Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.035 Spring 2016

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 is open book and open laptop. Additionally, you may access the course website, but aside from that you may NOT USE THE NETWORK.

Please do not write in the boxes below.

I (xx/25)	II (xx/28)	III (xx/22)	IV (xx/25)	Total (xx/100)

Name:

Athena username:

I Regular Expressions and Finite-State Automata

For Questions 1, 2, and 3, let the alphabet $\Sigma = \{a, b\}$. Let language L be the language of all strings over Σ that end with the substring ab.

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

2. [10 points]: Draw a state diagram of a nondeterministic finite-state automaton (NFA) that recognizes language L. Remember to indicate starting and accepting states.

3. [10 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 Context-Free Grammars

Alice thinks top-down parsing is cool and wants to design her own top-down parsing algorithm. She wants her parser to use the rightmost derivation that expands the rightmost nonterminal symbol first.

To allow for efficient predictive parsing, Alice defines sets named *Last*. A terminal symbol $T \in \mathit{Last}(\beta)$, where β is a sequence of terminals or nonterminals, if T can appear as the last symbol in a derivation starting from β. The fixed-point definition for *Last* is as follows:

$$\begin{aligned}
&T \in \mathit{Last}(T)\\
&\mathit{Last}(S) \subseteq \mathit{Last}(\beta\ S)\\
&NT \text{ derives } \epsilon \text{ implies } \mathit{Last}(\beta) \subseteq \mathit{Last}(\beta\ NT)\\
&NT \to \beta\ S \text{ implies } \mathit{Last}(\beta\ S) \subseteq \mathit{Last}(NT)
\end{aligned}$$

where NT is a nonterminal and S is a terminal or nonterminal.

Consider how Alice's parser would approach the following grammar:

$$\begin{array}{rcl}
S & \to & X\,Y\\
X & \to & c\,X\\
X & \to & Y\\
Y & \to & d\,Y\\
Y & \to & \epsilon
\end{array}$$

Here, uppercase letters S, X, Y are nonterminals. Lowercase letters c, d are terminals. ϵ is the empty string.

4. [4 points]: What is the set of all nonterminal symbols that can derive ϵ?

5. [5 points]: Compute the minimal solutions for *Last*:

$$Last(S) = \{ \qquad\qquad\qquad\qquad \}$$

$$Last(X) = \{ \qquad\qquad\qquad\qquad \}$$

$$Last(Y) = \{ \qquad\qquad\qquad\qquad \}$$

$$Last(c\ X) = \{ \qquad\qquad\qquad\qquad \}$$

$$Last(d\ Y) = \{ \qquad\qquad\qquad\qquad \}$$

6. [5 points]: Describe a situation where Alice's parser may enter an infinite loop or infinite recursion when parsing this grammar.

7. [8 points]: Show that the grammar is ambiguous by providing:

A. A sentence in the language with two parse trees.

B. The two parse trees.

8. [6 points]: Provide a new grammar that expresses the same language but does not have either problem above.

- **A.** Write a grammar that is unambiguous and would not cause Alice's parser to enter infinite recursion.
- **B.** Briefly explain why your new grammar defines the same language.

III Short-Circuiting Conditionals

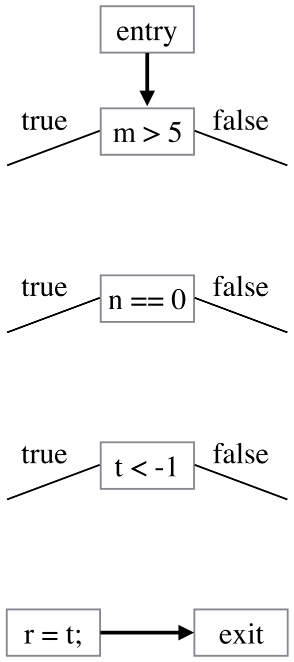
Consider introducing the boolean operation NAND as "`!&`" to a programming language. The "`!&`" operation has the following truth table:

`p`	`q`	`p !& q`
false	false	true
false	true	true
true	false	true
true	true	false

9. [2 points]: What is an equivalent boolean expression for `(p !& q)` that uses only the AND ("`&&`"), OR ("`||`"), or NOT ("`!`") operators?

