

Unit-4

Intermediate Code Generation

Mrs. Soma Ghosh
gsn.comp@coeptech.ac.in

Topics

Intermediate Code generation

What is intermediate code?

During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as **intermediate code**



intermediate code

The following are commonly used intermediate code representation :

- Syntax tree
- Postfix Notation
- Three-Address Code

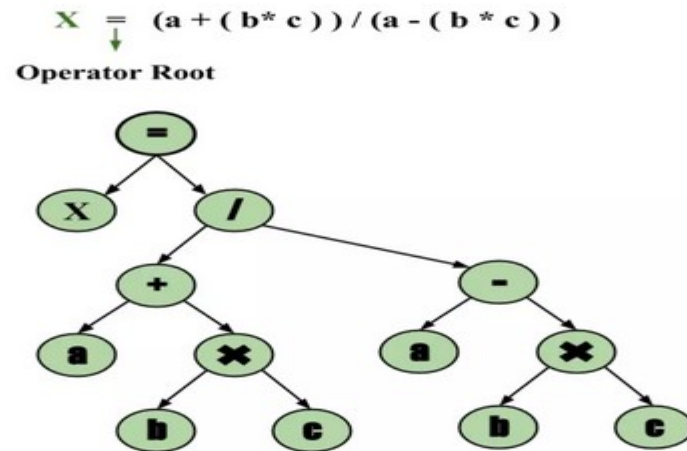
Syntax tree

Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Syntax tree

Example –

$$x = (a + b * c) / (a - b * c)$$



SDD for creating DAG's

- | Production | Semantic Rules |
|---------------------------|--------------------------------------------------|
| 1) $E \rightarrow E1 + T$ | $E.node = \text{new Node}('+', E1.node, T.node)$ |
| 2) $E \rightarrow E1 - T$ | $E.node = \text{new Node}('-', E1.node, T.node)$ |
| 3) $E \rightarrow T$ | $E.node = T.node$ |
| 4) $T \rightarrow (E)$ | $T.node = E.node$ |
| 5) $T \rightarrow id$ | $T.node = \text{new Leaf}(id, id.entry)$ |
| 6) $T \rightarrow num$ | $T.node = \text{new Leaf}(num, num.val)$ |

Postfix Notation

- The ordinary (infix) way of writing the sum of a and b is with operator in the middle : $a + b$
- The postfix notation for the same expression places the operator at the right end as $ab +$. In general, if $e1$ and $e2$ are any postfix expressions, and $+$ is any binary operator, the result of applying $+$ to the values denoted by $e1$ and $e2$ is postfix notation by $e1e2 +$. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

Postfix Notation

Example – The postfix representation of the expression $(a - b) * (c + d) + (a - b)$ is

$ab - cd + ab - +^* .$

Three-Address Code

A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form $x = y \text{ op } z$, here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.

For Example : $a = b + c * d$;

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;  
r2 = b + r1;  
a = r2
```

Three-Address Code

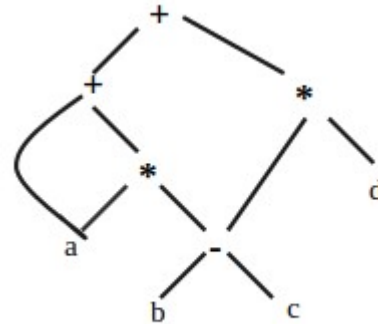
A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms :

- ❖ Quadruples
- ❖ Triples
- ❖ Indirect Triples

Three address code

- In a three address code there is at most one operator at the right side of an instruction
- Example: $a + a * (b - c) + (b - c) * d$
-

$t1 = b - c$
 $t2 = a * t1$
 $t3 = a + t2$
 $t4 = t1 * d$
 $t5 = t3 + t4$



Example

- do $i = i + 1$; while ($a[i] < v$);

```
L:      t1 = i + 1
        i = t1
        t2 = i * 8
        t3 = a[t2]
        if t3 < v goto L
```

Symbolic labels

