Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.035 Fall 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/20)	II (xx/20)	III (xx/20)	IV (xx/20)	V (xx/20)	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 contains the substring "aa" or "bb".

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. [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 Ambiguous Grammar

For each of the following grammars, state if it is ambiguous.

If the grammar is ambiguous, find a sentence in the language with two (or more) parse trees, and show the two parse trees.

Every lowercase letter indicates a terminal, and every uppercase letter indicates a non-terminal. Parsing starts at S.

4. [5 points]:

$$
\begin{aligned}
S &\rightarrow S\ c \\
S &\rightarrow c\ S \\
S &\rightarrow d
\end{aligned}
$$

5. [5 points]:

$$
\begin{aligned}
S &\rightarrow F + S \\
S &\rightarrow F \\
F &\rightarrow c \\
F &\rightarrow (\ S\ )
\end{aligned}
$$

6. [5 points]:

$$
\begin{aligned}
S &\rightarrow S + S \\
S &\rightarrow F \\
F &\rightarrow c \\
F &\rightarrow (S)
\end{aligned}
$$

7. [5 points]:

$$
\begin{aligned}
S &\rightarrow c \ ? \ S \ : \ S \\
S &\rightarrow c
\end{aligned}
$$

III Left Recursion

Consider the following grammar:

$$\begin{aligned} S &\rightarrow T\,\$ \\ T &\rightarrow T\,A \\ T &\rightarrow \epsilon \\ A &\rightarrow v = c \end{aligned}$$

The following is an implementation (in a C-like language) of a recursive descent parser. Note that a parse function returns true if it successfully parses a rule given the input stream, false otherwise. Assume all input ends with exactly one dollar sign "$".

```
1  bool parseS() {
2    if (parseT()) {
3      if (isDollarSign()) {
4        return true;
5      }
6    }
7    return false;
8  }
9
10 bool parseT() {
11   if (isV(token)) {
12     if (parseT()) {
13       if (parseA()) {
14         return true;
15       }
16     }
17     return false;
18   } else {
19     return true;
20   }
21 }
22
23 bool parseA() {
24   if (isV(token)) {
25     token = nextToken();
26     if (isEqual(token)) {
27       token = nextToken();
28       if (isC(token)) {
29         token = nextToken();
30         return true;
31       }
32     }
33   }
34   return false;
35 }
```

8. [6 points]: The recursive descent parser enters infinite recursion because S is left recursive. Redesign the grammar of this language to eliminate left recursion.

9. [14 points]: Write the new code for the recursive descent parser for the new grammar.

IV Control Flow and Short-Circuiting

Consider a programming language that includes a control flow construct called the "repeat-until" loop. A repeat-until loop is written as follows:

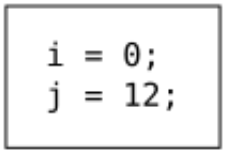
```
repeat {
    // body statements
} until (condition)
```

The repeat-until loop runs the code in the loop body, and then checks the condition. **If the condition evaluates to true, the loop ends;** otherwise, the loop repeats. Note that even if the condition is always true, the loop body will still run once.

10. [10 points]: The semantics of the programming language says that a compiled program should execute only as much as required to determine the value of a boolean condition. The program evaluates a compound condition from left to right. Complete the flowchart 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 i = 0;
int j = 12;
repeat {
    i += j;
    j = j * 2;
} until (i == 36 || (i >= 0 && j > 10))
```

(Hint: after this code runs, your final values should be `i = 12` and `j = 24`.)


i = 0;
j = 12;


end

11. [10 points]: In the lecture, we discussed the implementation of procedures called `shortcircuit` and `destruct`.

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`.

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.

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);
```

Implement the pseudocode of `destruct(n)` for a repeat-until loop:

If `n` is of the form `repeat { x } until (c)` then

V Code Generation for Procedures

Consider the following function in C and its corresponding assembly code generated by a compliler. Note that `long` is a 64-bit integer.

```
long bar(long x) {
  return x+x;
}
```

```
1 pushq   %rbp           // push the value of %rbp to the stack
2 movq    %rsp, %rbp     // copy the value of %rsp to %rbp
3 movq    %rdi, -8(%rbp) // copy the value of %rdi to the stack
4 movq    -8(%rbp), %rdi // copy the value on the stack to %rdi
5 addq    -8(%rbp), %rdi // add the value on the stack to %rdi
6 movq    %rdi, %rax     // copy the value of %rdi to %rax
7 popq    %rbp           // pop the top value from the stack to %rbp
8 ret                    // return from the function
```

