Informatics 2A 2015–16
Tutorial Sheet 4 (Week 6)

John Longley

This week's exercises concern material from Lectures 12 and 13 on formal languages, and Lecture 14 on morphology parsing for natural languages.

1. Recall the grammar for mathematical regular expressions from last week's sheet:

$$\begin{array}{rcl} \mathsf{RegExp} & \to & \mathsf{Atom} \mid \mathsf{RegExp} + \mathsf{RegExp} \mid \mathsf{RegExp}\ \mathsf{RegExp} \\ & & \mid \mathsf{RegExp}* \mid (\mathsf{RegExp}) \\ \mathsf{Atom} & \to & \mathsf{sym} \mid \boldsymbol{\emptyset} \mid \boldsymbol{\epsilon} \end{array}$$

   Using methods from Lecture 13, construct an LL(1) grammar for the language of regular expressions that is equivalent to the grammar given earlier. Your grammar should embody the usual precedence conventions for regular expressions: $*$ takes precedence over concatenation, which takes precedence over $+$. (If you are in doubt as to whether your grammar is LL(1), see whether you can construct a parse table for it.)

2. The *concrete syntax* for Micro-Haskell (MH) types (see Lecture 12 and Assignment 1) is specified by the LL(1)-grammar:

$$\begin{array}{rcl} \mathsf{Type} & \to & \mathsf{Type1}\ \mathsf{TypeOps} \\ \mathsf{TypeOps} & \to & \epsilon \mid \texttt{->}\ \mathsf{Type} \\ \mathsf{Type1} & \to & \texttt{Integer} \mid \texttt{Bool} \mid (\ \mathsf{Type}\ ) \end{array}$$

   with start symbol Type. The *abstract syntax* is specified by the grammar

$$\mathsf{Type} \to \texttt{Integer} \mid \texttt{Bool} \mid \mathsf{Type}\ \texttt{->}\ \mathsf{Type}$$

   (a) Write out the concrete parse tree for type expression:

```
Integer -> Bool -> Integer
```

   (b) Write out the abstract syntax tree that the above parse tree will be converted to.

   Consider the following grammar for an abstract syntax of arithmetic expressions (involving only the binary operations $-$ and $+$).

$$\mathsf{Exp} \to \mathsf{n} \mid \mathsf{Exp} + \mathsf{Exp} \mid \mathsf{Exp} - \mathsf{Exp}$$

   where (as in the last tutorial) n stands for some lexical class of *numeric literals*.

   (c) Give an LL(1)-grammar for a corresponding concrete syntax of bracketed arithmetic expressions.

(d) Write out the concrete parse tree for the arithmetic expression:

$$10 - 5 - 4$$

(e) Write out the abstract syntax tree that the above parse tree should be converted to.

(f) What difference arises in the process of translating concrete parse trees to abstract syntax trees between the case of MH types and that of arithmetic expressions? What is the origin of this difference?

3. In written English, the *E-deletion* rule states that when a suffix beginning with *e* or *i* is added to a stem ending in *e*, the (first) *e* is deleted. Examples: *love^ing* ⇒ *loving*, *queue^ed* ⇒ *queued*. (There are some exceptions to the rule, e.g. *canoeing*, but don't worry about dealing with these.)

   Design a finite-state transducer that applies this rule, in the same style as the transducer for E-insertion in the lecture slides. The inputs to the transducer should be strings over $\{a, \ldots, z, \hat{}, \#\}$, where # denotes a word boundary and ^ a morpheme boundary. The outputs should consist of words separated by spaces, using the alphabet $\{a, \ldots, z, \mathrm{SPC}\}$. For simplicity, you may assume that each word consists of at most two morphemes.

   You should take care to work within the framework of non-deterministic transducers as defined in the lectures, in which each transition may be associated with (at most) one input and one output character. If your transducer involves a large number of states, you need only draw a representative sample to illustrate the general pattern.