

Concepts of Programming Languages, CSCI 305, Fall 2021
Exam 2, Oct. 29

The exam will have two parts, a non-programming portion and a programming portion. You may not use notes, book, etc. for the non-programming portion. You can use the text, your notes and the web, but not other people, for the programming portion.

Multiple Choice

1. Which of the following is least likely to be considered a procedural language? (4 pts.)
 - a. Prolog
 - b. Python
 - c. JavaScript
 - d. C#
 - e. Pascal

2. Which of the following is NOT typically a static semantic error? (4 pts.)
 - a. Undeclared identifier
 - b. Subroutine call providing the wrong number of arguments
 - c. Subroutine call providing the wrong type of arguments
 - d. Non-void functions not explicitly returning a value
 - e. Out of bound array subscripts

3. Which of the following best describes what is meant by a “lexeme”. (4 pts.)
 - a. A syntactic category
 - b. A pattern of characters
 - c. A string of characters that is a lowest-level syntactic unit
 - d. Basic building block of a program
 - e. All of the above

4. What best describes why the following context-free grammar is ambiguous.

$\text{expr} \rightarrow \text{id}$ (4 pts.)
 $\text{expr} \rightarrow \text{number}$
 $\text{expr} \rightarrow - \text{expr}$
 $\text{expr} \rightarrow (\text{expr})$
 $\text{expr} \rightarrow \text{expr op expr}$
 $\text{op} \rightarrow + \mid - \mid * \mid /$

- a. More than one production is defined for the variable 'expr'
- b. An expression in the language has more than one parse tree
- c. An expression in the language has both a right-most and a left-most derivation
- d. The 'op' variable can be replaced by 4 different operators.
- e. This grammar is not ambiguous

5. Which is the least likely to be a task of a parser? (4 pts.)

1. Detect syntax errors
2. Collect characters into logical groupings
3. Create a parse tree
4. Produce diagnostic messages and recover
5. Build symbol table

New Material

6. Create a grammar for a language which would be able to generate the strings such as the following.

$$- a + -\sin(b + c) * (- (b - a))$$

Your language should allow bracketing (left and right parentheses), leading unary signs (+, -), the binary operations of addition, subtraction, division and multiplication (+, -, * /), and simple function calls which are passed a single argument.

The order of precedence for your language should be:

function calls (highest)
unary + and -
binary + and -
binary * and / (lowest)

Note that this is non-standard. The expression $a + b * c$ would be evaluated $(a+b) * c$, rather than the usual $a + (b * c)$.

Your grammar should use right associativity. The expression $a + b + c$ would be evaluated $a+(b+c)$, rather than the usual $(a+b)+c$. (10 pts.)

To verify your work, feel free to use your grammar to create a parse tree for the string:

$$- a + -\sin(b + c) * (- (b - a))$$

(This is optional and doing it won't add points to your overall score.)

7. Consider the following grammar. If this grammar is not in LL(1), fix the rules, so that it recognizes the same language, but an LL(1) parser can be used. (10 pts.)

$\text{expr} \rightarrow \text{id} := \text{expr}$
 $\text{expr} \rightarrow (\text{expr}) \text{ term_tail factor_tail}$
 $\text{expr} \rightarrow \text{id term_tail factor_tail}$
 $\text{term_tail} \rightarrow + \text{term term_tail}$
 $\text{term_tail} \rightarrow \epsilon$
 $\text{term} \rightarrow (\text{expr})$
 $\text{term} \rightarrow \text{id}$
 $\text{factor_tail} \rightarrow * \text{factor factor_tail}$
 $\text{factor_tail} \rightarrow \epsilon$
 $\text{factor} \rightarrow (\text{expr})$
 $\text{factor} \rightarrow \text{id}$

8. Consider the grammar defined by:

Variables - {rexp, rexp', rterm, rterm', rfactor, rfactor', rprimary}

Terminals - {+, •, *, a, b, \$}

Start symbol - rexp

Productions:

