

UNIT-44.1: INTERMEDIATE CODE GENERATOR

There are three parts in this unit, this first part asks 10 marks in exam

⊗ Intermediate Code Generation:

The front end translates the source program into intermediate representation from which the backend generates target code. Intermediate codes are machine independent codes, but they are close to machine instructions.

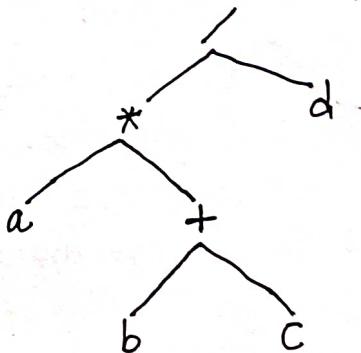
Advantages of using intermediate code representation:

- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

⊗. Intermediate Representations:1). Graphical Representations:

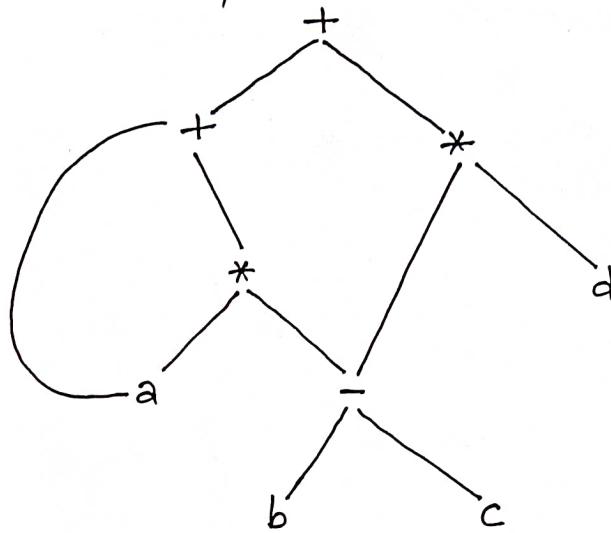
i) Syntax tree: Syntax tree is a graphical representation of given source program and it is also called variant of parse tree. A tree in which each leaf represents an operand and each interior node represents an operator is called syntax tree.

Example: Syntax tree for the expression $a^*(b+c)/d$



1) Directed acyclic graph (DAG): A DAG for an expression identifies the common sub expressions in the expression. It is similar to syntax tree, only difference is that a node in a DAG representing a common sub expression has more than one parent, but in syntax tree the common sub expression would be represented as a duplicate subtree.

Example: DAG for the expression $a + a * (b - c) + (b - c) * d$.



2) Postfix notation: The representation of an expression in operators followed by operands is called postfix notation of that expression. In general if x and y be any two postfix expressions and OP is a binary operator then the result of applying OP to the x and y in postfix notation by " $x\ y\ OP$ ".

Examples:

1. $(a+b)*c$ in postfix notation is: $ab+c*$
2. $a*(b+c)$ in postfix notation is: $abc+*$

3) Three Address Code: The address code that uses at most three addresses, two for operands and one for result is called three address code. Each instruction in three address code can be described as a 4-tuple: (operator, operand₁, operand₂, result). A quadruple (three address code) is of the form:

$x = y \text{ op } z$ where x, y and z are names, constants or compiler-generated temporaries and op is any operator.

We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).

Example 1: Three address code for expression: $(B+A) * (Y - (B+A))$

Solution:

$$t1 = B+1$$

$$t2 = Y - t1$$

$$t3 = t1 * t2$$

per line JTT maximum 3
variable हुन्ही पाई 1, 2, 3
वर्ति नहीं हो सके

Maths मा जैसी-जैसी precedence

जैसी हो सकती है अब
convert JTT here
bracket first.

