Introduction to Algorithms and Data Structures

Tutorial 8

your tutor

University of Edinburgh

24-28th February, 2025

Q1: Context-Free Grammars

This is the context-free grammar for arithmetic expressions from Lecture 22. The start symbol is Exp.

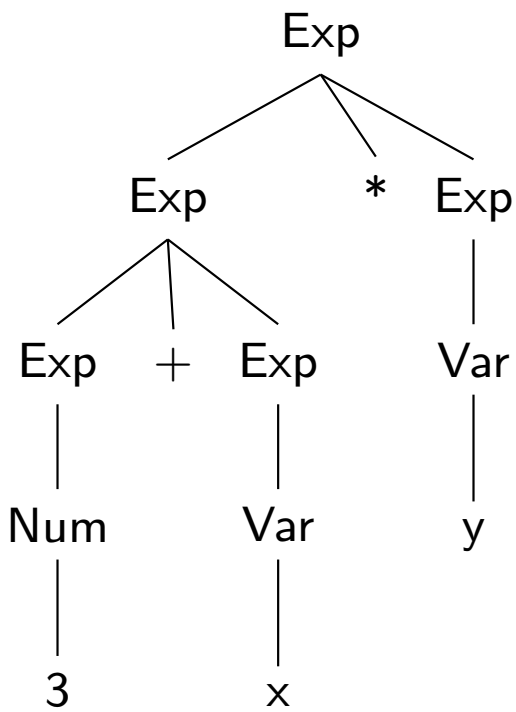
$$
\begin{array}{rcl}
\text{Exp} & \rightarrow & \text{Var} \mid \text{Num} \mid (\ \text{Exp}\ ) \\
\text{Exp} & \rightarrow & \text{Exp} + \text{Exp} \\
\text{Exp} & \rightarrow & \text{Exp} * \text{Exp} \\
\text{Var} & \rightarrow & x \mid y \mid z \\
\text{Num} & \rightarrow & 0 \mid \cdots \mid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

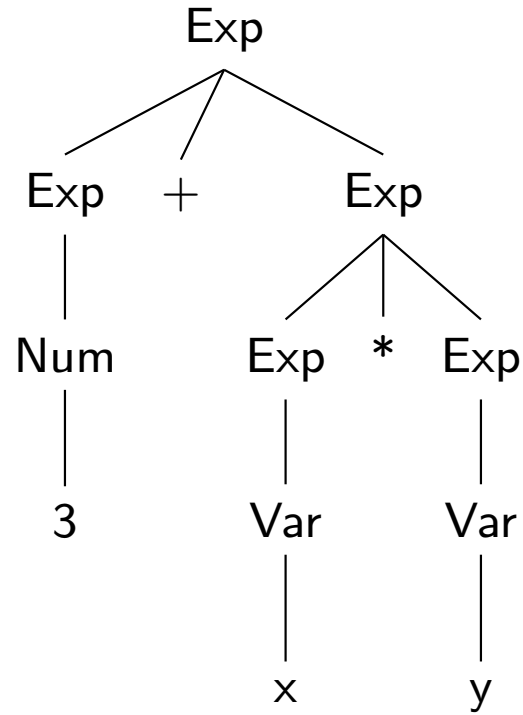
(a) Want *all* syntax trees for each of the following three strings:

