Informatics 2 – Introduction to Algorithms and Data Structures

Solutions for Tutorial 8

1. We are studying the following simple context-free grammar for arithmetic expressions from Lecture 22. The start symbol is Exp.

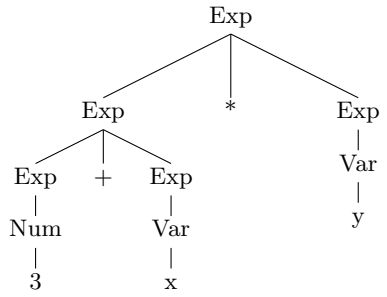
$$\begin{array}{rcl} \mathsf{Exp} & \to & \mathsf{Var} \mid \mathsf{Num} \mid (\ \mathsf{Exp}\ ) \\ \mathsf{Exp} & \to & \mathsf{Exp} + \mathsf{Exp} \\ \mathsf{Exp} & \to & \mathsf{Exp} * \mathsf{Exp} \\ \mathsf{Var} & \to & x \mid y \mid z \\ \mathsf{Num} & \to & 0 \mid \cdots \mid 9 \end{array}$$

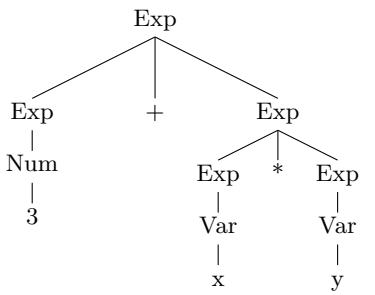
   (a) We need to draw all possible syntax trees for each of the following three strings.

$$3 + x * y \qquad 3 + (x * y) \qquad z + 10$$

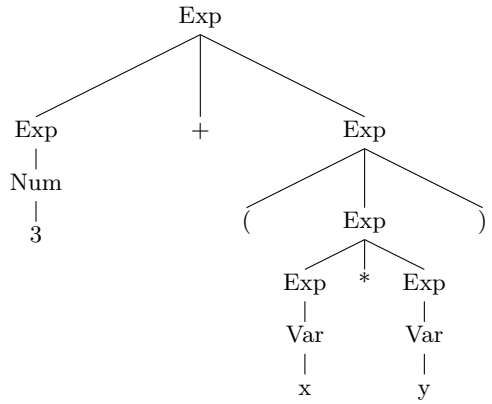
   answer:

   3 + x * y has two trees:





   3 + (x * y) has just one tree:



z + 10 has no trees. This is not a sentence of the language: our grammar doesn't cater for multi-digit numerals like 10.

(b) We need to design a new context-free grammar that generates exactly the same language as the one above, but with the property that it is *unambiguous*: every string in the language should have exactly one syntax tree. The grammar should enforce the familiar convention that * takes precedence over +.

answer: First for the grammar with the clause for $*$ omitted: The key observation is that a general expression is a list of one or more 'simple expressions', separated by +. Drawing inspiration from the comma list example, the following grammar does the trick:

$$\begin{array}{rcl} \mathsf{Exp} & \rightarrow & \mathsf{SimpleExp\ PlusList} \\ \mathsf{SimpleExp} & \rightarrow & \mathsf{Var} \mid \mathsf{Num} \mid (\ \mathsf{Exp}\ ) \\ \mathsf{PlusList} & \rightarrow & \epsilon \mid +\ \mathsf{SimpleExp\ PlusList} \end{array}$$

