Unit-4
Intermediate Code Generation

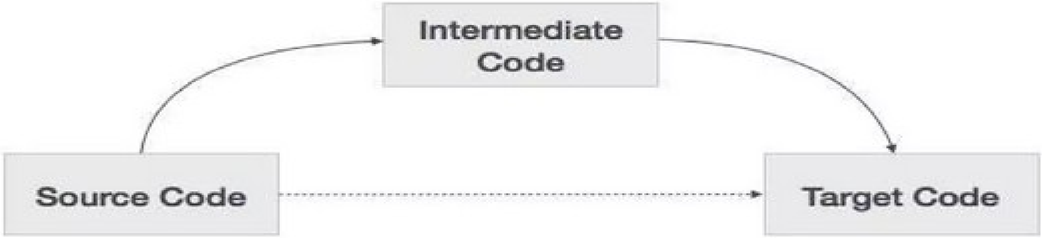
Mrs. Soma Ghosh
gsn.comp@coeptech.ac.in

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

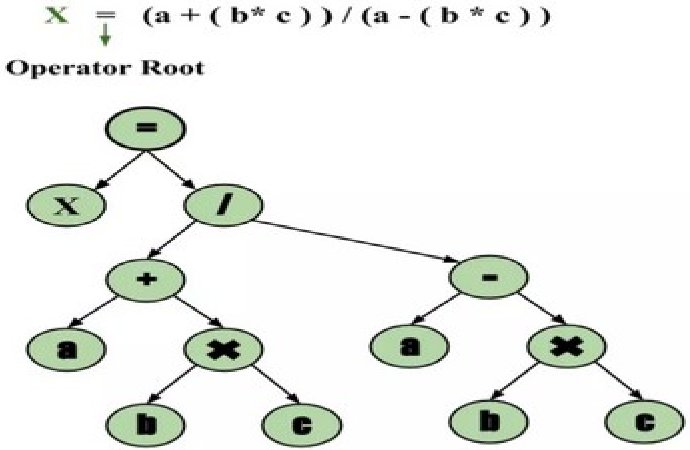
Syntax tree

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 -> E1+T	E.node= new Node(‘+’, E1.node,T.node)
2) E -> E1-T	E.node= new Node(‘-’, E1.node,T.node)
3) E -> T	E.node = T.node
4) T -> (E)	T.node = E.node
5) T -> id	T.node = new Leaf(id, id.entry)
6) T -> num	T.node = 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 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

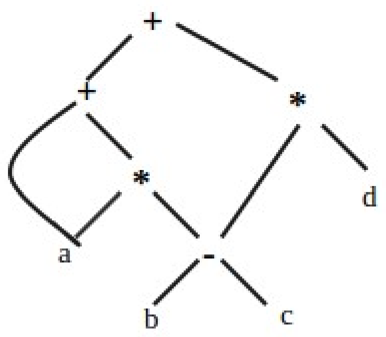
A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms :

- 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



https://chatgpt.com/c/67060859-6e60-8001-b762-7f3c70681777
sahhildeshpannde

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;`

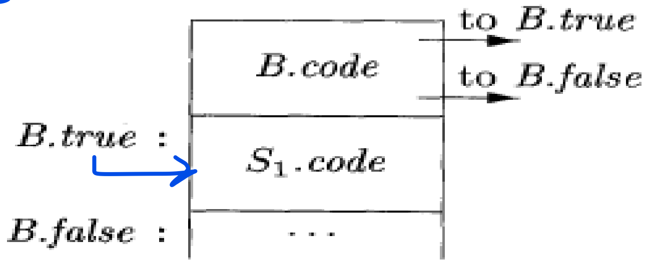
-
```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

→ check if x>200 is False

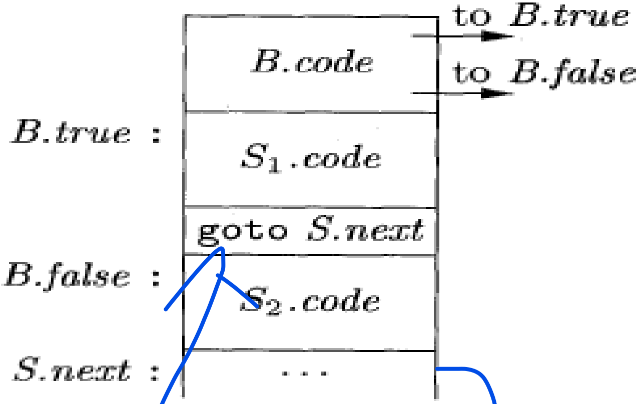
Flow-of-Control Statements

if+else

$$
\begin{aligned}
S &\rightarrow \textbf{if}\ (\ B\ )\ S_1 \\
S &\rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2 \\
S &\rightarrow \textbf{while}\ (\ B\ )\ S_1
\end{aligned}
$$

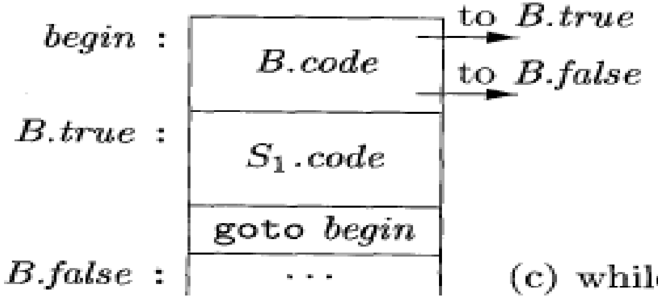


(a) if



(b) if-else

exit



(c) while

Syntax-directed definition

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

bahar aana cause B ke bad do S1

Generating three-address code for booleans

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

https://gemini.google.com/app/dd4c0c20a2f6554a?hl=en-IN

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

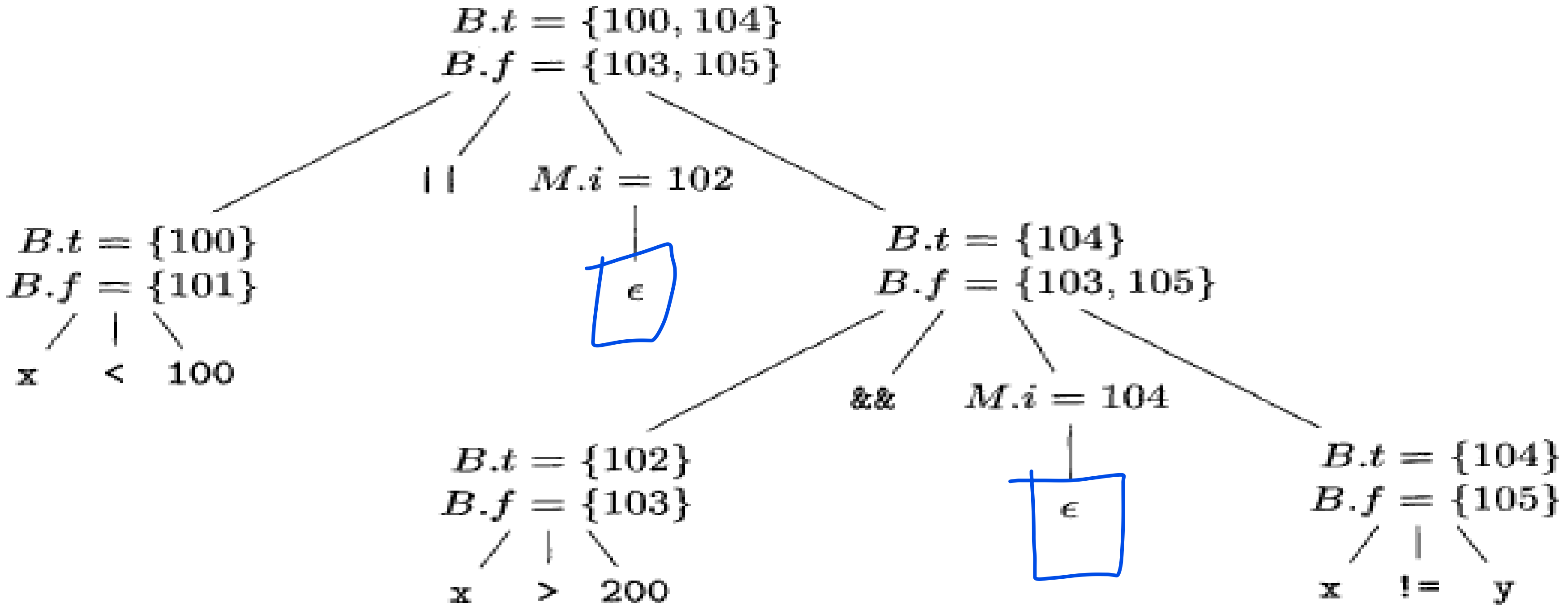
- $B \rightarrow B_1 \text{ || } M\ B_2 \mid B_1 \text{ \&\& } M\ B_2 \mid \text{ ! } B_1 \mid (\ B_1\ ) \mid E_1 \textbf{ rel } E_2 \mid \textbf{true} \mid \textbf{false}$
  $M \rightarrow \epsilon$

B1

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

Backpatching for Boolean Expressions

- Annotated parse tree for x < 100 || x > 200 && x ! = y



Flow-of-Control Statements

1) $S \rightarrow \textbf{if} \, (\, B \,) \, M \, S_1$ $\{ \; backpatch(B.truelist, \; M.instr);$
$S.nextlist \; = \; merge(B.falselist, \; S_1.nextlist); \; \}$

2) $S \rightarrow \textbf{if} \, (\, B \,) \, M_1 \, S_1 \, N \, \textbf{else} \, M_2 \, S_2$
