INTERMEDIATE CODE REPRESENTATIONS

INTERMEDIATE CODE REPRESENTATIONS

- Syntax Tree
- Postfix notation
- Three Address Code
- DAG- Directed Acyclic Graph.

Parse Tree

Grammar

E → E + T

E → E - T

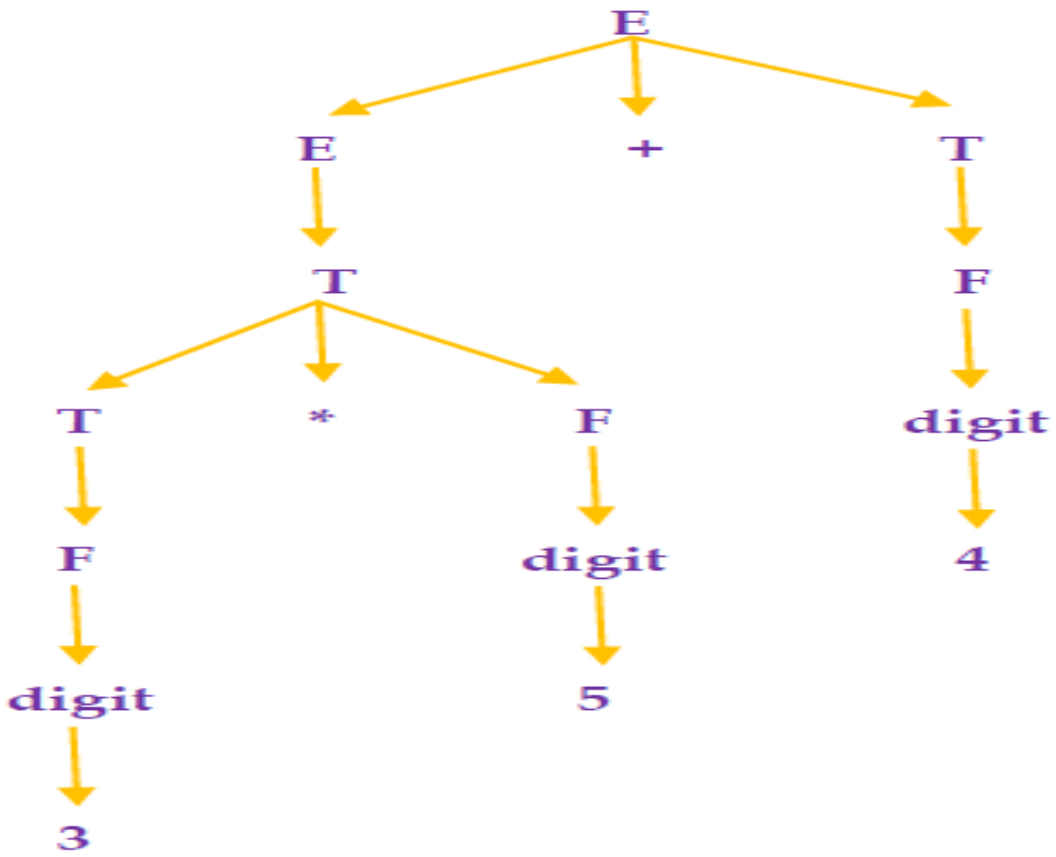
E→ T

T→T * F

T→F

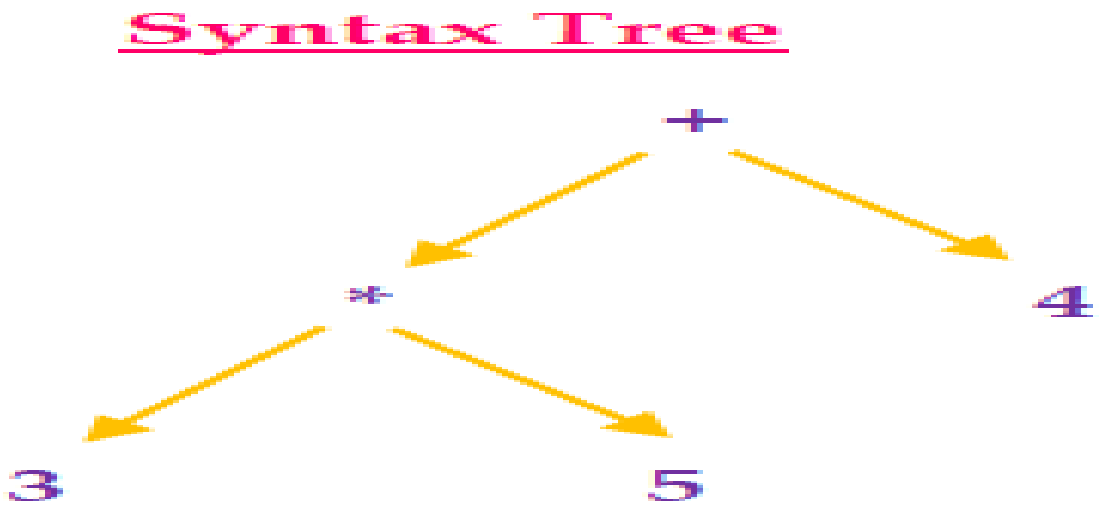
F→ digit

Parse Tree



Syntax Tree Or Abstract Syntax Tree(AST)

