

CS3300 - Compiler Design Semantic Analysis - IR Generation

V. Krishna Nandivada

IIT Madras

Copyright ©2000 by Antony L. Hosking. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from hosking@cs.purdue.edu.



Intermediate representations

Why use an intermediate representation?

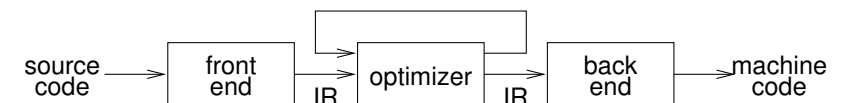
- 1 break the compiler into manageable pieces
– good software engineering technique
- 2 simplifies retargeting to new host
– isolates back end from front end
- 3 simplifies handling of “poly-architecture” problem
– m lang’s, n targets $\Rightarrow m + n$ components
- 4 enables machine-independent optimization
– general techniques, multiple passes

(myth)

An intermediate representation is a compile-time data structure



Intermediate representations



Generally speaking:

- front end produces IR
- optimizer transforms that representation into an equivalent program that may run more efficiently
- back end transforms IR into native code for the target machine



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
- program dependence graphs
- static single assignment form
- 3-address code
- hybrid combinations



Intermediate representations - properties

Important IR Properties

- ease of generation
- ease of manipulation
- cost of manipulation
- level of abstraction
- freedom of expression
- 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.



IR design issues

- Is the chosen IR appropriate for the (analysis/ optimization/ transformation) passes under consideration?
- What is the IR level: close to language/machine.
- Multiple IRs in a compiler: for example, High, Medium and Low

$t1 \leftarrow a[i, j+2]$	$t1 \leftarrow j + 2$	$r1 \leftarrow [fp-4]$
	$t2 \leftarrow i * 20$	$r2 \leftarrow r1 + 2$
	$t3 \leftarrow t1 + t2$	$r3 \leftarrow [fp-8]$
	$t4 \leftarrow 4 * t3$	$r4 \leftarrow r3 * 20$
	$t5 \leftarrow \text{addr } a$	$r5 \leftarrow r4 + r2$
	$t6 \leftarrow t5 + t4$	$r6 \leftarrow 4 * r5$
	$t7 \leftarrow *t6$	$r7 \leftarrow fp - 216$
		$f1 \leftarrow [r7+r6]$

(a) High-, (b) medium-, and (c) low-level representations of a C array reference.

- In reality, the variables etc are also only pointers to other data structures.



Intermediate representations

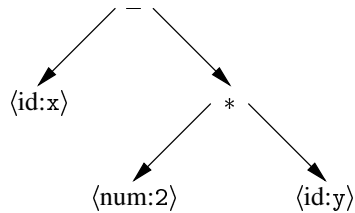
Broadly speaking, IRs fall into three categories:

- Structural
 - structural IRs are graphically oriented
 - examples include trees, DAGs
 - heavily used in source to source translators
 - nodes, edges 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
 - e.g., control-flow graphs
 - Example: GCC Tree IR.



Abstract syntax tree

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



This represents "x - 2 * y".

For ease of manipulation, can use a linearized (operator) form of the tree.

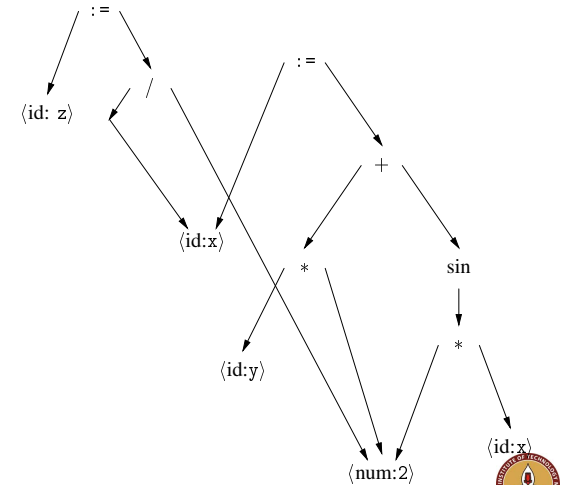
e.g., in postfix form: x 2 y * -



Directed acyclic graph

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

```
x := 2 * y + sin(2*x)
z := x / 2
```



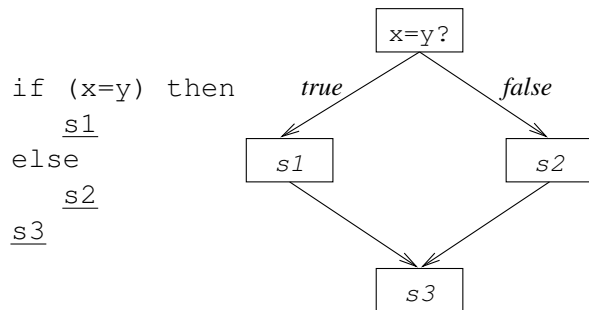
Q: What to do for matching names present across different functions



Control flow graph

The control flow graph (CFG) models the transfers of control in the procedure

- nodes in the graph are basic blocks straight-line blocks of code
- edges in the graph represent control flow loops, if-then-else, case, goto



3-address code

- At most one operator on the right side of an instruction.
- 3-address code can mean a variety of representations.
- In general, it allows 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



3-address code: Addresses

Three-address code is built from two concepts: addresses and instructions.

- An address can be
 - A name: source variable program name or pointer to the Symbol Table name.
 - A constant: Constants in the program.
 - Compiler generated temporary.



3-address code - implementation

Quadruples

- Has four fields: op, arg1, arg2 and result.
- Some instructions (e.g. unary minus) do not use arg2.
- For copy statement : the operator itself is ≡; for others it is implied.
- Instructions like `param` don't use neither arg2 nor result.
- Jumps put the target label in result.

x - 2 * y				
	op	result	arg1	arg2
(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



3-address code

Typical instructions 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 goto L`
- 7 procedure calls
`param x1, param x2, ... param xn`
`and`
`call p, n`
- 8 address and pointer assignments

How to translate:

```
if (x < y) S1 else
S2
```

?



3-address code - implementation

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
- require only three fields in record
- harder to reorder



3-address code - implementation

Indirect Triples

$x - 2 * y$

	exec-order	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)

- simplifies moving statements (change the execution order)
- more space than triples
- implicit name space management



Indirect triples advantage

```
for i:=1 to 10 do
begin
  a=b*c
  d=i*3
end
      (a)
```

Optimized version

```
a=b*c
for i:=1 to 10 do
begin
  d=i*3
end
      (b)
```

```
(1) := 1 i
(2) nop
(3) * b c
(4) := (2) a
(5) * 3 i
(6) := (4) d
(7) + 1 i
(8) LE I 10
(9) IFT goto (2)
```

Execution Order (a) : 123456789

Execution Order (b) : 341256789



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:

- PCC and FORTRAN 77 directly emit assembly code for control flow, but build and pass around expression trees for expressions.
- Many people have tried using a control flow graph with low-level, three address code for each basic block.



Intermediate representations

But, this isn't the whole story

Symbol table:

- identifiers, procedures
- size, type, location
- lexical nesting depth

Constant table:

- representation, type
- storage class, offset(s)

Storage map:

- storage layout
- overlap information
- (virtual) register assignments



- Many kinds of IR are used in practice.
- 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



Translating expressions

$$S \rightarrow \mathbf{id} = E ; \quad \{ \text{gen}(\text{top.get}(\mathbf{id.lexeme}) \text{'=' } E.\text{addr}); \}$$

$$E \rightarrow E_1 + E_2 \quad \{ E.\text{addr} = \mathbf{new Temp}(); \\ \text{gen}(E.\text{addr} \text{'=' } E_1.\text{addr} \text{'+' } E_2.\text{addr}); \}$$

$$\quad | - E_1 \quad \{ E.\text{addr} = \mathbf{new Temp}(); \\ \text{gen}(E.\text{addr} \text{'=' } \mathbf{minus} \text{' } E_1.\text{addr}); \}$$

$$\quad | (E_1) \quad \{ E.\text{addr} = E_1.\text{addr}; \}$$

$$\quad | \mathbf{id} \quad \{ E.\text{addr} = \text{top.get}(\mathbf{id.lexeme}); \}$$

- Builds the three-address code for an assignment statement.
- `addr` is an `synthetic-attribute` of `E`.
 - denotes the address that will hold the value of `E`.
- Constructs a three-address instruction and appends the instruction to the sequence of instructions.
- `top` is the top-most (current) symbol table.



Gap between HLL and IR

- High level languages may allow complexities that are not allowed in IR (such as expressions with multiple operators).
- High level languages have many syntactic constructs, not present in the IR (such as if-then-else or loops)

Challenges in translation:

- Deep nesting of constructs.
- Recursive grammars.
- We need a systematic approach to IR generation.

Goal:

- A HLL to IR translator.
- Input: A program in HLL.
- Output: A program in IR (may be an AST or program text)



IR generation for flow-of-control statements

$$P \rightarrow S \quad \left\{ \begin{array}{l} S.\text{next} = \text{newlabel}() \\ P.\text{code} = S.\text{code} \parallel \text{label}(S.\text{next}) \end{array} \right.$$

$$S \rightarrow \mathbf{assign} \quad \left\{ \begin{array}{l} S.\text{code} = \mathbf{assign}.\text{code} \end{array} \right.$$

$$S \rightarrow \mathbf{if} (B) S_1 \quad \left\{ \begin{array}{l} B.\text{true} = \text{newlabel}() \\ B.\text{false} = S_1.\text{next} = S.\text{next} \\ S.\text{code} = B.\text{code} \parallel \text{label}(B.\text{true}) \parallel S_1.\text{code} \end{array} \right.$$

$$S \rightarrow \mathbf{if} (B) S_1 \mathbf{else} S_2 \quad \left\{ \begin{array}{l} B.\text{true} = \text{newlabel}() \\ B.\text{false} = \text{newlabel}() \\ S_1.\text{next} = S_2.\text{next} = S.\text{next} \\ S.\text{code} = B.\text{code} \\ \quad \parallel \text{label}(B.\text{true}) \parallel S_1.\text{code} \\ \quad \parallel \text{gen}(\text{'goto' } S.\text{next}) \\ \quad \parallel \text{label}(B.\text{false}) \parallel S_2.\text{code} \end{array} \right.$$

- `code` is an `synthetic attribute`: giving the code for that node.
- Assume: `gen` only creates an instruction.
- `||` concatenates the code.



IR generation for flow-of-control statements

```

S → while ( B ) S1
    begin = newlabel()
    B.true = newlabel()
    B.false = S.next
    S1.next = begin
    S.code = label(begin) || B.code
             || label(B.true) || S1.code
             || gen('goto' begin)

S → S1 S2
    S1.next = newlabel()
    S2.next = S.next
    S.code = S1.code || label(S1.next) || S2.code
    
```

- *code* is a synthetic attribute: giving the code for that node.
- Assume: *gen* only creates an instruction.
- `||` concatenates the code.



IR generation for boolean expressions

```

B → B1 || B2
    B1.true = B.true
    B1.false = newlabel()
    B2.true = B.true
    B2.false = B.false
    B.code = B1.code || label(B1.false) || B2.code

B → B1 && B2
    B1.true = newlabel()
    B1.false = B.false
    B2.true = B.true
    B2.false = B.false
    B.code = B1.code || label(B1.true) || B2.code

B → ! B1
    B1.true = B.false
    B1.false = B.true
    B.code = B1.code

B → E1 rel E2
    B.code = E1.code || E2.code
             || gen('if' E1.addr rel.op E2.addr 'goto' B.true)
             || gen('goto' B.false)

B → true
    B.code = gen('goto' B.true)

B → false
    B.code = gen('goto' B.false)
    
```



Array elements dereference (Recall)

- Elements are typically stored in a block of consecutive locations.
- If the width of each array element is *w*, then the *i*th element of array *A* (say, starting at the address *base*), begins at the location: *base + i × w*.
- For multi-dimensions, beginning address of *A*[*i*₁][*i*₂] is calculated by the formula:
 $base + i_1 \times w_1 + i_2 \times w_2$
 where, *w*₁ is the width of the row, and *w*₂ is the width of one element.
- We declare arrays by the number of elements (*n*_{*j*} is the size of the *j*th dimension) and the width of each element in an array is fixed (say *w*).
 The location for *A*[*i*₁][*i*₂] is given by
 $base + (i_1 \times n_2 + i_2) \times w$
- Q: If the array index does not start at '0', then ?
- Q: What if the data is stored in column-major form?



Translation of Array references

- Extending the expression grammar with arrays:

```

S → id = E ;
L → id [ E ]
E → E1 + E2
L → id
L → L1 [ E ]
L → L
    
```



Translation of Array references (contd)

```
S → id = E ; { gen( top.get(id.lexeme) != E.addr); }
  | L = E ; { gen(L.addr.base '[' L.addr ']' != E.addr); }
E → E1 + E2 { E.addr = new Temp();
                  gen(E.addr != E1.addr '+' E2.addr); }
  | id          { E.addr = top.get(id.lexeme); }
  | L          { E.addr = new Temp();
                  gen(E.addr != L.array.base '[' L.addr ']); }
```

Nonterminal L has three synthesized attributes

- 1 $L.addr$ denotes a temporary that is used while computing the offset for the array reference.
- 2 $L.array$ is a pointer to the ST entry for the array name. The field $base$ gives the actual l-value of the array reference.



Translation of Array references (contd)

```
L → id [ E ] { L.array = top.get(id.lexeme);
               L.type = L.array.type.elem;
               L.addr = new Temp();
               gen(L.addr != E.addr '*' L.type.width); }
  | L1 [ E ] { L.array = L1.array;
               L.type = L1.type.elem;
               t = new Temp();
               L.addr = new Temp();
               gen(t != E.addr '*' L.type.width); }
               gen(L.addr != L1.addr '+' t); }
```

- 3 $L.type$ is the type of the subarray generated by L .
 - For any type t : $t.width$ gives get the width of the type.
 - For any type t : $t.elem$ gives the element type.



Translation of Array references (contd)

Example:

- Let a denotes a 2×3 integer array.
- Type of a is given by $array(2, array(3, integer))$
- Width of $a = 24$ (size of $integer = 4$).
- Type of $a[i]$ is $array(3, integer)$, width = 12.
- Type of $a[i][j] = integer$

Exercise:

- Write three address code for $c + a[i][j]$

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```



Q: What if we did not know the size of $integer$ (machine dependent)?

Some challenges/questions

- Avoiding redundant gotos. ??
- Multiple passes. ??
- How to translate implicit branches: `break` and `continue`?
- How to translate `switch` statements efficiently?
- How to translate procedure code?



What have we done in last few classes?

- Intermediate Code Generation.

To read

- Dragon Book. Sections 6.4, 6.5, 6.6, 6.7, 6.8, 6.9 and 2.8

