Quiz 1
6.035 Spring 2023 March 17, 2023

Solutions

1. **True/False** [12 pts] (parts a–f)

   (a) There exist adversarial NFAs (nondeterministic finite automata) which cannot be converted to DFAs (deterministic finite automata).

   False

   (b) The values in a shift-reduce parser's symbol stack evolve from lower-level elements (closer to leaf nodes) of the parse tree to higher-level elements (closer to the root node) of the parse tree over the course of parsing.

   True

   (c) In the Decaf language, the symbol tables in the IR must be written to and read from during program run time, because the compiler cannot know about variable descriptors at compile time.

   False

   (d) According to Professor Rinard's recommendation, most semantic checks should be performed while building the parse tree.

   False

   (e) In the Decaf language, an expression computing an integer can result in a CFG (control flow graph) with multiple distinct nodes and edges.

   True

   (f) The register that holds the return value of a function is a callee-save register.

   False

2. **Lexing** [10 pts] (parts a–c)

   Consider the following regular expression:

   ```
   a*(b|(c*))d
   ```

   (a) [2 pts] **Yes/No:**
   Are the following strings in the language defined by the above regular expression?

   i. _____ `abcd`

   No

ii. _____ `d`

Yes

(b) [4 pts] Draw an NFA (nondeterministic finite automaton) that accepts strings in the language defined by the above regular expression. Make sure to annotate which states are start states and which states are accept states.

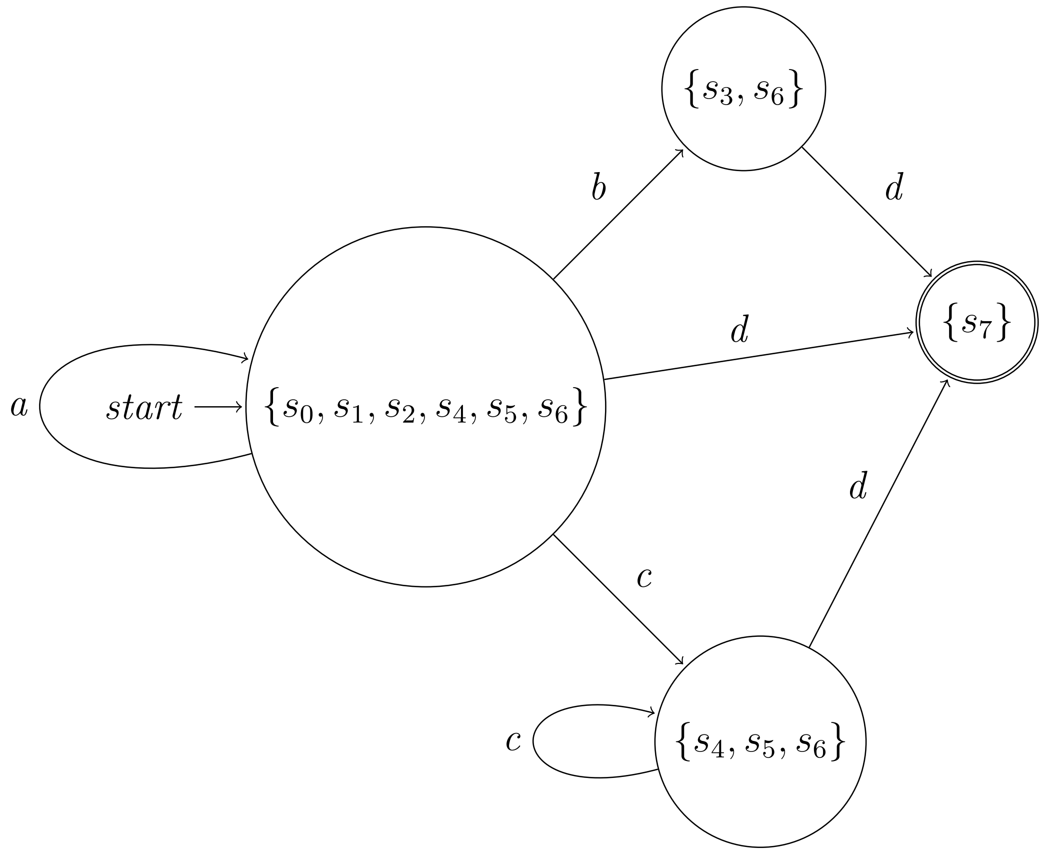
Answer:

$$\begin{tikzcd}
& & & s_2 \arrow[r, "b"] & s_3 \arrow[dr, "\epsilon"] & & \\
start \arrow[r] & s_0 \arrow[r, "a"] \arrow[r, bend left, "\epsilon"] & s_1 \arrow[l, bend left, "\epsilon"] \arrow[ur, "\epsilon"] \arrow[dr, "\epsilon"] & & & s_6 \arrow[r, "d"] & s_7 \\
& & & s_4 \arrow[r, "c"] \arrow[r, bend left, "\epsilon"] & s_5 \arrow[l, bend left, "\epsilon"] \arrow[ur, "\epsilon"] & &
\end{tikzcd}$$

-1 for each extra/missing/unlabeled node or edge

(c) [4 pts] Draw a DFA (deterministic finite automaton) that accepts strings in the language defined by the above regular expression. Make sure to annotate which state is the start state and which states are accept states.

Answer:



-3 if not a DFA

-1 for each extra/missing/unlabeled node or edge

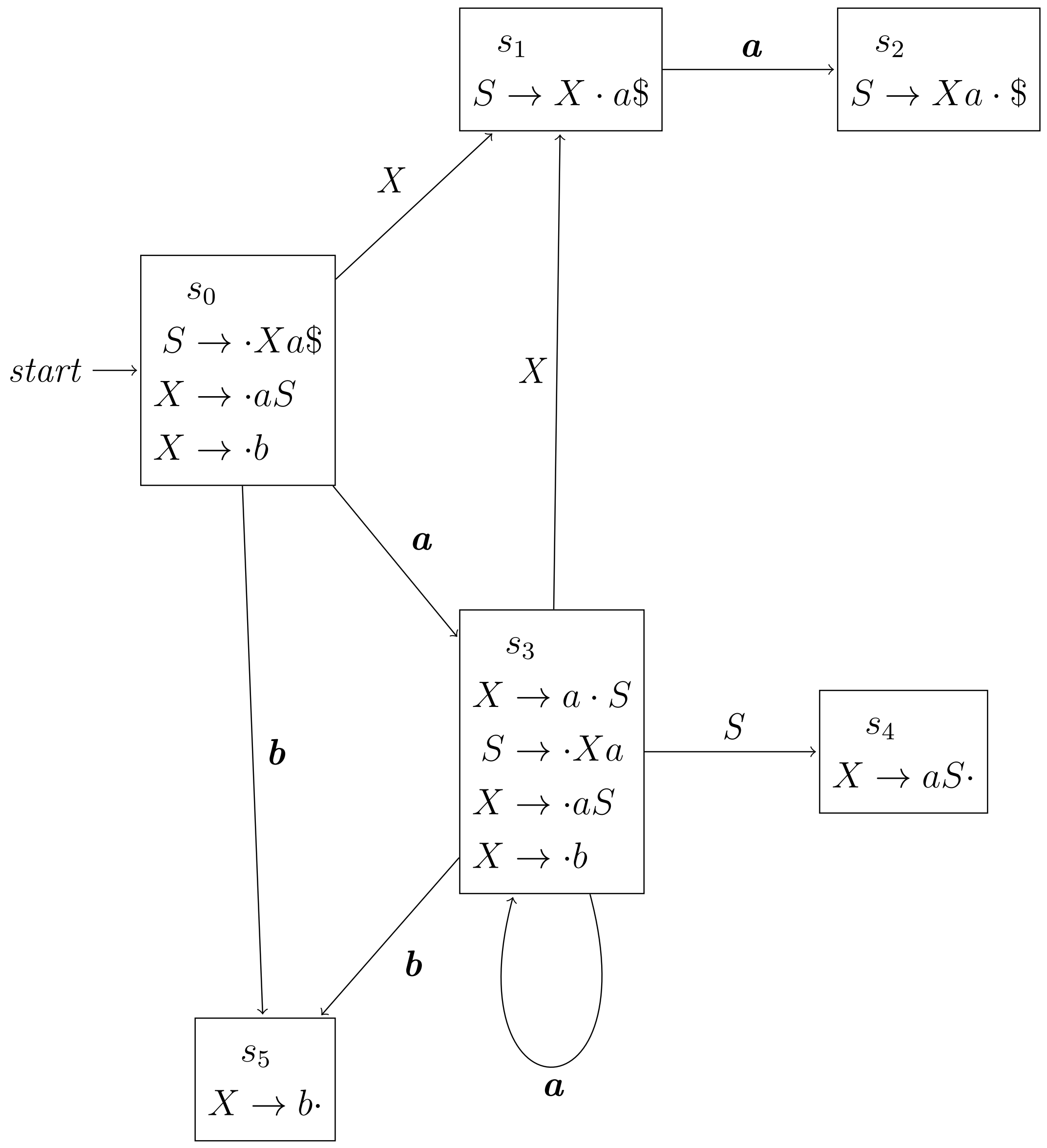
3. **Parsing** [10 pts] (parts a–b)

