Homework 4 – Spring 2023 (Due: May 03, 2023)

In this homework, you'll explore the concepts in data palooza: type systems, scoping, and binding (we've pushed memory management to HW5). 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?

This answer is as instructive as we can make it; a more concise answer with the correct relationships would get full marks on a test!

- `Int` is a fixed-bit representation of an integer, while `Integer` is an unbounded representation. Since any `Int` is also an `Integer`, and an `Int` can be used anywhere an `Integer` can be used, we conclude that **`Int` is a subtype of `Integer` (and `Integer` is a supertype of `Int`)!**
- `Float` is a fixed-bit representation of a floating-point number, while `Fractional` is a generic class type (that allows for *any* number that supports arbitrary division). Since any `Float` is (by definition) a `Fractional`, and a `Float` can be used anywhere a `Fractional` can be used, we conclude that **`Float` is a subtype of `Fractional` (and `Fractional` is a supertype of `Float`)!**
- Even though `Int`, `Integer`, `Float`, and `Fractional` all seem to support similar operations, note that they have *different division operators* (`div` versus `/`). This means that you *can't* use an `Int` where you expect a

`Fractional`, and vice-versa! So, **`Int`/`Integer` and `Float`/`Fractional` are not subtypes or supertypes of each other.**

    ○ Note: you can also make an argument about floating-point precision representing a different set of numbers than integers, which would preclude them from being related in a sub/supertype relationship.

- `Num` is a base type class for all of `Int`, `Integer`, `Float`, and `Fractional`. In any place where a `Num` is expected, we can substitute an `Int`, `Integer`, `Float`, or `Fractional`; and, any of those classes are (by definition) a `Num`. So, we conclude that **`Num` is a supertype of `Int`, `Integer`, `Float`, and `Fractional` (i.e. they are subtypes of `Num`)!**
    - ○ Note: class inheritance doesn't *always* imply subtyping; we'll learn more about this in our OOP lectures!

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`).

If you run `:t` on `div`, `mod`, and +, you'll notice that the first two take an `Integral` input, while the last takes a `Num` input. This is because `div` and `mod` encode *integer division*. You can't use these on any `Fractional` number (e.g. a `Float`, like 3.2).

In contrast, + works on any number - `Integrals`, `Fractionals`, and others (like complex numbers). So, they take any `Num` (the most general number type class).

This all boils down to Haskell's type system looking for the most generic types that any function can have as a parameter or return type.

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?

We covered this in lecture (see slide 64 in data palooza):

- **`float` is a subtype of `const float`**
- **`int` is a subtype of `const int`**
- **`int` and `float` (and `const int` and `const float`) do not represent the same set of numbers (precision with floating-point, bounds of integers), so they cannot be subtypes or supertypes of each other**
    - **ex: 1.5 is a float but not an int; fixed-bit floating-point representations cannot encode the number 9999999999999999999, but ints can.**

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.

There are many! To Matt, the prototypical example is Python's subtyping of Booleans as Integers (see Numeric docs). Consider the expression `3 + True` in Python: Python coerces the `True` into `1` (and `False` into `0`), and then performs integer addition.

You may ask, why are Python Booleans a subtype of Integers? Given the mapping of `True` to `1` and `False` to `0`,

1. **A Boolean can be used anywhere an Integer is expected**
2. **Every element in {0,1} belongs to the Integers**

There are many other examples of "widening" operations; we'll list a few:

- **Other Integer operations with Boolean inputs: `3 * True, 3 / True, 3 - True, 3 > True` also work**
- **In languages that support complex numbers, multiplying a real number by `i`. In Python, floats are a subtype of the builtin complex type.**
- **In languages/libraries with support for Rational numbers (like Haskell), a statement like `1 / 5` does widen the type from Integer to Rational. *Rationals are not Floats!***

An operation that takes an integer to a (fixed-precision) float is *not* a valid answer. Integers and floats are *not* supertypes or subtypes of each other; see the answer to part c.

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?

This is Ruby. It is dynamically typed:

- **get_most_followed_id returns a string**
- **you could guess that .to_i is an integer conversion, so the type of user_id is changing in the program**
    - **but, even if you didn't know that...**
- **we are performing integer addition to user_id, so at this point user_id has definitely changed types from string to integer**

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++.

This answer is as instructive as we can make it; a more concise answer would get full marks on a test!

This is JavaScript. First, we can immediately note that this language seems to use lexical scoping. If it was dynamically scoped, the first `console.log(x)` would print 1, since that's the last place (in the program) that x was set. In a dynamically scoped language like bash, this would have logged 1. Thus, we can infer that this language scopes variables to some lexical construct (a function, a block, or something else).

Furthermore, we note that this language is *not* Brewin' – if it was (and only had global variables), then the second `console.log(x)` wouldn't error. But, it does – and thus, our job is to figure out what the lexical scope is.

