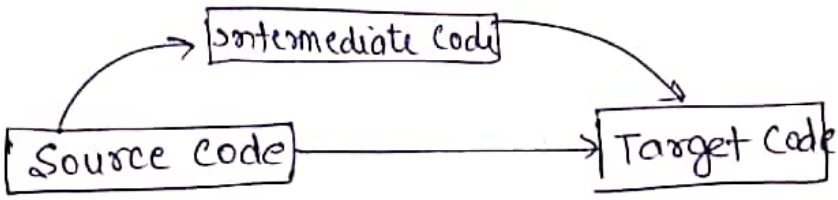
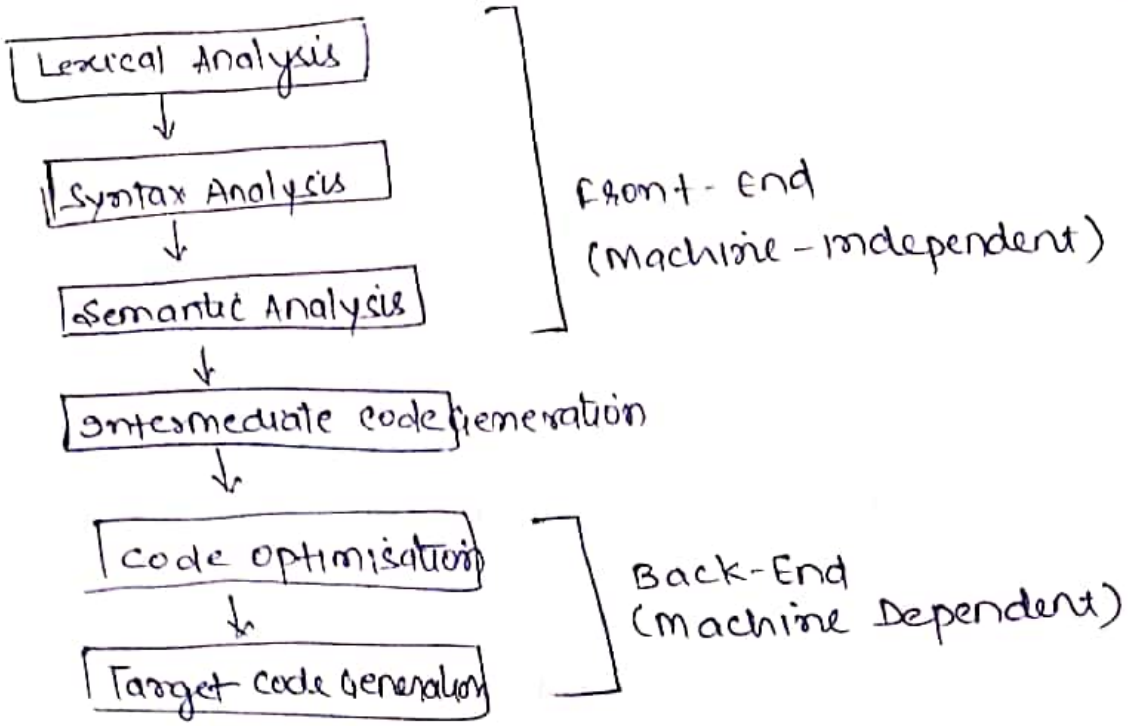
Unit-3

Intermediate Code

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Here are some of the reasons why we need an intermediate code.



- 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.
- A compiler for different source language (on the same machine) can be created by providing different front ends for corresponding source languages to existing back end.

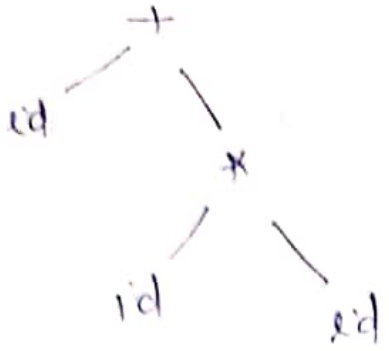


Forms of intermediate code

(1.) Abstract Syntax tree
(2) Polish Notation
(3) Three-Address code

Abstract syntax tree:- Abstract syntax tree or syntax tree is a condensed form of parse tree.
Each node in an abstract syntax tree represents an operator & the children of the operator are the operands (of this operator)

for e.g A sentence id + id * id would have the following syntax tree



(2) ~~Polish Notation~~ Postfix Notation

The ordinary (infix) way of writing the sum of a & 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+.

→ NO paranthesis are needed in postfix notation

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

ab-cd+ * ab- +

- It is also called as Reverse Polish Notation
- It places each binary arithmetic operator after its 2 operands

(3) Three-Address Code:-

A statement involving no more than three references (two for operands & 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

$a := b \ op \ c$

where a, b & c ax operands that can be names, constant, compiler generated temporaries & op represents the operator.

→ only single operation at the right side of the expression is allowed at a time.

for the expression like $a = b + c + d$ the three address code will be

$t_1 := b + c$

$t_2 := t_1 + d$

$a := t_2$

Here t_1 & t_2 ax the temporary names generated by the compiler. There are at the most three addresses allowed (TWO for operands & one for result). Hence the name of this representation is 3-address code.

Q Translate the following expression into Postfix.

(1) $a * (b+c)$ (2) $(a+b) * (cc+d)$

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

Sol

(1) $a*(bc+)$

$\Rightarrow abc+*$

(2) $(a+b) * (cc+d)$

$(ab+) * (ccd+)$

~~(ab~~ $ab+ccd+*$

(3) $a * -(bc+)$

$a*(bc+-)$

$abc+-*$

Q Translate the following into postfix form

(a) if e then x else y

(b) if $\underbrace{a}_{e_1}$ then if $\underbrace{\underbrace{c-d}_{c_2} \text{ then } \overbrace{a+c}^{x_2} \text{ else } \overbrace{a*c}^{y_2}}_{x_1}$ else $\underbrace{a+b}_{y_1}$

Sol (a) exy?

(b) acd-ac+ac*?ab+?

Roots

a x_1 y_1

a (e_2) (x_2) (y_2)

a (cd-) (ac+) (ac*) (ab+)

Evaluation of postfix expression

(1) Scan the postfix code from left to right.

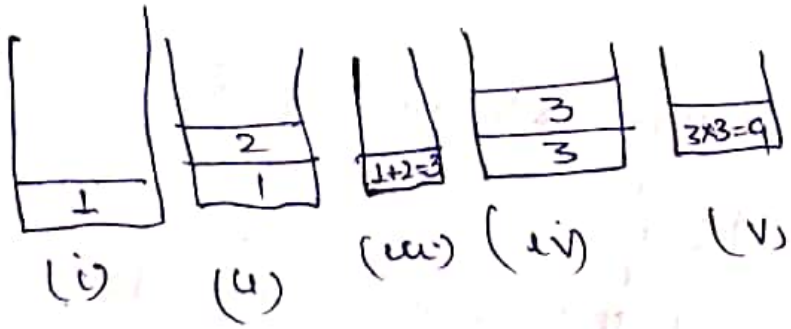
(2) If an operand is encountered, push it onto top of the stack.

3) If a K-ary operator is encountered, Pop the top K-operands from the stack. The resulting value is pushed on the top of stack.

(4) Repeat the above steps until the complete postfix code is scanned.

eg Evaluate the postfix ab+c* using a stack where a=1, b=2, c=3

Sol 1 2 + 3 *

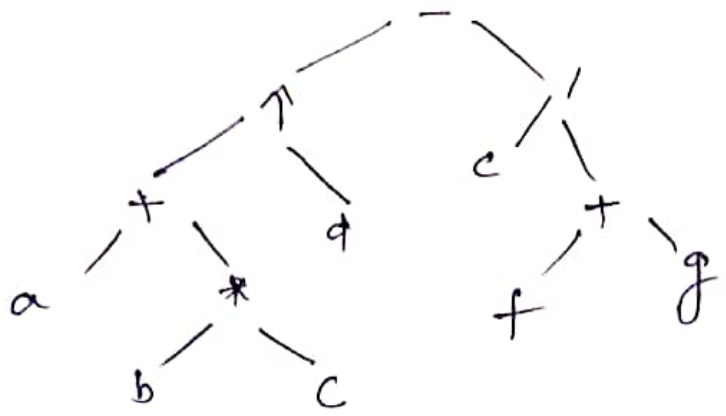


So the o/p is 9.