   Consider the following grammar, where the bolded symbols are terminals. The start symbol is S:

$$S \to X\ \mathbf{a}\ \$$$
$$X \to \mathbf{a}\ S$$
$$X \to \mathbf{b}$$

   (a) [4 pts] For the above grammar, draw the control DFA (deterministic finite automaton) for an LR(0) shift-reduce parser like the ones built in class. Make sure to annotate which state is the start state.

Answer:



-0.5 if did not annotate start state
-2 if did not expand $X \to a \cdot S$
-2 if did not expand $S \to \cdot Xa$
-1 per missing state/edge

(b) [6 pts] Consider the following grammar, which is a slightly tweaked version of the grammar on the previous page (still with start symbol S):

$$S \to \mathbf{a}$$
$$S \to \mathbf{b}\ S$$
$$S \to S\ \mathbf{c}$$

Consider the following implementations of a top-down parser for this language. As in class, the current input symbol is stored in the global variable `token`, and the function `NextToken()` advances `token` to the next input symbol. A procedure returns `true` if it successfully parsed, and `false` otherwise.

Which of the following implementations would correctly parse this language? Circle **Correct** or **Incorrect** for each implementation. There may be multiple or no correct implementations.

```
bool parse_S() {
 if (token == 'a') {
  // S -> a
  token = NextToken();
  return true;
 } else if (token == 'b') {
  // S -> b S
  token = NextToken();
  return parse_S();
 } else {
  // S -> S c
  if (parse_S()) {
   oldToken = token;
   token = NextToken();
   return oldToken == 'c';
  } else {
   return false;
  }
 }
}
```

Correct / Incorrect

```
bool parse_S() {
 if (token == 'a') {
  // S -> a
  token = NextToken();
  return true;
 } else if (token == 'b') {
  // S -> b S c
  token = NextToken();
  if (parse_S()) {
   oldToken = token;
   token = NextToken();
   return oldToken == 'c';
  } else {
   return false;
  }
 } else {
  return false;
 }
}
```

Correct / Incorrect

```
bool parse_S() {
 if (token == 'a') {
  // S -> a Sprime
  token = NextToken();
  return parse_Sprime();
 } else if (token == 'b') {
  // S -> b S
  token = NextToken();
  return parse_S();
 } else {
  return false;
 }
}

bool parse_Sprime() {
 if (token == 'c') {
  // Sprime -> c Sprime
  token = NextToken();
  return parse_Sprime();
 } else {
  // Sprime -> epsilon
  return true;
 }
}
```

Correct / Incorrect

Answer:

Incorrect (left recursion), Incorrect (different language), Correct

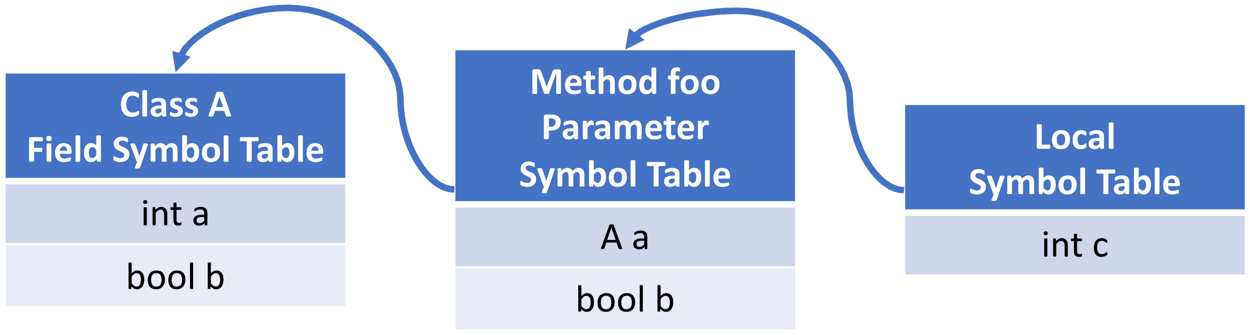
4. **IR and Semantic Checking** [12 pts] (parts a–h)

Consider the following class:

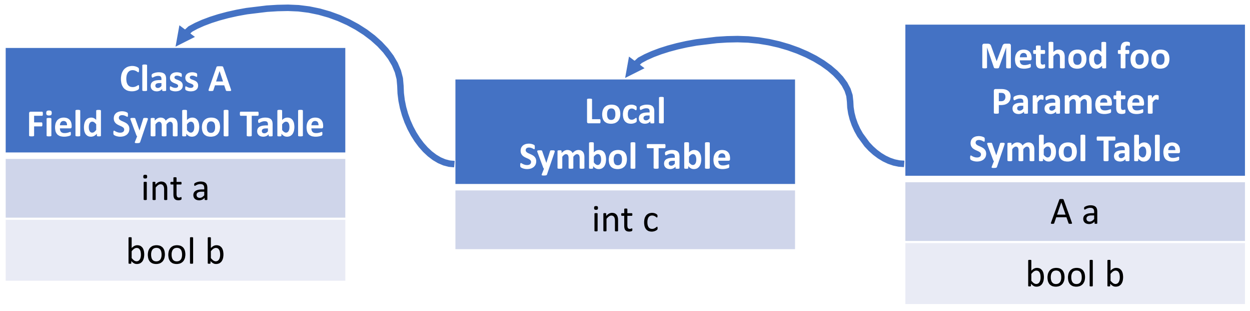
```
class A {
  int a;
  bool b;

  int foo(A a, bool b) {
    int c;
```

```
      ...
    }
}
```

When typechecking and generating code using the methodology described in class, the compiler starts at the local symbol table, and walks up the symbol table hierarchy until it finds the definition of the symbol. The correct symbol table hierarchy inside method `foo` is shown below:



Unfortunately, your partner botched the implementation! They accidentally flipped the order when generating the parameter and local symbol tables (as in below). However, lookup in your botched compiler still starts at the local symbol table.



(a) [3 pts] Give an example of an implementation of method `foo` that would type check and compile with a correct compiler but would generate a type error in your implementation, or explain why this is not possible:

Answer:

Any program that treats `a` as an `A`:

```
int foo(A a, bool b) {
  return a.a;
}
```

-1 if there is a type error in a correct compiler that is incidental to the solution
-1 if lookup starts at the parameter symbol table
-3 if says "not possible"
-2 if no implementation given

(b) [3 pts] Give an example of an implementation of method `foo` that would type check and compile with your implementation but could generate incorrect output when executed, or explain why this is not possible:

Answer:

Any program that doesn't access `a` but is conditional on the value of `b`:

```
int foo(A a, bool b) {
  return b ? 1 : 2;
}
```

-1 if there is a type error that is incidental to the solution
-1 if lookup starts at the parameter symbol table
-3 if says "not possible"
-2 if no implementation given

Consider the following class hierarchy:

```
class X {                 class Y extends X {       class Z extends Y {
  int xa;                   int ya;                   int za;
  int xb;                   int yb;                   int zb;
}                                                   }
                            Y foo(Y y) { ... }
                          }
```

Consider the following code snippets, from a language using the typing rules discussed in class. Assume that the following variables are in scope:

- A variable `x` of type `X`
- A variable `y` of type `Y`
- A variable `z` of type `Z`

Circle **Typechecks** or **Type Error** for each snippet. There may be multiple or no typechecking snippets.