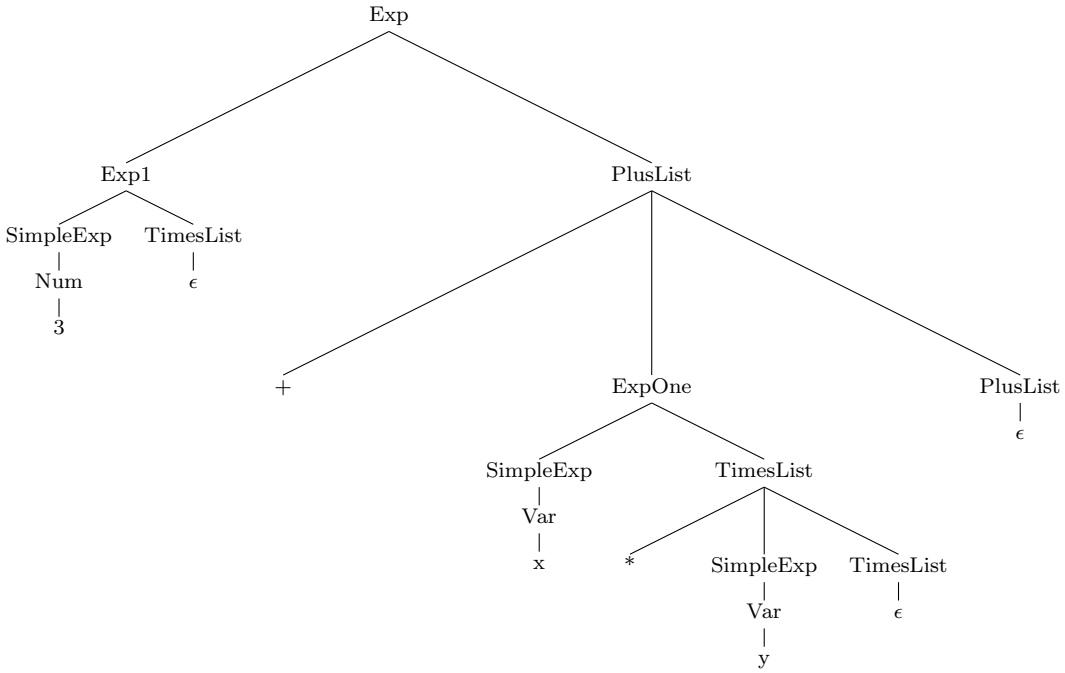
(with the same rules for Var and Num as before). Intuitively, we here distinguish two 'levels' of expressions, corresponding to Exp and SimpleExp. To cater for $*$ as well, and to enforce the precedence rule, we can extend this idea to allow three levels:

$$\begin{array}{rcl} \mathsf{Exp} & \rightarrow & \mathsf{Exp1\ PlusList} \\ \mathsf{Exp1} & \rightarrow & \mathsf{SimpleExp\ TimesList} \\ \mathsf{SimpleExp} & \rightarrow & \mathsf{Var} \mid \mathsf{Num} \mid (\ \mathsf{Exp}\ ) \\ \mathsf{TimesList} & \rightarrow & \epsilon \mid *\ \mathsf{SimpleExp\ TimesList} \\ \mathsf{PlusList} & \rightarrow & \epsilon \mid +\ \mathsf{Exp1\ PlusList} \end{array}$$

(plus the usual Var and Num rules). Other solutions are possible, but the above grammar turns out to be particularly well-adapted to 'left-to-right parsing'.)

(c) For the grammar we have designed in part (b), we must draw the *unique* syntax tree for any of the strings from part (a) that originally had more than one syntax tree.

answer: The unique syntax tree for 3 + x * y is now:



Here we include explicit ϵ's for clarity, though of course they contribute nothing to the string in question.

2. Consider the following context free grammar with start symbol S:

S	→	NP VP	PP	→	Pre NP
S	→	*I* VP PP	V	→	*ate*
NP	→	Det N	Det	→	*the* | *a*
VP	→	*ate* NP	N	→	*fork* | *salad*
VP	→	V	Pre	→	*with*

(a) *Convert this grammar to Chomsky Normal Form (see Lecture 23).*

Here we follow the order and numbering of steps given in the lecture slides and live lecture. (This is slightly different from the order in the video lecture from 2021. In practice, there is a lot of flexibility and the order of steps doesn't matter much for small examples.)

