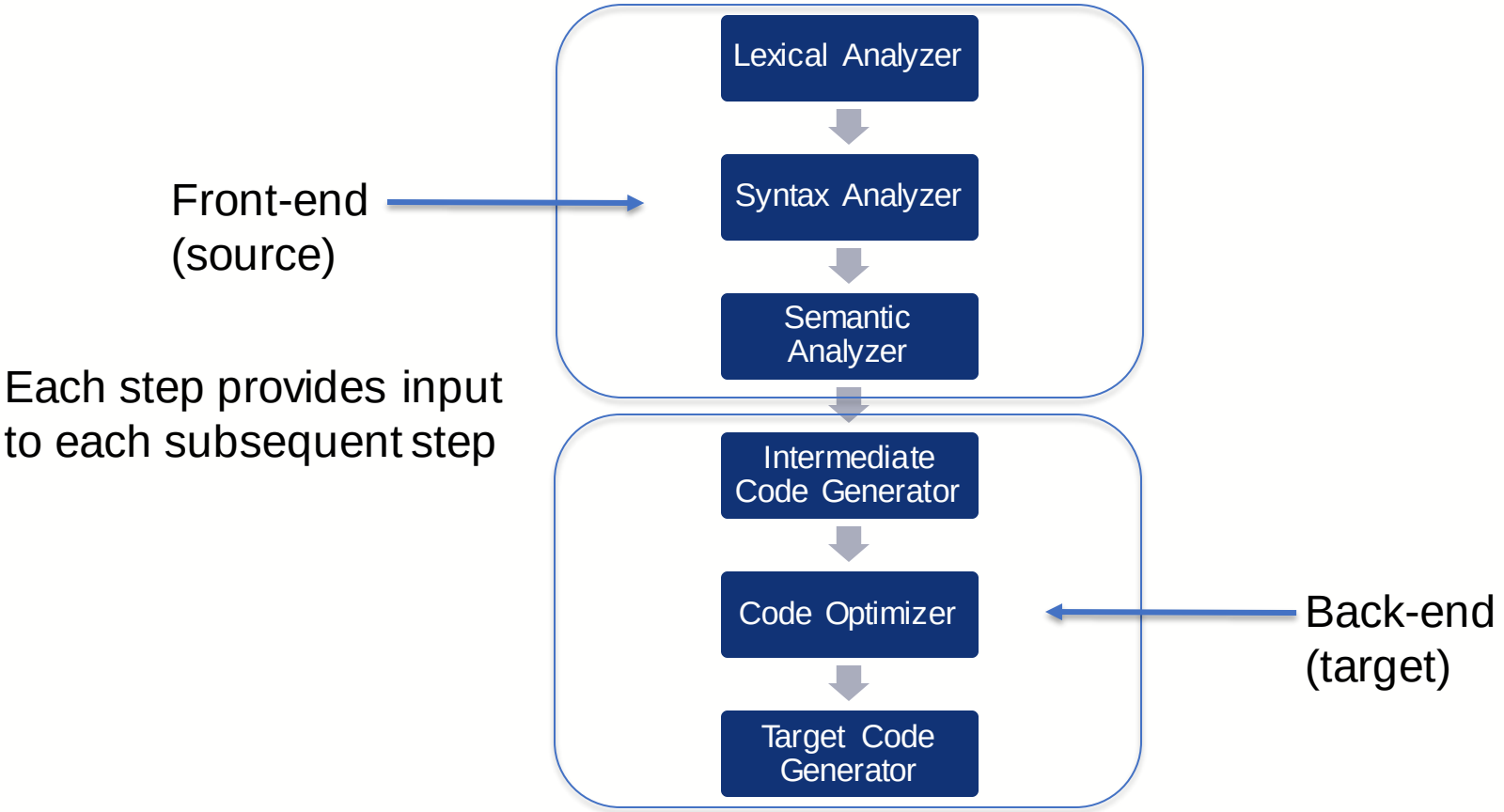
Johns Hopkins Engineering

Module 10: Compilers

EN605.204: Computer Organization

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

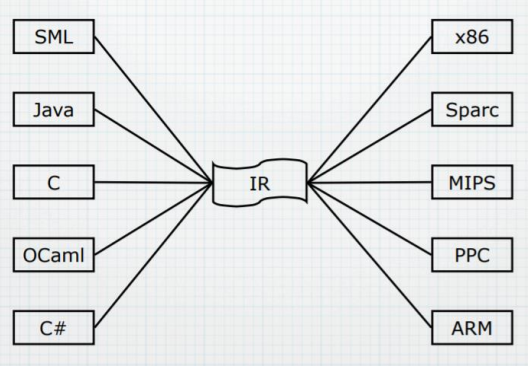
Compiler Phases



Each step provides input to each subsequent step

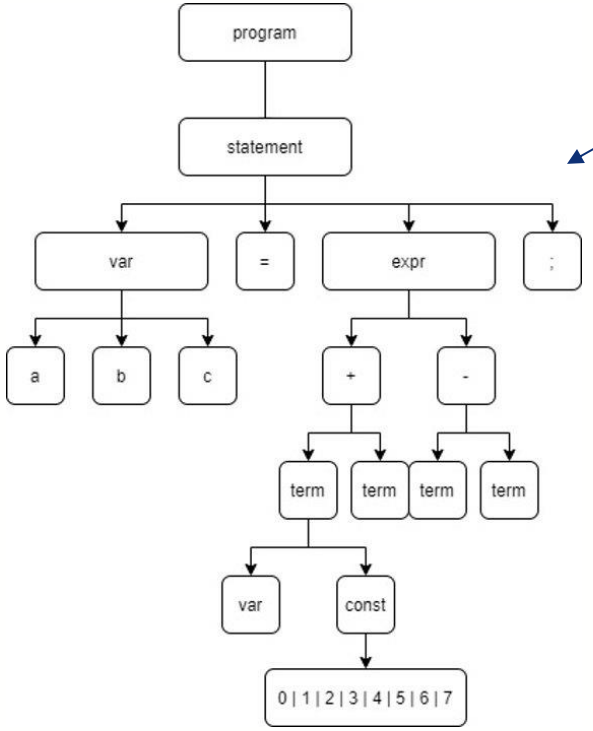
Why Intermediate Code Generation?

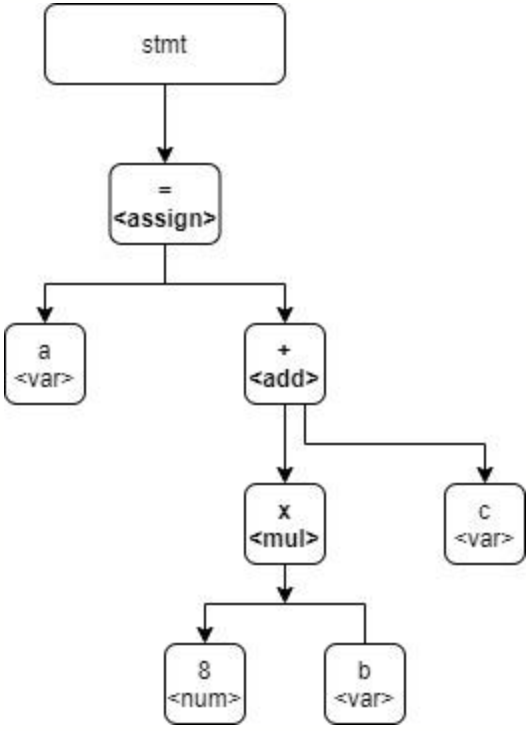
- Intermediate code will be sent to an optimizer
- If there is no intermediate representation, would need a new optimizer for every target
- "Retargeting": swap the back-end components of a compiler
  - Can swap front-end too
- Allows for machine-independent optimization
- Note: there are high- and low-level IR's
  - High-level: "structural" = graph or tree
  - Low-level: "linear" = three-address codes



Parse Tree vs. Abstract Syntax Tree

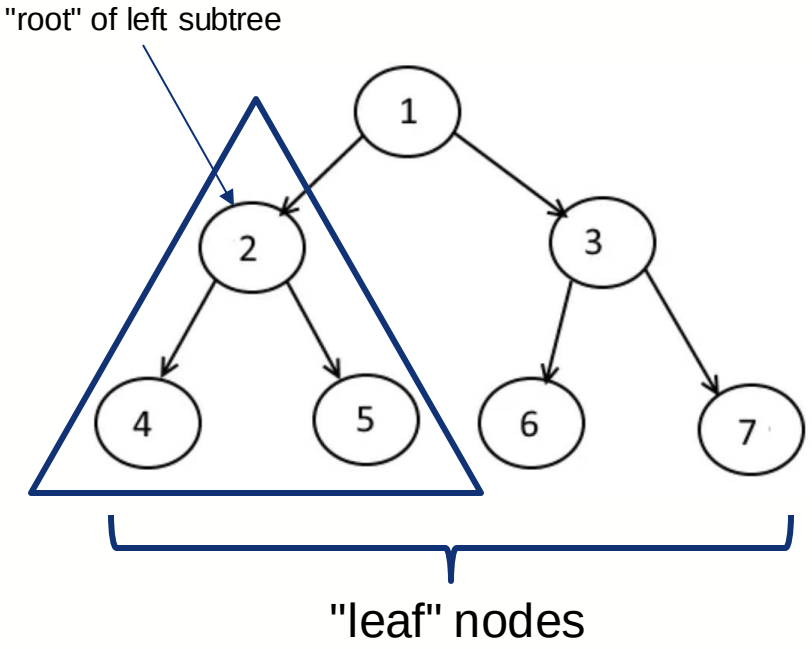
Parse Tree





Abstract Syntax Tree
(simplified parse tree)

In-order Traversal



Called in-order because the result provides the nodes "in-order" from left-to-right

1. Recursively visit left subtree until you reach a leaf, add node to set
2. Back up one level and add the "root" of the subtree to the set
3. Recursively visit the right subtree until you reach a "leaf" node, add node to set

Result: 4, 2, 5, 1, 6, 3, 7

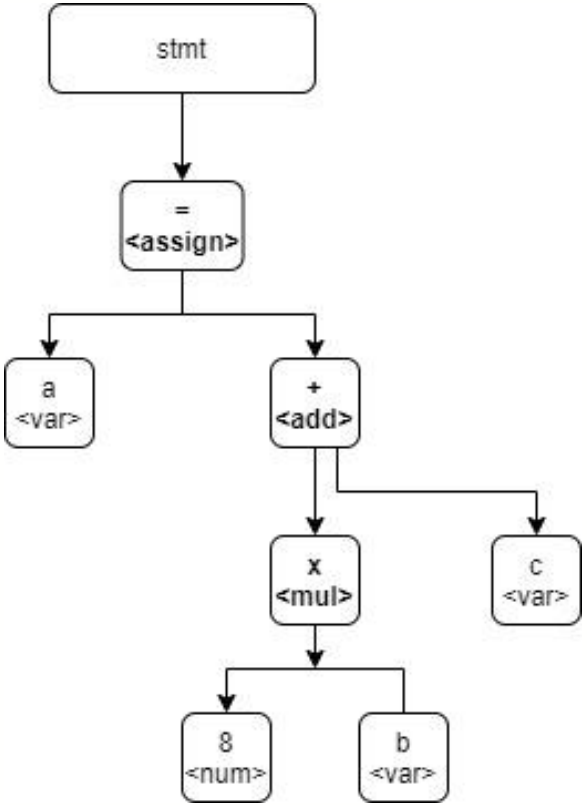
Three-address Codes

```
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

- The output of the intermediate code generator
- A simple format for representing intermediate code
- Has at most 3 operands and usually contains an assignment operator and a binary operator
- Simple format that aids in compiler optimization
- Shown are triples for calculating the quadratic formula:

```
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

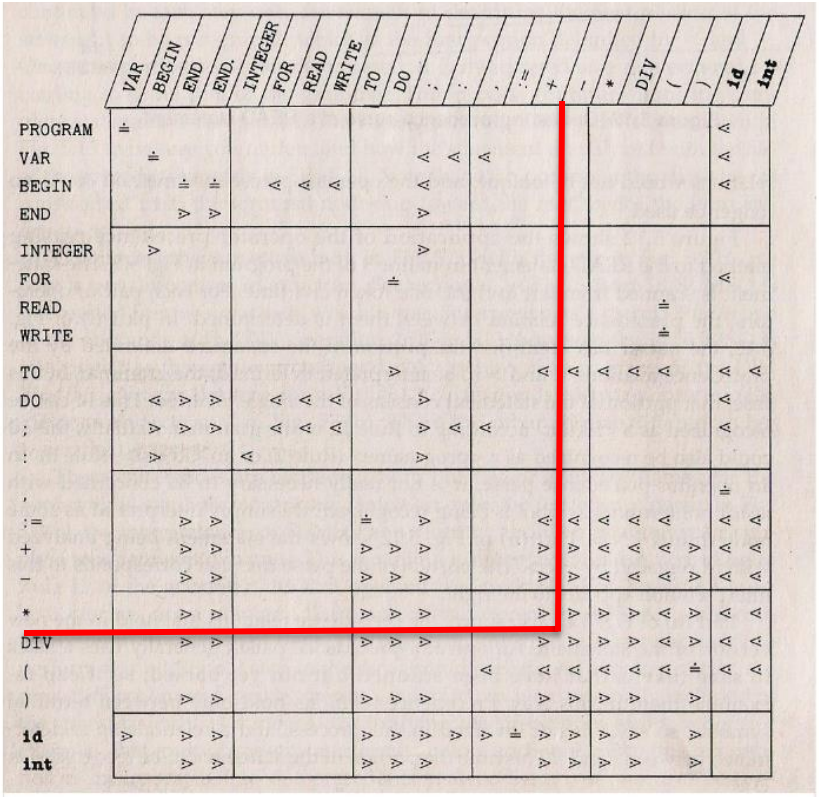
"Linearize" an Abstract Syntax Tree




```
a = 8b + c;
```