$\{ \; backpatch(B.truelist, \; M_1.instr);$
$backpatch(B.falselist, \; M_2.instr);$
$temp \; = \; merge(S_1.nextlist, \; N.nextlist);$
$S.nextlist \; = \; merge(temp, \; S_2.nextlist); \; \}$

goto ____

3) $S \rightarrow \textbf{while} \; M_1 \, (\, B \,) \, M_2 \, S_1$
$\{ \; backpatch(S_1.nextlist, \; M_1.instr);$
$backpatch(B.truelist, \; M_2.instr);$
$S.nextlist \; = \; B.falselist;$
$emit(\text{'}\mathbf{goto}\text{'} \; M_1.instr); \; \}$

end of prog

4) $S \rightarrow \{ \; L \; \}$ $\{ \; S.nextlist \; = \; L.nextlist; \; \}$

5) $S \rightarrow A \; ;$ $\{ \; S.nextlist \; = \; \textbf{null}; \; \}$

6) $M \rightarrow \epsilon$ $\{ \; M.instr \; = \; nextinstr; \; \}$

7) $N \rightarrow \epsilon$ $\{ \; N.nextlist \; = \; makelist(nextinstr);$
$emit(\text{'}\mathbf{goto} \; \_\text{'}); \; \}$

8) $L \rightarrow L_1 \; M \; S$ $\{ \; backpatch(L_1.nextlist, \; M.instr);$
$L.nextlist \; = \; S.nextlist; \; \}$

9) $L \rightarrow S$ $\{ \; L.nextlist \; = \; S.nextlist; \; \}$

Translation of a switch-statement

switch (E) {
    case V_1: S_1
    case V_2: S_2
        ...
    case V_{n-1}: S_{n-1}
    default: S_n
}

	code to evaluate E into t
	goto test
L_1:	code for S_1
	goto next
L_2:	code for S_2
	goto next
	...
L_{n-1}:	code for S_{n-1}
	goto next
L_n:	code for S_n
	goto next
test:	if t = V_1 goto L_1
	if t = V_2 goto L_2
	...
	if t = V_{n-1} goto L_{n-1}
	goto L_n
next:	

	code to evaluate E into t
	if t != V_1 goto L_1
	code for S_1
	goto next
L_1:	if t != V_2 goto L_2
	code for S_2
	goto next
L_2:	
	...
L_{n-2}:	if t != V_{n-1} goto L_{n-1}
	code for S_{n-1}
	goto next
L_{n-1}:	code for S_n
next:	