Example 2: Three address code for expression:

$$g = 2 * n + k$$

while $g \neq 0$ do

$$g = g - k$$

Solution:

$$t1 = 2$$

$$t2 = t1 * n$$

$$t3 = t2 + k$$

$$g = t3$$

JTT की 3
variable होनी
L1, L2, T1, i

This first four lines
for $g = 2 * n + k$

We use two variables
L1 and L2 when loop comes.
L1 for true condition and
L2 for false condition. Here
instead of $g=0$ we can also
write $g=false$

L1: if $g=0$ goto L2

$$t4 = g - k$$

$$g = t4$$

goto L1

L2:

These three lines are for
 $g=g-k$ which is body of loop
goes out of body i.e. L2 when $i=0$

②. Implementation of Three-Address Statements:

Three-address statements are implemented or represented by Quadruples and Triples.

Quadruples: A quadruple is a record structure with four fields,

Imply which are op, arg1, arg2 and result. The op field contains an internal code for the operator. The three-address statement $x = y \text{ op } z$ is represented by placing y in arg1, z in arg2 and x in result.

	OP	arg1	arg2	result
(0)	-	c		t1
(1)	*	b	t1	t2
(2)	-	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

(0), (1), (2), ...
numbering nothing
else

first do three-
address in rough
as below then it is
easy:

$$\begin{aligned} t1 &= -c \\ t2 &= b * t1 \\ t3 &= -c \\ t4 &= b * t3 \\ t5 &= t2 + t4 \\ a &= t5 \end{aligned}$$

Fig: Quadruple representation of three-address statement $a = b * -c + b * -c$

Triples: A triple is a record structure with three fields, which are [Imp] op, arg1 and arg2. The result field of quadruples is absent here all other things are same, as quadruples. The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure.

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

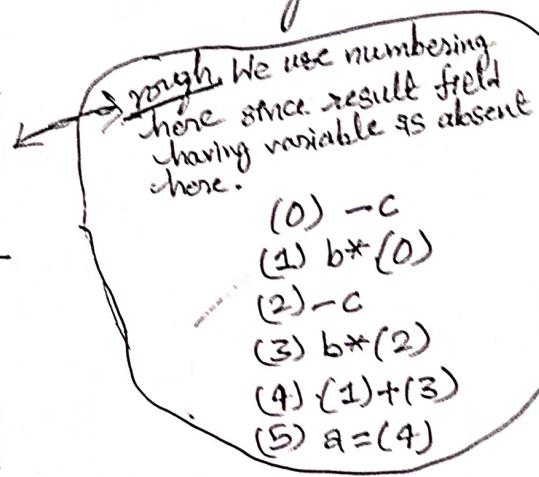


Fig: Triple representation of three-address statement $a = b * -c + b * -c$.

Example: Translate the expression $x = (a+b)* (c+d) - (a+b+c)$

- a. Quadruples
- b. Triples.

Solution:

Three address code is:

$$\begin{aligned}
t1 &= a + b \\
t2 &= c + d \\
t3 &= t1 * t2 \\
t4 &= t1 + c \\
t5 &= t3 - t4 \\
x &= t5
\end{aligned}$$

Quadruples:

	op	arg1	arg2	result
(0)	+	a	b	t1
(1)	+	c	d	t2
(2)	*	t1	t2	t3
(3)	+	t1	c	t4
(4)	-	t3	t4	t5
(5)	=	t5		x

Triples:

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

2) Boolean Expressions: [Imp]

Production

$$B \rightarrow B_1 \parallel B_2$$

production मा // symbol
आप logical OR होता है।
Semantic rule मा आप
मात्र concatenation

Semantic Rules

$B_1 \cdot \text{true} = B \cdot \text{true}$
 $B_1 \cdot \text{false} = \text{newlabel}()$
 $B_2 \cdot \text{true} = B \cdot \text{true}$
 $B_2 \cdot \text{false} = B \cdot \text{false}$
 $B \cdot \text{code} = B_1 \cdot \text{code} \parallel \text{label}(B_1 \cdot \text{false}) \parallel B_2 \cdot \text{code}$

logical OR सा कुनै रक्ती सालू
true वर तरीके true होने तक
false होना तरीके false होने।
So, यदि $B_1 \cdot \text{true}$ हो तो $B \cdot \text{true}$
नहीं। But if $B_1 \cdot \text{false}$ हो
अब $B \cdot \text{true}$ हो B_2 नहीं जाएगा।
जैसे we do not know B_2 so
we give it newlabel(). Now
 B_2 is true B is true
otherwise false
finally we
address code.