10. [10 points]: The semantics of the programming language says that a compiled program should execute only as much as required to determine the condition. The program evaluates a compound condition from left to right. Complete the flowchart that illustrates the control flow for evaluating the following statements:

```
if (((m > 5) !& (n == 0)) || (t < -1)) {
    r = t;
}
```



11. [10 points]: In the lecture, we discussed the implementation of a procedure called `shortcircuit`. 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 `shortcircuit(c, t, f)` for AND, OR, and NOT are as follows.

A. If `c` is of the form `(c1 && c2)` then

```
b2 = shortcircuit(c2, t, f);
b1 = shortcircuit(c1, b2, f);
return b1;
```

B. If `c` is of the form `(c1 || c2)` then

```
b2 = shortcircuit(c2, t, f);
b1 = shortcircuit(c1, t, b2);
return b1;
```

C. If `c` is of the form `(!c1)` then

```
b = shortcircuit(c1, f, t);
return b;
```

Implement the pseudocode of `shortcircuit(c, t, f)` for NAND:

If `c` is of the form `(c1 !& c2)` then

IV Code Generation for Procedures

Consider the following function and its corresponding assembly code generated by the GCC compliler:

```
long foo(long a, long b) {
  long x = a + b;
  return x;
}
```

```
pushq %rbp           // push the value of %rbp to the stack
movq  %rsp, %rbp     // copy the value of %rsp to %rbp
addq  %rsi, %rdi     // increment the value of %rdi by %rsi
movq  %rdi, %rax     // copy the value of %rdi to %rax
popq  %rbp           // pop the top value from the stack to %rbp
retq                 // return from the function
```

The GCC compiler follows the standard x86-64 calling convention. Specifically, a caller function uses registers to pass the first 6 arguments to the callee. Given the arguments in left-to-right order, the order of registers used is: %rdi, %rsi, %rdx, %rcx, %r8, and %r9. The callee is responsible for perserving the value of registers %rbp %rbx, and %r12-r15, as these registers are owned by the caller. The remaining registers are owned by the callee. The callee places its return value in %rax.

12. [6 points]: Express the following values in terms of the values of `a` and `b` passed to `foo()`:

A. [2 points]: What is the value in %rsi after the line 2 executes?

B. [2 points]: What is the value in %rdi after the line 3 executes?

C. [2 points]: What is the value in %rax after the line 4 executes?

13. [7 points]: Bob's compiler seems to have a bug. When his compiler combines the generated assembly code into the assembly code of other functions that the GCC compiler generated, the combined program does not work as expected. Bob's compiler generates the following assembly code for function `foo()`:

```
pushq %rbp
movq %rsp, %rbp
movq %rsi, %rbx
addq %rdi, %rbx
movq %rbx, %rax
popq %rbp
retq
```

Is there anything wrong with this piece of code? Explain.

14. [12 points]: You are a member of a team designing the calling convention for a new microprocessor. You have been put on the team to ensure that the following computation, which is critical to the success of the microprocessor, runs fast. As with many microprocessors, the ability of the compiler to store values in registers (as opposed to storing and reloading values from memory such as the call stack) can be critical to good performance.

```
int f(int x, int y) {
 return x + y;
}

int g(int n) {
 int i = -n;
 int j = 0;

 while (i < 0) {
   i = i + 1;
   j = j + f(-i, j);
 }
 return j;
}
```

Consider the following design choices:

A. [**6 points**]: One of the members of the team argues that since almost all of the computation happens in functions that do not call any other functions, there should be no callee save registers at all.

Is this is a good design for your computation? Why or why not?

B. [6 points]: One of the members of the team argues for a calling convention that passes the return value from the callee to the caller in a register. This person also argues that this register should be a callee-save register (owned by the caller) to maximize the ability of the caller to store values in registers without interference from the callee.

Is this a good design for your computation or not? Why or why not?