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

Q1 (a) syntax trees for $3 + x * y$ - two trees

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

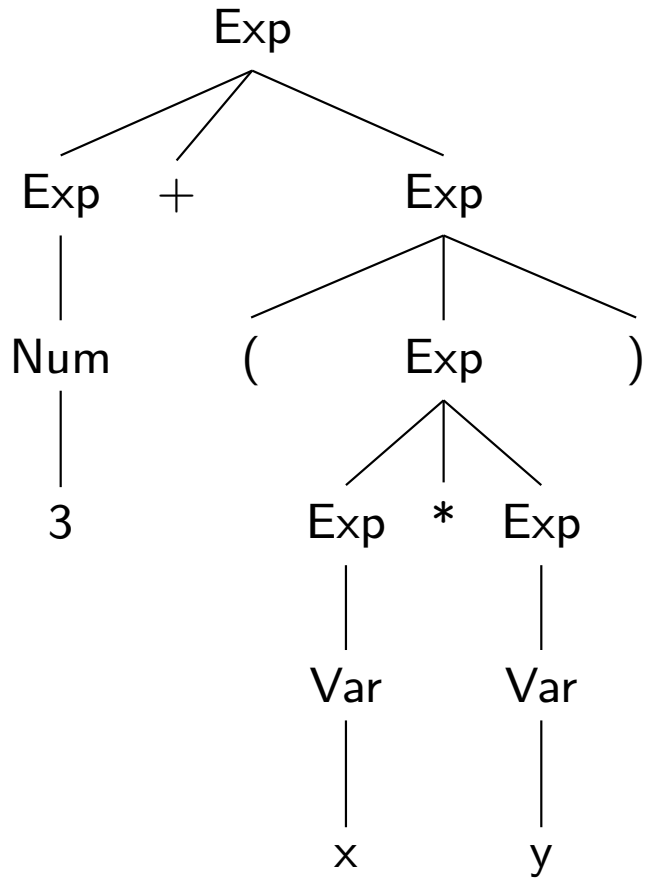




Q1 (a) syntax trees for $3 + (x * y)$ and $z + 10$

$3 + (x * y)$, just one tree:

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



$z + 10$, no trees: Grammar doesn't cater for multi-digit numerals like 10.

Q1 (b) - unambiguous CFG (for same language)

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

Want to transform to new CFG that

- generates *exactly* the same language.
- *unambiguous*: every string in the language will have *just one* syntax tree.
- Enforce the familiar convention that * takes precedence over $+$.

Where does the ambiguity come from? Not from Num or Var

Q1 (b) - unambiguous CFG (for same language)

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

Ignore the clause for $*$ for the first pass.

Otherwise, an Exp is a list of one or more 'simple expressions' (a Num or Var, or even a parenthesised expression (Exp), separated by $+$ operators.

(with inspiration from the "comma list" example), we get (Var and Num same):

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

Q1 (b) - unambiguous CFG (for same language)

Now cater for $*$. This is to have *higher precedence* meaning "done first".

