5.2.1 INTERMEDIATE LANGUAGES

The most commonly used intermediate representations were :-

- **Syntax Tree**
- **DAG (Direct Acyclic Graph)**
- **Postfix Notation**
- **3 Address Code**

5.2.1.1 GRAPHICAL REPRESENTATION

Includes both

- **Syntax Tree**
- **DAG (Direct Acyclic Graph)**

Syntax Tree Or Abstract Syntax Tree(AST)

- Graphical Intermediate Representation
- Syntax Tree depicts the hierarchical structure of a source program.
- Syntax tree (AST) is a condensed form of parse tree useful for representing language constructs.

EXAMPLE

Parse tree and syntax tree for **3 * 5 + 4** as follows.

Grammar

E → E + T

E → E - T

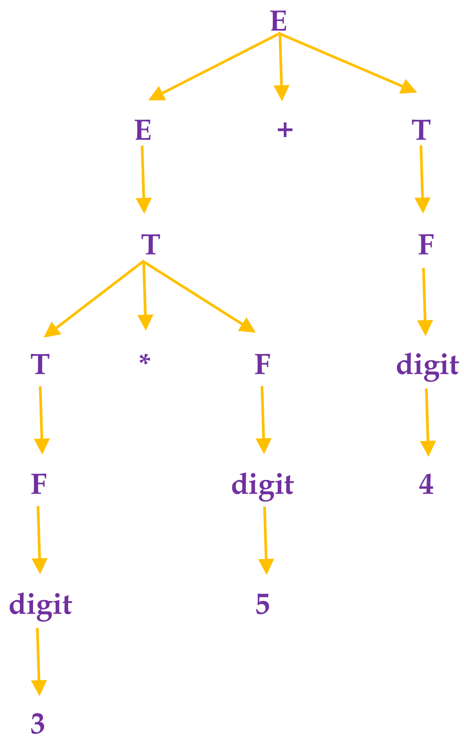
E → T

T → T * F

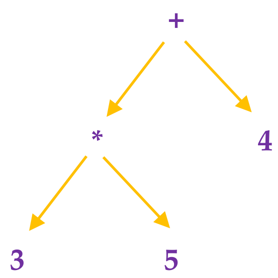
T → F

F → digit

Parse Tree



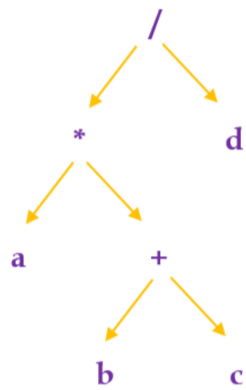
Syntax Tree



Parse Tree VS Syntax Tree

Parse Tree	Syntax Tree
A parse tree is a graphical representation of a replacement process in a derivation	A syntax tree (AST) is a condensed form of parse tree
Each interior node represents a grammar rule	Each interior node represents an operator
Each leaf node represents a terminal	Each leaf node represents an operand
Parse tree represent every detail from the real syntax	Syntax tree does not represent every detail from the real syntax Eg : No parenthesis

Syntax tree for a * (b + c) /d



Constructing Syntax Tree For Expression

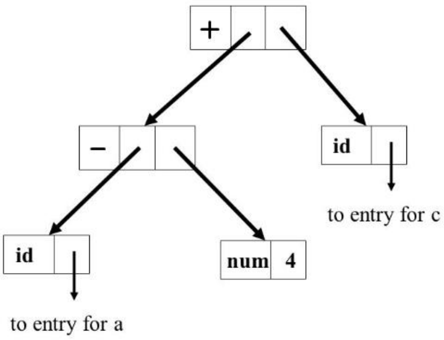
- Each node in a syntax tree can be implemented in arecord with several fields.
- In the node of an operator, one field contains operator and remaining field contains pointer to the nodes for the operands.
- When used for translation, the nodes in a syntax tree may contain addition of fields to hold the values of attributes attached to the node.
- Following functions are used to create syntax tree
    1. **mknode(op,left,right)**: creates an operator node with label op and two fields containing pointers to left and right.
    2. **mkleaf(id,entry)**: creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry for identifier
    3. **mkleaf(num,val)**: creates a number node with label num and a field containing val, the value of the number.
- Such functions return a pointer to a newly created node.

