INTERMEDIATE - CODE GENERATION

Variants of Syntax trees

Nodes in a syntax tree represents constructs in the source program, the children of a node represent the meaningful components of a construct. A Directed Acyclic Graph (DAG) for an expression identifier the common subexpression of the expressions.

Directed Acyclic Graphs for Expressions

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators. A DAG not only represents exp-ressions succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

$$a + a * (b - c) + (b - c) * d$$

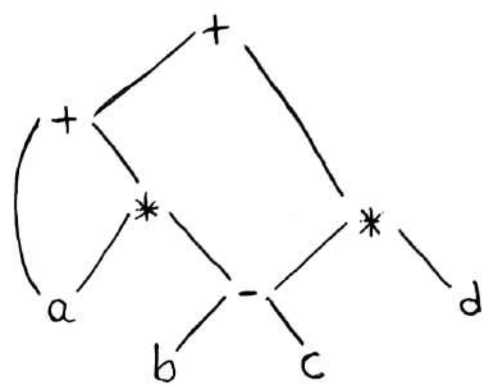


Fig:- 1 DAG for the expression $a + a * (b - c) + (b - c) * d$

Production	Semantic Rules
$E \rightarrow E_1 + T$	E.node = new Node('+', E_1.node, T.node)
$E \rightarrow E_1 - T$	E.node = new Node('-', E_1.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow (E)$	T.node = E.node
$T \rightarrow id$	T.node = new Leaf (id, id.entry)
$T \rightarrow num$	T.node = new Leaf (num, num.val)

Syntax directed defination to produce Syntax tree or DAG

1) p_1 = Leaf (id, entry b)
2) p_2 = Leaf (id, entry c)
3) p_3 = Node ('-', p_1, p_2)
4) p_4 = Leaf (id, entry a)
5) p_5 = Node ('*', p_4, p_3)
6) p_6 = Leaf (id, entry d)
7) p_7 = Node ('*', p_6, p_3)
8) p_8 = Node ('+', p_4, p_5)
9) p_9 = Node ('+', p_8, p_7)

Steps for constructing the DAG i-e; Fig:-1

The fig.1 shows the DAG for the expression

$$a+a*(b-c)+(b-c)*d$$

The leaf for a has two parents, because a appear twice in the expression. More intrestingly the two occurance of the common subexpression b-c are represent by one node, the node labeled -, that node has two parents, representing its two uses in the subexpression $a*(b-c)$ and $(b-c)*d$. Even though b and c appear twice in the complex expression, their nodes each have one parent, since both uses are in the common subexpression b-c.

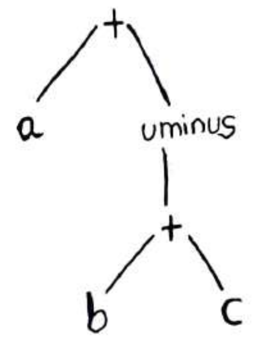
Advantages of DAG

1) More compact representation.

2) Gives clues regarding generation of efficient code.

Construct a DAG for the expressions given below and also give the sequence of steps for constructing the same

$a+-(b+c)$



a) DAG for the expression $a+-(b+c)$

p_1 = leaf (id, entry b)

p_2 = leaf (id, entry c)

p_3 = node ('+', p_1, p_2)

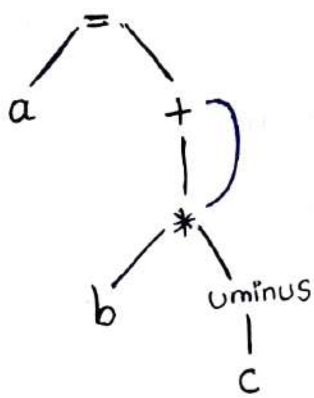
p_4 = leaf (id, entry uminus p_3)

p_5 = leaf (id, entry a)

p_6 = node ('+', p_5, p_4)

a] Steps for constructing the DAG.

