Intermediate Representations

- Decisions in IR design affect the speed and efficiency of the compiler.
- Some important IR properties:
  - Ease of generation
  - Ease of manipulation
  - Procedure size
  - Freedom of expression
  - Level of abstraction
- The importance of different properties varies between compilers:
  - Selecting an appropriate IR for a compiler is critical.

Types of Intermediate Representations

- Three major categories:
  - Structural:
    - Graphically oriented
    - Heavily used in source-to-source translators
    - Tend to be large
  - Linear:
    - Pseudo-code for an abstract machine
    - Level of abstraction varies
    - Simple, compact data structures
    - Easier to rearrange
  - Hybrid:
    - Combination of graphs and linear code
- Examples:
  - Structural: Trees, DAGs
  - Linear: Stack machine code
  - Hybrid: Control-flow graph

Level of Abstraction

- The level of detail exposed in an IR influences the profitability and feasibility of different optimizations.
- Two different representations of an array reference:

```
loadI 1 => r
sub r, x1 => x2
loadI 10 => x3
mul r, x2 => x4
sub r, x3 => x5
add r, x4 => x6
load @A => r
```

High level AST: Good for memory disambiguation

```
load 10 => x2
loadA @A => x3
add r, x2 => x6
load r => r
```

Low level linear code: Good for address calculation

```
load 10 => x2
loadA @A => x3
add r, x2 => x6
load r => r
```

High level linear code
Abstract Syntax Tree

An abstract syntax tree is the procedure’s parse tree with the nodes for most non-terminal nodes removed.

- Can use linearized form of the tree
  → Easier to manipulate than pointers
  \( x \ 2 \ y \ast - \) in postfix form
  \(- \ast 2 \ y \ x \) in prefix form
- S-expressions are (essentially) ASTs

Stack Machine Code

Originally used for stack-based computers, now Java

- Example: \( x - 2 \ast y \) becomes

  - push \( x \) push 2 push y multiply subtract

Advantages
- Compact form
- Introduced names are implicit, not explicit
- Simple to generate and execute code
Useful where code is transmitted over slow communication links (the net)

Directed Acyclic Graph

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

- Makes sharing explicit
- Encodes redundancy
  Some expression twice means that the compiler might arrange to evaluate it just once!

Three Address Code

Several different representations of three address code
- In general, three address code has statements of the form:
  \( x \leftarrow y \ op \ z \)
With 1 operator (op) and, at most, 3 names \( (x, y, z) \)

Example:
\[ z \leftarrow x - 2 \ast y \]
\[ w \leftarrow x / 2 \]

Advantages:
- Resembles many machines
- Introduces a new set of names
- Compact form

Three Address Code: Quadruples

Naïve representation of three address code
- Table of \( k \ast 4 \) small integers
- Simple record structure
- Easy to reorder
- Explicit names

The original FORTRAN compiler used "quads"

<table>
<thead>
<tr>
<th>Quadruples</th>
<th>RISC assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>load ( r1, y )</td>
<td>load y</td>
</tr>
<tr>
<td>load ( r2, 2 )</td>
<td>load 2</td>
</tr>
<tr>
<td>mult ( r3, r2, r1 )</td>
<td>mult 3 2 1</td>
</tr>
<tr>
<td>load ( r4, x )</td>
<td>load 4 X</td>
</tr>
<tr>
<td>sub ( r5, r4, r3 )</td>
<td>sub 5 4 2</td>
</tr>
</tbody>
</table>

Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

<table>
<thead>
<tr>
<th>Triples</th>
<th>Implicit names take no space!</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) load ( y )</td>
<td></td>
</tr>
<tr>
<td>(2) load 2</td>
<td></td>
</tr>
<tr>
<td>(3) mult 3 2 1</td>
<td></td>
</tr>
<tr>
<td>(4) load ( x )</td>
<td></td>
</tr>
<tr>
<td>(5) sub 5 4 2</td>
<td></td>
</tr>
</tbody>
</table>

Implicit names take no space!
Three Address Code: Indirect Triples
- List triples in a statement list data structure
- Implicit name space
- Uses more space than triples, but easier to reorder

```
stmt array
(103) (100)
load y
(101) (100)
load 2
(100) (102)
mul (100) (101)
(102) (103)
load x
(104) (103)
sub (103) (100)
```

- Major tradeoff between quads and triples is compactness versus ease of manipulation (note: multiple occurrences of same statement in stmt array is possible)
  - compile-time space critical?
  - compilation speed more important?

