CS160 Assignment 4: Code Generation

Your compiler will generate x86 assembly code for Patina programs.

For this assignment, you will tackling the most central part of a compiler: the generation of code of a lower level language. In this case, we will be using x86 as the target language. Cheat sheet #1 Cheat sheet #2

This would normally be far more difficult than each of the other individual stages of the compiler. However, we have introduced a number of simplifications to make it doable.

1. Function calls are disallowed, and any functions besides main. In essence, the patina files to be compiled will represent a simple script. (Functions will be included in bonus part #1.)

2. Arrays are no longer included in the language specification. This allows all data to be stored on the stack as supposed to on the heap. (Arrays will be included in bonus part #2.)

3. This matters for performance but not to correctness, no register allocation algorithm needs to be written. All data will be stored inside a single frame of the stack.

However, this still leaves a fair amount of complexity. For example, the following function calculating fibonacci numbers is still allowed.

```
fn main() -> unit {
    let fibn : int = 10;
    let firstval : int = 1;
    let secondval : int = 1;
    if (fibn < 0) then
        {let res : int = 0/0} else
    {
        while (fibn > 0){
            let sum : int = firstval + secondval;
            firstval = secondval;
            secondval = sum;
            fibn = fibn - 1
        };
        let res : int = secondval
    }
}
```

As you can see, this still contains the challenges for implementation of

1. jumps required for while loops and if expressions
2. binop expressions which when compiled require multiple lines of assembly.

For the former, you will be adding labels into the assembly to allow appropriate jumps. For example, the expression {if 0 then {2} else {1} + 1} should be translated into something equivalent to the following (ignore the binop compilation there for now).

```
    cmp $0 $1
    je L1
    mov $1 %rax
    jmp L2
    L1:
    mov $2 %rax
    L2:
    add %rax $1
```

For the latter, you will be reading variables from the stack into some fixed registers, pushing them onto the stack temporarily to allow for arbitrary depth arithmetic trees, and then writing them back there. For example, the expression x = x * 4 should be translated into something equivalent to the following (which is inefficient, but you only need correctness for now.)

```
    mov 16(%rbp) %rax
    push %rax
    mov $4 %rax
    mov %rax %rbx
    pop %rax
    add %rbx %rax
    mov %rax 16(%rbp)
```