What's curious is that the `console.log` in boop prints 2, and not an undefined variable error. Languages like C++ (and most mainstream languages) lexically scope to blocks (e.g., if-else, while, for-loop blocks). Since the declaration is not in the same block as the print statement, this would be an error in C++. In contrast, JavaScript's `var` keyword scopes to functions. This behaves similarly to Python (try it!). For full marks on a test, we'd expect you to note this.

An aside: JavaScript is one of the only mainstream languages that has *different scoping rules* depending on what keyword you use. If you're curious, check out the MDN docs on `let` (the other general declaration).

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.

This is Rust; the example is almost directly copied from the Wikipedia page on shadowing.

This language is lexically scoped: if it wasn't, the second print statement would also print "1". It seems to lexically scope based on blocks, similar to C++ (see part c).

However, the really interesting part happens on line 10: we somehow are allowed to redefine a variable with the same identifier, *and it has a different type!* In a statically typed language, this is only really possible if we treat them as different variables. The logical conclusion of this is that Rust must allow variable shadowing *in the same scope* (a relatively unique feature); we can confirm this by reading the docs on shadowing.

So the definition of `x` on line 10 is actually creating a new variable `x` of type `String`, which hides the old variable `x` of type `Int` defined on line 2. Both variables still exist on lines 10/11, but only the most recently defined x is accessible (the original x is shadowed and no longer accessible!). Note: This is super unusual - most of the time when we see the ability to "change a type" of a variable, we're dealing with a dynamically-typed language.

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!)

This is also Ruby.

The observations we're looking for you to make are:

- **for the "number/int" type, it seems like the same literal value always has the same object ID – a shared reference**
    - **this is not enough information to infer that *all* numbers behave like this. When you see this, it's usually safe to assume that this works for "small enough" numbers, i.e. a 32-bit or 64-bit int.**
    - **in Ruby, this isn't true for arbitrarily-large numbers; try: 99999999999999999999999999999999999999999999999**
- **but, for the "string" type, it seems like the same literal value can have different object IDs – not a shared reference**
- **yet, the equality operator seems to account for this:**
    - **for the "number/int" type, it could be either performing a value comparison or an ID comparison - you can't tell from this example**
    - **but, for the "string" type, it's definitely performing a value comparison - the strings are equal, even though their object IDs aren't!**

- **you could then infer that Ruby probably uses object reference semantics (which it does); however, technically this example isn't enough to confirm or deny that**

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!)

For this one, we can annotate it. On an exam, we would expect an annotation of the form:

```
nth_fibonacci :: Float -> Float
```

or, alternatively,

```
nth_fibonacci :: Int -> Float
```

(You can use different type classes, like `Double` or `Floating`; these would also be accepted on an exam.)

How did we get there? In detail:

- `phi` and `psi` are both constants; in particular, a `Float`
- the `**` operator (exponent) on a `Float` expects another `Float` as its right-hand side argument, so we can infer that n is a `Float`
- a `Float` divided by another `Float` returns a `Float`, so we conclude that the return value is a `Float`

In fact, this function adheres to all FP principles, and we could write an equivalent function in Haskell. In GHCi:

```
nth_fibonacci n =
  let phi = (1 + sqrt(5))/2
      psi = (1 - sqrt(5))/2
  in (phi**n - psi**n)/sqrt(5)
```

`:t nth_fibonacci`
would give us something of the form
`nth_fibonacci :: Float -> Float`

(there's some small nuance about `**` only applying to `Float`s in Haskell, but we're not expecting you to know that)

Aside: in a purely mathematical sense, this actually will *always* return an integer for an integer input. But, that involves a math proof, and Haskell's type system doesn't do that. `Integer -> Integer` would also be too narrow of a type!

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!

First, to clarify: this function takes in *any* Python object. If it has the `network_type` attribute, it returns the value of that object; if not, it returns None. If you're curious, read the `hasattr()` docs.

In Python, everything is an object! This means that our input can be *literally anything*: an integer literal, a function, a class we define, or something else. That means we don't put any restriction on what it can be; so, we'll use a type variable for it; we'll call ours `object`.

If we add some extra code, we could write an annotation for the return value. If we have an ADT of the form:

```
data StringOrNone = Str String | None
```

We could then annotate the function like so:

```
get_network_type :: object -> StringOrNone
```

This idea of a function "maybe" returning something is really common; Haskell provides support for this using the `Maybe` monad. Alternatively, you could write:

```
get_network_type :: object -> Maybe String
```

However, we didn't cover the monads (or `Maybe`) in class; you're not expected to provide this answer.

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

```python
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?

No! In Python, + is overloaded and also operates on other types: lists, strings, and any class that implements the `__add()__` method – broader than Num!

(many students answered something to the form of "we want an `int + int` to be an `int`". That is true, though in Haskell, we typically look for the most generic type – this is not how + is typed, for example)

Union Busting

4. **

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

```cpp
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?

