CS331 Compiler Design
Spring 2003

FINAL EXAM

GRADE	POINT RANGE	DISTRIBUTION
A	82 – 100	3
B	71 – 81	4
C	60 – 71	3
D	53 – 59	1
F	0 – 50	0
AVERAGE SCORE		74
STANDARD DEVIATION		10.5

1. Consider the following grammar:

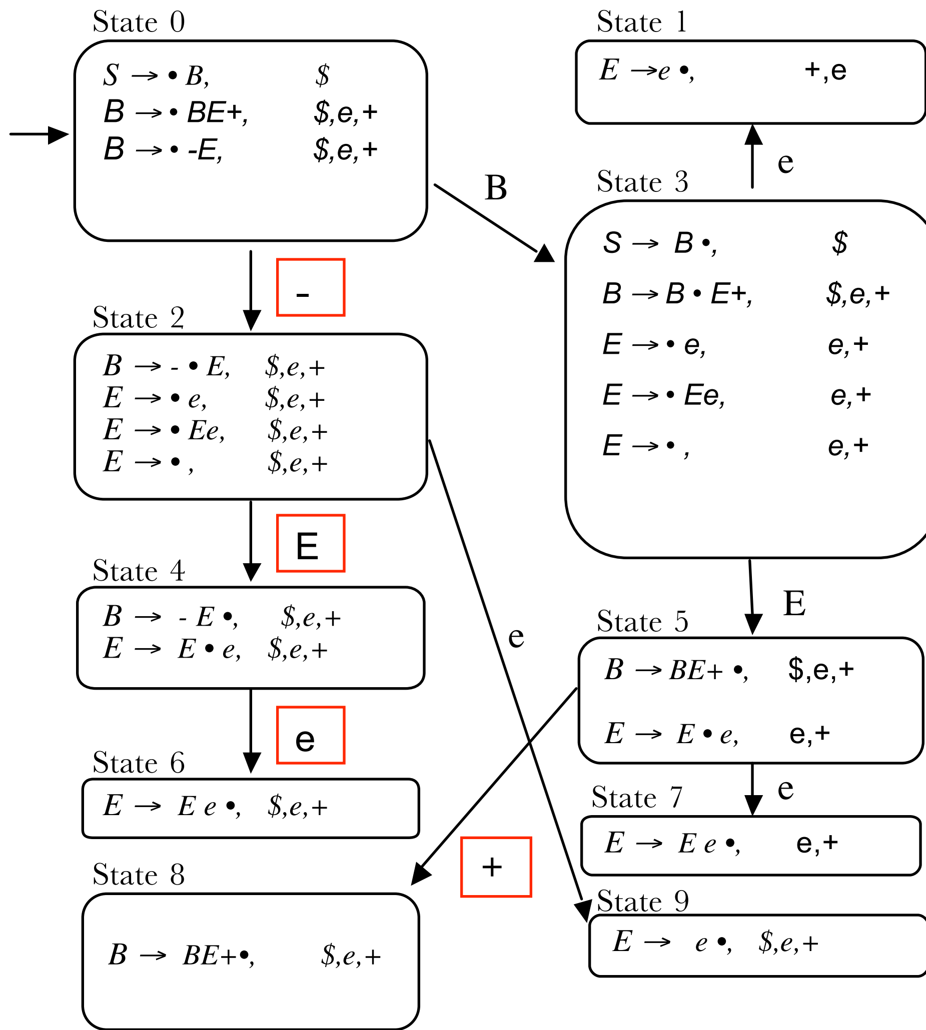
$$
\begin{aligned}
S &\rightarrow B \\
B &\rightarrow B\ E\ + \\
&\ \ \mid\ -\ E \\
E &\rightarrow e \\
&\ \ \mid\ E\ e \\
&\ \ \mid\ \varepsilon
\end{aligned}
$$

Below is a partial DFA for the grammar above. Notice we did not have to add a new start state (i.e., augment the grammar) as is usually done for LR(1) parsers.

(a) Complete state 0 by performing closure on the item listed. Don't forget to include lookahead symbols for the new items.

(b) Fill in all elements of states 3 and 8, and the lookahead items in states 1, 5 and 7.

(c) Fill in the missing transition labels (4 of them).



(d) Fill in the following table with the reduce productions and lookahead tokens for the given states. Write "not a reduce state" if that state does not have a reduction action.