EXAMPLE

a – 4 + c

The tree is constructed bottom up

P_1 = **mkleaf(id,entry a)**
P_2 = **mkleaf(num, 4)**
P_3 = **mknode(-, P_1, P_2)**
P_4 = **mkleaf(id,entry c)**
P_5 = **mknode(+, P_3, P_4)**



Syntax Tree

Syntax directed definition

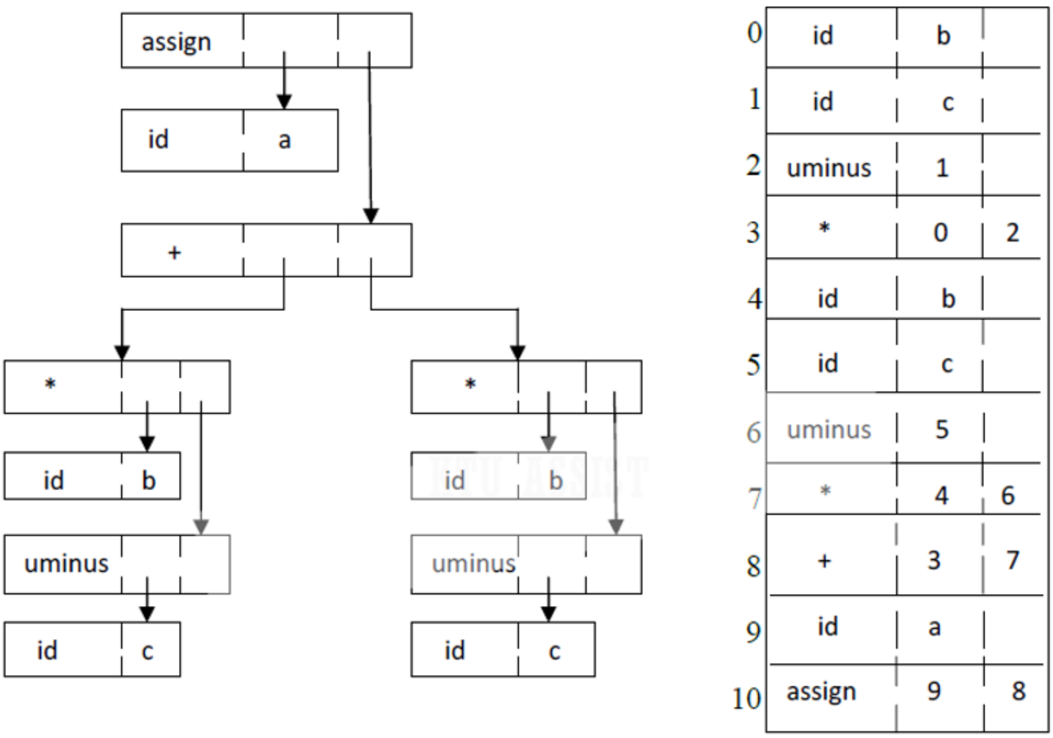
- Syntax trees for assignment statements are produced by the syntax-directed definition.
- Non terminal S generates an assignment statement.
- The two binary operators + and * are examples of the full operator set in a typical language. Operator associates and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input a:=b* -c + b* -c

PRODUCTION	SEMANTIC RULE
S → id : = E	S.nptr : = mknode('assign',mkleaf(id, id.place), E.nptr)
E → E_1 + E_2	E.nptr : = mknode('+', E_1.nptr, E_2.nptr)
E → E_1 * E_2	E.nptr : = mknode('*', E_1.nptr, E_2.nptr)
E → - E_1	E.nptr : = mknode('uminus', E_1.nptr)
E → (E_1)	E.nptr : = E_1.nptr
E → id	E.nptr : = mkleaf(id, id.place)

Syntax-directed definition to produce syntax trees for assignment statements

- The token id has an attribute *place* that points to the symbol-table entry for the identifier.
- A symbol-table entry can be found from an attribute **id.***name*, representing the lexeme associated with that occurrence of id.

- If the lexical analyser holds all lexemes in a single array of characters, then attribute name might be the index of the first character of the lexeme.
- Two representations of the syntax tree are as follows.



(a) (b)

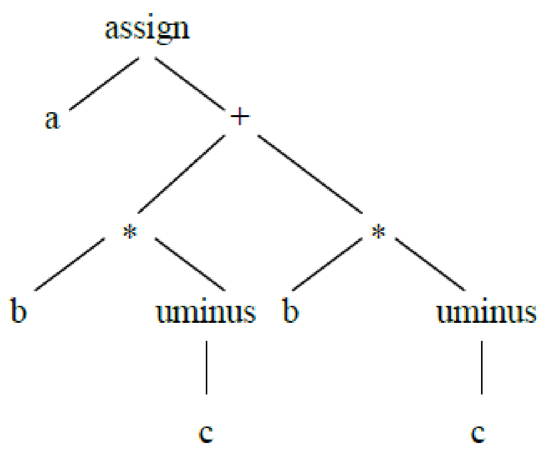
- In (a), each node is represented as a record with a field for its operator and additional fields for pointers to its children.
- In Fig (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node.
- All the nodes in the syntax tree can be visited by following winters, starting from the root at position 10.

