Practical Assignment 2 - 2023/2024 PFL
Haskell Compiler

Group: *T05_G10*

Members:

- *Guilherme Correia da Silva Coutinho, up202108872 - 50%*
- *Xavier Ribeiro Outeiro, up202108895 - 50%*

Strategy Used

This assignment was split in two parts:

Low-level Assembler

The goal was to implement a low-level machine that receives a set of configurations of the form (c, (e, s)) where c is a list of instructions (or code) to be executed, e is the evaluation stack, and s is the storage. We use the evaluation stack to evaluate arithmetic (composed of integer numbers only, which can positive or negative) and boolean expressions.

The instructions of the machine are: push-n, add, mult, sub, true, false, eq, le, and, neg, fetch-x, store-x, noop, branch(c1, c2) and loop(c1, c2).

An example of an instruction is:

```
([Push 10, Push 4, Push 3, Sub, Mult], ("-10", ""))
```

To solve this, we defined some data types:

```
data StackVal = IntVal Integer | BoolVal Bool deriving (Show)
type Stack = [StackVal]
type State = [(String, StackVal)]
```

- **StackVal** represents either an integer or a boolean that is going to be stored temporarily in the stack
- **Stack** is represented as a list of StackVals

- **State** is represented as a list of tuples, where each tuple has a string and a StackVal

After defining these data types and also using the provided ones in the file template, we implemented the required functions in the template.

For the main function `run`, we implemented a `case of` to switch the behaviour of the function depending on the instruction that was inputted to the program.

Then we implemented the `testAll` function to run a set of tests to verify that the machine was well implemented. The tests returned all True.

Imperative Language Compiler

Now considering a small imperative programming language with arithmetic and boolean expressions, and statements consisting of assignments of the form x := a, sequence of statements (instr1 ; instr2), if then else statements, and while loops. With this, our assignment was to build a compiler of this language to translate it to instructions for the previous low-level machine.

To start, we were asked to define the following data types:

```haskell
-- Arithmetic expressions
data Aexp = Var String
  | IntConst Integer
  | Add2 Aexp Aexp
  | Sub2 Aexp Aexp
  | Mul Aexp Aexp
  | BinaryOp String Aexp Aexp
  deriving (Show)

-- Boolean expressions
data Bexp = BTrue
  | BFalse
  | Eq Aexp Aexp
  | Leq Aexp Aexp
  | Not Bexp
  | And2 Bexp Bexp
  | Beq Bexp Bexp
  deriving (Show)

-- Statements
data Stm = Assign String Aexp
  | Seq [Stm]
  | If Bexp Stm Stm
  | While Bexp Stm
  | Compound Stm Stm
  deriving (Show)

-- Program
type Program = [Stm]
```

After defining these data types, we implemented the required functions in the template and some additional ones like:

- **tokenize** - Takes a string and transforms it into a list of Tokens. This function will be called in the `parse` function.
- **parseProgram** - This function takes a list of Tokens and returns a `Program` which is then going to be used in the function `compile`. This function is the one who makes the call to the function `run` of the low-level machine.
- **parseStm** - This function is called by `parseProgram`, it has a `case of` implemented to switch between which statement is inputted. If an arithmetic statement is inputted, it calls `parseAexp`, if it's a boolean expresseion, it calls `parseBexp`, otherwise, it calls itself when encountering an `if`, an `else` or a `while`.

Then we implemented the `testAll2` function to run a set of tests to verify that the compiler was well implemented. The tests returned all True.

Conclusions

This work has proven to be a great way to learn both a new language and a different way of thinking, which was new to us. We believe that we have reinforced our prior knowledge and also expanded it.