# 6.110 Quiz 1 (Spring 2025)

Before starting the quiz, write your name on this page and read the following instructions:

- There are 5 problems on this quiz. It is 16 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 50 points total. The point breakdown each problem is given in the table below, and is also 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, **except for Godbolt.** You may also use any handwritten notes. You **may not** use Godbolt, any compilers, 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.

| Problem | Title | Points |
|---|---|---|
| 1 | Regular Languages | 8 |
| 2 | Top-Down Parsing | 9 |
| 3 | Semantics | 7 |
| 4 | Control Flow Graphs | 11 |
| 5 | Code Generation | 14 |
| | Feedback | 1 |
| | **Total** | **50** |

Name 6.110 Staff

MIT Email `6.110-staff@mit.edu`

1. **Regular Languages** [8 pts] (parts a–b)

   The EECS department is re-numbering all of the classes in Course 6 and needs your help! They would like to to write a grammar that can recognize valid course numbers, subject to some new constraints.

   For this problem, we will work over the alphabet $\Sigma = \{.\,(\text{period}), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Let $\mathsf{L}$ be the language consisting of all strings over $\Sigma$ satisfying all of the following conditions:

   - The string must start with the character **6**
   - The string must contain exactly one period **(.)**, and it must appear immediately after the first **6**
   - The last character must be a **0**, **1**, or **2**
   - Additionally, a **9** cannot occur in the string unless there is a **3** somewhere before it

   In this numbering system, **6.131415390**, **6.2392**, and **6.110** are in $\mathsf{L}$, but **6.1190**, **6.1357**, **6.08**, and **6.1.02** are not.

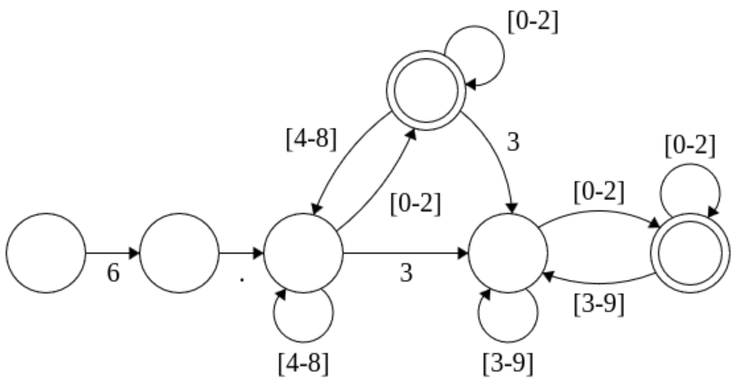
   (a) [4 pts] Write a regular expression that recognizes the language $\mathsf{L}$. You may only use regular expressions of the form introduced in lecture (i.e. the only operations that are allowed are concatenation, $|$, and $*$). You may also use the range operation to specify a range of numbers from **[first-last]** inclusive.

   **Answer:**

   $6.(([0-8]*3[0-9]*)|([0-8]*))[0-2]$

(b) [4 pts] Give a DFA that recognizes the language L. You may omit transitions to the failure state.

**Answer:**



2. **Parsing** [9 pts]  (parts a–b)

Alyssa has created a small language called *LIST* for manipulating linked lists of numbers. *LIST* is made of the following atomic (simple) expressions:

- The *nil* literal, `[]`, which represents a linked list with no items
- The first ten natural numbers (0, 1, 2, ..., 9)

*LIST* consists of two operators:

- The *cons* operator, `::`, is an operator that prepends a number to a linked list. The left side (LHS) of the operator must be a number. The right side (RHS) of the operator must be a linked list. The result of the *cons* operator is a linked list with *lhs* prepended to *rhs*.
- The *append* operator, `@`, is an operator that appends the second list (`l2`) to the first list (`l1`)

For example, executing the following code yields the following results:

```
[]                           # [] ← the empty list
1 :: 2 :: []                 # [1, 2]
(1 :: 2 :: []) @ (3 :: []) # [1, 2, 3]
5 :: 6                       # Error: RHS of :: operator must be a list
7 @ 8                        # Error: Both sides of @ operator must be lists
```

Alyssa wants to write a compiler for *LIST*. Before she writes any code, she comes up with the following (faulty) grammar:

```
<expr> ::= []
<expr> ::= <expr> :: <expr>
<expr> ::= <expr> @ <expr>
<expr> ::= ( <expr> )
<expr> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

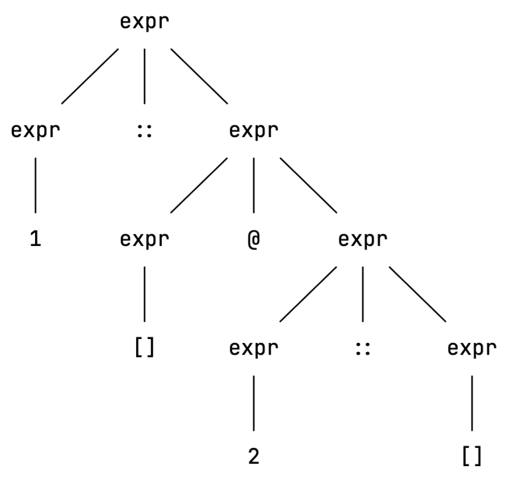
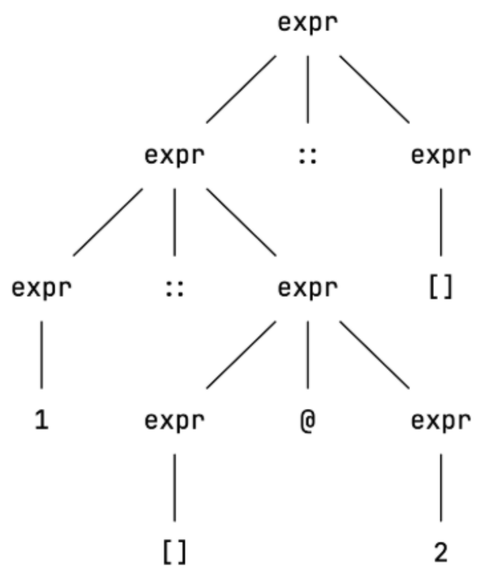
(a) [4 pts] Given Alyssa's grammar, the following expression will be ambiguous:

```
1 :: [] @ 2 :: []
```

Give two different parse trees for which the value that the tree evaluates to is a **different value** for each tree. If that expression evaluates to an error, write **ERROR**.

*For reference, here is Alyssa's faulty grammar:*

```
<expr> ::= []
<expr> ::= <expr> :: <expr>
<expr> ::= <expr> @ <expr>
<expr> ::= ( <expr> )
<expr> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

| Parse tree 1 | Parse tree 2 |
|---|---|
|  expr<br>expr<br>::<br>expr<br>1<br>expr<br>@<br>expr<br>[]<br>expr<br>::<br>expr<br>2<br>[] |  expr<br>expr<br>::<br>expr<br>expr<br>::<br>expr<br>[]<br>1<br>expr<br>@<br>expr<br>[]<br>2 |
| Evaluated value: [1,2] | Evaluated value: ERROR |

(b) [5 pts] Alyssa wants to rewrite her *LIST* grammar to be unambiguous. In addition to being unambiguous, your grammar must also abide by the following rules:

- The cons operator (`::`) must have a higher precedence than the append operator (`@`).
- The cons operator must be right-associative.
- The append operator must be left-associative.
- Both left-recursion and right-recursion are permitted.

*For reference, here is Alyssa's faulty grammar:*

```
<expr> ::= []
<expr> ::= <expr> :: <expr>
<expr> ::= <expr> @ <expr>
<expr> ::= ( <expr> )
<expr> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

**Answer:**

```
<expr> ::= <cons_expr> | <expr> @ <cons_expr>
<cons_expr> ::= <atom> | <atom> :: <cons_expr>
<atom> ::= [] | <number> | ( <expr> )
<number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

3. **Semantics** [7 pts] (parts a–c)

Consider the following program, which is from an object-oriented extension of Decaf that uses the typing rules discussed in class:

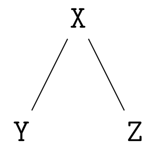
```
class X { ... }

class Y extends X {
    X foo(Z z) { ... }
}

class Z extends X {
    Z foo(Y y) { ... }
}

void main() {
    X x;
    Y y;
    Z z;
    ____(*)____
}
```

Class Hierarchy Diagram



(a) [5 pts] For each of the following statements, indicate whether they are valid or invalid at location (∗). Note: Y and Z both inherit from X.

| Statement | Valid or invalid? |
|---|---|
| `y = z.foo(y);` | invalid |
| `z = z.foo(x);` | invalid |
| `x = y.foo(y);` | invalid |
| `x = z.foo(y);` | valid |
| `x = y.foo(x);` | invalid |

(b) [1 pt] Suppose `Z` now extends `Y` instead of `X`. Which previously invalid call(s) are now valid?

**Answer:**

```
y = z.foo(y);
```

(c) [1 pt] Ignoring the change in the last part, suppose now that `Y` were to extend `Z` instead of `X`. Which previously invalid call(s) are now valid?

**Answer:**

```
x = y.foo(y);
```

4. **Control Flow Graphs** [11 pts]

   Louis Reasoner has just finished designing the control flow graph (CFG) IR for his Decaf compiler, listed below. Assume this is for a machine that only uses unsigned (non-negative) values.

| **Instruction** | **Semantics** |
|---|---|
| `d = const <const>` | Load constant `const` to variable `d` |
| `d = <op> s1, s2` | Evaluate `<op>` with values of `s1` and `s2` and store the result in variable `d`. `s1` and `s2` must be variables.<br><br>`<op>` can be: `add, sub, mul, div, and, or, eq, neq, lt, gt` |
| `d = <op> s` | Evaluate `<op>` with value of `s` and store the result in variable `d`. `s` must be a variable.<br><br>`<op>` can be: `not` |
| `jmp` | Unconditional jump. The destination is represented by an arrow. |
| `br <cond>` | Conditional jump. `cond` is a variable.<br>The destination is represented by two arrows, labeled `T` and `F`. |

   Louis needs your help converting his AST to his new CFG IR.

   (a) [3 pts] Linearize the expression `x = 2 * (x + y / x)` to a few lines of the IR listed above. You may assume `x` and `y` are all local variables and can be used as is in your CFG. You may also introduce temporaries $t_1, t_2, \ldots, t_n$ as necessary.

   **Answer:**

```
t1 = const 2
t2 = div y,  x
t3 = add x,  t2
x  = mul t1, t3
```

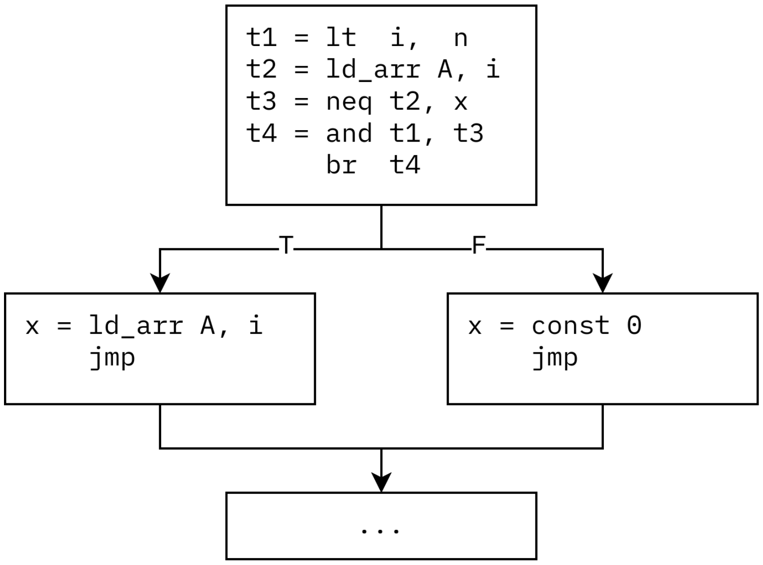
(b) [2 pts] Alyssa reminded Louis that his IR doesn't support arrays yet. To this end, Louis added two instructions, `ld_arr` and `st_arr`. These two instructions are not bounds checked.

| **Instruction** | **Semantics** |
|---|---|
| `ld_arr <arr>, <idx>` | Load index `<idx>` of array variable `arr`. |
| `st_arr <arr>, <val>` | Store `<val>` to array variable `arr`. |

Now, for the following code:

```
if (i < n && A[i] != x) { // n is the size of A
    x = A[i];
} else {
    x = 0;
}
```

Louis's compiler generates the following CFG.

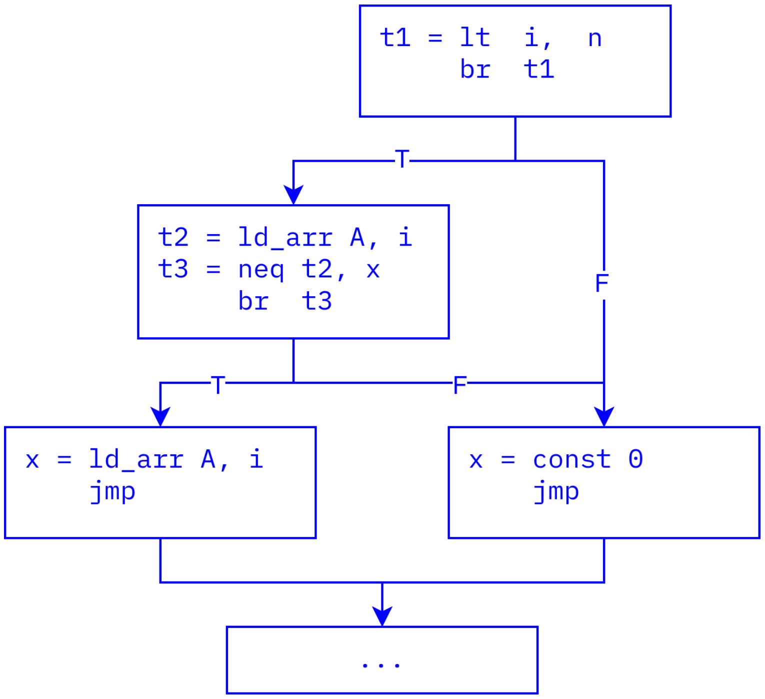


Explain what could go wrong with the CFG above.

**Answer:**
Access to array A is in the same block as the bounds check. When i is out of bounds, the `ld_arr` instruction will still execute, leading to undefined behavior (most likely segfault in practice).

(c) [4 pts] Louis recalls something called a short-circuit logic operator that could avoid the issue. Redraw Louis's CFG to use short-circuiting.

**Answer:**



(d) [2 pts] Louis now feels tempted to remove `and` and `or` instructions from his IR because they seem "useless." Do you agree with him? Explain with any reasonable justification one way or the other.

**Answer:**

Open-ended; credit may be given for well-justified arguments on both sides.

Yes: Short-circuit `and`s and `or`s can already be implemented via branching, so dedicated instructions for `and` and `or` are no longer necessary; Removing them simplifies code and reduces maintenance burden, etc.

No: Jumps are expensive on hardware, so non-short-circuit logical operators are still valuable for short expressions with no side effects; They can be repurposed to represent bitwise `and`/`or`, etc.

5. **Code Generation** [14 pts] (parts a–c)

   Ben Bitdiddle is looking to implement a new feature into Decaf. He wants to implement a right shift operator, where the symbol >> is used to perform a right logical shift (unsigned). An example of the use of this operator is below:

```
int a;
a = 0xF; // 0b1111
a = a >> 2; // a is equal to 0b0011;
```

   We will explore how we can go about helping Ben add this feature to Decaf in a hypothetical code generation scenario. For all x86 assembly code you may write, please follow the standard C calling convention and use AT&T syntax. As a reminder, AT&T syntax has the source operand is on the left, and the destination operand is on the right. Keep the stack pointer 16-byte-aligned when calling other procedures. Assume that when functions are called, the stack pointer is already 16-byte-aligned.

(a) [4 pts] Ben Bitdiddle wants to use this operator to create a Decaf function `right_shift_one(int num)`, which takes the number given and applies a right logical shift by one.

```
// Performs a right logical shift by 1 on the number
long right_shift_one(long num) {
   return num >> 1L;
}
```

Ben pulls up a popular x86 reference website and finds the following assembly instruction, listed here in AT&T syntax for your reference:

| Instruction | Op/En | 64-Bit Mode | Leg Mode | Description |
|---|---|---|---|---|
| SHR 1, r/m32 | M1 | Valid | Valid | Unsigned divide r/m32 by 2, once. |
| SHR 1, r/m64 | M1 | Valid | N.E. | Unsigned divide r/m64 by 2, once. |
| SHR CL, r/m32 | MC | Valid | Valid | Unsigned divide r/m32 by 2, CL times. |
| SHR CL, r/m64 | MC | Valid | N.E. | Unsigned divide r/m64 by 2, CL times. |
| SHR imm8, r/m32 | MI | Valid | Valid | Unsigned divide r/m32 by 2, imm8 times. |
| SHR imm8, r/m64 | MI | Valid | N.E. | Unsigned divide r/m64 by 2, imm8 times. |

For this table, r/m32 are 32 bit register or memory locations, r/m64 are 64 bit register or memory locations, and imm8 are 8 bit immediate values.

**Examine the assembly code below, and write (around 2-5 lines) that completes this function.** *(Hint: Recall that the first argument to a function is passed in register %rdi, and the return value is passed in register %rax)*

```
right_shift_one:
    pushq    %rbp
    movq     %rsp, %rbp
```

**Answer:**

```
    shrq    $1, %rdi
    movq    %rdi, %rax
```

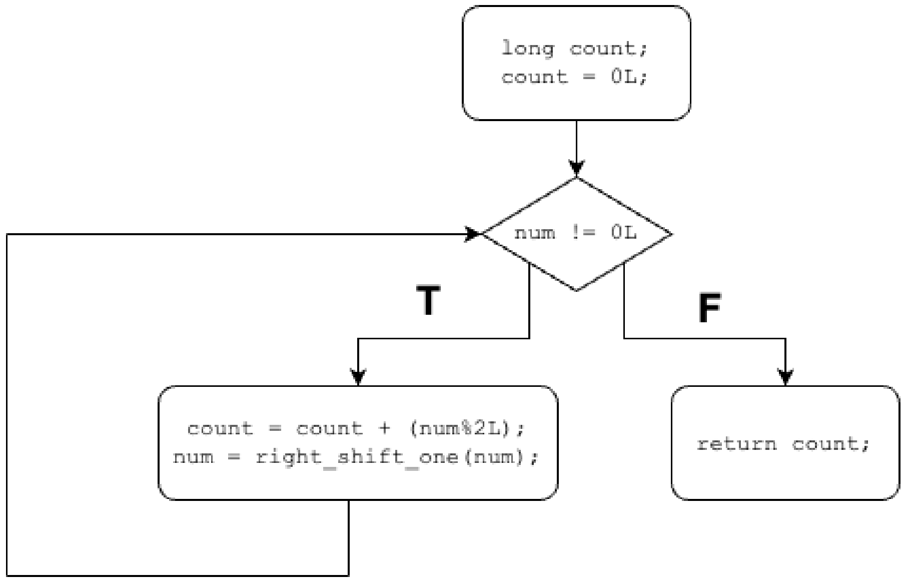
```
    popq     %rbp
    ret
```

(b) [4 pts] Now, Ben wants to use this function to implement a simple algorithm that counts the number of binary ones in a number.

```
// Counts the number of binary ones in a number. num must be positive
long count_ones(long num){
   long count;
   count = 0L;
   while (num != 0L) {
       count = count + (num % 2L);
       num = right_shift_one(num);
   }
   return count;
}
```

Draw the control flow graph of his function, which is written in Decaf. You may use the form given in Lecture 6 (Codegen), and quote the Decaf statements/expressions literally. Be sure to annotate the edges with branch conditions as appropriate:

**Answer:**



(c) [6 pts] Ben Bitdiddle takes the CFG that you drew and hand-wrote the x86 code that corresponds to the function. However, some parts of it are missing! Fill in the blanks in Ben's code that will make his x86 code functionally correct and also follow all calling conventions. Assume `right_shift_one` is implemented correctly.

**Please put your final answers in the table on the next page.** The table provides a helpful hints for what types may go in the blanks. You may use this page for scratch work.

```
count_ones:
   # allocate space on the stack
   pushq   %rbp
   movq    %rsp, %rbp
   subq    ___(1)___ , %rsp
   movq    %rdi, -16(%rbp) # num
   movq    $0, -8(%rbp) # count
.while_header:
   # num != 0L
   cmpq    ______(2)______
   ___(3)___    .exit
.while_body:
   # num % 2
   movq    $0, %rdx
   movq    -16(%rbp), %rax
   movq    $2, %rdi
   idivq   %rdi
   # remainder stored in %rdx
   addq    %rdx, ___(4)___
   # function call to right_shift_one
   # move num to parameter 1
   movq    ______(5)______
   call    right_shift_one
   # move result to num
   movq    %rax, -16(%rbp)
   __________(6)__________
.exit:
   # move count to return register
   movq    -8(%rbp), %rax
   # restore the stack
   addq    ___(1)___ , %rsp
   movq    %rbp, %rsp
   pop     %rbp
   ret
```

**Final answers for question 5(c):**

| | **Hint** | **Your answer** |
|---|---|---|
| **(1)** | An immediate | any multiple of `$16` |
| **(2)** | An immediate, followed by a register or memory location | `$0, -16(%rbp)` |
| **(3)** | An instruction | `je`, `jz`, `jle`, `jbe` |
| **(4)** | A register/memory location | `-8(%rbp)` |
| **(5)** | Two register/memory locations | `-16(%rbp)`, `%rdi` |
| **(6)** | A full instruction | `jmp .while_header` |

---

[1pt] Feelsbox, or tell us a concept that you struggled with that you would like the TAs to go over again.

**Answer:**

N/A