

# *Estruturas de Linguagem*

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

### ***Funções***

**Francisco Sant'Anna**

**francisco@ime.uerj.br**

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

# 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
- funções

```
data Cmd = Atr String Exp  
          | Seq Cmd Cmd  
          | Cnd Exp Cmd Cmd  
          | Rep Exp Cmd  
          | Fun String Cmd
```

```
data Exp = Num Int  
          | Add Exp Exp  
          | Sub Exp Exp  
          | Var String  
          | App String Exp
```

# Funções

```
def duplica (v):  
    return v + v  
  
print (duplica(10)) # 20
```

```
Seq  
  (Fun "duplica"  
    (Atr "ret"  
      (Add (Var "arg")  
            (Var "arg"))))  
    (Prt (App "duplica" (Num 10))))
```

```
def soma (v):  
    if v != 0:  
        return v + soma(v-1)  
    else:  
        return 0  
  
print (soma(10)) # 55
```

```
Seq  
  (Fun "soma"  
    (Cnd (Var "arg")  
      (Atr "ret"  
        (Add (Var "arg")  
              (App "soma"  
                    (Sub (Var "arg")  
                          (Num 1))))))  
      (Atr "ret" (Num 0))))  
    (Prt (App "soma" (Num 10))))
```

# Funções

- Como relacionar (App “soma” ...) com  
(Fun “soma” ...) ?

- Parecem variáveis (Var “x”) vs  
(Attr “x” ...)

```
data Cmd = Attr String Exp
         | Seq Cmd Cmd
         | Cnd Exp Cmd Cmd
         | Rep Exp Cmd
         | Fun String Cmd
```

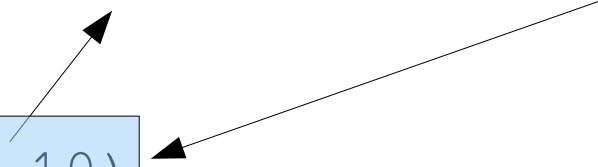
- ... mas não são!
- Attr guarda Exp, enquanto que Fun guarda Cmd
- Precisamos de uma “memória” para comandos!
- avalíaExp e avalíaCmd agora devem manipular os dois tipos de memória

# Argumentos / Retornos

- Parâmetro é passado como “arg” na memória de variáveis.
- Retorno é guardado como “ret” na memória de variáveis.

- `Fun “f” ... (Var “arg”) ... (Atr “ret” (Num 1))`

- `App “f” (Num 10)`



# Ambiente

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

escreve :: [(String, a)] -> String -> a -> [(String, a)]
escreve l id v = (id, v) : l

consulta :: [(String, a)] -> String -> a
consulta [] id = undefined
consulta ((id', v') : l) id = if id == id' then
    v'
    else
        consulta l id

type Env = (Mem, Cod)

avaluaExp :: Env -> Exp -> Int
avaluaCmd :: Env -> Cmd -> Env
```

# avaliaExp e avaliaCmd

```
avaliaExp :: Env -> Exp -> Int
```

```
avaliaExp _ (Num v) = v
avaliaExp env (Add e1 e2) = (avaliaExp env e1) + (avaliaExp env e2)
avaliaExp env (Sub e1 e2) = (avaliaExp env e1) - (avaliaExp env e2)
avaliaExp (mem, _) (Var id) = consulta mem id
```

```
avaliaExp (mem, cod) (App id e) = ret where
    ret = consulta mem' "ret"
    (mem', _) = avaliaCmd (mem', cod) fun
    mem' = escreve mem "arg" arg
    arg = avaliaExp (mem, cod) e
    fun = consulta cod id
```

```
avaliaCmd :: Env -> Cmd -> Env
```

```
avaliaCmd (mem, cod) (Atr id exp) = (escreve mem id v, cod) where
    v = avaliaExp (mem, cod) exp
avaliaCmd env (Seq c1 c2) = avaliaCmd env' c2 where
    env' = avaliaCmd env c1
avaliaCmd env (Cnd exp c1 c2) = if (avaliaExp env exp) /= 0 then
    avaliaCmd env c1
    else
    avaliaCmd env c2
avaliaCmd (mem, cod) (Fun id c) = (mem, escreve cod id c)
```