29/2/24

UNIT-3

INTERMEDIATE CODE GENERATOR

Intermediate language :-

* compiler is to convert the source program into machine program directly, ~~but~~ it is not always possible to generate such a machine code directly in one pass, then typically compilers generate an easy to represent form of source language which is called intermediate language.

2m (R)

Properties of intermediate language :-

* The intermediate language is an easy form of source pr language which can be generator efficiently by the compiler.

* the generation of intermediate language should lead to efficient code generation.

* The intermediate code language should be flexible and effective mediator.

Three address code :-

* In three address code form at the most three addresses are used to represent any statement. The general form of three cod address code representation,

$\boxed{a := b \; op \; c}$

* Where a, b c represents name, constants, etc...
op represents operators

$a = b+c+d$

$t_1 = d$	$t_1 = b+c$
$t_2 = c+t_1$	$t_2 = t_1+d$
$t_3 = b+t_2$	$a = t_2$
$a = t_3$	

* t_1, t_2 represents temporary names. There are at the most at three address method (two for operands and one for result).

10m (B) 8m

Implementation of three address code :-

* There are three representations used for three address code such as quadruples, triples, indirect triples

quadruples :- quadtable :-

* The quadruples is a structure with at the most four fields such as op, argument1, argument2, result.

* The op field is used to represent the internal code for operator, the argument1, and argument2 represent the two operands used and result field is used to store the result operands of the expression.

eg:- consider the input $x := -a*b + -a*b$

$t_1 := \text{Uminus } a$
$t_2 := t_1 * b$
$t_3 := \text{Uminus } a$
$t_4 := t_3 * b$
$t_5 := t_2 + t_4$
$x := t_5$

	op	arg1	arg2	result
(0)	Uminus	a		t_1
(1)	*	t_1 (0)	b	t_2
(2)	Uminus	a		t_3
(3)	*	t_3 (2)	b	t_4
(4)	+	t_2 (1)	t_4	t_5
(5)	:=	t_5 (4)		x

Triples :-

* In the triples representation the use of temporary variables is avoided by refering the pointers in the single symbol table.

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

$t_1 := \text{uminus } a$

$t_2 := t_1 * b$

$t_3 := \text{uminus } a$

$t_4 := t_3 * b$

$t_5 := t_2 + t_4$

$x := t_5$

	op	arg1	arg2	result
(0)	uminus	a		
(1)	*	(0)	b	
(2)	uminus	a		
(3)	*	(2)	b	
(4)	+	(2)	(3)	
(5)	:=	(4)		*

Indirect triples :-

* In the indirect triples representation the listing of triples is been done and listing pointers are used instead of using statements.

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

$t_1 := \text{uminus } a$

$t_2 := t_1 * b$

$t_3 := \text{uminus } a$

$t_4 := t_3 * b$

$t_5 := t_2 + t_4$

$x := t_5$

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

	op	arg1	arg2	result
(0)	uminus	a		
(1)	*	(11)	b	
(2)	uminus	a		
(3)	*	(13)	b	
(4)	+	(12)	(14)	
(5)	:=	(15)		x

4/8/24

Execute 3 address code

$-(a*b)+(c+d)-(a+b+c+d)$ Execute the three address code implementation for the above instruction.

Solution :-

$t_1 := a*b$

$t_2 := \text{uminus } t_1$

$t_3 := c+d$

$t_4 := t_2 + t_3$

$t_5 := a+b$

$t_6 := t_5 + t_3$

$t_7 := t_4 - t_6$

Quadruples:-

location	op	arg1	arg2	result
(0)	*	a	b	t1
(1)	uminus	t1		t2
(2)	+	c	d	t3
(3)	+	t2	t3	t4
(4)	+	a	b	t5
(5)	+	t5	t3	t6
(6)	-	t4	t6	t7