The compiler follows the standard Linux x86-64 calling convention:

> A caller procedure/function passes the first 6 arguments, from left to right, in %rdi, %rsi, %rdx, %rcx, %r8, %r9. Any remaining arguments are passed on the stack, from right to left.
>
> The caller owns registers %rsp, %rbp, %rbx, and %r12-%r15. The callee procedure/function is responsible for ensuring that these registers have the same value after the call as before the call. Note that %rsp and %rbp are the stack and base registers. Registers %rsp, %rbp, %rbx, and %r12-%r15 are the *callee-save* registers.
>
> The callee owns the remaining registers %rax, %rcx, %rdx, %rsi, %rdi, and %r8-%r11. These registers can have different values after the call as before the call. These registers are the *caller-save* registers.
>
> The callee places its return value in %rax.

Which of the following possible generated code sequences for `bar` are correct in the sense that 1) they compute the correct return value for `bar` and 2) they follow the the standard Linux x86-64 calling convention? Provide your answer by circling either Correct or Incorrect below each code sequence.

12. [5 points]:

```
1 pushq   %rbp           // push the value of %rbp to the stack
2 movq    %rsp, %rbp     // copy the value of %rsp to %rbp
3 movq    %rdi, -8(%rbp) // copy the value of %rdi to the stack
4 addq    %rdi, %rdi     // add the value of %rdi to %rdi
5 movq    %rdi, %rax     // copy the value of %rdi to %rax
6 popq    %rbp           // pop the top value from the stack to %rbp
7 retq                   // return from the function
```

Correct Incorrect

13. [5 points]:

```
1 movq    %rdi, -8(%rsp) // copy the value of %rdi to the stack
2 addq    -8(%rsp), %rdi // add the value on the stack to %rdi
3 movq    %rdi, %rax     // copy the value of %rdi to %rax
4 retq                   // return from the function
```

Correct Incorrect

The code in the next two questions is also compiled by a compiler that adheres to the Linux x86-64 calling convention, which is repeated below for your convenience.

> A caller procedure/function passes the first 6 arguments, from left to right, in %rdi, %rsi, %rdx, %rcx, %r8, %r9. Any remaining arguments are passed on the stack, from right to left.
>
> The caller owns registers %rsp, %rbp, %rbx, and %r12-%r15. The callee procedure/function is responsible for ensuring that these registers have the same value after the call as before the call. Note that %rsp and %rbp are the stack and base registers. Registers %rsp, %rbp, %rbx, and %r12-%r15 are the *callee-save* registers.
>
> The callee owns the remaining registers %rax, %rcx, %rdx, %rsi, %rdi, and %r8-%r11. These registers can have different values after the call as before the call. These registers are the *caller-save* registers.
>
> The callee places its return value in %rax.

The compiler will try to allocate variables in registers to minimize movement between memory and registers. In other words, it will decide that the value of a specific variable should be stored in a specific register, then access that value from that register directly. The goal is to ensure that as many values are accessed from registers as possible and to minimize any need to save and restore registers to and from memory.

14. [6 points]: Into what registers should the compiler allocate `n`, `i`, `x`, when it compiles the procedure `g()` below? If there are multiple equivalent register assignments, you only need to write one.

Assume that functions `g()` and `f()` are compiled separately.

```
1  long g() {
2    long n = 100;
3    long i = 0;
4    long x = 0;
5    while (i < n) {
6      x = x + f(i);
7      i = i + 1;
8    }
9    return x;
10 }
```

Allocate `n` in register:

Allocate `i` in register:

Allocate `x` in register:

15. [4 points]: Into what registers should the compiler allocate `j` and `y`, when it compiles the procedure `f()` below?

```
1 long f(long j) {
2   long y = j * j;
3   y = j + 10;
4   return y;
5 }
```

Allocate `j` in register:

Allocate `y` in register: