UNIT - IV

- **Intermediate Code Generation (ICG)**: Intermediate languages – Graphical representations, Three Address code, Quadruples, Triples.

- **Code Optimization**: Principal sources of optimization, Optimization of Basic blocks.

- Intermediate languages
- Implementation of 3-address statements
- Translation of simple statements and control flow statements.

Intermediate Code Generation

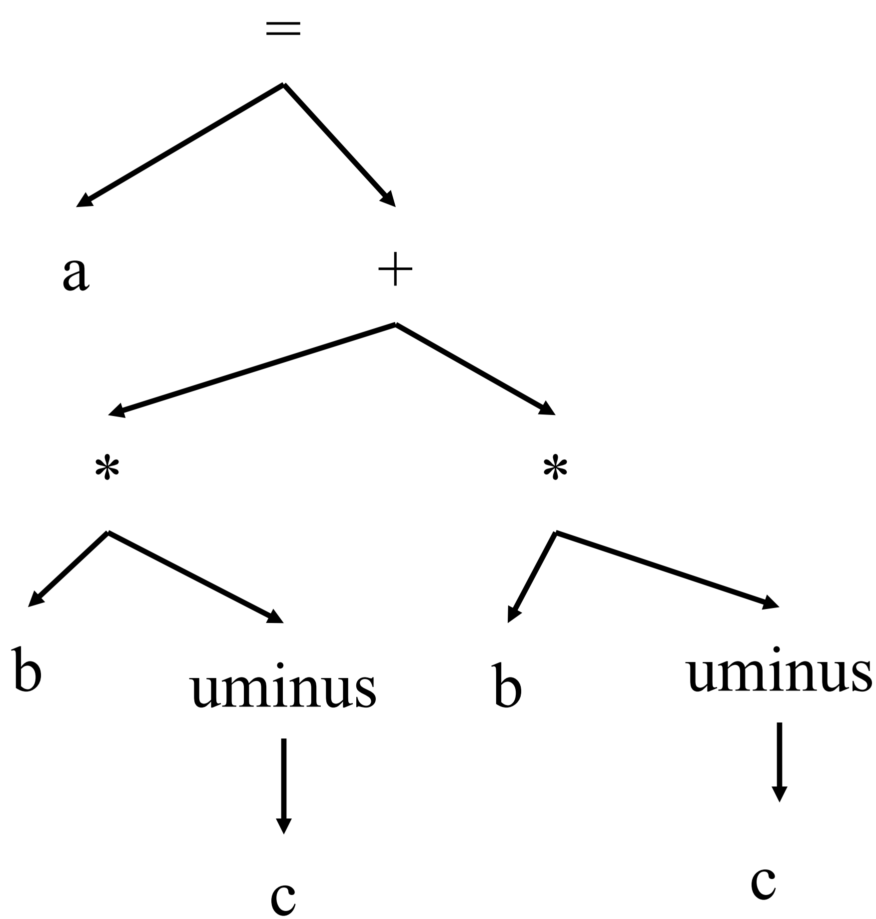
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.

- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.

  - syntax trees (abstract syntax trees)
  - postfix notation
  - three-address code

Syntax Trees

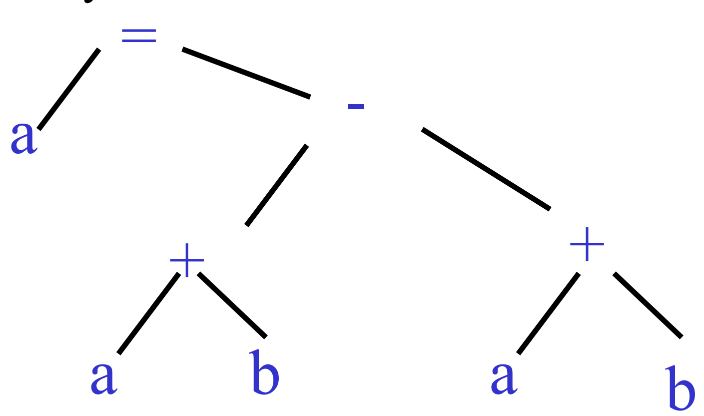
- Consider the assignment **a=b*-c+b*-c**



