Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.035 Fall 2017

Test I Solutions

Mean 83.63 Median 86 Std. dev 12.03

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 where any "a" character is followed by at least two "b" characters.

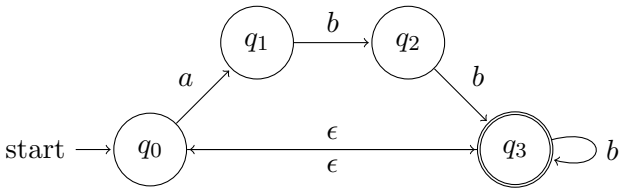
1. [5 points]: Write a regular expression that recognizes language L.

Solution: $(abb|b)^*$ **Rubric:**

- -1 for each category of string accepted but shouldn't
- -1 for each category of string not accepted but should

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.

Solution: See Problem 3. All DFA are NFA. Alternative solution:

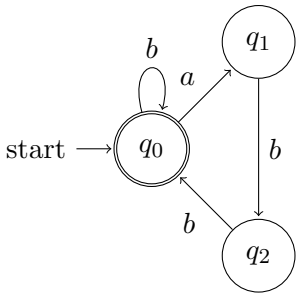


Rubric:

- -2 for not accepting L for same reason as problem 1.
- -1 for each category of string not accepted by L or above Regex

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.

Solution:



Rubric:

- Same as problem 2

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

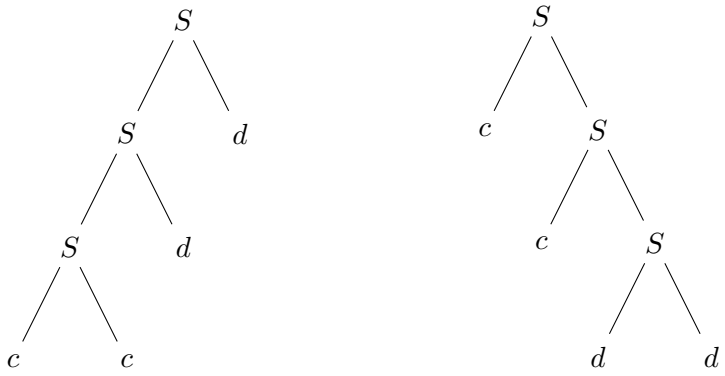
Rubric: For each of the next four problems

- +5 If not ambiguous correctly stated
- +2 If ambiguous correctly stated, +1 for an ambiguous status, +1 for each correct parse tree.

4. [5 points]:

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

Solution: Ambiguous. Example: *ccdd*



5. [5 points]:

$$
\begin{aligned}
S &\rightarrow T \,||\, S \\
S &\rightarrow U \\
U &\rightarrow T \,\&\&\, U \\
U &\rightarrow T \\
T &\rightarrow c
\end{aligned}
$$

Solution: Not ambiguous.

6. [5 points]:

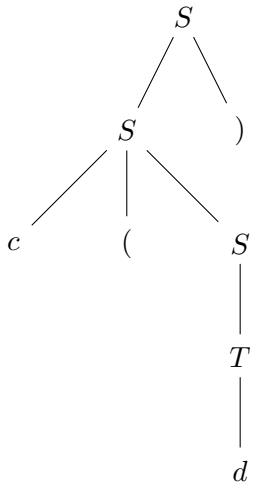
$$
\begin{aligned}
S &\rightarrow c\,(\,T \\
T &\rightarrow S\,) \\
T &\rightarrow d\,,\,T \\
T &\rightarrow d\,)
\end{aligned}
$$

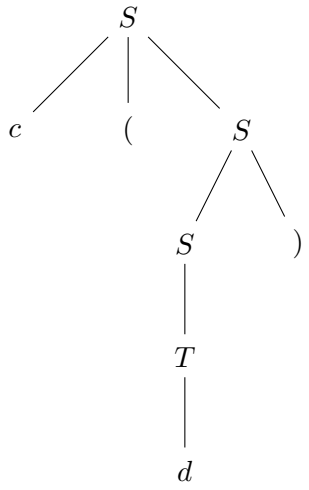
Solution: Not ambiguous.

