6.1100 Spring 2024 Miniquiz #3

There are 5 pages. Please submit your answers on Gradescope by Feb 29th, 2024, 11:59pm.

Name:

Email:

1. Intermediate Representation

Consider the following class.

```
class foo {
    bool x[10];
    int y;
    void bar(int a) {
        int i = 1;
        y += a+i;
    }
}
```

Using the representations discussed in lecture, construct a diagram containing the details one would need for implementing a successful compiler. Namely, your diagram should illustrate:

- A class descriptor for `foo`.
- A method symbol table and field symbol table.
- A method descriptor.
- Two field descriptors, one of which is an array descriptor.
- A local symbol table for `bar` and any relevant variable descriptors.
- A reasonable representation of the statements executed by `bar`.

Make sure to also draw arrows between relevant components.

(On the actual quiz, you will not need to remember the exact representations we used in lecture, but it is good practice to draw out such a diagram.)

(This page is intentionally left blank in case you need space to draw the diagram.)

2. Scope and Semantics

In the following program, **B** is a subclass of **A**.

```
class A { ... }
class B extends A { ... }

B f (A x) { ... }
A g (B x) { ... }

void main() {
    A x;
    _____(1)_____
    if (true) {
        B x;
        _____(2)_____
    }
}
```

Alyssa P. Hacker wants to insert a line of code from the options below in either location **(1)** or location **(2)**. For each of the options, indicate whether they would be valid at each of the locations. An example is given in the first row.

	Code to be inserted	Location **(1)**	Location **(2)**
example	`x = f(x);`	*valid*	*valid*
a.	`x = f(f(x));`		
b.	`x = f(g(x));`		
c.	`x = g(f(x));`		
d.	`x = g(g(x));`		

3. Short-Circuiting and Semantics

You have implemented a Decaf compiler correctly while your friend Melon Usk forgot to implement short-circuiting.

For each Decaf program, circle one of the statements, and then state the output of the programs compiled by the two compilers.

```
int a = 1, b = 1;
bool c;
bool function_x() { a = 2; return true;}
bool function_y() { b = 3; return false;}
void main() {
    c = function_x() && function_y();
    printf("%d, %d, %d", a, b, c);
}
```

The two programs have the same output *The two programs have different outputs*

Outputs: ____

```
int a = 1, b = 1;
bool c;
bool function_x() { a = 2; return true;}
bool function_y() { b = 3; return false;}
void main() {
    c = function_x() || function_y();
    printf("%d, %d, %d", a, b, c);
}
```

The two programs have the same output *The two programs have different outputs*

Outputs: ____

After testing a few example programs on your friend's compiler, you found that Melon Usk not only did not implement short-circuiting but also reversed the precedence of the **&&** and **||** operators! In a correct Decaf compiler, the **&&** operator has higher precedence (i.e. binds more tightly) than the **||** operator, but in Melon Usk's compiler, the **||** operator has higher precedence than the **&&** operator.

For each Decaf program, circle one of the statements, and then state the output of the two compilers.

```
int a = 1, b = 1;
bool c;
bool function_x() { a = 2; return true;}
bool function_y() { b = 3; return false;}
void main() {
    c = function_y() && function_x() || function_y();
    printf("%d, %d, %d", a, b, c);
}
```

The two programs have the same output　　　*The two programs have different outputs*

Outputs: ____

```
int a = 1, b = 1;
bool c;
bool function_x() { a = 2; return true;}
bool function_y() { b = 3; return false;}
void main() {
    c = function_x() || function_y() && function_x();
    printf("%d, %d, %d", a, b, c);
}
```

The two programs have the same output　　　*The two programs have different outputs*

Outputs: ____