Question 1

Language and Requirements

This language starts with the same semantics as Egg-eater, and adds support for:

- IEEE-754 double-precision 64-bit floating-point numbers ("floats").
- Performing appropriate arithmetic and comparison over floats.
- An enhanced suite of numerical predicates.

The runtime system must add support for:

- Structural equality over floats.
- Printing floats.

Additional Syntax

For this addition the grammar for the language doesn't change much, only adding a single terminal expression which is parsed into its own entry in the `expr` data-type.

```
type 'a expr =
  ...
  | EFloat of float * 'a
```

For clarity's sake, we're also going to rename `ENumber` to something more specific like `EInteger`.

Examples

Floating point numbers can be used to perform arithmetic - with other floats and with integers.

```
begin
  2.1 + 6.3;  # 8.4
  8.4 - 6.3;  # 2.1
  2.5 * 2.2;  # 5.5

  4 + 4.3;    # 8.3
  -2.3 - 6;   # -8.3
  8.5 * 2;    # 17.0

  add1(3.3);  # 4.3
  sub1(-10.2) # -11.2
end
```

They can also be compared against other floats and against integers.

```
begin
  4.3 > 2.3; # true
  4.3 < 2.3; # false
  4.3 > 4.3; # true
  4.3 < 4.3; # true

  4.3 > 4;  # true
  4.3 < 4;  # false
  4.0 >= 4; # true
  4.0 <= 4  # true
end
```

Representation

In order to follow the IEEE-754 specification our floats need to use every one of their available 64 bits and cannot accommodate a tag. Thus, as with tuples, the `snakeval` itself is going to be a tagged pointer to somewhere on the heap where the actual IEEE-754 value is stored.

The updated tag scheme is as follows, where `w` is a "wildcard" bit:

- Integers: `www0`
- Booleans: `1111`
- Tuples: `w001`
- Floats: `w011`

Modifications to Pre-Compilation Phases

Since floating-point numbers have no internal structure at a language level they require no special treatment before compilation.

- If they make it past the parser they are vacuously well-formed.
- No special desugaring is required.
- In `anf` they will be treated as a non-immediate lettable `cexpr` as tuples are, because they live on the heap.
- They have no local variables so there is no impact on stack allocation.

Semantics

Arithmetic

As shown in the above examples, all our existing arithmetic operations work on floats and any such operation which involves a float results in a float.

Comparison

As shown in the above examples, all our existing comparison operations work on floats and floats are comparable with integers.

Equality

Because floats are allocated on the heap, pointer equality and structural equality for them can disagree. As with tuples this is a bit strange from a language perspective for what are seemingly basic values, but the performance of == when used appropriately is desirable enough to be maintained.

```
4.3 == 4.3;            # false
let a = 4.3 in a == a; # true
equal(4.3, 4.3)        # true
```

Thus, one should use structural equality to compare floats with anything else. Using structural equality with one integer and one float should follow real principles, especially because `==` *absolutely* doesn't work between integers and floats.

```
equal(4, 4.0); # true
equal(4.0, 4); # true

4.0 == 4; # false
4 == 4.0 # false
```

Predicates

- `isNum` should return true for floats as they are indeed numbers.

```
isNum(4.3) # true
```

- The new `isInteger` should only be true for integers.

```
isInteger(4.3); # false
isInteger(4)    # true
```

- The new `isFloat` should only be true for floats.

```
isFloat(4.3); # true
isFloat(4)    # false
```

Compilation Changes

The biggest change to the compiler for this feature, as may be expected, is in the final assembly emission - to both instantiate the floating-point values in memory and integrate them into our existing operations.

Since floats have their own series of assembly instructions (`fadd`, `fsub`, `fcom`, etc.), the complexity of the pre-operation type-checking increases. Now, instead of being able to assume all "numbers" are machine integers, the type of each operand needs to be explicitly checked for integer-ness or float-ness beforehand.

Given the results of this checking the appropriate assembly instruction(s) need to be emitted. Additionally, both the order of operand types and the commutativity (or not) of the operation itself impact which instruction(s) are correct.

In our current type paradigm, as our numeric power grows the number of run-time type checks will grow faster - something to ponder for the future.

Question 2

Language and Requirements

This language starts with the same semantics as that described in Question 1, and adds support for:

- Complex numbers, as in `A{+|-}Bi`, where both A and B are real numbers (integers and floats).
- Performing appropriate arithmetic over complex numbers.
- An enhanced suite of numerical predicates.

