Functional Programming – Series 4

Exercise 1. The task in this exercise is to generalize the binary tree types from series 3 to *one* parameterized tree type.

a. Define a parametrised type for *binary* trees:

$BinTree\ a\ b$

where a is the type of the elements at the internal nodes of the tree, and b the type of the elements at the leaves.

Note that *BinTree* may be considered as a *function* on *types*.

b. Redefine all binary tree types of series 3 (*Tree1a* – *Tree1c*, *Tree4*) as specific instances of the type $BinTree\ a\ b$.

For types *Tree1c* and *Tree4* you will need a type which contains only *one* element, to represent a "non-value". Call this type *Unit*.

c. Write a function *pp* of type $BinTree\ a\ b \to RoseTree$ which is usable for all instances of type $BinTree\ a\ b$.

Tokenizing and Parsing

The exercises below are about parsing expressions of various formats, where in the first exercises no tokenization is needed, whereas later an input string has to be tokenized first. Your parse functions do not have to give error messages.

Exercise 2. An expression os of the form:

– *Number*

– '(' *Expr Op Expr* ')'

Note that a compound expression always is surrounded by brackets, and that an expression does not contain variables. Possible operations (*Op*) are: $+$, $-$, $*$, $/$, $\hat{}$.

Examples of such expressions are:

6

((3*5)+9)

a. In this part we assume that expressions do not contain spaces, there are no identifiers, and a number consists of one digit only. Thus, each *token* consists of exactly *one* character.

Write a function *parse* to transform an expression (given as a string) into a tree of type *BinTree a b* with adequate choices for *a* and *b*. Numbers have to be transformed into *Int*s by the function *read*.

b. In this part expressions may also contain variables that consist of a single letter. As before, each character is a token by itself. However, there now are two sorts of leaves in the parse tree: numbers and variables. That has to be expressed in the choice for *a* and/or *b* in *BinTree a b* (*Hint*: use the type constructor *Either*).

Exercise 3. This exercise is about *tokenization* (or *scanning*) by defining several *finite state machines* (*fsa*).

a. Define an appropriate type for the states that your *fsa* may take, and decide which states to use as *start state* and as *success state(s).*

b. Write an *fsa* for each of the following:

- *numbers* consisting of a sequence of digits, possibly containing a decimal point, You may assume that numbers are positive (challenge: also treat negative numbers, where you may assume that the negation sign is indicated by '$\sim$' to distinguish it from the binary operation '$-$'),
- *identifiers* consisting of letters and/or digits, but starting with a letter,
- *operators* consisting of one or more operation symbols such as $*$, $++$, $>=$,
- *brackets* '(' and ')'
- *white space* consisting of spaces (`tab` and `enter` need not be considered).

c. Write a *tokenizer* (*scanner*) to split a string into a list of *tokens*, using the *fsa*s from part `b`. Note that the first character of a token determines which *fsa* has to be used.

Exercise 4. Write a variant of the *parse* function that producses a tree for an expression in which numbers and identifiers may consist of several characters as described above. Besides, an expression may contain white space. However, you may staill assume that every compound expression is surrounded by brackets, so that priority of operators is irrelevant.

Exercise 5. Write a function *eval* which calculates the value of an expression by first parsing it and then evaluating the resulting tree. An expression may contains variables, so in order to calculate the value of an expression, you'll need to give a value to each variable by some *valueOf* construction. That can be done, e.g., by defining a *function* which gives a value to each variable, or by defining a *lookup table* (as a list of variable-value pairs). The function *eval* should have this function or lookup table as an extra argument.