State	Production	Lookahead tokens
0	not a reduce state	
2	$E \rightarrow \varepsilon$	$,e,+
4	$B \rightarrow - E$	$,e,+

(e) For each state with a shift-reduce conflict, list the state, the lookahead token and the production that are in conflict. Write "none" if there are no shift-reduce conflicts in any state.

State	Production	Lookahead token
2	$E \rightarrow \varepsilon$	e
3	$E \rightarrow \varepsilon$	e
4	$B \rightarrow - E$	e

(f) For each state with a reduce-reduce conflict, list the state, and the conflicting productions. Write "none" if there are no reduce-reduce conflicts. Answer: none

(g) For each of the following configurations of the LR(1) parser, say what action is taken by the parser (one of "shift", "reduce with production ...", "error" or "accept") and write the next configuration (except in the case of an accept or error). You can assume that in the case of a shift-reduce conflict the parser chooses to shift. The ▷ symbol indicates the current position in scanning the input string.

Configuration	Action	Next configuration
- ▷ e + $	shift	- e ▷ + $
BEe ▷ e + $	reduce $E \rightarrow Ee$	B E ▷ e + $
Be ▷ $	error	

2. Consider the following grammar in which S is the start non-terminal and the productions for the non-terminals A, B, and C are not shown. There are no other productions for S.

$$S \rightarrow AB \mid AC$$
$$A \rightarrow \ldots$$
$$B \rightarrow \ldots$$
$$C \rightarrow \ldots$$

Consider now the LL(1) parsing table for this grammar. What conditions must be satisfied by the First and Follow sets of the non-terminal, such that the row corresponding to S in this table does not contain multiply-defined entries?

Write your answer in the table below considering six separate cases. For example, in the top left entry consider the case where $\varepsilon \notin \text{First}(B)$ and $\varepsilon \notin \text{First}(C)$ and $\text{First}(A) = \{\varepsilon\}$. Write what **additional** conditions must be met, or write "never" if no additional conditions would help.

	$\varepsilon \notin \text{First}(B)$ $\varepsilon \notin \text{First}(C)$	$\varepsilon \notin \text{First}(B)$ $\varepsilon \in \text{First}(C)$	$\varepsilon \in \text{First}(B)$ $\varepsilon \in \text{First}(C)$
$\text{First}(A) = \{\varepsilon\}$	$\text{First}(B) \cap \text{First}(C) = \phi$	$\text{First}(B) \cap \text{First}(C) = \phi$ $\text{First}(B) \cap \text{Follow}(S) = \phi$	Never (since both *AB* and *AC* end up in the column for $)
$\text{First}(A) \neq \{\varepsilon\}$	Never (since both *AB* and *AC* end up in the column for anything in $\text{First}(A) - \{\varepsilon\}$)	never	never

3. Consider the following grammar.

$$S \rightarrow S\, a \mid b \mid \textbf{error}\ a$$

(a) Show the DFA for recognizing viable prefixes of this grammar. Use LR(0) items, and treat **error** as a terminal.

States:

I_0 = { S → .Sa, S → . b, S → . error a }
I_1 = { S → S. a }
I_2 = { S → b. }
I_3 = { S → error . a }
I_4 = { S → error a. }
I_5 = { S → Sa.}

Transitions:

I_0 -S→ I1
I_0 -b→ > I2
I_0 -error→ I3
I_1 -a → I5
I_3 -a→ I4

(b) Some bottom-up parsers use error productions in the following way: When a parsing error is encountered (i.e., the machine cannot shift, reduce, or accept):

a. First, pop and discard elements of the stack one at a time until a stack configuration is reached where the error terminal of an error production can be shifted on to the stack.

b. Second, discard tokens of the input one at a time until one is found that ca on to the stack.

c. Third, resume normal execution of the parser.

Show the sequence of stack configurations of an SLR(1) parser for the grammar above on the following input. Be sure to show all shift, reduce, and discard actions (for both stack and input).

bacadfa

Stack (with top at right)	Input	Action
	bacadfa$	shift b
b	acadfa$	reduce S → b
S	acadfa$	shift a
Sa	cadfa$	error; discard top of stack
S	cadfa$	discard top of stack
	cadfa$	shift error on to stack
error	cadfa$	discard input
error	adfa$	shift a
error a	dfa$	error; discard top of stack
error	dfa$	discard stack
	dfa$	shift error on to stack
error	dfa$	discard input
error	fa$	discard input
error	a$	shift a
error a	$	reduce S → error a
S	$	- accept -

There was no need to build a parse table for this problem. Either the DFA alone or deducing moves directly from the grammar was enough.

