Concepts of Programming Languages, CSCI 305, Fall 2020
Exam 3, Oct. 30

Name _____

This is a closed book exam. You may not use notes, the text book, Internet, etc.

1. Advantages of references over pointers are that (4 pts.)
    a. Dereferencing is done automatically
    b. Memory can be reclaimed via garbage collection
    c. The compiler can do more optimization
    d. Safety
    e. All of the above

2. The terms static/dynamic are well defined. (true or false) (4 pts.)
    a. True
    b. False

3. Java is (4 pts.)
    a. Strongly and statically typed
    b. Weakly and statically typed
    c. Strongly and dynamically typed
    d. Weakly and dynamically typed
    e. None of the above

4. Which of the following is NOT a task of lexical analyzer? (4 pts.)
    a. Build a parse tree
    b. Remove comments
    c. Remove whitespace
    d. Collect characters into logical groupings
    e. These are all tasks of the lexical analyzer.

5. A Horn clause typically consists of a head and a body. What is a Horn clause with no body? (4 pts.)
    a. Horn clauses are not allowed to not have a body.
    b. This is a fact
    c. This is a query
    d. None of the above

6. Define a context free grammar for the language on {a,b} which has more a's than b's. This can be written more formally as:

$L=\{w \mid w \in \{a, b\}^* \wedge (n_a(w) > n_b(w)\}$ where

$n_a(w)$ is the number of a's in the string w

$n_b(w)$ is the number of b's in the string w

(10 pts.)

Strings in L	Strings not in L
a	ε
aba	b
bbbaaaa	ab
ababababa	abab
aaaaabababa	bbbaa

S → ME | EM | SS
M → aMb | bMa | ε
E → Aa | a

M is for Match
E is for Extra as.

7. Consider the grammar with the variables {bexp, bterm, bfactor} and terminals {true, false, and, or, not, (,)}.

    1. bexp → bexp or bterm
    2. bexp → bterm
    3. bterm → bterm and bfactor
    4. bterm → bfactor
    5. bfactor → not bfactor
    6. bfactor → (bexp)
    7. bfactor → true
    8. bfactor → false

    Tell the FIRST and FOLLOW predict sets.

	EPS	FIRST	FOLLLOW
bexp	false	{not, (, true, false}	{ or ,) }
bterm	false	{not, (, true, false}	{and, or,) }
bfactor	false	{not, (, true, false}	{and, or,) }
true	false	{true}	{and, or,) }
false	false	{false}	{and, or,) }
and	false	{and }	{not , (, true, false }
or	false	{ or }	{not , (, true, false }
not	false	{not}	{not , (, true, false }
(	false	{ (}	{not , (, true, false }
)	false	{) }	{or, and) }

(10 pts.)

8. Consider the grammar below, with the EPS, FIRST and FOLLOW sets specified.

   a. Augment the grammar with the sets which can be used to create the parsing table. That is, follow each production with the set which will be needed to create the parsing table. (5 pts.)

1. E → TE' { (, id }
2. E'→ +TE' { + }
3. E'→ ε { (, $ }
4. T→ FT' { (, id }
5. T'→ *FT' { * }
6. T'→ ε { +,), $ }
7. F→ (E) { (}
8. F→ id { id }

Note that $ is returned by the lexical analyzer when the end of the source code is reached.

	EPS	FIRST	FOLLOW
E	false	{(, id}	{)}
T	false	{(, id}	{+, $}
E'	true	{+}	{), $}
F	false	{(, id}	{*, +,), $}
T'	true	{*}	{+,), $}
+	false	{+}	{(, id}
*	false	{*}	{(, id}
(	false	{(}	{(, id}
)	false	{)}	{*, +,), $}
id	false	{id}	Φ
$	false	{$}	Φ

   b. Give the parsing table (5 pts.)

	+	*	(	)	id	$
E	-	-	1	-	1	-
T	-	-	4	-	4	-
E'	2	-	-	3	-	3
F	-	-	7	-	8	-
T'	6	5	-	6	-	6

9. A classic example of an ambiguous grammar are the grammar productions for if-then-else in C, C++, and Pascal. Here is an ambiguous grammar for if-then-else construct where the variables are: STMT and EXPR.

