P. Thiemann, G. Radanne Wintersemester 2019/20

Functional Programming

http://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2019/

Exercise Sheet 5 – IO, Parsing

2017-06-19

1 The IO type

Exercise 1 (Numbers game)
In the Numbers game, the computer tries to guess a user-imagined number between 1 and 100. Here is an example. Texts after the > prompt are user inputs.

```
Choose a number between 1 and 100!
Is it 50?
> greater
Is it 75?
> smaller
Is it 62?
> smaller
Is it 56?
> Yes
I won in 4 attempts!
```

Implement this game using the `IO` type and the `do` notation. In your Stack project, add this program in the /bin directory to easily produce a real binary.

Exercise 2 (Stack Calculator Interface)
In exercise sheet 1, we implemented a simple stack calculator. This calculator was missing a crucial component: a command line interface!

Using `IO`, add a command line interface to your implementation of the stack calculator. Each line should represent a command (for example "push 3" or "add"). The program should show the stack at each step. "exit" should exit the program.

2 Parsing

Download the file `ParserCon.hs` from the lecture page. It contains a parser module similar to the one developed during the lecture, but equipped with `Functor`, `Applicative`, `Alternative`, `Monad` and `MonadPlus` Instances.

Exercise 3 (Parsing)
Define the parser combinators described below:

- `pmany :: parser t r -> parser t [r]`

  `pmany p` accepts `p` zero or more times and summarizes the results in a list.

- `pmany1 :: Parser t r -> parser t [r]`

  `pmany1 p` accept `p` one or more times and summarize the results in a list.

- `pIntList :: Parser Char [Integer]`

  `pIntList` accepts lists in Haskell syntax that contain integer literals.

  For example `pIntList "[1, 22,33 \ n, 44]" == ([1, 22, 33, 44], "")`

- `pPaliAB :: Parser Char String`

  `pPaliAB` accepts palindromes from the characters `'a'` and `'b'`

- `pPali :: (Eq r) => parser t r -> parser t [r]`

  `pPali p` accepts the palindromes that consist of elements that accept `p`.

  For example: `pPaliAB = pPali (lit 'a' `palt` lit 'b')`.

- `pTwice :: (Eq t) => parser t [t] -> parser t [t]`

  For all `ts` accepting `p`, `ts ++ ts` is accepted by `pTwice p`.

Exercise 4 (While)
Implement a parser for the following grammar of a simple programming language:

```
stmts :: = stmt ';' stmts
       | stmt
stmt :: = 'while' exp 'do' stmts 'done'
       | id ': =' exp
exp :: = 'if' exp 'then' exp 'else' exp 'fi'
       | aexp cmp aexp
       | 'not' exp
       | aexp
aexp :: = num
       | id
       | '(' aexp op aexp ')'
cmp :: = '<=' | '>' | '==' | '! ='
op :: = '+' | '-' | '*' | '/'
num :: = "[0-9] +"
id :: = "[a-zA-Z] [a-zA-Z0-9] *"
```

In the grammar above, terminal symbols are either literals in single quotes (for example, `'if'`) or regular expressions in double quotes (for example, `"[0-9]+"`).

An example program of the language:

```
x: = 0; y: = 5;
while x <= 10 do
y: = (y * 5); x: = (x + 1)
done;
y: = if y> 10000 then 10000 else y fi
```

On the homepage you will find the module `MiniWhile.hs` with some basic structure to get you started. Notably, a lexer:

```
lexer :: string -> Maybe [Token]
```

which should be used to preprocess the string.

Your task is to implement a parser for the language and to extend the various type definitions to the full language.