliaCmd amb (Atr id exp)      = ...
avaliaCmd amb (Seq c1 c2)      = avaliaCmd amb' c2
                                where amb' = avaliaCmd amb c1
avaliaCmd amb (Cnd exp c1 c2) =
                                if (avaliaExp amb exp) /= 0 then
                                    avaliaCmd amb c1
                                else
                                    avaliaCmd amb c2
```



# Verificação Estática

- Nem todo programa com sintaxe ok é válido:
  - `"ola" + 10;` // tipos incompatíveis
  - `x = a + b;` // var não declarada
  - `strlen(s, 10);` // param inexistente
- Linguagem com declaração de variáveis:
  - `verificaCmd :: Cmd -> Bool`
  - `verificaCmd :: [String] -> Cmd -> ([String], Bool)`

```
data Cmd = Atr String Exp  
         | Seq [Cmd]  
         | Dcl String
```

```
c1 = Atr "x" (Num 1)  
c2 = Seq (Dcl "x") c1
```