

# Especificação interpretador em Haskell

Esta é a especificação de um interpretador para uma linguagem imperativa com expressões definidas por constante, variável, operadores aritméticos e um operador relacional. Os comandos da linguagem são atribuição, sequência, condicional, iteração, declaração de variável e impressão.

O processo de construção pode ser dividido em duas partes como tradicionalmente se trabalha com interpretadores e compiladores: front-end e back-end. O front-end será constituído por um parser. Como elemento intermediário é necessário definir a estrutura da árvore sintática abstrata. Vamos considerá-lo como parte do back-end, uma vez que toda a definição semântica terá a AST como ponto de partida.

## 1 Passos para o back-end

- Declare a AST (expressões e comandos)

```
data Exp =  Constant Int
          |  Variable String
          |  Minus Exp Exp
          |  Greater Exp Exp
          |  Times Exp Exp
deriving Show

data Com =  Assign String Exp
          |  Seq Com Com
          |  Cond Exp Com Com
          |  While Exp Com
          |  Declare String Exp Com
          |  Print Exp
deriving Show
```

- Declare as estruturas de dados do ambiente e algumas funções para manipulá-lo. Ao darmos a semântica de linguagens imperativas, costumamos utilizar um ambiente para armazenar, por exemplo, os valores

associados a variáveis. O conjunto de valores das variáveis será mantido em uma lista de inteiros. A posição de uma variável (*location*) na lista vai ser definida pela posição na lista. Esta é a razão de ter a posição como sinônimo para inteiro. A tabela de símbolos é chamada de "Index" e é declarada como uma lista de String.

```
type Location = Int
type Index = [String]
type Stack = [Int]
```

Outra alternativa para estabelecer a relação entre variáveis e valores seria lidar, por exemplo, com uma lista [(String, Value)], em que o tipo Value é sinônimo para Int.

Para manipular o ambiente, considere as seguintes funções

- Lembrando que a posição é um inteiro, temos

```
position :: String -> Index -> Location
position name index = let
    pos n (nm:nms) = if name == nm
                      then n
                      else pos (n+1) nms
    in pos 1 index
```

- Obter o enésimo valor (específico de uma posição)

```
fetch :: Location -> Stack -> Int
fetch n (v:vs) = if n == 1 then v else fetch (n-1) vs
```

- Uma função para computar um ambiente atualizado. Obtém-se uma nova pilha baseada no número da posição atualizada, do valor armazenado e do conteúdo anterior da pilha.

```
put :: Location -> Int -> Stack -> Stack
```

- Selecione uma mônada para ser usada como modelo para cálculo, em vez de uma máquina virtual. Por exemplo, o tipo

```
newtype M a = StOut (Stack -> (a, Stack, String))
```

possui o construtor StOut para representar um mônada *State e Output*, mapeando uma Stack em uma tripla ((a, Stack, String)).

Semelhante à função parse que vimos na aula sobre Parsing, podemos descrever a função

`unStOut (StOut f) = f`

A partir disso, defina o tipo `M` como instância de **Monad**. O resultado de uma computação é realmente a primeira parte da tupla `(a, Stack, String)`. Mas, há também a pilha e uma String.

Algumas funções sobre valores monádicos

- para retornar o valor de um ambiente como principal resultado de uma computação

`getfrom :: Location -> M Int`

- para modificação da pilha

`write :: Location -> Int -> M ()`

- para modificar a pilha, sem modificação do índice, para colocar um valor no topo da pilha. Pode ser útil ao declarar uma variável, por exemplo.

`push :: Int -> M ()`

- Escrever a expressão de avaliação que, dada uma expressão e uma tabela (índice), retorna um valor monádico com um inteiro.

```
eval1 :: Exp -> Index -> M Int
eval1 exp index = case exp of
  Constant n -> return n
  Variable x -> let loc = position x index
                 in getfrom loc
...
```

- Escrever a função para execução de comandos que recebe como argumento um comando e uma tabela (índice), retornando um valor monádico com unit (tupla vazia) como resultado.

```
exec :: Com -> Index -> M ()
exec stmt index = case stmt of
  Assign name e -> let loc = position name index
                   in do { v <- eval1 e index ;
                          write loc v }
  Seq s1 s2 -> do { x <- exec s1 index ;
                   y <- exec s2 index ;
                   return () }
```

```

...
  Declare nm e stmt -> do { v <- eval1 e index ;
                           push v ;
                           exec stmt (nm:index) ;
                           pop }
...

```

- A função **output** é utilizada na execução do comando **Print**.

```

output :: Show a => a -> M ()
output v = StOut (\n -> (( , n, show v))

```

## 2 Passos para o front-end

- Preparar a gramática da linguagem. Pode-se começar pela expressões

```

⟨rexp⟩ ::= ⟨rexp⟩ ⟨relop⟩ ⟨expr⟩ | ⟨expr⟩
⟨expr⟩ ::= ⟨expr⟩ ⟨addop⟩ ⟨term⟩ | ⟨term⟩
⟨term⟩ ::= ⟨term⟩ ⟨mulop⟩ ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ ::= ⟨var⟩ | ⟨digiti⟩ | (⟨expr⟩)
⟨var⟩ ::= ⟨Identifier⟩
⟨digiti⟩ ::= ⟨digit⟩ | ⟨digiti⟩ ⟨digiti⟩
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨addop⟩ ::= + | -
⟨mulop⟩ ::= * | /
⟨relop⟩ ::= > | < | =

```

A gramática da linguagem também deve incluir comandos

```

⟨com⟩ ::= ⟨assign⟩ | ⟨seqv⟩ | ⟨cond⟩ | ⟨while⟩ | ⟨declare⟩ |
⟨printe⟩
⟨assign⟩ ::= ⟨identif⟩ := ⟨rexp⟩
⟨seqv⟩ ::= { ⟨com⟩ ; ⟨com⟩ }
⟨cond⟩ ::= if ⟨rexp⟩ then ⟨com⟩ else ⟨com⟩
⟨while⟩ ::= while ⟨rexp⟩ do ⟨com⟩
⟨declare⟩ ::= declare ⟨identif⟩ = ⟨rexp⟩ in ⟨com⟩
⟨printe⟩ ::= print ⟨rexp⟩

```

Se necessário, modificar a estrutura de dados no back-end para lidar com toda a linguagem.

- Escolher uma biblioteca com um combinador de parser. Pode ser "parsec"(<http://hackage.haskell.org/package/parsec>) ou mesmo pode-se escrever um. De qualquer forma, deve-se levar em consideração o que vimos sobre parsing. De forma semelhante, haverá parsers para as diversas construções da linguagem.
- Os parsers simples devem ser combinados a fim de obter o parser para a linguagem completa.
- Último passo: combinar front-end e back-end

### 3 Exemplo

Considere o seguinte programa

```
declare x = 150 in
  declare y = 200 in
    { while x > 0 do { x := x - 1; y := y - 1 };
    }
```

A AST para este programa será a seguinte:

```
s1 = Declare "x" (Constant 150)
      (Declare "y" (Constant 200)
        (Seq (While (Greater (Variable "x") (Constant 0))
                     (Seq (Assign "x" (Minus (Variable "x")
                                              (Constant 1))
                               )
                         (Assign "y" (Minus (Variable "y")
                                              (Constant 1))
                               )
                     )
        )
      )
      (Print (Variable "y"))
    )
```

Um função para utilizar o interpretador pode ser definida da seguinte forma

```
interp a = unStOut (exec a []) []
```

Por exemplo, a execução do comando `s1` seria dada da seguinte forma e com o resultado abaixo

```
Main> interp s1  
(((), [] , "50")
```