   STMT -> if EXPR then STMT | if EXPR then STMT else STMT

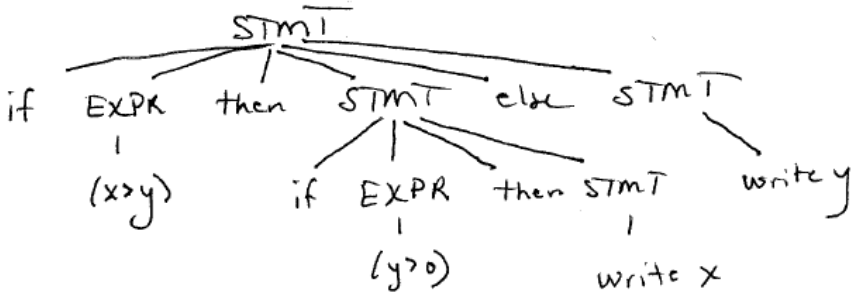
a. Define what it means for a grammar to be ambiguous. (5 pts.)

   At least one statement in the language has more than one parse tree.

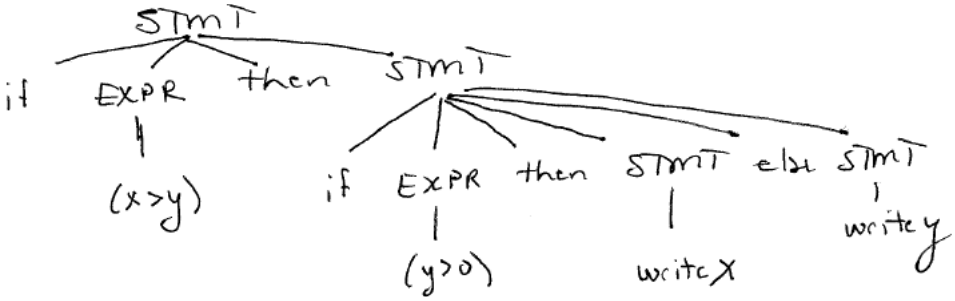
b. Show that this grammar is ambiguous. (5 pts.)

   The following string has two parse trees:
   if (x>y) then if (y>0) then write x else write y

   Possible tree:



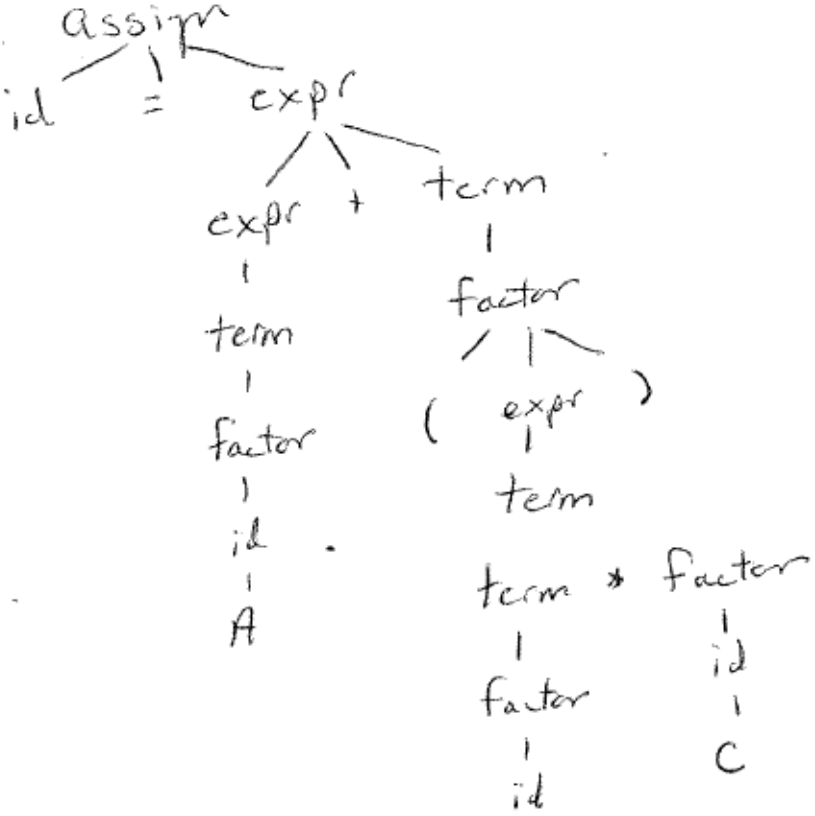
Alternative tree:



