Intermediate Representations
The Glue That Holds a Compiler Together

Comp 412
Mea Culpa

Due to my complicated schedule, Lab 1 has not been graded
- No excuse
- I declare amnesty on Lab 1. All of you will receive full credit.
- We will do better on the exam and the next two labs.

Caution
- Several people have not yet taken the exam
- Be circumspect in your discussions of the contents of the exam
Where In The Course Are We?

**Obvious answer:** *at the start of Chapter 5 in EaC2e*

**More important answer**
- We are on the cusp of the art, science, & engineering of compilation
- Scanning & parsing are applications of automata theory
- Syntax-directed translation is mostly software engineering
- The mid-section of the course will focus on issues where the compiler writer needs to choose among alternatives
  - The choices matter; they affect the quality of compiled code
  - There may be no “best answer” or “best practice”

**To my mind, the fun begins at this point**
Intermediate Representations

Front End emits IR for the rest of the compiler to use

- Scanner & parser work from the syntax of the source code
  - Parser emits IR using SDT or some similar technique
- Rest of the compiler works from IR
  - Analyzes IR to learn about the code
  - Transforms IR to improve final code
- IR determines what a compiler can do to the code
  - Can only manipulate details that are represented in the IR
Intermediate Representations

**IR** is the vehicle that carries information between phases

- **Front end**: produces an **IR** version of the input program
- **Optimizer**: transforms the **IR** into an equivalent **IR** that runs faster
  - Each “pass” reads and writes **IR**
- **Back end**: systematically transforms the **IR** into native code

**IR determines both the compiler’s ambition & its chances for success**

- The compiler’s knowledge of the code is encoded in the **IR**
- The compiler can only manipulate what is represented by the **IR**
Intermediate Representations

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

Some important IR properties

- Ease of generation
- Ease of manipulation
- Cost of manipulation
- Procedure size
- Expressiveness
- Level of abstraction

Example

*Ease of Manipulation:* in your register allocator lab, some of you used Java’s ArrayList class. ArrayList provides many useful features that you did not need to implement yourself.

*Cost of Manipulation:* ArrayList is efficient, unless you are inserting items into the middle of a long list, as occurs when inserting spills and restores.

*Impact on the Compiler:* to combat the cost of insertion, some of you printed the output on the fly rather than updating the IR.

The importance of different properties varies between compilers

⇒ Selecting an appropriate IR for a compiler is critical
Intermediate Representations

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

Some important IR properties

• Ease of generation
• Ease of manipulation
• Cost of manipulation
• Procedure size
• Expressiveness
• Level of abstraction

The importance of different properties varies between compilers

⇒ Selecting an appropriate IR for a compiler is critical

Example

Expressiveness: a compiler with a near-source level IR may have trouble representing the results of some optimizations.

```
for i = 0 to n
    for j = 0 to m
        a[i,j] = 0
        *p++ = 0
```

Simple initialization After OSR

The implementation of operator strength reduction (OSR) can only produce this result if the IR can represent *p++. (§ 10.7.2 in EaC2e.)
Intermediate Representations

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

Some important IR properties

- Ease of generation
- Ease of manipulation
- Cost of manipulation
- Procedure size
- Expressiveness
- Level of abstraction

Example

*Level of Abstraction:* copying a string is a complex operation that involves an internal loop. Explicitly representing the loop and its details exposes all those details to uniform optimization (a good thing).

Explicitly representing the loop makes it difficult to move the copy to another location in the code — moving control flow constructs is difficult, at best.

Representing the copy as a single operation, like the S370 `mvcl` makes it easy to move.

The importance of different properties varies between compilers

⇒ Selecting an appropriate IR for a compiler is critical
Intermediate Representations

Today, we will focus on representing the operations in the code
- Arithmetic expressions
- Assignment
- Control-flow in the program

Later, we will come back and talk about representing named entities
- Symbol tables
- Renaming
- Storage models
Taxonomy of Intermediate Representations

Three major categories

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

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

• Hybrid IRs
  – Combination of graphs and linear code
  – Example: control-flow graph

Examples:
- Trees, DAGs
- 3 address code
- Stack machine code
- Control-flow graph
- SSA Form
Level of Abstraction

The level of detail exposed in an IR influences the profitability and feasibility of different optimizations.