7. [5 points]:

$$
\begin{aligned}
S &\rightarrow c\,(\,S \\
S &\rightarrow S\,) \\
S &\rightarrow T \\
T &\rightarrow d\,,\,S \\
T &\rightarrow d
\end{aligned}
$$

Solution: Ambiguous. Example: $c(d)$





III Implementing Object-Orientation: Descriptors and Symbol Tables

Use the diagram on the next page to answer the following three questions about this fragment of code.

```
1  class BinaryOperation {
2      int left;
3      int right;
4      int eval(){ return 0; }
5  }
6
7  class Plus extends BinaryOperation {
8      int eval(){ return left + right; }
9  }
10
11 class Divide extends BinaryOperation {
12     bool isDivisible;
13     int getRemainder(){
14       isDivisible = (left % right == 0);
15       return left % right;
16     }
17     int eval(){ return left/right; }
18 }
```

8. [7 points]: Complete the entries of the class descriptors for each class. Use an arrow to connect the entry to a descriptor or symbol table where appropriate.

Solution: Rubric:

- +1 for writing parents.
- +1 for each parent arrow.
- +1 for writing fields and +1 for their arrows
- +1 for writing methods and +1 for their arrows

9. [7 points]: Complete the entries of the field symbol tables for each class. Use an arrow to connect the entry to a descriptor or symbol table where appropriate.