10. Consider the following grammar:

assign → id = expr
id → A | B | C | D
expr → expr + term | term
term → term * factor | factor
factor → (expr) | id

a. Does this grammar accept the string D = A + (B * C)? If so, give a parse tree for the string. If not, explain why not. (5 pts)



b. If * has a higher precedence than + in the above grammar, modify the grammar so that + has a higher precedence than *. Otherwise, explain why * does not have a higher precedence than +. (5 pts)

assign → id = expr
id → A | B | C | D
expr → expr * term | term
term → term + factor | factor
factor → (expr) | id

11.

The driver for a table-driven SLR(1) parser, along with its parsing and production tables are given.

```
state = 1 . . number_of_states
symbol = 1 . . number_of_symbols
production = 1 . . number_of_productions
action_rec = record
    action : (shift, reduce, shift_reduce, error)
    new_state : state
    prod : production

parse_tab : array [symbol, state] of action_rec
prod_tab : array [production] of record
    lhs : symbol
    rhs_len : integer
-- these two tables are created by a parser generator tool

parse_stack : stack of record
    sym : symbol
    st : state
parse_stack.push(⟨null, start_state⟩)
cur_sym : symbol := scan                    -- get new token from scanner
loop
    cur_state : state := parse_stack.top.st -- peek at state at top of stack
    if cur_state = start_state and cur_sym = start_symbol
        return                              -- success!
    ar : action_rec := parse_tab[cur_state, cur_sym]
    case ar.action
        shift:
            parse_stack.push(⟨cur_sym, ar.new_state⟩)
            cur_sym := scan                 -- get new token from scanner
        reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len)
        shift_reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len-1)
        error:
            parse_error
```

Figure 2.28 **Driver for a table-driven SLR(1) parser.** We call the scanner directly, rather than using the global input_token of Figures 2.16 and 2.18, so that we can set cur_sym to be an arbitrary symbol.

Grammar:

1. program → stmt_list $$
2. stmt_list → stmt_list stmt
3. stmt_list → stmt
4. stmt → id := expr
5. stmt → read id
6. stmt → write expr
7. expr → term
8. expr → expr add_op term
9. term → factor
10. term → term mult_op factor
11. factor → (expr)
12. factor → id
13. factor → number
14. add_op → +
15. add_op → -
16. mult_op → *
17. mult_op → /

Top-of-stack state	Current input symbol																		
	sl	*s*	*e*	*t*	*f*	*ao*	*mo*	id	lit	r	w	:=	(	)	+	-	*	/	$$
0	s2	b3	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	b5	–	–	–	–	–	–	–	–	–	–	–
2	–	b2	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	b1
3	–	–	–	–	–	–	–	–	–	–	–	s5	–	–	–	–	–	–	–
4	–	–	s6	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
5	–	–	s9	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
6	–	–	–	–	–	s10	–	r6	–	r6	r6	–	–	–	b14	b15	–	–	r6
7	–	–	–	–	–	–	s11	r7	–	r7	r7	–	–	r7	r7	r7	b16	b17	r7
8	–	–	s12	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
9	–	–	–	–	–	s10	–	r4	–	r4	r4	–	–	–	b14	b15	–	–	r4
10	–	–	–	s13	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
11	–	–	–	–	b10	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
12	–	–	–	–	–	s10	–	–	–	–	–	–	–	b11	b14	b15	–	–	–
13	–	–	–	–	–	–	s11	r8	–	r8	r8	–	–	r8	r8	r8	b16	b17	r8

Figure 2.27 SLR(1) parse table for the calculator language. Table entries indicate whether to shift (s), reduce (r), or shift and then reduce (b). The accompanying number is the new state when shifting, or the production that has been recognized when (shifting and) reducing. Production numbers are given in Figure 2.24. Symbol names have been abbreviated for the sake of formatting. A dash indicates an error. An auxiliary table, not shown here, gives the left-hand-side symbol and right-hand-side length for each production.

