Intermediate representations

source code → \textbf{front end} \quad \textbf{ir} \quad \textbf{opt.} \quad \textbf{ir} \quad \textbf{back end} → target code

**front end** produce 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**

---

**Important IR Properties**

- ease of generation
- ease and cost of manipulation
- level of abstraction
- expressiveness (e.g.: operations, addressing modes)
- size of typical procedure

Subtle design decisions in the IR have far reaching effects on the speed and effectiveness of the compiler.

Level of exposed detail is a crucial consideration.

---

**Intermediate representations**

---

**Why use an intermediate representation?**

1. break the compiler into manageable pieces
   good software engineering technique
2. allow a complete pass before code is emitted
   lets compiler consider more than one option
3. simplifies retargeting to new host
   isolates back end from front end
4. simplifies handling of “poly-architecture” problem
   \( m \) lang’s, \( n \) targets \( \Rightarrow m + n \) components  
   \textit{(myth)}
5. enables machine-independent optimization
   general techniques, multiple passes

An intermediate representation is a compile-time data structure

---

**Intermediate representations**

---

**Representations talked about in the literature include:**

- abstract syntax trees (AST)
- linear (operator) form of tree
- directed acyclic graphs (DAG)
- control flow graphs (CFG)
- program dependence graphs (PDG)
- static single assignment form (SSA)
- stack code
- three address code
- hybrid combinations
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
- node and edge annotations 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

Abstract syntax tree

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

```
- 
  <id,x>  *
  <num,2>  <id,y>
```

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)
```

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

```
x=y
  s1
s2
  s3
```
Stack machine code

Several stack-based computers have been built.
Compilers can directly generate stack code

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

P-code, BCPL, reverse polish notation (RPN)
calculators, JAVA VM

Three address code

Three address code is a term used to describe a variety
of representations.

In general, they allow statements of the form:

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

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

Simpler form of expression

\[ x - 2 \times y \]

becomes

\[ t_1 \leftarrow 2 \times y \]
\[ t_2 \leftarrow x - t_1 \]

Advantages

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

Can include forms of prefix or postfix code

Three address code

Typical statement types include:

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 — \text{goto} L
6. conditional branches — if \( x \text{ relop } y \) \text{goto} L
7. procedure calls — param \( x_1, \ldots \text{param } x_n \)
   call \( p, n \)
8. procedure returns — \text{return} y
9. address and pointer assignments — \( x \leftarrow \&y \) and \( x \leftarrow *y \)

Three address code

Until recently, compile-time space was a serious issue

- machines had small memories
- compiler touches space it allocates

Compact forms of three address code

- quadruples
- triples
- indirect triples

Major tradeoff is compactness versus ease of
manipulation

Today, speed (and locality) may be more important
### Three address code

#### Quadruples

\[
x - 2 \times y
\]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>load</td>
<td>t1</td>
</tr>
<tr>
<td>(2)</td>
<td>loadi</td>
<td>t2</td>
</tr>
<tr>
<td>(3)</td>
<td>mult</td>
<td>t3</td>
</tr>
<tr>
<td>(4)</td>
<td>load</td>
<td>t4</td>
</tr>
<tr>
<td>(5)</td>
<td>sub</td>
<td>t5</td>
</tr>
</tbody>
</table>

- simple record structure with four fields
- easy to reorder
- explicit names of intermediate results (*temporaries*)

#### Triples

\[
x - 2 \times y
\]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>load</td>
</tr>
<tr>
<td>(2)</td>
<td>loadi</td>
</tr>
<tr>
<td>(3)</td>
<td>mult</td>
</tr>
<tr>
<td>(4)</td>
<td>load</td>
</tr>
<tr>
<td>(5)</td>
<td>sub</td>
</tr>
</tbody>
</table>

- use table index as implicit name for temporaries
- require only three fields in record
- harder to reorder

### Three address code

#### Indirect Triples

\[
x - 2 \times y
\]

<table>
<thead>
<tr>
<th>stmt</th>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(100)</td>
<td>load</td>
<td>y</td>
</tr>
<tr>
<td>(2)</td>
<td>(101)</td>
<td>loadi</td>
<td>2</td>
</tr>
<tr>
<td>(3)</td>
<td>(102)</td>
<td>mult</td>
<td>(100)</td>
</tr>
<tr>
<td>(4)</td>
<td>(103)</td>
<td>load</td>
<td>x</td>
</tr>
<tr>
<td>(5)</td>
<td>(104)</td>
<td>sub</td>
<td>(103)</td>
</tr>
</tbody>
</table>

- use array called `stmt` to list pointers to triples
- simplifies moving statements
- more space than triples, same as quadruples for single statement
- `stmt[i] = stmt[j]` possible ⇒ more compact representation of program than quadruples
- implicit name space management for temporaries

### Other hybrids

An attempt to get the best of both worlds.
- graphs where they work
- linear codes where it pays off

Unfortunately, there appears to be little agreement about where to use each kind of IR to best advantage.

For example:
- F77 directly emit assembly code for control flow, but build and pass around expression trees for expressions.
- Many systems use a control flow graph with three address code for each basic block.
- Source-to-source translators typically use AST and dependence graph
Intermediate representations

*But, this isn’t the whole story.*

Symbol table:
- variables, procedures, temporaries
- type (size, representation, storage layout)
- storage class, offset

Constant table:
- type
- storage class, offset

Advice

- Many kinds of IR are used in practice.
- Best choice depends on application.
- There is no widespread agreement on this subject.
- A compiler may need several different IRs.
- Choose IR with right level of detail.
- Keep manipulation costs in mind.

Next Three Lectures

1. *Code Optimization:* Barbara Ryder will be lecturing for me. This is her research area - the lecture should be great!!

2. *Code Generation:* Craig Nevill-Manning will be lecturing for me.

3. *Code Generation:* Me again ...