```
100:    t1 = i + 1
101:    i = t1
102:    t2 = i * 8
103:    t3 = a[t2]
104:    if t3 < v goto 100
```

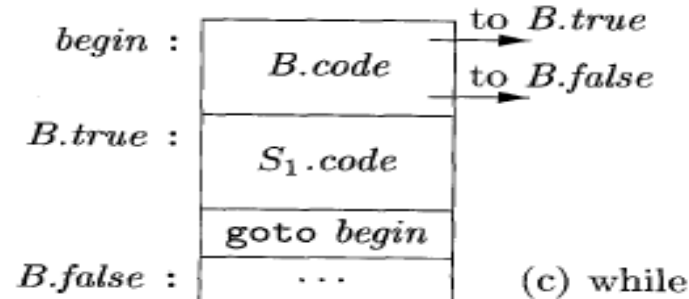
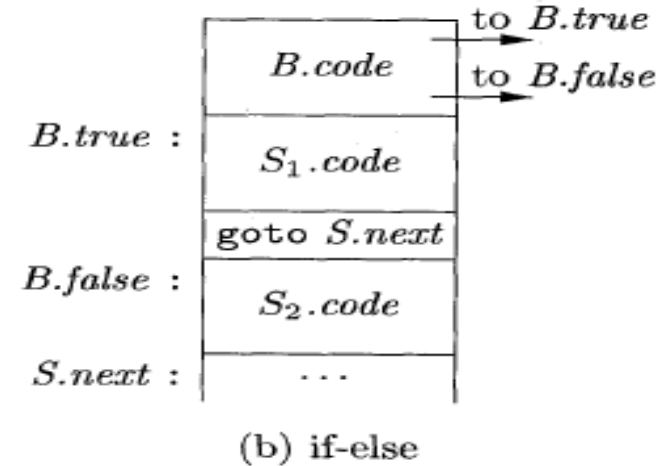
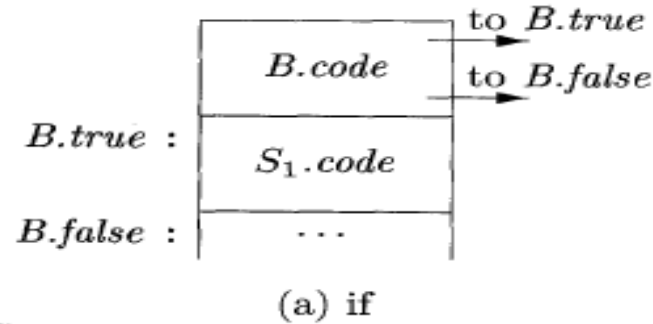
Position numbers

Short-Circuit Code

- `if (x < 100 || x > 200 && x != y) x = 0;`
- - `if x < 100 goto L2`
 - `ifFalse x > 200 goto L1`
 - `ifFalse x != y goto L1`
 - `L2: x = 0`
 - `L1:`

Flow-of-Control Statements

$S \rightarrow \text{if } (B) S_1$
 $S \rightarrow \text{if } (B) S_1 \text{ else } S_2$
 $S \rightarrow \text{while } (B) S_1$



Syntax-directed definition

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \text{assign}$	$S.code = \text{assign}.code$
$S \rightarrow \text{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$
$S \rightarrow \text{if} (B) S_1 \text{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen('goto' S.next)$ $\parallel label(B.false) \parallel S_2.code$
$S \rightarrow \text{while} (B) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen('goto' begin)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

Generating three-address code for booleans

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \parallel B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \parallel label(B_1.false) \parallel B_2.code$
$B \rightarrow B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \parallel label(B_1.true) \parallel B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \text{ rel } E_2$	$B.code = E_1.code \parallel E_2.code$ $\parallel gen('if' E_1.addr \text{ rel } op E_2.addr 'goto' B.true)$ $\parallel gen('goto' B.false)$
$B \rightarrow \text{true}$	$B.code = gen('goto' B.true)$
$B \rightarrow \text{false}$	$B.code = gen('goto' B.false)$

translation of a simple if-statement

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

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

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The example is represented below in quadruples format:

```
r1 = c * d;  
r2 = b + r1;  
a = r2
```

Op	arg ₁	arg ₂	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2. The results of respective sub-expressions are denoted by the position of expression.

```
r1 = c * d;  
r2 = b + r1;  
a = r2
```

Op	arg ₁	arg ₂
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Backpatching

- Previous codes for Boolean expressions insert symbolic labels for jumps
- It therefore needs a separate pass to set them to appropriate addresses
- We can use a technique named backpatching to avoid this
- We assume we save instructions into an array and labels will be indices in the array
- For nonterminal B we use two attributes B.truelist and B.falselist together with following functions:
 - makelist(i): create a new list containing only I, an index into the array of instructions
 - Merge(p1,p2): concatenates the lists pointed by p1 and p2 and returns a pointer to the concatenated list
 - Backpatch(p,i): inserts i as the target label for each of the instruction on the list pointed to by p

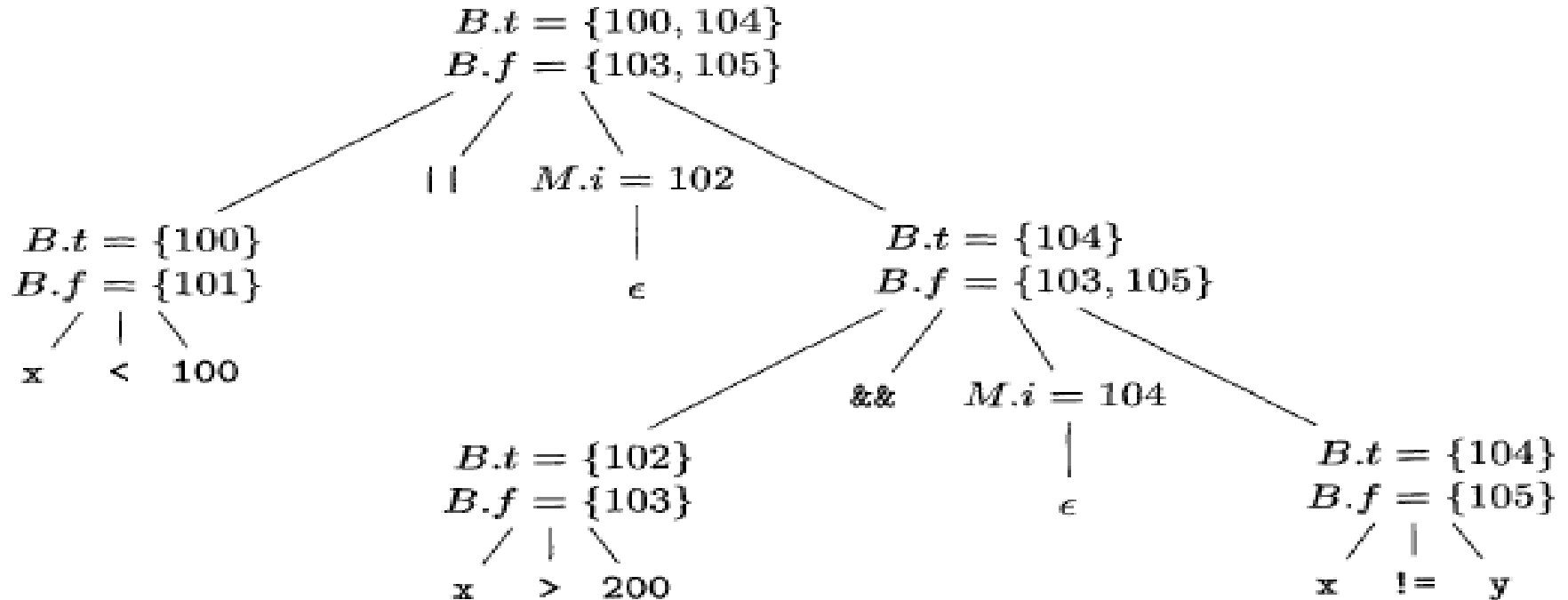
Backpatching for Boolean Expressions

- $B \rightarrow B_1 \ || \ M \ B_2 \mid B_1 \ \&\& \ M \ B_2 \mid ! \ B_1 \mid (\ B_1 \) \mid E_1 \ \mathbf{rel} \ E_2 \mid \mathbf{true} \mid \mathbf{false}$
 $M \rightarrow \epsilon$

- 1) $B \rightarrow B_1 \ || \ M \ B_2$ { *backpatch*(B_1 .*false*list, M .*instr*);
 B .*true*list = *merge*(B_1 .*true*list, B_2 .*true*list);
 B .*false*list = B_2 .*false*list; }
- 2) $B \rightarrow B_1 \ \&\& \ M \ B_2$ { *backpatch*(B_1 .*true*list, M .*instr*);
 B .*true*list = B_2 .*true*list;
 B .*false*list = *merge*(B_1 .*false*list, B_2 .*false*list); }
- 3) $B \rightarrow ! \ B_1$ { B .*true*list = B_1 .*false*list;
 B .*false*list = B_1 .*true*list; }
- 4) $B \rightarrow (\ B_1 \)$ { B .*true*list = B_1 .*true*list;
 B .*false*list = B_1 .*false*list; }
- 5) $B \rightarrow E_1 \ \mathbf{rel} \ E_2$ { B .*true*list = *makelist*(*nextinstr*);
 B .*false*list = *makelist*(*nextinstr* + 1);
emit('if' E_1 .*addr* *rel* E_2 .*addr* 'goto -');
emit('goto -'); }
- 6) $B \rightarrow \mathbf{true}$ { B .*true*list = *makelist*(*nextinstr*);