Here are two different representations of an array reference:

```
loadI 1   => r1
sub  r_j, r_1 => r2
loadI 10  => r3
mult  r_2, r_3 => r4
sub   r_i, r_1 => r5
add   r_4, r_5 => r6
loadI  @A  => r7
add   r_7, r_6 => r8
load  r_8   => r_{Aij}
```

High-level Abstract Syntax Tree
Good for memory disambiguation

Low-level Linear Code
Good for optimizing the address calculation
Level of Abstraction

People tend to confuse level of abstraction with structure

- Structural IRs are often considered high-level
- Linear IRs are often considered low-level
- Those generalizations are not necessarily true

Low level AST

```
aload
+
+
*
-
@
10
j
1
j
1
```

High-level linear code

```
loadArray A, i, j
```

In Chapter 11 of EaC2e, we will see trees that have a lower level of abstraction than the machine code
A syntax tree represents the front end’s parse of the code, in detail

Syntax trees are often used in source-to-source systems
- Captures the precise (syntactic) form of the input program
- Has all of the detail that you could need
  - Compiler generated all the detail it has from the parse

Syntax trees tend to be inefficient
- Lots of unnecessary nodes and edges
- Lots of implicit detail that might be useful to represent explicitly

Syntax trees can be represented with a linear notation (e.g., prefix or postfix)
Abstract Syntax Tree

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

\[
\begin{align*}
\text{AST for } x - 2 \ast y
\end{align*}
\]

- \textbf{ASTs} are space efficient trees that capture most of the interesting information found in a syntax tree.
  - Can regenerate source code in a treewalk, with a little cleverness
  - Many fewer nodes and edges than in a syntax tree.
- \textbf{S}-expressions in Scheme or Lisp, are (essentially) \textbf{ASTs}

In practice, \textbf{ASTs} tend to be large — not because they must, but they grow (see the digression on page 228 in EaC2e).
A directed acyclic graph (DAG) is an AST with a unique node for each value (an AST with sharing)

- Makes sharing explicit
- Encodes redundancy

If the compiler uses graphical IRs, a DAG is a natural way to represent redundancy.

With two copies of the same expression, the compiler may be able to arrange the code to evaluate it only once.
Implementing Trees

In earlier courses, you learned to build purpose-built trees (or to use some generic library)

- Nodes connected by edges
- Nodes of various types and arities
- Allocating nodes of different sizes complicates both allocation & fragmentation in the heap (malloc())

Knuth showed that you can map any arbitrary tree onto a binary tree

- Allocate uniform-size, binary nodes
- Two “pointers”: child and sibling
- Simplify allocation and traversal

---

1 See Knuth Volume 1, pp. 332-334; also EaC2e pp. 744-746
**Dependence Graph**

Compilers use a dependence graph to understand & preserve the flow of values in a block.

1. load 8 => r1
2. load 12 => r2
3. mult r1, r2 => r3
4. add r1,r3 => r4

Original Code

In Lab 3, you will build dependence graphs.
Call Graph

When compilers try to optimize multiple procedures together, they build a call graph to represent the flow of control between procedures

- Nodes in the call graph represent procedures
- Edges represent individual calls, making it a multi-graph
  - Each edge represents a different "calling environment"

Call graphs are arbitrary graphs

- Recursion (direct or indirect) creates cycles
- Complex call patterns create complex graphs
  - Think about a recursive descent parser
- Some interprocedural optimizations change the call graph as they manipulate it
  - Think about inline substitution or procedure cloning
Digression (or Rant)

The role of string data

Principle:

*For the sake of compactness and efficiency, the compiler should almost never store a string value in the IR*

- Strings are expensive to compare
- Strings are large relative to their information content
- In most cases, we need just one copy of a string

How to handle strings in your IR

- Use a hash table or map to convert each string to a small integer
- Represent the string with the integer
- Compare strings as integers
Taxonomy of Intermediate Representations

Three major categories

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

Examples:
Trees, DAGs

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

Examples:
3 address code
Stack machine code

• Hybrid IRs
  – Combination of graphs and linear code
  – Example: control-flow graph

Examples:
Control-flow graph
SSA Form
Three Address Code

Several 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 (\(x\), \(y\), & \(z\))

Example:
\[ z \leftarrow x \; 2 \; \ast \; y \; \text{becomes} \]
\[ t \leftarrow 2 \; \ast \; y \]
\[ z \leftarrow x \; \ast \; t \]

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

The concept of “three address code” has many implementations.

See Lab 1 and Lab 3
Three Address Code: As Quadruples

Naïve representation of three address code

- Table of \( k \times 4 \) small integers
- Simple record structure
- Easy, albeit slow, to reorder
- Explicit names

The original FORTRAN compiler used “quads”

Store opcode & operands as small integers, of course.