$a = b * -c + b * -c$



b] DAG for the expression $a = b * -c + b * -c$

p_1 = leaf (id, entry b)

p_2 = leaf (id, entry uminus c)

p_3 = node ('*', p_1, p_2)

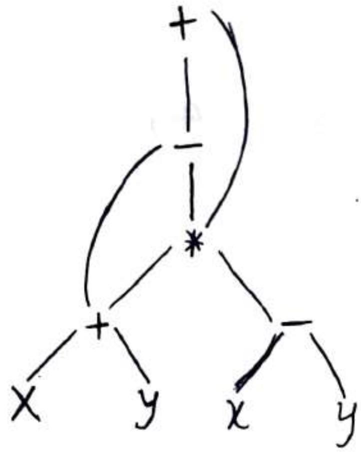
p_4 = node ('+', p_3, p_3)

p_5 = leaf (id, entry a)

p_6 = node ('=', p_5, p_4)

b] Steps for constructing the DAG

$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$



c] DAG for the expression $((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$

$p_1 = leaf(id, entry\ x)$

$p_2 = leaf(id, entry\ y)$

$p_3 = node('+', p_1, p_2)$

$p_4 = node('-', p_1, p_2)$

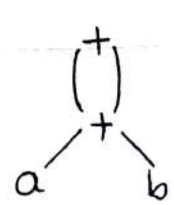
$p_5 = node('*', p_3, p_4)$

$p_6 = node('-', p_3, p_5)$

$p_7 = node('+', p_6, p_5)$

c] Steps for constructing the DAG.

$a+b+a+b$



d] DAG for the expression $a+b+a+b$

$P_1 = leaf(id, entry\ a)$

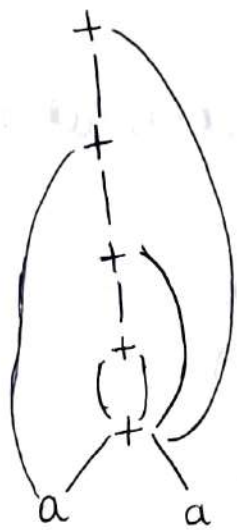
$P_2 = leaf(id, entry\ b)$

$P_3 = node('+', P_1, P_2)$

$P_4 = node('+', P_3, P_3)$

d] Steps for constructing the DAG

$a + a + ((a + a + a + (a + a + a + a))$



e] DAG for the expression $a + a + (a + a + a + (a + a + a + a))$

$P_1 = leaf(id, entry\ a)$

$P_2 = leaf(id, entry\ a)$

$P_3 = node('+', P_1, P_2)$

$P_4 = node('+', P_3, P_3)$

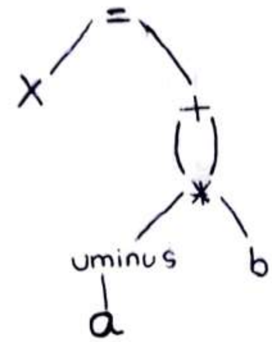
$P_5 = node('+', P_4, P_3)$

$P_6 = node('+', P_5, P_1)$

$P_7 = node('+', P_6, P_3)$

e] Steps for constructing the DAG

$x = -a * b + -a * b$



f] DAG for the expression $x = -a * b + -a * b$

$p_1 = leaf(id, entry\ uminus\ a)$

$p_2 = leaf(id, entry\ b)$

$p_3 = node('*', p_1, p_2)$

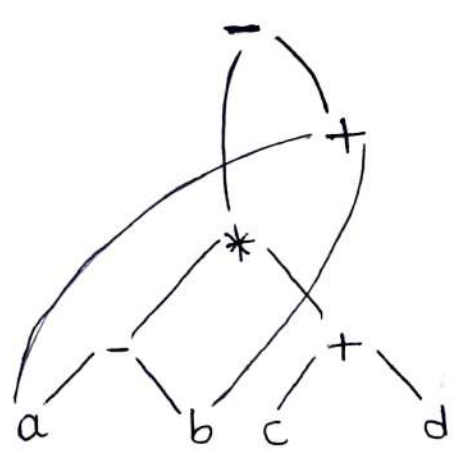
$p_4 = node('+', p_3, p_3)$

$p_5 = leaf(id, entry\ x)$

$p_6 = node('=', p_5, p_4)$

f] steps for constructing the DAG.

$(a-b) * (c+d) - (a+b)$



g] DAG for the expression $(a-b) * (c+d) - (a+b)$

$p_1 = leaf(id, entry\ a)$

$p_2 = leaf(id, entry\ b)$

$p_3 = leaf(id, entry\ c)$

$p_4 = leaf(id, entry\ d)$

$p_5 = node('-', p_1, p_2)$

$p_6 = node('+', p_3, p_4)$

$p_7 = node('+', p_1, p_2)$

$p_8 = node('*', p_5, p_6)$

$p_9 = node('-', p_8, p_7)$

9] Steps for constructing the DAG

Three - Address Code

In three address code, there is atmost one operator on the right side of an instruction. Thus a source-language expression like $x+y*z$ might be translated into the sequence of three address instructions.