(c) [1 pt] x = y.foo(x); **Typechecks / Type Error**

(d) [1 pt] y = y.foo(y); **Typechecks / Type Error**

(e) [1 pt] z = y.foo(z); **Typechecks / Type Error**

(f) [1 pt] x = y.foo(z); **Typechecks / Type Error**

(g) [1 pt] z = y.foo(x); **Typechecks / Type Error**

(h) [1 pt] z = z.foo(z); **Typechecks / Type Error**

Answer:

i. Type Error
ii. Typechecks
iii. Type Error
iv. Typechecks
v. Type Error
vi. Type Error

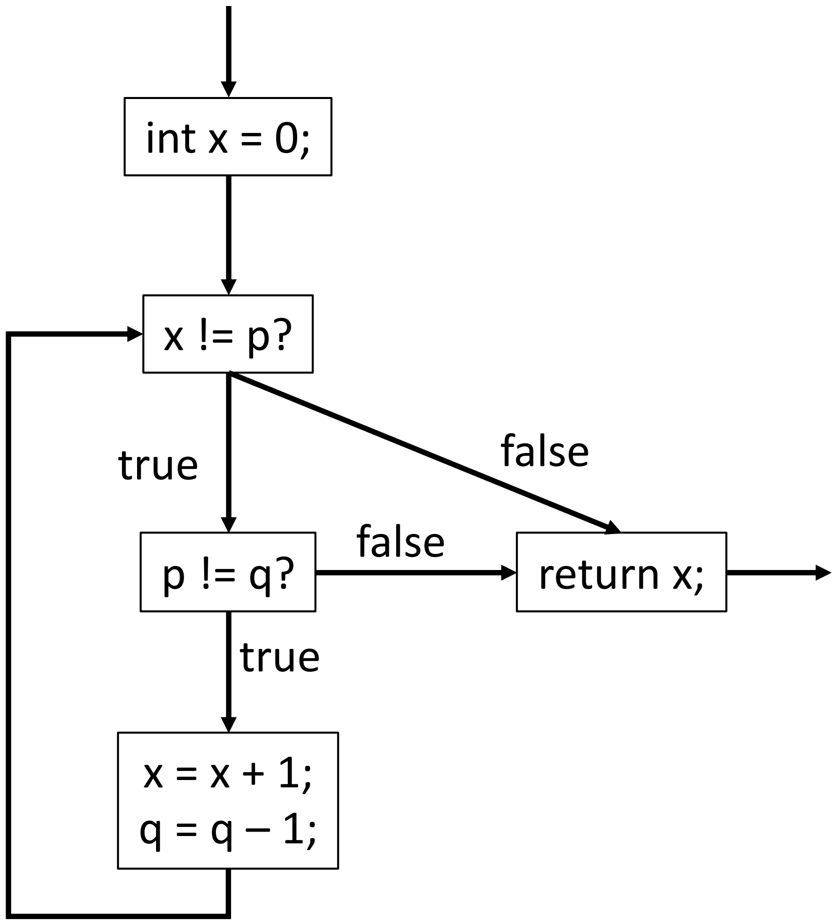
5. **Codegen** [14 pts] (parts a–f)

(a) [4 pts] Consider the following Decaf code:

```
int foo(int p, int q) {
  int x = 0;

  while (x != p && p != q) {
    x = x + 1;
    q = q - 1;
  }
  return x;
}
```

Draw a CFG (control flow graph) of basic blocks for the above code (as written; do not perform any optimizations). Be sure to draw identify the entry and exit node(s) for the CFG. Draw condition checks by including the expression followed by a question mark, with `true` and `false` edges.

Answer:



-2 if no short circuiting

-1 if did not mark entry/exit nodes

-0.5 per unlabeled edge (if more than one)

Uh oh! When building your group's compiler for this language, your teammate forgot to implement short circuiting: all expressions are fully evaluated.

