Midterm Exam
CS 314, Spring '14
March 14
SAMPLE SOLUTION

DO NOT OPEN THE EXAM UNTIL YOU ARE TOLD TO DO SO

Name: _____

Rutgers ID number: _____

Section: _____

WRITE YOUR NAME ON EACH PAGE IN THE UPPER RIGHT CORNER.

Instructions

We have tried to provide enough information to allow you to answer each of the questions. If you need additional information, make a *reasonable* assumption, write down the assumption with your answer, and answer the question. There are **5** problems, and the exam has **8** pages. Make sure that you have all pages. The exam is worth **250** points. You have **80 minutes** to answer the questions. Good luck!

This table is for grading purposes only

1	/ 50
2	/ 50
3	/ 50
4	/ 40
5	/ 60
total	/ 250

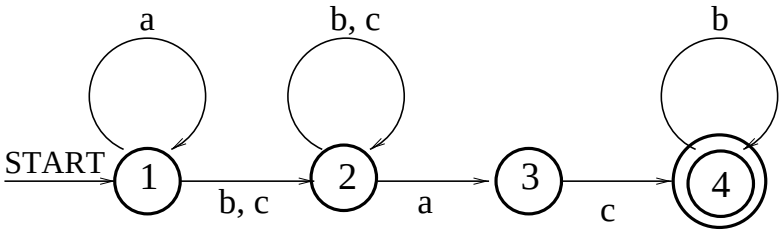
NAME: _____

Problem 1 – Regular Expressions, DFAs, and CFGs (50 pts)

1. Build a DFA (Deterministic Finite Automaton) that recognizes the language defined by the regular expression

$$\boxed{a^*\ (b|c)^+\ a\ c\ b^*}$$

Specify your DFA by extending the state transition diagram below. The start state is **state 1**, and the final (accepting) state is **state 4**. You are only allowed to add edges with their appropriate labels, i.e., valid labels are *a*, *b*, and *c*. Note that an edge may have more than one label. You may decide not to use state 2 and/or state 3, but you **must not** add any states. (20 pts)

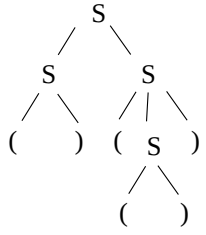


2. The following grammar generates the language of matching parenthesis. The start symbol is `S`

```
S ::= S S | ( S ) | ( )
```

(a) Is the string ( )(( )) in the language generated by the above grammar? Provide a proof for your answer by giving a left-most derivation (**use symbol** $\Rightarrow_L$) and a parse tree. (20 pts).

S $\Rightarrow_L$
S S $\Rightarrow_L$
( ) S $\Rightarrow_L$
( ) (S) $\Rightarrow_L$
( ) (( ))

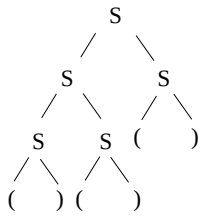


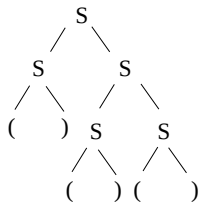
(b) Prove that this grammar is ambiguous by using parse trees (10pts).

A grammar is ambiguous if there is at least one string in the language generated by the grammar that has two distinct parse trees.

Claim: ( )( )( ) has two distinct parse trees.

Proof:





Problem 2 - LL(1) parsing and top-down parsing (50 pts)

Here is a context-free grammar for a boolean expression language. `Start` is the start symbol of the grammar, and { !, and, or, not, true, false } are the tokens (terminal symbols), with eof the special "token" signaling the end of the input.

1: Start ::= BoolExpr **!**
2: BoolExpr ::= **and** BoolExpr BoolExpr
3: BoolExpr ::= **or** BoolExpr BoolExpr
4: BoolExpr ::= **not** BoolExpr
5: BoolExpr ::= BoolOpnd
6: BoolOpnd ::= **true**
7: BoolOpnd ::= **false**

1. Give the LL(1) parse table for the grammar. **Insert the number of the rule or leave an entry empty**.(30 pts)

	!	**and**	**or**	**not**	**true**	**false**	**eof**
Start		1	1	1	1	1	
BoolExr		2	3	4	5	5	
BoolOpnd					6	7	

2. Write a **recursive descent parser** that evaluates the boolean expression either to *true* or *false*. Basically, this is an interpeter (not a compiler!) for the boolean expression language. You should use (imperative) pseudo code as we used in class. The function `next_token()` returns the next input token. Use the global variable `token` to hold the current token. If an error is detected, just call the function `exit` to terminate the execution of the parser. (20 pts)

```
Start() {
  token = next_token();
  switch token { case 'and', 'or', 'not', 'true', 'false': result = BoolExpr(); break;
                 default: exit;}
  if (token == '!') {
    token = next_token();
    if (token == 'eof')  print("Result:", result); else exit; }
  else exit; }

Boolean BoolExpr() {
  switch token { case 'and': token = next_token();
                             return BoolEpxr() AND BoolEpxr(); break;
                 case 'or': token = next_token();
                            return BoolEpxr() OR BoolEpxr(); break;
                 case 'not': token = next_token();
                             return NOT BoolEpxr(); break;
                 case 'true', 'false': return BoolOpnd(); break;
                 default: exit;}}

Boolean BoolOpnd() {
  switch token { case 'true': token = next_token(); return TRUE;
                 case 'false': token = next_token(); return FALSE; }}
```

NAME: _____

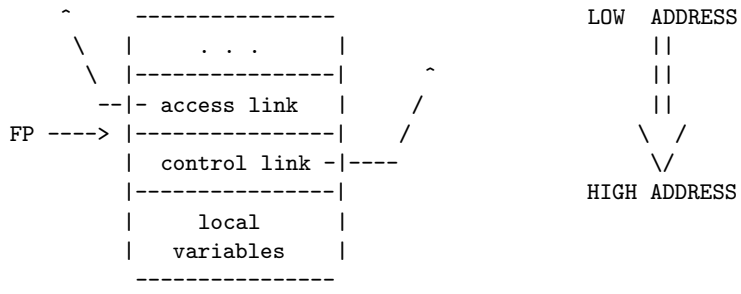
Problem 3 - Scoping (50 pts)

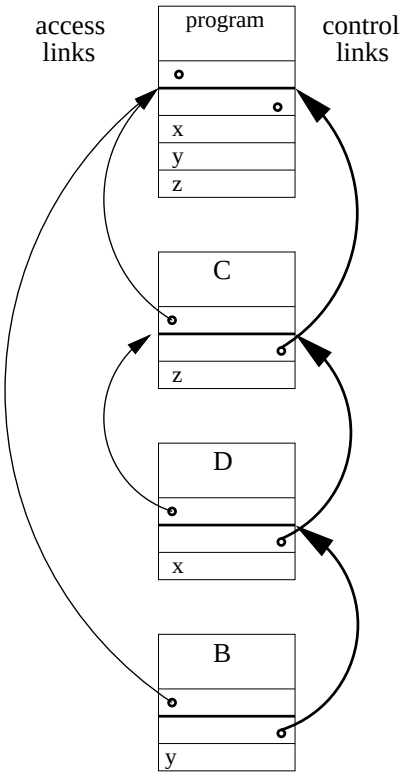
Assume the following program while answering the **three questions** on the next page.

```
Main A()
 {    x, y, z: integer;

      procedure B()
      {  y: integer;
         y=0;
         x=z+3;
         z=y+2;
                // <---  (*1*)
      }

      procedure C()
      {  z: integer;

         procedure D()
         {  x: integer;
            x = z + 2;
            y = x + 1;
            call B();
         }
         // statement body of C
         z = 7;
         call D();
      }
      // statement body of A
      x = 20;
      y = 21;
      z = 22;
      call C();
      print x, y, z;
 }
```

NAME: _____

1. Show the runtime stack when execution reaches the point marked (*1*) in the code. Each stack frame has to be labeled with its corresponding procedure name, and has to contain the names of locally allocated variables. **DO NOT** include the values of the variables in the stack frames. **DO** include all control and access links between stack frames. Use the following stack frame layout (template) as discussed in class, with arrows for pointer values to specific memory locations within the runtime stack. Program `Main` has its own stack frame.





