Midterm Exam
CS 314, Spring '17
March 8
Sample Solution version B

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 **6** 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	/ 30
2	/ 30
3	/ 60
4	/ 30
5	/ 40
6	/ 60
total	/ 250

NAME: ______________

Problem 1 - Regular Expressions and FSAs (30 pts)

Assume that *lower* stands for lower case letters, i.e., {a, b, c, ... z}, *upper* for upper case letters, i.e., {A, B, C, ... Z}, *digit* for digits, i.e., {0, 1, ... 9}, and *special* for special symbols, i.e., {\$, #, %, ... }. The following regular expression describes the set of valid identifiers in some programming language:

$$\boxed{(upper|lower)special^{*}digit^{+}special^{+}}$$

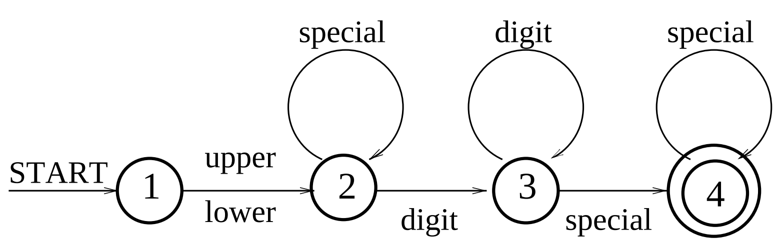
1. Give an example of a string of **length 7** that

   is a valid identifier in this language: A12345&

   is not a valid identifier in this language: doOr123

   (3 pts each)

2. Build a DFA (Deterministic Finite Automaton) for the regular expression of valid identifiers as defined above. 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 *lower*, *upper*, *digit*, and *special*. Note that an edge may have more than one label. (24 pts)



NAME: ______________

Problem 2 – Context Free Grammars and Regular Expressions (30 pts)

A *context-free language* is a language that can be specified using a context-free grammar. A *regular* language is a language that can be specified using a regular expression.

For the four languages given below, if the language is context-free, give a compact context-free grammar in Backus-Naur-Form (BNF). If the language is regular, give a compact regular expression using the regular expression syntax introduced in class. If a language is context-free and regular, give both specifications, a BNF and a regular expression. You do not have to justify why you believe a language is not context-free or not regular.

1. { $a^{3n}b^n$ | n $\geq$ 0 }, with alphabet Σ = {a, b}

   S ::= aaa S b |ϵ

2. { $a^{3n}b^{2m}$ | n $\geq$ 0, m >0 }, with alphabet Σ = {a, b}

   S ::= A B

   A ::= aaa A | ϵ

   B ::= bb B | bb

   $(aaa)^*(bb)^+$

3. { $w_1 w_2 w_2^R w_1^R$ | $w_1, w_2 \in \Sigma^*$ }, w^R is w in reverse , and alphabet Σ = {a, b}

   S ::= a S a | b S b | X

   X ::= a X a | b X b | ϵ

4. { w | w has at least 4 symbols}, with alphabet Σ={a, b}

   S ::= A B C D X

   A ::= a | b

   B ::= a | b

   C ::= a | b

   C ::= a | b

   X ::= a X | b X | ϵ

   (a | b) (a | b) (a | b) (a | b)$^+$

NAME: ________________

Problem 3 – Context Free Grammars (60 pts)

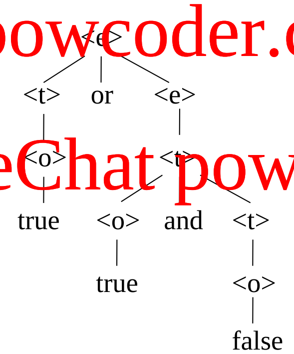
Assume the following context-free logical expression grammar **in BNF** over the alphabet (set of tokens) Σ = { **true**, **false**, **or**, **and** } with the **start symbol** <e>.

```
<e> ::= <t> or <e> | <t>
<t> ::= <o> and <t> | <o>
<o> ::= true | false
```

1. Give a left-most derivation ($\Rightarrow_L$) for the sentence `true or true and false`.

   <e> $\Rightarrow_L$
   <t> or <e> $\Rightarrow_L$
   <o> or <e> $\Rightarrow_L$
   true or <e> $\Rightarrow_L$
   true or <t> $\Rightarrow_L$
   true or <o> and <t> $\Rightarrow_L$
   true or true and <t> $\Rightarrow_L$
   true or true and <o> $\Rightarrow_L$
   true or true and false

2. Show the corresponding parse tree for your left-most derivation.



3. Is the grammar LL(1)? Justify your answer using $FIRST^+$ sets.

   The grammar is not LL(1). For example, for the two rules for nonterminal <e>, $FIRST^+$(<t> or <e>) = {true, false}, and $FIRST^+$(<t>) = {true, false}. Therefore, these sets are not disjoint, which is a requirement for the grammar to be LL(1).

4. What is the associativity of the `or` operator as specified in the grammar? right associative

   What is the associativity of the `and` operator as specified in the grammar? right associative

5. Does the grammar implement the precedence of the `or` operator over the `and` operator (`or` binds stronger than `and`)? If not, give a minimal change to the grammar (fewest number of changed rules) such that `or` has precedence over `and`. In addition, change the grammar rules, if necessary, to make `and` left associative.

```
<e> ::= <e> and <t> | <t>
<t> ::= <o> or <t> | <o>
<o> ::= true | false
```

NAME: ________________

Problem 4 - Scoping (30 pts)

Assume the following program while answering the questions below.