$$B \rightarrow B_1 \& B_2$$

logical AND.
 $B \cdot \text{true}$ होने के लिए
 $B_1 \& B_2$ true होना पड़ेगा

$$B_1 \cdot \text{true} = \text{newlabel}()$$

$$B_1 \cdot \text{false} = B \cdot \text{false}$$

$$B_2 \cdot \text{true} = B \cdot \text{true}$$

$$B_2 \cdot \text{false} = B \cdot \text{false}$$

$$B \cdot \text{code} = B_1 \cdot \text{code} \parallel \text{label}(B_1 \cdot \text{true}) \parallel B_2 \cdot \text{code}$$

$B_1 \cdot \text{true}$ होना मा $B \cdot \text{true}$ होने
जैसे decide जैसे सवेदने, logical
AND मा $B \cdot \text{true}$ होने $B_1 \& B_2$ को
true होने पड़ेगा, So, B to be
true now it depends on value
of B_2 so newlabel() is given.

$$B \rightarrow !B_1$$

$$B_1 \cdot \text{true} = B \cdot \text{false}$$

$$B_1 \cdot \text{false} = B \cdot \text{true}$$

$$B \cdot \text{code} = B_1 \cdot \text{code}$$

$$E \rightarrow \text{id}_1 \text{ relop } \text{id}_2$$

relation operator (can be
<, >, =, <=, >=, != etc.)

$$E \cdot \text{code} = \text{gen}(\text{'if'} \text{ id}_1 \cdot \text{place} \text{ relop } \text{id}_2 \cdot \text{place}\text{'goto' E.true}) \parallel \text{gen}(\text{'goto' E.false})$$

$$B \rightarrow \text{true}$$

$$B \cdot \text{code} = \text{gen}(\text{goto } B \cdot \text{true})$$

$$B \rightarrow \text{false}$$

$$B \cdot \text{code} = \text{gen}(\text{goto } B \cdot \text{false})$$

3) Flow of control statements: [Imp]

Production

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

Semantic Rules

$$B \cdot \text{true} = \text{newlabel}()$$

$$B \cdot \text{false} = S \cdot \text{next}$$

$$S_1 \cdot \text{next} = S \cdot \text{next}$$

$$S \cdot \text{code} = B \cdot \text{code} \parallel \text{label}(B \cdot \text{true}) \parallel S_1 \cdot \text{code}$$

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

$$B \cdot \text{true} = \text{newlabel}()$$

$$B \cdot \text{false} = \text{newlabel}()$$

$$S_1 \cdot \text{next} = S \cdot \text{next}$$

$$S_2 \cdot \text{next} = S \cdot \text{next}$$

$$S \cdot \text{code} = B \cdot \text{code} \parallel \text{label}(B \cdot \text{true}) \parallel S_1 \cdot \text{code} \parallel \\ \text{gen}(\text{goto } S \cdot \text{next}) \parallel \text{label}(B \cdot \text{false}) \parallel S_2 \cdot \text{code}$$

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

$\begin{aligned} \text{begin} &= \text{newlabel}() \\ B.\text{true} &= \text{newlabel}() \\ B.\text{false} &= S.\text{next} \\ S_1.\text{next} &= \text{begin} \\ S.\text{code} &= \text{label(begin)} // B.\text{code} // \text{label}(B.\text{true}) // \\ S_1.\text{code} &= \text{gen(goto begin).} \end{aligned}$

