In this assignment, you'll implement a compiler with float included.

1 The Garter Language

As usual, we have concrete and abstract syntax, along with a specification of semantics.

1.1 Concrete Syntax

The major addition to Garter is float.

```
<expr>:
      | let <bindings> in <expr>
      | if <expr>: <expr> else: <expr>
      | <decls> in <expr>
      | <binop-expr>
<binop-expr>:
            | IDENTIFIER
            | NUMBER
            | FLOAT
            | true
            | false
            | !<binop-expr>
            | <prim1>(<expr>)
            | <expr> <prim2> <expr>
            | IDENTIFIER(<exprs>)
            | IDENTIFIER()
            | (<expr>)
<prim1>:
       | add1 | sub1
       | print | isbool | isnum | isfloat
       | cos | sqrt
<prim2>:
       | + | - | * | / | //
       | < | > | <= | >=
       | ==
       | && | ||
<decls>:
       | <decls> and <decl>
       | <decl>
<decl>:
      | def IDENTIFIER(<ids>): <expr>
      | def IDENTIFIER(): <expr>
<ids>:
     | IDENTIFIER
     | IDENTIFIER, <ids>
<exprs>:
       | <expr>
       | <expr>, <exprs>
<bindings>:
          | IDENTIFIER = <expr>
          | IDENTIFIER = <expr>, <bindings>
```

1.2 Abstract Syntax

The abstract syntax is very similar to Diamondback, also:

```
pub struct FloatWrapper(pub f64);
// Implement Eq for FloatWrapper with epsilon comparison
impl PartialEq for FloatWrapper {
    fn eq(&self, other: &Self) -> bool {
        (self.0 - other.0).abs() < f32::EPSILON as f64
    }
}
impl Eq for FloatWrapper {}

#[derive(Copy, Clone, Debug, PartialEq, Eq)]
pub enum Prim {
    ...
    IsFloat,
    Sqrt,
    Cos,
    Div,
    FloorDiv,
}

pub enum Exp<Ann> {
    ...
    Float(FloatWrapper, Ann),
}
```

The struct `FloatWrapper` here is to satisfy the requirement of `Eq` and `PartialEq`. And the immediate value will change to:

```
pub enum ImmExp {
    ...
    Float(FloatWrapper),
}
```

1.3 Semantics and Representations of Floats

1.3.1 How to represent floats in memory?

First of all, we will see any groups of `[0-9]` without `.` as numbers and with `.` or `[eE]` as floats in parser.

We will use `f32` in garter language. But why do we create a `pub struct FloatWrapper(pub f64)` using `f64`?

First, we try to distinguish between numbers and floats in memory. In diamondback, we decide `true` to be `0xFF_FF_FF_FF_FF_FF_FF_FF` and `false` to be `0x7F_FF_FF_FF_FF_FF_FF_FF`. Their last two bits are

`11`. And numbers have `0` in their last one bit. So we define floats as 64-bit numbers with `01` in their last two bits.

How to ensure that floats are `01`? Why do we only support `f32` but use `f64` to represent it? We know that `f64` (called double-precision floating point) has 1 bit of sign bit, 11 bits of exponent bits, and 52 bits of significand precision bits. And `f32` (called single-precision floating point) has 1 bit of sign bit, 8 bits of exponent bits, and 23 bits of significand precision bits. Let's take an example.

```
1.5
```

will be

```
0x3FF8_0000_0000_0000
```

```
0b0(1 bit) 011_1111_1111_(11 bits) 1000_0000_0000_...(52 bits)
```

in `f64`, and

```
0x3FC0_0000
```

```
0b0(1 bit) 011_1111_1(8 bits) 100_0000_0000_...(23 bits)
```

in `f32`. When we use `1.1 as f64 as f32`, it will automatically convert an `f64` binary number to an `f32` binary number by handling the first 12 bits and making sure the last `52 - 23 = 29` bits of significand precision bits are zeros. In detail, starting from 0th bit, rust will add 1 to the last 29th bit if the last 28th bits is 1, and do nothing if it is 0. Therefore, we will simulate it, and we can safely represent `f32` using a 64-bit number with the last two bits being `01` in memory. In `stub.rs`, we write

```
fn u64_to_u8_array(value: u64) -> [u8; 8] {
    let byte0 = ((value >> 56) & 0xFF) as u8;
    let byte1 = ((value >> 48) & 0xFF) as u8;
    let byte2 = ((value >> 40) & 0xFF) as u8;
    let byte3 = ((value >> 32) & 0xFF) as u8;
    let byte4 = ((value >> 24) & 0xFF) as u8;
    let byte5 = ((value >> 16) & 0xFF) as u8;
    let byte6 = ((value >> 8) & 0xFF) as u8;
    let byte7 = (value & 0xFF) as u8;

    [byte0, byte1, byte2, byte3, byte4, byte5, byte6, byte7]
}
```

```
fn sprint_snake_val(x: SnakeVal) -> String {
    if x.0 & TAG_MASK == 0 {
        // it's a number
        format!("{}", unsigned_to_signed(x.0) >> 1)
    } else if x == SNAKE_TRU {
        String::from("true")
    } else if x == SNAKE_FLS {
        String::from("false")
    } else if x.0 & 3 == 1 {
        // it's a float
        format!("{}", f64::from_be_bytes(u64_to_u8_array(x.0 - 1)) as f32)
    } else {
        format!("error: cannot print {}", x.0)
    }
}
```

