Informatics 2A: Tutorial Sheet 4 - SOLUTIONS

Mary Cryan

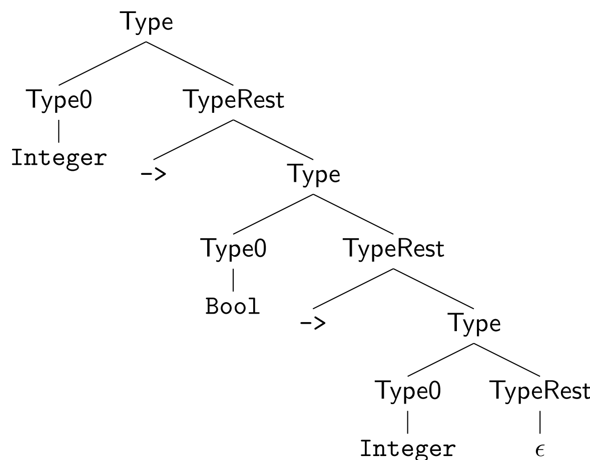
1. A possible LL(1) grammar is as below (note the distinction between $\boldsymbol{\epsilon}$ and ϵ!). It's wise to also talk to them about the similarity between this and what we did to get an LL(1) grammar for the arithmetic expressions:

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

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/$\boldsymbol{\emptyset}$/$\boldsymbol{\epsilon}$					

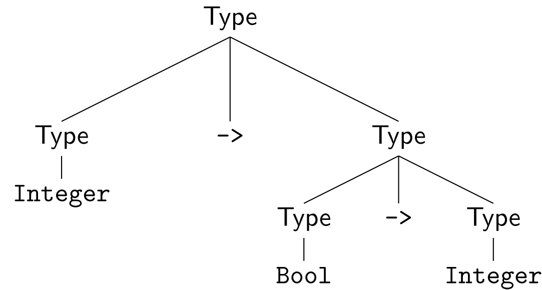
2. (a) The parse tree is:



   (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)
```

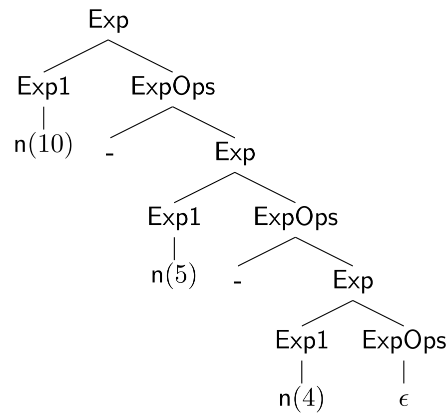
which is the correct bracketing by the Haskell convention that the `->` operation associates to the right. The required tree is:



(c)

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

(d) The parse tree is:

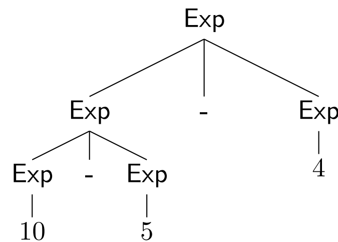


(At the numeral leaves, the labels on the parse tree are strictly speaking just n, but it's informative to display the actual lexical tokens in question.)

(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. This question is really to get the students to think about what they have learned in the "Fixing Grammars" lecture. The actual "work" they need to do to answer this is very minor.

   They will remember the algorithm of Lecture 13 which takes a CFG and converts it to Chomsky Normal Form. This has 4 steps: (i) removing ϵ-productions, (ii) removing unit-productions $X \to Y$ where Y is non-terminal, (iii) removing terminal symbols from "mixed" right-hand sides of production rules (by adding dummy non-terminals) and (iv) breaking up long (at least three) right-hand sides by introducing more dummy non-terminals.

   We only need to carry out the first two of these steps in order to get an equivalent CFG for $\mathcal{G}$ which is cycle-free (see those steps on slide 14 of lecture 13 if you want to discuss more about it). Also, if the original grammar generated the empty string, we add an extra non-terminal $\widehat{S}$ which becomes the new start non-terminal, and two extra production rules

$$\widehat{S} \to \epsilon \mid S.$$

   Note that after having carried out step (i) of CNF, we are assured that every right-hand side of every terminal is non-empty, and for each non-terminal X in N, there is no derivation $X \Rightarrow^* \epsilon$ (we make a note of whether S could derive the empty string, and will correct for this at the end). However, after (i) we may still have productions which have a single non-terminal on the right-hand side, and these may give rise to "cycles" as defined in the question.

   Step (ii) of CNF conversion eliminates all unit production rules with a non-terminal on the right-hand side, resulting in a grammar where every production rule is either $X \to a$ for a single non-terminal, or alternatively is $X \to x_1 \ldots x_k$ for $k \geq 2$. Since the non-terminals no-longer can generate an empty string, this "at least two symbols" constraint on the right-hand-side means we can never generate $X \Rightarrow^+ X$ by introducing initial non-terminals and then "vanishing" them.

   The adjustment given above for the case where $\epsilon \in L(\mathcal{G})$ does not induce a cycle (since we use a dummy start non-terminal for this).