Homework 4 - Spring 2023

- Vincent Lin
- [UID REDACTED]
- Section 1D

Question 1

Question 1a

`Num` seems to be a supertype of all the other types as it represents a general numeric type.

From the attached reading:

> The `Num` class provides several basic operations common to all numeric types; these include, among others, addition, subtraction, negation, multiplication, and absolute value:

`Float` is also a subtype of `Fractional` because the latter is the one the provides the ordinary division operator `/`.

`Int` and `Integer` are not subtypes of `Fractional` but are subtypes of `Num` as their range of possible values are a subset of that of `Num` and they implement all the operations of `Num` described above.

Question 1b

The outputs of the hint is as follows:

```
ghci> :t div
div :: Integral a => a -> a -> a
ghci> :t (+)
(+) :: Num a => a -> a -> a
```

We see that `div` only works on `Integral` types, that is numbers that have an integer representation. This makes sense because division on integers is a very different operation from floating point division, which has its own operator (`/`). The addition operation the other hand is well-defined for all numeric types, so the signature for `+` is a lot more flexible, accepting a broader supertype of `Num` for its operands.

Question 1c

`float` is a subtype of `const float`. This is because the two requirements are satisfied:

- `float` and `const float` have an identical set of values, so by definition `const float`'s set of values is a **superset** of that of `float`.
- `float` supports all the operations that `const float` has, like arithmetic, comparison, etc. operations. `float` in addition supports assignment, but this does take away the subtype property.

`int` is a subtype of `const int` for the identical reasons as above.

The `int` types and `float` types are not subtypes or supertypes of each other, assuming a standard modern architecture where `int` and `float` are both 32 bits. `int` cannot hold floating point values. `float` cannot hold all integers that `int` can because some of `float`'s bits are reserved for the exponent part of its floating point representation, resulting in a smaller range of values for its mantissa part.

Part 1d

```c
// This is in C.
int main(void)
{
    double d = 3.14;
    float f = 3.14;
    printf("%f\n", d + f);
    return 0;
}
```

`float` is a subtype of `double`, but when a `float` is added to a `double`, the resulting type is a `double`.

Question 2

Question 2a

Assuming this code runs without errors, this language is most likely **dynamically typed**. This is because the variable `user_id` is first bound to a string, but then it is rebound to an integer later, namely the result of the string plus 1.

Question 2b

The scoping strategy seems to be by function. In the example, the variable `x` in the function definition is defined within the `if` block, but is still in scope at the line `console.log(x)`. This shows that this language does not use block scoping like C/C++. Instead, the variable is visible for the rest of the function after it is defined, similar to in Python. Variables are expectedly still out of scope outside of the function they are defined in, and this example demonstrates that with the last `console.log(x)` statement that raises an error.

Question 2c

The scoping strategy seems to be by block. In the example, `x` is shown to have a value of 1 inside the `{}` block from lines 3-6, but `0` before and after it (due to it having a value set to 0 on line 2). This is similar to languages like C/C++.

Question 2d

Variables seem to be names that are bound to values in memory, and these names can be reassigned at runtime to reference other values. The equality comparison operator `==` also seems to check equality of **value**, not equality of **reference**. These binding semantics seem similar to those in Python.

In the example, `n1 == n2` is true because their values (3) are equal, and they *happen* to be the same object in memory as well (perhaps the integer 3 is a value that is **interned** by the language). `s1 == s2` is true despite `s1` and `s2` referencing distinct objects in memory because their values (`"hello"`) are the same.

Question 3

Question 3a

```
nth_fibonacci :: Integer -> Integer
```

Question 3b

I'm unsure how to do this because it doesn't seem that we can just do something like `data A = Nothing | String`. I asked ChatGPT:

> **Prompt:** what would the haskell equivalent of a Union[str, None] return type look like?

And the relevant code output (with prior context about the `get_network_type` Python function) was:

```
data Obj = Obj { networkType :: Maybe String }

getNetworkType :: Obj -> Maybe String
getNetworkType obj = networkType obj
```

Question 3c

This is not the best annotation if it were to apply to the Python function `add`. This is because the `+` operator is overloaded such that it can work not just on numeric types but also on strings, where it acts as the string concatenation operator. `String` is not a subtype of `Num`, so the boss' annotation does not capture the most general case possible.

Question 4

Question 4a

We can see that the members of a `union` in C++ share the same memory and thus the same bit pattern. We output a strange number with `w.f` because the bit pattern for the `int 123` is simply being *reinterpreted* as if the bit pattern were that for a `float`.

This also means that `w.f` is an **implicit cast** - the data for the `int 123` is reused for the data for the `float` version.

Question 4b

In contrast to C++, it seems that `union`s in Zig can only assume the type of the member with which it is initialized. In the example, the `i64` member `n` is initialized, which prohibits the subsequent action of assigning the `f64` typed `f` member on the next line.

It is debatable if one system is better than the other. Zig's system can prevent mistakes involving interpreting the data with the wrong type. However, there are practical applications for using a `union` to store raw data as

bits and then interpret them with different types later, such as interpreting the contents of a network packet, and this is more convenient in C++'s system.

Question 5

Question 5a

The scope of `name` and `res` is the `boop` function. Those names are variables local to the function and are not defined (or are possibly bound to other values) outside of it, so attempting to access them will not cause the intended behavior. The object bound to `res` is a **value** and not a **variable**, so scope does not apply to it.

The lifetime of `name` and `res` is the `boop` function. Variables in Python are names bound to values on the heap instead of containers representing the memory for the value itself. Thus, after the function finishes executing, the names `name` and `res` are no longer defined (unless they were defined with other values outside the function, in which case the `name` and `res` within `boop` **shadowed** them). The lifetime of the object bound to `res` is from the line it was defined (`res = {...}`) until the end of the program because the two calls to `boop` at the end of the program reference this object as function return values.

The scope and lifetime of `name` and `res` happen to be the same in this example, and this is a consequence of how local variables work in Python. The object bound to `res` has a different scope and lifetime because as a value, the prior does not apply to it but the latter does, and the object as a value will persist in memory from the time it was created until it is **garbage collected**; in Python, this can only happen *after* the object's reference count drops to zero.

Question 5b

`x` is only in scope within the `{}` block, but more importantly, its lifetime is also confined to that block. `n` is in scope and has a lifetime beyond the block, but since it's set to a pointer to a variable `x` whose lifetime expires outside the block, the memory associated with it can be reclaimed. This results in undefined behavior if we attempt to dereference (`*x`) because that memory is no longer in use by our program.

Question 5c

Skipped.

Question 6

This pattern is a way to execute a block of code without modifying the global namespace. It does this by defining an anonymous function and then immediately executing it. Scoping rules in JavaScript state that variables defined inside a function have a scope local to it, so within the function we are free to define whatever helper variables we want without interfering with the global namespace. The function itself is also anonymous, so it does not add any name to the global scope either.

This is similar in spirit to the `do {} while (0)` pattern in C macros, where we take advantage of block scoping to define a multi-line sequence of statements without interfering with the enclosing scope:

```
#define do_something()      \
    do                      \
    {                       \
```

```
        int a = thing1();   \
        int b = thing2(a);  \
        if (b < 0)          \
            printf("hi\n"); \
    } while (0)

int main(void)
{
    int a = 5;
    do_something();
    // a is still 5 here.
}
```