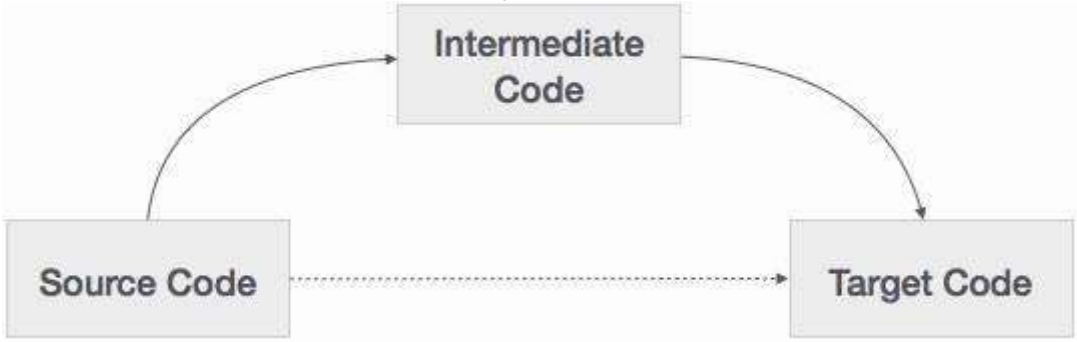
Unit -VI
Intermediate Code Generation

Intermediate languages, declarations, assignments statements, Boolean expressions, case statements, back patching, procedure calls.

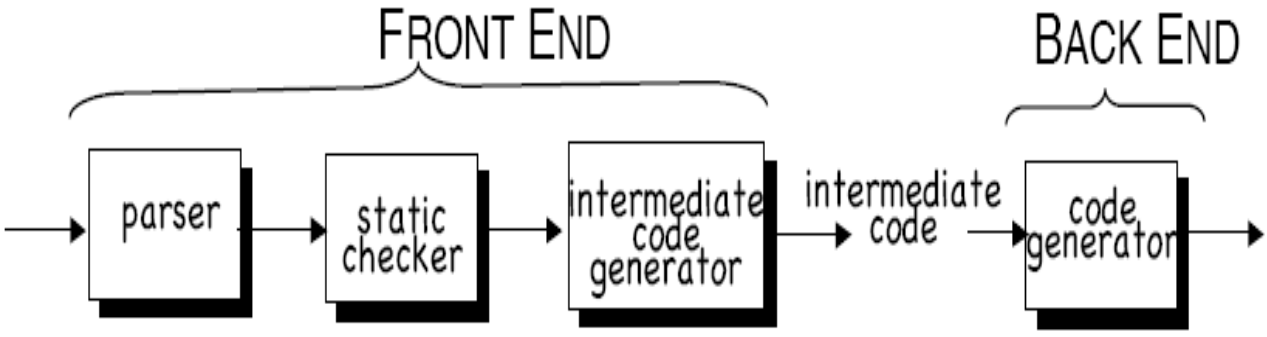
Introduction:

A source code can be directly translated into its target machine code, then why do we have to need to translate the source code into an intermediate code which is then translated to its target code? Let us see 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 each new machine required its own inbuilt compiler.
- Intermediate code eliminates the need for a new inbuilt compiler for every unique machine by keeping the analysis portions the same for all the compilers.
- The second part of the compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques to the intermediate code.

In intermediate code generation, we use syntax-directed methods to translate the source program into an intermediate form programming language construct such as declaration, assignments and flow-of-control statements.



Intermediate code is often the link between the compiler's front end and back end.

Building compilers this way makes it easy to retarget code to a new architecture or do machine-independent optimization.

Intermediate Representation:

There are three types of intermediate representation:

1) Syntax trees
2) Postfix notation

3) Three address code

The semantic rule for generating three address codes from a common programming language construct is similar to those for constructing syntax trees of generated postfix notation.

Graphical representation:

A syntax tree draws the natural hierarchical structure of the source program. A DGA (Directed Acyclic Graph) gives the same information but in a more compact way because some common sub-expression are identified and these common expressions are eliminated. A syntax tree for the assignment statement a:=b*-c+b*-c appears in the following figure 1.

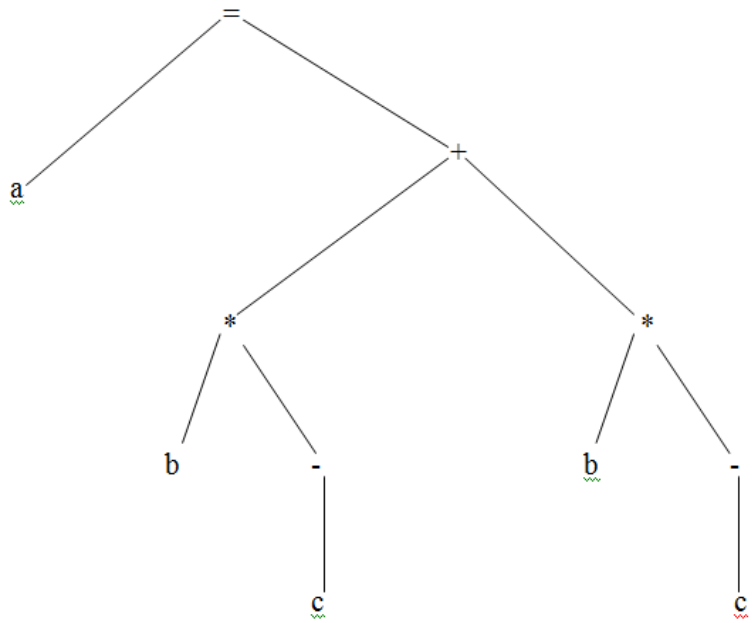


Figure1. Syntax tree for a=b*-c +b*-c

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes in which a node appears immediately after its children. The postfix notation for the syntax tree for the above figure is,

abc-*bc-*+=

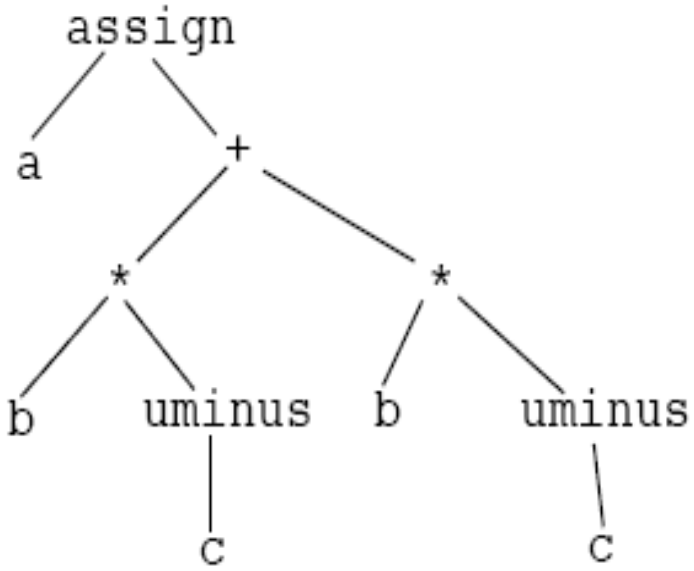
The edges in a syntax tree do not appear in postfix notation. They can be recovered in the order in which the nodes appear and the number of operands that the operator at a node expects. The recovery of edges is done from expression in postfix notation.

Syntax trees for an assignment statement are produced by the syntax-directed definition in figure 2.

Production	**Semantic Rules**
S -> id:=E	S.nptr :=mknode(‘:=’, mkleaf(id, id, place), E.nptr)
E ->E1+E2	E.nptr :=mknode(‘+’,E1.nptr ,E2.nptr)
E ->E1*E2	E.nptr :=mknode(‘*’,E1.nptr ,E2.nptr)
E ->-E1	E.nptr :=mknode(‘-’,E1.nptr)
E ->(E1)	E.nptr :=E1.nptr
E ->id	E.nptr := mkleaf (id, id, place)

Figure 2. Syntax directed definition

This same syntax-directed definition will produce the dag if the function mknode (op1, child) and mknode (op, left, right) return a pointer to an existing node whenever possible, instead of constructing new nodes. The token id has an attribute place that points to the symbol-table entry for the identifier id. name to represent the lexeme associated with the occurrence of id. If the lexical analyzer holds the all lexemes in a single array of characters, then the attribute name might be the index of the first character of the lexeme. Two representations of the syntax tree appear. Each node is represented as a record with a field for its operator and if additional fields for pointers to its children. Nodes are allocated from an array of records and the index or position of node servers as the pointer to the node. All the nodes in the syntax tree can be visited by position Io.



Syntax Tree for a:=b*-c+b*-c

Equivalently, we can use POSTFIX: a b c uminus * b c uminus * + assign

(Postfix is convenient because it can run on an abstract stack machine)

Position	**OP**	**Arg1**	**Arg2**
0	Id	b	
1	Id	c	
2	uminus	1	
3	*	0	2
4	Id	b	
5	Id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	Id	a	
10	Assign	9	8

mkleaf() and mknode() table

Three Address code:-

Three address codes are a sequence of statements of the general form:

X: = Y op Z

Where X, Y and Z are names (Variables, Non-terminals), constants, or compiler-generated temporaries; op stands for an operator such as a fixed or floating-point arithmetic operator or a logical operator on boolean-valued data. Note that no built-up arithmetic expressions are permitted, as there is **only one operator on the right side of a statement**. Thus a source language expression like x+y*z can be translated into a sequence,

t1:=y*z

t2:=x+t1

Where t1 and t2 are compiler-generated temporary names. This complicated arithmetic expression and of the nested flow of the control statement make three address codes desirable for target code generation and optimization. The use of names for the intermediate values computed by a program allows the three address codes to be easily rearranged, unlike postfix notation. Three address codes are a linearized representation of a syntax tree a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three address code sequences. A variable name can appear directly in three address statements so has no statements corresponding to the leaves in the figure.

Code for Syntax Tree	**Code for DAG**
t1:= uminus(c) t2:= b*t1 t3:= uminus(c) t4:= b*t3 t5:= t2+t4 a:=t5	t1: = uminus(c) t2:= b*t1 t3=t2+t2 a:=t3

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

The reason for the term “Three address code” is that each statement usually contains three addresses, two for the operands and one for the result. In the implementations of three address codes, a programmer's defined name is replaced by a pointer to a symbol table entry for that name.

Types of Three Address Code Statements:-

Three address statements are a like to assembly code. Statements can have symbolic labels and there are statements for the flow of control. A symbolic label represents the index of three address statements in the array holding intermediate code. Actual indices (index) can be sub-situated. For the labels either by making a separate pass or by using “back patching”. Here are the common three address statements used:

a) Assignment statements of form x: =y op z, where op is binary arithmetic or logical operation.
b) Assignment instruction of form x: = op y, where unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators e.g. converting a fixed point number to a floating-point number.
c) Copy statements of form x: =y where the value of y is assigned to x.

d) The unconditional jump goto L. The three address statements with label L are the next to be executed.

e) Conditional jumps such as if x relop y goto L. This instruction applies a relational operator (<, =,>, <=,>=,!=, etc.) to x and y, and executes the statements with the 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 the usual sequence.

f) Param x and call p, n for produce calls and return y, where y represents a returned value is optional. The typical uses of the sequence of three address statements are given below:

```
Param x1
Param x2
Param xn
Call p, n
```

Generated part of a call of the procedure is p(x1, x2…,xn). The integer n indicating the number of actual parameters in "call p, n" is not redundant because the call can be nested.

g) Indexed assignments of the form x: =y[i] and x[i]:=y. The first of these sets x to the value in the location i memory units outside location y. The statement x[i]:=y sets the contents of the location i units beyond x to the value y. In both these instructions x, y and i refer to data objects.

h) Address and pointer assignments of the form x: =&y, x: =*y, *x: =y. The first of these sets the value x to be the location of y. Most probably y is a name, possibly a temporary, that denotes on expression with an 1-value such as

A [i, j] and x is a pointer name or temporary. That is the 1-value of x is the 1-value (location) of some object. In this statement x: =y1 presumably y is a pointer or a temporary whose 1-value is a location. The 1-value of x is made equal to the contents of the location. Finally, +x: =y sets the 1-value of the object pointed to by to the 1-value of y.

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set must be rich enough to implement the operations in the source language. A small operator set is easier to implement on a new target machine. However, a restricted instruction set may force the front end to generate long sequences of statements for some source, language operations. The optimizer and code generator may then have to work harder if good code is to be generated.

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. The value of non-terminal E on the left side of

E->E1+E will be computed into a new temporary t. In general three-address code for id: =E consists of code to evaluate E into some temporary t, followed by the assignment id.place:=t. If an expression is a single identifier, say y then y itself holds the value of the expression. For the movement, we create a new name every time a temporary is needed; a technique for

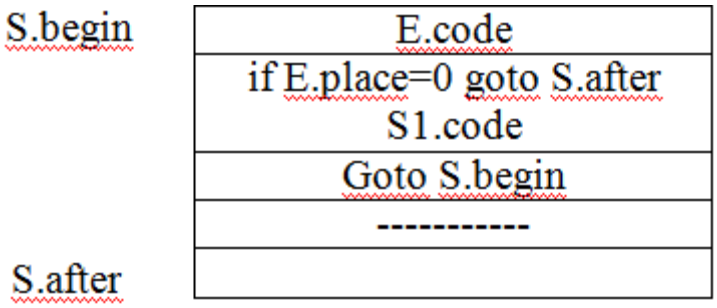
reusing temporaries is given in the figure. The s-attribute definition in the figure generates a three-address code for the assignment statement: given input a: =b+-c+b+-c, it produces the code in the figure. The synthesized attribute s.code represents the three-address code in the figure. The synthesized attribute s.code represents the three-address code for the assignments s. The non-terminal E has two attributes as below:

1) E.place the name that will hold the value of E&
2) E.code the sequence of three-address code statements evaluating E.

The function new temp returns a sequence of distinct names t1, t2… in response to successive calls. For convenience, we use the notation gen ('x':='y'+'z') in the figure to represent the three-address statement x: =y+z. Expressions appearing instead of operands like '+' are taken literally. In practice, three-address statements might be sent to an output file, rather than built up into the code attributes. Flow-of-control statements can be added to the language of assignments in production and semantic rules like the ones for while statements in the figure. In the figure, the code for s-while E do s, 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 s1 resp.

For ASSIGNMENT statements and expressions, we can use this SDD:

Production	Semantic Rules
S -> id := E	S.code := E.code || gen(id.place ':=' E.place)
E -> E1 + E2	E.place := newtemp();
	E.code := E1.code || E2.code ||
	gen(E.place ':=' E1.place '+' E2.place)
E -> E1 * E2	E.place := newtemp();
	E.code := E1.code || E2.code ||
	gen(E.place ':=' E1.place '*' E2.place)
E -> - E1	E.place := newtemp();
	E.code := E1.code || gen(E.place ':=' 'uminus' E1.place)
E -> (E1)	E.place := E1.place; E.code := E1.code
E -> id	E.place := id.place; E.code := ''



This attribute represents labels created by a function new label that returns a new label every time it is called. Note that s.after becomes the label of the statements that come after the code for the while statement. We assume that a non-zero expression represents true i.e. when the value of F becomes zero, control leaves the while statement. F: an expression that governs the flow of control may in general Boolean expression containing relational and logical operators. The semantic rules for while statements in the section differ from those in

the figure to allow for control within Boolean expressions. Postfix notation can be obtained by adapting the semantic rules in the table. The postfix notation for an identifier is the identifier itself. The rule for the other productions is to concatenate only the operator after the code for the operands.

E,g. Associated with the production E-E1 is the semantic rule

E.code := E1.code||'-'.

Implementation of the three-address statement:-

A three-address statement is an abstract form of intermediate code. These statements can be implemented as records with fields of operator and operands. The descriptions of three-address instructions specify the component of each type of instruction. In a compiler, these instructions can be implemented as objects or records with fields implemented as objects or records with fields for operators and operands. Three such representations are called quadruples, triples, and indirect triples.

1) Quadruples: -

A quadruple or just quad has 4 fields which we called op, arg1, arg2, and result. Operator minus is used to distinguish the unary minus operator.

- op: the operator
- arg1: the first operand
- arg2: the second operand
- result: the destination

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

Three Address Code:

t1 := (c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5

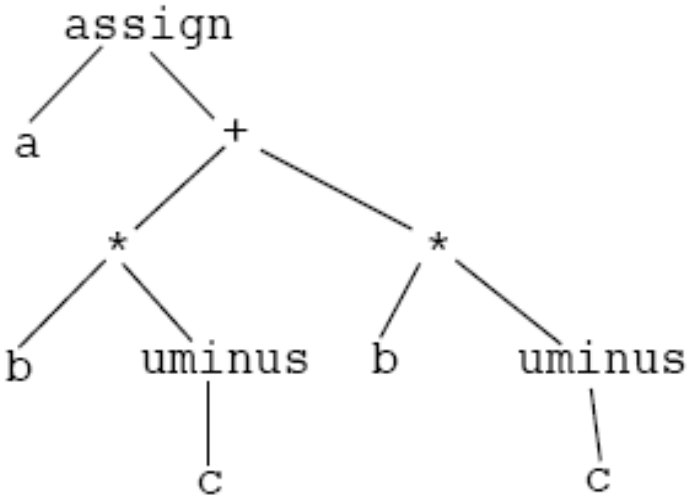
Quadruples:

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

2) Triples: -

Each instruction in triples presentation has three fields: op, arg1, and arg2. The results of respective sub-expressions are denoted by the position of expression. Triples represent

similarities with DAG and syntax tree. They are equivalent to DAG while representing expressions.



Syntax tree for a:=b*-c+b*-c

Position	Op	arg_1	arg_2
0	Uminus	c	
1	*	b	(0)
2	Uminus	c	
3	*	b	(2)
4	+	(1)	(3)
5	Assign	a	(4)

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

Position	Op	arg_1	arg_2
0	[ ]	x	I
1	assign	(0)	Y

Triples for x[i]=y

Position	Op	arg_1	arg_2
0	[ ]	y	i
1	assign	x	(0)

Triples for x=y[i]

Triples face the problem of code immovability while optimization, as the results are positional, and changing the order or position of an expression may cause problems.

3) Indirect triples: -

This representation is an enhancement over triples representation. It uses pointers instead of positions to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code. It uses array pointers to triples in the desired order.

Position	Pointer Index
0	21
1	22
2	23
3	24
4	25
5	26

Pointer Index	Op	arg_1	arg_2
21	Uminus	C	
22	*	B	(21)
23	Uminus	C	
24	*	B	(23)
25	+	(22)	(24)
26	Assign	A	(25)

Position and array pointer index	Indirect triples for a:=b*-c+b*-c

Declarations:-

A variable or procedure has to be declared before it can be used. Declaration involves the allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

Example:

We take the example of C programming language where an integer variable is assigned 2 bytes of memory and a float variable is assigned 4 bytes of memory.

int a;
float b;
Allocation process:
{offset = 0}
int a;
id.type = int
id.width = 2
offset = offset + id.width
{offset = 2}
float b;
id.type = float
id.width = 4
offset = offset + id.width
{offset = 6}

To enter this detail in a symbol table, a procedure enter can be used. This method may have the following structure:

enter(name, type, offset)

This procedure should create an entry in the symbol table, for a variable name, having its type set to type and relative address offset in its data area.

Keeping Track of scope Information:-

When nested procedure processing of declaration is temporarily suspended. e.g.

P->D
D->D;D|id:T proc id; D;S

The production for non-terminal S for statements and T for types is not shown because we focus on the declaration. For simplicity suppose that there is a separate symbol table for each procedure in language. One possible implementation of the symbol table is

using a linked list for entries of names. A new symbol table is created when procedure declaration.

D-> proc id ; D; S and enters for the declaration in D are created in the new table.

Storage Layout:-

The type and relative address are saved in the symbol table entry for the name. For string, compile-time data is not sufficient, hence required runtime memory allocation. This storage comes in blocks of contiguous bytes where a byte is the smallest unit of address memory.

Assignment Statement: -

A lookup operation is used to check whether the entry is present in the symbol table or not for names. Lookup (idname) is present then the pointer returns otherwise lookup returns nil to indicate that no entry was found.

e.g.

P->MD

M->E

D->D; S|id: T|Proc id; N D ; S

N->E

Non-terminal P becomes the new start symbol when these productions are added to the symbol table. The expression can be of type integer, real, array and records. As part of the translation of assignments into three address codes, we show how names can be lookup in the symbol table and how elements of array and records can be accessed.

T->TT1 {T.type:pointer(T1.type);T.width=4}

This translation schema is represented in the symbol table. Each symbol table has a header contacting a pointer to the table for the enclosing procedure. A pointer to the symbol table for the procedure appears on top of the stack pointer.

Reusing Temporary names: -

The newtemp generates the new temporary name each time a temporary need, these names are used to hold values. Temporary can be used by changing newtemp. e.g.

E->E1+E2
Evaluate E1 into t1
Evaluate E2 into t2
t:=t1+t2

The loop counter is temporary variable.

Addressing Array elements: -

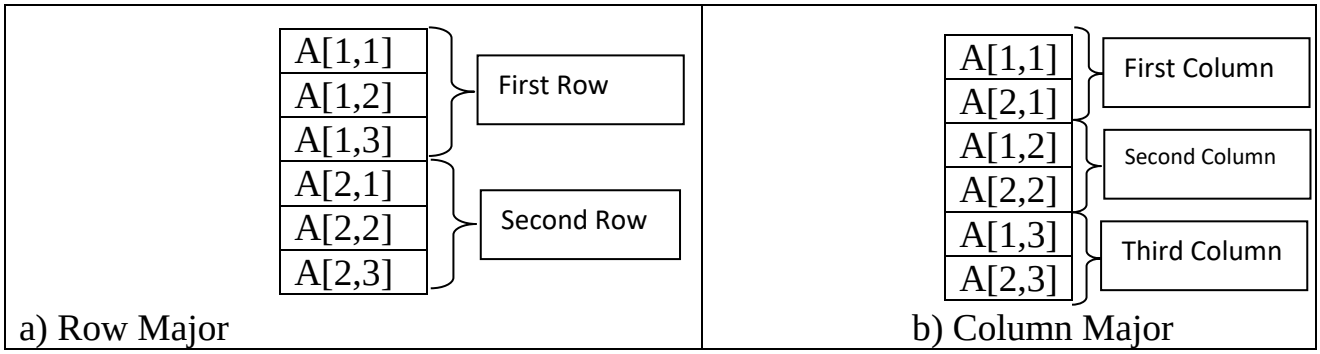
Array elements can be accessed quickly if they are stored in a block of a consecutive memory location. In C or Java language, array elements are numbered 0, 1,…..,n-1 for an array with n elements. If the width of each array element is ‘w’ then the i^{th} element of the array has a location.

base + i * w

Where the base is the relative address of the storage allocated for the array i.e. A[0]

In two dimensional array, we write A[i1][i2] in C or Java. For elements i2 in row i1. Let w1 be the width of the row and w2 is the width of elements in a row. The relative address A [i1][i2] can be calculated: base + i1 * w1+i2*w2.

In k dimension the formula is,
base + i1 * w1 + i2 * w2 + ------ +ik * wk
Layout for two dimensional array



Type conversion:-

Consider expression like x+i where x is of the type float and i is of type integer. The representation of floating-point and integer are different in computation. Different machine instructions are used for operations on integers and floats. The operator '+' needs the same type values hence compiler needs to correct the value of the operand in the same type.

Suppose the integer is converted to floats when necessary using a unary operator (float)

t1=float (i)
t2=x+t1

Type conversion rules vary from language to language. The rules for java distinguish between widening conversion which preserves information and narrowing conversion which can lose information.

e.g. :

int x = 10;
byte y = (byte)x;

In Java, typecasting is classified into two types,

- Widening Casting(Implicit)

- Narrowing Casting(Explicitly done)

Widening or Automatic type conversion:

- Automatic Typecasting takes place when,
- The two types are compatible

- The target type is larger than the source type

Narrowing or Explicit type conversion:
When you are assigning a larger type value to a variable of a smaller type, then you need to perform explicit typecasting.

Accessing fields in records: -

The compiler must keep track of both the types and relative addresses of the fields of a record. A separate symbol table is created for a record type.

P ↑into +1

Pointer (record (t)), t can be extracted where t is the record type.

Boolean expression:-

Boolean expression is a collection of Boolean operators (&&, ||, and !) applied to elements that are Boolean variables or relational expressions. Relational expression of the form E1 rel E2 where E1 and E2 are operands,

e.g.

B-> B||B| B&&B | !B |(B) | E rel E | true | false

We use the attribute rel op to indicate which of the six comparisons <, >, =, <=, >=,! = is represented by rel. || has the lowest precedence then && then!

Give information of ||, && and !

Short-circuit code:

Translate the Boolean Expression into 3 Address Code without generating code for any of the Boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called Short Circuit or Jumping Code.

e.g.

if (x<100 || x>200 && x!=y) x=0;

If the expression is true then control reaches label L2 otherwise reaches to label L2.

if (x<100) goto L2
if false x>200 goto L1
if false x!=y goto L1
L2: x=0
L1:

Jumping code

Flow of control statement: -

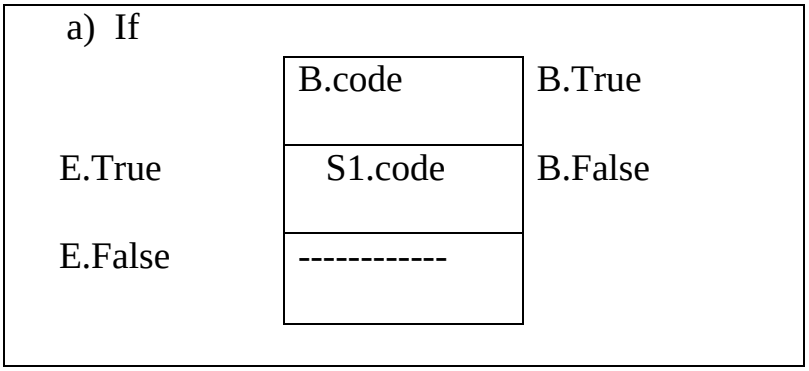
Now consider the translation of Boolean expression into three-address code the control statements are if-then-else and while. The grammar for such a statement is as,

S-> if (B) then S1
S-> if (B) then S1 else S2
S-> while (B) do S1

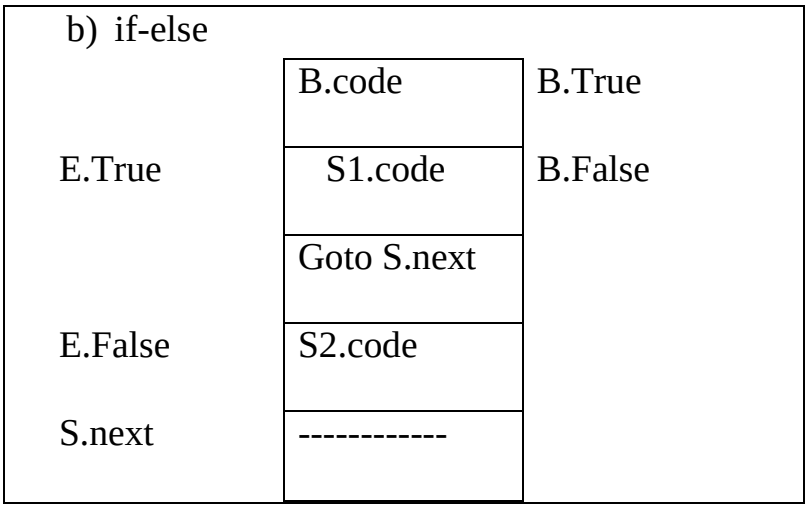
while generating a three-address code.

1) To generate a new symbol table the function new label is used.
2) The expression B.true and B.false are the labels associated.
3) S.code and B.code are for generating three-address codes.

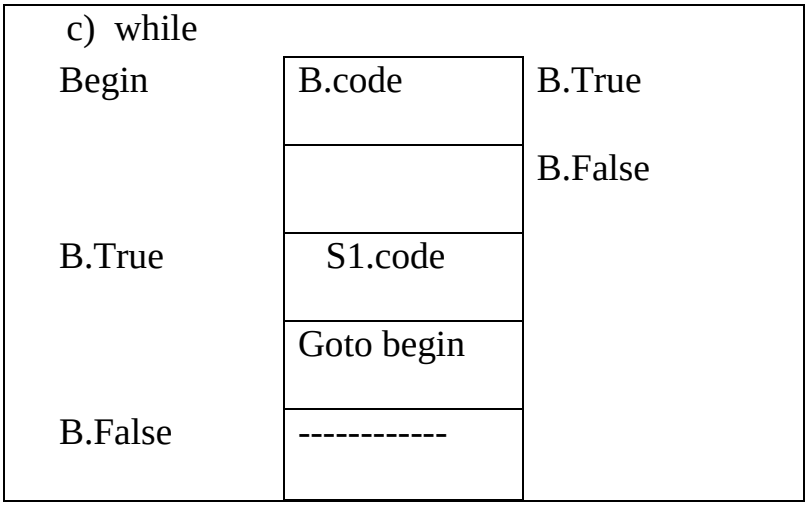
a) If



b) if-else



c) while



Case statement: -

The switch or case statement is evaluated in a variety of languages.

Syntax: -

```
switch expression
    {
      case value : statement
      case value : statement
      default : statement
```

}

e.g. Generation of three address codes.

Production rule	**Semantic action**

switch E
{
case v1 : s1
case v2 : s2
 	
case v1-1 : s1-1
default : sn
}
	Evaluate E into t such that t=E goto check
	t1:code for s1
	goto last
	t2:code for s2
	goto last
	Ln-1 : code for sn-1 goto last
	Ln : code for sn
	goto last
	check: if t=v1 goto L1
	if t=v2 goto v2
	
	if t=vn-1 goto Ln-1
	goto Ln
	last:

Backpatching:-

A key problem when generating code for Boolean expression flow of control statement is that of matching the same instruction with the target of a jump.
e.g. if (B) then S contains a jump when B is false the instruction following the code s.

In one pass compiler, B must be translated before s is examined. In one pass compiler, a separate pass is needed to combine labels to address.

In the backpatching list of jumps is passed as-synthesized attributes. When a jump has been specified the target of the jump is temporarily left unspecified. Each jump is put in the list of jumps whose labels are determined. All of the jumps on a list have the same target label.

To overcome the problem of processing incomplete information in one pass the backpatching technique is used. Backpatching is the activity of filling up unspecified information of labels using appropriate semantic action in during the code generation process.

We generate instruction into an instruction array and labels will be represented in this array. To manipulate the list of jumps we use three functions.

1) makelist (i):-

Create a newlist containing only i on index into an array of instruction makelist return the pointer to the newly created list.

2) merge (p1,p2):-

Concatenates the list pointed by p1 and p2 returns a pointer to the concatenated list.

3) backpatch (p, i):-

Inserts i is the target label for each of the instructions on the list pointed by P.

e.g.B->B1|mB2{backpatch(B1.falselist,m.instr

B.truelist=merge(B1.truelist,B2.truelist);

B1.falselist=B2.truelist);B1.flaselist=B2.falselist}

It can be used to translate the flow of control statement in one pass.

Procedure call:-

Procedure or function is used to avoid the repetition of code. Consider the following code for the procedure call.

S-> call id (L)

L->L,E

L->E

Here the non-terminal S denotes the statement and non-terminal L denotes the list of parameters and E-denotes the expression.

The data structure queue is used to hold the various parameters of the procedure. The keyword param is used to denote the list of the parameters passed to the procedure. The call to the procedure is given by call id, where id denotes the name of the procedure, E.place gives the name value of the parameter inserted in the queue.

For L-> E, the queue gets empty and a single pointer to the symbol table is obtained. This pointer denotes the value of E.

Questions:

1) Why there is a need for intermediate code?
2) What are the types of intermediate representation?
3) What are the three address codes? Explain three address codes for example.
4) Explain the types of three address codes in detail.
5) Explain the implementation of three address codes with examples.
6) Write a triple for:
   a) X[i]=y b) x=y[i]
7) Explain Boolean expression with an example.
8) Explain the flow of the control statement with an example.
9) What is backpatching? Explain with an example.
10) Explain the procedure call with an example.
11) Give the names of backpatching functions.