Functional Programming – Series 6

In this series of exercises we will work with *grammars* and a *parser generator*, and we will transform the *parse tree* which results from the parser generator into an *Abstract Syntax Tree* (AST), according to some embedded language.

The class ToRoseTree. Until now you had to write functions to translate trees into rose trees such that they could be graphically shown. However, the module *FPPrac.Trees* contains a class *ToRoseTree* and a *generic* function *toRoseTree* that is defined for trees in this class:

$$toRoseTree :: ToRoseTree\ a \Rightarrow a \rightarrow RoseTree$$

In order to let your tree-types be instances of this class, you'll need some GHC-compiler directives, an extra import, and some additional deriving-clauses. These additions are included in the files in the zip-file `FP-6.zip` that you have to download from Blackboard:

- `FP_TypesEtc.hs`: contains types and some elementary functions. You'll have to change and extend several of these.

  This file also shows the additional compiler directives (on the top line), imports, and deriving-clauses (at the end of definitions of algebraic types) needed for the class *ToRoseTree*.

- `FP_ParseGen.hs`: contains the parser generator. *Do not change this file.* If you do, you're at your own.

- `FP_Grammar.hs`: contains an example of a small grammar and some test calls. This file imports the two files above, and also contains the necessary compiler directives. So this is the file that you have to load in ghci.

- The file `PreParserGen.hs` contains an elementary form of the parser generator. This form is not needed for the practical exercises, but gives a more clear view on the essential structure of the parser generator.

Exercise 1. (a.) Write a grammar for expressions and statements that contain at least

- *expressions*:
  - numbers, possibly containing a decimal dot,

  - variables,
  - arithmetical expressions, where compound expressions may be between brackets. There should be arithmetical operations (such as $+, *, -$) as well as comparative operations (such as $==$, $<$, $\leqslant$). At this moment you may assume that the programmer writes code that is type-correct.

- *statements*:
  - assignment,
  - a *repeat* statement to repeat a sequence of statements a number of times, where any arithmetical expression is allowed to determine how many repetitions have to be executed.

Extend the type *Alphabet* (in `FP_TypesEtc.hs`) with constructors for the necessary non-terminals.

(*b.*) Formulate the grammar as a function of type

$$Alphabet \rightarrow [\,[\,Alphabet\,]\,]$$

Note that the function *parse* assumes that tokens are *2-tuples* of the form:

$$(Alphabet, String)$$

where the meaning of a tuple (nt, str) is as follows:

- nt is the non-terminal that indicates the syntactic category to which the string str belongs. A practical way to add these non-terminals is to first split the input string in substrings, and then add the appropriate non-terminal (a function doing that is called a *lexer*),
- str is the string that indicates the token "itself",

Generate the parsetree, and present it using the function *prpr* (for "pretty prinit") as well as graphically, using *toRoseTree* ad *showRoseTree*.

Exercise 2. Create a syntactic category for *reserved words* (such as *repeat*, *if*, *then*) that may not be used as variables. Extend your grammar and your lexer such that this is taken care of.

Exercise 3. Parse trees contain a lot of (semantically) redundant information, for example, in order to generate the instructions to execute an expression or statement on a processor, a parse tree is not the most pleasant starting point.

(*a.*) Define algebraic types for expressions and statements that is more suitable as such a starting point. For example, such types are also used in the exercises on code-generation, playing the role of embedded languages.

(*b.*) Write a function that transforms a parse tree into expressions according to these algebraic data types (note that such data types correspond to *abstract syntax trees*).

Exercise 4. (*a.*) Extend the above grammar for *expressions* with an *if-then-else* expression, where the conditional, as well as the *then* and *else* parts can be arbitrary expressions.

(*b.*) In order to calculate an *if-then-else* expression on the *core* of Session 5, you'll need some extra instructions. Explain which extra instructions you need and why.

Exercise 5. Extra – for enthousiasts. Rewrite your grammar by taking priority of operations into account, such that many brackets are not necessary anymore.