to print the floats. Also, if a `f64` number, which doesn't have all zeros in the last 29 bits, is stored in memory, it will lose precision, like `2.1` is `0x4000_cccc_cccc_cccd` in `f64` and `0x4000_cccc_c000_0000` in our memory (not `0x4006_6666` in normal `f32`). And `1.1` (`0x3ff1_9999_9999_999a` in `f64`) will be `0x3ff1_9999_a000_0000` in our memory. Therefore, if we add `0.1` and `0.2` many times in garter, it won't be equal to the normal result. It's a limitation of float. In conclusion, a `f32` will be

```
0b0(1 bit) 000_bbbb_bbbb_(11 bits) bbbb_bbbb_...bbb(23 bits)
0_0000_0000_...(29 bits)
```

in our memory. So it's safe to use the last two bits as "float-type" bits. The first 3 bits in the 11 exponent bits are always 0 because any value larger than `f32::MAX` (or smaller than `f32::MIN`) will report an `overflow` bug.

1.3.2 How to calculate on floats?

In `x87`, we can use FPUs to handle floating point arithmetic. The FPU has eight registers called `st0` to `st7` and an array of those eight registers which is a stack. `st0` refers to the register that is at the top of the stack. Numbers can be loaded onto the stack from memory and stored in memory. So we add the following regs and FPU instructions,

```
pub enum Reg {
    ...
    St0,
    St1,
    St2,
    St3,
    St4,
    St5,
    St6,
```

```
    St7,
    Ax
}

pub enum FloatMem {
    RegMem(MemRef),
    VarMem(String)
}

pub enum FloatArg {
    ToReg(Reg, Arg64),
    Mem(FloatMem),
    Reg(Reg),
    Blank
}

pub enum Instr {
    ...
    Fld(FloatMem),
    Fild(FloatMem),
    Fstp(FloatMem),
    Fistp(FloatMem),
    Fadd(FloatArg),
    Faddp(FloatArg),
    Fsub(FloatArg),
    Fsubp(FloatArg),
    Fmul(FloatArg),
    Fmulp(FloatArg),
    Fdiv(FloatArg),
    Fdivp(FloatArg),
    Fstsw(FloatArg),
    Fcom(FloatArg),
    Fcomp(FloatArg),
    Fcompp(FloatArg),
    Fabs,
    Fld1,
    Fcos,
    Fsqrt
}
```

To compare, we can first use `Instr::Fcom` and use `Instr::Fstsw` to store the FPU status bits into `Reg::Ax`. Then we check `C0` and `C3` bits to jump.

For arithmetic operations, let's take an example. `1.5 + 2.1` will be

```
mov rax, 0x3ff8000000000000
mov qword [rsp + -8], rax
fld qword [rsp + -8]
mov rax, 0x4000cccccc0000000
mov qword [rsp + -8], rax
fld qword [rsp + -8]
```

```
faddp
fstp [rsp + -8]
mov rax, qword [rsp + -8]
ret
```

If all of the expressions of an operation are numbers, we will use the semantics in diamondback. However, if one of the expressions is float, we will also load numbers to FPU. To handle numbers together with floats, we can use `fild` and `fistp` to load onto and store from FPU stack. For example, `11 // 3` will be

```
mov rax, 11
mov qword [rsp + -8], rax
fld qword [rsp + -8]
mov rax, 3
mov qword [rsp + -8], rax
fld qword [rsp + -8]
fdivp
fistp [rsp + -8]
mov rax, qword [rsp + -8]
ret
```

These are just examples, not what we will do in our garter.

1.3.3 What about overflow and underflow?

In 1.3.1 we mentioned that floats larger than `f32::MAX` (which is `3.4028235e38`) or smaller than `f32::MIN` (which is `-3.4028235e38`) will report an `overflow` error. How to realize it? First, at compile time before the program runs, check whether a float constant is overflow. Second, at runtime, whenever an arithmetic operation happens in FPU stack, check whether the result is overflow. For example,

```
3.3e100
```

```
3.3e30 * 3.3e30
```

will both report an `overflow` error.

We know that any `f32` between 0 and `f32::MIN_POSITIVE` (which is `1.1754944e-38`) will cause underflow. However, we assume two `f32` are equal by letting `(value1 - value2).abs() < f32::EPSILON` in garter (`f32::EPSILON` is `1.1920929e-7`). In this case, any float between 0 and `f32::MIN_POSITIVE` is equal to 0, so we don't need to consider underflow errors.

1.3.4 Errors

There are a number of new errors that can occur now. Your implementation should catch all of these cases statically; that is, at compile time before the program runs:

- If a float constant is larger than the range of `f32`, report an `overflow` error
- If a numeric constant is larger than the range of number, report an `overflow` error.

You should raise, at runtime, all errors in diamondback plus:

- `-`, `+`, `*`, `/`, `//`, `add1`, `sub1`, `sqrt` and `cos` should raise an error (by printing it out) with the substring `"arithmetic expected a number or float"` if the operation's argument(s) are not numbers or floats.
- `<`, `<=`, `>`, `>=`, `==` and `!=` should raise an error with the substring `"comparison expected a number or float"` if the arguments are not both numbers or floats.
- `/`, `//` should raise an error with the substring `"division by zero"` if divides zero.
- `+`, `-`, `*`, `/`, `//`, `add1`, `sub1`, `sqrt` and `cos` should raise an error with the substring `"overflow"` if the result overflows number range and both expressions are numbers. They also should raise an error with the substring `"overflow"` if the result overflows the float range and one of the expressions is float.
- `sqrt` should raise an error with the substring `"sqrt expected a non-negative value"` if the argument is negative.