Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.035 Fall 2018

Test I Solutions

UNKNOWN

Mean XX.X   Median XX.X   Std. dev XX.XX

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.

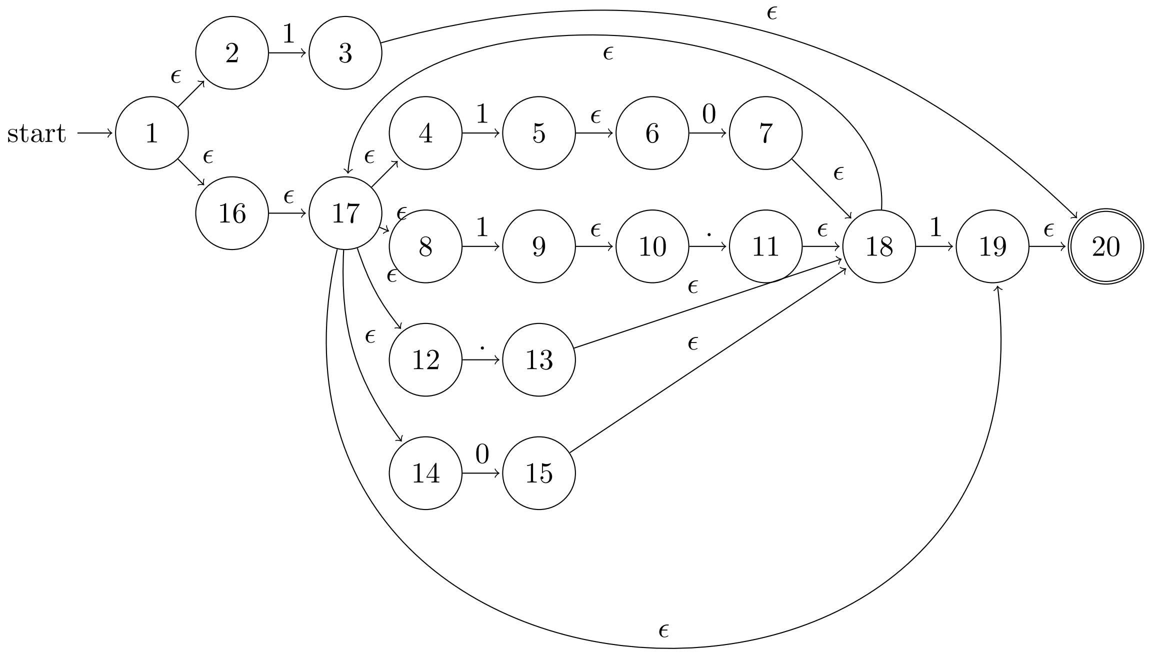
1. [5 points]: Write a regular expression that recognizes language L.

Solution: $(1.|10|.|0)^*1?$ **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 4. All DFA are NFA. Alternative solution:

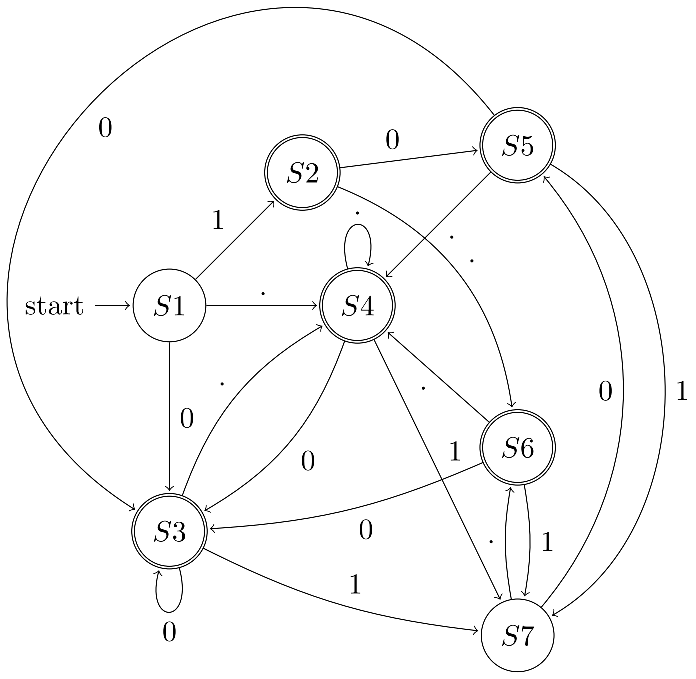


Rubric:

- -1 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: Letting $S1 = \{1, 2, 4, 8, 12, 14, 16, 17, 19\}$,
$S2 = \{3, 5, 6, 9, 10, 20\}$,
$S3 = \{4, 8, 12, 14, 15, 17, 18, 19\}$,
$S4 = \{4, 8, 12, 13, 14, 17, 18, 19, 20\}$,
$S5 = \{4, 7, 8, 12, 14, 17, 18, 19, 20\}$,
$S6 = \{4, 8, 11, 12, 14, 17, 18, 19, 20\}$,
$S7 = \{5, 6, 9, 10\}$,



Rubric:

– Same as problem 3

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.
Solution: Items:

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

Rubric:

– -0.6 for each item not on the list and for each extra item on the list.

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.

Solution:

$$\begin{tikzcd}
 & & S2 & & \\
\text{start} \arrow[r] & S1 \arrow[ur, "\text{X}"] \arrow[r, "\text{Y}"] \arrow[dr, "\text{num}"'] & S3 \arrow[r, "+"] & S5 \arrow[r, "\text{Y}"] \arrow[dl, "\text{num}"] & S6 \\
 & & S4 & &
\end{tikzcd}$$

$S1 = \{S \;\to\; \cdot\, X\, \$, X \;\to\; \cdot\, Y \,+\, Y, Y \;\to\; \cdot\, \text{num}\}$
$S2 = \{S \;\to\; X \;\cdot \$\}$
$S3 = \{X \;\to\; Y \;\cdot\; + Y\}$
$S4 = \{Y \;\to\; \text{num}\, \cdot\}$
$S5 = \{X \;\to\; Y \,+\, \cdot\, Y, Y \;\to\; \cdot\, \text{num}\}$
$S6 = \{X \;\to\; Y \,+\, Y\, \cdot\}$

Rubric:

- +5 for solution with num + num or Y + Y
- +10 for correct DFA

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

	Action			Goto	
State	+	num	\$	X	Y
S1	err	shift to S4	err	goto S2	goto S3
S2	err	err	accept		
S3	shift to S5	err	err		
S4	reduce(1)	reduce(1)	reduce(1)		
S5	err	shift to S4	err		shift to S6
S6	reduce(3)	reduce(3)	reduce(3)		

Rubric:

– +2 for each of correct states, reduces, accept/error, shifts, goto. Should correspond to DFA in previous problem.

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.

Stack
Y
+
Y

Rubric:

– Minus 1.6 for each symbol not on the stack in the proper location.

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
```

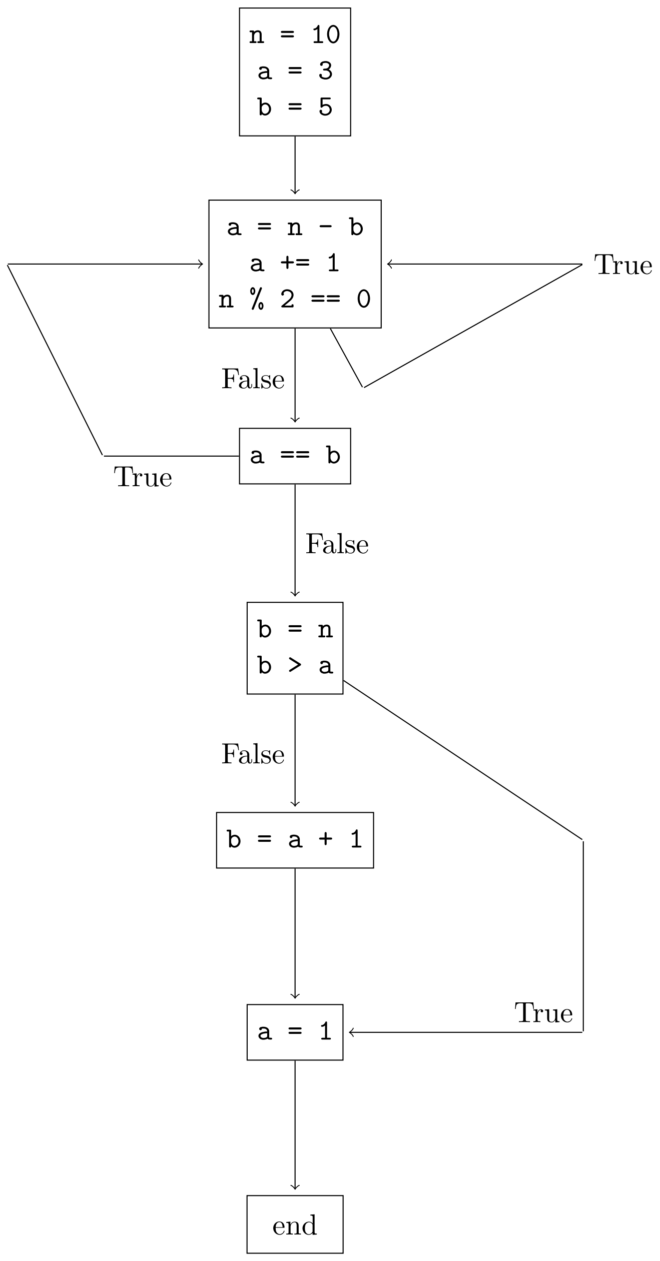
If the condition in the if statement is True then control flows to the line specified after the goto. Otherwise, control flows to the following statement. Statements are then evaluated sequentially. Note that the line specified by the goto may occur before or after the goto statement.

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;
```

Solution: Rubric:

- 8 lines of added code in the correct blocks → +5 points.
- 8 added edges connected properly → +5 points.
- -1 for each non-maximal block (i.e., not including condition in previous block)


n = 10
a = 3
b = 5
a = n - b
a += 1
n % 2 == 0
True
False
a == b
True
False
b = n
b > a
False
b = a + 1
True
a = 1
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. Solution:

```
bc2 = shortcircuit(c2, f, t)
bc1 = shortcircuit(c1, bc2, t)
return bc1
```

Rubric:

- +1.5 for each correct short-circuit, +0.5 for correct return, +0.5 for correct ordering of statements

B. Solution:

```
bc2 = shortcircuit(c2, f, t)
bc1 = shortcircuit(c1, f, bc2)
return bc1
```

Rubric:

- +1.5 for each correct short-circuit, +0.5 for correct return, +0.5 for correct ordering of statements

C. Solution:

```
bc2 = shortcircuit(c2, t, f)
bc3 = shortcircuit(c3, t, f)
bc1 = shortcircuit(c1, bc2, bc3)
return bc1
```

Rubric:

- +1 for each correct short-circuit, +0.5 for correct return, +0.5 for correct ordering of statements

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
```

Solution:

```
--------------------------
t1 = x
--------------------------
t2 = y
--------------------------
t3 = t1 + t2
--------------------------
a = t3
--------------------------
t4 = a
--------------------------
t5 = c
--------------------------
t6 = d
--------------------------
t7 = 3
--------------------------
t8 = t7 * t6
--------------------------
t9 = t5 + t8
--------------------------
t10 = t4 * t9
--------------------------
b = t10
--------------------------
```

Rubric:

- -1 for each incorrect line

8 points total.

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: [1]

```
_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?

Solution: No. `%edi` is not any part of `%rax`.

Rubric:

- +2 for correct answer
- +2 for good argument

4 points total.

[1] You can generate this on your `gcc` equipped machine with:
gcc -O0 -c -fno-asynchronous-unwind-tables -fno-dwarf2-cfi-asm -save-temps codeGen.c && less codeGen.s

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.

Solution:

Variable: `a`	**Stack** offset: `-12`
Variable: `x`	**Stack** offset: `-4`
Variable: `y`	**Stack** offset: `-8`
Variable: `z`	**Stack** offset: `-16`

Rubric: +2 for each correct response

8 points total.

OTHERWISE BLANK PAGE

APPENDIX I: BROWN CSC10330 X64 HANDOUT GUIDE

APPENDIX II: Notre Dame Introduction to X86 Assembly for Compiler Writers