emit('goto -'); }
- 7) $B \rightarrow \mathbf{false}$ { B .*false*list = *makelist*(*nextinstr*);
emit('goto -'); }
- 8) $M \rightarrow \epsilon$ { M .*instr* = *nextinstr*; }

Backpatching for Boolean Expressions

- Annotated parse tree for $x < 100 \parallel x > 200 \&\& x \neq y$



Flow-of-Control Statements

- 1) $S \rightarrow \text{if}(B) M S_1$ { *backpatch*(*B.truelist*, *M.instr*);
 S.nextlist = *merge*(*B.falselist*, *S₁.nextlist*); }
- 2) $S \rightarrow \text{if}(B) M_1 S_1 N \text{ else } M_2 S_2$
 { *backpatch*(*B.truelist*, *M₁.instr*);
 backpatch(*B.falselist*, *M₂.instr*);
 temp = *merge*(*S₁.nextlist*, *N.nextlist*);
 S.nextlist = *merge*(*temp*, *S₂.nextlist*); }
- 3) $S \rightarrow \text{while } M_1 (B) M_2 S_1$
 { *backpatch*(*S₁.nextlist*, *M₁.instr*);
 backpatch(*B.truelist*, *M₂.instr*);
 S.nextlist = *B.falselist*;
 emit('goto' *M₁.instr*); }
- 4) $S \rightarrow \{ L \}$ { *S.nextlist* = *L.nextlist*; }
- 5) $S \rightarrow A ;$ { *S.nextlist* = **null**; }
- 6) $M \rightarrow \epsilon$ { *M.instr* = *nextinstr*; }
- 7) $N \rightarrow \epsilon$ { *N.nextlist* = *makelist*(*nextinstr*);
 emit('goto -'); }
- 8) $L \rightarrow L_1 M S$ { *backpatch*(*L₁.nextlist*, *M.instr*);
 L.nextlist = *S.nextlist*; }
- 9) $L \rightarrow S$ { *L.nextlist* = *S.nextlist*; }

Translation of a switch-statement

		code to evaluate E into t	
		goto test	code to evaluate E into t
	$L_1:$	code for S_1	if $t \neq V_1$ goto L_1
		goto next	code for S_1
	$L_2:$	code for S_2	goto next
		goto next	$L_1:$ if $t \neq V_2$ goto L_2
		...	code for S_2
	$L_{n-1}:$	code for S_{n-1}	goto next
		goto next	$L_2:$...
	$L_n:$	code for S_n	$L_{n-2}:$ if $t \neq V_{n-1}$ goto L_{n-1}
		goto next	code for S_{n-1}
switch (E) {	test:	if $t = V_1$ goto L_1	goto next
case $V_1: S_1$		if $t = V_2$ goto L_2	$L_{n-1}:$ code for S_n
case $V_2: S_2$...	next:
...			
case $V_{n-1}: S_{n-1}$		if $t = V_{n-1}$ goto L_{n-1}	
default: S_n		goto L_n	
}	next:		

