hw

1.

a)

`Fractional` is a supertype of `Float`. `Int` and `Integer` are subtypes of the `Integral` type and `Int` is also a subtype of `Integer` since `Int` is bounded to a certain amount of bits while `Integer` is unbounded. All of these are subclasses of `Num`.

b)

In Haskell, `div` and `mod` are only allowed for members of the `Integral` supertype (in other words, integer variants like `Int` and `Integral`). This is because they perform integer operations like truncated division and modulo. `(+)` is under no such constraints and is thus implemented for all subtypes of `Num`, including fractional (floating point) numbers.

c)

In C++, `float` is a supertype of `const float` since it allows all of the same operations as float, as well as the addition assignment operation which a `const float` does not allow since it must remain constant. The same relationship holds for `int` and `const int`. `const float` and `const int` are unrelated (so it follows that `float` and `int` are unrelated as well.)

2.

a)

This language is dynamically typed because we initially have the variable `user_id` holding a value that is a string and then later we convert it to an integer, meaning that the type of the variable changes during runtime (or really it means that the variable does not have a type and that it holds a value whose type changes at runtime).

b)

This language seems to use a scoping strategy where variables are not bound to their block scope (e.g. if they are defined in an `if` statement they can be accessed outside of that statement) but rather variables are bound to their function scope (they can only be used in the function they are defined in). This is somewhat similar to languages I've used before but slightly different in that I would've expected the `x = 1` line to either set a global variable `x` or perhaps mutate the previous definition of `x` (if it was still in scope after being defined in the previous function).

c)

This language uses block scoping strategies and defines closures within curly brackets. Variables can be shadowed and redeclared in nested scopes. This is similar to other languages I've used in that variables are scoped to the block they are defined in. It is maybe slightly different in that it can define a closure/block without any real reason (i.e. the closure in the example is not part of a conditional statement or loop or something).

d)

It looks like this language uses object references to hold values (judging by the `object_id` field). I can't see any immediate differences between `n1 == n2` and `s1 == s2` but it is possible that the former compared the `object_id` fields while the latter would have to have compared the actual values of the objects (the strings themselves). This would make it similar to Python.

3.

a)

If I could add a type annotation it would look something like this:

```
def nth_fibonacci(n: int) -> int:
    # ...
```

However this would not work that great because this function returns a float, even though it is meant to return an integer. To make this type annotation hold we would need to convert the return value to an `int()` before returning.

b)

We could set the return type to an enum `Option` that either contains `Some(value)` or `None` .

c)

This is not the best annotation because it loses some specificity due to the fact that `Num` is a supertype of all numbers and thus this annotation would allow us to add, say, an integer and a float (which we would maybe not want to do without requiring an explicit conversion). Perhaps a better annotation would be to specify with generics that both arguments must be a subset of the `Num` type AND both must be of the same type (and return said type).

4.

a)

This is because a union only holds one value at a time and C++ will implicitly cast the bits to whichever type you access at that point. This works because C++ is weakly typed, which is why it allows odd behavior like this where data can be interpreted as any type without conversion or proper validity checking.

b)

This seems to show that Zig is strongly typed, as it implements the necessary checks to make sure you can't accidentally (or on purpose) subject yourself to any of the weird behavior above caused by unsafe casting. I think that this type system is better because it seems more modern and intuitive.

5.

a)

`name` is in scope for the `boop` function only, while its lifetime is bound to the return value of the function (in this case it dies after being called because the return value is not assigned to anything). `res` is also in scope only in the `boop` function (once it becomes defined) and its lifetime lasts the same amount of time. The object bound to `res` is not ever really in scope since it is a value but its lifetime remains for the duration of the scope of `res` and further until the lifetime of the return value of the `boop` function ends. In this way it is different from the actual `res` variable.

b)

This is because we are setting `n` to point to the address of the variable `x`, but `x` is only in scope for the block enclosed in curly brackets. Thus after the block ends the variable is out of scope and dies and the value the address points to becomes undefined.