Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.035 Fall 2018

Test I

You have 50 minutes to finish this quiz.

Write your name and athena username on this cover sheet.

Some questions may be harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

This exam permits only one double sided cheat sheet.

Please do not write in the boxes below.

I (xx/15)	II (xx/30)	III (xx/10)	IV (xx/12)	V (xx/20)	Total (xx/87)

Name:

Athena username:

I Regular Expressions and Finite-State Automata

For Questions 1 through 3, let the alphabet $\Sigma = \{., 0, 1\}$. Let language L be the language of all strings over Σ where any "1" character is not followed by a "1" character.

1. [5 points]: Write a regular expression that recognizes language L.

2. [5 points]: Draw a state diagram of a nondeterministic finite-state automaton (NFA) that recognizes language L. Remember to indicate starting and accepting states.

3. [5 points]: Draw a state diagram of a deterministic finite-state automaton (DFA) that recognizes language L. Note that you can either build a DFA directly from the English description or convert your NFA into a DFA. Remember to indicate starting and accepting states.

II Parsing

Consider the following grammar,

$$\begin{aligned} S &\rightarrow X\ \$ \\ X &\rightarrow Y\ +\ Y \\ Y &\rightarrow \text{num} \end{aligned}$$

where \$ indicates that the end of the input has been reached.

4. [5 points]: List the items generated by the grammar above.

5. [10 points]: Draw a DFA corresponding to the grammar above using the items in problem 4. Please specify which items belong to each state.

6. [10 points]: Complete the entries in the following parse table for the DFA in problem 5.

	Action			Goto	
State	+	num	$	X	Y

7. [5 points]: The string 5 + 6$ is parsed using a shift-reduce parser and the grammar above. Draw the stack after the second reduce operation. Please mark where the stack begins.

III Control Flow

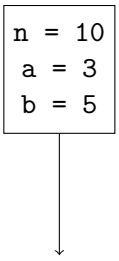
Consider a programming language that includes a control flow construct called a "goto". Given a set of labels and statements, $l1 : s1, l2 : s2, ...ln : sn$, goto is written as follows:

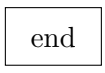
```
li: si
if (c1) goto lj
// statements
lj: sj
if (c2) goto li
// statements
```

If the condition in the if statement is True then control flows to the line specified after the goto. Otherwise, control flows to the following statement. Statements are then evaluated sequentially. Note that the line specified by the goto may occur before or after the goto statement.

8. [10 points]: The semantics of the programming language say that a compiled program should only evaluate expressions until the first match with the control expression's value is found. The program evaluates a compound condition from left to right. Recall that a control flow graph consists of nodes representing maximal basic blocks and edges representing control flow. No branches may come out of or into the middle of a basic block. Complete the control flow graph on the next page that illustrates the control flow for evaluating the following statements, including short-circuit logic for conditionals, assuming the compiler is not performing any optimizations:

```
int n = 0;
int a = 3;
int b = 5;
l1:  a = n-b;
a += 1;
if ((n % 2 == 0) || a == b ) goto l1
b = n;
if (b > a) goto l2
b = a + 1;
l2: a = 1;
```


n = 10
a = 3
b = 5


end

IV Short Circuiting

9. [12 points]: In the lecture, we discussed the implementation of procedures called `destruct`, `next` and `shortcircuit`.

The procedure `destruct(n)` generates the control-flow representation for structured code represented by `n`. This procedure creates a control flow graph for `n` and returns the begin and end nodes of the graph.

The procedure `next(n1) = n2` allows you to specify `n2` as the subsequent control-flow node to be executed after `n1`.

The procedure `shortcircuit(c, t, f)` generates the short-circuit control-flow representation for a conditional `c`. This procedure makes the control flow to node `t` if `c` is true and flow to node `f` if `c` is false. The procedure returns the begin node for evaluating condition `c`.

Recall that the pseudocode of `destruct(n)` for an if-else statement is as follows:

If `n` is of the form `if (c) { x1 } else { x2 }` then

```
e = new nop
(b1, e1) = destruct(x1)
(b2, e2) = destruct(x2)
bc = shortcircuit(c, b1, b2)
next(e1) = e
next(e2) = e
return (bc, e)
```

Recall that a NAND b evaluates to False if both a and b are True, and evaluates to True otherwise. Also recall that a NOR b evaluates to True if both a and b are False and evaluates to False otherwise. A ternary expression `a?b:c` evaluates b if a is True and evaluates c if a is False. Implement the following functions using `shortcircuit`. You may find it helpful to draw the control flow graph of each condition.

A. `nand_shortcircuit(c1 NAND c2, t, f){`

`}`

B. `nor_shortcircuit(c1 NOR c2, t, f){`

`}`

C. `ternary_shortcircuit(c1 ? c2: c3, t, f){`

`}`

V Code Generation for Procedures

10. [8 points]:

You want to flatten the following lines into temps in your nascent compiler so your code generation procedure is ready to write them out.

Linearize the following statements, with a new temporary for each intermediate and each expression as a single 3-address operation.

a = x + y;
b = a * (c + d*3);

// started for you below

```
t1 = x
t2 = y
t3 = t1 + t2
a = t3
```

11. [4 points]: You've written your `foo` function in Decaf (with the added ability to declare and set a variable's initial value in one statement):

```
int foo(int x) {
  int y = x;
  y = 1024 / y;
  return y + 1;
}
```

For which your (unoptimized) compiler outputs the following:

```
_foo:
  pushq   %rbp
  movq    %rsp, %rbp
  movl    $1024, %eax

  movq    %rdi, -8(%rbp)
  movq    -8(%rbp), %rdi
  movq    %rdi, -16(%rbp)
  cqto
  idivq   -16(%rbp)
  movq    %rax, -16(%rbp)
  movq    -16(%rbp), %rax
  movl    %eax, %ecx
  movl    $1, %eax
  addl    %eax, %ecx
  movl    %ecx, %edi
  popq    %rbp
  retq
```

Does the `foo` function obey standard calling convention by placing the return value in `%edi`? Why or why not?

12. [8 points]: Here's another function, `bar`.

```
void bar(int x) {
  int y = x;
  int a = y * y;
  a = a / 2;
  int z = a + y;
}
```

```
_bar:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $2, %eax
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %r12
        movl    %r12, -8(%rbp)
        movl    -8(%rbp), %r12
        imull   -8(%rbp), %r12
        movl    %r12, -12(%rbp)
        movl    -12(%rbp), %r12
        movl    %eax, -20(%rbp)
        movl    %r12, %eax
        cltq
        movl    -20(%rbp), %edi
        idivl   %edi
        movl    %eax, -12(%rbp)
        movl    -12(%rbp), %eax
        addl    -8(%rbp), %eax
        movl    %eax, -16(%rbp)
        retq
```

For each variable, designate whether it is found in a register or on the stack. If it is on the stack, specify its offset on the stack from `%rbp`. Use the register or offset that the variable has exclusive use of.

Variable: `a` **Register** **Stack** offset: _____

Variable: `x` **Register** **Stack** offset: _____

Variable: `y` **Register** **Stack** offset: _____

Variable: `z` **Register** **Stack** offset: _____

OTHERWISE BLANK PAGE

APPENDIX I: BROWN CSC10330 X64 HANDOUT GUIDE
APPENDIX II: Notre Dame Introduction to X86 Assembly for Compiler Writers