CS-3323 Principles of Programming Languages Midterm Exam

March 4, 2020

Name: ___________________ **ID#** ___________________

Instructions: This exam is worth a total of 20pts. The maximum time to complete it is 75 minutes. You can use books, slides, handouts, and any sort of notes. However, no digital device can be used (e.g. no laptops, phones or calculators of any kind).

Problem 1 (5pts)

(1pt) Assuming right associativity for the MINUS binary operator (-), left-associativity for exponentiation (**), and ** having a higher precedence than -, show the evaluation steps and the final result of evaluating the expression: 9 - 1 - 2 ** 6 ** 0.5

(1pt) Consider the regular expression: $(a|b) * c + (bc|dc)*$

Write two strings that would be rejected and two string that would be accepted by the above regular expression. All strings must be at least 4 symbols long.

(0.5pt) What's the total number of tokens that a C scanner would produce for the following code snippet ?

```
while ( a + b < 10 ) b = b + 2 ;
```

(1pt) For the below statements, mark either True or False:

Statement	**True/False**
Left/right operator associativity is necessary for the + operator	
Operator associativity deals with the order of evaluation of a sequence of operations of the exact same type	
LL parsing works in a bottom-up fashion	
LL parsers are perfectly suitable to handle left-recursion	

(0.5pt) Write a regular expression that would accept the following 4 strings:

- a b
- a a b b c c
- a b b b c c c c
- b b b b b b c c c c c

(1.0pt) Write a small grammar that permits to produce lists of function arguments for a C-like language. The tokens to use in this grammar are: ID, INT, FLOAT, LEFTPAR '(', RIGHTPAR ')' and COMMA ' , '. ϵ is the empty string.

You can use the pipe character '|' to separate multiple right-hand-sides for the same non-terminal. The grammar can have either left-recursion or right-recursion, but not both types.

Below are 3 examples, one per line, of the strings that could be accepted:

- (int a, int b)
- (int a, float b, int c)
- (int a)
- ( )

Problem 2 (10pts)

Consider the following grammar, which allows to recognize array references:

1. varref $\rightarrow$ T_ID dimlist
2. dimlist $\rightarrow$ dimlist '[' fact ']'
3. dimlist $\rightarrow$ ϵ
4. fact $\rightarrow$ varref
5. fact $\rightarrow$ T_NUM
6. fact $\rightarrow$ '-' fact

(2pts) Assuming a right-most derivation, write all the sentential forms for the string: T_ID [T_NUM] [- T_ID]

(3pts) For the same string, and assuming again a right-most derivation, draw the corresponding parse tree:

(5pts) For the same string and assumptions, use the below LR parsing table to accept or reject it.

State 0
0 $accept: . varref $end

T_ID shift, and go to state 1

varref go to state 2

State 1
1 varref: T_ID . dimlist

$default reduce using rule 3 (dimlist)

dimlist go to state 3

State 2
0 $accept: varref . $end

$end shift, and go to state 4

State 3
1 varref: T_ID dimlist .
2 dimlist: dimlist . '[' fact ']'

'[' shift, and go to state 5

$default reduce using rule 1 (varref)

State 4

0 $accept: varref $end .

$default accept

State 5
2 dimlist: dimlist '[' . fact ']'

T_ID shift, and go to state 1
T_NUM shift, and go to state 6
'-' shift, and go to state 7

varref go to state 8
fact go to state 9

State 6
5 fact: T_NUM .

$default reduce using rule 5 (fact)

State 7
6 fact: '-' . fact

T_ID shift, and go to state 1
T_NUM shift, and go to state 6
'-' shift, and go to state 7

varref go to state 8
fact go to state 10

State 8

4 fact: varref .

$default reduce using rule 4 (fact)

State 9

2 dimlist: dimlist '[' fact . ']'

']' shift, and go to state 11

State 10
6 fact: '-' fact .

$default reduce using rule 6 (fact)

State 11

2 dimlist: dimlist '[' fact ']' .

$default reduce using rule 2 (dimlist)

Input String	**Stack Contents**	**Action Taken**
$ T_ID [- T_NUM] [T_ID]	$ (0)	

Additional space for Problem 2.

Input String	**Stack Contents**	**Action Taken**

Problem 3 (5pts)

Design an LL grammar that is capable of recognizing C++-style declarations of containers. The containers that your grammar **must** support are: **vector**, **map**, **set**, and **pair**. The basic data-types to consider are: **int**, **float**, **string**.

(1pt) List **all** the tokens you will use in your grammar. You can use the examples in the 3rd, 4th and 5th part of this problem to identify the tokens.

(1pt) Define the rules for two non-terminals, *datatype* and *container_name*. These non-terminals will produce the majority of tokens defined above, but will not be the only ones.

(1pt) Create all the remaining rules for the grammar. You must use the above non-terminals. You should define a non-terminal *container* that permits to derive C++ declarations of template strings. Your grammar must meet all the requirements of an LL grammar. If necessary, you can use additional non-terminals, but please keep it to the minimum.

The following is an incomplete list of strings that should be accepted by your grammar:

- $vector < int > \quad grades$
- $vector < vector < int >> \quad matrix$
- $set < int > \quad numbers$
- $map < set < string >, pair < float, float >> \quad points$

(1pt) For the grammar you proposed in the previous step, show all the sentential forms for the string: $vector < set < int >> mytable$

(1pt) Enhance your proposed grammar with a rule (or rules) to produce pointers to containers. You must use the * (STAR) token operator. The grammar should allow to define single- (e.g. int *) and multi-level pointers (e.g. int ***).

Examples of acceptable strings are:

- $vector < int > *\ \ ii$
- $vector < int > * * *\ \ ii$
- $set < string > *\ \ aa$
- $set < string* > *\ \ aa$
- $map < int*, int >\ \ \ xy$

This page can be used as scratch area.