1 Optimization

In this section, we'll talk more about optimization.

Definition 1.1: Optimization

An **optimization** (for a compiler) is a version that produces programs that evaluate the same answer as the prior version, but are "better" on some cost metric.

1.1 Examples of Cost Metrics

Some examples of cost metrics that we might want to improve on include

- **Time:** How long does it take for the program to run?
- **Space:** How much process memory does the program use?
- **Binary Size:** The size of the compiled binary, and the number of instructions.
- **Executed Instructions:** The overall number of instructions executed (compared to the total number of instructions). This is also similar to the number of jumps in the resulting assembly.

Remarks:

- The first two metrics – time and space – are generally the most important ones.
- We might also care about properties like compile time, extensibility, debuggability, and platform independence, although these are harder to measure.

1.2 High-Level Optimization Suggestions

Some suggestions for optimizations include

- **Register Allocation:** Storing values in registers rather than in memory, since access to registers are generaly faster than access to memory.
- **Dead Code Elimination:** Remove code that the compiler can prove will never run.

  (Example.) Consider the following code:

  ```
  if false 3 4
  ```

  Here, we know for sure the `3` will never execute.

  This also includes things like removing unused variables from compilation.
- **Constant Folding:** Evaluate "what you can solve" in the compiler.

  (Example.) Consider the following code:

  ```
  (+ (* 2 3) input)
  ```

  Here, we know that `(* 2 3)` should be `6`, so this is basically equivalent to

  ```
  (+ 6 input)
  ```
- **Common Subexpression Elimination:** Eliminate repeated code in favor of a single instance of it.

- **Memory Packing:** Eliminate unused memory due to alignment (e.g., struct alignment).
- **Loop Unrolling:** We can unroll a loop if we know that it has a constant bound. In other words, essentially hardcode all iterations.

(Example.) Consider the following code:

```
(let (x 0) (loop (if (< x 3) (set! x (add1 x)) (break x))))
```

This is equivalent to

```
(let (x 0) (block
    (set! x (add1 x))
    (set! x (add1 x))
    (set! x (add1 x))
))
```

- **Type-Directed Compilation:** We can remove some code that involves type checking if we know for sure that we're working with the correct types.

(Example.) Consider the code

```
(+ 1 (* input 2))
```

While type checking is necessary for `(* input 2)`, it's probably not necessary for the plus expression since we can assume that both sides are numbers.

- **Peephole Optimization:** We can remove redundent move operations in the resulting assembly.
- **Inlining Functions:** We can inline function calls, especially if we have a small one.

(Example.) Consider the following function and resulting code:

```
(fun (f x)
    <body>)
(f 10)
```

This could be functionally equivalent to

```
(let (x 10) <body>)
```

Note that we might need to consider things like recursion or other function calls in the function body, since that might prevent us for optimizing.

- **Instruction Selection:** We can also possibly exploit the structure of our code.

(Example.) Consider the following code:

```
(if (< x 10) ... ...)
```

Generally, our compiler would put either `true` or `false` into `rax`, and then evaluate `rax` when deciding where to jump. However, in this particular code, we can probably just conditionally jump on the spot.