1. $\text{rexp} \rightarrow \text{rterm rexp}'$
2. $\text{rexp}' \rightarrow + \text{rterm rexp}'$
3. $\text{rexp}' \rightarrow \epsilon$
4. $\text{rterm} \rightarrow \text{rfactor rterm}'$
5. $\text{rterm}' \rightarrow \bullet \text{rfactor rterm}'$
6. $\text{rterm}' \rightarrow \epsilon$
7. $\text{rfactor} \rightarrow \text{rprimary rfactor}'$
8. $\text{rfactor}' \rightarrow * \text{rfactor}'$
9. $\text{rfactor}' \rightarrow \epsilon$
10. $\text{rprimary} \rightarrow a$
11. $\text{rprimary} \rightarrow b$

with the parsing table:

	a	b	+	•	*	\$
rexp	1	1				
rexp'			2			3
rterm	4	4				
rterm'			6	5		6
rfactor	7	7				
rfactor'			9	9	8	9
rprimary	10	11				

Hand-execute table-driven recursive descent parsing (pseudo code is given on the next page) on the following string, showing all pushes and pops to the stack.

(10 pts.)

$b* + a \$$

```

terminal = 1 .. number_of_terminals
non_terminal = number_of_terminals + 1 .. number_of_symbols
symbol = 1 .. number_of_symbols
production = 1 .. number_of Productions

parse_tab : array [non_terminal, terminal] of record
    action : (predict, error)
    prod : production
prod_tab : array [production] of list of symbol
-- these two tables are created by a parser generator tool

parse_stack : stack of symbol

parse_stack.push(start_symbol)
loop
    expected_sym : symbol := parse_stack.pop()
    if expected_sym ∈ terminal
        match(expected_sym)                -- as in Figure 2.17
        if expected_sym = $$ then return    -- success!
    else
        if parse_tab[expected_sym, input_token].action = error
            parse_error
        else
            prediction : production := parse_tab[expected_sym, input_token].prod
            foreach sym : symbol in reverse prod_tab[prediction]
                parse_stack.push(sym)

```

9. Consider the grammar defined by:
 Variables - {G, S, M, A, E, B}
 Terminals - {a, b, \$\$}
 Start symbol - G

Productions:

1. $G \rightarrow S \$$
2. $S \rightarrow AM$
3. $M \rightarrow S$
4. $M \rightarrow \epsilon$
5. $A \rightarrow aE$
6. $A \rightarrow bAA$
7. $E \rightarrow aB$
8. $E \rightarrow bA$
9. $E \rightarrow \epsilon$
10. $B \rightarrow bE$
11. $B \rightarrow aBB$

Give the EPS, FIRST and FOLLOW predict sets.

(15 pts)

	EPS	FIRST	FOLLOW
G			
S			
A			
M			
E			
B			
\$			Leave blank
a			Leave blank
b			Leave blank

10. Consider the following Prolog program that takes a list and puts the reverse of the list in the second variable. For example,

```
?-reverse([a, b, c, d],X).
```

```
X = [d, c, b, a].
```

Prolog program:

```
reverse([],[]).
```

```
reverse([H|T],Ans):- reverse(T,SubAns), append(SubAns,[H], Ans).
```

(Note that append is a set of tuples of the form (X,Y, Z), where Z consists of the elements of X followed by the elements of Y.

Example: ([a],[b],[a, b]) is in the relation append.)

Trace how this program would execute, given the query

```
?- my_reverse([a, b, c, d],X).
```

By trace, give the “calls”, “exits”, “redo”, “fail”, along with the levels and goals which would occur in resolving this query. Assume that the first call is at level 10.

(10 pts.)

Old Material

11. Consider three different systems, or three components of the same system:

- Code that needs to be very robust because system errors could cause loss of life or severe financial loss
- Real-time code that needs to be very fast, where delayed execution could cause failure, and
- Code that needs to be easily modified as users with different needs are constantly being added.

