Estruturas de Linguagem

Interpretação de Programas (com programação funcional)

Francisco Sant'Anna

francisco@ime.uerj.br

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

Programa em C

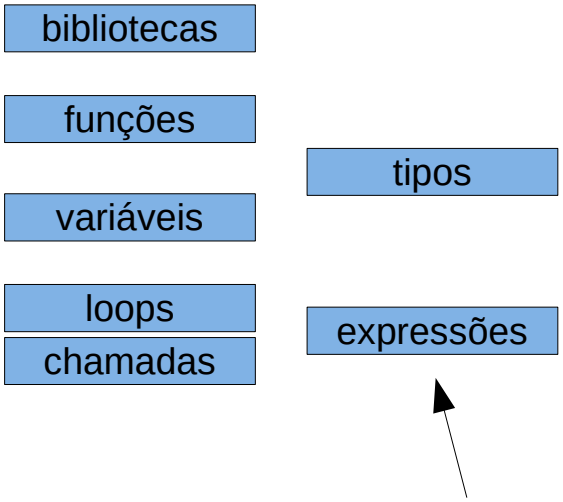
- Como executar?
- Como representar?
- Como interpretar?

```
#include <stdio.h>

int main()
{
   int c = 1;

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

   return 0;
}
```



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?
  - `avalia :: 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
```

```
...

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

main = print (avalia 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
```

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

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

main = print (avalia 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 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?
  - `avaliaCmd :: Cmd -> ???`
  - `avaliaCmd :: Tela -> Cmd -> Tela`
  - `avaliaCmd :: Teclado -> Tela -> Cmd -> Tela`
  - **`avaliaCmd :: Mem -> Cmd -> Mem`**
- 2 questões
  - o que um programa faz efetivamente?
  - como manter a memória (ambiente)?

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

main = avaliaCmd ??? c1 -> ???
```

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

main = avaliaCmd ??? c2 -> ???
```

Memória (Ambiente)

- `type Mem = [(String,Int)]`
  - Associa um identificador a um valor inteiro
  - O valor mais recente é adicionado no início

```
Cmd        Mem

           []

x = 10

           [(“x”,10)]

x = 20

           [(“x”,20),(“x”,10)]

y = x

           [(“y”,20),(“x”,20),(“x”,10)]
```

```
type Mem = [(String,Int)]

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

mem = avaliaCmd [] prog
```

Memória (Ambiente)

- Como manipular a memória?
  - **consulta :: Mem -> String -> Int**
  - **escreve :: Mem -> String -> Int -> Mem**

```
type Mem = [(String,Int)]

consulta :: Mem -> String -> Int
consulta []                id = 0
consulta ((id',v'):l)      id = if id == id' then
                                  v'
                                else
                                  consulta l id

escreve :: Mem -> String -> Int -> Mem
escreve mem id v = (id,v):mem
```

Comandos (Statements)

- Como manter a memória?
  - **avaliaCmd :: Mem -> Cmd -> Mem**

```
type Mem = [(String,Int)]
consulta :: Mem -> String -> Int
escreve  :: Mem -> String -> Int -> Mem

data Cmd = Atr String Exp
         | Seq Cmd Cmd

avaliaCmd :: Mem -> Cmd -> Mem
avaliaCmd mem (Atr id exp) =
                              escreve mem id v where
                                v = avaliaExp mem exp
avaliaCmd mem (Seq c1 c2)  =
                              avaliaCmd mem' c2 where
                                mem' = avaliaCmd mem c1

avaliaExp :: Mem -> Exp -> Int
...
avaliaExp mem (Var id) = consulta mem id
```

Comandos (Statements)

- Condicional

```
type Mem = [(String,Int)]

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

avaliaExp :: Mem -> Exp -> Int
...

avaliaCmd :: Mem -> Cmd -> Mem
...
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`

Verificação Estática

- 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
```