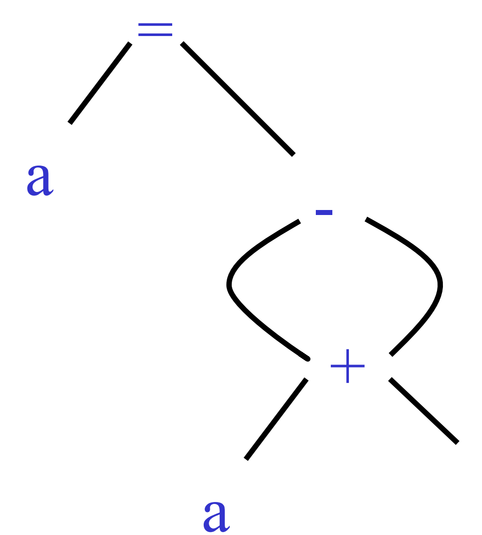
DAG (Directed Acyclic Graph)

- A **dag** is similar to syntax tree , and it identifies the common sub expressions in the expression.
- A node in a dag representing a common sub expression has more than one parent .
- In a syntax tree, the common sub expressions would be represented as duplicated subtree.
-
- Ex : consider an expression : a= (a+b) - (a+b)

Syntax tree

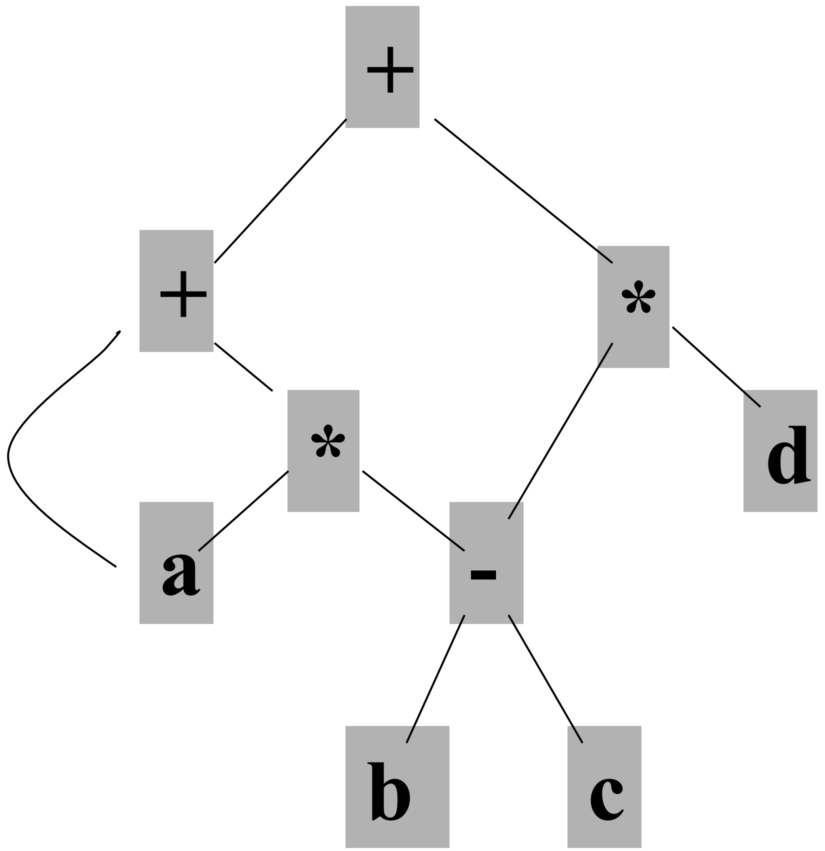


DAG



Directed Acyclic Graphs for Expressions

a + a * (b – c) + (b – c) *d



Postfix notation

Postfix notation is a linear representation of a syntax trees.

Consider an expression : a= b * -c + b * -c

postfix notation : **a b c - * b c - * + =**

Constructing Syntax Trees for Expressions

1. Syntax trees are constructed using following functions:
   i. ***mknode(op , left, right) : creates an operator node with label op and two fields containing pointers to left child and right child.***
   ii. ***mkleaf (id ,entry) : creates an identifier node with label id and a field containing entry, a pointer to the symbol table.***
   iii. ***mkleaf (num , val) : creates a number node with label num and a field containing val , the value of the number***

2. **Each function returns a pointer to a newly created node**

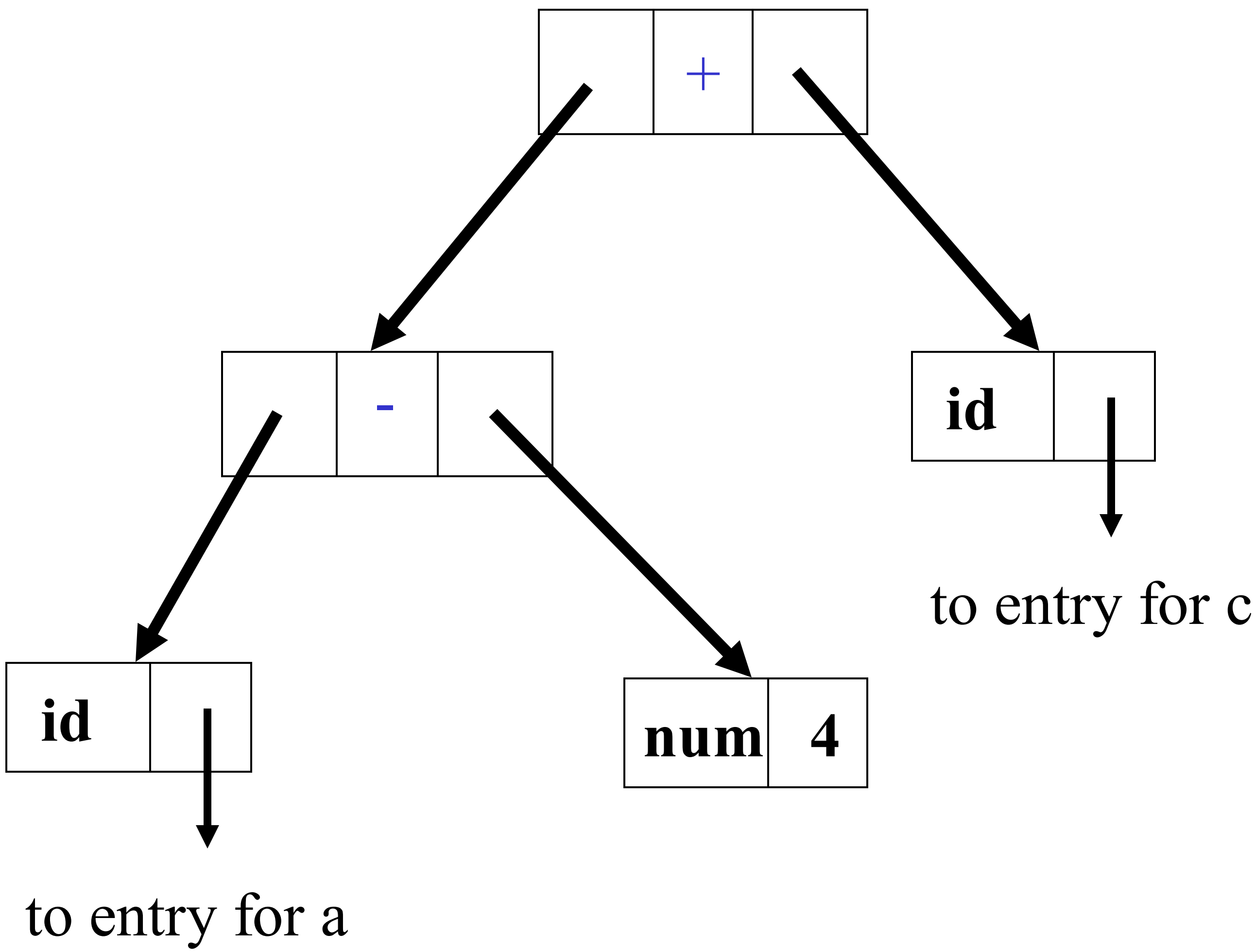
The following sequence of function calls creates the syntax tree for the expression : **a-4+c**

- **1) p1= mkleaf (id , entrya);**
- **2) p2=mkleaf(num , 4);**
- **3) p3=mknode('-' , p1 ,p2);**
- **4) p4=mkleaf(id , entryc);**
- **5) p5=mknode('+' , p3 , p4);**