(b) [2 pts] Give an example of standard Decaf code for which your group's compiler will generate code that produces incorrect outputs, or explain why this is not possible.

Answer:

Any program where a short-circuit is not taken and a side-effect occurs:

```
x = -1;
if (x > 0 && a[x]) {
  ...
}
```

-1 if gives short-circuiting example with no side effects
-2 if incorrect

(c) [2 pts] Your non-short-circuiting compiler made it all the way to the Derby! These programs have boolean expressions that come from the following grammar:

$$b \rightarrow \text{true} \mid \text{false} \mid x \mid b == b \mid b\ != b \mid b \&\& b \mid b || b \mid !b$$

Give an example of code with boolean expressions exclusively from this grammar for which your group's compiler will generate code that produces incorrect outputs, or explain why this is not possible.

Answer:

This is not possible because the right hand side of any short circuit cannot have side effects in this restricted boolean grammar

-1 if no explanation
-2 if incorrect

Consider the following code:

```
int foo() {
  int x = bar();
  while (x < 100) {
    int y = baz();
    x += y;
  }
  return x;
}
```

Your task is to fill in the following x86-64 assembly skeleton by picking registers for variables x and y that minimize the total number of pushes and pops during execution (assuming that the loop body is executed at least 100 times), while satisfying the calling conventions described in class. You are allowed to pick from the following registers:

Caller-save	Callee-save
%r8, %r9, %r10	%r12, %r13, %r14

Remember that in this assembly syntax, the source operand is on the left hand side, and the destination operand is on the right hand side. **For compactness, assume we can push and pop multiple registers onto the stack with a single push/pop instruction; if no registers are given for a push/pop instruction, the push/pop is removed from the code.**

(d) [2 pts] First, fill in the code with y stored in %r10. Your code must minimize the total number of pushes/pops during execution (assuming that the loop body is executed at least 100 times), while satisfying calling conventions described in class.

```
foo:
  pushq ________________________   // store callee-save registers
  callq bar
  movq  %rax,  %_________          // store x
cond:
  cmpq  %_________, $100           // loop condition
  jge   end
body:
  pushq ________________________   // store caller-save registers
  callq baz
  popq  ________________________   // restore caller-save registers
  movq  %rax,  %r10                // store y
  addq  %r10, %_________           // add y to x
  jmp   cond
end:
  movq  %_________,  %rax          // return x
  popq  ________________________   // restore callee-save registers
  retq
```

Answer:

x *is stored in any callee-save register. You must push and pop* ***x*** *at the beginning and end of the function.*

There are no pushes/pops required for ***y****.*

-1 if using a caller-save instead of a callee-save
-1 if missing a necessary push/pop
-1 if including an unnecessary push/pop

(e) [2 pts] Now, fill in the code with y stored in `%r14` (the code is otherwise identical). Your code must still minimize the total number of pushes and pops during execution (assuming that the loop body is executed at least 100 times), while satisfying the calling conventions described in class.

```
foo:
  pushq ________________________   // store callee-save registers
  callq bar
  movq  %rax,  %________          // store x
cond:
  cmpq  %________, $100           // loop condition
  jge   end
body:
  pushq ________________________   // store caller-save registers
  callq baz
  popq  ________________________   // restore caller-save registers
  movq  %rax,  %r14               // store y
  addq  %r14, %________           // add y to x
  jmp   cond
end:
  movq  %________,  %rax           // return x
  popq  ________________________   // restore callee-save registers
  retq
```

Answer:

x is still stored in any callee-save register other than `r10`*. You must push and pop x at the beginning and end of the function.*

y must be pushed/popped at the beginning and end of the function (but not around the call site).

-1 if using a caller-save instead of a callee-save
-1 if missing a necessary push/pop
-1 if including an unnecessary push/pop

(f) [2 pts] Which of these choices for where to store y results in code with the fewest stack pushes/pops (assuming at least 100 iterations of the loop)?

Answer:

The first choice (x in callee-save, y in caller-save) has fewer pushes/pops.

-1 if incorrect because prior answers were incorrect