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

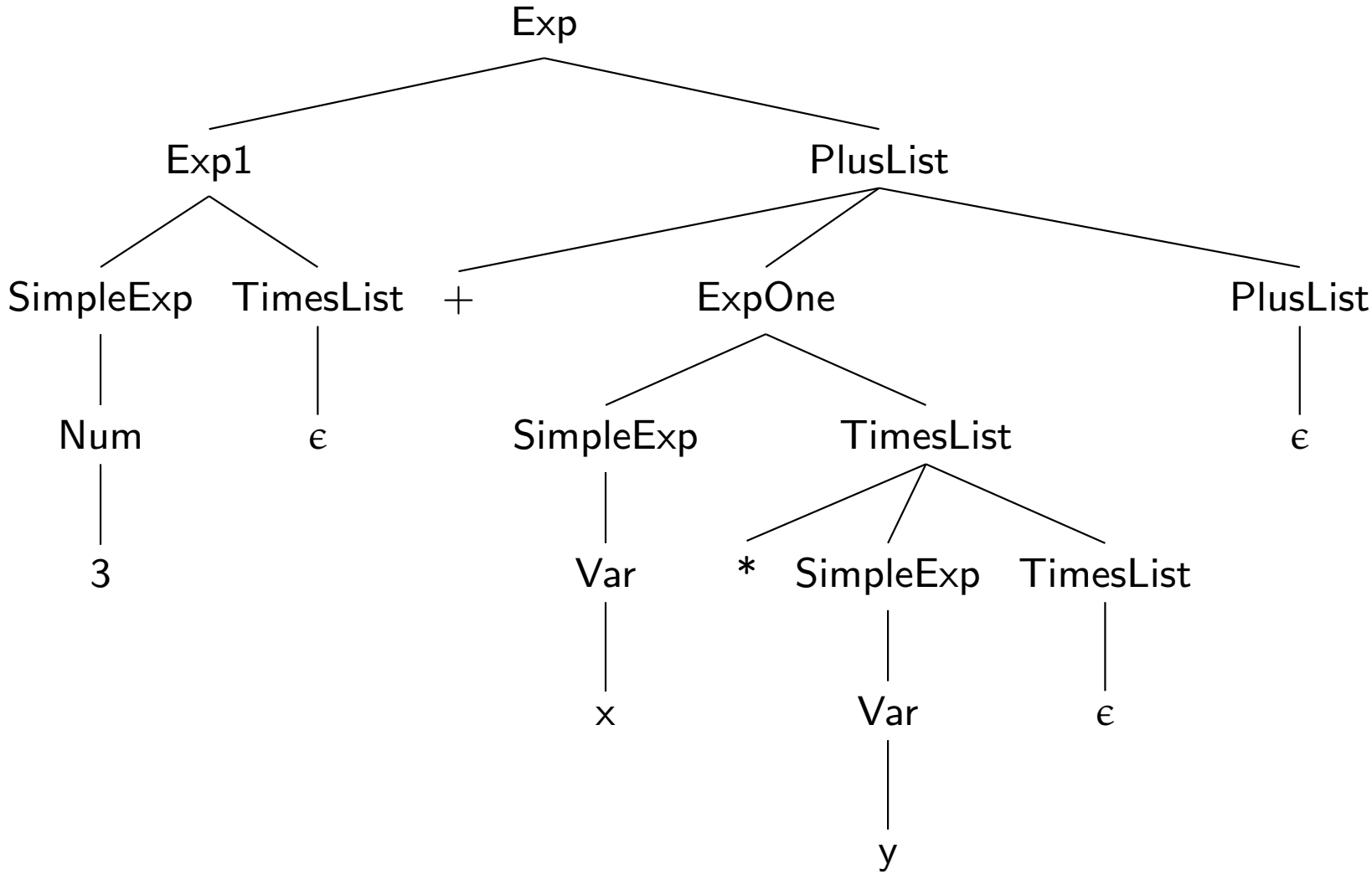
(Var and Num same as before)

Try out $3 + x * y$ to see it has a unique tree.

Not the only way to remove ambiguity for the given grammar ... this particular formulation turns out to be particularly well-adapted to 'left-to-right parsing'.

Q1 (c) - reconsider $3 + x * y$

answer: The unique syntax tree for 3 + x * y in the new grammar is:



We include explicit ϵ's for clarity, though they contribute nothing to this string.

Q2: CNF and CYK

Consider the following context free grammar with start symbol S:

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

Our aim is to run the CYK algorithm for parsing.

(a) Convert to CNF (first step)
Notice that this grammar has no ϵ-production rules.
$\Rightarrow$ can shorten our conversion procedure, no need for Step 2 or Step 3.

Q2: (a) CNF conversion

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

Step 1: Deal with right-hand-sides with ≥ 3 non-terminals.

Only S $\rightarrow$ *I* VP PP.

Introduce "intermediate" non-terminal X, and replace rule with
S $\rightarrow$ *I* X, X $\rightarrow$ VP PP.

Q2: (a) CNF conversion

S	$\rightarrow$	NP VP	PP	$\rightarrow$	Pre NP
S	$\rightarrow$	*I* X	X	$\rightarrow$	VP PP
V	$\rightarrow$	*ate*			
NP	$\rightarrow$	Det N	Det	$\rightarrow$	*the* | *a*
VP	$\rightarrow$	*ate* NP	N	$\rightarrow$	*fork* | *salad*
VP	$\rightarrow$	V	Pre	$\rightarrow$	*with*

Step 4: Eliminate any unit productions (with a non-terminal on the right).

Only such production is VP $\rightarrow$V.

