CSIS2258/COMP3258 Functional Programming 2014/15

Assignment 3 (Deadline: 11:59 AM HKT, May 1)

Code Correctness (10%)

Make sure that your code compiles and produces no extra warnings with GHC 7.8.3; that you did not change the provided names or types in the `Assignment3.hs` skeleton and **did not add extra import statements** on top of the provided ones. Please include any extra code you wrote to test your code.

Code Style (10%)

Your code should be well-written (e.g. sensible names of functions or variables where appropriate), documented, well-formatted, and produce no warnings with HLint.

Submit your solution on Moodle in one file (use the attached `Assignment3.hs` and `Parsing.hs` skeleton) before the deadline (which is a cut-off time; no late submissions will be accepted).

Plagiarism

Please do this assignment on your own; if, for a small part of an exercise, you use something from the internet or were advised by your classmate, please mark and attribute the source in a comment. Do not use ***publicly accessible*** code sharing websites (pastebin, lpaste, fpcomplete, GitHub public repository…) for your assignment to avoid being suspected of plagiarism.

Overview

In this assignment, you will implement a simulator for a simplified computer named Little Man Computer, which is consist of two parts:

1) Write a parser for parsing the instructions
2) Write an evaluator for evaluating the instructions.

You are strongly recommended to read the description of LMC carefully. To simplify it, we restrict a valid LMC program with additional rules:

1) Every line must contain an instruction
2) Each line starts with either a label or a space
3) DAT instructions will appear only at the bottom of a program

---

Exercise 1 (10%)

(a) Write a function `option :: a -> Parser a -> Parser a` that tries to apply parser **p**. If **p** fails, it returns the value **x**, otherwise the value returned by **p**.

(b) Use `option` to define a function `optionMaybe :: Parser a -> Parser (Maybe a)`

that tries to apply parser **p**. If p fails, it return **Nothing**, otherwise it returns **Just** the value returned by **p**.

```
> parse (optionMaybe int) "12a"
[(Just 12, "a")]
> parse (optionMaybe int) "a"
[(Nothing, "a")]
```

Exercise 2 (10%)

(a) Write a function `sepBy1 :: Parser a -> Parser b -> Parser [a]` that parses *one or more* occurrences of **p**, separated by **sep** and return a list of values returned by **p**.

(b) Use `sepBy1` to define a function `sepBy :: Parser a -> Parser b -> Parser [a]` that parses *zero or more* occurrences of **p**, separated by **sep** and return a list of values returned by **p**.

```
> parse (sepBy (char 'a') (char ' ')) "a a b"
[("aa"," b")]
```

Exercise 3 (10%)

Given the following definition:

```
data Instruction = ADD Label
                 | SUB Label
                 | STA Label
                 | LDA Label
                 | BRA Label
                 | BRZ Label
                 | BRP Label
                 | DAT Int
                 | INP
                 | OUT
                 | HLT
```

write a function `instruction :: Parser Instruction` that parses an instruction.

Exercise 4 (10%)

Each line of a LMC program may start with a label and/or end with a comment. Write a parser that parses a line of a LMC program.

```
line :: Parser (Maybe Label, Instruction)
```

If everything implemented is implemented correctly, you should be able to use the provided function `parseLMC` to parse a LMC program.

Exercise 5 (5%)

Write a function `showProgram :: Program -> String` that prints out a LMC program. (`Program` is just a type synonym of `[(Maybe Label, Instruction)]`).
Each line of the output should be in one of the following formats:

```
"<space>INSTRUCTION"
```

```
"<space>INSTRUCTION<space>OPERAND"
"LABEL<space>INSTRUCTION"
"LABEL<space>INSTRUCTION<space>OPERAND"
```

And lines are separated with '\n'. (You can use this function to test whether `parseLMC` works correctly)

Exercise 6 (15%)

To interpret a LMC program, we need to maintain an environment that stores the state of intermediate interpretation. The definition of Env is given:

```
data Env
    = Env
    { mailboxes :: [Mailbox]
    , accumulator :: Accumulator
    , pc :: Addr -- program counter
    , instructions :: [Instruction]
    , labelAddr :: [(Label, Addr)]
    }
```

Here, to give fields meaningful names and access them easily, we use Haskell record syntax to define this datatype. The record syntax reference is given at the end of this sheet and more detailed explanation can be found on Haskell Wikibook.

(a) Write a function `initMailboxes :: Program -> [Mailbox]` that initialises the mailboxes according to all the **DAT** instructions. (`Mailbox` is a type synonym of `(Label, Int)`)

(b) Write a function `initLabelAddr :: [Maybe Label] -> [(Label, Addr)]` that attaches each Label with a unique address starting from 0. (`Addr` is a type synonym of `Int`)

(c) Write a function `mkInitEnv :: Program -> Env` that initialises the environment according to the program.

Exercise 7 (20%)

Write a function `decode :: Instruction -> IOEnv` that decodes the current instruction and recursively decodes the next one. (`IOEnv` is a type synonym of `StateT Env IO`)

If everything implemented is implemented correctly, you should be able to use the provided function `evalProgram` to interpret a LMC program.

HINT: You may refer to the definition of decoding **INP** and **OUT** to define remaining ones.You may want to define more auxiliary functions for updating the environment like `set/getAccumulator` provided.

Challenge yourself

Deal with errors more appropriately instead of simply calling error function when interpreting a LMC program.

HINT: Wrap the `IOEnv` into `ErrorT`.

LMC Instruction Code Reference

INP: ask user to type in a number then store it in the accumulator
STA: get the value in the accumulator then store it in the mailbox of the given label
LDA: get the value in the mailbox of the given label then store it in the accumulator
ADD: get the value in the mailbox of the given label, add it with the accumulator and store
SUB: subtract the value in mailbox of the given label from the value in the accumulator
OUT: print out the value in the accumulator
BRA: jump to the instruction with the given label
BRZ: check the value in the accumulator. if it is zero jump to the instruction with the given label
BRP: check the value in the accumulator. if it is positive, jump to the instruction with the given label
HLT: terminate the program
DAT: declare a variable with an initial value. The initial value is zero by default

Haskell Record Syntax Reference

1) construct an instance of `Env`
   1) `env = Env {mailboxes = [], accumulator = Nothing, pc = 0, instructions = [], labelAddr = []}`
   2) `env = Env [] Nothing 0 [] []`
2) access a field: `pc env`
3) update a field: `env { pc = 0 }`