RISC assembly code

\[
\begin{align*}
\text{load} & : r1, y \\
\text{loadI} & : r2, 2 \\
\text{mult} & : r3, r2, r1 \\
\text{load} & : r4, x \\
\text{sub} & : r5, r4, r3
\end{align*}
\]

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Op1</th>
<th>Op2</th>
<th>Op3</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>1</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>loadi</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
Three Address Code: As Triples

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

<table>
<thead>
<tr>
<th>Implicit Name</th>
<th>Opcode</th>
<th>Op₁</th>
<th>Op₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>load</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>loadI</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>mult</td>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td>(4)</td>
<td>load</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>sub</td>
<td>(4)</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Remember, for a long time, 640 KB was a lot of RAM
Three Address Code: As Indirect Triples

- List first triple in each statement
- Implicit name space for statements
- Uses more space than triples, but easier to reorder

<table>
<thead>
<tr>
<th>Stmt List</th>
<th>Implicit Names</th>
<th>Indirect Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>(100)</td>
<td>(100)</td>
<td>load y</td>
</tr>
<tr>
<td>(105)</td>
<td>(101)</td>
<td>loadI 2</td>
</tr>
<tr>
<td></td>
<td>(102)</td>
<td>mult (100) (101)</td>
</tr>
<tr>
<td>(103)</td>
<td></td>
<td>load x</td>
</tr>
<tr>
<td>(104)</td>
<td></td>
<td>sub (103) (102)</td>
</tr>
</tbody>
</table>

Standard trick: when you need the ability to rearrange objects in memory, add a level of indirection.

- Major tradeoff between quads and triples is compactness versus ease of manipulation
  - In the past compile-time space was critical
  - Today, speed may be more important
BEGIN INTEGER K;
   ARRAY A[ 1:I-J ];
   K := 0;
L:  IF I > J
   THEN K := K + A[I-J]*6
   ELSE BEGIN I := I+1; I := I+1; GO TO L END
END

FIGURE 11.1. An ALGOL Program Segment.

(1) BLOCK
(2) - I, J
(3) BOUNDS 1, (2)
(4) ADEC A
(5) := 0, K
(6) - I, J
(7) BMZ (13), (6)
(8) - I, J
(9) * A[ (8) ], 6
(10) + K, (9)
(11) := (10), K
(12) BR (18)
(13) + I, 1
(14) := (13), I
(15) + I, 1
(16) := (15), I
(17) BRL L
(18) BLCKEND

FIGURE 11.7. Triples for Program Segment of Figure 11.1.

BEGIN INTEGER K;
ARRAY A[1:1-J];
K := 0;
L:
IF I > J
THEN K := K + A[I-J]*6
ELSE BEGIN I := I+1; I := I+1; GO TO L END
END

FIGURE 11.1. An ALGOL Program Segment.

1. (1) 10. (8) (1) BLOCK (8) + K, (7)
2. (2) 11. (9) (2) - I, J (9) := (8), K
3. (3) 12. (10) (3) BOUNES 1, (2) (10) BR 18
4. (4) 13. (11) (4) ADEC A (11) + I, 1
5. (5) 14. (12) (5) := 0, K (12) := (11), I
6. (2) 15. (11) (6) BMZ 13, (2) (13) BRL L
8. (2) 17. (13) (OPER, TRIPLE
9. (7) 18. (14)

FIGURE 11.8. Indirect Triples for Figure 11.1.

Two-Address Code

Two-address code allows statements of the form

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

Each operation has 1 operator (\textit{op}) and, at most, 2 names (x and y)

Example

\[ z \leftarrow x - 2 \times y \quad \text{becomes} \]

- Can be very compact

\[
\begin{align*}
  t_1 & \leftarrow 2 \\
  t_2 & \leftarrow \text{load } y \\
  t_2 & \leftarrow t_2 \times t_1 \\
  z & \leftarrow \text{load } x \\
  z & \leftarrow z - t_2
\end{align*}
\]

We write:

\[
\begin{align*}
  r1 + r2 & \Rightarrow r2 \\
  \text{as:} & \\
  \text{add } r1, r2
\end{align*}
\]

Problems

- Difficult name space
  - Destructive operations make reuse hard
  - Good model for machines with destructive ops (PDP-11, x86)

- We would like destructive operations to become a thing of the past

Not many arguments in favor of two-address code
Stack Machine Code

Originally used for stack-based computers, now Java

• Example:

  \[ x - 2 \times y \]  
  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 or where memory is limited

• Java bytecode was designed for transmission over slow links
• Follows a long line of bytecode-like *IRs* designed to be compact

In a stack machine, most operations are destructive.
Taxonomy of Intermediate Representations

Three major categories

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

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

• Hybrid IRs
  – Combination of graphs and linear code
  – Example: control-flow graph

Examples:
- Trees, DAGs
- 3 address code
- Stack machine code
- Control-flow graph
- SSA Form