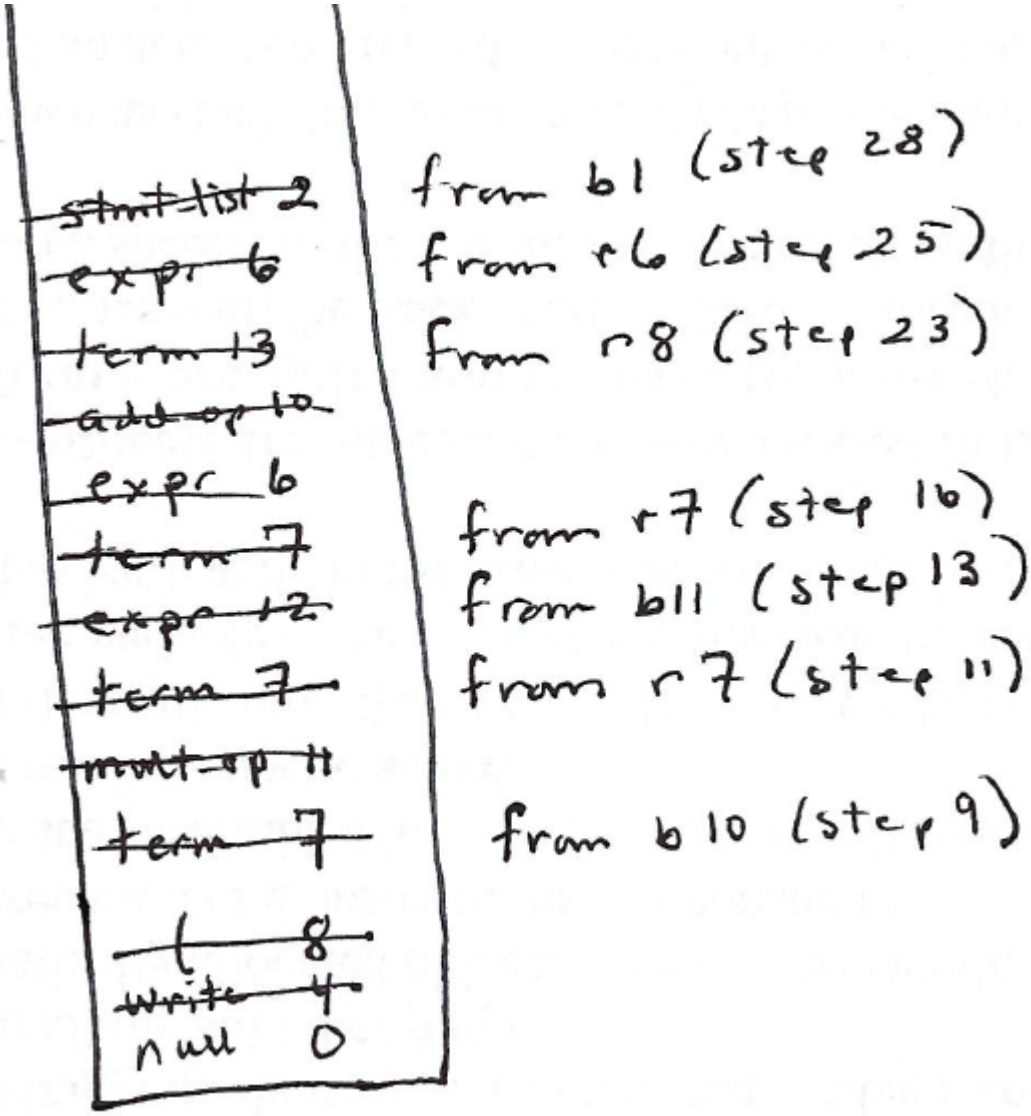
Show all pushes and pops on the stack when executing the program:

write (20/A)+B $$

(10 pts.)

	table lookup	**stack (sym, state) cur_state is the top of the stack**	**cur_sym**	**input stream**
				write (20/A)+B $$
		(null,0)	write	(20/A)+B $$
enter loop				
1	(0, write) → s4	(null,0)(write,4)	**(**	20/A)+B $$
2	(4, **(**) → s8	(null,0)(write,4)(**(**,8)	number(20)	/A)+B $$
3	(8,number) → b13	(null,0)(write,4)(**(**,8)	factor	/A)+B $$
4	(8,factor) → b9	(null,0)(write,4)(**(**,8)	term	/A)+B $$
5	(8,term) → s7	(null,0)(write,4)(**(**,8) (term,7)	/	A)+B $$
6	(7,/) → b17	(null,0)(write,4)(**(**,8) (term,7)	multi_op	A)+B $$
7	(7,mult-op) → s11	(null,0)(write,4)(**(**,8) (term,7)(mult_op,11)	id(A)	)+B $$
8	(11,id) → b12	(null,0)(write,4)(**(**,8) (term,7)(mult_op,11)	factor	)+B $$
9	(11,factor) → b10	(null,0)(write,4)(**(**,8)	term	)+B $$
10	(8,term) → s7	(null,0)(write,4)(**(**,8) (term,7)	**)**	+B $$
11	(7,**)**) → r7	(null,0)(write,4)(**(**,8)	expr	)+B $$ (reduce puts **)** back)
12	(8,expr) → s12	(null,0)(write,4)(**(**,8) (expr,12)	**)**	+B $$
13	(12,**)**) → b11	(null,0)(write,4)	factor	+B $$
14	(4,factor) → b9	(null,0)(write,4)	term	+B $$
15	(4,term) → s7	(null,0)(write,4)(term,7)	+	B $$
16	(7,+) → r7	(null,0)(write,4)	expr	+B $$ (reduce puts + back)
17	(4,expr) → s6	(null,0)(write,4)(expr,6)	+	B $$
18	(6,+) → b14	(null,0)(write,4)(expr,6)	add_op	B $$
19	(6,add_op) → s10	(null,0)(write,4)(expr,6) (add_op,10)	id(B)	$$
20	(10,id) → b12	(null,0)(write,4)(expr,6) (add_op,10)	factor	$$
21	(10,factor) → b9	(null,0)(write,4)(expr,6)	term	$$

		(add_op,10)		
22	(10,term) → s13	(null,0)(write,4)(expr,6)(add_op,10)(term,13)	$$	
23	(13,$$) → r8	(null,0)(write,4)	expr	$$ (reduce puts **$$** back)
24	(4,expr) → s6	(null,0)(write,4)(expr,6)	$$	
25	(6,$$) → r6	(null,0)	stmt	$$ (reduce puts **$$** back)
26	(0,stmt) → b3	(null,0)	stmt_list	$$
27	(0,stmt_list) → s2	(null,0)(stmt_list,2)	$$	
28	(2, $$) → b1	(null,0)	program	
	cur_state=0 & cur_sym=program so return☺			



Write programs in Prolog for the following. Do not use the computer to write these programs, simply write your program onto the test page.

12. Write Prolog code which takes a position and a list, and returns the element at the given position in the list. You can assume that the length of the list exceed the position given and that the position is at least 1.

Example:

?- element_at(3, [a, b, c, d, e], X).

X = c (5 pts.)

```
element_at(1,[H|_],H).
element_at(N,[_|T],X):-
        M is N-1, element_at(M,T,X).
```

13. Write Prolog code which take a list and a variable, and returns the smallest element of the list in the variable. You can assume that the list will contain at least one element. (5 pts.)

```
min_numlist([H|[]],H).
min_numlist([H|T], Min) :-
        min_numlist(T, SubMin),
        SubMin<H-> Min=SubMin; Min=H.
```