This happens because C++ lets you access any type of the union at any time, even if it's not the one that's "active". `1.7236e-43` happens to be the floating-point representation encoded by the *bits* that 123 sets for an `int`.

In programming language terminology, this is called an untagged union.

Implicitly, this is an argument that C++ is weakly-typed.

- **`float`'s don't have a singular bit representation in C++: their precision depends on compiler flags, machine architecture, etc.!**
    - **We could even interpret this as an implicit cast**
- **Thus, this program does not have clear behavior across different environments; we could frame this as undefined behavior or as type-unsafe behavior.**

    - **In Carey's slides, this is probably in the "unsafe memory access" category, or an undefined type conversion**
- **There is also a more strict definition of "undefined behavior" in the C++ spec. Whether or not this is actually undefined behavior is not clear-cut (see: this SO post). Our interpretation of the C++11 spec on Unions (Section 9.5, page 220), is that this is undefined behavior.**

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?

First, interpreting the error: it's indicative that Zig does not allow arbitrary access for any of the types that are part of the union. Instead, we can only use the "active" union field (in Zig's language). We can verify this in the Zig docs.

In programming language terminology, this is called a tagged union.

This leans towards a more strongly-typed type system. To compare and contrast this to C++:

- **Zig's approach is "safer": it requires us to explicitly tell the language when we want to access various union fields, and prevents undefined behavior. The idea is to prevent mistakes!**
- **C++'s approach is faster. There is some overhead when Zig checks union access; C++ does not have that overhead.**

Of course – there's a spectrum of valid opinions here, and we wouldn't ask you what the *correct* opinion is on an exam – just to compare and contrast.

Matt's opinion: Zig's approach is better. Many bugs in systems code come from memory unsafety (in fact, Matt has exploited memory safety bugs in C code in an internship)! There's a whole class of programming languages that are designed to be as fast as C/C++, but much safer by preventing memory access issues. Rust and Zig are the most popular of these languages.

How to Save a Lifetime

5.

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

```python
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?

For `name`: the scope and the lifetime of the variable itself are the same (starting from the parameter name, and ending at the return statement). This is not the same as the string that is bound to `name`, which has a lifetime that exists before, during, and after the function call.

For `res`: the scope and the lifetime are the same (starting from the declaration, and ending at the return statement). Note that, even though the object's lifetime persists after the return statement, the identifier/variable `res` doesn't!

For the object bound to `res`: the lifetime starts within the boop function, but extends until the end of the program – since we then print the result! Note that this is *different* from the lifetime of `res`. For values, we don't really have a concept of a scope.

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++ is lexically scoped, so the scope of the variable x ends with the closing brace. In addition, for local variables in C++, the lifetime and scope of variables is the same! So, after the program reaches the closing brace, the lifetime of x – and, the memory it refers to – is over.

Since n points to the same memory address as x, after the closing brace we're accessing a value whose lifetime already ended. Essentially, this is a dangling pointer! This demonstrates one way in which C/C++ are weakly typed.

Aside: Rust's Reference and Borrow system (the "borrow-checker") prevents errors of this type! This lets Rust provide "compile-time memory safety".

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!

This has to do with C++'s memory management model and optimizations. Broadly (and in most programming languages), when a variable's scope and lifetime ends, the memory *holding* its value isn't zeroed out. Instead, it just stays there! Usually, it'll only get changed once that piece of memory is "given" to another variable. On top of that, there's no garbage collector, so memory isn't constantly being reallocated. And, we haven't made another addition to the stack. So, it's likely that the memory that holds 42 still does hold the same bits when we print it.

This is still undefined behavior, and you shouldn't write code like this!

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?

IIFEs are a way to avoid "namespace pollution": when too many variables are in the global namespace, and can cause confusing code or name collisions. This is a problem in many languages, but particularly a problem in JavaScript (due to how the module system *used* to interact with the global namespace).

To briefly explain what's going on in the first few lines of code: we're defining an anonymous function (basically, a lambda). Wrapping the function definition in

parentheses makes it an expression. Then, we immediately call this function, hence the name: Immediately Invoked Function Expression

Like most programming languages, JavaScript is lexically scoped. In this example, the variable `a` is scoped to the parent function. In particular, it is *not* in the global scope, which we can see by `console.log(a)` exhibiting a reference error.

So, the benefit of using the IIFE is: we can define a variable with a very limited scope by wrapping it in a function.

You may ask, why couldn't we just do:

```
{
  var a = 5;
  console.log(a**2);
}
```

Matt's answer is: look at Question 2 part b) :)

As an aside: Matt chose to include this question since he's been asked it in interviews for full-time frontend engineering roles!