CMSC 430
Practice Midterm 1

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?

   Break the input stream up into tokens using deterministic finite automata (DFA).

   (b) What is the primary function of the parser and what computational mechanism is used to accomplish it?

   Provide syntactic structure to token stream using push-down automata (PDA = DFA + stack).

   (c) How do you decide what should be handled by the scanner versus the parser? (Hint: think of the complexity of languages)

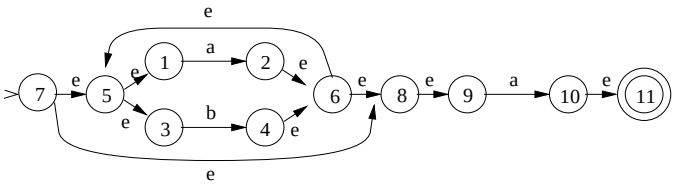
   Scanner handles simple constructs that can be easily described by regular expressions, parser handles more complicated syntax described by context-free grammars.

2. (20 points) Scanner construction.

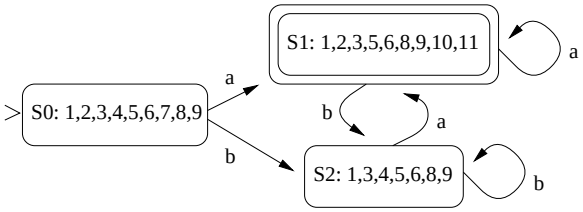
   (a) Construct a regular expression for recognizing all non-empty strings composed of the letters `a` and `b` that do not end in `b`.

   $(a \mid b)^* a$

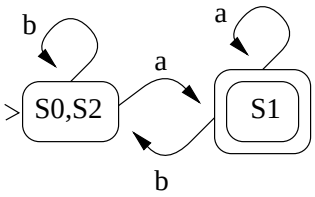
   (b) Convert the regular expression to an NFA using the construction algorithm given in class.



   (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. (16 points) Consider the following grammar:

$$\mathrm{E} \quad \rightarrow \quad \mathrm{E} + \mathrm{E} \mid \mathrm{a}$$

   (a) When is a grammar ambiguous?

   When there are multiple left-most (or right-most) derivations of the same string.

   (b) Show that the grammar is ambiguous for the string a + a + a

(leftmost derivation 1)

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow a + E \\ &\Rightarrow a + E + E \\ &\Rightarrow a + a + E \\ &\Rightarrow a + a + a \end{aligned}$$

(rightmost derivation 1)

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow E + a \\ &\Rightarrow E + E + a \\ &\Rightarrow E + a + a \\ &\Rightarrow a + a + a \end{aligned}$$

(leftmost derivation 2)

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow E + E + E \\ &\Rightarrow a + E + E \\ &\Rightarrow a + a + E \\ &\Rightarrow a + a + a \end{aligned}$$

(rightmost derivation 2)

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow E + E + E \\ &\Rightarrow E + E + a \\ &\Rightarrow E + a + a \\ &\Rightarrow a + a + a \end{aligned}$$

   (c) What happens if you write a recursive-descent parser for this grammar?

   May infinite loop trying to match E.

   (d) Fix the grammar to avoid this problem.

$E \rightarrow a\ E'$

$E' \rightarrow +\ E\ E' \mid \epsilon$

4. (16 points) Consider the following grammar:

$$\begin{array}{lcl} S & \rightarrow & ABd \\ A & \rightarrow & aA \mid \epsilon \\ B & \rightarrow & b \mid cA \end{array}$$

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

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

(b) Construct its recursive-descent parser (with lookahead), given the following functions:

```
lookahead;              // current token

match(t) {              // matches token
  if (lookahead == t)  // found match
    lookahead = next_token(); // get next
  else error( );        // else error
}

parser() {
  lookahead = next_token(); // init
  S( );                 // start symbol
  match("$");           // match EOF
}
```

Answer

```
S() {
      A(); B(); match("d");
    }

A() {
      if (lookahead == "a")
         { match("a"); A(); }
      else if (lookahead == "b") ||
              (lookahead == "c") ||
              (lookahead == "d"))
         { ; }
      else error();
    }

B() {
      if (lookahead == "b")
        { match("b"); }
      else if (lookahead == "c")
        { match("c"); A(); }
      else error();
    }
```