Indirect Triples :-

Location	op	arg1	arg2	result
(0)	*	a	b	
(1)	uniminus	(11) (0)		
(2)	+	c	d	
(3)	+	(12) (1)	(13) (2)	
(4)	+	a	b	
(5)	+	(15) (4)	(13) (2)	
(6)	−	(14) (3)	(16) (5)	

Triples :-

	Location	op	arg1	arg2	result
11	(0)	*	a	b	
12	(1)	uniminus	(0) (10)		
13	(2)	+	c	d	
14	(3)	+	(12)	(2) (13)	
15	(4)	+	a	b	
16	(5)	+	(4)	3 (2)	
17	(6)	−	(3)	(5)	

Advantage:-

* The advantage of quadraple representation is that one can quickly access the values of temporary variables using symbol table * the quadraple representation is more beneficial for code optimization.

Disadvantage:-

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

Types of three address code :-

Assignment statement	x: = y op z → binary x: = op z → unary
copy statement	x: = y
unconditional jump	goto L
conditional jump	if x relop y goto L ...
Array statements	x: = y[i] x[i]:= y

Declaration statement :-

In a declarative statements the data items along with their data types are declared.

S → D	{offset : = 0}
D → id : T	{enter table (id. name, T. type, offset); (offset:= offset + T. width)}
T → int	T. type:= integer T. width:= 4
T → real	T. type: = real T. width : = 8
T → array [num] of T1	T. type : = array (num.val, T1. type) T. width: = num. val × T1. width
T. type T → *T1	T. type := pointer (T1. type) T. width : = 4

* Initially, the value of offset is set to 0. The computation of offset can be done by using the formula Offset = offset + width.

* T. type and T. width attributes.

* D → id : T is a declarative statement for id declaration. The entire table is a function used for creating the symbol table entry for the identifier.

* The width of array is obtained by multiplying the width of each element by number of elements in the array.

* The width of pointer type is supposed to be 4.

Assignment statement :-

* Assignment statement mainly deals with the expression. The expression can be type of integer, real, array and record. eg:

10m eg1: obtain the translation schema for obtaining the three address code for the grammar $S \to id := E$

$E \to E1 + E2$

$E \to E1 * E2$

$E \to -E1$

$E \to (E1)$

$E \to id$

solution :-

$S \to id := E$	{enter_table (id.name, E.type)
	if id.name ≠ Nil then
	append (id.name ':=' E.place)
	else
	error;
	}

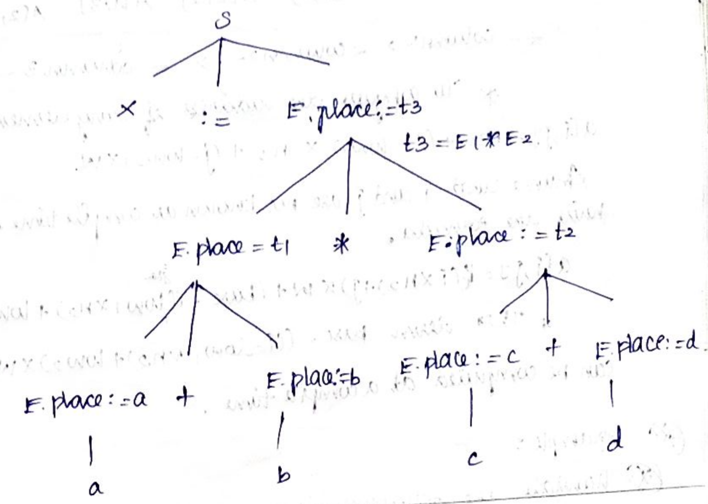
$E \to E1 + E2$	{ E.place = newtemp(); append (E.place ':=' E1.place '+' E2.place)
$E \to E1 * E2$	{E.place = newtemp(); append (E.place ':=' E1.place '*' E2.place)
$E \to -E1$	{E.place = newtemp(); append (E.place ':=' 'uminus' E1.place }
$E \to (E1)$	{E.place := E1.place}

$E \to id$	{enter_table (id.name, E.type)
	if id.name ≠ Nil then
	append (id.name ':=' E.place)
	else error;
	}

Eg2: $x = (a+b) * (c+d)$

solution :-

$E \to id$	E.place = a	
$E \to id$	E.place = b	
$E \to E1 + E2$	E.place = t_1	$t_1 = a + b$
$E \to id$	E.place = c	
$E \to id$	E.place = d	
$E \to E1 + E2$	E.place = t_2	$t_2 = c + d$
$E \to E1 * E2$	E.place = t_3	$t_3 = t_1 * t_2$
$S \to id := E$		$x = t_3$



Array:-

* For Accessing any element of an array what we need its address. for statically declared array it is possible to compute the relative address of each element.

Typically there are 2 representation of array.

1. Row major representation
2. column major representation.

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

← row → ← row2 →

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

← column1 → ← column2 → ← column 3 →

* To compute the address of any element

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

Assume that i and j are not known at compile time, we can write the formula,

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

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

Example:-

Translate the following integer array operations into three address code. $A[i,j] := B[i,j] + C[k]$. where A and B are of size 10x20 and c contains 50 elements.

Solution:-

We will assume, $low_1 = 1$

$low_2 = 1$ and its given that $W = 4$ bytes

$N_1 = 10$

$N_2 = 20$

$A[i,j] := B[i,j] + C[k]$

Formula,

$A[i,j] = ((i \times n_2) + j) \times W + (base_A - ((low_1 \times n_2) + low_2) \times W)$

$= ((i \times 20) + j) \times 4 + (base_A - ((1 \times 20) + 1) \times 4)$

$= 4 \times (20i + j) + (base_A - 84)$

∴ Hence, the three address code for $A[i,j]$ will be

$t_0 = i * 20$

$t_1 = t_0 + j$

$t_2 = c_1$ /* $c_1 = base_A - 84$ */

$t_3 = 4 * t_1$

$t_4 = t_2[t_3]$

* Similarily, the value of $B[i,j]$ can be completed as $t_5 = c_2$

$t_6 = t_5[t_3]$.

The value of $C[k]$ will be obtained as follows is equal to

$k[\cancel{k}] = k * W (base_c - low_1 * W)$

$= k * 4 (base_c - 1 * 4)$

$C[k] = 4k + c_3$

∴ Hence the three address code for $C[k]$ will be

$t_7 = 4 * k$

$t_8 = c_3$ $(c_3 - base_c - 4)$

$t_9 = t_8[t_7]$

Hence the three address code for given expression will be

$t_0 = i * 20$

$t_1 = t_0 + j$

~~$t_2 = c_1$~~

$t_3 = 4 * t_1$

$t_5 = c_2$

$t_6 = t_5[t_3]$

$t_7 = 4 * k$

$t_8 = c_3 (c_3 - base_c - 4)$

$t_9 = t_8[t_7]$

Sasi Rekha. P

$t_{10} = t_6 + t_9$

$t_2 := c_1$

$t_2[t_3] = t_{10}$

Boolean expression:-

* Normally, there are two types of Boolean expression
    * For computing the logical ~~numbe~~ values
    * In conditional expressions using if, then, else or while-do.

Consider the boolean expression,

E → E1 OR E2	{ E.place = newtemp() append (E.place ':=' E1.place 'OR' E2.place)}
E → E1 AND E2	{ E.place = newtemp () append (E.place ':=' E1.place 'AND' E2.place)}
E → NOT E1	{ E.place = newtemp() append (E.place ':=' 'NOT E1.place)}
E → (E1)	{ E.place := E1.place}
E → TRUE	{ E.place = newtemp() append (E.place ':=' 1)}
E → FALSE	{ E.place = newtemp () append (E.place ':=' 0)}

Flow of control statement:-

S → if E then S1

{E.true = newlabel;
E.false = S.next ;
S1.next = S.next ;
S1.code = E.code || gen(E.true ':') || S1.code } ;