We eliminate VP $\rightarrow$V by simply replacing it with VP $\rightarrow$*ate*.
(that's the only possible substitution for V)

Q2: (a) CNF conversion

S	$\rightarrow$	NP VP	PP	$\rightarrow$	Pre NP
S	$\rightarrow$	*I* X	X	$\rightarrow$	VP PP
V	$\rightarrow$	*ate*			
NP	$\rightarrow$	Det N	Det	$\rightarrow$	*the* | *a*
VP	$\rightarrow$	*ate* NP	N	$\rightarrow$	*fork* | *salad*
VP	$\rightarrow$	V	Pre	$\rightarrow$	*with*
VP	$\rightarrow$	ate			

Step 5: Eliminate terminals in any right-hand side with ≥ 2 symbols.

This is only *ate* in VP $\rightarrow$ *ate* NP, and *I* in S $\rightarrow$ *I* VP PP.

Change to rules:

S	$\rightarrow$	*I* X	$\Rightarrow$	S	$\rightarrow$	I X
				I	$\rightarrow$	*I*
VP	$\rightarrow$	*ate* NP	$\Rightarrow$	VP	$\rightarrow$	Ate NP
				Ate	$\rightarrow$	*ate*

note: We don't *really* need to introduce "Ate" here, as we already have V $\rightarrow$ *ate* and V will no longer be involved in other rules. Not usually the case.

Q2: (a) CNF conversion

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

S	$\rightarrow$	NP VP	I	$\rightarrow$	*I*
PP	$\rightarrow$	Pre NP	Ate	$\rightarrow$	*ate*
S	$\rightarrow$	I X	Det	$\rightarrow$	*the* | *a*
X	$\rightarrow$	VP PP	N	$\rightarrow$	*fork* | *salad*
NP	$\rightarrow$	Det N	Pre	$\rightarrow$	*with*
VP	$\rightarrow$	Ate NP	VP	$\rightarrow$	*ate*

Q2: (b) CYK chart

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		VP,Ate		VP			X
2			Det	NP			
3				N			
4					Pre		PP
5						Det	NP
6							N

Q2: (c) How many complete analyses?

How many complete analyses of "I ate the salad with a fork"?
Just the one:

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

Q2: (d) Expanding the CNF grammar

Now add a further production rule to your CNF grammar to allow for 'the salad with a fork'. Revise the CYK chart.

We could add a new rule

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

- This additional rule allows us to match "the salad with a fork" to NP ("the salad" to the NP on rhs, "with a fork" to PP on rhs")

  We can add an NP entry to cell (2,7), and 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.

Q3: LL(1) Predictive Parsing

Consider the following LL(1) grammar:

Terminals:	$($, $)$, $*$, n
Nonterminals:	Exp, Ops
Productions:	Exp $\rightarrow$ n Ops $\mid$ (Exp)
	Ops $\rightarrow$ ϵ $\mid$ $*$ n Ops
Start symbol:	Exp

The corresponding parse table:

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

Q3: LL(1) Predictive Parsing

(a) Using the parsing table, apply the LL(1) parsing algorithm to the input:

$$(\; n * n \;)$$

Operation	Input remaining	Stack state
	(n * n)$	Exp

Q3: LL(1) Predictive Parsing

(a) Using the parsing 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)

Q3: LL(1) Predictive Parsing

(a) Using the parsing 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)

Q3: LL(1) Predictive Parsing

(a) Using the parsing 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)

Q3: LL(1) Predictive Parsing

(a) Using the parsing 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)

Q3: LL(1) Predictive Parsing

(a) Using the parsing 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)

Q3: LL(1) Predictive Parsing

(a) Using the parsing 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)

Q3: LL(1) Predictive Parsing

(a) Using the parsing 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)

Q3: LL(1) Predictive Parsing

(a) Using the parsing 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	)$	)

Q3: LL(1) Predictive Parsing

(a) Using the parsing 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!

Q3: LL(1) Predictive Parsing

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

(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 $*$

Q3: LL(1) Predictive Parsing

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

(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."

Q3: LL(1) Predictive Parsing

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

(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 n): the stack will empty before end of input is reached.
  Message: "") Found after end of expression."

Q3: LL(1) Predictive Parsing

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

(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 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."