Informatics 2A: Tutorial Sheet 4 - SOLUTIONS

John Longley

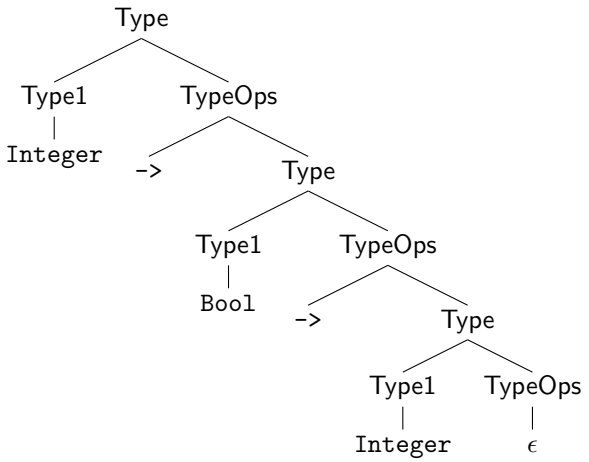
1. A possible LL(1) grammar is as follows (note the distinction between $\boldsymbol{\epsilon}$ and ϵ!):

$$\begin{array}{rcl}
\mathsf{RegExp} & \rightarrow & \mathsf{RegExp1\ PlusOps} \\
\mathsf{PlusOps} & \rightarrow & \epsilon \mid +\,\mathsf{RegExp1\ PlusOps} \\
\mathsf{RegExp1} & \rightarrow & \mathsf{RegExp2\ ConcatOps} \\
\mathsf{ConcatOps} & \rightarrow & \epsilon \mid \mathsf{RegExp2\ ConcatOps} \\
\mathsf{RegExp2} & \rightarrow & \mathsf{RegExp3\ StarOps} \\
\mathsf{StarOps} & \rightarrow & \epsilon \mid *\ \mathsf{StarOps} \\
\mathsf{RegExp3} & \rightarrow & \mathsf{Atom} \mid (\ \mathsf{RegExp}\ ) \\
\mathsf{Atom} & \rightarrow & \mathsf{sym} \mid \boldsymbol{\emptyset} \mid \boldsymbol{\epsilon}
\end{array}$$

For the sake of completeness, the parse table for this is:

	sym/$\boldsymbol{\emptyset}$/$\boldsymbol{\epsilon}$	+	*	(	)	$
RegExp	RegExp1 PlusOps			RegExp1 PlusOps		
PlusOps		+RegExp1 PlusOps			ϵ	ϵ
RegExp1	RegExp2 ConcatOps			RegExp2 ConcatOps		
ConcatOps	RegExp2 ConcatOps	ϵ		RegExp2 ConcatOps	ϵ	ϵ
RegExp2	RegExp3 StarOps			RegExp3 StarOps		
StarOps	ϵ	ϵ	* StarOps	ϵ	ϵ	ϵ
RegExp3	Atom			(RegExp)		
Atom	sym/$\boldsymbol{\emptyset}$/$\boldsymbol{\epsilon}$					

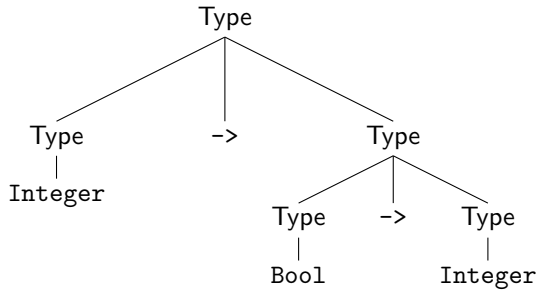
2. (a) The parse tree is:



   (b) By *abstract syntax tree* we mean the intended parse tree of the abstract syntax grammar. In this case it is the parse tree that implictly brackets the type as

```
Integer -> (Bool -> Integer)
```

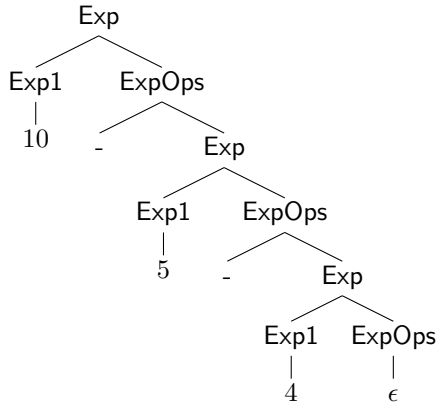
which is the correct bracketing by the Haskell convention that the `->` operation associates to the right. The required tree is:



(c)

$$\begin{array}{rcl} \mathsf{Exp} & \rightarrow & \mathsf{Exp1\ ExpOps} \\ \mathsf{Exp1} & \rightarrow & \mathsf{n} \mid \mathsf{(\ Exp\ )} \\ \mathsf{ExpOps} & \rightarrow & \epsilon \mid +\ \mathsf{Exp} \mid -\ \mathsf{Exp} \end{array}$$

(d) The parse tree is:

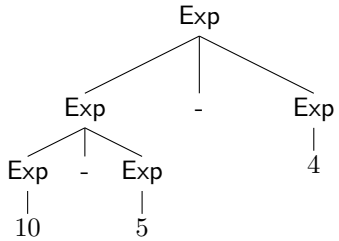


(It's OK to put `n` in place of the numbers. We can view `n` as a terminal in the grammar, in which case `n` should label the parse tree; or, as above, we can view `n` as standing for the class of all numeric terminals.)

(e) This time, standard conventions require the abstract syntax tree that implictly brackets the expression as

$$(10 - 5) - 4$$

This leads to the abstract syntax tree below.

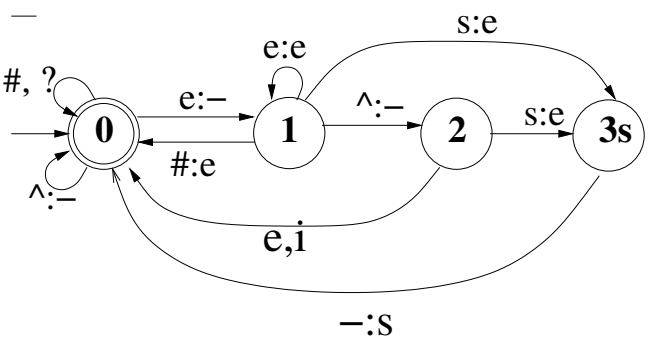


(f) The difference is that structurally similar concrete parse trees are dealt with in different ways in the translation to abstract syntax tree. The concrete syntax does not distinguish between right-associativity and left-associativity of binary operations. Rather, irrespective of the associativity properties of the operations, expressions are concretely parsed as a *list* of top-level operations. The right-associativity of `->` is catered for by translating concrete parse trees to abstract syntax trees in one way. The left-associativity of + and − is handled by translating in a different way.

The main point to take away from this is: when using LL(1) parse technology, the choice of whether to implement infix operations as left-associative or right-associative is made in the translation from concrete syntax to abstract syntax, it is not made in the formulation of the grammar for the concrete syntax.

3. The following transducer does the job. In the state '3s', we have used s as a sample letter distinct from i and e; the machine should contain a state like this (with similar transitions in and out) for every such letter.

   We write '?' to mean 'any letter except e'. For typesetting reasons, we have written '–' in place of ϵ.



   The state 3s illustrates a possible approaches to solving the following problem: given any input letter X other than e or i, output e followed by X. (In practice, one would probably adopt a slightly different approach to this problem: e.g. use a version of finite state transducers that allows for a bit of lookahead.)

   [Actually, we do also want a state 3i which I haven't shown: this one will have a transition from state 1 but not from state 2.]