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

      procedure D()
       {
          x = z + 2;
       }
      procedure C()
       {
          z = 3;
          x = z + 4;
          call D();
       }
      procedure B()
       {  x, z: integer;

          z = 1;
          x = y + z;
          call C();
       }

      // statement body of A
      x = 6;
      y = 7;
      z = 8;
      call B();
      print x, y, z;
 }
```


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

   x= ___5___ , y= ___7___ , z= ___3___

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

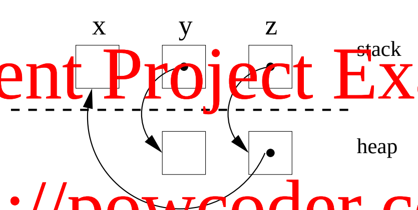
   x= ___6___ , y= ___7___ , z= ___8___

NAME: ______________

Problem 5 – 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 = 2;
  *z = &x;
  *y = x;
  x = x + 3;
  **z = *y + 3;
  printf("x=%d, *y=%d, **z=%d\n", x, *y, **z);
  return 0;
}
```



1. What is the output of the above program? (5 pts each)

   x=<u>5</u>, *y= <u>2</u>, **z= <u>5</u>

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

   x is allocated on the <u>stack</u>

   y is allocated on the <u>stack</u>

   z is allocated on the <u>stack</u>

   *x is allocated on the <u>not defined</u>

   *y is allocated on the <u>heap</u>

   *z is allocated on the <u>heap</u>

   **y is allocated on the <u>not defined</u>

   **z is allocated on the <u>stack</u>

3. Assume the following code segment:

```
int *x;
*x = 5;
printf("%d\n", *x);
```

   Is there a problem with this code? Assume that when you ran the code a couple of times, it printed "5". If you believe there is a problem, give a possible "fix" for the problem? (9 pts)

The content of variable `x` is not initialized. However, its content is used as an address of a memory location, and that memory location is assigned the value `5`.

To fix the problem, `x` should point to an object on the heap that is allocated as follows:

```
x = (int *) malloc(sizeof(int *))
```

This statement should be placed before statement `*x = 5`.

NAME: ______________

Problem 6 – Syntax-Directed Translation (60 pts)

Assume the following logical expression grammar:

<expr>	::=	*and* <expr> <expr> |
		<const>
<const>	::=	true | false

instr. format	description	semantics
	memory instructions	
`LOADI #true` $\Rightarrow r_x$	load constant value #true into register r_x	$r_x \leftarrow$ true
`LOADI #false` $\Rightarrow r_x$	load constant value #false into register r_x	$r_x \leftarrow$ false
	logical instructions	
`AND` r_x , $r_y \Rightarrow r_z$	'and' logical operation on truth values in registers r_x and r_y , and store result into register r_z	$r_z \leftarrow r_x$ *and* r_y

Here is a recursive descent parser that implements a compiler for the above grammar. Note that 'and', 'true',and 'false' are the new tokens. Here is the important part of the code:

```
int expr() {
        int reg, left_reg, right_reg;
        switch (token) {
        case 'and': next_token();
                    left_reg = expr(); right_reg = expr(); reg = next_register();
                    CodeGen(AND, left_reg, right_reg, reg);
                    return reg;
        case 'true':
        case 'false': return const();
        }
}

int const() {
        int reg;
        switch (token) {
        case 'true':  next_token(); reg = next_register();
                      CodeGen(LOADI, true, reg);
                      return reg;
        case 'false': next_token(); reg = next_register();
                      CodeGen(LOADI, false, reg);
                      return reg;
        }
}
```


Make the following assumptions:

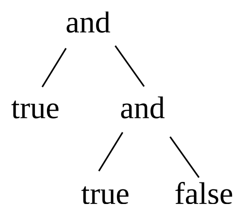
- The value of variable "token" has been initialized correctly.
- The function `CodeGen` has been extended to deal with logical binary operations and logical arguments.
- **The first call to function `next_register()` the shown parser returns integer value "1".** In other words, the first register that the generated code will be using is register r_1.
- Your parser "starts" by calling function `expr()` on the entire input.

NAME: ______________

1. Show the code that the recursive descent parser generates for input

`and true and true false`

will produce:




```
loadI #true => r1
loadI #true => r2
loadI #false => r3
and r2, r3 => r4
and r1, r4 => r5
```

2. Change the basic recursive-descent parser to implement an interpreter for our example language. You may insert pseudo code in the spaces marked by ______. You may also assume the availability of a type `boolean`. No error handling is necessary. **There are many possible solutions.**


```
boolean  expr() {

       boolean bval1, bval2, bval;
       switch (token) {
       case 'and': next_token();
                   bval1 = expr(); bval2 = expr();

                   bval = bval1 && bval2; // && is ``and'' operator

                   return bval;
       case 'true':
       case 'false': return const();
       }
}

boolean const() {

       boolean bval;
       switch (token) {
       case 'true':  next_token();

                     bval = TRUE; // TRUE is boolean constant for ``true''

                     return bval;

       case 'false': next_token();

                     bval = FALSE; // FALSE is boolean constant for ``false''

                     return bval;
       }
}
```