Direct Acyclic Graph (DAG)

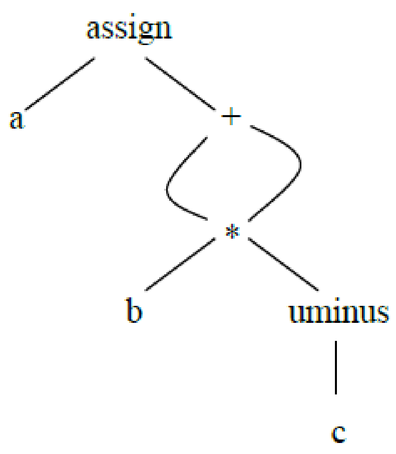
- Graphical Intermediate Representation
- Dag also gives the hierarchical structure of source program but in a more compact way because common sub expressions are identified.

<u>EXAMPLE</u>

a=b*-c + b*-c



(a) Syntax tree



(b) Dag

Postfix Notation

- Linearized representation of syntax tree
- In postfix notation, each operator appears immediately after its last operand.
- Operators can be evaluated in the order in which they appear in the string

<u>EXAMPLE</u>

Source String : a := b * -c + b * -c

Postfix String: a b c uminus * b c uminus * + assign

<u>Postfix Rules</u>

1. If E is a variable or constant, then the postfix notation for E is E itself.
2. If E is an expression of the form E1 op E2 then postfix notation for E is E1′ E2′ op, here E1′ and E2′ are the postfix notations for E1and E2, respectively
3. If E is an expression of the form (E), then the postfix notation for E is the same as the postfix notation for E.
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

5.2.1.2 THREE-ADDRESS CODE

- In Three address statement, at most 3 addresses are used to represent any statement.
- The reason for the term "three address code" is that each statement contains 3 addresses at most. Two for the operands and one for the result.

General Form Of 3 Address Code

a = b op c

where,

a, b, c are the operands that can be names, constants or compiler generated temporaries.

op represents operator, such as fixed or floating point arithmetic operator or a logical operator on Boolean valued data. Thus a source language expression like **x + y * z** might be translated into a sequence

$t_1 := y*z$

$t_2 := x+t_1$ where, t_1 and t_2 are compiler generated temporary names.

Advantages Of Three Address Code

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged - unlike postfix notation.

Three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph.

Three Address Code corresponding to the syntax tree and DAG given above (page no:)

(a) Code for the syntax tree	(b) Code for the dag
$t_1 := -c$	$t_1 := -c$
$t_2 := b * t_1$	$t_2 := b * t_1$
$t_3 := -c$	$t_5 := t_2 + t_2$
$t_4 := b * t_3$	$a := t_5$
$t_5 := t_2 + t_4$	
$a := t_5$	

Types of Three-Address Statements

1. **Assignment statements**

   x := y op z, where op is a binary arithmetic or logical operation.

2. **Assignment instructions**

   x : = *op* y, where op is a unary operation . Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that for example, convert a fixed-point number to a floating-point number.

3. **Copy statements**

   x : = y where the value of y is assigned to x**.**

4. **Unconditional jump**

   goto L The three-address statement with label L is the next to be executed

5. **Conditional jump**

   if x relop y goto L This instruction applies a relational operator **(<, =, =, etc,)** to **x** and y, and executes the statement with label **L** next if x stands in relation **relop** to **y.** If not, the three-address statement following if **x relop y goto L** is executed next, as in the usual sequence.

6. **Procedural call and return**

   param x and **call p, n** for procedure calls and **return y**, where **y** representing a returned value is optional. Their typical use is as the sequence of three-address statements

   param x_1
   param x_2
   ……….
   param x_n
   call p,n

   generated as part of the call procedure **p(x_1 , x_2, . . . , x_n)** . The integer **n** indicating the number of actual-parameters in **"call p , *n*"** is not redundant because calls can be nested.

7. **Indexed Assignments**

   Indexed assignments of the form **x = y[i] or x[i] = y**

8. **Address and pointer assignments**

   Address and pointer operator of the form **x := &y**, **x := *y** and ***x := y**

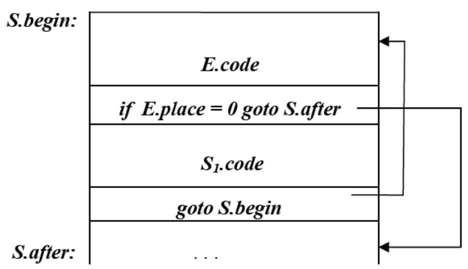
Syntax-Directed Translation Into Three-Address Code

- When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. for example **id** : = **E** consists of code to evaluate E into some temporary **t**, followed by the assignment **id**.*place* : = **t**.
- Given input **a:= b * - c + b + - c**, it produces the three address code in given above (page no:) The synthesized attribute **S.code** represents the three address code for the assignment S. The nonterminal E has two attributes:
    1. **E.***place* the name that will hold the value of E, and
    2. **E.**code. the sequence of three-address statements evaluating E.