Q Construct syntax tree & postfix notation for the following expression.

$$(a+(b*c))\uparrow d-e/(f+g)$$

Sol Syntax-tree



Postfix notation

$(a+(b*c))\uparrow d-e/(f+g)$

$(a+T_1)\uparrow d-e/(f+g)$ where $T_1 = bc*$

$T_2 \uparrow d-e/(f+g)$ $T_2 = aT_1 +$

~~$T_2 \uparrow d-e/$~~ $T_3-e/(f+g)$ $T_3 = T_2 d \uparrow$

$T_3 - e/T_4$ $T_4 = fg+$

$T_3 - T_5$ $T_5 = eT_4/$

~~$T_3 T_5$~~ T_6 $T_6 = T_3 T_5 -$

Now substitute the values of temporary variable

$T_3 T_5 -$

$T_3 e T_4 / -$

$T_3 e f g + / -$

$T_2 d \uparrow e f g + / -$

$a T_1 + d \uparrow e f g + / -$

$a b c * + d \uparrow e f g + / -$

Implementation of 3-address code

There are 3 representations used for 3-address code such as quadruples, triples & indirect triples.

Quadruple Representation :- It is a structure with at the most four fields such as

1.) op
2) arg1
3) arg2
4) result.

op field is used to represent the internal code for operator arg1 & arg2 represent the 2 operands used & result field is used to store the result of an expression

e.g Consider the input statement

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

three adress code is

$t_1 = -a$

$t_2 = t_1 * b$

$t_3 = -a$

$t_4 = t_3 * b$

$t_5 = t_2 + t_4$

$x = t_5$

Quadruple

	OP	Arg 1	Arg 2	result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

First number the 3-address instructions

(0) $t_1 = -a$

(1) $t_2 = t_1 * b$

(2) $t_3 = -a$

(3) $t_4 = t_3 * b$

(4) $t_5 = t_2 + t_4$

(5) $x = t_5$

Triples :- In the triple representation the use of temporary variables is avoided instead when a reference to another triple's value is needed, a pointer to that triple is used.

~~$x = -a * b + -a * b$ the triple representation for the expression is as given~~

So, it consist of only three fields namely op, arg1 & arg2.

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

Three address code

$t_1 = \text{uminus } c$

$t_2 = b * t_1$

$t_3 = \text{uminus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

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

Disadvantages of triples

(1) Temporaries are implicit & difficult to rearrange code.

(2) It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Triples for arrays

(1) x[i] := y

	op	arg	arg2
(0)	[]=	x	i
(1)	assign	(0)	y

comments

The L.H.S of the expression is computed.

The computed reference will be one argument & the R.H.S will be a variable

(2) x := y[i]

	op	arg 1	arg2
(0)	=[]	y	i
(1)	assign	x	(0)

comments

Here y[i] is the R.H.S

The computed RHS is assigned to LH.S variable x.

Indirect Triples

This representation makes use of pointer to the listing of all references to computations which is made separately & stored. It is similar in utility as compared to quadruple representation but requires less space than it.

Ex:

#	OP	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	&t+element
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Q Write quadruple, triple & indirect triples for following expression: $(x+y)*(y+z)+(x+y+z)$

Sol three address code is:

$t_1 = x+y$

$t_2 = y+z$

$t_3 = t_1 * t_2$

$t_4 = t_1 + z$

$t_5 = t_3 + t_4$

#	OP	Arg 1	Arg 2	Result
(1)	+	x	y	t_1
(2)	+	y	z	t_2
(3)	*	t_1	t_2	t_3
(4)	+	t_1	z	t_4
(5)	+	t_3	t_4	t_5

Quadruple

#	OP	Arg 1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples

Indirect Triples

#	OP	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

list of pointers to table

#	statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Types of Three Address Statements

Language construct	Intermediate Code form	Meaning
Assignment statement	$x := y \text{ op } z$	Here binary operation is performed using operator 'op'.
Assignment statement	$x = \text{op } y$	Here the unary operation is performed. The operator 'op' is a unary operator.
Copy statement	$x = y$	Here the value of y is assigned to x
Unconditional Jump	goto L	The control flow goes to the statement labeled by L.
Conditional jump	if x relop y goto L	The relop indicates the relational operators such as <, >, <=, >=. If x relop y is true then it executes goto L statement.
Procedure calls	param x_1 param x_2 : param x_n call p, n return y	Here the parameters $x_1, x_2, \ldots x_n$ are used as parameters to the procedure p. The return statement indicates the return value y.

Language Construct	Intermediate code	Meaning
Array statements	$x = y[i]$ $x[i] = y$	The value at i^{th} index of array y is assigned to x The value of identifier y is assigned at the index i of the array x.
Address & pointer assignments	$x = \&y$ $x = *y$ $*x = y$	The value of x will be the address or location of y. The y is a pointer whose value is assigned to x The r value of object pointed by x is set by the r-value of y.