- **In this sequence , p1 ,p2 , ……p5 are pointers to nodes**
- **entrya and entryc are pointers to symbol table entries for identifiers a and c**
- **The syntax tree is constructed bottom up.**

Draw the Syntax Tree

a-4+c



SDD for Syntax Trees

PRODUCTION	**SEMANTIC RULE**
$E \rightarrow E_1 + T$	$E.\mathit{nptr} = \mathit{mknode}(\text{'+'}, E_1.\mathit{nptr}, T.\mathit{nptr})$
$E \rightarrow E_1 - T$	$E.\mathit{nptr} = \mathit{mknode}(\text{'-'}, E_1.\mathit{nptr}, T.\mathit{nptr})$
$E \rightarrow T$	$E.\mathit{nptr} = T.\mathit{nptr}$
$T \rightarrow (E)$	$T.\mathit{nptr} = E.\mathit{nptr}$
$T \rightarrow \mathbf{id}$	$T.\mathit{nptr} = \mathit{mkleaf}(\mathbf{id}, \mathbf{id}.\mathit{entry})$
$T \rightarrow \mathbf{num}$	$T.\mathit{nptr} = \mathit{mkleaf}(\mathbf{num}, \mathbf{num}.\mathit{val})$

The synthesized attribute nptr for E and T keeps track of the pointers returned by the function calls

Three Address Code

- **Three address code is a linear representation of syntax tree**
- **Statements of general form** $x = y \text{ op } z$
- **Ex:** $x = y + z * w$
  should be represented as

$$t_1 = z * w$$
$$t_2 = y + t_1$$
$$x = t_2$$

- **Where t_1 and t_2 are compiler generated temporary names**

Example of 3-address code

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

t_1=- c
t_2=b * t_1
t_3=- c
t_4=b * t_3
t_5=t_2 + t_4
a=t_5

Code for the syntax tree

t_1=- c
t_2=b * t_1
t_5=t_2 + t_2
a=t_5

Code for the DAG

Types of Three-Address Statements.

Assignment Statement:	**x=y op z**
Assignment Statement:	**x=op z (op is unary operator)**
Copy Statement:	**x=z**
Unconditional Jump:	**goto L**
Conditional Jump:	**if x relop y goto L**
Stack Operations:	**Push/pop**

Procedure (function):

param x_1
param x_2
…
param x_n
call p,n

Index Assignments:

x=y[i]
x[i]=y

Address and Pointer Assignments:

x=&y
x=*y
*x=y

Implementations of 3-address statements

- In compiler, The Three Address statements are implemented using following representations:

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

- **A quadruple is a record with four fields , op , arg1, arg2 and result.**

op	arg1	arg2	result

Implementations of 3-address statements using quadruple

- 3-address code

t_1=- c
t_2=b * t_1
t_3=- c
t_4=b * t_3
t_5=t_2 + t_4
a=t_5

	op	*arg1*	*arg2*	*result*
(0)	**uminus**	**c**		$\mathbf{t_1}$
(1)	*	**b**	$\mathbf{t_1}$	$\mathbf{t_2}$
(2)	**uminus**	**c**		$\mathbf{t_3}$
(3)	*	**b**	$\mathbf{t_3}$	$\mathbf{t_4}$
(4)	+	$\mathbf{t_2}$	$\mathbf{t_4}$	$\mathbf{t_5}$
(5)	=	$\mathbf{t_5}$		**a**

Temporary names must be entered into the symbol table as they are created.

Implementations of 3-address statements

- Triples

t_1=- c
t_2=b * t_1
t_3=- c
t_4=b * t_3
t_5=t_2 + t_4
a=t_5

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

Temporary names are not entered into the symbol table.

Other types of 3-address statements

- e.g. ternary operations like

  x[i]=y　　　　x=y[i]

- require two or more entries. e.g.

x[i]=y

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

x=y[i]

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

Implementations of 3-address statements

- ## Indirect Triples

	statement
(0)	**(14)**
(1)	**(15)**
(2)	**(16)**
(3)	**(17)**
(4)	**(18)**
(5)	**(19)**

triple

	op	*arg1*	*arg2*
(14)	**uminus**	**c**	
(15)	*	**b**	**(14)**
(16)	**uminus**	**c**	
(17)	*	**b**	**(16)**
(18)	+	**(15)**	**(17)**
(19)	**assign**	**a**	**(18)**

Syntax-Directed Translation into 3-address code.

1. Assignment statements:

- The following grammar is defined for assignment statements:

  S->id=E

  E->E1+E2 | E1*E2 | -E1 | (E1) | id

  In order to write syntax directed definition for the above grammar, the following attributes are defined for the non terminal E:

  - E.*place:* the name that will hold the value of E
    - Identifier will be assumed to already have the place attribute defined.
  - E.*code* :hold the three address code statements that evaluate E (this is the `translation' attribute).

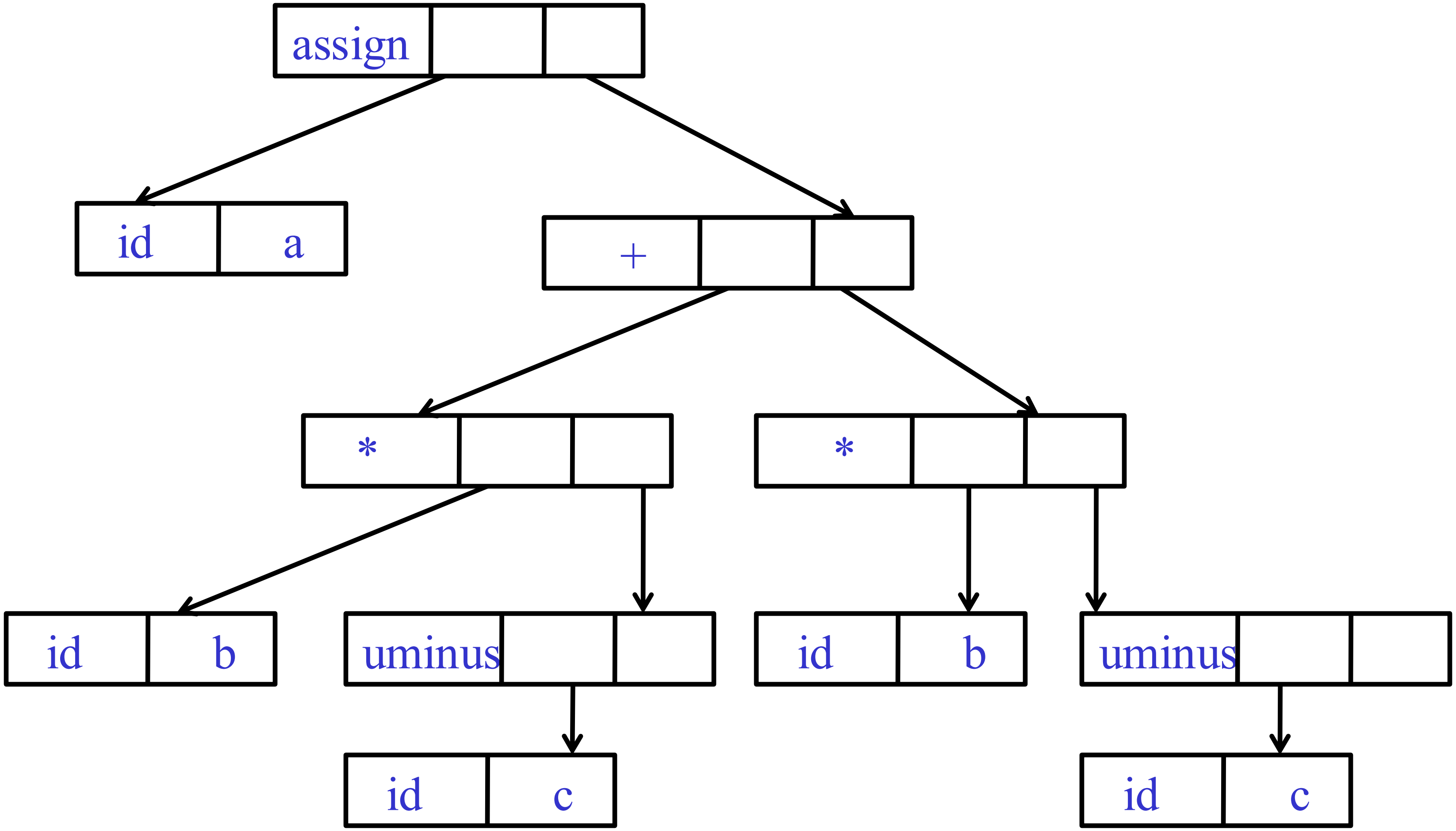
- The function ***newtemp*** returns sequence of temporary variables.
- The function ***gen*** generates a single three address statement.

Ex: gen(x ':=' y '+' z) represent the 3-address statement x := y+z

SDD for Syntax Trees using function calls

PRODUCTION	**SEMANTIC RULE**
S → id := E	**E.*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 Tree for a = b*-c +b* -c using function calls



Syntax-Dir. Definition for 3-address code

PRODUCTION	**SEMANTIC RULE**
$S \rightarrow id := E$	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.*entry* ; E.*code* = ‘’

Boolean Expressions:

- **Boolean expressions are composed of the Boolean operators (*and* , *or*, and *not*) applied to the elements that are Boolean variables or relational expressions.**

- **E → E *or* E | E *and* E | *not* E |(E) | id1 *relop* id2 | true | false**

Methods of implementing Boolean expressions

- There are two principal methods of representing the value of a Boolean expression.
- The first method is to encode true and false numerically and to evaluate a Boolean expression analogously to an arithmetic expression.
- The second principle method is by flow of control , that is , representing the value of a Boolean expression by a position reached in a program. Here we have adopted the first method

consider the conditional statement :

if a<b 1 else 0

Numerical representation:

Ex:

100: if a<b goto 103

101: t1 := 0

102: goto 104

103: t1 := 1

104:

Translation scheme for producing 3-address code for above example

E → *id1 relop id2*

{E.place := newtemp;
emit ('if' *id1*.place *relop*.op *id2*.place '*goto*' nextstat+3)
emit (E.place ':=' '0')
emit ('*goto*' nextstat+2)
emit (E.place ':=' '1') }

Assume that
emit places three Address statements into an output file

nextstat gives the index of the three address statement in the output sequence

emit increments *nextstat* after producing each three address statement

Semantic Actions for producing Three Address Codes for Boolean Expressions:

E → E1 *or* E2

{ E.place := newtemp();
emit (E.place ':=' E1.place '*or*' E2.place); }

E → E1 *and* E2

{ E.place := newtemp();
emit (E.place ':=' E1.place '*and*' E2.place);}

E → *not* E

{ E.place := newtemp();
emit (E.place ':=' '*not*' E.place); }

E → (E1)

{ E.place := E1.place }

E → *id1 relop id2*

{ E.place := newtemp;
emit ('if' *id1*.place *relop*.op *id2*.place '*goto*' nextstat+3)
emit (E.place ':=' '0')
emit ('*goto*' nextstat+2)
emit (E.place ':=' '1') }

E → *true*

{ E.place := newtemp;
emit (E.place ':=' '1') }

E → *false*

{ E.place = newtemp;
emit (E.place ':=' '0') }

Flow of control statements:

In second method the boolean expressions are evaluated using flow of control statements ;if-then ,if –else ; while-do.

The following grammar is used to produce such statements:-

S → if E then S1
S → if E then S1 else S2
S → while E do S1

Functions and Attributes used in the translation of control Statements :-

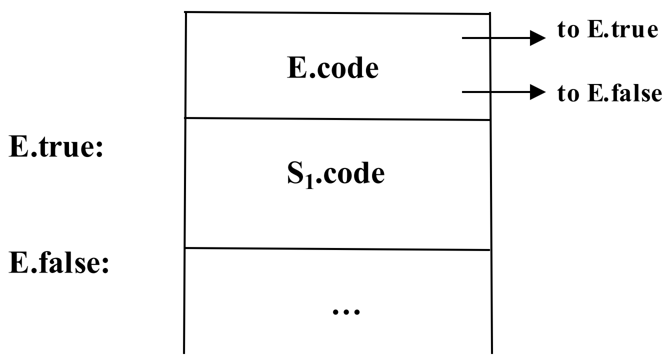
Flow of control statements may be converted to three address code by use of the following functions:-

- newlabel – returns a new symbolic label each time it is called.
- gen ( ) – “generates” the code (string) passed as a parameter to it.

The following attributes are associated with the non-terminals for the code generation:-

- code – contains the generated three address code.
- true – contains the label to which a jump takes place if the Boolean expression associated (if any) evaluates to “true”.
- false – contains the label to which a jump takes place if the Boolean expression (if any) associated evaluates to “false”.
- begin – contains the label / address pointing to the beginning of the code chunk for the statement “generated” (if any) by the non-terminal.
- next - contains the label / address pointing to the end of the code chunk for the statement “generated” (if any) by the non-terminal

- The simulation of the flow of control branching for each statement is depicted pictorially as follows



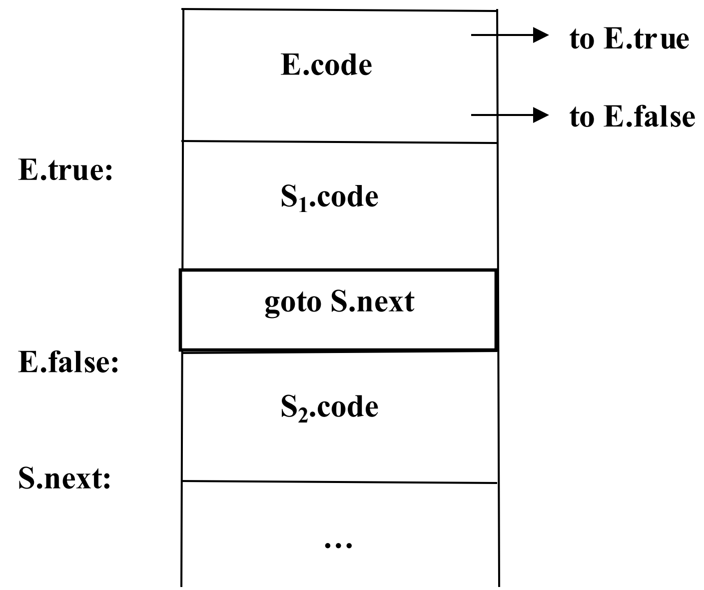
if - then

S → if E then S1

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

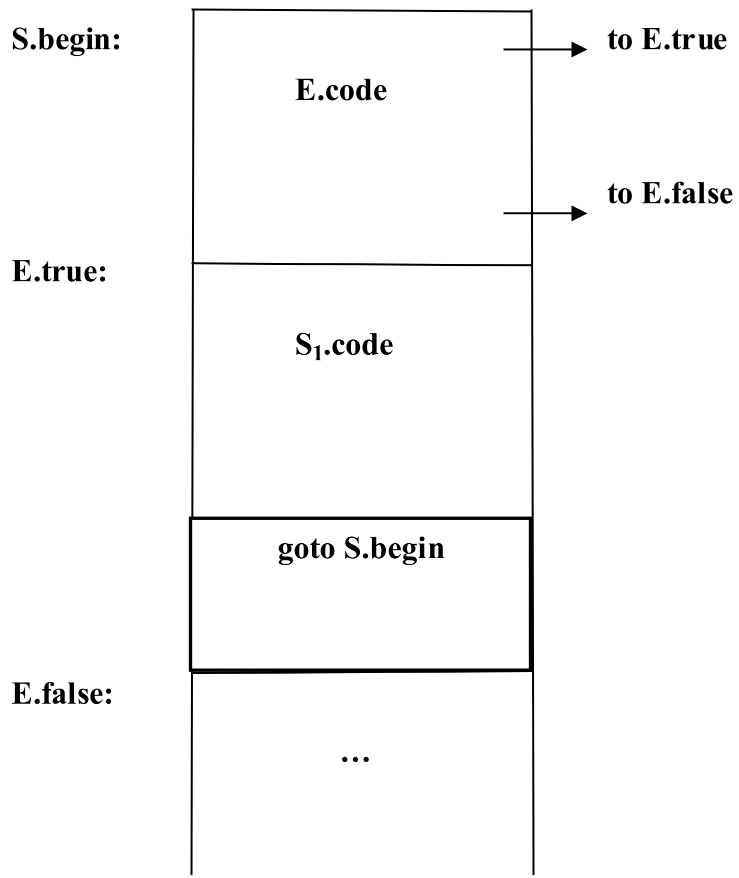
S → if E then S1 else S2

E.true := newlabel ;
E.false := newlabel ;
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



if – then - else

S → while E do S1



while - do

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 ‘:’) || S1.code || gen(‘goto’ S.begin)