Static Single Assignment Form (SSA)
- The main idea: each name defined exactly once in program
- Introduce \( \phi \)-functions to make it work

```
Original          SSA-form
x ← ...          x_0 ← ...
y ← ...          y_0 ← ...
while (x < k) goto next
    x ← x + 1
    y ← y + x
    z ← φ(x, y)
    goto loop
loop:    x_1 ← φ(x_0, x_2)
y_1 ← φ(y_0, y_2)
z_1 ← x_1 + 1
y_2 ← y_1 + z
if (x_1 < k) goto loop
next:     ...
```

Strengths of SSA-form
- Sharper analysis
- \( \phi \)-functions give hints about placement
- (sometimes) faster algorithms

Two Address Code
- Allows statements of the form \( x ← x op y \)
  - Has 1 operator (op) and, at most, 2 names (x and y)

Example:
\[
z ← x - 2 * y \]
  - becomes
  \[
  t_1 ← 2
  t_2 ← load y
  t_3 ← t_1
  z ← load x
  z ← z - t_2
  \]

- Can be very compact

Problems
- Machines no longer rely on destructive operations
- Difficult name space
  - Destructive operations make reuse hard
  - Good model for machines with destructive ops (PDP-11)

Using Multiple Representations
- Repeatedly lower the level of the intermediate representation
  - Each intermediate representation is suited towards certain optimizations
  - Example: the Open64 compiler
    - WHIRL intermediate format
      - Consists of 5 different IRs that are progressively more detailed

Control-flow Graph (CFG)
- Models the transfer of control in the procedure
- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example
```
if (x = y)
    a ← 2
    b ← 5
    c ← a * b
```

Maximal length sequences of straight-line code

Memory Models
Two major models
- Register-to-register model
  - Keep all values that can legally be stored in a register in registers
  - Ignore machine limitations on number of registers
  - Compiler back-end must insert loads and stores
- Memory-to-memory model
  - Keep all values in memory
  - Only promote values to registers directly before they are used
  - Compiler back-end can remove loads and stores
- Compilers for RISC machines usually use register-to-register
  - Reflects programming model
  - Easier to determine when registers are used
The Rest of the Story...

Representing the code is only part of an IR.

There are other necessary components:
- Symbol table (already discussed)
- Constant table
  → Representation, type
  → Storage class, offset
- Storage map
  → Overall storage layout
  → Overlap information
  → Virtual register assignments

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

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
- `intrinsic` \( \rightarrow \) \( \rightarrow \)
- `adder` \( \rightarrow \) \( \rightarrow \)
- `subtractor` \( \rightarrow \) \( \rightarrow \)
- `multiplier` \( \rightarrow \) \( \rightarrow \)
- `divisor` \( \rightarrow \) \( \rightarrow \)

### Direct instructions
- `increment k a` \( \rightarrow \) \( \rightarrow \)
- `local[k] = local[k] + a`

### Branch instructions
- `goto L` \( \rightarrow \) \( \rightarrow \)
- `branch to L`
- `iffollow L` \( \rightarrow \) \( \rightarrow \)
- `branch if i = 0`
- `ifne L` \( \rightarrow \) \( \rightarrow \)
- `branch if i != 0`
- `ifnonnull L` \( \rightarrow \) \( \rightarrow \)
- `branch if o != null`
- `if_icmp L` \( \rightarrow \) \( \rightarrow \)
- `branch if i1 != i2`
- `if_icmp L` \( \rightarrow \) \( \rightarrow \)
- `branch if i1 != i2`
- `if_icmp L` \( \rightarrow \) \( \rightarrow \)
- `branch if i1 != i2`
- `if_acmp L` \( \rightarrow \) \( \rightarrow \)
- `branch if o1 == o2`
- `if_acmp L` \( \rightarrow \) \( \rightarrow \)
- `branch if o1 != o2`

### Constant loading
- `iconst_0` \( \rightarrow \) \( \rightarrow \)
- `iconst_1` \( \rightarrow \) \( \rightarrow \)
- `acconst_null` \( \rightarrow \) \( \rightarrow \)
- `ldc_int i` \( \rightarrow \) \( \rightarrow \)
- `ldc_string s` \( \rightarrow \) \( \rightarrow \)

### Locals operations
- `iload k` \( \rightarrow \) \( \rightarrow \)
- `aload k` \( \rightarrow \) \( \rightarrow \)
- `istore k` \( \rightarrow \) \( \rightarrow \)
- `astore k` \( \rightarrow \) \( \rightarrow \)
Java Byte Codes

Stack operations

dup  \([...,v] \rightarrow [...,v,v]\)
pop  \([...,v] \rightarrow [...]\)
swap \([...,v_1,v_2] \rightarrow [...,v_2,v_1]\)

Functions

invoke \([...,\text{args}] \rightarrow [...]\) push stack frame, ...
ireturn \([...,i] \rightarrow [...]\) ret i, pop stack frame
areturn \([...,o] \rightarrow [...]\) ret o, pop stack frame
return \([...] \rightarrow [...]\) pop stack frame

Java Byte Code Interpreter

```
pc = code.start
while (true) {
    new.pc = pc + inst.len(code[pc]);
    switch (opcode(code[pc])) {
        case (const 1):
            push(i); break;
        case load:
            pushlocal(code[pc+1]); break;
        case store:
            t = pop();
            local[code[pc+1]] = t; break;
        case add:
            t1 = pop(); t2 = pop();
            push(t1 + t2); break;
        case feq:
            t = pop();
            if (t = 0) new.pc = code[pc+1]; break;
        ...
    }
    pc = new.pc;
}
```