Q Write three address code for the following expression.

if A < B then 1 else 0

Sol

(1) if (A < B) ~~then~~ goto (4)
(2) T1 = 0
(3) goto (5)
(4) T1 = 1
(5) goto --

Q Write three Address code for the following code.

if A < B & C < D then t = 1 else t = 0

(1) if (A < B) ~~then~~ goto (3)
(2) goto (4)
(3) if (C < D) ~~then~~ goto (5)
(4) t = 0
(5) ~~t = 1~~ goto (7)
(6) t = 1
(7) goto --

Q. Three-address code for switch statement

```
switch (i+j)
  {
  case 1 : x = y+z
    default : p = q+r
  case 2 : u = u+w
  }
```

Sol (1) $t_1 = i+j$

(2) if (t_1=1) goto (5)

(3) if (t_1 = 2) goto (8)

(4) goto (11)

(5) $t_2 = y+z$

(6) $x = t_2$

(7) goto (13)

(8) $t_3 = u+w$

(9) $u = t_3$

(10) goto (13)

(11) $t_4 = (q+r)$

(12) $p = t_4$

(13) goto --

Q (19)

```
while (A<C & B>D) do
  if A=1 then C=C+1
  else
    while A<=D
      do A=A+B
```

Sol

(1) if (A<C) goto 3
(2) goto (15)
(3) if (B>D) goto (5)
(4) goto
(5) if (A=1) goto (7)
(6) goto (10)
(7) T1 = C+1
(8) C = T1
(9) goto (1)
(10) if (A<=D) then goto (12)
(11) goto (1)
(12) T2 = A+B
(13) A = T2
(14) goto (10)
(15) goto --

Three address code for do-while statement

Q Generate the three address code for the following code

```
c = 0
do
{
if (a<b) then
    x++;
else
x--;
c++;
} while (c<5)
```

Sol

1. c = 0
2. if (a<b) ~~then~~ goto(4)
3. goto (7)
4. T1 = x+1
5. x = T1
6. goto(9)
7. T2 = x-1
8. x = T2
9. ~~if~~ T3 = C+1
10. C = T3
11. if (C<5) ~~then~~ goto (2)
12. ~~stop~~ goto (13)
13. stop.

OR

(1) c = 0
(2) if (a<b) goto (4)
(3) goto (7)
(4) x = x+1;
(5) ~~goto~~ C = C+1
(6) goto (10)
(7) x = x-1
(8) ~~goto (9)~~ C = C+1
(9) goto (10)
(10) if (C<5) goto (2)
(11) goto (12)
(12) stop

Q Three address code for the following code (21)

```
int i
i=1
while i<10 do
    if x>y then
    i=x+y
    else
    i=x-y
```

Note:- for declaration part we don't write 3-address code.

Sol (1) i=1

(2) if (i<10) goto (4)

(3) goto (10)

(4) if (x>y) goto (6)

(5) goto (8)

(6) i=x+y

(7) goto (2)

(8) i=x-y

(9) goto (2)

(10) stop

Three address code of do-while condition

do $x = y + z$ while $(a < b)$

(1) ~~x+0~~ $t_1 = y + z$

(2) $x = t_1$

(3) if $(a < b)$ ~~goto~~ goto (1)

(4) goto . .

(1) $t_1 = y + z$

(2) $x = t_1$

(3) if $(a < b)$ goto (1)

(4) ~~next~~ Stop

8

```
c=0
do
{if (a<b) then
  x++;
```

Three address code for procedure call

Suppose

$P(A_1, A_2 \dots A_n)$

this is given

Three address code for the above statement

```
param A1
param A2
 !
param An
call P, n
```

P → Name of procedure.
n → No of actual parameter

eg

```
void main()   // this is also a function
 {
int x, y;
swap(&x, &y);
 }
void swap(int *a, int *b)
{
  int i;
i = *b;
*b = *a;
*a = i;
}
```

(1) call main
(2) Param &x.
(3) param &y
(4) call swap, 2
(5) i = *b
(6) *b = *a
(7) *a = i
(8) stop.

Q Three address code for for statement

For loop

```
for (i=1; i<=20; i++)
    x=y+z;
```

Sol

(1) i = 1
(2) if (i<=20) goto (3)
(3) goto (10)
(4) $t_1 = i+1$
(5) $i = t_1$
(6) goto (2)
(7) $t_2 = y+z$
(8) $x = t_2$
(9) goto (4)
(10) stop