Quiz 1
6.035 Spring 2023 March 17, 2023

Before starting the quiz, write your name on this page.

There are 5 problems on this quiz. It is 13 pages long; make sure you have the whole quiz. You will have 50 minutes in which to work on the problems. You will likely find some problems easier than others; read all problems before beginning to work, and use your time wisely. The quiz is worth 58 points total. The point breakdown for the parts of each problem is printed with the problem. Some of the problems have several parts, so make sure you do all of them!

This is an open-book quiz. You may use a laptop to access anything on or directly linked to from the course website. You may also use any handwritten notes. You **may not** use the broader internet, any search engines, large language models, or other resources.

Do all written work on the quiz itself. If you are running low on space, write on the back of the quiz sheets and be sure to write (OVER) on the front side. It is to your advantage to show your work — we will award partial credit for incorrect solutions that are headed in the right direction. If you feel rushed, try to write a brief statement that captures key ideas relevant to the solution of the problem.

Name ______________________________

Kerberos ______________________________

Problem	Points	Score	Grader
1	12		
2	10		
3	10		
4	12		
5	14		
Total	58		

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).

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.

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.

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

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

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

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`

   ii. _____ `d`

   (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.

   (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.

3. **Parsing** [10 pts] (parts a–b)

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

$$
\begin{aligned}
S &\rightarrow X\ \mathbf{a}\ \$ \\
X &\rightarrow \mathbf{a}\ S \\
X &\rightarrow \mathbf{b}
\end{aligned}
$$

   (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.

(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;
 }
}
```

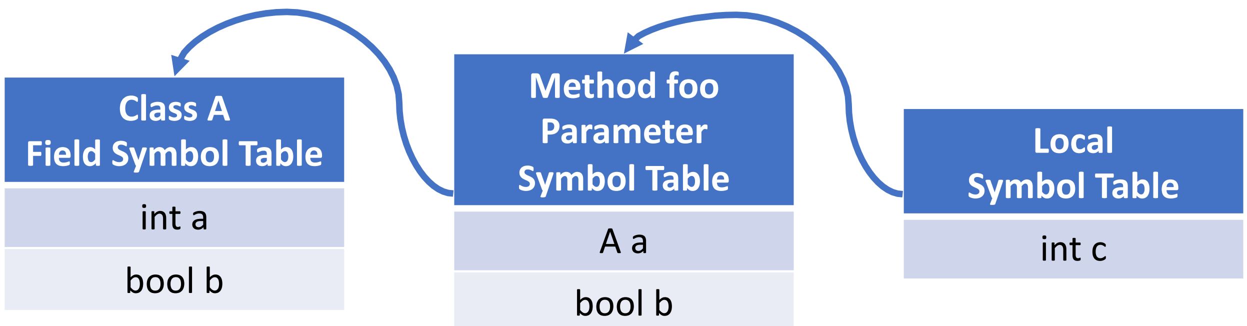
Correct / Incorrect

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

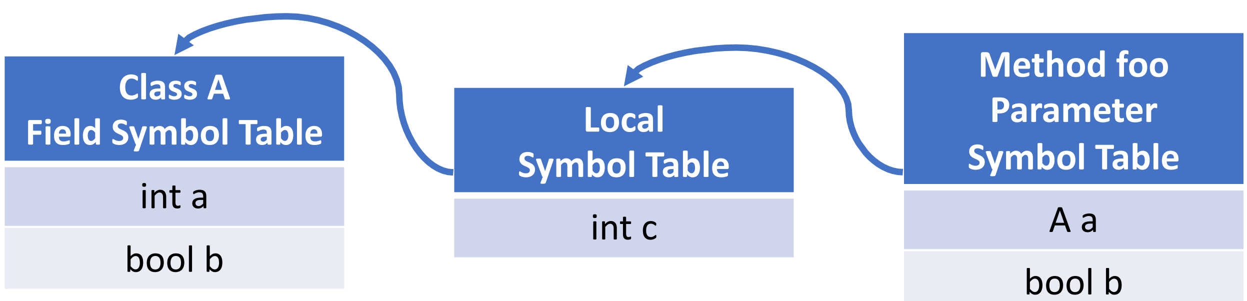
Consider the following class:

```
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:

(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:

Consider the following class hierarchy:

```
class X {
  int xa;
  int xb;
}
```

```
class Y extends X {
  int ya;
  int yb;

  Y foo(Y y) { ... }
}
```

```
class Z extends Y {
  int za;
  int zb;
}
```

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**

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.

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.

(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:

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

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.

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
```

(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
```

(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)?