Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.s081 Spring 2018

Test I Solutions

I Regular Expressions and Finite-State Automata

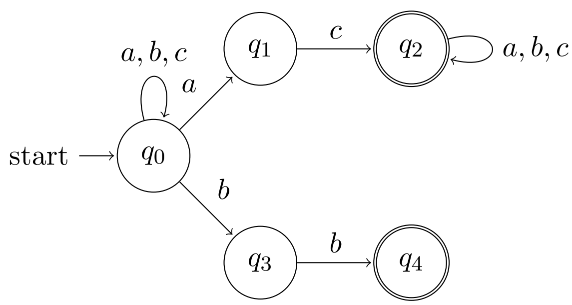
For Questions 1, 2, and 3, let the alphabet $\Sigma = \{a, b, c\}$. Let language L be the language of all strings over Σ that contain the substring "ac" or end in the substring "bb".

1. [10 points]: Write a regular expression that recognizes language L.

Solution: $(a|b|c)^*((ac(a|b|c)^*)|bb)$

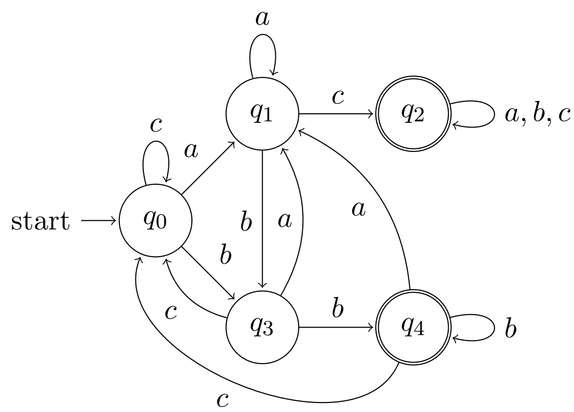
2. [9 points]: Draw a state diagram of a nondeterministic finite-state automaton (NFA) that recognizes language L. Remember to indicate starting and accepting states.

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



3. [9 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:

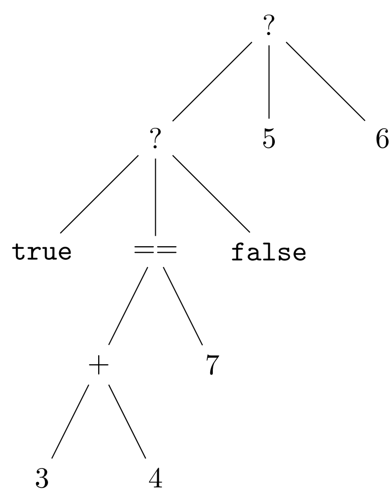

start
q0
q1
q2
q3
q4
a
b
c
a, b, c

II Hacking the Grammar

For Questions 4 through 6, consider the following grammar for a language with expressions:

$E \to E \; ? \; E : E$
$E \to E == E$
$E \to E + E$
$E \to n$
$E \to b$
Where n is an integer token and b is a boolean token.

4. [10 points]: Hack the grammar to give + higher precedence than ==, to give == higher precedence than ? :, and to make all operators left associative. Specifically, the grammar should produce a parse tree for the string "`true ? 3 + 4 == 7 : false ? 5 : 6`" that reflects the evaluation order `((true ? ((3 + 4) == 7) : false) ? 5 : 6)`. This evaluation order is also reflected in the following abstract syntax tree:



Solution:

$$
\begin{aligned}
E \to & E \; ? \; E \; : \; F \\
& | F \\
F \to & F == G \\
& | G \\
G \to & G + c \\
& | G + b \\
& | c \\
& | b
\end{aligned}
$$

5. [10 points]: Remove left recursion from your answer to Question 5 to make the language parseable by a recursive descent parser with one token of lookahead. Do not worry about maintaining associativity.

Solution:

$$
\begin{aligned}
E \rightarrow & F \ ? \ E \ : \ E \\
& | F \\
F \rightarrow & G == F \\
& | G \\
G \rightarrow & c + G \\
& | b + G \\
& | c \\
& | b
\end{aligned}
$$

III Semantics: Systems Languages

We are designing a language called QuizOneScript, which will be a simple extension of IMP. We will start with a version of IMP that only includes the standard constructs (expression, assignments, `while`, and `if` statements) as well as heaps and types (but not scopes, functions, or closures).

Recall the following inference rule in class for an assignment of the form `x = e`:

$$\frac{(\sigma, h, e) \rightarrow \textit{Integer(n)} \quad \sigma[x : a] = \sigma' \quad h[a : n] = h' \quad \neg(a \in \textit{dom}(h))}{(\sigma, h, x = e) \rightarrow (\sigma', h')}$$

and the rule for variable lookup:

$$\frac{\sigma(x) = a \quad h[a] = \textit{Integer(n)}}{(\sigma, h, x) \rightarrow \textit{Integer(n)}}$$

In QuizOneScript, we introduce two new expressions, address expressions with the `&` symbol and dereference expressions with the `*` symbol. Given a variable `x`, `&x` evaluates to the address pointing to the value of `x` and `*x` evaluates to the value pointed to by `x`. To help you represent the values of these expressions, we introduce *Ptr(a)* which is a value type representing the address a.

