Prof. Dr. Peter Thiemann
Janek Spaderna
janek.spaderna@pluto.uni-freiburg.de

Winter Term 2022/23
Issued 2023-01-15
Discussion on 2023-01-20

Functional Programming
https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2022/

Exercise Sheet 10

Download the file `Parser.hs` from the lecture page. It contains a parser module similar to the one developed during the lecture.

The **`Parser`** type is equipped with **`Functor`**, **`Applicative`** and **`Monad`** instances. The primitive operations such as `pmap` are not exported. Use the typeclass functions like `fmap` instead.

Additionally, there is an instance for `Alternative`. It provides the function `empty` corresponding to `pempty`, and the operator `(<|>)` corresponding to `palt`. Through the **`Alternative`** instance the functions `some` and `many` are available. Read their documentation to learn what they do and what their differences are.

Exercise 1 (Parser combinators)

Define the combinators described below. Do not rely on the **`Parser`** type's internal representation; i. e. use only the operations exported from `Parser.hs`. (Functions from other modules, such as `Control.Monad` for example, may of course be used.)

-
  ```
  psepby  :: Parser t r -> Parser t sep -> Parser t [r]
  psepby1 :: Parser t r -> Parser t sep -> Parser t [r]
  ```

  ``p `psepby` s`` accepts `p` zero or more times. Instances of `p` have to be separated by `s`.

  ``p `psepby1` sep`` requires at least one occurence of `p`

-
  ```
  ppali :: Parser t r -> Parser t [r]
  ```

  `ppali p` accepts palindromes that consist of elements accepted by `p`. If you have trouble defining `ppali` try your hand first at

  ```
  ppaliAB :: Parser Char String
  ```

  which should accept palindromes consisting of `'A'` and `'B'`.

-
  ```
  pcounted :: Parser Char r -> Parser Char [r]
  ```

  `pcounted p` parses an integer n followed by n instances of `p`. For example

  ```
  exParser (pcounted (lit 'a')) "3aaaaa" == [("aaa", "aa")]
  ```

Exercise 2 (MiniLang parser)

The goal of this exercise is to write a parser for the language *MiniLang* from exercise sheet 7. The grammar is specified below. Terminal symbols are either literals in single quotes (for example, `'if'`) or regular expressions in double quotes (for example, `"[0-9]+"`).

```
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]*"
```

For example, a program in the language may look like this:

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

1. First, write a *lexer*, or also called a *tokenizer*. It is a parser of type `Parser Char Token`. The file `MiniLangParser.hs` linked on the lecture home page contains basic definitions to get you started. In particular, implement the `ptoken` parser.

   Note Be careful in your handling of keywords and identifiers: `find` should be tokenized as one identifier instead of keyword `fi` and identifier `nd`.

2. Write parsers for the non-terminals. For example, the parser for the non-terminal `exp` should have type `Parser Token MiniLang.Expr`.

   Because we first parse the input into a list of tokens, we don't have to worry about, for example, white space in these parsers and can instead focus on the grammar.

3. Combine your lexer and the parsers from step 2 into a function

   ```
   parseProgram :: String -> Maybe MiniLang.Prog
   ```

Exercise 3 (MiniLang frontend)

Write an executable `mini-lang` to parse and execute a MiniLang program. Use the parser from exercise 2 and the interpreter from sheet 7.

The executable should read the program source code from the standard input. Command line arguments of the form *`var`*`=`*`value`* should initialize the program memory. (Unspecified variables should still default to `0`.) Print the program memory after execution to the terminal.