PFL Project 2

Grupo	Nome	Número	Contribuição
T06_G07	Rúben Filipe Vaz Fonseca	202108830	50%
	Miguel Nogueira Marinho	202108822	50%

Instalação e execução

Para executar o programa, basta seguir os seguintes passos:

1. Abrir o interpretador de Haskell.

2. Carregar o ficheiro main.hs, presente no ficheiro src.

3. Chamar cada teste individualmente para ver os resultados, ou chamar a função run ou Main.parse para testar a parte 1 e a parte 2, respectivamente.

Aviso: Para correr a função parse, é preciso adicionar o prefixo Main. para especificar que a função a ser chamada é a da Main e não a do Parsec. Exemplo: Main.parse “x := 1;”

Parte 1

Nesta parte, é nos pedido para implementar uma máquina de baixo nível com configurações com a forma (*c*,*e*,*s*) em que *c* é a lista de instruções, *e* uma pilha para avaliar expressões aritméticas e booleanas e *s* uma “storage”.

Inicialmente, estão definidas as instruções da máquina de acordo com o enunciado do trabalho:

```
data Inst =
  Push Integer | Add | Mult | Sub | Tru | Fals | Equ | Le | And | Neg | Fetch String | Store String | Noop |
  Branch Code Code | Loop Code Code
  deriving Show
type Code = [Inst]
```

Em seguida, temos a implementação de uma pilha e algumas funções para o uso desta, assim como a implementação da “storage”:

```haskell
-- State --
data State = State [(String,String)]
    deriving Show

emptyState :: Main.State
emptyState = Main.State []

addState :: Main.State -> (String, String) -> Main.State
addState (Main.State state) (key, value) =
  Main.State $ sort $ (key, value) : filter (\(k, _) -> k /= key) state

isEmptyState :: Main.State -> Bool
isEmptyState (Main.State [])= True
isEmptyState (Main.State _) = False
```

```haskell
-- Stack --
data Stack = Stk [String]
    deriving Show

push :: String -> Stack -> Stack
push x (Stk xs) = Stk (x:xs)

pop :: Stack -> Stack
pop (Stk (_:xs)) = Stk xs

top :: Stack -> String
top (Stk (x:_)) = x

emptyStk :: Stack
emptyStk = Stk []

isEmpty :: Stack -> Bool
isEmpty (Stk [])= True
isEmpty (Stk _) = False
```

Em seguida, temos a implementação da criação de uma pilha e uma “storage” vazia, assim como funções de tradução da pilha e “storage” para string, de acordo com a apresentação pretendida. Na nossa implementação, decidimos representar os valores booleanos na pilha True e False como “tt” e “ff”, respetivamente. Já na “storage”, cada

elemento é representado da forma (key=value) como se verifica em state2str.

```
createEmptyStack :: Stack
createEmptyStack = emptyStk

checkStr :: String -> String
checkStr s = if s == "ff" then "False" else if s == "tt" then "True" else s

stack2Str :: Stack -> String
stack2Str stk = if isEmpty stk then "" else if isEmpty (pop stk) then checkStr (top stk) else checkStr (top stk) ++ "," ++ stack2Str (pop stk)

createEmptyState :: Main.State
createEmptyState = emptyState

state2Str :: Main.State -> String
state2Str (Main.State pairs) =
  if null pairs
    then ""
    else init (concatMap pairToStr pairs)
  where
    pairToStr (key, value) = key ++ "=" ++ checkStr value ++ ","
```

