CMSC 430
Spring '09 Midterm 1

You have until 4:45pm to complete the midterm.

Feel free to ask questions on the midterm.

In grammars, capital letters represent nonterminals, lower case letters represent terminals.

One sentence answers are sufficient for the "essay" questions.

---

1. (4 points) Compiler front ends.

   (a) What is the primary interaction between the scanner and parser?

   Scanner converts input program into a stream of tokens for the parser, which converts it into the intermediate representation.

   (b) What do scanner generators (e.g., JLex) and parser generators (e.g., CUP) take as input? What do they output?

   They take regular expressions for tokens, and context-free grammars & syntax-directed actions as input, respectively. They output code implementing scanners and parsers.

---

2. (4 points) Regular expressions and context-free grammars.

   Consider the language of all strings composed of the letters `a` and `b` that end in `a` (e.g., `baba` is in the language, `aab` and ϵ are not).

   (a) Write a regular expression for the language.

   One solution: $(a \mid b)^*a$

   (b) Write a context-free grammar for the language.

   One solution: $S \rightarrow Ea$

   $E \rightarrow \epsilon \mid aE \mid bE$

---

3. (8 points) Parsing.

   (a) What is a left-most derivation? Name one useful property of left-most derivations for a compiler.

   A derivation where the left-most non-terminal is expanded at each step. Unique for non-ambiguous grammars.

   (b) Name one advantage and disadvantage for bottom-up parsing (compared to top-down parsing).

   Bottom-up parsing is more powerful (accepts more grammars), allows left-recursion, and can handle some ambiguous grammars by specifying precedence and associativity. It is less intuitive and harder to add clear error messages.

   (c) Give an example grammar which can be processed by a LL(2) parser, but not by an LL(1) parser.

   $S \rightarrow aa \mid ab.$

   (d) Give an example grammar which can be processed by a LR(0) parser, but not by an LL(k) parser.

   $S \rightarrow Sa \mid b.$

---

4. (14 points) Top-down parsers.

   Consider the following grammar:

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

   (a) Calculate FIRST and FOLLOW for A, B:

Nonterminals	FIRST	FOLLOW
A	{ a , c, b }	{ b, $ }
B	{ c, ϵ }	{ b }

   (b) Construct a LL(1) parser for this grammar.

	a	b	c	$
A	A $\rightarrow$ a	A $\rightarrow$ Bb	A $\rightarrow$ Bb	
B		B $\rightarrow \epsilon$	B $\rightarrow$ cA	

   (c) Show the steps taken by your LL(1) parser to parse the input "cbb".

   ```
   Stack          Input

   $ A          c b b $
   $ bB         c b b $
   $ bAc        c b b $
   $ bA           b b $
   $ bbB          b b $
   $ bb           b b $
   $ b              b $
   $                  $
   ```

5. (4 points) Shift-reduce parsing.

Consider the following ACTION/GOTO tables:

State	Action a	Action b	Action $	Goto A	Goto B
0	shift 1	shift 4		6	4
1	reduce A → a	reduce B → aB	reduce A → a	0	5
2	reduce B → Ab	shift 5	accept	1	5
3	reduce A → b	shift 1	reduce A → ϵ	3	5
4	shift 1	reduce A → b	accept	5	3
5	reduce B → A	shift 3	reduce B → bA	3	2
6	reduce A → a	shift 2		4	1

Show the contents of the stack and input buffer for the shift-reduce parse of "b a", assuming State 0 is the start state:

Stack	Input	Action
$ 0	b a $	action[0,b] → shift 4
$ 0 b 4	a $	action[4,a] → shift 1
$ 0 b 4 a 1	$	action[1,$] → reduce A → a
$ 0 b 4 A	$	goto[4,A] = 5
$ 0 b 4 A 5	$	action [5,$] = reduce B → bA
$ 0 B	$	goto[0,B] = 4
$ 0 B 4	$	action [4,$] = accept

6. (16 points) LR(1) parse table construction.

Consider the following (already augmented) grammar:

P1	S	→	E
P2	E	→	a E
P3	E	→	a

(a) Derive its canonical sets of LR(1) items

(b) Given the following sets of LR(1) states, build its LR(1) parse table:

State	ACTION d	ACTION e	ACTION $	GOTO A	GOTO B
0				2	1
1	shift 3				
2		shift 4			
3			reduce S→bBd		
4			reduce S→bAe		

7. (4 points) LR(1) conflicts.

Consider the following possible sets of LR(1) items in the states of a shift/reduce parser:

State 0:
[A → a • , b]
[B → a • b, b]
[C → b a •, a]

State 1:
[A → b • , a]
[B → b • b , b]
[C → a b • , a]

(a) For the items in State 0, list any conflicts that exist and describe for what lookaheads they occur.

A shift/reduce conflict occurs for lookahead 'b' because the parser can either shift on the production for B or reduce on the productions for A.

(b) For the items in State 1, list any conflicts that exist and describe for what lookaheads they occur.

A reduce/reduce conflict occurs for lookahead 'a' because the parser can either perform a reduction to either A or C.

8. (6 points) LALR parsers

(a) Give an example of two LR(1) states that would be merged in a LALR(1) parser.

[A → • a, b] and [A → • a, a]

(b) Name one advantage of LALR(1) parsers over SLR(1) parsers.

LALR(1) parsers are more powerful, and thus can handle more grammars and SLR(1) parsers.

(c) Name one advantage and disadvantage of LALR(1) parsers compared to LR(1) parsers.

LALR(1) parsers are smaller, but may have additional reduce/reduce conflicts.

---

9. (8 pts) Attribute grammars

Consider the following example grammar. It performs actions to compute the type of a variable as either INT or CHAR, using a global variable curType.

DECL	→	TYPE ID	{ ID.type = curType; }
TYPE	→	INT	{ curType = INT; }
	|	CHAR	{ curType = CHAR; }

(a) Why is the example grammar not an attribute grammar?

The actions utilize curType, which is not an attribute or constant.

(b) Change the actions in the grammar so that it is an attribute grammar.

DECL	→	TYPE ID	{ ID.type = TYPE.type; }
TYPE	→	INT	{ TYPE.type = INT; }
	|	CHAR	{ TYPE.type = CHAR; }

(c) Would it be easy or hard to have written the type checker in your programming project as an attribute grammar? Why?

Hard, since it will be difficult to collect and maintain information for the type of each symbol.

---

10. (4 pts) Type expressions

Given the following C declarations:

```
int * bar(int zod[2]);
```

(a) Write the type expression for "bar"

array(int, 0..1) → pointer(int)

(b) Why do we use type expressions instead of a list of variable types permitted by the language/compiler?

Because the programmer can define arbitrary new types (e.g., by declaring new structures).

---

11. (12 pts) Syntax-directed translation and type checking

Consider the following grammar productions for evaluating positive constant expressions. Assume you have an attribute E.zero that is set to TRUE if the expression is zero, and FALSE if it is non-zero (or undefined). Also assume that the terminal CONST has an attribute *val* representing its value.

E	→	CONST	{ CONST.val; E.zero = ?? }
	|	E_1 + E_2	{ E.zero = ?? }
	|	E_1 * E_2	{ E.zero = ?? }
	|	E_1 / E_2	{ E.zero = ?? }
	|	(E_1)	{ E.zero = ?? }

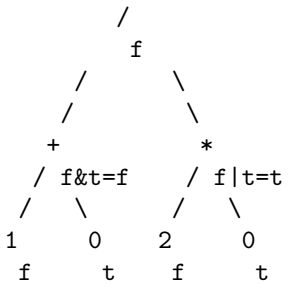
(a) Add rules to the attribute grammar to calculate E.zero for each grammar production.

E	→	CONST	{ E.zero = (CONST.val == 0); }
	|	E_1 + E_2	{ E.zero = E_1.zero && E_2.zero }
	|	E_1 * E_2	{ E.zero = E_1.zero || E_2.zero }
	|	E_1 / E_2	{ E.zero = E_1.zero }
	|	(E_1)	{ E.zero = E_1.zero }

(b) Using the zero attribute, add a compile-time message warning of divide by zero errors, using the function parser.msg("...").

Add the following to the E → E_1 / E_2 production: if (E_2 == 0) parser.msg("Division by zero"); }

(c) Provide the parse tree for the expression
(1 + 0) / (2 * 0)
and show the calculation for E.zero for each node.



---

12. (6 pts) Symbol tables

Consider the following program in a lexically-scoped language such as C.

```
int y,z
int foo( ) { int y; ...                                  }
int bar( ) { int z; { int x;  x = y; // HERE } }
```

(a) Why are nested symbol tables frequently used to represent nested scopes?

To ensure symbol lookup finds the closest lexically-nested variable declaration.

(b) Construct the logical state of nested symbol tables when the compiler reaches the point marked HERE in the example.