Syntax-directed definition to produce three-address code for assignments.

PRODUCTION	SEMANTIC RULES
$S \rightarrow id : = E$	*S.code : = E.code || gen(id.place ':=' E.place)*
$E \rightarrow E_1 + E_2$	*E.place := newtemp;* *E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)*
$E \rightarrow E_1 * E_2$	*E.place := newtemp;* *E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)*
$E \rightarrow - E_1$	*E.place := newtemp;* *E.code := E_1.code || gen(E.place ':=' 'uminus' E_1.place)*
$E \rightarrow (E_1)$	*E.place : = E_1.place;* *E.code : = E_1.code*
$E \rightarrow id$	*E.place : = id.place;* *E.code : = ' '*

Semantic rule generating code for a while statement



PRODUCTION	SEMANTIC RULES
S → while *E* do S_1	*S.begin := newlabel;* *S.after := newlabel;* *S.code := gen(S.begin ':') ||* *E.code ||* *gen ('if' E.place '=' '0' 'goto' S.after)||* *S_1.code ||* *gen ('goto' S.begin) ||* *gen (S.after ':')*

- The function ***newtemp*** returns a sequence of distinct names **t_1, t_2**,……… in respose of successive calls. Notation *gen*(x ':= 'y '+' z is used to represent the three address statement **x := y + z.**
- Expressions appearing instead of variables like **x, y** and **z** are evaluated when passed to ***gen,*** and quoted operators or operand, like '**+**' are taken literally.
- Flow of control statements can be added to the language of assignments. The code for **S→ while E do S_1** is generated using new attributes ***S.begin*** and ***S.after*** to mark the first statement in the code for E and the statement following the code for **S**, respectively.
- The function ***newlabel*** returns a new label every time is called. We assume that a nonzero expression represents true; that is when the value of ***E*** becomes zero, control laves the while statement

Implementation Of Three-Address Statements

A three address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such, representations are

- **Quadruples**
- **Triples**
- **Indirect triples**

5.2.1.3 QUADRUPLES

- A quadruple is a record structure with four fields, which are ***op, ag1, arg2*** and ***result***
- The op field contains an internal code for the operator. The three address statement **x:= y op z** is represented by placing **y** in ***arg1***, **z** in ***arg2*** and **x** in ***result***.
- The contents of ***arg1, arg2***, and ***result*** are normally pointers to the symbol table entries for the names represented by these fields. If so temporary names must be entered into the symbol table as they are created.

EXAMPLE 1

Translate the following expression to quadruple triple and indirect triple

a + b * c | e ^ f + b * a

For the first construct the three address code for the expression

t1 = e ^ f
t2 = b * c
t3 = t2 / t1
t4 = b * a
t5 = a + t3
t6 = t5 + t4

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

Exceptions

- ⇨ The statement **x := op y**, where op is a unary operator is represented by placing **op** in the operator field, **y** in the argument field & n in the result field. The ***arg2*** is not used
- ⇨ A statement like **param t1** is represented by placing **param** in the operator field and t1 in the arg1 field. Neither ***arg2*** not result field is used
- ⇨ Unconditional & Conditional jump statements are represented by placing the target in the result field.

5.2.1.4 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.
- Since, there fields are used this intermediated code formal is known as triples

Advantages

- ❖ No need to use temporary variable which saves memory as well as time

Disadvantages

- ❖ Triple representation is difficult to use for optimizing compilers
- ❖ Because for optimization statements need to be suffled.
- ❖ for e.g. statement 1 can be come down or statement 2 can go up ect.
- ❖ So the reference we used in their representation will change.

EXAMPLE 1

a + b * c | e ^ f + b * a

t1 = e ^ f
t2 = b * c
t3 = t2 / t1
t4 = b * a
t5 = a + t3
t6 = t5 + t4

Location	OP	arg1	arg2
(0)	^	e	f
(1)	*	b	c
(2)	/	(1)	(0)
(3)	*	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

EXAMPLE 2

A ternary operation like x[i] : = y requires two entries in the triple structure while x : = y[i] is naturally represented as two operations.

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

x[i] := y

	op	*arg1*	*arg2*
(0)	= [ ]	y	i
(1)	assign	x	(0)

x := y[i]

INDIRECT TRIPLES

- This representation is an enhancement over triple representation.
- It uses an additional instruction array to led the pointer to the triples in the desired order.
- Since, it uses pointers instead of position to stage reposition the expression to produce an optimized code.

EXAMPLE 1

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

Location	op	arg1	arg2
(0)	^	E	f
(1)	*	B	c
(2)	/	(1)	(0)
(3)	*	B	a
(4)	+	A	(2)
(5)	+	(4)	(3)