Homework 4 – Fall 2022 (Due: May 3, 2022)

In this homework, you'll explore the concepts in data palooza: type systems, scoping, and binding. Compared to other homeworks, you won't be writing *any* code for this one – these questions are focused on your understanding of key concepts. Some questions have multiple, distinct answers which would be acceptable, so there might not be a "right" answer: what's important is your ability to justify your answer with clear and concise reasoning that utilizes the appropriate terminology discussed in class. Each question has a time estimate; you'll know you're ready for the exam when you can solve them roughly within their time constraints.

We understand, however, that as you learn these questions may take more time. For that reason, only **starred red** questions need to be completed when you submit this homework (the rest should be used as exam prep materials). Note that for some multi-part questions, not all parts are marked as red so you may skip unstarred subparts in this homework.

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.

Type, Set, Math

1. Type relations are very important for mathematical operators in programming languages.

a) ** (8 min.) Recall that in Haskell, we've seen a variety of number types (well, type classes): `Int`, `Integer`, `Num`, `Float`, and `Fractional`. Since we didn't cover these in-depth in class, you may want to read A Gentle Introduction to Haskell's section on Numbers.

Which of these types are supertypes of other types? Which are subtypes? Which are not related at all?

b) ** (4 min.) The input and output types for Haskell's `div`, `mod`, and `+` are different, even though they all operate on numbers. Why?

Hint: Try running `:t div` and `:t +` in `ghci`).

c) ** (4 min.) C++ has `float`, `int`, `const float`, and `const int` (among other number types). Which of these are supertypes or subtypes of each other, and which are unrelated?

d) (4 min.) Give one example of an expression with a mathematical operation in any programming language where one of the input types is a subtype of the result's type. Briefly explain why.

Classify That Language (Late Night Edition)

We're such huge fans of "Classify That Language", so we've brought you a special episode in this homework.

2.

a) ** (2 min.) Consider the following code snippet from a language that Matt worked with in his summer internship:

```
user_id = get_most_followed_id(group) # returns a string
user_id = user_id.to_i
# IDs are zero-indexed, so we need to add 1
user_id += 1
puts "Congrats User No. #{user_id}, you're the most followed
user in group #{group}!"
```

Without knowing the language, is the language dynamically or statically typed? Why?

b) ** (5 min.) Here's an (adapted) example from a lesson that Matt taught a couple of years ago on a "quirky" language:

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
// this prints:
```

```
// 2
// Error: x is not defined
```

Briefly explain the scoping strategy this language seems to use. Is it similar to other languages you've used, or different?

Hint: Try writing the same function in C++.

c) ** (2 min.) This block of code from a mystery language (that Matt adores!) compiles and outputs properly.

```
fn main() {
  let x = 0;
  {
    let x = 1;
    println!(x);  // prints 1
  }

  println!(x);    // prints 0

  let x = "Mystery Language";
  println!(x);    // prints 'Mystery Language'
}
```

We'll note that even though it doesn't look like it, **this language is statically typed.** With that in mind, what can you say about its variable scoping strategies? How is it similar or different to other languages you've used?

Hint: The scope of `let x = 0;` is only lines 2, 7, 8, and 9.

d) ** (5 min.) With reference to variable binding semantics, examine the behavior of this language.

In particular, comment on the differences between `n1 == n2` and `s1 == s2`. Is this similar to other languages you've used?

```
n1 = 3
n2 = 3
n1.object_id
# 7
n2.object_id
# 7
puts n1 == n2
# true
# ...
s1 = "hello"
s2 = "hello"
puts s1.object_id
# 25700
puts s2.object_id
# 32880
puts s1 == s2
# true
s2 = s1
puts s2.object_id
# 25700
```

(fun fact: Matt has written type coercion libraries for this language, and had to use what we learned this week to do it!)

Zeta Programming

3. You're working on a project at Zeta to add Haskell-like type inference to Python.

   (Fun fact: Meta actually does have an open-source gradual typing/type inference Python system called Pyre. Microsoft has one called Pyright)

a) ** (2 min.) Consider this function: if you could add a Haskell-like type annotation, what would it look like? Alternatively, if you can't annotate it like Haskell, why not?

```
from math import sqrt

def nth_fibonacci(n):
  phi = (1 + sqrt(5))/2
  psi = (1 - sqrt(5))/2
  return (phi**n - psi**n)/sqrt(5)
```

(Fun fact: this actually works!)

b) (2 min.) Consider this function. Assume that `network_type` is always a String. Your boss says that it's not possible to annotate this function, because its return type is not the same for every input (which breaks Haskell's rules about functions). You're convinced there's still a way to do this. What could that look like?

```
def get_network_type(obj):
  if not hasattr(obj, 'network_type'):
    return None
  return obj.network_type
```

Hint: get algebraic!

c) ** (3 min.) Consider this function.

```
def add(a,b):
  return a + b
```

It seems easy, right? Your boss provides the annotation:

```
add :: Num -> Num -> Num
```

Is this the best annotation? Why or why not?

Union Busting

4. **

a) ** (4 min.) Examine the following C++ code:

```
union WeirdNumber {
  int n;
  float f;
};

int main() {
  WeirdNumber w;
  w.n = 123;
  std::cout << w.n << std::endl;
  std::cout << w.f << std::endl;
}
```

This evaluates to:

```
123
1.7236e-43
```

Why? What does this say about C++'s type system?

b) ** (7 min.) Zig is an up-and-coming language that in some ways, is a direct competitor to C and C++. Let's examine a similar union in Zig.

```
const WeirdNumber = union {
  n: i64,
  f: f64,
};

test "simple union" {
  var result = WeirdNumber { .n = 123 };
  result.f = 12.3;
}
```

```
test "simple union"...access of inactive union field
.\tests.zig:342:12: 0x7ff62c89244a in test "simple union"
(test.obj)
result.float = 12.3;
```

Assuming you haven't used Zig (but, crucially – you do know how to read error messages): what does this tell you about Zig's type system? How would you compare it to C++'s – and in particular, do you think one is better than the other?

How to Save a Lifetime

5.

a) ** (5 min.) Consider this Python code:

```
num_boops = 0
def boop(name):
  global num_boops
  num_boops = num_boops + 1
  res = {"name": name, "booped": True, "order": num_boops }
  return res

print(boop("Arjun"))
print(boop("Sharvani"))
```

For `name`, `res`, and the object bound to `res`, explain:

- What is their scope?
- What is their lifetime?
- Are they the same/different, and why?

b) ** (3 min.) Consider this C++ code:

```
int* n;
{
  int x = 42;
  n = &x;
}
std::cout << *n;
```

This is undefined behavior. In the language of scoping and lifetimes, why would that be the case?

c) (4 min.) It turns out, this code tends to work and print out the expected value of 42 – even though it is undefined behavior. Why might that be the case?

Hint: This has to do with *something else* covered in data palooza!

Bonus Question

These is a **bonus question**; it extends upon course material, but is not at all required. The concepts it tests you on are relevant for the midterm, but we will not ask you questions like this on an exam. In particular, this used to be a common interview question!

IIFEs

6. (8 min.) JavaScript has a (now-antiquated) pattern called an Immediately Invoked Functional Expression, or IIFE for short. Here is a code example which demonstrates one such behavior.

```
// Define a nameless function that takes in no parameters
(function () {
  var a = 5;
  console.log(a**2);
})(); // note: the () here is immediately invoking the function

// prints out 25

console.log(a) // ReferenceError: a is not defined
```

In your own words, what does this do, why is it necessary, and how does it relate to scoping?