Faculdade de Engenharia da Universidade do Porto

Haskell Low-Level Machine

Trabalho Prático 2

Programação Funcional e Lógica

Turma 8 – Grupo 13

Joana Cerqueira Maia – up202108835 – 50%

Matilde Faro Martins Castelo Pires – up202108853 – 50%

Part 1

On the first part we started by defining the types and data. It was not difficult to decide the types of the machine's stack and state but it wasn't obvious that we had to create a new data definition, Param, for variables because the stack can contain not only integers but also booleans.

```haskell
data Param =
  Int Integer | TT | FF
  deriving (Show, Eq)
```

```haskell
type Code = [Inst]
type Stack = [Param]
type State = [(String, Param)]
```

When it comes to the functions, the most importante one was the run function, which runs a list of instructions and returns an empty code list, a stack and the output values in the storage. We decide to use the case method for better visual experience and a recursive method. Because we had to deal with errors, we used if statements so it would be more organize.

```haskell
-- interpretador que executa uma lista de instruções
run :: (Code, Stack, State) -> (Code, Stack, State)
run ([], stack , state) = ([], stack , state)
run (code:next, stack, state) =
  case code of
```

The other functions were easier to implement but we just realized that we needed to have a where clause on the conversion functions when the variables TT and FF were not being converted to True and False on the final output.

```haskell
-- conversão da stack para uma string
stack2Str :: Stack -> String
stack2Str = intercalate "," . map stackToString
  where
    stackToString FF = "False"
    stackToString TT = "True"
    stackToString (Int n) = show n
```

```haskell
-- conversão do estado da máquina para string
state2Str :: State -> String
state2Str = intercalate "," . map(\(string, param) -> string ++ "=" ++ stateToString param) . sortOn fst
  where
    stateToString FF = "False"
    stateToString TT = "True"
    stateToString (Int n) = show n
```

Parte 2

On the last part we also started by defining the data. We analysed each instruction in order to choose the right data definition for them between data Aexp or data Bexp. Regarding the data Stm, we chose the paramenters through the explanation of the assignement of part 2. One important aspect is that during the making of this part we added more things to the data definitions whenever we needed more.

```
data Aexp =
    IntExp Integer| VarExp String | AddExp Aexp Aexp | MulExp Aexp Aexp | SubExp Aexp Aexp | EmptyAexp
    deriving (Show, Eq)

data Bexp =
    TrueExp | FalseExp | EquExp Aexp Aexp | EqExp Bexp Bexp | LeExp Aexp Aexp | NegExp Bexp | AndExp Bexp Bexp | EmptyBexp
    deriving (Show, Eq)

data Stm =
    IfExp Bexp Stm Stm| WhileExp Bexp Stm | SeqExp Stm Stm | AssignExp String Aexp
    deriving(Show, Eq)
```

For the lexer function we created a new data, such as the slides of the class, called Token.

```
data Token =
  PlusTok | MinusTok | TimesTok | OpenTok | CloseTok | IntTok Integer | VarTok String | LessTok | EqualTok
  | DotTok | CommaTok | EqTok | NotTok | AndTok | IfTok | ThenTok | ElseTok | WhileTok | DoTok | TrueTok | FalseTok | AssignTok
  deriving (Show, Eq)
```

First we made the compiler by dividing it into 3 auxiliar functions: compA, compiler for aritmetic expressions ; compB, compiler for boolean expressions ; compStm, compiler for statements. Then we use the compile function to concatenate the compStm function that will also call the other auxiliar functions.

```
  -- compilador auxiliar para expressões aritméticas
compA :: Aexp -> Code
compA (IntExp x) = [Push x]
  compA (VarExp s) = [Fetch s]
  compA (AddExp a1 a2) = compA a1 ++ compA a2 ++ [Add]
  compA (MulExp a1 a2) = compA a1 ++ compA a2 ++ [Mult]
  compA (SubExp a1 a2) = compA a1 ++ compA a2 ++ [Sub]

  -- compilador auxiliar para expressões booleanas
compB :: Bexp -> Code
compB TrueExp = [Tru]
  compB FalseExp = [Fals]
  compB (EquExp a1 a2) = compA a1 ++ compA a2 ++ [Equ]
  compB (EqExp b1 b2) = compB b1 ++ compB b2 ++ [Equ]
  compB (LeExp a1 a2) = compA a1 ++ compA a2 ++ [Le]
  compB (AndExp b1 b2) = compB b1 ++ compB b2 ++ [And]
  compB (NegExp b) = compB b ++ [Neg]

  -- compilador para statements
compStm :: Stm -> Code
compStm (IfExp b s1 s2) = compB b ++ [Branch (compStm s1) (compStm s2)]
  compStm (WhileExp b s) = [Loop (compB b) (compStm s)]
  compStm (SeqExp s1 s2) = compStm s1 ++ compStm s2
  compStm (AssignExp s a) = compA a ++ [Store s]

  -- compilador principal
compile :: Program -> Code
  compile = concatMap compStm
```

Finally we tried to make the most difficult function for us that was the parse function.

First, we created the lexer that converts a string into a list of tokens based on the lexer from the slides.

```
-- converte a string numa lista de strings
lexer :: String -> [Token]
lexer [] = []
lexer ('+' : xs) = PlusTok : lexer xs
lexer ('-' : xs) = MinusTok : lexer xs
lexer ('*' : xs) = TimesTok : lexer xs
lexer ('(' : xs) = OpenTok : lexer xs
lexer (')' : xs) = CloseTok : lexer xs
lexer ('<' : '=' : xs) = LessTok : lexer xs
lexer ('=' : '=' : xs) = EqualTok : lexer xs
lexer (':' : '=' : xs) = DotTok : lexer xs
lexer (';': xs) = CommaTok : lexer xs
lexer ('=' : xs) = EqTok : lexer xs
lexer ('w': 'h': 'i': 'l': 'e': xs) = WhileTok : lexer xs
lexer ('d': 'o': xs) = DoTok : lexer xs
lexer ('i': 'f': xs) = IfTok : lexer xs
lexer ('t': 'h': 'e': 'n': xs) = ThenTok : lexer xs
lexer ('e': 'l': 's': 'e': xs) = ElseTok : lexer xs
lexer ('a': 'n': 'd': xs) = AndTok : lexer xs
lexer ('n': 'o': 't': xs) = NotTok : lexer xs
lexer ('T': 'r': 'u': 'e': xs) = TrueTok : lexer xs
lexer ('F': 'a': 'l': 's' : 'e': xs) = FalseTok : lexer xs
lexer (x : xs) | isSpace x = lexer xs
lexer s@(c : xs)
  | isDigit c = IntTok (read n) : lexer ns
  where
    (n, ns) = span isDigit s
lexer s@(c : xs)
  | isLetter c = VarTok v : lexer vs
  where
    (v, vs) = span isLetter s
```

Then we created many auxiliar functions for the parse of aritmetic and boolean expressions that were also based on the ones from the slides.

Last but no leaste, we finish by creating another auxiliar parse function for statements (based on the same logistics as the ones before) and calling it on the main parse function.

Note: We couldn’t test if the parse function works because we have an error on the calling of the parseLoop function, that converts the output of the statement parse to a list of statements so that the compiler can work.