E.code
S1.code

⟹ E has a conditional expression.
⟹ S1, S2 is next attribute
⟹ S is code attribute

S → if E then S1 else S2
{ E.true = newlabel ;
E.false = S.nextlabel;
S1.next = S.next ;
S2.next = S.next ;
S.code = E.code || gen (E.true ':') || S1.code ||
gen('goto' S.next) || gen(E.false ':') ||
S2.code }

S → while E do S1

{S.begin = newlabel ;
E.true = newlabel ;
E.false = S.next ;
S1.next = S.begin ;
S.code = gen (S.begin ':') || E.code ||
gen (E.true ':') || S.code ||
gen (goto S.begin) }

c. Generate the three address code for

```
While (i<10)
{
  x = 0;
  i = i+1;
}
```

Solution:-

```
100: L1 : if i <10 goto L2
101 : goto Lnext
102 : L2 : x = 0
103 :  i = i+1
104 :  goto L1
105 : Lnext
```

2. if [(a<b) and (c>d) or (a>d)] then
x = x * y * z
else
x = z+1
construct three address code for above program.

solution :-

100 : if a < b goto 102
101 : ~~if~~ goto 110
102 : if c > d goto 106
103 : goto 104
104 : if a > d goto 106
105 : goto 110
106 : t1 = x+y
107 : t2 = t1 * z
108 : x : = t2
109 : goto 112
110 : t3 = z+1
111 : z : = t3
112 : stop

7/8/24

Backpatching :-

* Backpatching 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 :

i) mklist(i) :- create the new list. the index (i) is passed an argument to this function where I is an index to the array of quadruple.

ii) merge-list(P1,P2) : This function concatenates the list pointer by P1 and P2. It returns the pointer to the concatenated list.

iii) backpatch(p, i) : Insert i as a target label for the statement pointed by P. ~~for the~~

Backpatching using boolean expression :-

Consider the grammar for boolean expression,

$E \rightarrow E1 \text{ OR } M E2$

$E \rightarrow E1 \text{ AND } M E2$

$E \rightarrow \text{NOT } E1$

$E \rightarrow (E1)$

$E \rightarrow id_1 \text{ relop } id_2$

$E \rightarrow \text{TRUE}$

$E \rightarrow \text{FALSE}$

$M \rightarrow \epsilon$

OR (AND - using merge

$\boxed{\text{M-merge}}$

Solution :-

* M is the marker Non-~~functional~~ terminal. the purpose of M is to mark the exact points where the semantic action is picked up.

$E \rightarrow E_1$ OR ME_2	{
	 backpatch (E_1 : Flist , M. state);
	 E. Tlist:= merge (E_1. Tlist , E_2.Tlist);
	 E. Flist = E_2. Flist;
	 }

$E \rightarrow E_1$ AND ME_2	{
	 backpatch (E_1. Tlist, M. state);
	 E. Flist: = merge (E_1. Flist , E_2. Flist);
	 E. Tlist = E_2 . Tlist;
	 }

$E \rightarrow$ NOT E_1	{
	 E. Tlist = E_1 Flist
	 E. Flist = E_1. Tlist
	 }

$E \rightarrow (E_1)$	{ E. Tlist = E_1.Flist E. Flist = E_1 . Tlist }

$E \rightarrow id_1$ relop id_2	{
	 E. Tlist : = mklist (nextstate);
	 E. Flist = mklist (nextstate+1);
	 append ('if' id_1 . place relop
	 id_2 . place 'goto_'),
	 append ('goto_'),
	 }

$E \rightarrow$ TRUE	{
	 E . Tlist := mklist (nextstate);
	 append ('goto_'),
	 }

$E \rightarrow$ FALSE	{
	 E.Flist: = mklist (nextstate);
	 append ('goto_'),
	 }

$M \rightarrow \epsilon$	M. state = nextstate

(8) 2. Using backpatching generate an intermediate code for the following expression and generated automated parse tree.

