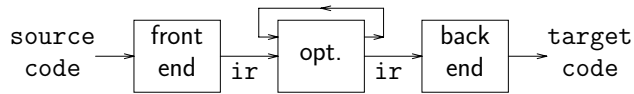


## Intermediate representations

---



**front end** produces an intermediate representation (*IR*) for the program.

**optimizer** transforms the code in IR form into an equivalent program that may run more efficiently.

**back end** transforms the code in IR form into native code for the target machine

The IR encodes knowledge that the compiler has derived about the source program.

## Intermediate representations

---

Broadly speaking, IRs fall into three categories:

### Structural

- structural IRs are graphically oriented
- examples: trees, directed acyclic graphs
- heavily used in source to source translators
- nodes, edges tend to be large

### Linear

- pseudo-code for some abstract machine
- large variation in level of abstraction
- simple, compact data structures
- easier to rearrange

### Hybrids

- combination of graphs and linear code
- attempt to take best of each
- examples: control-flow graph

## Intermediate representations

---

### Advantages

- compiler can make multiple passes over program
- break the compiler into manageable pieces
- support multiple languages and architectures using multiple front & back ends
- enables machine-independent optimization

### Desirable properties

- easy & inexpensive to generate and manipulate
- contains sufficient information

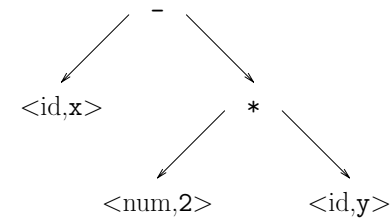
### Examples

- abstract syntax tree (AST)
- directed acyclic graph (DAG)
- control flow graph (CFG)
- three address code
- stack code

## Abstract syntax tree

---

An abstract syntax tree (AST) is the procedure's parse tree with the nodes for most non-terminal symbols removed.



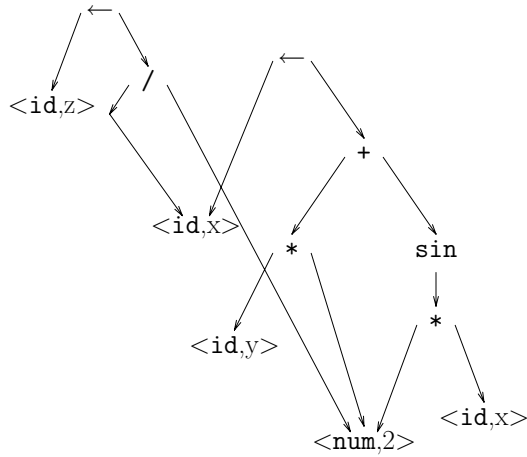
This represents “ $x - 2 * y$ ”.

For ease of manipulation, can use a linearized (operator) form of the tree.

$x$   $2$   $y$   $*$   $-$  in postfix form.

## Directed acyclic graph

A directed acyclic graph (DAG) is an AST with a unique node for each value.



```
x ← 2 * y + sin(2*x)
z ← x / 2
```

## Three address code

Three address code generally allow statements of the form:

$$x \leftarrow y \text{ op } z$$

with a single operator and, at most, three names.

Complex expressions like

$$x - 2 * y$$

are simplified to

$$t1 \leftarrow 2 * y$$
$$t2 \leftarrow x - t1$$

Advantages

- compact form (direct naming)
- names for intermediate values

Register transfer language (RTL)

- only load/store instructions access memory
- all other operands are registers
- version of three address code for RISC

## Control flow graph

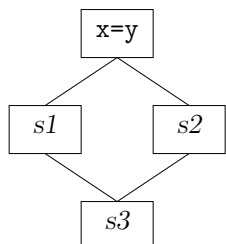
The control flow graph (CFG) models the transfers of control in the procedure.

- nodes in the graph are *basic blocks*  
maximal-length straight-line blocks of code
- edges in the graph represent control flow  
loops, if-then-else, case, goto

Example

```
if (x=y)
  then s1
  else s2
s3
```

becomes



## Three address code

Typical statement types

1. assignments —  $x \leftarrow y \text{ op } z$
2. assignments —  $x \leftarrow \text{op } y$
3. assignments —  $x \leftarrow y[i]$
4. assignments —  $x \leftarrow y$
5. branches — **goto** L
6. conditional branches — **if**  $x \text{ relop } y$  **goto** L
7. procedure calls — **param** x and **call** p
8. address and pointer assignments

Can represent three address code using *quadruples*

x - 2 * y				
(1)	load	t1	y	
(2)	loadi	t2	2	
(3)	mult	t3	t2	t1
(4)	load	t4	x	
(5)	sub	t5	t4	t3

## Stack machine code

---

Can simplify IR by assuming implicit stack

### Example

```
z = x - 2 * y
```

becomes

```
push x
push 2
push y
multiply
subtract
store z
```

### Advantages

- compact form
- introduced names are implicit, not explicit
- simple to generate and execute code

### Disadvantages

- processors operate on registers, not stacks
- difficult to reuse values on stack

## Virtual machines

---

Can interpret IR using “virtual machine”

### Examples

- P-code for Pascal
- postscript for display devices
- Java byte code for everywhere

### Result

- easy & portable
- much slower

### Just-in-time compilation (JIT)

- begin interpreting IR
- find performance critical section(s)
- compile section(s) to native code
- ...or just compile entire program
- compilation time becomes execution time

## Intermediate representations

---

But this isn't the whole story

Symbol table:

- identifiers, procedures
- size, type, location
- lexical nesting depth

Constant table:

- representation, type
- storage class, offset(s)

Storage map:

- storage layout
- overlap information
- (virtual) register assignments

## Java virtual machine (JVM)

---

The JVM consists of four parts

### Memory

- stack (for function call frames)
- heap (for dynamically allocated memory)
- constant pool (shared constant data)
- code segment (instructions of class files)

### Registers

- stack pointer (SP), local stack pointer (LSP), program counter (PC)

### Condition codes

- stores result of last conditional instruction

### Execution unit

1. reads current JVM instruction
2. change state of virtual machine
3. increment PC (modify if call, branch)

## Java byte codes

### Arithmetic instructions

```
ineg      [...i] → [...-i]
iadd      [...i1,i2] → [...i1+i2]
isub      [...i1,i2] → [...i1-i2]
imul      [...i1,i2] → [...i1*i2]
idiv      [...i1,i2] → [...i1/i2]
```

### Direct instructions

```
iinc k a  [...] → [...]    local[k] ← local[k]+a
```

### Branch instructions

```
goto L    [...] → [...]    branch to L
ifeq L    [...i] → [...]    branch if i = 0
ifne L    [...i] → [...]    branch if i != 0
ifnull L  [...o] → [...]    branch if o = null
ifnonnull L [...o] → [...]  branch if o != null
if_icmpeq L [...i1:i2] → [...] branch if i1 = i2
if_icmpne L [...i1:i2] → [...] branch if i1 != i2
if_icmpgt L [...i1:i2] → [...] branch if i1 > i2
if_icmplt L [...i1:i2] → [...] branch if i1 < i2
if_acmpeq L [...o1:o2] → [...] branch if o1 = o2
if_acmpne L [...o1:o2] → [...] branch if o1 != o2
```

## Java bytecode interpreter

```
pc = code.start
while (true) {
    new_pc = pc + inst_len(code[pc]);
    switch (opcode(code[pc])) {
        case iconst_1:
            push(1); break;
        case iload:
            push(local[code[pc+1]]); break;
        case istore:
            t ← pop();
            local[code[pc+1]] ← t; break;
        case iadd:
            t1 ← pop(); t2 ← pop();
            push(t1 + t2); break;
        case ifeq:
            t ← pop();
            if (t = 0) new_pc = code[pc+1]; break;
        ...
    }
    pc ← new_pc;
}
```

## Java byte codes

### Constant loading

```
iconst_0  [...] → [...0]
iconst_1  [...] → [...1]
aconst_null [...] → [...null]
ldc_int i  [...] → [...i]
ldc_string s [...] → [...str(s)]
```

### Locals operations

```
iload k   [...] → [...local[k]]
aload k   [...] → [...local[k]]
istore k  [...i] → [...]    local[k]←i
astore k  [...o] → [...]    local[k]←o
```

### Stack operations

```
dup       [...v] → [...v,v]
pop       [...v] → [...]
swap     [...v1,v2] → [...v2,v1]
```

### Functions

```
invoke    [...args] → [...]  push stack frame, ...
ireturn   [...i] → [...]    ret i, pop stack frame
areturn   [...o] → [...]    ret o, pop stack frame
return    [...] → [...]    pop stack frame
```