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

State	Action a	Action $	Goto A	Goto 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:

Stack	Input	Action
$ 0	a $	shift 1
$ 0 a 1	$	reduce A $\rightarrow$ a
$ 0 A	$	goto[0,A] = 2
$ 0 A 2	$	accept

6. (20 points) Consider the following augmented grammar:

$$\begin{array}{lcll} P1 & S & \rightarrow & E \\ P2 & E & \rightarrow & E + E \\ P3 & & \mid & a \end{array}$$

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

State 0:
[S $\rightarrow \bullet$ E , {$ }]
[E $\rightarrow \bullet$ E + E , {$,+}]
[E $\rightarrow \bullet$ a , {$,+}]

State 1:
[S $\rightarrow$ E $\bullet$, {$ }]
[E $\rightarrow$ E $\bullet$ + E , {$,+}]

State 2:
[E $\rightarrow$ a $\bullet$, {$,+}]

State 3:
[E $\rightarrow$ E + $\bullet$ E , {$,+}]
[E $\rightarrow \bullet$ E + E , {$,+}]
[E $\rightarrow \bullet$ a , {$,+}]

State 4:
[E $\rightarrow$ E + E $\bullet$, {$,+}]
[E $\rightarrow$ E $\bullet$ + E , {$,+}]

GOTO(0, E) = 1
GOTO(0, a) = 2
GOTO(1, +) = 3
GOTO(3, a) = 2
GOTO(3, E) = 4
GOTO(4, +) = 3

(b) Build the LR(1) parse table

State	Action a	Action +	Action $	Goto E
0	shift 2			1
1		shift 3	accept	
2		reduce P3 (E $\rightarrow$ a)	reduce P3 (E $\rightarrow$ a)	
3	shift 2			4
4		shift 3 reduce P2 (E $\rightarrow$ E + E)	reduce P2 (E $\rightarrow$ E + E)	

(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).

For the shift/reduce conflict in state 4 with lookahead +, resolving the conflict in favor of reduce would make + left-associative; resolving it in favor of shift would make + right-associative.

7. (16 points) Consider the following sets of LR(1) items in the states of a LR(1) parser:

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

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

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

State 3:
[A → • a , b]
[B → • a , b]

(a) Find all conflicts, listing state, pair of LR(1) items, and lookahead(s) causing conflict.

State 2, shift/reduce conflict on lookahead=a between [A → • a , c] and [B → a • , a]

(b) List states that would be merged in a LALR(1) parser.

State 0 and 2

(c) List additional conflicts in the LALR(1) parser, if any.

State 0&2, reduce/reduce conflict on lookahead=b between [A → a • , b] and [B → a • , b]

(d) What are the advantages of LALR over LR parsers?

Smaller number of DFA states.

---

8. Syntax-directed translation.

(a) What are the main differences between attribute grammars and ad-hoc, syntax-directed translation?

Attribute grammars only allow attributes to be calculated from the attributes of parents, siblings, or children. Ad-hoc syntax-directed translation allows arbitrary actions.

(b) Name one advantage and disadvantage of each.

Attribute grammars are easier to analyze and prove properties. Ad-hoc methods are more flexible.

---

9. Type checking.

(a) Consider the following C code.

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

What is the type expression for `foo`?

pointer (char) × array (4, int) → int

(b) 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)?

E	→	const	{ E.odd = ?? }
	|	$E_1 + E_2$	{ E.odd = ?? }

Give the actions for both productions.

Answer

E	→	const	{ E.odd = const.odd }
	|	$E_1 + E_2$	{ E.odd = E_1.odd **xor** E_2.odd }

---

10. 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 */
   }
      /* Point B */
}
      /* Point C */
```

(a) How can the compiler organize symbol table information at compile time to handle nested scoping? Draw the logical state of the symbol table(s).

The compiler can used nested symbol tables, one table per scope. Symbol tables contain a pointer to the table of their enclosing scope.

(b) What symbols are visible at points A and B?

The following symbols are visible (from outer to inner):

A	*x, f, p, y, z, i, j*
B	*x, f, p, y, z*
C	*x, y, f*