

## CMSC 430 Practice Midterm

In grammars, capital letters represent nonterminals,  
lower case letters represent terminals.  
One sentence answers are sufficient for the "essay" questions.

1. (6 points) Compiler front end.
    - (a) What is the primary function of the scanner and what computational mechanism is used to accomplish it?
    - (b) What is the primary function of the parser and what computational mechanism is used to accomplish it?
    - (c) How do you decide what should be handled by the scanner versus the parser? (Hint: think of the complexity of languages)
- 
2. (16 points) Scanner construction.
    - (a) Construct a regular expression for recognizing all non-empty strings composed of the letters a and b that do not end in b.
    - (b) Convert the regular expression to an NFA using Thompson's construction algorithm.
    - (c) Convert the NFA to a DFA (show the sets of NFA states for each DFA state).
    - (d) Minimize the DFA (show the sets of DFA states for each minimized DFA state).

3. (10 points) Consider the following grammar:

$$E \rightarrow E + E \mid a$$

- (a) When is a grammar ambiguous?
- (b) Show that the grammar is ambiguous for the string  $a + a + a$
- (c) What happens if you write a recursive-descent parser for this grammar?
- (d) Fix the grammar to avoid this problem.

4. (10 points) Consider the following grammar:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a \mid \epsilon \\ B &\rightarrow b \mid \epsilon \end{aligned}$$

- (a) Calculate FIRST for S, A, B:
- (b) Construct its recursive-descent parser (with lookahead), given the following functions:

```
tok; // current token

match(x) { // matches token
  if (tok != x) // if wrong token
    error(); // exit with error
  tok = getToken(); // get new token
}

parser() {
  tok = getToken(); // initialize
  S( ); // start symbol
  match("$"); // match EOF
}
```

5. (6 points) Consider the following ACTION/GOTO tables:

State	Action		Goto	
	a	\$	A	B
0	shift 1	reduce $B \rightarrow \epsilon$	2	3
1	shift 3	reduce $A \rightarrow a$	0	
2		accept	3	0
3	shift 1	accept		1

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

6. (16 points) Consider the following augmented grammar:

$$\begin{array}{l|l} P1 & S \rightarrow E \\ P2 & E \rightarrow E + E \\ P3 & \quad \quad \quad a \end{array}$$

- (a) Derive the canonical sets of LR(1) items
- (b) Build the LR(1) parse table
- (c) For each shift/reduce or reduce/reduce conflict you find (if any), describe what would happen for the different ways to resolve the conflict(s).

---

7. (10 points) Consider the following sets of LR(1) items in the states of a LR(1) parser:

State 0:	State 2:
[ A → • a , b ]	[ A → • a , c ]
[ A → a • , c ]	[ A → a • , b ]
[ B → a • , b ]	[ B → a • , a ]

State 1:	State 3:
[ A → • a , a ]	[ A → • a , b ]
[ A → • a , b ]	[ B → • a , b ]
[ B → a • , b ]	

- Find all conflicts, listing state, pair of LR(1) items, and lookahead(s) causing conflict.
- List states that would be merged in a LALR(1) parser.
- List additional conflicts in the LALR(1) parser, if any.
- What are the advantages of LALR over LR parsers?

---

8. (6 points) Syntax-directed translation.

- What are the main differences between attribute grammars and ad-hoc, syntax-directed translation?
- Name one advantage and disadvantage of each.

---

9. (8 points) Type checking.

- Consider the following C code.

```
int foo(char *x, int bar[4]);
```

What is the type expression for `foo`?

- Consider the following grammar productions. Assume you have an attribute `const.odd` which is set to either true or false. How can we then extend the type checker to determine whether an expression is odd (or even)?

$$\begin{array}{l} E \rightarrow \text{const} \quad \{ E.\text{odd} = ?? \} \\ \quad | \quad E_1 + E_2 \quad \{ E.\text{odd} = ?? \} \end{array}$$

Give the actions for both productions.

---

10. (12 points) Symbol table, run-time environments.

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

```
int x,y;
int f( int p) {
    int y,z;
    {
        int i,j;
        /* Point A - innermost scope */
    }
    /* Point B - function scope */
}
/* Point C - global scope */
```

- How can the compiler organize symbol table information at compile time to handle nested scoping? Draw the logical state of the symbol table(s).
- What symbols are visible at points A and B?
- Name some information stored for each procedure at run time.
- How does the compiler allocate storage for `x` at run time?
- How does the compiler allocate storage for `y` at run time?
- How does the compiler allocate storage for `p` at run time?