4. Consider the following grammar over the alphabet { g, h, i, b }:

$A \rightarrow B\ C\ D$
$B \rightarrow b\ B \mid \varepsilon$
$C \rightarrow C\ g \mid g \mid C\ h \mid i$
$D \rightarrow A\ B \mid \varepsilon$

(a) Fill in the table below with the First and Follow sets for the non-terminals in this grammar:

	First	*Follow*
A	{ g, b, i }	{ \$, b }
B	{ b, ε }	{ \$, g, b, i }
C	{ g, i }	{ \$, g, b, i, h }
D	{ g, b, i, ε }	{ \$, b }

(b) Fill in the column headings, and the row corresponding to B and C in the predictive LL(1) parsing table for this grammar. You may have some entries with more than one production

	g	*h*	*i*	*b*	*\$*
B	ε		ε	bB | ε	ε
C	Cg | Ch | g		Cg | Ch | i		

5. Write a translation scheme that computes the number of occurrences of the symbol a for any $w \in G$. Use only a single synthesized attribute, called *count.*

1	$S_1 \rightarrow aaS_2$	{S_1.count = 2 + S_2.count}
2	$S_1 \rightarrow S_2bb$	{S_1. count = S_2. count }
3	$S_1 \rightarrow aS_2b$	{S_1. count = 1 + S_2. count }
4	$S \rightarrow a$	{S. count = 1}
5	$S \rightarrow b$	{S. count = 0}

If we eliminate rules 1 and 2 from the grammar G above, is the resulting grammar G' LL(1)? Justify your answer.

$$S \rightarrow aSb$$
$$S \rightarrow a$$
$$S \rightarrow b$$

The grammar is not LL(1) since for the nonterminal symbol S the rules 1 and 2 have the property: FIRST(aSb) = FIRST(a) ={a} ≠ Φ. This means that the parse table would have two entries for non-terminal symbol S and input symbol a, thus producing a conflict.

6. Which of LL(1), SLR(1), and LR(1) can parse strings in the following grammar, and why? Justify your response.

$$E \rightarrow A \mid B$$
$$A \rightarrow a \mid c$$
$$B \rightarrow b \mid c$$

The grammar is ambiguous; there are two possible derivations of the string *c*. None of LL(1), SLR(1), or LR(1) is able to parse any ambiguous grammar. While it is possible to work through complete LL(1), SLR(1), and LR(1) constructions to show exactly where they fail, that is not necessary. As soon as you show that a grammar is ambiguous you immediately know that none of LL(1), SLR(1), or LR(1) can possibly work.

7. Consider the following grammar, in which id, =, +, and string are terminals:

$$S' \rightarrow S$$
$$S \rightarrow \text{id} = S \mid S + S \mid \text{id} \mid \text{string}$$

Which of the following are viable prefixes of the language generated by the grammar? For each viable prefix, give a suffix that produces a right-sentential form.

(a) S is a viable prefix; possible suffixes include **$, + id, + string**.

(b) + S + is not a viable prefix

(c) S + id = is a viable prefix; possible suffixes include **id, string, id + id, id + string,** etc.

Note that the question asked for a *right sentential form*, so suffixes including non-terminals are not valid.

8. Consider the following grammar:

$$S \rightarrow ScB \mid B$$
$$B \rightarrow e \mid efg \mid efCg$$
$$C \rightarrow SdC \mid S$$

Give an LL(1) grammar that generates the same language.

First, eliminate left-recursion:

$S \rightarrow B\ S'$
$S' \rightarrow c\ B\ S' \mid \varepsilon$
$B \rightarrow e \mid e\ f\ g \mid e\ f\ C\ g$
$C \rightarrow S\ d\ C \mid S$

Then eliminate common prefixes by left factoring:

$S \rightarrow B\ S'$
$S' \rightarrow c\ B\ S' \mid \varepsilon$
$B \rightarrow e\ B'$
$B' \rightarrow \varepsilon \mid f\ B''$
$B'' \rightarrow g \mid C\ g$
$C \rightarrow S\ C'$
$C' \rightarrow d\ C \mid \varepsilon$

9. Give a grammar with the following First and Follow sets. Your grammar sh two productions per non-terminal and no epsilon productions. The X, Y, Z no and the terminals are a, b, c, d, e, f.