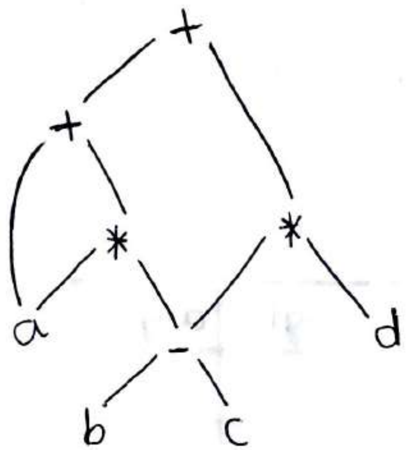
$t_1 = y * z$

$t_2 = x + t_1$

where t_1 and t_2 are compiler-generated temporary names.

Example :- Three-address code is a linearized representation of a syntax tree or a DAG in which explict names correspond to the interior nodes of the graph.

$$a + a * (b-c) + (b-c) * d$$



$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

a] DAG

b] Three-address code

A DAG and its corresponding three-address code.

Three ways representation of three-address code are :-

1] Quadruples
2] Triples
3] Indirect Triples.

<u>Quadruples</u> :- A quadruple has four fields, which we call OP, arg1, arg2 and result. The OP field contains an internal code for the operator. For instance, the three-address instruction $x = y + z$ is represented by placing + in OP, y in arg1 and z in arg2, x in result.

$a = b * -c + b * -c$

$t_1 =$ uminus c
$t_2 = b * t_1$
$t_3 =$ uminus c
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

a] Three-Address Code

	OP	arg1	arg2	Result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

b] Quadruples

Three-Address Code and its Quadruple Representation

Triples :- A Triple has only three fields, which we call op, arg1 and arg2. Using triples, we refer to the result of an operation x op y by its position, rather than by an explicit temporary name.

$$a = b * -c + b * -c$$

t_1 = minus c
$t_2 = b * t_1$
t_3 = minus c
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

a] Three-Address Code

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

b] Triples

Three-Address Code and its Triple representation

Indirect Triples :- Indirect Triples consists of a listing of pointers to triples, rather than a listing of triples themselves.

$$a = b * -c + b * -c$$

Instruction

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	.....

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Indirect Triples Representation of three-address code

In the quadruple representation using temporary names the entries in the symbol table against those temporaries can be obtained.

*] The advantage with quadruple representation is that one can quickly access the value of temporary variables using symbol table.

*]. The quadruple representation is benefical for code optimization.

*] Indirect Triple saves some amount of space as compared with quadruple representation.

Static Single - Assignment Form

Static Single - assignment form (SSA) is an intermediate representation that facilitates certain code optimization. In SSA all assignments to variables with distinct names, hence the term static single assignment. Below figure:* shows the same intermediate program in three-address code and in static single assignment form.

a] Three-address code	b] static - Single - assignment Form
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

Fig. * Intermediate Program in three address code and Static Single - Assignment form

Write the quadruple, Triple and Indirect Triple for the statement.

$$a = b * -c + b * -c$$

t_1 = minus c

$t_2 = b * t_1$

t_3 = minus c

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

a] Three - Address Code

	OP	Arg1	Arg2	Result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

b] Quadruple

	OP	Arg1	Arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

b] Triple

Indirect Triples:-

Instruction

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	OP	Arg1	Arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Translate the arithmetic expression $a + a * (b - c) + (b - c) * d$ into quadruple, triple and indirect triple.

The three address code will be

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = t_1 * d$

$t_4 = t_2 + t_3$

$t_5 = a + t_4$

	OP	Arg1	Arg2	Result
0	−	b	c	t_1
1	*	a	t_1	t_2
2	*	t_1	d	t_3
3	+	t_2	t_3	t_4
4	+	a	t_4	t_5

b] Quadruple

Triple

	OP	ARG1	ARG2
0	−	b	c
1	*	a	(0)
2	*	d	(0)
3	+	(1)	(2)
4	+	a	(3)

Indirect Triple

Instruction

10	(0)
11	(1)
12	(2)
13	(3)
14	(4)