Em seguida, temos a função run que consiste num interpretador que recebe uma lista de instruções Code, uma pilha, inicialmente vazia, e uma “storage”. Nesta implementação, vai-se percorrendo a lista Code até que não exista mais nenhuma instrução ou que seja feita alguma má configuração, sendo assinalado erro. A função apresenta várias opções de possíveis instruções e verifica em cada elemento da lista Code qual a instrução que está presente e em seguida verifica se esta é válida e caso seja, realiza a alteração na pilha e na “storage”.

```
-- run :: (Code, Stack, State) -> (Code, Stack, State)
run :: (Code, Stack, Main.State) -> (Code, Stack, Main.State)
run ([], stack, state) = ([], stack, state)  -- If the code is empty, return the current state
run (inst:code, stack, state) =
  case inst of
    Push n -> run (code, push (show n) stack, state)
    Tru -> run (code, push "tt" stack, state)
    Fals -> run (code, push "ff" stack, state)
    Add ->
      case stack of
        Stk (x:y:xs) ->
          case (reads x, reads y) of
            ([(xVal, "")], [(yVal, "")]) ->
              run (code, push (show (xVal + yVal)) (pop (pop stack)), state)
            _ -> error $ "Run-time error"
        _ -> error $ "Run-time error"
    Mult ->
      case stack of
        Stk (x:y:xs) ->
          case (reads x, reads y) of
            ([(xVal, "")], [(yVal, "")]) ->
              run (code, push (show (xVal * yVal)) (pop (pop stack)), state)
            _ -> error $ "Run-time error"
        _ -> error $ "Run-time error"
    Sub ->
      case stack of
        Stk (x:y:xs) ->
          case (reads x, reads y) of
            ([(xVal, "")], [(yVal, "")]) ->
              run (code, push (show (xVal - yVal)) (pop (pop stack)), state)
            _ -> error $ "Run-time error"
        _ -> error $ "Run-time error"
    Equ ->
      case stack of
        Stk (x:y:xs) -> run (code, push (if x == y then "tt" else "ff") (pop (pop stack)), state)
        _ -> error $ "Run-time error"
```

```
Le ->
  case stack of
    Stk (x:y:xs) ->
      case (reads x, reads y) of
        ([(xVal, "")], [(yVal, "")]) ->
          run (code, push (if (xVal :: Integer) <= (yVal :: Integer) then "tt" else "ff") (pop (pop stack)), state)
        _ -> error $ "Run-time error"
    _ -> error $ "Run-time error"
Fetch x ->
  case lookup x (getStatePairs state) of
    Just value -> run (code, push value stack, state)
    Nothing    -> error $ "Run-time error"
Store x ->
  case stack of
    Stk (value:rest) -> run (code, pop stack, addState state (x, removeQuotes value))
    _                -> error $ "Run-time error"
Branch c1 c2 ->
  case stack of
    Stk ("tt":xs) -> run (c1 ++ code, pop stack, state)
    Stk ("ff":xs) -> run (c2 ++ code, pop stack, state)
    _             -> error $ "Run-time error"
Neg ->
  case stack of
    Stk(x:xs) ->
      case x of
        "tt" -> run(code, push "ff" (pop stack), state)
        "ff" -> run(code, push "tt" (pop stack), state)
        _    -> error $ "Run-time error"
    _ -> error $ "Run-time error"
And ->
  case stack of
    Stk (x:y:xs) ->
      if isBoolean x && isBoolean y
        then run(code, push(logicalAnd x y) (pop(pop stack)), state)
        else error $ "Run-time error"
    _ -> error $ "Run-time error"
Loop c1 c2 -> run (c1 ++ [Branch (c2 ++ [Loop c1 c2]) [Noop]] ++ code, stack, state)
Noop -> run (code, stack, state)
```

Por fim, são usadas duas funções auxiliares(isBoolean, logicalAnd) que são utilizadas na opção And e que consistem em verificar se os dois primeiros valores do topo da pilha são booleanos. Servem essencialmente para tornar este caso da função run mais legível.

```
isBoolean :: String -> Bool
isBoolean "tt" = True
isBoolean "ff" = True
isBoolean _    = False

logicalAnd :: String -> String -> String
logicalAnd "tt" "tt" = "tt"
logicalAnd _    _    = "ff"
```

Parte 2

Nesta parte, é nos pedido para considerar uma linguagem imperativa simples que consiste em expressões aritméticas e booleanas, e instruções compostas por atribuições do tipo x := a, sequência de instruções (instr1 ; instr2), instruções condicionais (if then else), e ciclos while. Todos estes statements estão divididos entre ‘;’.

Primeiramente, é nos pedido para criar 3 “datas” diferentes para representar expressões e statements da nossa linguagem.

```
data Aexp
  = ALit Integer
  | AVar String
  | AAdd Aexp Aexp
  | ASub Aexp Aexp
  | AMul Aexp Aexp
  deriving Show

data Bexp
  = BLit Bool
  | BNot Bexp
  | BAnd Bexp Bexp
  | BEq Aexp Aexp
  | BBEq Bexp Bexp
  | BLe Aexp Aexp
  deriving Show

data Stm
  = Assign String Aexp
  | If Bexp Program Program
  | While Bexp Program
  deriving Show

type Program = [Stm]
```

Esta foi a nossa solução. Os statements da nossa linguagem consistem em loops, ifs e assignments. Um programa seria uma lista de statements. Para representar expressões booleanas, usamos a enumeração Bexp e para representar expressões aritméticas, usamos Aexp. No data Bexp, para distinguir igualdade de inteiros de booleanos, usamos 'BEq' para inteiros e 'BBEq' para booleanos.