2. Show the output of a program execution assuming **static (lexical) scoping** for all variables:

   `x=` 25, `y=` 10, `z=` 12

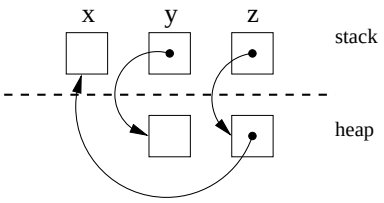
3. Show the output of a program execution assuming **dynamic scoping** for all variables:

   `x=` 20, `y=` 10, `z=` 22

NAME: _____

Problem 4 – Pointers and Memory Allocation in C (40 pts)

```
int main() {
  int   x;
  int  *y;
  int **z;

  z = (int **) malloc (sizeof(int *));
  y = (int *) malloc (sizeof(int));
  x = 0;
  *z = &x;
  *y = x;
  x = x + 5;
  **z = *y + 7;
  printf("x=%d, *y=%d, **z=%d\n", x, *y, **z);
  return 0;
}
```



1. What is the output of the above program?

   x=7, *y=0, **z=7

2. Specify, whether the following program objects are allocated on the **stack** (includes global variables), on the **heap**, or **not defined**.

   x is allocated on the stack
   y is allocated on the stack
   z is allocated on the stack
   *x is allocated on the not defined
   *y is allocated on the heap
   *z is allocated on the heap
   **y is allocated on the not defined
   **z is allocated on the stack

NAME: _____

Problem 5 – Compiler Project (60 pts)

Assume the following logical expression grammar:

<expr> ::= + <expr> <expr> | * <expr> <expr> | <const> | <var>
<const> ::= 1 | 2 | 3 | 4 | 5
<var> ::= a | b | c

instr. format	description	semantics
memory instructions		
`LOADI` R_x `#const`	load constant value #const into register R_x	$R_x \leftarrow$ const
`LOAD` R_x <id>	load value of variable <id> into register R_x	$R_x \leftarrow$ <id>
`STORE` <id> R_x	store value of register R_x into variable <id>	<id> $\leftarrow R_x$
logical instructions		
`ADD` R_x R_y R_z	add values in registers R_y and R_z , and store result into register R_x	$R_x \leftarrow R_y + R_z$
`MUL` R_x R_y R_z	multiply values in registers R_y and R_z , and store result into register R_x	$R_x \leftarrow R_y * R_z$

In your first programming project, you were ask to write a recursive descent compiler for the tinyL language. Here is code that generates code for a limited form of arithmetic expressions, but is written in the spirit of a solution of the first project.

```
int expr() {
        int reg, left_reg, right_reg;
        switch (token) {
        case '+': next_token();
                    left_reg = expr(); right_reg = expr(); reg = next_register();
                    CodeGen(ADD, reg, left_reg, right_reg);
                    return reg;
        case '*' : next_token();
                    left_reg = expr(); right_reg = expr(); reg = next_register();
                    CodeGen(MUL, reg, left_reg, right_reg);
                    return reg;
        case '1': case '2': case 3: case 4: case 5: return const();
        case 'a': case 'b': case 'c': return var();
        }
}

int const() { int reg;
              switch (token) {
              case '1': case '2': case '3': case '4': case '5':
                      next_token(); reg = next_register();
                      CodeGen(LOADI, reg, TokenToInt(token), EMPTY_FIELD); return reg; }}

int var() { int reg;
            switch (token) {
            case 'a': case 'b': case 'c':
                      next_token(); reg = next_register();
                      CodeGen(LOAD, reg, TokenToID(token), EMPTY_FIELD); return reg; }}
```

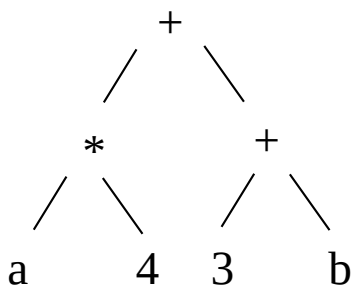
NAME: _____

Make the following assumptions:

- The value of variable "token" has been initialized correctly.
- **The first call to function `next_register()` the shown parser returns integer value "1".** In other words, the first register that the generated code will be using is register R_1.
- Your parser "starts" by calling function `expr()` on the entire input.

Show the code that the above specified recursive descent parser generates for the following input expression:

```
+ * a 4 + 3 b
```



Answer:

```
LOAD R1 a
LOADI R2 #4
MUL R3 R1 R2
LOADI R4 #3
LOAD R5 b
ADD R6 R4 R5
ADD R7 R3 R6
```