	OP	ARG1	ARG2
0	−	b	c
1	*	a	(0)
2	*	d	(0)
3	+	(1)	(2)
4	+	a	(3)

Write the quadruple, Triple and Indirect triple for the statement

$$X = -a * b + -a * b$$

a] Three address code

t_1 = minus a

$t_2 = t_1 * b$

t_3 = minus a

$t_4 = t_3 * b$

$t_5 = t_2 + t_4$

$X = t_5$

b] Quadruple

	OP	ARG1	ARG2	Result
0	minus	a		t_1
1	*	t_1	b	t_2
2	minus	a		t_3
3	*	t_3	b	t_4
4	+	t_2	t_4	t_5
5	=	t_5		X

c] Triple

	OP	ARG 1	ARG 2
0	minus	a	
1	*	(0)	b
2	minus	a	
3	*	(2)	b
4	+	(1)	(3)
5	=	(4)	X

Indirect Triple

11	(0)
12	(1)
13	(2)
14	(3)
15	(4)
16	(5)

	OP	ARG 1	ARG 2
0	minus	a	
1	*	(0)	b
2	minus	a	
3	*	(2)	b
4	+	(1)	(3)
5	=	(4)	X

Control Flow :-

The translation of statement such as if-else statement and while statements is tied to the translation of boolean expression. In programming langu-ages, bodean expression are often used to

a] Alter the flow of control :- Boolean expression are used as conditional expression in statements that alter the flow of control.

For example ;- If (E) then S

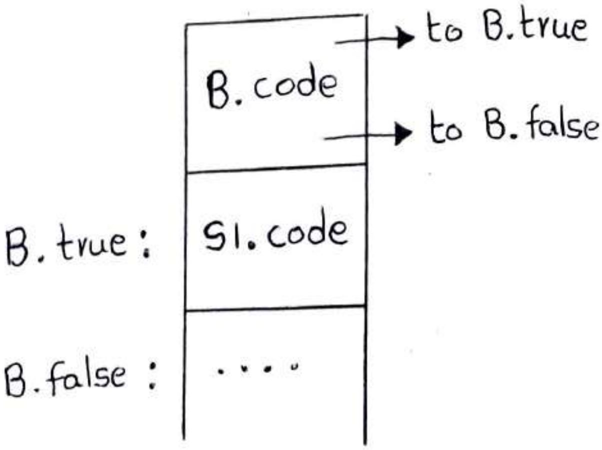
The expression E must be true if statement S is reached.

b] Compute logical values :- A boolean expression can represent true or false as values. Such boolean expressions can be evaluated in analogy to arithmetic expression using three address instructions with logical operators.

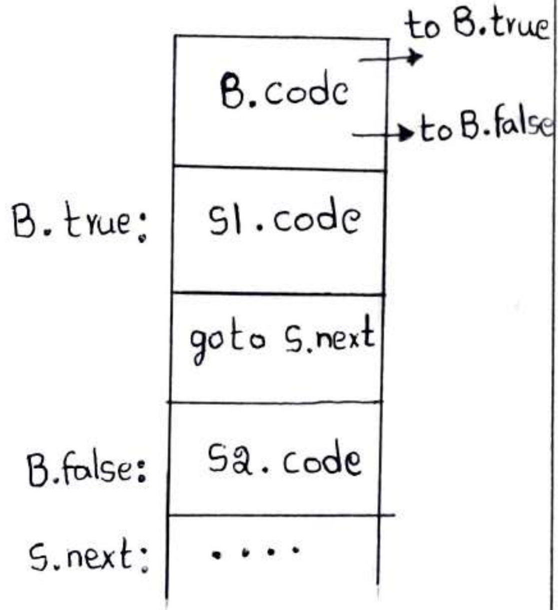
Flow of Control Statements

In this section we will discuss the translation of Boolean expression into three address code. The control statements are if-then-else and while-do. The grammar for such statements is as shown below.

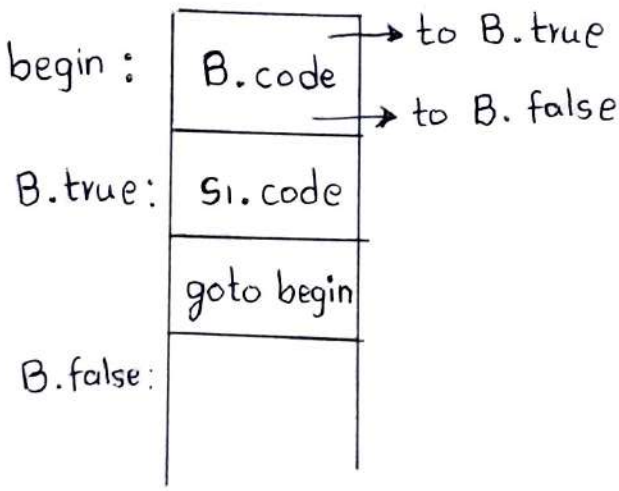
$$S \longrightarrow \text{if } E \text{ then } S_1$$
$$\mid \text{if } E \text{ then } S_1 \text{ else } S_2$$
$$\mid \text{while } E \text{ do } S_1$$



a] if



b] if else



c] while

Fig:* Code for if, if-else and while statements.

The translation of if (B) S_1 consists of B.code followed by S_1.code as illustrated in Fig:*(a) within B.code are jumped based on the value of B. If B is true, control flows to the first instruction of S_1.code and if B is false, control flows to the instruction immediately following S_1.code

While generating three address code :-

*] To generate new symbolic lable the function new-label() is used.

*] With the expression E.true and E.false are the labels associated.

*] S.code and E.code is for generating three address code.

<u>if</u>

$S \longrightarrow$ if E then S_1

<u>Semantic rule</u>:-

E.true = new_label()

E.false = S.next

S_1.next = S.next

S.code = E.code || gen.code(E.true) || S_1.code

Consider the statement if a<b then s=a

The three address code for if - then is

if (E) goto L1
goto L2
L1 : S_1
L2 :

if a<b goto L1
goto L2
L1 : s = a
L2 :

<u>if - else</u> :-

$S \longrightarrow$ if E then S_1 else S_2

$S \rightarrow$ if E then S_1 else S_2

Semantic Rule :-

E.true = new_label()

E.false = new_label()

S_1.next = S.next

S_2.next = S.next

S.code = E.code || gen_code (E.true) || S_1.code || gen_code ('goto', S.next) || gen-code (E.false) || S_2.code

Eg:- Consider the statement if (a<b) then (s=a) else (s=b)

The three address code for if then else is

```
if (E) goto L1
 goto L2
L1 : S1
 goto L3
L2 : S2
L3 : . . .
```

```
if(a < b) goto L1
      goto L2
L1 : s = a
      goto L3
L2 : s=b
L3 : . . . .
```

While

$S \longrightarrow$ while (E) do S_1

Semantic Rule :

S.begin = new_label()

E.true = new_label()

E.false = S.next

S_1.next = S.begin

S.code = gen-code (S.begin) || E.code || gen-code (E.true) ||
S1.code || gen-code ('goto', S.begin)

Eg:- consider the statement while (a<b) do (s=a)

```
while E goto L1
   goto L2
L1 : S1
L2 :
```

```
while (a<b) goto L1
   goto L2
L1 : s=a
L2 :
```

Generate three address code for the following segment of code.

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 : ...
```

Syntax-directed defination for flow of Control statements

PRODUCTION	SEMANTIC RULES
P → S	S.next = new.label() P.code = S.code ‖ label (S.next)
S → assign	S.code = assign. code
S → if (B) S_1	B.true = newlabel () B.false = S1.next = S. next S.code = B . code ‖ label (B. true) ‖ S_1.code
S → if (B) S_1 else S_2	B.true = newlabel () B.false = newlabel () S1.next = S2.next = S.next S.code = B.code ‖ label (B.true) ‖ S1.code ‖ gen ('goto', S.next) ‖ label (B.false) ‖ S2.code
S → while (B) S_1	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)

Switch Statement :- The Switch or case statement is available in a variety of language. Our switch statement syntax is shown in the fig.(a). There is a selector expression E, which is to be evaluated, followed by n constant values $V_1, V_2, \ldots, V_n$ that the expression might take, perphaps including a default "value", which always matches the expression if no other value does

```
Switch (E) {
    case V1 : S1
    case V2 : S2
    ....
    case Vn-1 : Sn-1
    default : Sn
}
```

fig.(a): switch-statement syntax

Arrays :- As we know array is a collection of contiguous storage of elements. For accessing any elements of an array what we need is its address. For statically declared arrays it is possible to compute the relative address of each element.

Typically there are two representations of arrays.

1. Row Major Representation.
2. Column Major Representation.

There representations are as shown below

Row Major Representation

A[1, 1]	A[1, 2]	A[1, 3]	A[2, 1]	A[2, 2]	A[2, 3]
ROW 1			ROW 2		

Column Major Representation

A[1, 1]	A[2, 1]	A[1, 2]	A[2, 2]	A[1, 3]	A[2, 3]
COL 1		COL 2		COL 3	

Fig :- Row Major and Column Major Representation

To compute the address of any element :

Let base is the address at a[ ] and w is the width of the element (required memory units) then to compute i^{th} address of a[ ]

$$base + (i - low) * w$$

where low is lower bound on subscript.

$$base + (i - low) * W$$

$$\Rightarrow base + i * W - low * W$$

$$\Rightarrow i * W + (base - low * W)$$

Let $C = (base - low * W)$ is computed at compile time. Then the relative address of a[i] can be computed as

$$C + (i * W)$$

Similarly for calculation of relative address of two dimensional array we need to consider i and j subscripts. Considering row major representation we will compute the relative address for a[i, j] using following formula.

$$a[i, j] = base + ((i - low_1) * n_2 + (j - low_2)) * W$$

where low_1 and low_2 are the two lower bounds on values of i and j.

Assuming that i and j are not known at compile time we can re-write the formula as:

$$a[i, j] = ((i * n_2) + j) * W + (base - ((low_1 * n_2) + low_2) * W)$$

The term $(base - ((low_1 * n_2) + low_2) * W)$ can be computed at compile time.

For example :- Consider an array A of size $10 * 20$ assuming $low_1 = 1$ and $low_2 = 1$. The computation of $A[i, j]$ is possible by assuming that $W = 4$ as

$$A[i, j] = ((i * n_2) + j) W + (base - ((low_1 * n_2) + low_2) * W)$$

Given

$$n_2 = 20$$
$$W = 4$$
$$low_1 = 1$$
$$low_2 = 1$$

$$A[i, j] = ((i * 20) + j) * 4 + (base - ((1 * 20) + 1) * 4)$$

$$A[i, j] = 4 * (20i + j) + (base - 84)$$

The value (base-84) can be computed at compile time.

Question no: 1

Generate the three address code for the expression $X := A[i, j]$ for an array $10 * 20$ Assume $low_1 = 1$ and $low_2 = 1$

Given that :- $low_1 = 1$, $low_2 = 1$, $n_1 = 10$, $n_2 = 20$

$$A[i, j] = ((i * n_2) + j) * W + (base - ((low_1 * n_2) + low_2) * W)$$

$$A[i, j] = ((i * 20) + j) * 4 + (base - ((1 * 20) + 1) * 4)$$

$$\Rightarrow 4 * (20i + j) + (base - 84)$$

The three address code for this expression can be

$t_1 = i * 20$

$t_1 = t_1 + j$

$t_2 = c$ /* Computation of c = base - 84. */

$t_3 = 4 * t_1$

$t_4 = t_2[t_3]$

$X = t_4$

Question no. 2

Generate the intermediate code for the statement

$$Sum = A[i, j] + B[i, j]$$

Solution :-

Before computing the three address code we will assume $low_1 = 1$, $low_2 = 2$, and array is of size 10*10 $n_1 = 10$ and $n_2 = 10$ and width = 4

$A[i, j] = ((i * n_2) + j) * W + (base - ((low_1 * n_2) + low_2) * W)$

$= ((i * n_2) + j) * W + Addr\,A$

$= ((i * 10) + j) * 4 + Addr\,A$

$= (10i + j) * 4 + Addr\,A$

$B[i, j] = ((i * n_2) + j) * W + (base - ((low_1 * n_2) + low_2) * W)$

$= ((i * n_2) + j) * W + Addr\,B$

$= ((i * 10) + j) * 4 + Addr\,B$

$= (10i + j) * 4 + Addr\,B$

The three address code will then be

$i = 1$

$j = 1$

$T_1 = i$

$T_2 = j$

$T_3 = 10 * T_1$

$T_4 = T_3 + T_2$

$T_5 = Addr\ A$

$T_6 = 4 * T_4$

$T_7 = T_5[T_6]$ /* A[i,j] */

$T_8 = 10 * T_1$

$T_9 = T_8 + T_2$

$T_{10} = Addr\ B$

$T_{11} = 4 * T_9$

$T_{12} = T_{10}[T_{11}]$ /* B[i,j] */

$T_{13} = T_7 + T_{12}$ /* A[i,j] + B[i,j] */

$SUM = T_{13}$

Back Patching :-

