

Compiladores — Folha laboratorial 1

Pedro Vasconcelos, DCC/FCUP

Setembro 2020

Interpretador para programas sequenciais

Pretende-se escrever um interpretador em Haskell para programas sequenciais com atribuições e expressões aritméticas.

Os programas são constituídos por *expressões* e *comandos* (“*statements*”). As expressões podem ser números, variáveis e operações aritméticas sobre inteiros. Os comandos podem ser atribuições, incrementos ou sequências de outros comandos. Podemos ainda executar comandos no meio de expressões de forma semelhante ao “operador vírgula” na linguagem C.

Dois exemplos (usando sintaxe concreta da linguagem C e com os valores finais das variáveis anotados):

```
// Exemplo 1
a = 3; b = 2; a = a * b;    // a: 6, b: 2
// Exemplo 2
a = 1; b = (a++, a*2);      // a: 2, b: 4
```

Estes programas não têm decisões, ciclos nem funções; logo terminam sempre (eventualmente com um erro de execução como uma divisão por zero).

Sintaxe abstrata

Para facilitar o processamento é conveniente converter a sintaxe concreta acima numa *árvore de sintaxe abstrata* (AST); esta conversão é efetuada pelos analisadores lexical e sintático (*lexer* e *parser*) que vamos estudar mais tarde.

Nos exercícios seguintes vamos assumir que temos já programas em sintaxe abstrata, ou seja, como valores de dois tipos algébricos em para comandos `Stm` e expressões `Exp`. As declarações em Haskell são:

```
type Ident = String                -- identificadores

data BinOp = Plus | Minus | Times | Div -- operadores aritméticos

-- comandos
data Stm = AssignStm Ident Exp      -- var = exp
        | IncrStm Ident            -- var ++
        | CompoundStm Stm Stm      -- stm1; stm2
```

```
-- expressões
data Exp = IdExp Ident      -- x, y, z ..
        | NumExp Int        -- -1, 0, 1, 2, ...
        | OpExp Exp BinOp Exp -- exp1+exp2, exp1*exp2, ...
        | SeqExp Stm Exp     -- (stm, exp)
```

O exemplo 1 acima pode ser representado da seguinte forma:

```
exemplo1 = CompoundStm
          (AssignStm "a" (NumExp 3))
          (CompoundStm
            (AssignStm "b" (NumExp 2))
            (AssignStm "a" (OpExp (IdExp "a") Times (IdExp "b"))))
          )
```

O esqueleto do código fornecido num repositório Git que contém três módulos Haskell:

Interpreter.hs definições dos tipos da sintaxe abstrata e declarações das funções pedidas (para completar).

Tests.hs definições de alguns de casos de teste para as funções pedidas.

Main.hs módulo principal que importa os outros e executa todos os testes.

Além deste módulos tem também um ficheiro **cabal** para automatizar a compilação e execução dos testes:

```
$ cabal build
$ cabal run
```

Inicialmente todos os testes falham porque falta implementar as funções necessárias (exercícios seguintes).

Pode também executar o interpretador GHCi (*read-eval-print-loop*) para testar as suas funções de forma interactiva:

```
$ cabal repl
...
Interpreter> interpExp (OpExp (NumExp 1) Plus (NumExp 2)) []
(3, [])
```

Exercício 1: Listar identificadores

Escreva duas funções recursivas para listar todos os identificadores de comandos e expressões:

```
idsStm :: Stm -> [Ident]
idsExp :: Exp -> [Ident]
```

Note que as duas funções devem chamar-se mutuamente porque comandos podem conter expressões e as expressões podem conter comandos.

Investigue o que acontece se um identificador ocorrer mais do que uma vez; como poderia garantir que os resultado não têm repetidos?

Exercício 2: Interpretador funcional

Escreva duas funções mutuamente recursivas para interpretar comandos e expressões:

```
interpStm :: Stm -> Table -> Table
interpExp :: Exp -> Table -> (Int, Table)
```

Vamos representar uma *tabela de associações* de variáveis a inteiros como listas de pares. Por exemplo, a lista `[("x", 2), ("y", 0)]` associa $x \mapsto 2$ e $y \mapsto 0$.

A função `interpStm` recebe um comando e tabela e retorna a tabela modificada; exemplo:

```
> interpStm example []
[("a",6), ("b",2)]
```

A função `interpExp` recebe uma expressão e uma tabela e retorna um *par*: o valor inteiro da expressão e a uma nova tabela (porque a expressão poderá eventualmente modificar variáveis).

```
> interpExp (OpExp (Var "a") Plus (Num 1)) [("a",2)]
(3, [("a",2)])
```

(Neste caso a tabela ficou inalterada porque a expressão não modifica variáveis.)

Sugestões: use a função `lookup` do prelúdio para procurar o valor (se existir) associado a um identificador. Deve ainda definir uma função auxiliar `update` para construir uma tabela com o valor de um identificador modificado.

Exercício 3: Extras

Modifique a definição do tipo `Table` e do interpretador de forma usar uma estrutura de dados mais eficiente para associar valores às variáveis, por exemplo, `Data.Map` ou `Data.HashMap`.

Os módulos `Map` e `HashMap` exportam algumas operações com os mesmos nomes idênticos aos do prelúdio (e.g. `lookup`). Para evitar conflitos, deve importar usando nomes qualificados:

```
import qualified Data.Map as Map
-- usar Map.empty, Map.lookup, Map.insert, etc.
```

Note ainda que se mudar o tipo `Table` necessita de adaptar também os testes. Por exemplo, o teste do exemplo 1 pode ficar:

```
prop_interp_example1 =
  interStm example1 [] == Map.fromList [("a",8),("b",6)]
```

A ordem dos pares num `Map` não é significativa, logo não é necessário ordenar o resultado.

Mais documentação:

- <http://hackage.haskell.org/package/containers>
- <http://hackage.haskell.org/package/unordered-containers>