PFL2

Introduction

For the second Practical Assignment (TP2) of Functional and Logic Programming (PFL) 2023/2024 @ FEUP, we implemented, in Haskell, a low-level machine that executes basic arithmetic and logic operations and a small imperative programming language with its own compiler that transforms a program into a sequence of instructions for the machine.

Our group is T12_G06, composed of:

- João Mendes Silva Belchior - up202108777@up.pt (50%)
- José Francisco Reis Pedreiras Neves Veiga - up202108753@up.pt (50%)

Execution

To execute the program, you must have GHC installed. Then, you can run the following commands:

```
$ ghci
$ :l main.hs
```

The low-level machine

This machine is composed of a list of instructions (code), an evaluation stack that computes the instructions and a memory (state) that stores the values of the variables, where:

```
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]

type Stack = [Either Integer Bool]

type State = [(String, Either Integer Bool)]
```

The stack is implemented as a list of `Either Integer Bool` values, where `Left Integer` represents an integer value and `Right Bool` represents a boolean value. The state is implemented as a list of pairs `(String, Either Integer Bool)`, where `String` is the name of the variable and `Either Integer Bool` is the value of the variable.

To execute the machine, we should call the

```
run :: (Code, Stack, State) -> (Code, Stack, State)
```

function, which is recursive and, depending on the first instruction of the code (using pattern matching), calls itself with the new code, stack and state until the code is empty. The function returns the final state of the machine.

All the instructions are correctly implemented and tested for the correct inputs, and return an error if the input is not correct. For example, when we try to add two boolean values or fetch a variable that is not in the state.

The imperative programming language

This small language consists of arithmetic and boolean expressions and statements of the form:

- `x := a;` (assign the value of the arithmetic expression `a` to the variable `x`)
- `(instr1; instr2)` (sequence of instructions)
- `if then else` statements
- `while do` statements

We started with defining the basic structures of our language, which are:

```
data Aexp = Num Integer | Var String | AddA Aexp Aexp | SubA Aexp Aexp |
MultA Aexp Aexp deriving Show
data Bexp = Bool Bool | EqA Aexp Aexp | EqB Bexp Bexp | LeA Aexp Aexp |
NegB Bexp | AndB Bexp Bexp deriving Show

data Stm = Assign String Aexp | Skip | Seq Stm Stm | If Bexp Stm Stm |
While Bexp Stm deriving Show
type Program = [Stm]
```

Then, we implemented the compiler, which transforms a program into a sequence of instructions for the low-level machine to compute. The compiler is in a way similar to the `run` function of the machine, since it is recursive and, depending on the first instruction of the program (using pattern matching), calls itself with the new program until the program is empty. The compiler returns the code that the machine will compute.

We compile statements, arithmetic and boolean expressions:

```
compA :: Aexp -> Code
compA (Num x) = [Push x]
compA (Var x) = [Fetch x]
compA (AddA x y) = compA x ++ compA y ++ [Add]
compA (SubA x y) = compA x ++ compA y ++ [Sub]
compA (MultA x y) = compA x ++ compA y ++ [Mult]

compB :: Bexp -> Code
compB (Bool x) = if x then [Tru] else [Fals]
compB (EqA x y) = compA x ++ compA y ++ [Equ]
compB (EqB x y) = compB x ++ compB y ++ [Equ]
compB (LeA x y) = compA y ++ compA x ++ [Le]
compB (NegB x) = compB x ++ [Neg]
compB (AndB x y) = compB x ++ compB y ++ [And]
```

```
compile :: Program -> Code
compile [] = []
compile (Assign x aexp:xs) = compA aexp ++ [Store x] ++ compile xs
compile (Skip:xs) = Noop:compile xs
compile (Seq instr1 instr2:xs) = compile (instr1:instr2:xs)
compile (If bexp thenStm elseStm:xs) = compB bexp ++ [Branch (compile
(thenStm:xs)) (compile (elseStm:xs))]
compile (While bexp doStm:xs) = Loop (compB bexp ++ compile (doStm:xs))
[Noop] : compile xs
```

Now the hardest part arrives, implementing the parser that transforms a string of our language into a program.

Firstly, we developed a `lexer` that transforms the string into a list of tokens, which are the most basic components of the language, such as numbers, variable names, operators and keywords.

We handled (whitespaces) and `\n` with a rather simple:

```
filter (/= ' ') . filter (/= '\n')
```

and made sure that variable names don't contain reserved keywords, start with a lowercase letter and can have capital letters, numbers and underscores:

```
lexer (x : xs)
  | x `elem` ['a'..'z'] =
    let (var, rest) = span (\c -> c `elem` (['a'..'z'] ++ ['A'..'Z'] ++
['0'..'9'] ++ ['_'])) (x:xs) in
      if isValidVar var
        then var : lexer rest
        else error "Run-time error"

isValidVar :: String -> Bool
isValidVar x = not (any (`isInfixOf` x) ["if", "then", "else", "while",
"not", "and", "True", "False", "do"])
```

Secondly, we call the `parse` function:

```
parse :: String -> Program
parse = buildData . lexer . filter (/= ' ') . filter (/= '\n')
```

where the `buildData` function that is called will handle all the cases for the different statements, expressions and operators of the language and return the `Program`.

To ensure the correct precedence for the different arithmetic and boolean operators, the idea is to build a parse tree, where the leaves are the tokens and the nodes are the operators. Then, we can traverse the tree

in a way that respects the precedence of the operators, and build the `Aexp` and `Bexp` from the leaves to the root.

All these auxiliary functions are of the form:

```
buildStatement :: [String] -> (Statement, [String])
```

for building statements and:

```
buildOperator :: [String] -> (Operator, [String])
```

for building arithmetic and boolean expressions. The main idea is to parse/consume the maximum number of tokens possible, and then return the remaining tokens to be parsed by other functions.

Some examples:

```
buildAssign :: [String] -> (Stm, [String])
buildAssign [] = error "Run-time error"
buildAssign (x:":=":xs)
  | isValidVar x = (Assign x (fst (buildAexp (takeWhile (/= ";") xs))),
tail (dropWhile (/= ";") xs))
  | otherwise = error "Run-time error"
buildAssign _ = error "Run-time error"
```

```
buildBexp :: [String] -> (Bexp, [String])
buildBexp = buildAnd

buildAnd :: [String] -> (Bexp, [String])
buildAnd xs =
  let (left, rest) = buildEqB xs
  in buildAnd' left rest
```

```
buildAexp :: [String] -> (Aexp, [String])
buildAexp  = buildAddSub

buildAddSub :: [String] -> (Aexp, [String])
buildAddSub xs =
  let (left, rest) = buildMult xs
  in buildAddSub' left rest
```

Like so, we respect the precedence of the operators, since we first parse the tokens that are of lower precedence and then the ones of higher precedence. The precendes are: (from higher to lower)

- For arithmetic expressions: `( )` ; `+` and `-` ; `*`
- For boolean expressions: `( )` ; `<=` and `>=` ; `==` ; `not`; `=` ; `and`

where `==` is used for arithmetic equality and `=` is used for boolean equality.

Conclusions

Haskell is a very hard language to grasp but very efficient and powerful for implementing parsers and compilers. It gives us a new perspective on some topics like recursion, pattern matching and pure functional programming.