Na próxima alínea, é nos pedido para definir um compilador que compila a nossa linguagem e a traduz nas instruções que nos foram dadas na parte 1. Aqui, como sabemos que um programa é uma lista de statements, decidimos explorar toda a lista, e detectar que tipo de statement é que encontramos. Dentro desses statements fazemos o mesmo, só que chamamos a função responsável pela compilação de cada tipo de expressão.

```
compA :: Aexp -> Code
compA (ALit s) = [Push s]
compA (AVar var) = [Fetch var]
compA (AAdd e1 e2) = compA e2 ++ compA e1 ++ [Add]
compA (ASub e1 e2) = compA e2 ++ compA e1 ++ [Sub]
compA (AMul e1 e2) = compA e2 ++ compA e1 ++ [Mult]

compB :: Bexp -> Code
compB (BLit True) = [Tru]
compB (BLit False) = [Fals]
compB (BNot e) = compB e ++ [Neg]
compB (BAnd e1 e2) = compB e1 ++ compB e2 ++ [And]
compB (BBEq e1 e2) = compB e1 ++ compB e2 ++ [Equ]
compB (BEq e1 e2) = compA e1 ++ compA e2 ++ [Equ]
compB (BLe e1 e2) = compA e2 ++ compA e1 ++ [Le]

comp :: Stm -> Code
comp (Assign s e) = compA e ++ [Store s]
comp (If e p1 p2) = compB e ++ [Branch (compile p1) (compile p2)]
comp (While e p1) = [Loop (compB e)  (compile p1)]

compile :: Program -> Code
compile [] = []
compile (x:xs) = comp x ++ compile xs
```

Como por exemplo, no statement de assign, sabemos que temos de ter uma string com o nome da variável e uma expressão aritmética que nos vai dar o valor que queremos para essa variável. Então chamamos o compA, que é responsável pela compilação das expressões aritméticas, e o mesmo vai chamar-se a si próprio dependendo da expressão encontrada. Finalmente é adicionada a instrução Store à lista do tipo Code.

Finalmente, é nos pedido para criar um parser que transforme as instruções da nossa linguagem imperativa nos seus respectivos statements. Para realizar esta tarefa usamos a livraria Parsec. Um dos componentes mais importantes da livraria do Parsec, foi a função para avaliar expressões. Esta permitiu-nos definir prioridades para os diferentes operadores usados tanto para booleanos como para inteiros. A nossa implementação do parser consistiu em chamar o parser do Parsec e aplicar a string passada. Depois é chamada uma função que tem um parser para uma lista de statements, ou um programa. Nessa função é chamada o stmParser para 0 ou mais statements.

```
stmParser :: Parser Stm
stmParser = try assignParser <|> try ifParserElseInclosed <|> try ifParser <|> try whileParser
```

Este stmParser chama outros parsers para cada statement possível. Dentro desses parsers, tentamos encontrar a estrutura exacta de cada statement. Se um falhar, o stmParser tenta o próximo parser até encontrar o statement correspondente ou testar todos.

```
assignParser :: Parser Stm
assignParser = do
  spaces
  var <- identifier
  spaces
  string ":="
  spaces
  expr <- try aexpParser
  spaces
  char ';'
  return (Assign var expr)
```

Este seria um exemplo de um desses parsers.

Dentro dos statements parsers são depois chamados outros parsers, dependendo do tipo de statement encontrado. No caso do assign, é chamado um parser de expressões aritméticas. Dentro deste parser, temos outros parsers que vão ser responsáveis pela construção da expressão no tipo que queremos.

```
aexptable =
  [ [binary "*" AMul AssocLeft]
  , [binary "+" AAdd AssocLeft, binary "-" ASub AssocLeft]
  ]

binary name fun assoc = Infix (reservedOp name >> return fun) assoc

reservedOp :: String -> Parser String
reservedOp op = do
  try (spaces >> string op)
  spaces
  return op
```

Aqui temos a definição da tabela de propriedade dos operadores das expressões aritméticas. Também existe um parser para expressões com parênteses, mas não está definido nesta tabela. Uma solução semelhante é aplicada às expressões booleanas.

Com estes parsers todos juntos, conseguimos implementar com sucesso uma parser para a nossa linguagem.

Conclusões

Com este trabalho, foi possível aprofundar o nosso conhecimento sobre parsers e compiladores. Também ficamos a perceber melhor a linguagem Haskell.

A área em que tivemos mais dificuldades foi o parser, devido à pouca informação sobre o Parsec e à quantidade de bugs que foram encontrados.

Bibliografia

https://hackage.haskell.org/package/parsec-3.1.17.0/docs/Text-Parsec.html
https://hackage.haskell.org/package/parsec-3.1.17.0/docs/Text-Parsec-Expr.html

DPL Week 2 - 03 Parsing with Parsec