6.110 Quiz 1 (Spring 2024)

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

- There are 6 problems on this quiz. It is 15 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, 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	Conceptual Questions	6
2	Regular Languages	8
3	Parsing	12
4	Semantics	8
5	Linearizing Expressions	4
6	Code Generation	12
	Total	**50**

Name 6.110 Staff

Kerberos `6.110-staff`

1. **Conceptual Questions** [6 pts] (parts a–f)

   State whether each of the following statements are true or false, by writing either T or F in the blank space before each statement.

   (a) All regular languages are also context-free languages.

   True

   (b) All NFAs with at most n states can be converted into DFAs with at most n^2 states.

   False

   (c) In the high-level IR discussed in lecture, when there are two nested scopes, the symbol table for the inner scope contains a pointer to the symbol table for the outer scope.

   True

   (d) When destructuring the high-level IR as discussed in lecture, if two statements end up in the same basic block in the control-flow graph, the must have shared the same scope in the high-level IR.

   True

   (e) Recall that short-circuit semantics evaluates only the minimal number of operands required to determine the condition. Under short-circuit semantics, some evaluations of the boolean exclusive-or operator (XOR) may evaluate only the first operand.

   False

   (f) In the x86 assembly calling convention defined in class, the caller must always save all caller-saved registers on the stack before calling any procedure.

   False

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

   For this problem, we will work over the alphabet $\Sigma = \{A, B, C\}$. Let L be the language consisting of all strings over Σ satisfying all of the following conditions:

   - The letter immediately before the first occurrence of the letter B is an A.
   - The letter immediately before the first occurrence of the letter C is a B.
   - The letter C appears at least once in the string.

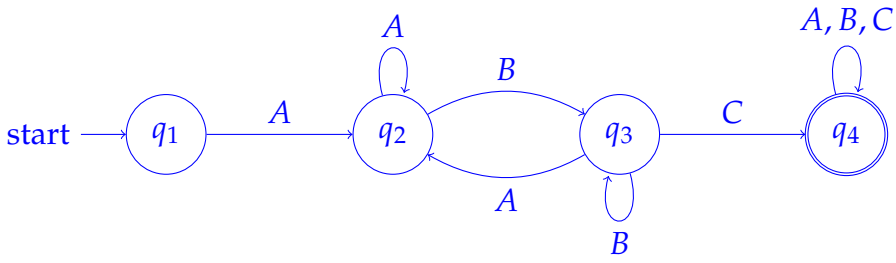
   For example, the strings ABC and $AABABBCCBA$ are in L, but the strings BC and $ABAC$ are not.

   (a) [4 pts] Write a regular expression that recognizes the language L. You may only use regular expressions of the form introduced in lecture (i.e. the only operations that are allowed are concatenation, $|$, and $*$)

Answer:
$A(A|B)*BC(A|B|C)*$
Other answers are also possible, such as $AA*\Big(B\Big|B(A|B)*B\Big)C(A|B|C)*$.

(b) [4 pts] Give a DFA that recognizes the language L.

Answer:



3. **Parsing** [12 pts] (parts a–c)

   This question involves parsing boolean expressions containing the ternary conditional operator. The *ternary (conditional) operator* `?:` is an operator that takes in three expressions. Its value is equal to either of the latter two input expressions, chosen based on the value of the first input expression. In other words, an expression

   ```
   <condition> ? <value-if-true> : <value-if-false>
   ```

   evaluates to the value of `<value-if-true>` if `<condition>` evaluates to true, and evaluates to the value of `<value-if-false>` if `<condition>` evaluates to false.

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

   ```
   bool b;
   b = true ? false : true; // b = false
   b = false ? false : true; // b = true
   ```

   Your teammate Melon Usk proposes the following grammar for expressions containing the ternary operator that only involves booleans:

   ```
   <expr> ::= <expr> ? <expr> : <expr>
   <expr> ::=  true | false
   ```

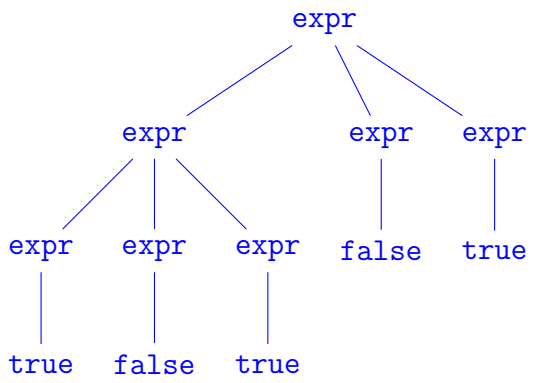
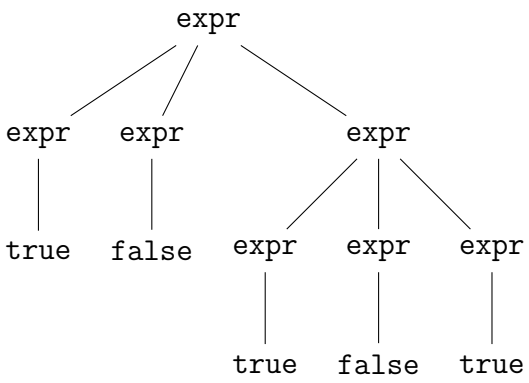
(a) [4 pts] You immediately notice that this grammar is ambiguous: the expression

```
true ? false : true ? false : true
```

can be parsed in two ways. Complete the following table that shows the two parse trees and the value that the expression evaluates to for each parse tree. One of the parse trees has already been provided as an example.

For reference, here is Melon Usk's proposed grammar:

```
<expr> ::= <expr> ? <expr> : <expr>
<expr> ::=  true | false
```

Parse tree 1 (left-associative)	Parse tree 2 (right-associative)
	
Evaluated value: `true`	Evaluated value: `false`

(b) [4 pts] You decide that the ternary operator should be *right-associative*, as in the second parse tree from part (a). (This is also the case in most real programming languages.) Propose a modification to Melon Usk's proposed grammar so the only valid parse trees are the ones that respect right-associativity.

For reference, here is Melon Usk's proposed grammar:

```
<expr> ::= <expr> ? <expr> : <expr>
<expr> ::=  true | false
```

Answer:

```
<expr> ::= <bool> | <bool> ? <expr> : <expr>
<bool> ::=  true | false
```

(c) [4 pts] Your teammate Alyssa P. Hacker now wants to add the `&&` (logical and) operator to the grammar. The `&&` operator should have a higher precedence (i.e. binds more tightly) than the ternary operator. Namely, the following string

```
true ? false : true && false ? true : false
```

should be interpreted as

```
(true ? false : ((true && false) ? true : false))
```

Propose a new grammar that parses all boolean expressions containing the `&&` operator and the ternary operator and respects the precedence as described. For full credit, the grammar should also respect right-associativity of the ternary operator, as in part (b).

Answer:

```
<expr> ::= <term> | <term> ? <expr> : <expr>
<term> ::= <bool> && <term>
<bool> ::=  true | false
```

4. **Semantics** [8 pts] (parts a–b)

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

```
class A { ... }

class B extends A {
    C f(A a) { ... }
}

class C extends A {
    B f(B b) { ... }
}

void main() {
    A a;
    B b;
    C c;
    ____(*)____
}
```

(a) [5 pts] For each of the following statements, indicate whether they are valid or invalid at location `(*)`.

Statement	Valid or invalid?
`a = b.f(c);`	valid
`a = c.f(a);`	invalid
`b = b.f(b);`	invalid
`b = c.f(b);`	valid
`c = b.f(c);`	valid

(b) [3 pts] Fill in exactly one of each of `a`, `b`, and `c` in the blanks in the following expression so that it type-checks:

```
c .f( b ).f( a )
```

5. **Linearizing Expressions** [4 pts]

   Convert the following Decaf statement into a linearized form by writing an equivalent list of simplified statements.

   Original statement: `a = b * (c + d) / (e - f);`

   We have already written some statements for you, and you may only add statements in the form of

   $$t_i \leftarrow t_j \ \texttt{op} \ t_k,$$

   where $t_0, t_1, t_2, \ldots$ are temporary variables, and `op` is a binary arithmetic operation.

```
// load variables
t1 ← b
t2 ← c
t3 ← d
t4 ← e
t5 ← f

// compute the expression (ADD YOUR STATEMENTS HERE)
t6 ← t2 + t3
t7 ← t1 * t6
t8 ← t4 - t5
t0 ← t7 / t8
// store the result
a ← t0
```

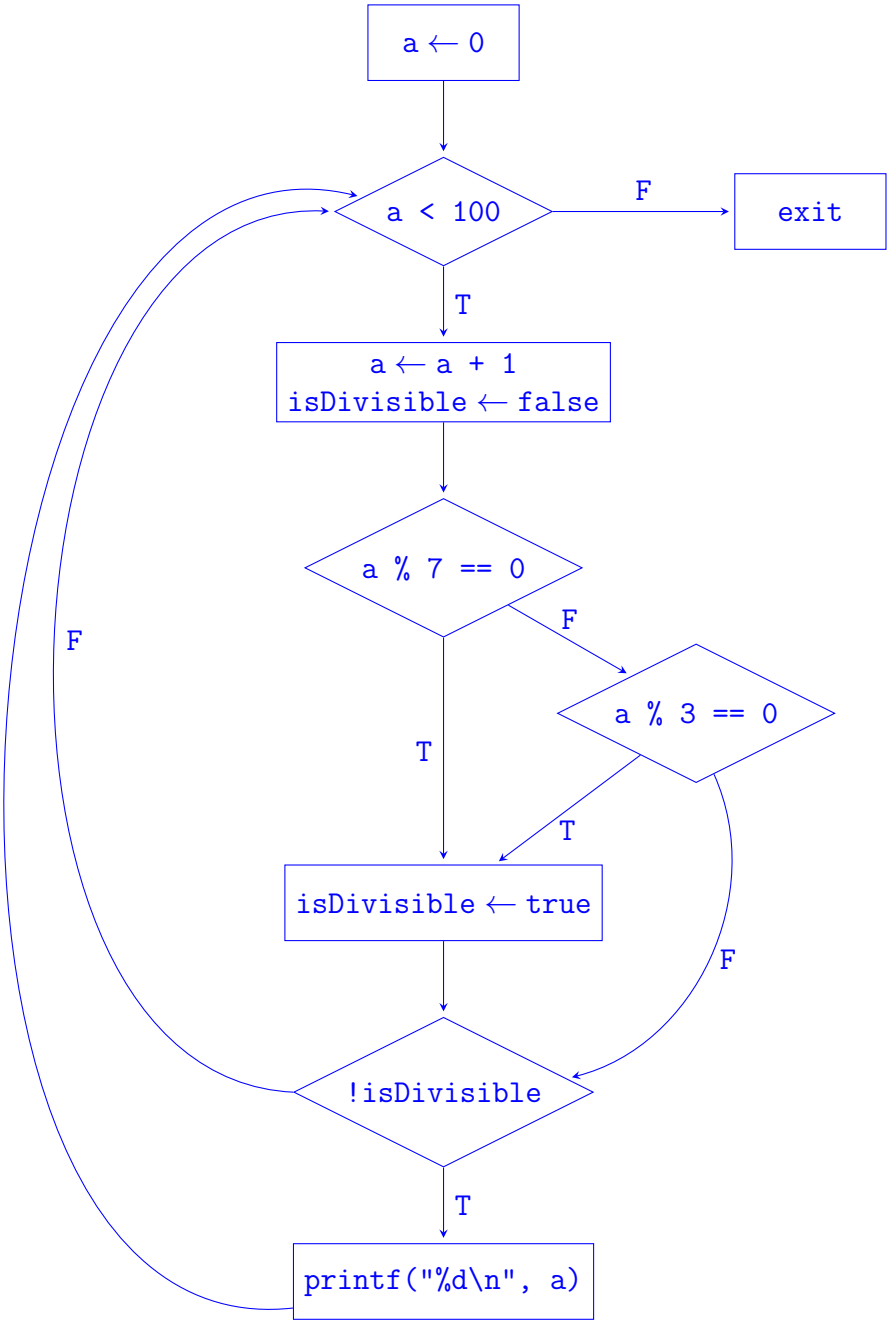
6. **Code generation** [12 pts] (parts a–b)

Ben Bitdiddle writes the following Decaf program, which prints all positive numbers less than 100 that are divisible by either 7 or 3.

```
void main() {
    int a = 0;
    while (a < 100) {
        a += 1;
        bool isDivisible = false;
        if (a % 7 == 0 || a % 3 == 0) {
            isDivisible = true;
        }
        if (!isDivisible) {
            continue;
        }
        printf("%d\n", a);
    }
}
```

(a) [5 pts] Draw a control flow graph for this program. Recall that boolean operations in Decaf have short-circuiting semantics and evaluate from left to right. Do not perform any optimizations.

Answer:



(b) [7 pts] For better code modularity, Ben Bitdiddle decides to move the logic of checking whether a number is divisible to its own function. The new program is as follows:

```
bool isDivisibleByThreeOrSeven(int num) {
    // ... omitted for clarity ...
}

void main() {
    int a = 0;
    while (a < 100) {
        a += 1;
        if (isDivisibleByThreeOrSeven(a)) {
            printf("%d\n", a);
        }
    }
}
```

Ben Bitdiddle then implemented `main` in x86 assembly by hand. Ben's implementation is on the last page of this quiz, but some parts of his code are missing, as indicated by blanks. Fill in the blanks in Ben's code. **Please put your answers in the table on this page.**

The completed code should store all local variables on the stack, follow the standard C calling convention, and keep the stack pointer 16-byte-aligned when calling other procedures. Assume that `isDivisibleByThreeOrSeven` is already implemented, and when `main` is called, the stack pointer is 16-byte-aligned.

	Hint	Your answer
(1)	This should be an immediate	any multiple of `$16`
(2)	This is an instruction	`jge`, `je`, or `jns`
(3)	This is a register or a memory location	`%rdi`
(4)	This is a register or a memory location	`%rax`
(5)	This is an instruction	`jne`, `jl`, or `jz`
(6)	This is a register or a memory location	`%rsi`
(7)	This should be an immediate	multiple of `$16`, same as **(1)**

Assembly code for Problem 6 (b)

The code on this page is for reference only. You may tear this page off, but you should return it with the rest of your quiz at the end of the quiz. You may use this page as scratch paper, but **please indicate your final answers in the table in question 6 (b).**

```
str:
    .string "%d\n"
    .align 16
main:
    push    %rbp
    movq    %rsp, %rbp
    subq    __(1)__, %rsp
    movq    $0, -8(%rbp)  //  int a = 0;
while_cond:
    cmpq    $100, -8(%rbp)
    __(2)__   exit_loop
while_body:
    addq    $1, -8(%rbp)
if_cond:
    movq    -8(%rbp), __(3)__
    call    isDivisibleByThreeOrSeven
    cmpq    $1, __(4)__
    __(5)__   while_cond
if_body:
    leaq    str(%rip), %rdi
    movq    -8(%rbp), __(6)__
    movq    $0, %rax
    call    printf
    jmp     while_cond
exit_loop:
    addq    __(7)__, %rsp
    movq    %rbp, %rsp
    pop     %rbp
    movq    $0, %rax    //  return with exit code 0
    ret
```