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 4, 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: $1|(1.|10|.|0)^*$ **Rubric:**

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

– -1 for each category of string not accepted but should

2. [5 points]: Write a regular expression that recognizes language C, the language of all strings in L with an even number of zeros.

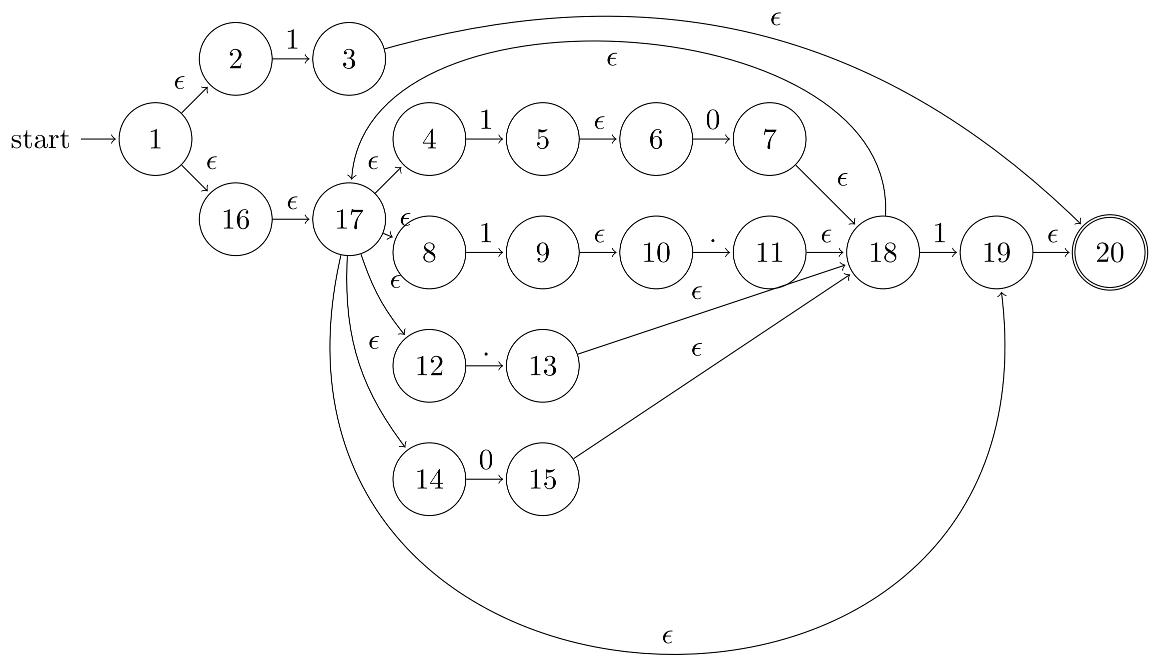
Solution: $1|(1?0(1.|.)^*1?0(1.|.)^*)^*$ **Rubric:**

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

– -1 for each category of string not accepted but should

3. [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:

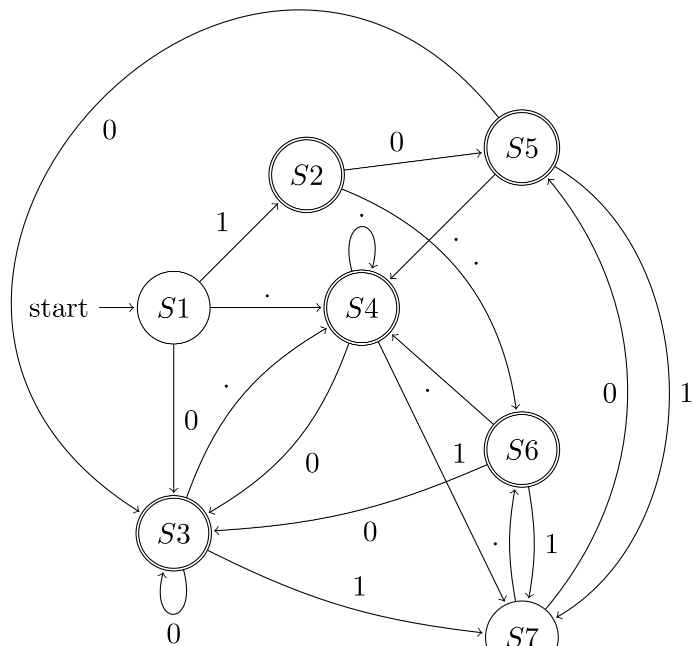


Rubric:

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

4. [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 simple grammar,

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

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

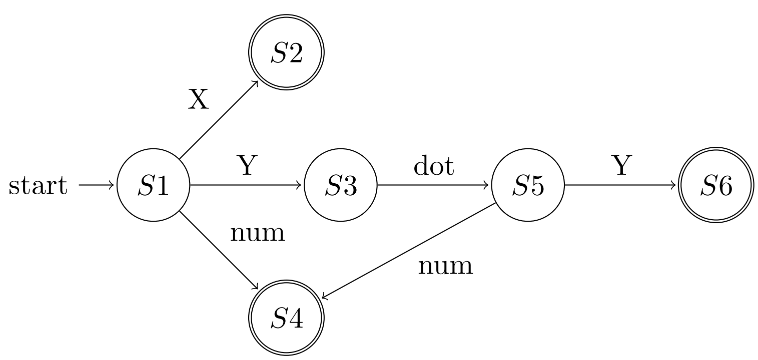
5. [5 points]: List the items generated by the grammar above.
Solution: Items:

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

Rubric:

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

6. [10 points]: Draw a DFA corresponding to the grammar above using the items in problem 5. Please specify which items belong to each state.
Solution:



$S1 = \{S \rightarrow \cdot X \ \$, X \rightarrow \cdot Y \text{ dot } Y, Y \rightarrow \cdot \text{ num}\}$
$S2 = \{S \rightarrow X \cdot \$\}$

$S3 = \{X \rightarrow Y \cdot \text{dot}\, Y\}$
$S4 = \{Y \rightarrow \text{num} \cdot\}$
$S5 = \{X \rightarrow Y\, \text{dot} \cdot Y, Y \rightarrow \cdot\, \text{num}\}$
$S6 = \{X \rightarrow Y\, \text{dot}\, Y \cdot\}$

Rubric:

- -1 for each missing state and edge and for each additional state and edge
- -0.5 for each item missing from a state and for each item in the wrong state

7. [10 points]: Complete the entries in the following parse table for the DFA in problem 6.

	Action			Goto	
State	dot	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:

- Minus 0.3 for each entry that does not correspond to the DFA drawn in problem 6.

8. [5 points]: The string 5.6$ is parsed through the parse table in problem 7. Draw the state stack and symbol stack after the second goto operation.

State	Symbol
S6	
S5	Y
S3	.
S1	Y

Rubric:

- Minus 0.7 for each symbol and state not on the stacks in the proper location according to the parse table in problem 7.

III Control Flow and Short-Circuiting

Consider a programming language that includes a control flow construct called "switch". A switch statement is written as follows:

```
switch (exp) {
  case c1:
    // first case's statements
    // and/or break or fallthrough
  case c2:
    // second case's statements
    // and/or break or fallthrough
  default:
    // otherwise these statments
    // and/or break
}
```

The control expression of the switch statement is evaluated and then compared with the expressions specified in each case. Once a match is found the program executes the statements listed within the scope of that case. The scope of each case can not be empty, and every possible value of the control expression must match the value of at least one case.

The `break` statement ceases execution of the current case. The `fallthrough` statement ceases execution of the current case and begins execution of the following case's statements– program execution continues to the next case even if the expression of the case label does not match the value of the switch statement's control expression.

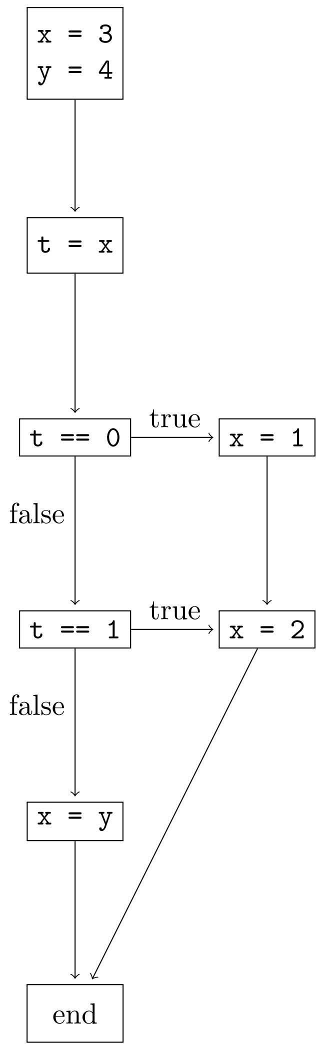
9. [10 points]: The semantics of the programming language say that a compiled program should only evaluate expressions until the first match with the control expression's value is found.

Complete the control flow graph on the next page that illustrates the control flow for evaluating the following statements, including evaluate-once logic for the control expression, assuming the compiler is not performing any optimizations:

```
var x = 3
var y = 4
switch (x) {
  case 0:
    x = 1
    fallthrough
  case 1:
    x = 2
    break
  default:
    x = y
}
```

Solution: Rubric:

- 8 relations (allowing either $[x = 3, y = 4]$ or $[t = x] \rightarrow [t == 0]$) $\rightarrow$ +8 points.
- +1 for $[t = x]$, indicating unoptimized code.
- +1 for correct code written (besides $[t = x]$), allowing x to be substituted for t.



10. [12 points]: In the lecture, we discussed the implementation of procedures called `shortcircuit`, `next` 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 `next(n1) = n2` allows you to specify `n2` as the subsequent control-flow node to be executed after `n1`.

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.

We will also introduce new procedures `eval` and `equal_exp`:

The procedure `eval(exp)` generates the control-flow representation for evaluation of the expression `exp`. The procedure returns the begin and end nodes of the graph, in addition to a value which can be used once the end node has been reached.

The procedure `equal_exp(lhs, rhs)` returns a conditional equality expression from `lhs` and `rhs`, evaluating to true if they are equal. Where they are expressions, `lhs` and `rhs` are evaluated at the same time as the returned expression.

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 switch statement:

If `n` is of the form

```
switch(exp1) {
  case exp2:
    x1()
    break
  case exp3:
    x2()
    break
  default:
    x3()
    break
}
```

then the pseudocode of `destruct(n)` is:

Solution:

```
e = new nop
(val, bv, ev) = eval(exp)
case1C = equal_exp(val, exp2)
case2C = equal_exp(val, exp3)
(b1, e1) = destruct(x1)
(b2, e2) = destruct(x2)
(b3, e3) = destruct(x3)
sc2 = shortcircuit(case2C, b2, b3)
sc1 = shortcircuit(case1C, b1, sc2)
next(e1) = e
next(e2) = e
next(e3) = e
next(ev) = sc1
return (bv, e)
```

Rubric:

- +1 for `new nop`.
- +2 for exactly one `eval(exp)`
- +1 for each `equal_exp`.
- +1 for each `destruct`.
- +1 for each of two `shortcircuit` and +1 each for correct arguments (they only work in rubric order)
- +3 for `next(e1,e2 and e3) = e`
- +2 for `next(ev) = sc1`
- +1 for `return`.

12 points total.

IV Code Generation for Procedures

To test a compiler you're building, you've written a test program in C:

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

int main(int argV, char ** argC) {
  long a = 1;
  return foo(a);
}
```

For which your compiler outputs the following:

```
  .section        __TEXT,__text,regular,pure_instructions
  .macosx_version_min 10, 13
  .globl  _foo
  .p2align        4, 0x90
_foo:                                  ## @foo
## BB#0:
  pushq   %rbp
  movq    %rsp, %rbp
  movl    $1024, %eax            ## imm = 0x400
                                 ## kill: %RAX<def> %EAX<kill>
  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    %ecx, %eax
  popq    %rbp
  retq

  .globl  _main
  .p2align        4, 0x90
_main:                                 ## @main
## BB#0:
  pushq   %rbp
  movq    %rsp, %rbp
  subq    $32, %rsp
  movl    $0, -4(%rbp)
  movl    %edi, -8(%rbp)
  movq    %rsi, -16(%rbp)
  movq    $1, -24(%rbp)
  movq    -24(%rbp), %rdi
  callq   _foo
  addq    $32, %rsp
  popq    %rbp
  retq

.subsections_via_symbols
```

Note: Assembly `long`s (`l` suffix) are 32 bits- double words. On this platform C's `long` is 64 bits while `int` is 32 bits (it varies by system!). [1]

[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

11. [4 points]: Which registers does the `cqto` operation and the `idivq` operation affect in the `foo` function?

Hint: Utilize your open book materials.

Solution:
`cqto` →
Source: %rax
Destination: %rdx:%rax

`idivq` →
Source: %rdx:%rax
Destination: %rax

12. [2 points]: Does the `foo` function obey standard calling convention by placing the return value in `%eax`? Why or why not?

Solution: Yes. `%eax` is bytes $5 - 8$ of `%rax`.

13. [9 points]: Your compiler over-generates code. Re-write the `foo` function as optimally as you can while obeying standard calling convention.

`_foo:`	`_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 %ecx, %eax`	____________________________
`popq %rbp`	____________________________
`retq`	____________________________

Solution: Most basic, removing these lines, 3 points.

```
  movq    %rdi, -8(%rbp)
  movq    -8(%rbp), %rdi
```

```
  movq    %rax, -16(%rbp)
  movq    -16(%rbp), %rax
  movl    %eax, %ecx
  movl    %ecx, %eax
```

We claim this is most optimal:

```
_foo:
  xorq    %rdx, %rdx      #idivq needs rdx clear. Also, cqto or movq $0, %rdx work
  movq    $1024, %rax     #also works: movl   $1024, %eax
  idivq   %rdi            #solution is in %rax
  retq
```

Three points, if your optimization appears to obey calling convention and work.

Two points, for removing `idivq` stack usage, and associated stack save.

One final point if you were able to omit `pushq` and `popq`. No point awarded if you removed `movq    %rsp,   %rbp` without `pushq` or if you don't obey calling convention.

E.g., this solution didn't get the point.

```
_foo:
  pushq   %rbp
  movq    %rsp,   %rbp
  # rest of function
  popq    %rbp
```

9 points total.

14. [5 points]: You've completed the optimization stage of your compiler including a constant propagation system, and the compiler outputs the following assembly for the `main` function.

```
_main:
  pushq   %rbp
  movq    %rsp, %rbp
  pushq   %edi
  movq    $1, $rdi
  callq   _foo
  popq    %edx
  addq    %edx, %rax
  popq    %rbp
  retq
```

Unfortunately you've already edited the main function. What is the body of the `main` function C code for the optimized method? (That is, decompile most directly from the above assembly!)

Please fill in the body for `main`.

```
int main(int argV, char ** argC){



}
```

Solution:

```
int main(int argV, char ** argC){
  return foo(1) + argV;
}
```

APPENDIX I: BROWN CSC10330 X64 HANDOUT GUIDE