



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.

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 (*myth*)
5. enables machine-independent optimization
general techniques, multiple passes

An intermediate representation is a compile-time data structure

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

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

Stack machine code

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

Example

$x - 2 * y$

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

$x - 2 * y$

becomes

$t1 \leftarrow 2 * y$

$t2 \leftarrow x - t1$

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 — **goto** L
6. conditional branches — **if** $x \text{ relop } y$ **goto** L
7. procedure calls — **param** x_1, \dots **param** x_n
call p, n
8. procedure returns — **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 * y				
(1)	load	t1	y	
(2)	loadi	t2	2	
(3)	mult	t3	t2	t1
(4)	load	t4	x	
(5)	sub	t5	t4	t3

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

Three address code

Triples

x - 2 * y			
(1)	load	y	
(2)	loadi	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

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

Three address code

Indirect Triples

x - 2 * y				
	stmt	op	arg1	arg2
(1)	(100)	(100)	load	y
(2)	(101)	(101)	loadi	2
(3)	(102)	(102)	mult	(100) (101)
(4)	(103)	(103)	load	x
(5)	(104)	(104)	sub	(103) (102)

- use array called **stmt** to list pointers to triples
- simplifies moving statements
- more space than triples, same as quadruples for single statement
- $\text{stmt}[i] = \text{stmt}[j]$ possible \Rightarrow 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

But, this isn't the whole story.

Symbol table:

- variables, procedures, temporaries
- type (\rightarrow size, representation, storage layout)
- storage class, offset

Constant table:

- type
- storage class, offset

- 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 ...

Read ASU 8.1-8.7 and 9.1-9.2.