Intermediate Representations
Where In The Course Are We?

• Rest of the course: compiler writer needs to choose among alternatives
  • Choices affect
    § the quality of compiled code
    § time to compile
  • There may be no “best answer”
Intermediate Representations

- Front end - produces an intermediate representation (IR)
- Middle end - transforms the IR into an equivalent IR that runs more efficiently
- Back end - transforms the IR into native code

IR encodes the compiler’s knowledge of the program
- Middle end usually consists of many passes
JikesRVM

(IBM Open Source Java JIT compiler)
JikesRVM (IBM Open Source Java JIT compiler)

Class Files (Bytecode) → Middle End → HIR= High-level Intermediate Representation

- Bytecode to HIR
- HIR Optimizer
- Tail Recursion Elimination
- Escape Analysis
- Load Elimination
- Loop Unrolling

- HIR to LIR
- LIR Optimizer
- Copy Propagation
- Constant Propagation
- Constant Subexpression Elimination
- Basic Block Reordering
JikesRVM (IBM Open Source Java JIT compiler)
JikesRVM (IBM Open Source Java JIT compiler)
Intermediate Representations

- Decisions in IR design affect the speed and efficiency of the compiler

- The importance of different properties varies between compilers
  - Selecting the “right” IR for a compiler is critical
Some Important IR Properties

- Ease of generation
  - speed of compilation
- Ease of manipulation
  - improved passes
- Procedure size
  - compilation footprint
- Level of abstraction
  - improved passes
Types of Intermediate Representations

Three major categories

• Structural
• Linear
• Hybrid
Types of Intermediate Representations

Three major categories

• Structural
  → Graphically oriented
  → Heavily used in source-to-source translators
  → Tend to be large

• Linear

• Hybrid

Examples: Trees, DAGs
Types of Intermediate Representations

Three major categories

• Structural

• Linear
  → Pseudo-code for an abstract machine
  → Level of abstraction varies
  → Simple, compact data structures
  → Easier to rearrange

• Hybrid

Examples: 3 address code, Stack machine code
Types of Intermediate Representations

Three major categories

• Structural

• Linear

• Hybrid

→ Combination of graphs and linear code

Examples:
  Control Flow Graph
Level of Abstraction

- Two different representations of an array ref:

High level AST

- \(A\)
- \(i\)
- \(j\)

**subscript**

Low level Linear Code

- loadI 1 => \(r_1\)
- sub \(r_j, r_1\) => \(r_2\)
- loadI 10 => \(r_3\)
- mult \(r_2, r_3\) => \(r_4\)
- sub \(r_i, r_1\) => \(r_5\)
- add \(r_4, r_5\) => \(r_6\)
- loadI @$A$ => \(r_7\)
- add \(r_7, r_6\) => \(r_8\)
- load \(r_8\) => \(r_{Aij}\)

Good for memory disambiguation

Good for address calculation
Level of Abstraction

- Structural IRs are usually considered high-level
- Linear IRs are usually considered low-level
- Not necessarily true:

\[
\text{loadArray } A, i, j
\]

Low level AST     High level linear code
Abstract Syntax Tree

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

\[ x - 2 * y \]

Parse Tree

Abstract Syntax Tree
Directed Acyclic Graph

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

- Makes sharing explicit
- Encodes redundancy

With two copies of the same expression, the compiler might be able to arrange the code to evaluate it only once.
Stack Machine Code

Originally used for stack-based computers, now Java

- Example:

\[ x - 2 \cdot y \] becomes

```
push x
push 2
push y
multiply
subtract
```
Stack Machine Code

- Operations take operands from a stack
- Compact form
- A form of one-address code
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code
Stack Machine Code Advantages

\[ x - 2 \times y \]

Result is stored in a temporary! Explicit name for result.

Multiply pops two items off of stack and pushes result! Implicit name for result.

push 2
push y
multiply
push x
subtract
Three Address Code

Different representations of three address code

• In general, three address code has statements of the form:

\[ x \leftarrow y \text{ op } z \]

With 1 operator (\textit{op}) and (at most) 3 names (\textit{x, y, & z})
Three Address Code

Example:

\[ z \leftarrow x - 2 \times y \]

becomes

\[ t \leftarrow 2 \times y \]
\[ z \leftarrow x - t \]

Explicit name for result.
Three Address Code Advantages

- Resembles many real (RISC) machines
- Introduces a new set of names
- Compact form
Three Address Code: Simple Array

Naïve representation of three address code
• Table of $k \times 4$ small integers

RISC assembly code

<table>
<thead>
<tr>
<th>Destination</th>
<th>Two operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>1 y</td>
</tr>
<tr>
<td>loadI</td>
<td>2 2</td>
</tr>
<tr>
<td>mult</td>
<td>3 2 1</td>
</tr>
<tr>
<td>load</td>
<td>4 x</td>
</tr>
<tr>
<td>sub</td>
<td>5 4 3</td>
</tr>
</tbody>
</table>
Three Address Code: Array of Pointers

- Index causes level of indirection
- Easy (and cheap) to reorder
- Easy to add (delete) instructions
Three Address Code: Array of Pointers

- Index causes level of indirection
- Easy (and cheap) to reorder
- Easy to add (delete) instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>r1</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>r2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>r3</th>
<th>r2</th>
<th>r1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>r4</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>r5</th>
<th>r4</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Three Address Code: Linked List

- No additional array of indirection
- Easy (and cheap) to reorder than simple table
- Easy to add (delete) instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 1</th>
<th>Register 2</th>
<th>Register 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>r1</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>loadi</td>
<td>r2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>r3</td>
<td>r2</td>
<td>r1</td>
</tr>
<tr>
<td>load</td>
<td>r4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>r5</td>
<td>r4</td>
<td>r3</td>
</tr>
</tbody>
</table>
Control-Flow Graphs

Linear IR

1. a := 0
2. b := a * b
3. L1: c := b/d
4. if c < x got L2
5. e := b / c
6. f := e + 1
7. L2: g := f
8. h := t - g
9. if e > 0 goto L3
10. goto L1
11. L3: return

Hybrid IR: CFG

1. a := 0
2. b := a * b
3. L1: c := b/d
4. if c < x got L2
5. e := b / c
6. f := e + 1
7. L2: g := f
8. h := t - g
9. if e > 0 goto L3
10. goto L1
11. L3: return
Control-Flow Graphs

• **Node**: an instruction or sequence of instructions (a basic block)
  - Two instructions $i, j$ in same basic block iff execution of $i$ guarantees execution of $j$

• **Directed edge**: potential flow of control

• **Distinguished start node** *Entry*
  - First instruction in program
Control-flow Graph

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

a ← 3
b ← 4

c ← a * b
```

Basic blocks — Maximal length sequences of straight-line code
Using Multiple Representations

- Repeatedly lower the level of the intermediate representation
  - Each intermediate representation is suited towards certain optimizations
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 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**
- **Constant table**
  - Representation, type
  - Storage class, offset
- **Storage map**
  - Overall storage layout
  - Overlap information
  - Virtual register assignments
Symbol Tables

Traditional approach to building a symbol table uses hashing

- One table scheme
  - Lots of wasted space

$h("fie")$
$h("fee")$
$h("fum")$
$h("foe")$

| h("fie") | fee | integer | scalar | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h(&quot;fee&quot;)</td>
<td>foe</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>h(&quot;fum&quot;)</td>
<td>fum</td>
<td>float</td>
<td>scalar</td>
</tr>
</tbody>
</table>
| h("foe") | fie | char * | array  | ...

Hash table