A < B OR C < D AND P < Q

Solution :-

100 : if A < B goto_

101 : goto_

102 : if C < D goto_ 103 : goto_ 104 : if P < Q goto_ 105 : goto_	backpatch (E_1 . Tlist, 104) M. state = nextstate : 104 E. Tlist = {104} E. Flist = {103, 105}
	backpatch (E_1 . Flist, 103) E. Tlist = {100, 104} E. Flist = {103, 105}

Xml canonical form :-

* It is the use of the algorithm , to generate the canonical form of an xml document .

* The steps during the creation of canonical form include;

i) Encoding the document in unwired character .
ii) Normalizing the line breaks before parsing
iii) Replacing character and parsed Entitle reference .
iv) converting element in the start and end tag parse .

sub-canonicalization :-

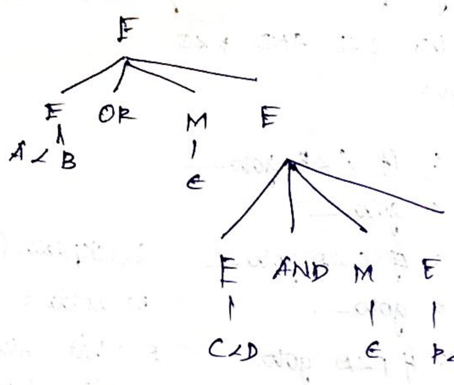
i) Normalizing the attribute value
ii) adding default attribute
iii) Referenced attribute .

Normalization refers to three characteristics :-

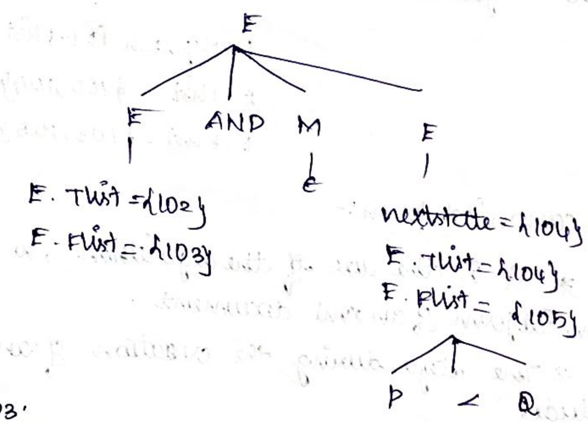
i) timing durability
ii) Environment and profit
iii) ~~Emp~~ ~~expto~~ employability .

Annotated parse tree:-

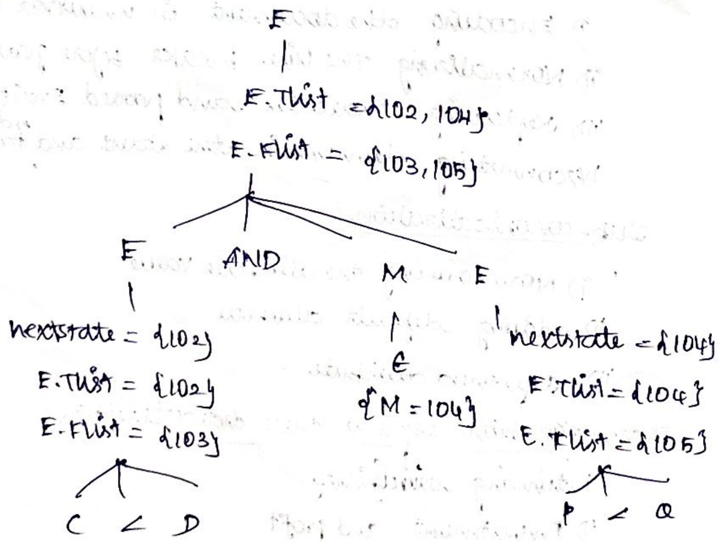
step1:-



step2:-



step3:



step4: