# Intermediate Code

See Chapter 5, especially Section 5.3

We are using parse trees as intermediate representations for BLP programs. There are other options:

- There are other graph-based representations
- There are linear representations, such as *3-address code*. These often simulate assembly languages.
- Interpreted languages often use a higher-level intermediate language. Java byte code, for example, looks closer to a programming language than an assembly language.

# Here is an example of 3-address code

| **0** | **id** | **z** | |
|---|---|---|---|
| 1 | id | x | |
| 2 | id | y | |
| 3 | id | p | |
| 4 | int constant | | 2 |
| 5 | int constant | | 3 |
| 6 | * | (4) | (1) |
| 7 | * | (5) | (2) |
| 8 | + | (6) | (7) |
| 9 | * | (3) | (8) |
| 10 | = | (0) | (7) |

2*x

3*y

2*x+3*y

p*(2*x+3*y)

z = 3*y

Goals for intermediate representations:

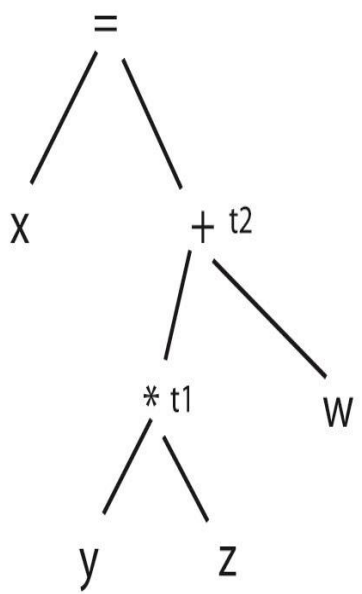
- Simple structure, so final code generation is easy and optimizations are possible.
- Make few assumptions about the target machine so the code is portable.
- If the intermediate representation is executable by a simulator, decoupling of the front and back ends is easy.

Here are some typical 3-address statements:

- arithmetic expressions with binary ops
- assignments
- goto Label
- if (condition) goto L
- arg x (push x onto the stack in preparation for a call)
- call f n  (call f with n arguments)
- return y

3-address code is often implemented with either *quads* or *triples*.

Example: x = y*z+w




Quads

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

Triples

| * | y | z | |
|---|---|---|---|
| + | (0) | w | (1) |
| = | x | (1) | (2) |

Quads are easy to rearrange for optimization, but the symbol table becomes large because of the temporaries.

Triples avoid the temporaries but are hard to move around because that would require finding and changing indices.

So what are the advantages of 3-address code, or similar intermediate representations over the tree-based representation we use? What are the disadvantages?