In-order traversal: a, =, 8, *, b, +, c

Operator Precedence



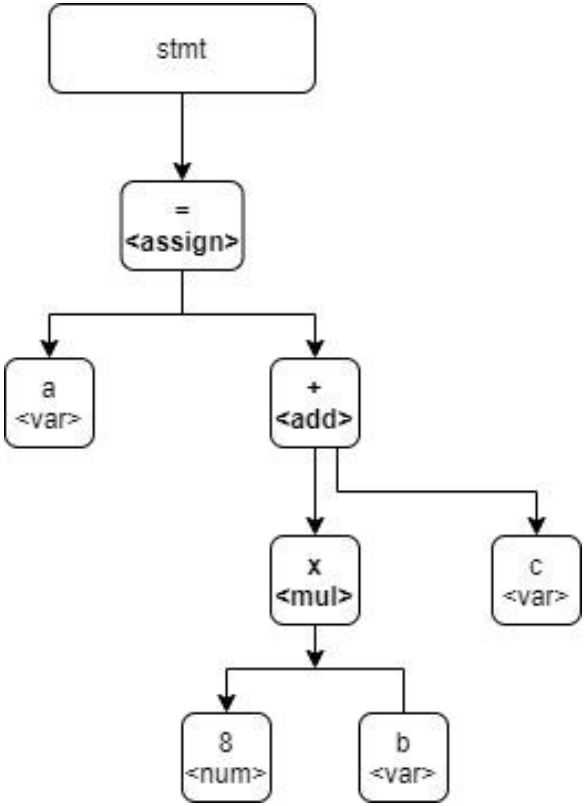

```
a = 8b + c;
```

In-order traversal: a, =, 8, *, b, +, c

Apply "precedence": a, =, 8, *, b, +, c

The left facing operator means "takes precedence over", so multiplication takes precedence over addition

Construct 3-Address Codes




```
a = 8b + c;
```

In-order traversal: a, =, 8, *, b, +, c

Apply "precedence": a, =, 8, *, b, +, c

Generate "three address codes":

t1 = 8 * b
t2 = t1 + c
a = t2

Intermediate Code Optimization

- Code optimization is performed on the resulting intermediate representation
- Can be very simple or very clever
- Simple example, remove unnecessary or redundant code

```
int i = 5 + 5;
int j = 20;
int k = 30;

i = i + 0;

printf("i: %d, j: %d", i, j);
```

Intermediate Code Optimization

- Code optimization is performed on the resulting intermediate representation
- Can be very simple or very clever
- Simple example, remove unnecessary or redundant code

```
int i = 10;
int j = 20;
int k = 30;

i = i + 0;

printf("i: %d, j: %d", i, j);
```

"Constant folding" - evaluate expression once at compile-time rather than run-time

Unused, redundant

Intermediate Code Optimization

- "loop jamming/fusion": if loops iterate the same number of times or use the same looping condition, combine them (below)
- Reduces number of loops (faster), reduces size of binary (smaller)

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
  a[i] = 1;
for (i = 0; i < 100; i++)
  b[i] = 2;
```

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
  a[i] = 1;
  b[i] = 2;
}
```

Intermediate Code Optimization

- "loop unrolling": to avoid overhead of a jump and condition checking, pre-produce the loop instructions (faster, but resulting binary is larger)

```
int x;
 for (x = 0; x < 100; x++)
 {
    delete(x);
 }
```

```
int x;
 for (x = 0; x < 100; x += 5 )
 {
     delete(x);
     delete(x + 1);
     delete(x + 2);
     delete(x + 3);
     delete(x + 4);
 }
```

Target Code Generator: Instruction Selection

- Recall from slide 1 the IR can be "structural" or "linear", often both
  - "structural" representation can be rebuilt from "linear" if needed
- Instructions are generated when a "template" matches a portion of a tree, this is known as "tiling"
  - Ex: template MUL(X, Y) will match the mult. operation from the AST on slide 9