Discuss the pros and cons of interpreted, compiled, and some combination of interpretation/compilation. Tell what type of system you would recommend for the 3 systems above and why. (15 pts.)

Extra Credit

The programming language FORTRAN was created around

- a. 1957
- b. 1977
- c. 1987
- d. 1997
- e. 2007

(1 pt.)

The programming language MATLAB first appeared around

- a. 1960
- b. 1970
- c. 1980
- d. 1990
- e. 2000

(1 pt.)

The programming language Lua first appeared around

- a. 1963
- b. 1973
- c. 1983
- d. 1993
- e. 2003

(1 pt.)

The programming language F# first appeared around

- a. 1975
- b. 1985
- c. 1995
- d. 2005
- e. 2015

(1 pt.)

The programming language Go first appeared around

- a. 1979
- b. 1989
- c. 1999
- d. 2009
- e. 2019

(1 pt.)

**Concepts of Programming Languages, CSCI 305, Fall 2021
Exam 2, Computer Portion, Oct. 29**

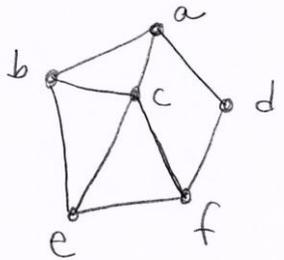
Programming Portion

Turn in the first portion of the exam before beginning this portion. To complete this portion of the exam you may use your notes, any previous assignments, the text and/or the Internet. You may not communicate with anyone other than me during this exam.

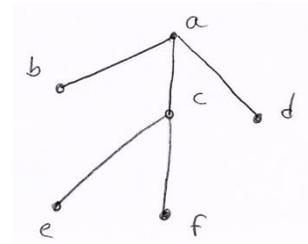
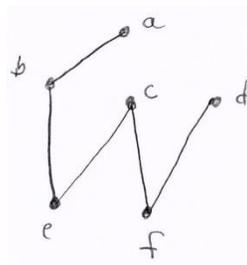
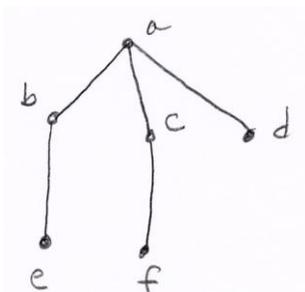
When you are done, email your answers to me:

12. Jake plans to write a spanning tree function in Prolog and has decided that he needs some helper functions.

Recall that a spanning tree of a graph, is a graph which contains all of the vertices in the original graph, however, it is also a tree. For example, the following graph



has several spanning trees.



Jake wants the following functions.

- a. The helper function ‘subsetOf’ takes two lists and returns true if all of the elements in the first list are contained in the second list.

For example:

?- subsetOf([1,2,3],[1,2,3,4]).

true.

?- subsetOf([1,2,3],[5,6,1,2]).
false.

?- subsetOf([], [a,b,c]).
true.

?- subsetOf([a,b,c], []).
false.

Define subsetOf in prolog using Horn clauses.

The above examples return true or false directly. Your program may return values (true/false or 1/0) in a variable if you prefer.

Feel free to use the built-in prolog function ‘member’, which takes an element and a list, and returns true if the element is in the list. Do not use any other built-in functions.

For example,
?- member(3, [a, b, c, 1, 2, 3, 4]).
true.

Hint: your prolog “programs” are likely to be very short.

(5 pts.)

- b. The helper function 'sameElements' takes two lists and returns true if the lists contain the same elements, ignoring repetition.

For example:

?- sameElements([a, b, c, a, a, a],[c, b, b, b, a]).
true .

?- sameElements([],[]).
true .

?- sameElements([a,b,c], [b, a, a, a]).
false.

Define sameElements in prolog using Horn clauses. (5 pts.)