The runtime system must add support for:

- Structural equality over complex numbers.

- Printing complex numbers.

Additional Syntax

The parser will have a more complex addition for this feature as the components of a complex number can be either an integer or a float, and that should be enforced. Despite this, the `expr` data-type again only gains one entry.

```
type 'a expr =
  ...
  | EComplex of 'a expr * 'a expr * 'a
```

Even though the parsing is more narrow than this constructor, having the components of this entry be `expr` standardizes the identification of their integer/float-ness at run-time as their memory representations will follow our tagging scheme.

Examples

Complex can be used to perform arithmetic - both with other complex numbers and with real numbers (integers and floats).

```
begin
  4+3i + 3+4i; # 7+7i
  4+3i - 3+4i; # 1-1i
  4-3i + 3-4i; # 7-7i
  4-3i - 3-4i; # 1+1i

  4+3i + 3; # 7+3i
  4+3i - 3; # 1+3i
  3 + 4+3i; # 7+3i
  3 - 4+3i; # -1+3i

  4+3i + 3.0;     # 7.0+3i
  4+3i - 3.0;     # 1.0+3i
  3.0 + 4+3i;     # 7.0+3i
  3.0 - 4+3i;     # -1.0+3i
  4+3.0i + 3+4i; # 7+7.0i
  4+3i + 3+4.0i; # 7+7.0i

  add1(4+3i);    # 5+3i
  add1(4.0+3i); # 5.0+3i
  add1(4+3.0i); # 5+3.0i
  sub1(4+3i);    # 3+3i
  sub1(4.0+3i)  # 3.0+3i
  sub1(4+3.0i)  # 3+3.0i
end
```

Representation

Since complex numbers consist of two independent `snakeval`'s, the `snakeval` for a complex number itself has to be a tagged pointer to wherever on the heap its two component numbers are stored.

The updated tag scheme is as follows, where `w` is a "wildcard" bit:

- Integers: `www0`
- Booleans: `1111`
- Tuples: `w001`
- Floats: `w011`
- Complex Numbers: `w101`

As we are nearing the end of the tag-space provided by our heap-aligned pointers, perhaps we will soon need to revisit how we structure and recognize our heap-allocated values.

Modifications to Pre-Compilation Phases

Since complex numbers' internal structure "wraps" terminal expressions they require little special treatment before compilation.

- Raise an `InternalCompilerException` in `is_well_formed` if either of the components of an `EComplex` are not `EInteger` or `EFloat`.
  - This is an `ICE` since this invariant should be enforced by a previous phase (the parser).
- No special desugaring is required.
- In `anf` they can be treated as a non-immediate lettable `cexpr` as tuples and floats are, as they too are heap-allocated.
- They have no local variables so there is no impact on stack allocation.

Semantics

Arithmetic

As seen in the above examples, the rules of complex arithmetic apply over both operations which involve two complex numbers and those which have one complex argument and one real argument. The rules for arithmetic with floats apply to each of the components of a complex number separately.

Comparison

Complex numbers cannot be compared, following the example of languages like Racket. A `comparison expects real numbers` error should be thrown in this case.

Equality

As with floats, pointer equality and structural equality will often differ against realistic expectations due to their representation in memory.

```
4+3i == 4+3i;            # false
let a = 4+3i in a == a;  # true
equal(4+3i, 4+3i)        # true
```

Predicates

- `isNum` should return true for complex numbers as they are still numbers, however complex.

```
isNum(4+3i); # true
```

- The new `isReal` should be false for complex numbers and true for integers and floats.

```
isReal(4.3); # true
isReal(4);   # true
isReal(4+3i) # false
```

- The new `isComplex` should only be true for complex numbers.

```
isComplex(4.3); # false
isComplex(4);   # false
isComplex(4+3i) # true
```

Compilation Changes

As with floats, the bulk of changes for this feature are in the code emission phase in order to handle their creation and use with our existing operations.

Building off what we described for floats, the introduction of complex numbers again increases the amount of run-time type checking needed for our arithmetic and comparison operations (though for comparison it is less bad since we don't care about the components of complex numbers).

As each complex number has two component numbers, both of which can be an integer or float, every number involved in the operation (for both real and complex operands) needs to be checked to determine which "branch" of the operation to execute. These branches will include all combinations of complex vs. real and integer vs. float operands. These branches' code will largely rely on the arithmetic that we already know from integers and floats, and simply construct new complex number data structures for results.