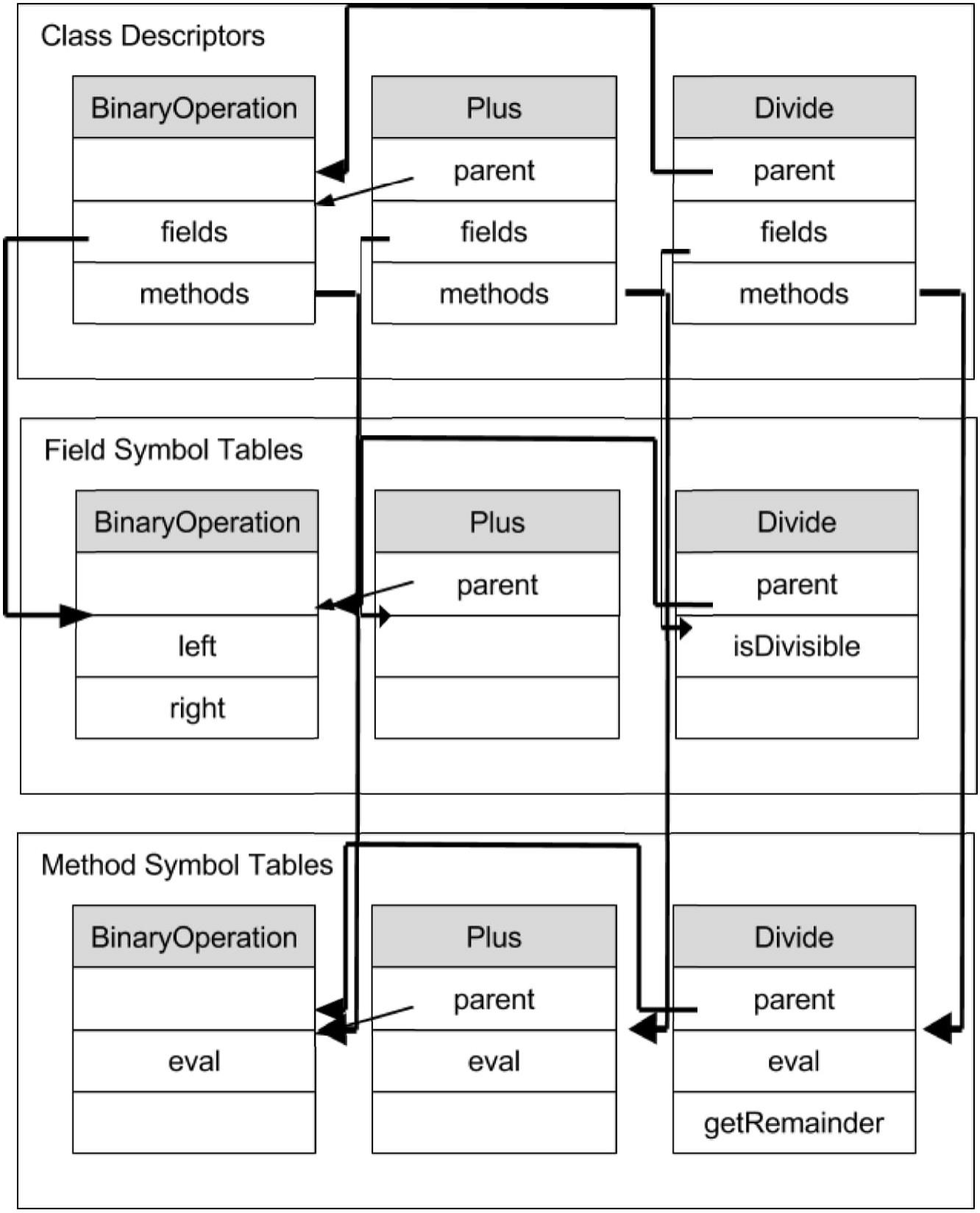
Solution: See Below **Rubric:**

- +1 for writing parent with arrow.
- +2 for left field.
- +2 for right field.
- +2 for isDivisible field.

10. [7 points]: Complete the entries of the method symbol tables for each class. Use an arrow to connect the entry to a descriptor or symbol table where appropriate.

Solution: Rubric:

- +1 for writing parents.
- +1 for each parent arrow.
- +1 for each eval
- +2 for getRemainder


Class Descriptors
BinaryOperation
Plus
Divide
parent
parent
fields
fields
fields
methods
methods
methods
Field Symbol Tables
BinaryOperation
Plus
Divide
parent
parent
left
isDivisible
right
Method Symbol Tables
BinaryOperation
Plus
Divide
parent
parent
eval
eval
eval
getRemainder

IV Control Flow and Short-Circuiting

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

```
loop {
    // first body statements
} while test {
    // second body statements
} repeat;
```

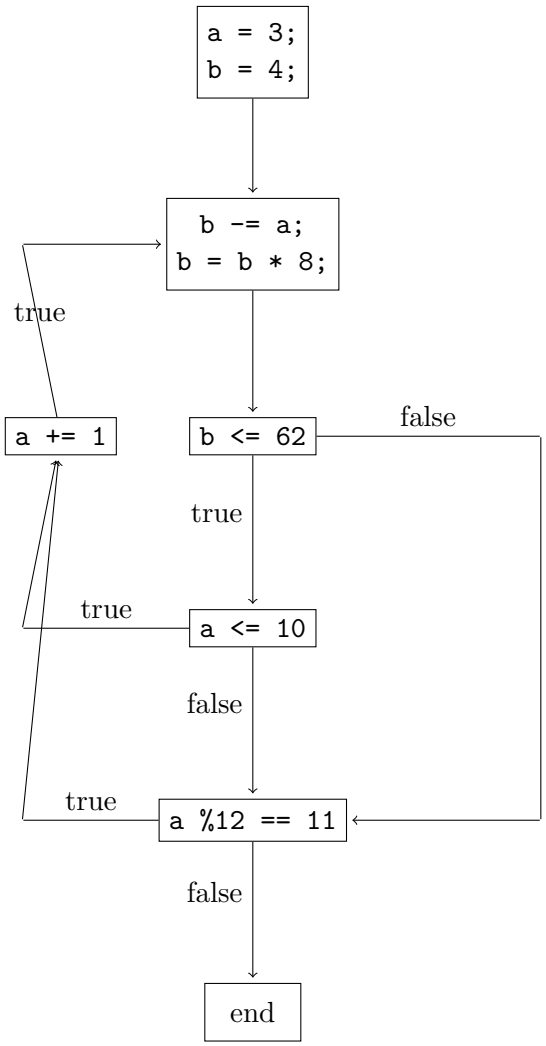
The loop-with-test loop runs the code in the first loop body, then checks the test condition. **If the condition evaluates to false, the loop ends;** otherwise, the second loop body evaluates and the loop repeats. Note that even if the test condition is always false, the first loop body will still run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

