Intermediate Language

- A language between the source and the target
- Provides an intermediate level of abstraction
    - More details than the source
    - Fewer details than the target
- Provides an intermediate level of abstraction
    - e.g., the source language has no notion of registers, the IR may have a notion of registers.
    - Why are we using it? Just experience. Some compilers, in fact, choose to have multiple IRs, and thus multiple lowerings.
        - Intuition: make some decisions upfront (source to IR); hopefully these decisions do not make much difference to optimization opportunity, but simplifies reasoning about the final code generation (as the abstraction level is much closer to the target).
        - An IR would usually always loose some information (information that the compiler developer considers extraneous to optimization opportunity). e.g., loop structure converted to gotos. Ideally the abstraction lowering should not loose much information (should not preclude optimization opportunity!).

The design of an IR is an *art* --- hard to say that this is the best possible IR which will allow the best code generation/optimization. We will consider IR which resembles high-level assembly

- Uses register names, but has an unlimited number
- Uses control structures like assembly language
- Uses opcodes but some are higher-level
    - e.g., `push` translates to several assembly instructions
    - Most opcodes correspond directly to assembly opcodes

Each instruction is of the form

```
x = y op z
x = op y
```

- `y` and `z` are registers and constants
- Common form of IR
- This particular IR is also called *three-address code*

The expression `x + y*z` is translated

- t1 = y * z
- t2 = x + t1

In this representation, each subexpression has a "name" : an effect of allowing only one expression at a time.

IR code generation is very similar to assembly code generation. But use any number or IR registers to hold intermediate results.

```
igen(e, t)
```

- code to compute the value of expression `e` in register `t`.

Example:

```
igen(e1+e2, t) =
  igen(e1, t1)  //t1 is a fresh register
  igen(e2, t2)  //t2 is a fresh register
  t = t1 + t2
```

Unlimited number of registers, means IR code generation is simple. Contrast with stack machine, where we were using stack slots to save intermediate results (many instructions to save/restore); here we can just coin a new register name, and save results to it.

LLVM IR is an example of IR. It resembles three-address code, but with usually higher-level opcodes than assembly.

- IR designer's dilemma example: how high-level should the opcodes be? Too low-level may preclude optimization opportunity (assembly opcodes may be higher-level than IR opcodes, e.g., vector instructions). High-level IR opcodes make the IR design large and bulky (starts looking almost like a CISC ISA). Big problem with IR: needs to be designed for all possible ISAs (makes the design decisions even harder). Typical design choice: support as many opcodes as necessary for all the common optimizations on all the common ISAs.
- The same IR may be used for multiple high-level programming languages. e.g., LLVM may be the target for both C and Java programs. Can again increase the complexity of LLVM IR, because we need to try and retain high-level semantics of all supported languages (making them too low-level for simplicity would preclude optimization).
- Yet, if one can design an IR successfully, it is all perhaps worth the effort. Do all the hard work related to optimization once and reap the benefits everywhere (for all languages, and for all ISAs).