Computer Science: EDAN40: Functional Parsing

4 min read • original

EDAN40: Functional Parsing

Summary

In this assignment you will create a parser and an interpreter for a small imperative language. The main goal is to get acquainted with a monadic way of solving a classical problem in computer science.

Introduction

The following program is written in the programming language to be parsed and interpreted in this assignment.

```
read k;
read n;
m := 1;
while n-m do
  begin
    if m - m/k*k then
      skip;
    else
      write m;
    m := m + 1;
  end
```

The language has just one data type, integer, and variables are not declared. In the `while` and `if` statements a positive expression value is interpreted as true while 0 and negative values mean false.

The program above reads two integers, `k` and `n`, and writes all integers between `1` and `n` that are multiples of `k`.

The grammar for the language is given by

```
program ::= statements
statement ::= variable ':=' expr ';'
        | 'skip' ';'
        | 'begin' statements 'end'
        | 'if' expr 'then' statement 'else' statement
        | 'while' expr 'do' statement
        | 'read' variable ';'
        | 'write' expr ';'
statements ::= {statement}
variable ::= letter {letter}
```

An explanation of this grammar and a parser for an expression, `expr`, can be found in the document Parsing with Haskell, by Lennart Andersson. The intended semantics for the language should be obvious from the keywords for anybody familiar with Java language.

Program structure

You are given the stub of a solution:

CoreParser.hs

defines the `Parser` type and implements the three elementary parsers, `char`, `return` and `fail`, and the basic parser operators `#`, `!`, `?`, `#>`, and `>->`, described in Lennart Andersson's description (and during the lecture).

The class `Parse` with signatures for `parse`, `toString`, and `fromString` with an implementation for the last one is introduced.

The representation of the Parser type is visible outside the module, but this visibilty should not be exploited.

`Parser.hs`

contains a number of derived parsers and parser operators.

`Expr.hs`

contains a data type for representing an arithmetic expression, an expression parser, an expression evaluator, and a function for converting the representation to a string.

`Dictionary.hs`

contains a data type for representing a dictionary.

`Statement.hs`

contains a data type for representing a statement, a statement parser, a function to interpret a list of statements, and a function for converting the representation to a string.

`Program.hs`

contains a data type for representing a program, a program parser, a program interpreter, and a function for converting the representation to a string.

`Test*.hs`

contains test data.

In a test using the program in the introduction with the following definitions

```
src = "read k; read n; m:=1; ... "
p = Program.fromString src
```

the expression `Program.exec p [3,16]` should return `[3,6,9,12,15]`.

Assignment and hints

1. In `Parser.hs` implement the following functions. All the implementations should use other parsers and parser operators. No implementation may rely on the fact that the parsers return values of type `Maybe(a, String)`. This means e.g. that the words `Just` and `Nothing` may not appear in the code.

   `letter :: Parser Char.`

   `letter` is a parser for a letter as defined by the Prelude function `isAlpha`.

   `spaces :: Parser String.`

   `spaces` accepts any number of whitespace characters as defined by the Prelude function `isSpace`.

   `chars :: Int -> Parser String.`

   The parser `chars n` accepts `n` characters.

   `require :: String -> Parser String.`

   The parser `require w` accepts the same string input as `accept w` but reports the missing string using `err` in case of failure.

`-# :: Parser a -> Parser b -> Parser b.`

The parser `m -# n` accepts the same input as `m # n`, but returns just the result from the `n` parser. The function should be declared as a left associative infix operator with precedence 7. Example:

```
(accept "read" -# word) "read count;" -> Just ("count", ";")
```

`#- :: Parser a -> Parser b -> Parser a.`

The parser `m #- n` accepts the same input as `m # n`, but returns the result from the `m` parser.

2. Implement the function `value` in `Expr`. The expression `value e dictionary` should return the value of `e` if all the variables occur in `dictionary` and there is no division by zero. Otherwise an error should be reported using `error`.
3. Implement the type and the functions in the `Statement` module. Some hints:
    1. The data type `T` should have seven constructors, one for each kind of statement.
    2. Define a parsing function for each kind of statement. If the parser has accepted the first reserved word in a statement, you should use `require` rather than `accept` to parse other reserved words or symbols in order to get better error messages in case of failure. An example:
    ```
    assignment = word #- accept ":=" # Expr.parse
                      #- require ";" >-> buildAss
    buildAss (v, e) = Assignment v e
    ```
    3. Use these functions to define `parse`.
    4. The function `exec :: [T] -> Dictionary.T String Integer -> [Integer] -> [Integer]` takes a list of statements to be executed, a dictionary containing variable/value pairs, and a list of integers containing numbers that may be read by `read` statements and the returned list contains the numbers produced by `write` statements.
    The function `exec` is defined using pattern matching on the first argument. If it is empty an empty integer list is returned. The other patterns discriminate over the first statement in the list. As an example the execution of a conditional statement may be implemented by
    ```
    exec (If cond thenStmts elseStmts: stmts) dict input =
        if (Expr.value cond dict)>0
        then exec (thenStmts: stmts) dict input
        else exec (elseStmts: stmts) dict input
    ```
    For each kind of statement there will be a recursive invocation of `exec`. A write statement will add a value to the returned list, while an assignment will make a recursive call with a new dictionary.
4. In the `Program` module you should represent the program as a `Statement` list. Use the parse function from the `Statement` module to define the parse function in this module. Use the `exec` function in the `Statement` module to execute a program.
5. Implement `toString :: T -> String` in `Statement` and `Program`. A newline character should be inserted after each statement and some keywords, but no indentation of lines is required. However, it will be appreciated. No spurious empty lines should appear in the output.

Provided documents

There is a zip file containing a partial implementation.

Lennart Andersson's "Parsing in Haskell" describing building parsers using `Maybe`.

Submission requirements

This assignment is to be solved individually. You are to submit your solution to fp@cs.lth.se with a subject line of "`Assignment 4 by` *`username`*", where *`username`* is your login, no later than 11th December 2013. The solution must contain a zip-file containing a directory with all modules. To pass, your program must work and consist of readable modules. Every value (or function) *exported* from a module must have an explicit type. You should also state how many hours you have used for this assignment.

Original URL:
http://cs.lth.se/english/course/edan40_functional_programming/programming_assignments/functional_parsing