Implementation of syntax directed definition using two parser is the most convienient method. If we decide to generate the three address code for given syntax directed definition using single passe only, then the main problem that occurs is the decision of address of the labels. The goto statements refer there label statements and in one passe it becomes difficult to know the locations of these label statements. If we use two passes instead of one pass then in one pass we can leave these address unspecified and in the second pass. this incom-plete information can be filled up. To overcome the problem of processing the incomplete information in one pass the back patching technique is used.

Back Patching Definition :-

Back patching is the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.

To generate code using backpatching. In the semantic actions following functions are used.

1. mklist(i) :- Creates the newlist. The index i is passed as an argument to this function, where i is an index to the array of quadruple.

2. merge_list(p_1, p_2) :- Function concatenates two lists pointed by p_1 and p_2. It returns the pointer to the concat-enated list.

3. backpatch(p, i) :- Insert i as target label for the statement pointed by pointer p.

Backpatching for Boolean Expressions

Consider the grammar for Boolean Expressions

$B \rightarrow B_1 \| M B_2$

$B \rightarrow B_1 \&\& M B_2$

$B \rightarrow !B_1$

$B \rightarrow (B_1)$

$B \rightarrow E_1$ relop E_2

$B \rightarrow$ True

$B \rightarrow$ False

$M \rightarrow \epsilon$

Here a new non-terminal M is inserted as a marker non-terminal. The purpose of M is to mark the exact point where the semantic action is picked up.

Production	Semantic Rules
1] $B \rightarrow B_1 \| M B_2$	{backpatch(B_1.falselist , M.instr); B.truelist = merge (B_1.truelist , B_2.truelist) B.falselist = B_2.falselist ;}
2] $B \rightarrow B_1 \&\& M B_2$	{backpatch (B_1.truelist , M.instr); B.truelist = B_2.truelist ; B.falselist = merge (B1.falselist , B_2.falselist); }

Production	Semantic Rules
3] $B \rightarrow !B_1$	{ $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
4] $B \rightarrow (B_1)$	{ $B.truelist = B_1.truelist$; $B.falselist = B_1.falselist$; }
5] $B \rightarrow E_1$ rel E_2	{ B.truelist = makelist (next instr); B.falselist = makelist (next instr t_1); emit('if' E_1.addr rel op E_2.addr 'goto —'); emit ('goto —'); }
6] $B \rightarrow$ true	{ B.truelist = makelist (next instr); emit ('goto —'); }
7] $B \rightarrow$ false	{ B.falselist = makelist (next instr); emit ('goto —'); }
8] $M \rightarrow \epsilon$	{ M.instr = next instr ; }

Fig :- Translation scheme for boolean Expression

Example :- Using Backpatching, generate an intermediate code for following expression

A<B OR C<D AND P<Q

Solution :-

For the given expression

A<B OR C<D AND P<Q we will scan it from left to right.

A<B matches with the rule $E \rightarrow id_1$ relop id_2.

In response to this statement the three address code will be generated as,

	as semantic actions are
100 if A<B goto ___	append ('if' id_1.place relop id_2.place goto ___)
101 goto ___	append ('goto ___')

Similarly for the remaining part of expression the three address code will be

102 if C<D goto ___

103 goto ___

104 if P<Q goto ___

105 goto ___

Now. for the expression

if $A<B$ OR $C<D$ AND $P<Q$, we will solve AND operation first then OR, as AND has higher precedence over OR

Hence,

102 if $C<D$ goto 104
103 goto 107
104 if $P<Q$ goto 106
105 goto 107

Now consider

$A<B$ OR $C<D$ AND $P<Q$

Finally the 3-address with backpatching will be

100 if $A<B$ goto 106
101 goto 102
102 if $C<D$ goto 104
103 goto 107
104 if $P<Q$ goto 106
105 goto 107
106
107

Syntax - Directed Translation of Switch Statement

```
            Code to evaluate E into t
            goto test

      L1 :  code for S1
            goto next

      L2 :  code for S2
            goto next

            .......

      Ln-1 : code for Sn-1
            goto next

      Ln  : code for Sn
            goto next

     test : if t = V1 goto L1
            if t = V2 goto L2
            ......
            if t = Vn-1 goto Ln-1
            goto Ln

     next :
```

Fig.:- Translation of Switch Statement