CS3300 - Compiler Design

Semantic Analysis - IR Generation

V. Krishna Nandivada

IIT Madras

Acknowledgement

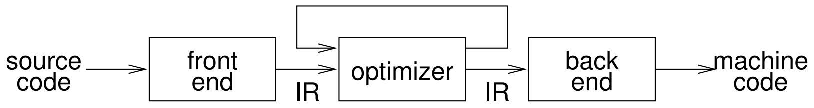
Intermediate representations

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 (myth)
4. enables machine-independent optimization
   – general techniques, multiple passes

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 ← a[i,j+2]      t1 ← j + 2        r1 ← [fp-4]
                   t2 ← i * 20       r2 ← r1 + 2
                   t3 ← t1 + t2      r3 ← [fp-8]
                   t4 ← 4 * t3       r4 ← r3 * 20
                   t5 ← addr a       r5 ← r4 + r2
                   t6 ← t5 + t4      r6 ← 4 * r5
                   t7 ← *t6          r7 ← fp - 216
                                     f1 ← [r7+r6]
(a)                (b)               (c)
```

(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.

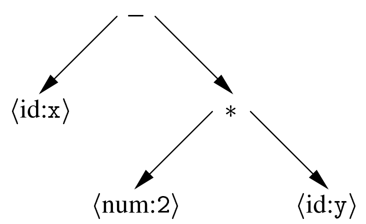
Intermediate representations

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.

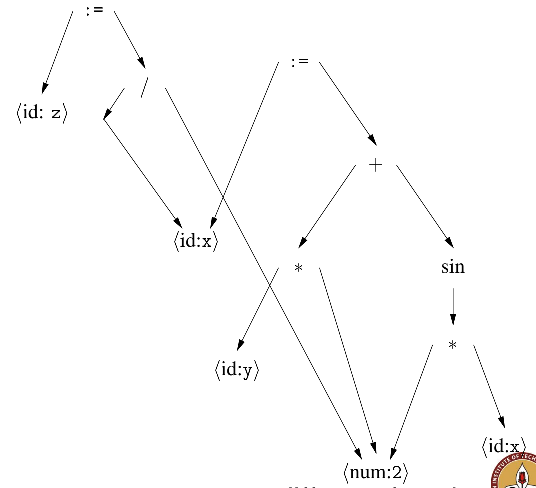


This represents "`x − 2 ∗ y`".
For ease of manipulation, can use a linearized (operator) form of the tree.
e.g., in postfix form: `x 2 y ∗ −`

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
```



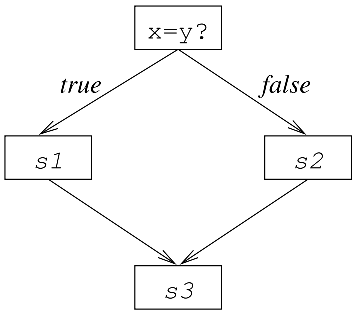
Q: What to do for matching names present across different functions?

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

```
if (x=y) then
   s1
else
   s2
s3
```



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 ← y op z`
  with a single operator and, at most, three names.
  Simpler form of expression:
  `x - 2 * y`
  becomes
  `t1 ← 2 * y`
  `t2 ← 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

Typical instructions types include:

1. assignments `x ← y op z`
2. assignments `x ← op y`
3. assignments `x ← y[i]`
4. assignments `x ← y`
5. branches `goto L`
6. conditional branches
   `if x goto L`
7. procedure calls
   `param` x_1, `param` x_2, ...`param` x_n
   and
   `call p, n`
8. address and pointer assignments

How to translate:

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

?

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 - 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)  := (3) a
(5)  * 3 i
(6)  := (5) d
(7)  + 1 i
(8)  := (7) i
(9)  LE i 10
(10) IFT goto (2)
```

Execution Order (a) : 1 2 3 4 5 6 7 8 9 10
Execution Order (b) : 3 4 1 2 5 6 7 8 9 10

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

Advice

- 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

Gap between HLL and IR

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)

Translating expressions

```
S -> id = E;   {gen(top.get(id.lexeme) '=' E.addr);}

E -> E1 + E2   {E.addr = new Temp();
                gen(E.addr '=' E1.addr '+' E2.addr);}

     | - E1    {E.addr = new Temp();
                gen(E.addr '=' - E2.addr);}

     | (E1)    {E.addr = E1.addr;}

     | id      {E.addr = top.get(id.lexeme);}
```

- Builds the three-address code for an assignment statement.
- addr: a `synthesized`-attr of E – denotes the address holding the val of E.
- Constructs a three-address instruction and appends the instruction to the sequence of instructions.
- *top* is the top-most (current) symbol table.

IR generation for flow-of-control statements

$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$

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

IR generation for flow-of-control statements

$S \rightarrow$ **while** $(B) \; S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \;\|\|\; B.code$ $\|\|\; label(B.true) \;\|\|\; S_1.code$ $\|\|\; gen('goto'\; begin)$
$S \rightarrow S_1 \; S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \;\|\|\; label(S_1.next) \;\|\|\; S_2.code$

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

IR generation for boolean expressions

$B \rightarrow B_1 \;\|\|\; B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.false) \;\|\|\; B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.true) \;\|\|\; B_2.code$
$B \rightarrow \; ! \; B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \;\|\|\; E_2.code$ $\|\|\; gen('if'\; E_1.addr\; \mathbf{rel}.op\; E_2.addr\; 'goto'\; B.true)$ $\|\|\; gen('goto'\; B.false)$
$B \rightarrow$ **true**	$B.code = gen('goto'\; B.true)$
$B \rightarrow$ **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: $base + i \times w$.
- 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:

$$
\begin{array}{lcl}
S & \rightarrow & \mathbf{id} = E \; ; \\
 & | & L = E \; ; \\
E & \rightarrow & E_1 + E_2 \\
 & | & \mathbf{id} \\
 & | & L
\end{array}
\qquad
\begin{array}{lcl}
L & \rightarrow & \mathbf{id} \; [\; E \;] \\
 & | & L_1 \; [\; E \;]
\end{array}
$$

Translation of Array references (contd)

$$
\begin{array}{llll}
S & \rightarrow & \textbf{id} = E\ ; & \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \} \\
 & | & L = E\ ; & \{\ gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \} \\
E & \rightarrow & E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp\,();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
 & | & \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \} \\
 & | & L & \{\ E.addr = \textbf{new}\ Temp\,();\ gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \}
\end{array}
$$

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)

$$
\begin{array}{llll}
L & \rightarrow & \textbf{id}\ [\ E\ ] & \{\ L.array = top.get(\textbf{id}.lexeme);\ L.type = L.array.type.elem;\ L.addr = \textbf{new}\ Temp\,();\ gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
 & | & L_1\ [\ E\ ] & \{\ L.array = L_1.array;\ L.type = L_1.type.elem;\ t = \textbf{new}\ Temp\,();\ L.addr = \textbf{new}\ Temp\,();\ gen(t\ '='\ E.addr\ '*'\ L.type.width);\ \} \ gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}
\end{array}
$$

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 adddress 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?

Closing remarks

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