6. [7 points]: Write the inference rule for an address expression evaluation, (σ, `h, &x`)

Solution:

$$\frac{\sigma(x) = a}{(\sigma, h, \texttt{\&x}) \rightarrow \textit{Ptr(a)}}$$

7. [8 points]: Write the inference rule for a dereference expression evaluation, (σ, `h, *x`)

Solution:

$$\frac{(\sigma, h, \texttt{x}) \rightarrow \textit{Ptr(a)} \quad h[a] = \textit{Integer(n)}}{(\sigma, h, \texttt{*x}) \rightarrow \textit{Integer(n)}}$$

After designing these language features, we come across the following code snippet:

```
x = 1;
y = &x;
x = 2;
print(*y);
```

Given the inference rule for assignment and the semantics described by our expressions, the above snippet would print 1 which may seem unintuitive.

8. [10 points]: Reformulate the original assignment rule above so that this code snippet would print 2 instead of 1. (Hint: a straightforward approach will provide two inference rules)

Solution:

$$\frac{(\sigma, h, e) \rightarrow \mathit{Integer(n)} \quad \sigma(x) = a \quad h[a : n] = h'}{(\sigma, h, x = e) \rightarrow (\sigma, h')}$$

$$\frac{(\sigma, h, e) \rightarrow \mathit{Integer(n)} \quad \sigma[x : a] = \sigma' \quad h[a : n] = h' \quad \neg(x \in dom(\sigma)) \quad \neg(a \in dom(h))}{(\sigma, h, x = e) \rightarrow (\sigma', h')}$$

We now want to add functions (not closures) to QuizOneScript. Invocation of a regular function introduces a new frame, but the function cannot refer to variables in its parent scope. Here is a semantics for a regular function:

$$\frac{}{(\sigma, h, \texttt{fun}\ x\ s) \rightarrow \mathit{Function}(x, s)}$$

$$\frac{\begin{array}{ccc}(\sigma_1, h_1, e_1) \rightarrow \mathit{Function}(x_2, s) & (\sigma_1, h_1, x_1) \rightarrow v & a \notin \mathit{dom}(h_1) \\ h_2 = h_1[a : v] & \sigma_2 = \{x_2 : a\} & (\sigma_2, h_2, s) \rightarrow h_3\end{array}}{(\sigma_1, h_1, e_1(x_1)) \rightarrow (\sigma_1, h_3)}$$

We use the standard stack model of frames, which we do not allocate in the heap. Note that creating a function *does not* capture the address of the current frame (as is done for closures).

Referring to the value of variables outside of the immediate scope is, however, a very handy feature. To support this, we will add *call-by-reference* functions that allow taking a reference to an argument. Specifically, to refer to a value outside of its immediate scope, the function must be passed references to those variables. Here is a syntax for a call-by-reference function:

```
x = 1;
f = fun&(y) {
  y = 2;
};
f(x);
print(x);
```

In this example, the call to `f` receives a reference to `x` in its local variable `y` and assigns 2 to the variable pointed to by that reference. This program therefore prints the value 2.

9. [8 points]: Provide a new version the rules from the previous page to support call-by-reference. To help you get started, here is a *complete* rule for creating a call-by-reference function:

$$\frac{}{(\sigma, h, \texttt{fun\&}\ x\ s) \to \mathit{RefFunction}(x, s)}$$

$$\frac{(\sigma, h_1, e_1) \to \mathit{RefFunction}(x_2, s) \qquad \sigma_1[x_1] = a \qquad \sigma_2 = \{x_2 : a\} \qquad (\sigma_2, h_1, s) \to h_2}{(\sigma_1, h_1, e_1(x_1)) \to (\sigma_1, h_2)}$$

IV Semantics: Switch

We are next going to add switch statements to QuizOneScript (see beginning of III and ignore features add in III). For this problem, however, assume that QuizOneScript supports strings.

Recall the following inference rules in class for an if statement of the form `if (e) {S} else {S}`:

$$\frac{(\sigma, h, e) \to Bool(True) \quad (\sigma, h, s_1) \to (\sigma', h')}{(\sigma, h, \texttt{if}\ (e)\ s_1\ s_2) \to (\sigma', h')} \qquad \frac{(\sigma, h, e) \to Bool(False) \quad (\sigma, h, s_2) \to (\sigma', h')}{(\sigma, h, \texttt{if}\ (e)\ s_1\ s_2) \to (\sigma', h')}$$

Switch statements are given by the grammar:

$S \to \texttt{switch}\ (E)\ \{C\}$

$C \to (\texttt{case}\ V\ \{S\})^*$

$V \to n$

These statements evaluate the initial expression and then sequentially compares to each value following the case keyword, only executing the block on the first match. Note that this implies that if the expression does not match any of the values, this statement is a no-op.

