Homework 6

This homework covers the following topics from class:

- Function palooza, part 2
    - More on Parameters (Positional vs. Named Parameters, Optional Parameters and Default Parameters, Variadic Functions), Returning Values and Error Handling (error objects, optionals, assertions/invariants)
- Function palooza, part 3
    - Error Handling Part 2 (Exceptions, Panics), First-class Functions (Lambdas/Closures Across Languages)

Each problem below has a time associated with it. This is the amount of time that we expect a thoroughly-prepared student would take to solve this problem on an exam. We understand, however, that as you learn these concepts, these questions may take more time to solve, and we don't want to overwhelm you with homework. For that reason, only **required** questions need to be completed when you submit this homework. That said, the exams will cover all materials covered (a) in class, (b) from class projects, (c) in the required homework problems, and (d) in the optional homework problems, so at a minimum, please review the optional problems and/or use them as exam prep materials.

You must turn in a PDF file with your answers via Gradescope - you may include both typed and hand-written solutions, so long as they are legible and make sense to our TAs. Make sure to clearly label each answer with the problem number you're solving so our TAs can more easily evaluate your work.

For homework #6, you must complete the following **required** problems:

- DATA9: Binding Semantics, Parameter Passing, Pass By Need (8 min)
- FUNC1: Default Parameters (5 min)
- FUNC2: Lambdas, Closures (5 min)
- FUNC3: Error Handling, Optionals, Exceptions (10 min)
- FUNC4: Results, Optionals, Errors, Exceptions (8 min)
- FUNC5: Exceptions (10 min)

For homework #6, you may optionally complete the following additional problems:

N/A

DATA9: Binding Semantics, Parameter Passing, Pass By Need (8 min)

Consider the following program that looks suspiciously like Python (but we promise you, it isn't!):

```
def foo(a):
  a = 3
  bar(a, baz())

def bar(a, b):
  print("bar")
  a = a + 1

def baz():
  print("baz")
  return 5

a = 1
foo(a)
print(a)
```

Assume that in this language, formal parameters are mutable.

Part A: (2 min.) Suppose you know this language has pass-by-value semantics. What would this program print out?

Your answer:

Part B: (2 min.) Suppose you know this language has pass-by-reference semantics. What would this program print out?

Your answer:

Part C (2 min.) Suppose you know this language has pass-by-object reference semantics. What would this program print out?

Your answer:

Part D: (2 min.) Suppose you know this language has pass-by-need semantics. What would this program print out?

Your answer:

FUNC1: Default Parameters (5 min)

(5 min.) Khoi has decided to create a new programming language called Lit and allow functions to have default parameters in any position (not just for the last N arguments as in C++ and Python). For example, he says he wants this to be a valid function definition:

```
func addNumbers(x: Int = 10, y: Int, z: Int = 40) -> Int {
  return x + y + z
}
```

He knows that if he uses this approach, there may be ambiguities, e.g.:

```
func main() {
  result := addNumbers(20, 30) // is this x=20, y=30, z=40 or x=10, y=20, z=30?
}
```

So he needs some advice. He wants to know what to require for function calls to ensure there's no ambiguity as to which arguments go to which parameters. How would you design function calls to eliminate ambiguity as to which arguments are being passed to which formal parameters? For example, if you required all arguments to be named, this would eliminate any ambiguity:

```
result := addNumbers(y: 20, z: 30)
```

Give at least two additional approaches that might work. Be creative, there's no right answer!

Your answer:

FUNC2: Lambdas, Closures (5 min)

(5 min.) Consider the follow JavaScript program which passes a lambda to the *callLambda* function:

```
function callLambda(action) {
  action();
  action();
}

function main() {
  let counter = 0;

  // Pass a lambda to another function
  callLambda(() => {
    counter++;
    console.log(`Counter: ${counter}`);
  });

  console.log(`Final Counter in main: ${counter}`);
}

main();
```

This code outputs:

```
Counter: 1
Counter: 2
Final Counter in main: 2
```

What must be happening in JavaScript to enable this functionality? How can the calls to the lambda expression (e.g., action()) possibly modify variables defined in main? Specifically, explain how you think JavaScript closures work to enable this behavior.

Your answer:

FUNC3: Error Handling, Optionals, Exceptions (10 min)

(10 min.) Consider the following C++ struct:

```
template <typename T>
struct Optional {
  T *value;
};
```

If `value` is `nullptr`, then we interpret the optional as a failure result. Otherwise, we interpret the optional as having some value (which is pointed to by `value`).

Next, consider two different implementations of a function that finds the first index of a given element in an `int` array:

```
Optional<int> firstIndexA(int arr[], int size, int n) {
  for (int i = 0; i < size; i++) {
    if (arr[i] == n)
      return Optional<int> { new int(i) };
  }
  return Optional<int> { nullptr };
}
```

```
int firstIndexB(int arr[], int size, int n) {
  for (int i = 0; i < size; i++) {
    if (arr[i] == n)
      return i;
  }
  throw std::exception();
}
```

Compare our generic `Optional` struct with C++'s native exception handling (throwing errors). Discuss the tradeoffs between each approach, and the different responsibilities that consumers of either API must adopt to ensure their program can handle a potential failure (i.e. element not found). Also discuss which approach is more suitable for this use case, and why.

Your answer:

FUNC4: Results, Optionals, Errors, Exceptions (8 min)

In this problem, you will choose the most appropriate error handling mechanism for each scenario and give a brief explanation justifying your choice.

For each problem, you may choose from:

- Error Objects
- Status Objects
- Result Objects
- Assertions
- Exceptions
- Panics

Part A: You have been asked to write a function that accepts a URL, validates its format, and if it is properly formatted, extracts and returns its domain. It's unimportant to know why the URL was malformed. What error handling mechanism would you choose and why?

Your answer:

Part B: You are building a module to load the configuration data for the flight control software that operates a rocket engine. If the configuration is invalid the rocket will fail catastrophically. What error handling mechanism would you choose and why?

Your answer:

Part C: You are building an internal API that will only be called by other software components you've built and will not accessed by external users. The API can only operate correctly on inputs of size 1 to 1000 elements, with other sizes causing the API to result in undefined behavior. What error handling mechanism would you choose and why?

Your answer:

Part D: You are building a database engine which reads and writes to cloud storage systems. These cloud storage systems have a 99.9% uptime but do occasionally fail, resulting in the transaction having to be repeated from scratch. What error handling mechanism would you choose and why?

Your answer:

FUNC5: Exceptions (10 min)

For this problem, you'll need to consult C++'s exception hierarchy. Consider the following functions which uses C++ exceptions:

```
void foo(int x) {
 try {
   try {
     switch (x) {
      case 0:
       throw range_error("out of range error");
      case 1:
       throw invalid_argument("invalid_argument");
      case 2:
       throw logic_error("invalid_argument");
      case 3:
       throw bad_exception();
      case 4:
       break;
     }
```

```
    }
    catch (logic_error& le) {
      cout << "catch 1\n";
    }
    cout << "hurray!\n";
  }
  catch (runtime_error& re) {
    cout << "catch 2\n";
  }
  cout << "I'm done!\n";
}


void bar(int x) {
  try {
    foo(x);
    cout << "that's what I say\n";
  }
  catch (exception& e) {
   cout << "catch 3\n";
   return;
  }
  cout << "Really done!\n";
}
```

(10 min) Without running this code, try to figure out the output for each of the following calls to bar():

```
bar(0);

bar(1);

bar(2);

bar(3);
```

```
bar(4);
```

Your answer: