Compiling L1 to x86-64

High-level overview:

- Generate some small number of x86-64 instructions for each L2 instruction, save in **`prog.S`** file, generating calls into C-defined runtime system to implement **`print`**, **`allocate`**, and **`array-error`**
- Compile **`prog.S`** like this:

```
as -o prog.o prog.S
```

- Compile the runtime system like this:

```
gcc -O2 -c -g -o runtime.o runtime.c
```

- Combine them into an executable like this:

```
gcc -o a.out prog.o runtime.o
```

Use linux to avoid Mac OS X stack alignment issues

Compiling the main function; generate this code:

```
        .text
        .globl go
go:
        # save callee-saved registers
        pushq   %rbx
        pushq   %rbp
        pushq   %r12
        pushq   %r13
        pushq   %r14
        pushq   %r15

        call «main label»

        # restore callee-saved registers and return
        popq   %r15
        popq   %r14
        popq   %r13
        popq   %r12
        popq   %rbp
        popq   %rbx
        retq
```

It matches **`runtime.c`**'s **`main()`**, which calls **`go()`**

Compiling simple assignments: prefix registers with **%** and constants and labels with **$**; note the destination is on the right

```
(rax <- 1)      ⇒   movq $1, %rax

(rax <- rbx)    ⇒   movq %rbx, %rax

(rax <- :f)     ⇒   movq $_f, %rax
```

For memory references, put parens around the register and prefix it with the offset

```
((mem rsp 0) <- rdi)
```

⇒

```
movq %rdi, 0(%rsp)
```

```
(rdi <- (mem rsp 8))
```

⇒

```
movq 8(%rsp), %rdi
```

Each of the **`aop=`** operations correspond to their own assembly instruction

`(rdi += rax)` $\Rightarrow$ **`addq %rax, %rdi`**

`(rdi -= rax)` $\Rightarrow$ **`subq %rax, %rdi`**

`(r10 *= r12)` $\Rightarrow$ **`imulq %r12, %r10`**

`(r14 &= r15)` $\Rightarrow$ **`andq %r15, %r14`**

Saving the result of a comparison requires a few extra instructions

```
                              cmpq %rbx, %rax
(rdi <- rax <= rbx)    ⇒     setle %dil
                              movzbq  %dil, %rdi
```

the **`cmpq`** instruction updates a condition code in some hidden place and then we need to use **`setle`** to extract the condition code from the hidden place. The **`setle`** instruction, however, needs an 8 bit register as its destination. So we use **`%dil`** here because that's an 8 bit register that overlaps with the lowest 8 bits of **`%rdi`**. That updates only those 8 bits, however so we need **`movzbq`** to zero out the rest

Saving the result of a comparison requires a few extra instructions

```
                               cmpq %rbx, %rax
(rdi <- rax <= rbx)   ⇒        setle %dil
                               movzbq  %dil, %rdi
```

Here's the table mapping regular register names to their 8-bit variants

```
r10 → r10b    r11 → r11b    r12 → r12b
r13 → r13b    r14 → r14b    r15 → r15b
r8  → r8b     r9  → r9b     rax → al
rbp → bpl     rbx → bl      rcx → cl
rdi → dil     rdx → dl      rsi → sil
```

Saving the result of a comparison requires a few extra instructions

```
                                  cmpq %rbx, %rax
(rdi <- rax <= rbx)   ⇒   setle %dil
                                  movzbq  %dil, %rdi
```

And if we had **<** we'd need to use **`setg`** or **`setl`** (for less than or greater than) and if we had **=** then we would use **`sete`**

The shifting, **`sop=`**, operations also use the 8-bit registers, this time for their sources

```
(rdi <<= rcx)   ⇒   salq %cl, %rdi

(rdi >>= 3)     ⇒   sarq $3, %rdi
```

The **`l`** is for "left shift" and the **`r`** stands for "right shift".

The same three instructions also work great when there is a constant on the left

```
(rdi <- rax <= 10)   ⇒   cmpq $10, %rax
                         setle %dil
                         movzbq  %dil, %rdi
```

But when the constant is on the right, we need to flip things around

```
(rdi <- 10 <= rax)  ⇒   cmpq $10, %rax
                        setge %dil
                        movzbq  %dil, %rdi
```

Why? Because **`cmpq`** needs a register "destination" for reasons that make little sense to me

So when we don't have any registers at all, we need to compute the answer at compile time and just use that

```
(rdi <- 10 <= 11)   ⇒   movq $1, %rdi

(rax <- 12 <= 11)   ⇒   movq $0, %rax
```

Labels and gotos are what you might guess; just replace the leading colon with an underscore and add a colon suffix when you define the label

```
:a_label            ⇒  _a_label:
(goto :a_label)     ⇒  jmp _a_label
```

For conditional jumps, we have the three same cases as we did for conditional comparisons, but we use two jumps instead of storing the result in a register

```
(cjump rax <= rdi :yes :no)
```

⇒

```
cmpq %rdi, %rax
jle _yes
jmp _no
```

For less than or equal to, **<=**, use `jge` (jump greater than or equal) or `jle` (jump less than or equal). For strictly less than, **<**, use `jg` (jump greater than) or `jl` (jump less than) and for equality, **=**, use `je`

Finally, compiling the instructions that modify **`rsp`**:

- Function header (entry to a function)
- The **`call`**, **`tail-call`**, and **`return`** instructions

Allocating local storage is the function header's job; for each stack variable, push 8 bytes, e.g.

```
(:myfunction 0 3 ....stuff...)
```

⇒

```
_myfunction:
   subq $24, %rsp  # allocate spill
...compiled stuff...
```

The **(return)** instruction frees local storage, pops the return address from the stack and jumps to it, e.g.,

```
(:myfunction 0 3 (return))
```

⇒

```
_myfunction:
   subq $24, %rsp  # allocate spill
   addq $24, %rsp  # free spill & args
   ret
```

The **(call)** instruction moves **rsp** based on the number of arguments and the return address and then jumps to the new function, e.g.

```
(call :anotherfunction 11)
```

⇒

```
subq $48, %rsp  # call L1 function
jmp _anotherfunction
```

Argument storage allocation is **(* (- 11 6) 8)** = **40** bytes, plus 8 more to move past the return address

The **`(call)`** also calls funtions defined in **`runtime.c`**. In that case, we can just use the **`call`** assembly instruction.

```
(call array-error 2)
```

⇒

```
call array_error  # runtime system call
```

The **(tail-call)** instruction moves **rsp** back to free the local storage and then jumps, e.g.

```
(:f 11 3 (tail-call :g 5))
```

⇒

```
_f:
   subq $24, %rsp  # allocate spill
   addq $64, %rsp  # free spill & args
   jmp _g  # tail call
```

Free **(* (- 11 6) 8)** = **40** bytes for **:f**'s args, plus **24** more for **:f**'s spill. Functions can only be called in tail position when they have six or fewer args, so we don't have to move the arguments around on the stack (since there aren't any).