For example, the code snippet below prints `"One"` when `x` is 1, `"Two"` when `x` is 2, `None` for any other values of `x` (no assignment happens).

```
str = "None";
switch(x)
{
   case 1 { str = "One"; }
   case 2 { str = "Two"; }
};
print(str);
```

10. [4 points]: Rewrite the code snippet on the previous page using QuizOneScript language constructs but without using switch statements. As noted above, you can assume QuizOneScript now supports strings.

Solution:

```
str = "None";
if(x == 1)
{
   str = "One";
} else {
    if (x == 2) {
        str = "Two";
    }
}
print(str);
```

11. [15 points]: Using the templates below, write the inference rule(s) for a switch statement evaluation (you could write below the templates themselves and use the entire page).

$$\frac{?}{(\sigma, h, \texttt{switch (}e\texttt{)}\ (v, s) :: C) \rightarrow \qquad ?}$$

$$\frac{?}{(\sigma, h, \texttt{switch (}e\texttt{)}\ \epsilon) \rightarrow \qquad ?}$$

Solution:

$$\frac{(\sigma, h, e) \rightarrow \mathit{Integer}(n) \quad (\sigma, h, v == \mathit{Integer}(n)) \rightarrow \mathit{Bool(true)} \quad (\sigma, h, s) \rightarrow (\sigma', h')}{(\sigma, h, \texttt{switch (}e\texttt{)}\ (v, s) :: C) \rightarrow (\sigma', h')}$$

$$\frac{(\sigma, h, e) \rightarrow \mathit{Integer}(n) \quad (\sigma, h, v == \mathit{Integer}(n)) \rightarrow \mathit{Bool(false)} \quad (\sigma, h, \texttt{switch (}e\texttt{)}\ C) \rightarrow (\sigma', h')}{(\sigma, h, \texttt{switch (}e\texttt{)}\ (v, s) :: C) \rightarrow (\sigma', h')}$$

$$\frac{}{(\sigma, h, \texttt{switch (}e\texttt{)}\ \texttt{[]}) \rightarrow (\sigma, h)}$$

V Semantics: Closures and Scope

IMP with closures has *static scoping* (also known as *lexical scoping*). An alternative scoping technique is *dynamic scoping*. For example, consider the following code:

```
1:  var x = 1;
2:  f = fun() {
3:    print(x);
4:  };
5:  g = fun() {
6:    var x = 2;
7:    f();
8:  };
9:  g();
```

If we execute this program with static scoping, then the `print` statement within `f` (Line 3) prints the value 1. Printing this value corresponds to the fact that Line 1 was the closest definition of `x` on the stack *at the time the program created the closure.*

In dynamic scoping, the `print` statement prints the value 2. Printing this value corresponds to the fact that Line 6 was the closest definition of `x` on the stack *at the time* `x` *was read.*

Here are the inference rules for function creation and function calls from IMP (with static scoping).

$$\frac{}{(a, h, \texttt{fun}\ x\ s) \to \mathit{Function}(a, x, s)}$$

$$\frac{\begin{array}{c}(a, h_1, x_1) \to \mathit{Function}(a_c, x_2, s) \qquad (a, h_1, e_2) \to v \\ a_2 \notin \mathit{dom}(h_3) \qquad a_3 \notin (\mathit{dom}(h_3) \cup \{a_2\}) \qquad \sigma = \{\rho : a_c, x_2 : a_3\} \\ h_2 = h_1[a_2 : \sigma][a_3 : v] \qquad (a_2, h_2, s) \to h_3\end{array}}{(a, h_1, x_1(e_2)) \to h_3}$$

For each of the questions below, you should not have to consider changes to *lookup* (as was defined in lecture). You also do not need to introduce a new value type constructor (you can reuse the *Function* constructor).

12. [2 points]: Provide a new inference rule for creating closures that support dynamic scoping (instead of static scoping).

$$\frac{}{(a, h, \texttt{fun}\ x\ s) \rightarrow \mathit{Function}(x, s)}$$

13. [3 points]: Provide a new inference rule for calling closures that uses dynamic scoping (instead of static scoping).

$$\frac{\begin{array}{c}(a, h_1, x_1 \rightarrow \mathit{Function}(x_2, s) \quad (a, h_1, e_2) \rightarrow v \\ a_2 \notin \mathit{dom}(h_3) \quad a_3 \notin (\mathit{dom}(h_3) \cup \{a_2\}) \quad \sigma = \{\rho : a, x_2 : a_3\} \\ h_2 = h_1[a_2 : \sigma][a_3 : v] \quad (a_2, h_2, s) \rightarrow h_3\end{array}}{(a, h_1, x_1(e_2)) \rightarrow h_3}$$