Programming Paradigms 2022
Session 11: Monadic parsing

Problems for solving and discussing

Hans Hüttel

22 November 2022

A file you will need

For this session, we will need the modules mentioned in chapter 13 of the textbook and the many functions defined in that chapter. From the entry for this session on the Moodle course page you can get the file Parsing.hs that contains all of this. You need to compile and import it as a module. To compile Parsing.hs, use the ghc compiler for Haskell. In a terminal, write

```
ghc -I. --make Parsing.hs
```

This will generate files called Parsing.hi and Parsing.o. To simplify things in the following, make sure that these files are in the same directory as your program that uses them.

Problems that we will definitely talk about

1. ***(Everyone at the table – 10 minutes)***

   Why not simplify the existing implementation of a parser for arithmetic expressions by using a revised definition of expr?

   We could write the following instead of what Graham Hutton writes for expr:

   ```
   expr :: Parser Int
   expr = do
                 t <- term
                 return t
                  <|>
                  do
                        t <- term
                        symbol "+"
                        e <- expr
                        return (t + e)
   ```

   Would it be a good idea?

2. ***(Work in pairs – 25 minutes)***

   The formation rules for regular expressions over the alphabet $\{a, b\}$ are

   $$R ::= a \mid b \mid R_1 \circ R_2 \mid R_1 \cup R_2 \mid (R_1)^* \tag{1}$$

   The task is now to build a parser for regular expressions.

   But before you can build a parser, you have to find a context-free grammar for regular expressions that you can use as the basis of your parser. The formation rules above are not good enough for that (why?)

   It is a good idea to see how a very similar issue is handled by Graham Hutton in the chapter for today, when he builds a parser for arithmetic expressions, and use his approach.

3. ***(Work in pairs – 15 minutes)***

   Based on (1), define an algebraic datatype Rexp in Haskell (using data) for regular expressions over the alphabet $\{a, b\}$ and extend your parser such that it will, when supplied with a syntactically correct regular expression R, give you the corresponding term in Rexp.

4. ***(Everyone at the table** – 20 minutes)*

   Given the regular expressions that we have seen: How would you define a parser for a given regular expression R that you could use to decide if a given string w matches R? *Hint:* There are many useful combinators in the text for today! Try to use them to build a parser for checking if strings can matched by $(a \cup b) \circ bc^*$.

More problems to solve at your own pace

a) Here are the formation rules in the abstract syntax for statements in a version of the imperative programming language known as **Bims**:

$$S ::= \texttt{skip} \mid x := a \mid S_1; S_2 \mid \texttt{if}\ a_1 = a_2\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S$$

Given the parser for arithmetic expressions from the text for today, write a parser for statements in **Bims**. *Hint:* You first need to come up with a corresponding concrete syntax for statements.