11. [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 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 a = 3;
int b = 4;
loop {
    b -= a;
    b = b * 8;
} while ((b <= 62 && a <= 10) || a % 12 != 11) {
    a += 1;
} repeat;
```

Solution: Rubric:

- 9 relations -¿ +9 points.
- +1 for correct code written.



12. [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 loop-with-test loop:

If `n` is of the form `loop { x1 } while (c) { x2 } repeat;` then

Solution:

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

Rubric:

- +1 for nop.
- +1 for each destruct.
- +1 for shortcircuit and +1 for correct arguments
- +2 for each next (1 for statement, 1 for argument)
- +1 for return.

V Code Generation for Procedures

Consider the following two functions in Decaf and its corresponding assembly code generated by a compliler.

```
int baz(int y) {
  return y*y;
}


int foo() {
  int x;
  x = 3;
  x += baz(x);
  return x;
}
```

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.

Rubric: For each of the next four questions

- +2 for correctly identifying correct or incorrecct.
- +4 for right justification (correct could be blank).
- +2 for good justification of the wrong answer

Which of the following possible generated code sequences for `baz` are correct in the sense that 1) they compute the correct return value for `baz` 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. If incorrect, please specify why.

13. [6 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, %r12     // copy the value of %rdi to %r12
4 mulq    %rdi, %r12     // mult the value of %rdi to %r12
5 movq    %r12, %rax     // copy the value of %rsp to %rbp
6 popq    %rbp           // pop the top value from the stack to %rbp
7 retq                   // return from the function
```

Correct Incorrect

Solution: Incorrect. The method is editing a callee-save register %r12, without saving/restoring its value.

14. [6 points]:

```
1 movq    %rdi, -8(%rsp) // copy the value of %rdi to the stack
2 mulq    -8(%rsp), %rdi // mult 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

Solution: Correct

Points were also awarded to a student identifying mulq shouldn't use a stack location as the source argument, deeming the above incorrect, though this wasn't a criteria specified above.

Which of the following possible generated code sequences for `foo` are correct in the sense that 1) they compute the correct return value for `foo` 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. If incorrect, please specify why.

15. [6 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    $3, %r11       // copy the value 3 to %r11
4 movq    %r11, %rdi     // move the value of %r11 to %rdi
5 call    baz            // call the baz method
6 addq    %rax, %r11     // add the value of %rax to %r11
7 movq    %r11, %rax     // move the value of %r11 to %rax
8 popq    %rbp           // pop the top value from the stack to %rbp
9 retq                   // return from the function
```

Correct Incorrect

Solution: Incorrect. The method is editing a caller-save register before calling `baz`, therefore having no guarantee that %r11 will preserve the same value after the call.

16. [6 points]:

```
1 movq    $3, -8(%rsp)   // copy 3 to the stack
2 movq    -8(%rsp), %rdi // copy the value on the stack to %rdi
3 call    baz
4 movq    -8(%rsp), %r10 // copy the value on the stack to %r10
5 addq    %rax, %r10     // add the value or %rax to %r10
6 movq    %r10, %rax     // copy the value of %r10 to %rax
7 retq                   // return from the function
```

Correct Incorrect

Solution: Incorrect. Edited part of the stack after the stack pointer before a method call, therefore having no guarantee that the value in the address space will be preserved.