Intermediate Representation

why an intermediate representation?

- Want to represent code in a form that is:
  - Closer to assembly than ASTs — low level operations, branches, memory operations
  - Not machine specific — no registers, instructions more “abstract” than machine instructions
- Makes it easier to perform certain kinds of optimizations

```
LA T1 <address of x>
LI T2, 10
SW T2, 0(T1)
LW T3, 0(T1)
ADDI T4, T3, 20
SW T4, 0(T1)
```

becomes

```
MV 10, $GX //X = 10
ADD $GX, 20, $GX //X = 20 + X
```

three-address code

- All operations take at most three operands: two source operands, one destination operand
- *Almost* the same as Risc-V assembly, except:
  - No registers, only temporaries
  - Operands can be literals, temporaries, or variables
    - Loads and stores are implicit
    - Encode address information in operand names for easy translation later
      - Temporaries: $Tx
      - Globals: $G<name>
      - Locals: $L<stack offset>

converting 3ac into assembly

- Simple approach: **macro expansion**
- Treat each 3AC instruction separately, generate code in isolation

```
ADD $GC, $GA, $GB
```

```
LA r1 <addr of A>
LW r2, 0(r1)
LA r3 <addr of B>
LW r4, 0(r3)
ADD r5, r3, r4
LA r6 <addr of C>
SW r5, 0(r6)
```

instruction selection

- Can be clever about how we turn 3AC into assembly by selecting appropriate assembly instructions
  - If one source operand is a literal, generate an immediate instruction
  - If source operand is a local variable, generate a load with an offset, rather than an address computation and a load

converting 3ac into assembly

- Generating better code:

```
ADD $GC, $LP4, $LL8
```

```
LW r1, 4(fp) //paramter at +4
LW r2, -8(fp) //local at -8
ADD r3, r1, r2
LA r4 <addr of C> //global C
SW r3, 0(r4)
```

next: simple optimizations