Example : Generate three address code for the expression

$\text{if } (x < 5 \text{ || } (x > 10 \text{ & } x == y)) x = 3;$

Solution:

L1: if $x < 5$ goto L2
else
 goto L3

L2: $x = 3$ goto L5

L3: if $x > 10$
 goto L4
else
 goto L5

L4: if $x == y$
 goto L2
else
 goto L5

L5: ...

think of if condition in C program
 This is equivalent to as:
 $\text{if } (x < 5 \text{ || } (x > 10 \text{ & } x == y))$
 $\{ x = 3$
 Logical OR
 $\}$
 i.e., if $x < 5$ it goes to $x = 3$
 OR if $x > 10$ and $x == y$ then
 also it goes to $x = 3$
 otherwise goes out of if block

4) Switch/case statements: [Impl]

Example: Convert the following switch statement into three address code:

Switch ($i+j$)

{ Case1: $x = y + z$

Case2: $u = v + w$

Case3: $p = q * w$

Default: $s = u / v$

}

Solution:

$t = i + j$

3 address code की तरफ से लिखें

L1: if $t == 1$ goto L2

else goto L3

L2: $x = y + z$ goto L8

L3: if $t == 2$ goto L4

else goto L5

L4: $u = v + w$ goto L8

L5: if $t == 3$ goto L6

else goto L7

L6: $p = q * w$ goto L8

L7: $s = u / v$ goto L8

L8: ...

5) Addressing array elements:

If the width of each array element is w , then the i th element of array 'A' begins in location,

$$A[i] = \text{base} + (i - \text{low}) * w$$

where low is the lower bound on the subscript and base is the relative address of the storage allocated for the array. That is base is the relative address of $A[\text{low}]$.

Example: Address of 15th element of array is calculated as below,

Suppose base address of array is 100 and type of array is integer of size 4 bytes and lower bound of an array is 10 then,

$$A[i] = \text{base} + (i - \text{low}) * w$$

$$\begin{aligned} \text{or } A[15] &= 100 + (15 - 10) * 4 \\ &= 100 + 20 \\ &= 120. \end{aligned}$$

⇒ Similarly for two dimensional array we have formula,

$$A[i_1, i_2] = \text{base}_A + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * w$$

where $\text{low}_1, \text{low}_2$ are the lower bounds on values i_1 and i_2 , n_2 is the number of values that i_2 can take.

Example: Let A be a 10×20 array, there are 4 bytes per word, assume $\text{low}_1 = \text{low}_2 = 1$. Find addressing array elements $A[i_1, i_2]$ and three-address code, for 2D array addressing.

Solution:

$$\begin{aligned} A[i_1, i_2] &= \text{base}_A + ((i_1 - 1) * 20 + i_2 - 1) * 4 \\ &= \text{base}_A + (20i_1 - 20 + i_2 - 1) * 4 \\ &= \text{base}_A + (20i_1 - 21 + i_2) * 4 \\ &= \text{base}_A + 80i_1 - 84 + 4i_2 \end{aligned}$$

base_A, i_1 and i_2 not given so we write them as t_1 & t_2

Now converting into three-address code as follows;

$$t_1 = 80$$

$$t_2 = t_1 * i_1$$

$$t_3 = 4$$

$$t_4 = t_3 * i_2$$

$$t_5 = \text{base}_A + t_2$$

$$t_6 = t_5 - 84$$

$$t_7 = t_6 + t_4$$

$$A[i_1, i_2] = t_7$$

④ Procedure Calls:

param x call p return y

Here, p is a function which takes x as a parameter and returns y .

Procedure calls have the form:

param x_1

param x_2

...

param x_n

call p, n

corresponding to the procedure call $p(x_1, x_2, \dots, x_n)$.

return statement:

They have the form $\text{return } y$, representing a returned value
is optional.

⑤ Back patching: The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. While generating code for boolean expressions and flow-of-control statements during one single pass we may not know the labels that control must go at the time the jump statements are generated. Hence a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels backpatching.