Step 1: Eliminate the ternary rule S → *I* VP PP. We can do this by introducing a fresh non-terminal *X* and replacing the rule by S → *I* X and X → VP PP.

Steps 2 and 3: There are no ϵ-rules, so these steps can be omitted. (In general, ϵ-rules are more typical of formal language grammars than natural language ones.)

Step 4: We deal with the unit rule VP → V by replacing it with VP → *ate*.

Step 5: We introduce a separate non-terminal for each terminal, which we'll do by writing the same word capitalized in Roman font (e.g. I, Ate, The). We also add corresponding expansion rules (e.g. I → *I*; Ate → *ate*) and then replace all terminals within non-unary right-hand sides by the corresponding non-terminal (e.g. S → I VP PP; VP → Ate NP).

Discarding the rules for non-terminals now unreachable from S (e.g. V, The), the resulting grammar is now as follows:

S	→	NP VP	I	→	*I*
PP	→	Pre NP	Ate	→	*ate*
S	→	I X	Det	→	*the* | *a*
X	→	VP PP	N	→	*fork* | *salad*
NP	→	Det N	Pre	→	*with*
VP	→	Ate NP	V	→	*ate*
VP	→	*ate*			

(Other minor variations are of course acceptable, provided they are indeed in CNF and are equivalent to the original grammar.)

(b) *Use the CYK algorithm from Lecture 23 to parse 'I ate the salad with a fork'.* Using the above CNF grammar, the CYK chart would be:

	1	2	3	4	5	6	7
0	I						S
1		V,VP,Ate		VP			X
2			Det	NP			
3				N			
4					Pre		PP
5						Det	NP
6							N

(c) *How many complete analyses of the sentence do you get?* Just the one:

(S (I *I*) (X (VP (V *ate*)(NP(Det *the*)(N *salad*)))
(PP (Pre *with*)(NP (Det *a*) (N *fork*)))))

(d) *Now add a further production rule to your CNF grammar to allow for the alternative prepositional phrase attachment, i.e. 'the salad with a fork'. Revise your CYK chart or graph to include any new entries this introduces.*

(e) We could add a new rule

$$\text{NP} \rightarrow \text{NP PP}$$

This would add an NP entry to cell (2,7), hence a VP entry to (1,7). However, no new S entry would be added to (0,7), so there is still just one complete parse. This is because the grammar lacks the means to derive *I* from NP.

3. The grammar:

Terminals:	(,), $*$, n
Nonterminals:	Exp, Ops
Productions:	Exp → n Ops | (Exp)
	Ops → ϵ | $*$ n Ops
Start symbol:	Exp

The parse table:

	(	)	$*$	n	$
Exp	(Exp)			n Ops	
Ops		ϵ	$*$ n Ops		ϵ

(a) *Using this table, apply the LL(1) parsing algorithm to the input* (n $*$ n).

Operation	Input remaining	Stack state
	(n * n)$	Exp
Lookup (,Exp	(n * n)$	(Exp)
Match (	n * n)$	Exp)
Lookup n, Exp	n * n)$	n Ops)
Match n	* n)$	Ops)
Lookup *, Ops	* n)$	* n Ops)
Match *	n)$	n Ops)
Match n	)$	Ops)
Lookup), Ops	)$	)
Match)	$	STACK EMPTIES
		AT END OF STRING:
		SUCCESS!

(b) *For each of the following three input strings, explain how and where an error arises in the course of the LL(1) parsing algorithm. In each case, suggest a reasonable error message.*

( ) n) n *

- For ( ), the parser will encounter a blank table entry at), Exp.
  Message: ") Found where expression expected."
- For n), the stack will empty before end of input is reached.
  Message: ") Found after end of expression."
- For n *, the end of input will be reached with n Ops still on the stack, and the parser gets stuck since the top of the stack is a terminal n no different from $.
  Message: "End of input found where numeric literal expected."