Functional Programming – Series 6a

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 AST (according to some embedded language).

Preparation. Update the *eventloop* package, using *cabal*, see the update/install scripts under *General* on Blackboard.

After updating you have the function

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

at your disposal, which translates a tree-type in the class $ToRoseTree$ into a $RoseTree$, such that you can show a tree graphically. In order to let your tree-types be instances of this class, you'll need some GHC-compiler directives, as well as additional deriving-clauses (see below).

Download from Blackboard the files

- `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. If you change this file, 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.

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

- *expressions*:
  - numbers, possibly containing a decimal point,
  - 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,
  - 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, and formulate the grammar as a function of type

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

Note that the function *parse* assumes that tokens are *3-tuples* of the form:

$$(Alphabet, String, Int)$$

where the meaning of a tuple (nt, str, k) 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",
- k is the position of the token in the total list of tokens, and is needed for error messages generated by the parser. It is convenient to add these numbers as a last step, once the list of all substrings are in fact already known.

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

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. 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.

Write a function that transforms a parse tree into expressions according to these algebraic data types (apart from expression in an "embedded languages", such expressions are also called "abstract syntax trees").

Exercise 4. 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.

Exercise 5.

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