First(X) = $\{b, d, f\}$	Follow(X) = $\{\$\}$
First(Y) = $\{b, d\}$	Follow(Y) = $\{c, e\}$
First(Z) = $\{c, e\}$	Follow(Z) = $\{a\}$
Follow(d) = $\{c, e\}$	Follow(a) = $\{\$\}$
Follow(e) = $\{a\}$	Follow(b) = $\{b, d\}$
Follow(f) = $\{\$\}$	Follow(c) = $\{c, e\}$

There are many ways to solve this problem. The following solution starts by using the First sets to generate the obvious productions that have the correct first (and only) terminal):

$X \rightarrow b \mid d \mid f$
$Y \rightarrow b \mid d$
$Z \rightarrow c \mid e$

To get two productions per non-terminal it is necessary to eliminate an X production. Since b and d are in the first of Y, this would work:

$X \rightarrow Y \mid f$
$Y \rightarrow b \mid d$
$Z \rightarrow c \mid e$

At this point the First sets are correct for the nonterminals; the next step is to modify the productions to have the correct Follow sets as well. Starting arbitrarily with the terminal c, since both c and e are in Follow(C) and in First(Z), Z can be added after c in some production. There is only one choice:

$X \rightarrow Y \mid f$
$Y \rightarrow b \mid d$
$Z \rightarrow cZ \mid e$

Similar reasoning about Follow(b) leads to:

$X \rightarrow Y \mid f$
$Y \rightarrow bY \mid d$
$Z \rightarrow cZ \mid e$

Since $ is not in the follow of Y, there must be something that comes after Y in the X → Y production. Since First(Z) is in the Follow(Y) try:

$X \rightarrow YZ \mid f$
Y -> $bY \mid d$
$Z \rightarrow cZ \mid e$

Since Follow(a) = {$} and Follow(Z) = {a}, a can be added as follows:

$X \rightarrow YZa \mid f$
$Y \rightarrow bY \mid d$
$Z \rightarrow cZ \mid e$

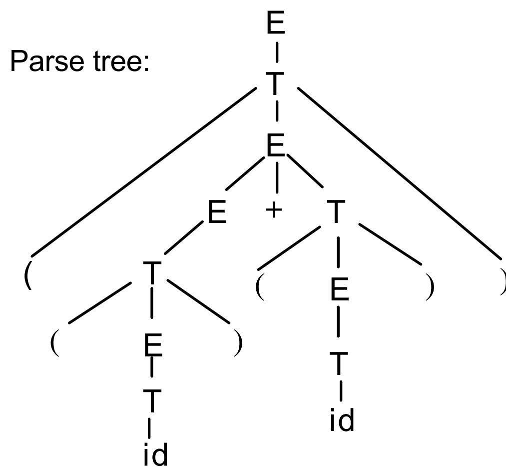
At this point it is routine to check that all the requirements are satisfied.

10. Consider the syntax-directed definition below, in which *mknode* constructs internal syntax tree nodes with two children; *mkleaf* constructs leaf syntax tree nodes, and nptr for E and T keeps track of the pointers returned by the function calls to *mknode* and *mkleaf*.

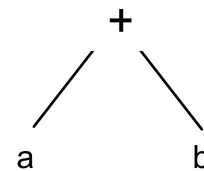
$E \rightarrow E_1 + T$	*E.nptr = mknode('+', E_1.nptr, T.nptr)*
$E \rightarrow T$	*E.nptr = T.nptr*
$T \rightarrow (E_1)$	*T.nptr = E_1.nptr*
$T \rightarrow$ id	*T.nptr := mkleaf(id, id.lexeme)*
$T \rightarrow$ num	*T.nptr := mkleaf(num, num.val)*

(a) Construct the parse tree and syntax tree for the expression **((a)+(b))**.

Parse tree:



Syntax tree:



(b) Say whether each attribute of a non-terminal is *inherited* or *synthesized* and why.

All the attributes are synthesized, as they are propagated up the tree.