IR Generation

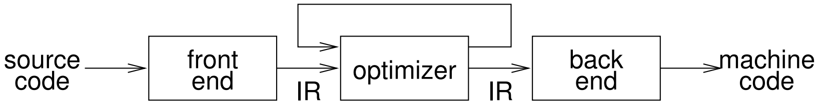
Intermediate representations

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

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
- 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

(a)	(b)	(c)

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

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

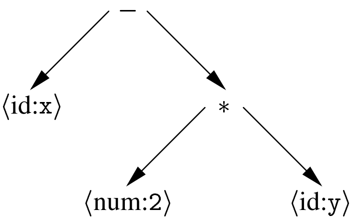
Intermediate representations

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 just the parse tree with the nodes for most non-terminal symbols removed.



This represents "$\texttt{x} - \texttt{2} * \texttt{y}$".

SDD for generating AST: Example-1

$E \to E + T$	
$\mid E - T$	
$\mid T$	
$T \to T * F$	
$\mid F$	
$F \to \texttt{id}$	

SDD for generating AST: Example-1

$E_1 \rightarrow E_2 + T$	$E_1.node = \texttt{new Node}(\texttt{+}, E_2.node, T.node)$
$\mid E_2 - T$	$E_1.node = \texttt{new Node}(\texttt{-}, E_2.node, T.node)$
$\mid T$	$E_1.node = T.node$
$T_1 \rightarrow T_2 * F$	$T_1.node = \texttt{new Node}(\star, T_2.node, F.node)$
$\mid F$	$T_1.node = F.node$
$F \rightarrow \texttt{id}$	$F.node = \texttt{new Leaf}(\texttt{id}.lexeme)$

SDD for generating AST: Example-2

$E \rightarrow TE'$	
$E'_1 \rightarrow +TE'_2$	
$\mid \varepsilon$	
$T \rightarrow FT'$	
$T'_1 \rightarrow *FT'_2$	
$\mid \varepsilon$	
$F \rightarrow \texttt{id}$	

SDD for generating AST: Example-2

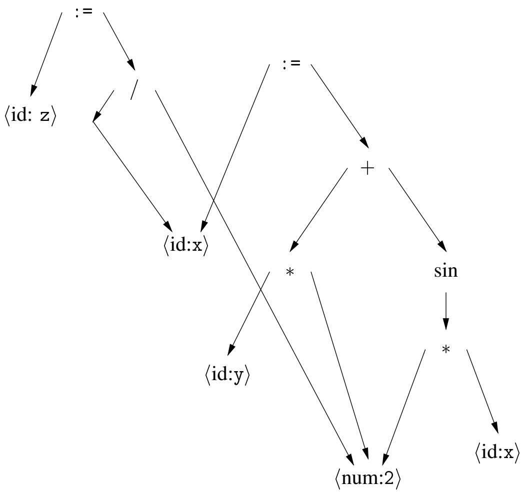
$E \rightarrow TE'$	$E.node = E'.node$
	$E'.in = T.node$
$E'_1 \rightarrow +TE'_2$	$E'_1.node = E'_2.node$
	$E'_2.in = \texttt{new Node}(\texttt{+}, E'_1.in, T.node)$
$\mid \varepsilon$	$E'_1.node = E'_1.in$
$T \rightarrow FT'$	$T.node = T'.node$
	$T'.in = F.node$
$T'_1 \rightarrow *FT'_2$	$T'_1.node = T'_2.node$
	$T'_2.in = \texttt{new Node}(\star, T'_1.in, F.node)$
$\mid \varepsilon$	$T'_1.node = T'_1.in$
$F \rightarrow \texttt{id}$	$F.node = \texttt{new Leaf}(\texttt{id}.lexeme)$

Homework: Draw the dependency graph for `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
```



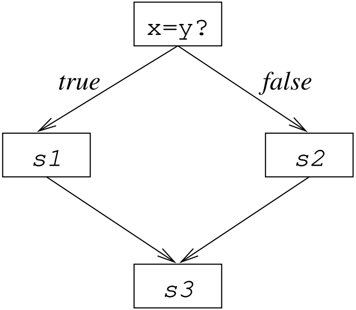
Value-number methods for constructing DAGs:
(1) Assign a unique number to every node.
(2) Search for a node with signature $(op, valLeft, valRight)$.

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

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.

A commonly used hybrid IR is 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

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;   {S.code = E.code ||
               gen(top.get(id.lexeme) '=' E.addr);}
E -> E1 + E2   {E.addr = new Temp();
               E.code = E1.code || E2.code ||
                gen(E.addr '=' E1.addr '+' E2.addr);}
   | - E1      {E.addr = new Temp();
               E.code = E1.code ||
                gen(E.addr '=' - E2.addr);}
   | (E1)      {E.addr = E1.addr;
               E.code = E1.code}
   | id        {E.addr = top.get(id.lexeme);
               E.code = ' '}
```

- addr: a `synthesized`-attribute of E – denotes the address holding the val of E.
- *top* is the top-most (current) symbol table.

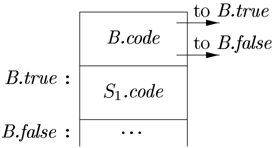
Translating expressions: Incremental Translation

```
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);}
```

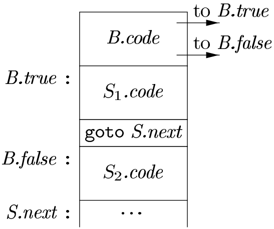
- Constructs a three-address instruction and appends the instruction to the sequence of instructions.

IR generation for flow-of-control statements

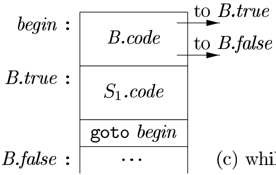
$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if } (B)\, S_1 \\
S & \rightarrow & \texttt{if } (B)\, S_1 \texttt{ else } S_2 \\
S & \rightarrow & \texttt{while } (B)\, S_1
\end{array}
$$



(a) if



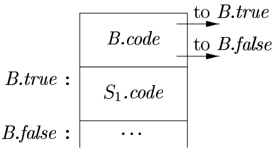
(b) if-else



(c) while

IR generation for `if`-statement

`code`	Code for the node; synthesized
`next`	Label of the next instruction; inherited
`true`	Label of next instruction if boolean expression evaluates to *true*; inherited
`false`	Label of next instruction if boolean expression evaluates to *false*; inherited

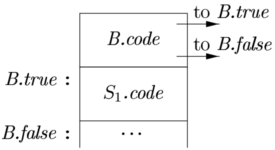


(a) if

```
P->S            {S.next = new Label();
                P.code = S.code || label(S.next)}

S->assign       {S.code = assign.code}

S->if (B) S1
```

IR generation for `if`-statement

`code`	Code for the node; synthesized
`next`	Label of the next instruction; inherited
`true`	Label of next instruction if boolean expression evaluates to *true*; inherited
`false`	Label of next instruction if boolean expression evaluates to *false*; inherited

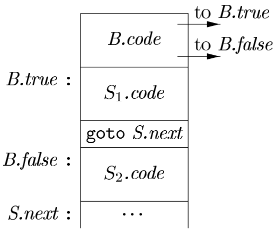


(a) if

```
P->S            {S.next = new Label();
                 P.code = S.code || label(S.next)}

S->assign       {S.code = assign.code}

S->if (B) S1    {B.true = new Label();
                 B.false = S1.next = S.next
                 S.code = B.code || label(B.true) || S1.code}
```

IR generation for `if else`-statement

`code`	Code for the node; synthesized
`next`	Label of the next instruction; inherited
`true`	Label of next instruction if boolean expression evaluates to *true*; inherited
`false`	Label of next instruction if boolean expression evaluates to *false*; inherited

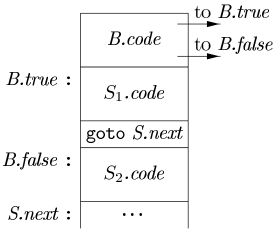


(b) if-else

```
S->if (B) S1
   else S2
```

IR generation for `if else`-statement

`code`	Code for the node; synthesized
`next`	Label of the next instruction; inherited
`true`	Label of next instruction if boolean expression evaluates to *true*; inherited
`false`	Label of next instruction if boolean expression evaluates to *false*; inherited

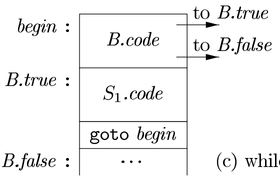


(b) if-else

```
S->if (B) S1   {B.true = new Label();
   else S2      B.false = new Label();
                S1.next = S2.next = S.next;
                S.code = B.code || label(B.true) || S1.code
                         || gen ('goto' S.next)
                         || label (B.false) || S2.code}
```

IR generation for `while`-statement

`code`	Code for the node; synthesized
`next`	Label of the next instruction; inherited
`true`	Label of next instruction if boolean expression evaluates to *true*; inherited
`false`	Label of next instruction if boolean expression evaluates to *false*; inherited

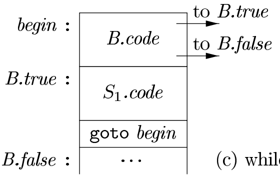


(c) while

```
S->while(B) S1
```

IR generation for `while`-statement

`code`	Code for the node; synthesized
`next`	Label of the next instruction; inherited
`true`	Label of next instruction if boolean expression evaluates to *true*; inherited
`false`	Label of next instruction if boolean expression evaluates to *false*; inherited



(c) while

```
S->while(B) S1      {begin = new Label();
                     B.true = new Label();
                     B.false = S.next
                     S1.next = begin
                     S.code = label(begin) || B.code
                              || label(B.true) || S1.code
                              || gen('goto' begin)}
```

IR generation for boolean expressions

$$\begin{array}{rcl} B & \rightarrow & B \mid\mid B \mid B \;\&\&\; B \mid !B \mid (B) \\ & & \mid E \textbf{ rel } E \mid \textbf{true} \mid \textbf{false} \end{array}$$

- For $B \rightarrow B_1 \mid\mid B_2$, if B_1 evaluates to **true**, then B_2 should not be evaluated, and control should transfer to B.`true`.
  - If B_1 evaluates to **false**, then B_2 should be evaluated.
- For $B \rightarrow B_1 \;\&\&\; B_2$, if B_1 evaluates to **false**, then B_2 should not be evaluated, and control should transfer to B.`false`,
  - If B_1 evaluates to **true**, then B_2 should be evaluated.
- **rel** corresponds to comparison operators: $<, \leq, ==, !=, >, \geq$.

IR generation for boolean expressions

```
B -> E1 rel E2
```

```
B -> true
```

```
B -> false
```

IR generation for boolean expressions

```
B -> E1 rel E2      t = new Temp()
                    B.code=E1.code||E2.code
                     || gen(t'='E1.addr rel.op E2.addr)
                     || gen('if' t 'goto' B.true)
                     || gen('goto' B.false);

B -> true           B.code = gen('goto' B.true)

B -> false          B.code = gen('goto' B.false)
```

IR generation for boolean expressions

```
B -> B1 || B2
```

```
B -> B1 && B2
```

```
B -> !B1
```

IR generation for boolean expressions

```
B -> B1 || B2           B1.true = B.true
                        B1.false = new Label()
                        B2.true = B. true
                        B2.false = B.false
                        B.code = B1.code || label(B1.false) || B2.code

B -> B1 && B2           B1.true = new Label()
                        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
```

IR generation for control-flow statements: Example

HLL Code:

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

IR Code:

```
    if x < 100 goto L1
    goto L3
L3: if x > 200 goto L4
    goto L2
L4: if x != y goto L1
    goto L2
L1: x = 0
L2:
```

An optimization to avoid redundant `goto`s

- Consider the HLL code: `if (x > 100) x = 0;`
- Using the previous SDD, we get the following IR:

```
    if x > 100 goto L1
    goto L2
L1: x = 0
L2:
```

- But this is equivalent to :

```
    ifFalse x > 100 goto L2
    x = 0
L2:
```

- Similarly, while translating `B1 || B2`, since code for `B2` immediately follows code for `B1`, there is no need of an unconditional jump for `B1.false`.
- To avoid redundant `goto`s, we can use a special `fall` label.
  - It indicates that the instruction to be executed immediately follows.

An optimization to avoid redundant `goto`s

```
S->if (B) S1         {B.true = fall;
                      B.false = S1.next = S.next
                      S.code = B.code || S1.code}

B->E1 rel E2
```

An optimization to avoid redundant `goto`s

```
S->if (B) S1         {B.true = fall;
                     B.false = S1.next = S.next
                     S.code = B.code || S1.code}

B->E1 rel E2         {t = new Temp()
                     B.code=E1.code||E2.code
                      || gen(t'='E1.addr rel.op E2.addr)
                      || if (B.true=fall)
                           gen('ifFalse' t 'goto' B.false)
                         else if (B.false=fall)
                           gen('if' t 'goto' B.true)
                         else
                           gen ('if' t 'goto' B.true)
                      || gen('goto' B.false)   }
```

Homework: Use `fall` to optimize IRs for `B1 || B2` and `B1 && B2`

One-Pass Code generation using Backpatching

- While generating code for `if (B) S`, code for `B` is generated before code for `S`.
- However, code for `B` contains a jump for instruction after `S`.
  - The target address for this jump can therefore only be known after generating code for `S`.
  - This requires a second pass to fill up the jump addresses.
- In Backpatching, instead of having jump targets as inherited attributes of boolean expressions, jump instructions with holes become synthesized attributes of branch statements.
  - Self-reading exercise: Chapter 6, Section 6.7
  - Part of the syllabus.

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:

$$base + i \times w$$

- 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

$$base + (i_1 \times n_2 + i_2) \times w$$

- 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.

Array elements dereference (Recall)

- 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:

$$base + i \times w$$

- Question: If the array index does not start at '0', at some 'k' then, how to calculate the address of $A[i]$?

- Homework: What if the data is stored in column-major form?

Translation of Array references

- Extending the expression grammar with arrays:

```
S -> id = E;
   | L = E;
E -> E1 + E2
   | id
   | L

L -> id [E]
   | L1 [E]
```

Translation of Array references (cont.)

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

    | L = E;  {gen(L.array.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)

Type and width:

- 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 $a[i][j] = integer$

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 adddress code for $c + a[i][j]$

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

Some challenges/questions

- How to translate implicit branches: `break` and `continue`?
- How to translate `switch` statements efficiently?
- How to translate procedure code?

Self-reading exercise: Dragon Book, Chapter 6, Sections 6.7,6.8,6.9