- syntax tree for **3 * 5 + 4**



POSTFIX NOTATION

1. If E is a variable,eg: 2 then the postfix notation for E is E or 2 itself.

2. If E is an expression of the form E1 op E2 then postfix notation for E is E1 E2 op

3. If E is an expression of the form (E*F), then the postfix notation for E is the same as the postfix notation for EF*.

4. For unary operation –E the postfix is E-

Ex: postfix notation for 9- (5+2) is 952+-

Postfix notation of an infix expression can be obtained using stack

POSTFIX NOTATION

(A+B/C*(D+E)-F)			
Symbol	**Stack**	**Postfix**	
(	(		//Push
A	(	A	//Added to postfix expression
+	(+		//Push
B	(+	AB	//Added to postfix expression
/	(+/		> precedence than + so added to stack
C	(+/	ABC	//Added to postfix expression
*	(+*	ABC/	// = precedence remove / from stack
(	(+*(		
D	(+*(	ABC/D	//Added to postfix expression
+	(+*(+		//Push
E	(+*(+	ABC/DE	//Added to postfix expression
)	(+*	ABC/DE+	// closing bracket remove+
-	(-	ABC/DE+*+	// * have >precedence than -,-&+ equal precedence
F	(-	ABC/DE+*+F	//Added to postfix expression
)		ABC/DE+*+F-	// closing bracket remove -

THREE ADDRESS CODE

- Intermediate code can be represented by means of three address statements.
- Atmost three addresses are used to represent any three address statement & only one operator on the right hand side.
- x = y* z
- x,y,z are the 3 addresses used here,these operands can be names,constants or compiler generated temporaries.
- t2 = x + t1
- t2,t1 are compiler generated temporaries.

Implementation of Three Address Statements

- **QUADRUPLES**
- **TRIPLES**

Translate the following expression to quadruple triple and indirect triple

- Q: a + b * c | e ^ f + b * a

- Three Address Code(1. **( )** 2. ^ 3. *, / 4. +, **-)from left to right.**

1. t1 = e ^ f

2. t2 = b * c

3. t3 = t2 / t1

4. t4 = b * a

5. t5 = a + t3

6. t6 = t5 + t4

QUADRUPLES

- A quadruple is a record structure with four fields, which are ***op, ag1, arg2*** and ***result*** **.**
- The three address statement **x = y * z** is represented by placing **y** in ***arg1, z*** in ***arg2*** and **x** in ***result*** and operator is *.
- Uses large no: of temporary variables, each temporary variables has its entry in symbol table.

QUADRUPLES

Location	OP	arg1	arg2	Result
(0)	^	e	f	t1
(1)	*	b	c	t2
(2)	/	t2	t1	t3
(3)	*	b	a	t4
(4)	+	a	t3	t5
(5)	+	t3	t4	t6

QUADRUPLES

- Advantages
  1. direct access of the location for temporaries by means of symbol table.
  2. statements can often move around which makes optimization easier, for example if we move a statement computing **x**, the statement using **x** requires no change.
- Disadvantages
  1. Large amount memory wastage.

TRIPLES

- In triples representation, the use of temporary variables is avoided & instead reference to instructions are made.
- So three address statements can be represented by records with only there fields OP, arg1 & arg2.

Location	OP	Arg1	Arg2
(0)	^	e	f
(1)	*	b	c
(2)	/	(1)	(0)
(3)	*	b	a
(4)	+	a	(2)
(5)	+	(2)	(3)

TRIPLES

- Advantages
  1. No need to use temporary variable which saves memory as well as time.

- Disadvntages
  1. Triple representation is difficult to use for optimizing compilers,for statement shuffling cannot be done.

Q: Translate the following expression to quadruple triples and Postfix notation & syntax tree

- **a = b * - c + b * - c**
- TAC
- t1 = uniminus c
- t2 = b* t1
- t3 = uniminus c
- t4 = b* t3
- t5 = t2 + t4
- a = t5

Translate the following expression to quadruple triples and Postfix notation & syntax tree

- QUADRUPLES

Location	OP	Arg1	Arg2	Result
(0)	Unary minus	c		T1
(1)	*	b	T1	T2
(2)	Unary minus	c		T3
(3)	*	b	T3	T4
(4)	+	T2	T4	T5
(5)	=	T5		a

Translate the following expression to quadruple triples and Postfix notation & syntax tree

- TRIPLES

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