Unit - III

Syntax - Directed Definition (SDD)

→ While doing the static analysis of the language we use syntax-directed definitions.

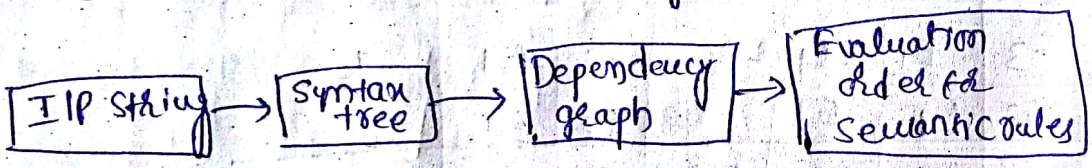
→ That means an augmented CFG is generated.

→ In other words the set of attributes are associated with each terminal & non-terminal symbols.

→ The attribute can be a string, a number, a type, a memory location or any thing else.

→ The syntax-directed definition is a kind of abstract specification.

The conceptual view of syntax-directed translation can be as shown in Fig



→ Firstly we parse the i/p token stream and a syntax tree is generated. Then the tree is being traversed for evaluating the semantic rules.

→ The implementation need not have to follow all the steps.

→ In single pass implementation semantic rules can be evaluated during parsing without explicity constructing a parsing tree, & dependency graph.

→ In such semantic evaluation, at the nodes of the syntax tree, values of the attributes are defined for the given i/p string.

Scanned by CamScanner

Such a parse tree containing the values of attributes at each node is called an "annotated" or "decorated the parsetree"

Let us see the form of syntax-directed definition

Definition: Syntax-directed definition is a generalization of CFG in which each grammar production $X \to \alpha$ is associated with it a set of semantic rules of the form $a := f(b_1, b_2 .. b_k)$ where a is an attribute obtained from the function f.

Attribute: the attribute can be a string, a number, a type, memory location or anything else.

Scanned by CamScanner

Syntax directed translation

Grammar + Semantic values = SDT

SDT for evaluation of expression

Ex:)

$E \to E + T$ {E. value = E.value + T value}

$\quad | \; T$ { E. value = T. value }

$T \to T * F$ { T. value = T.value * F. value }

$\quad | \; F$ { T.value = F. value }

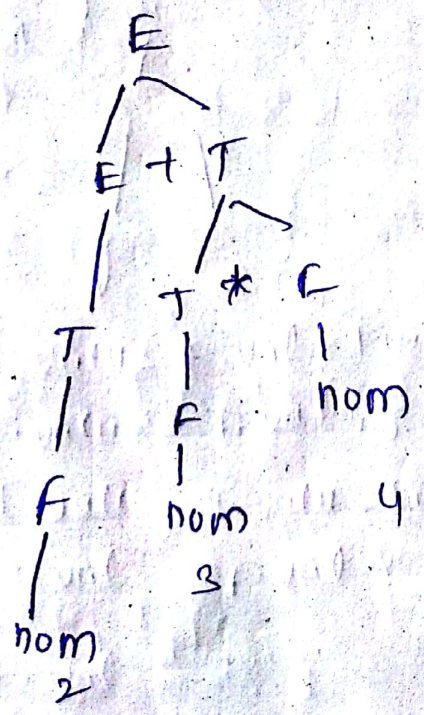
$F \to num$ { F. val = num. value }

Traverse top down left to right

$2 + 3 * 4$

num = token.

2, 3, 4 are lexical value



Scanned by CamScanner

2) $E \rightarrow E+T$ { printf("+"); } — ①

$\mid T$ { } — ②

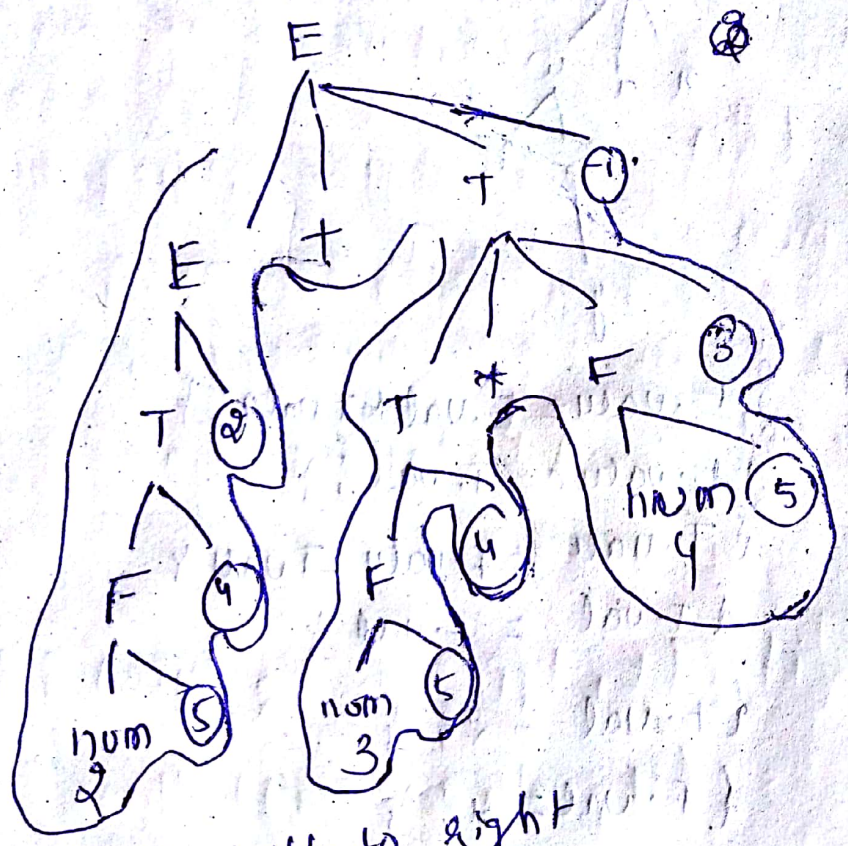
$T \rightarrow T*F$ { printf("*"); } — ③

$\mid F$ { } — ④

$F \rightarrow num$ { printf ~~("num")~~ (num.Lexval) } — ⑤

Syntaxt' caction on number =

2+3*4

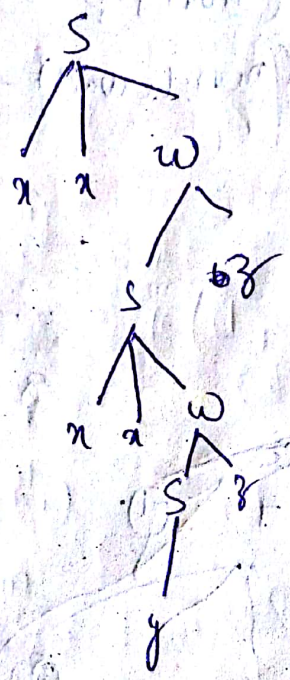


Top down Left to right

2 3 4 * +

$S \to xxw$ {printf("1");}
$\quad | \; y$ {printf("2");}
$w \to Sz$ {printf("3");}

String $xxxyzz$



O/P
23131

(4) $E \to E*T$ {E.val = E.val * T.val;}
$\quad | \; T$ {E.value = T.val;}
$T \to F-T$ {T.val = F.val - T.val}
$\quad | \; F$ {T.val = F.val}
$F \to 2$ {F.val = 2;}
$\quad | \; 4$ {F.val = 4;}

I/P: $4-2-4*2$

(5) $E \to E\#T$ {E.val = E.val * T.val;}
$\quad | \; T$ {E.val = T.val}
$T \to T\&F$ {T.val = T.val + F.val}
$\quad | \; F$ {T.val = F.val}
$F \to num$ {F.val = num.lexval}

$2\#3\&5\#6\&4 \to 2*(3+5)*(6+4)$
$2*8*10 = 60$

Scanned by CamScanner

Std to build Syntax tree

$E \to E_1 + T$ { $E.nptr = mknode(E_1.nptr, '+', T.nptr)$; }

$\quad | \; T$ { $E.nptr = T.nptr$ }

$T \to T_1 * F$ { $T.nptr = mknode(T_1.nptr, '*', F.nptr)$; }

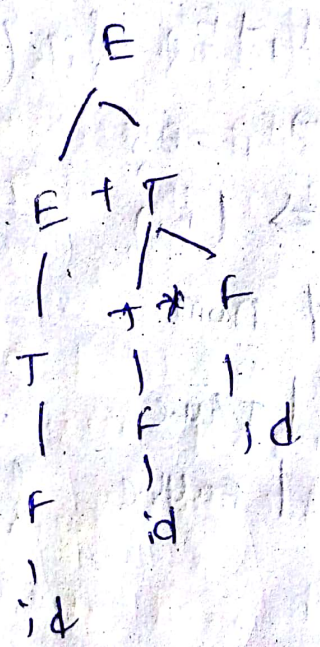
$\quad | \; F$ { $T.nptr = F.nptr$; }

$F \to id$ { $F.nptr = mknode(null, idname, null)$; }

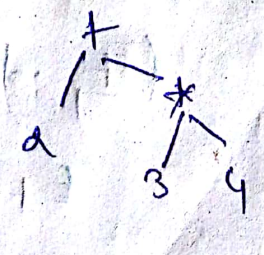
left pointer, value, right pointer

$2+3*y$

1) Concreate Syntax tree



2) Abstract Syntax tree



Address of node

100

10	+	10
null		null

→ assume nptr is 100

Address

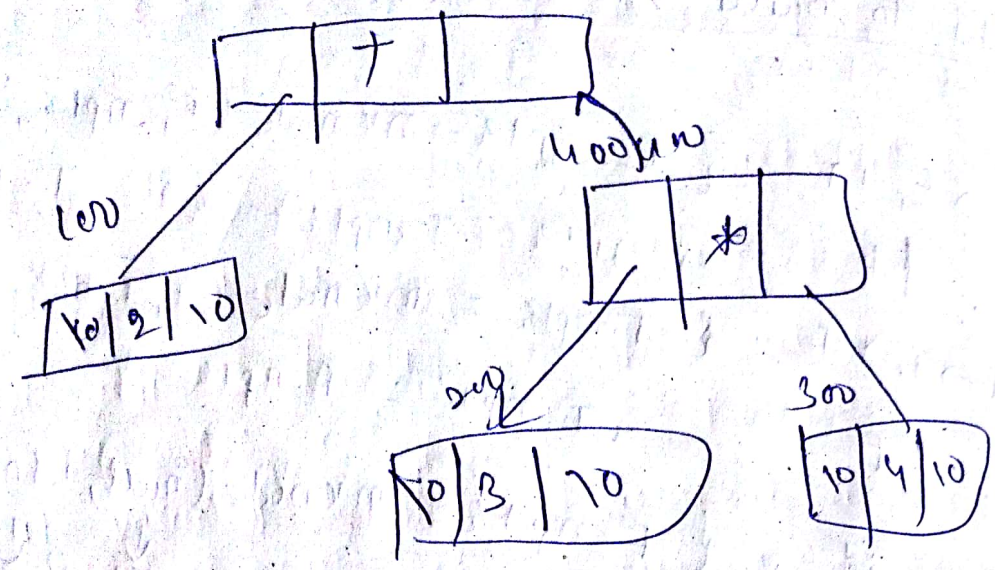
200

10	3	10
null		null

300

10	4	10

Scanned by CamScanner



Typechecking

2) $E \to E_1 + E_2$ → {if ((E_1.type == E_2.type) && (E_1.type ==

→ $E_1 == E_2$

⇒ (E_1) {E.type = E_1.type}

| num {E.type = int;}

| True {E.type = boolean;}

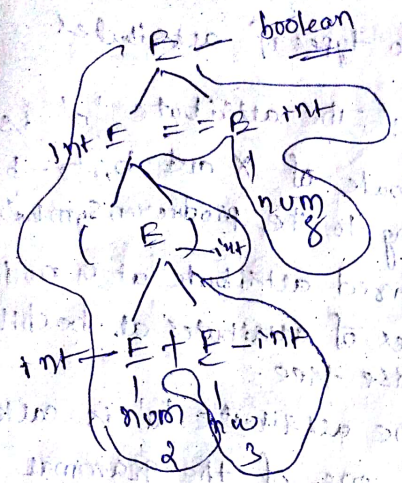
| False {E.type = boolean;}

⇒ 1) {if ((E_1.type == E_2.type) && (E_1.type = int)) then E.type = int else error;}

2) {if ((E_1.type == E_2.type) && (E_1.type = int|boolean)) then E.type = boolean else error.

Scanned by CamScanner

$(2+3) == 8$



Binary numbers

(3) $N \to L$ { count 1's
{ N.count = L.count }

$L \to LB$ { L.count = L.count + B.count }

$| B$ { L.count = B.count }

$B \to 0$ { B.count = 0 }

$| 1$ { B.count = 1 }

11 = 3
1011 = 5

N – number (can be a list)
L – List
LB → List Bits
B → Bit

1011

count 0's

$N \to L$
$L \to LB$
$| B$

$B \to 0$ { B.count = 1 }
$| 1$ { B.count = 0 }

Consider $X \to \alpha$ be a CFG and $a := f(b_1, b_2 .. b_k)$ where a is the attribute.

Then there are two types of attributes

1. Synthesized attribute: The attribute 'a' is called synthesized attribute of X and $b_1, b_2 .. b_k$ are attributes belonging to the production symbols.

   The value of synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.

2. Inherited attribute: The attribute 'a' is called inherited attribute of one of the grammar symbol on the right side of the production (i.e α) and $b_1, b_2 .. b_k$ are belonging to either X & α

   The inherited attributes can be computed from the values of attributes at the siblings and parent of that node.

1. Synthesized attribute

Ex:

$S \to EN$	print(E.val)
$E \to E+T$	$E.val = E_1.val + T.val$
$E \to E-T$	$E.val = E_1.val - T.val$
$E \to T$	$E.val = T.val$
$T \to T*F$	$T.val = T_1.val * F.val$
$T \to T/F \mid F$	
$F \to (E)$	
$F \to digit$	
$N \to ;$	

Scanned by CamScanner

2. Inherited attribute.

The value of inherited attribute at a node in a parse tree is defined using the attribute values at the parent or siblings.

S → TL
T → int
T → float
T → char
T → double
L → L, id
L → id

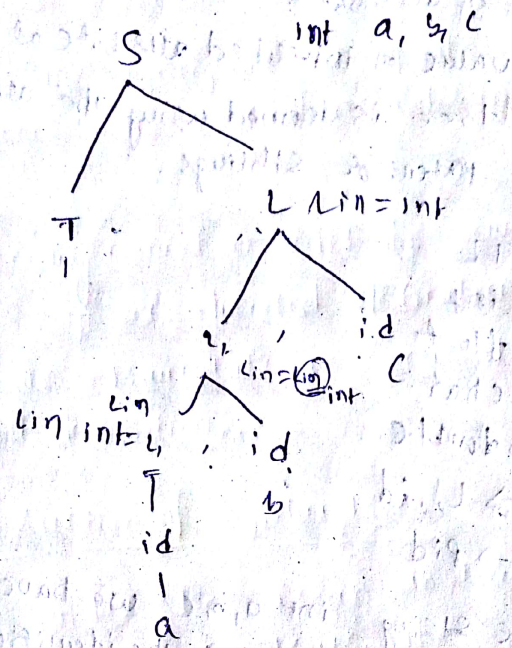
For the string int a, b, c we have to distribute the data-type int to all the identifiers a, b & c; such that a becomes integer, b becomes integer and c becomes integer.

Following steps are to be followed:

1. Construct the Syntax-directed definition using semantic action.
2. Annotate the parse tree with inherited attributes by processing in top-down fashion.

S → TL	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
T → char	T.type = char
T → double	T.type = double
L → L, id	L1.in := L.in Enter_type (id.entry, L.in)
L → id	Enter_type (id.entry, L.in)

Scanned by CamScanner



No. of bits

B → 0 {B.count = 1}
| 1 {B.count = 1}

|1| = 3
|0| = 5

1×2
2×

binary to decimal

N → L {N.dval = L.dval}
L → LB {L.decimalval = L.dval*2 + B.dval}
| B {L.dval = B.dval}
B → 0 {B.dval = 0}
| 1 {B.dval = 1}

Ex: 1 0 1 1

1×2

2*2+0

1×2 + 1 = 3

Dependency graph:

The directed graph that represents the interdependencies b/w synthesized & inherited attributes at nodes in the parse tree is called dependency graph.

Algorithm for constructing dependency graph

```
for (each node n in the parse tree) do
{
    for (each attribute 'a' of grammar at node n) do
    {
        for ~~(a~~ the attribute a, build a node
        of dependency graph.
        //constructing all nodes of graph.
    }
}
for (each node n in the parse tree) do
{
    for (each semantic rule b := f(c1, c2 .. ck)
    which is associated with productions) do
    for (i=1 to k) do
        construct edge from node ci to b
}
```

Scanned by CamScanner

4. Evaluation order.

→ The topological sort of the dependency graph decides the Evaluation order in a parse tree.

→ In deciding evaluation order the semantic rules in the syntax-directed definitions are used.

→ Thus, the translation is specified by "Syntax-directed definitions."

5. Construction of Syntax trees

The syntax tree is an abstract representation of the language constructs. The syntax trees are used to write the translation routines using syntax-directed definitions.

Directed Acyclic graph for Expressions (DAG)

→ The ~~directed ac~~ DAG is a directed graph drawn by identifying the common subexpressions.

→ Like syntax tree DAG has nodes representing the subexpressions in the expressions.

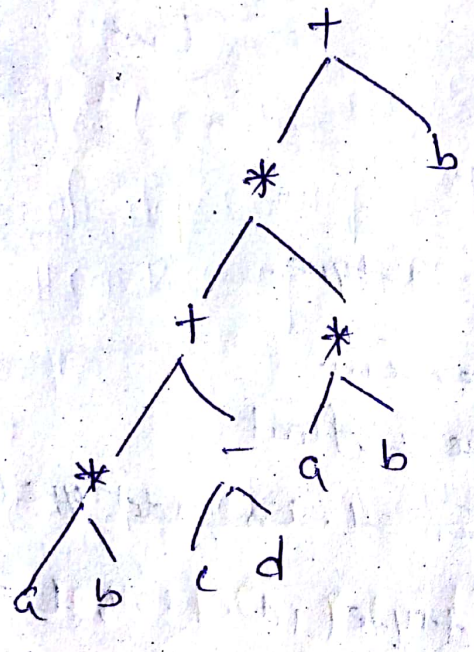
→ These nodes have operator, operand1 & operand2 where operands are childrens of that node.

Scanned by CamScanner

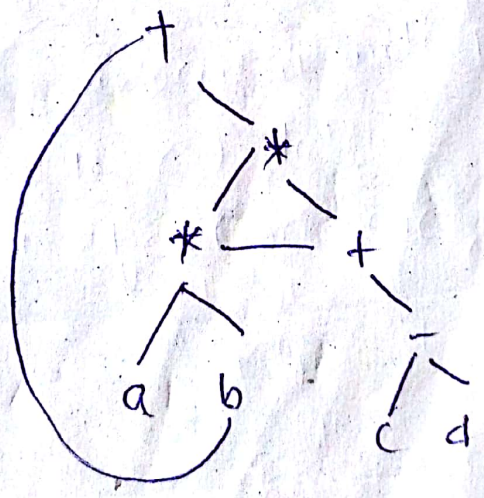
The difference b/w DAG & Syntax tree is that common subexpressions has more than one parent and in syntax tree the common subexpression would be represented as duplicated subtree.

Ex: Draw the syntax tree & DAG for

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

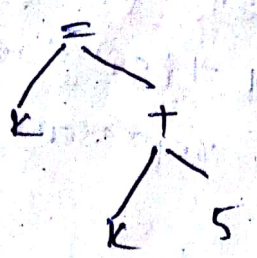


DAG is

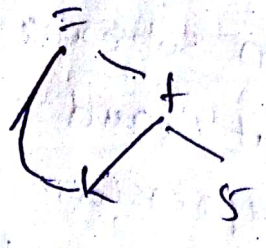


Scanned by CamScanner

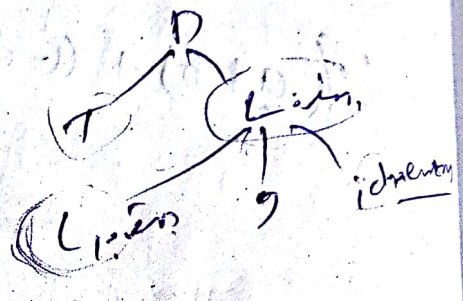
(1) DAG for k = k + 5



Syntax tree



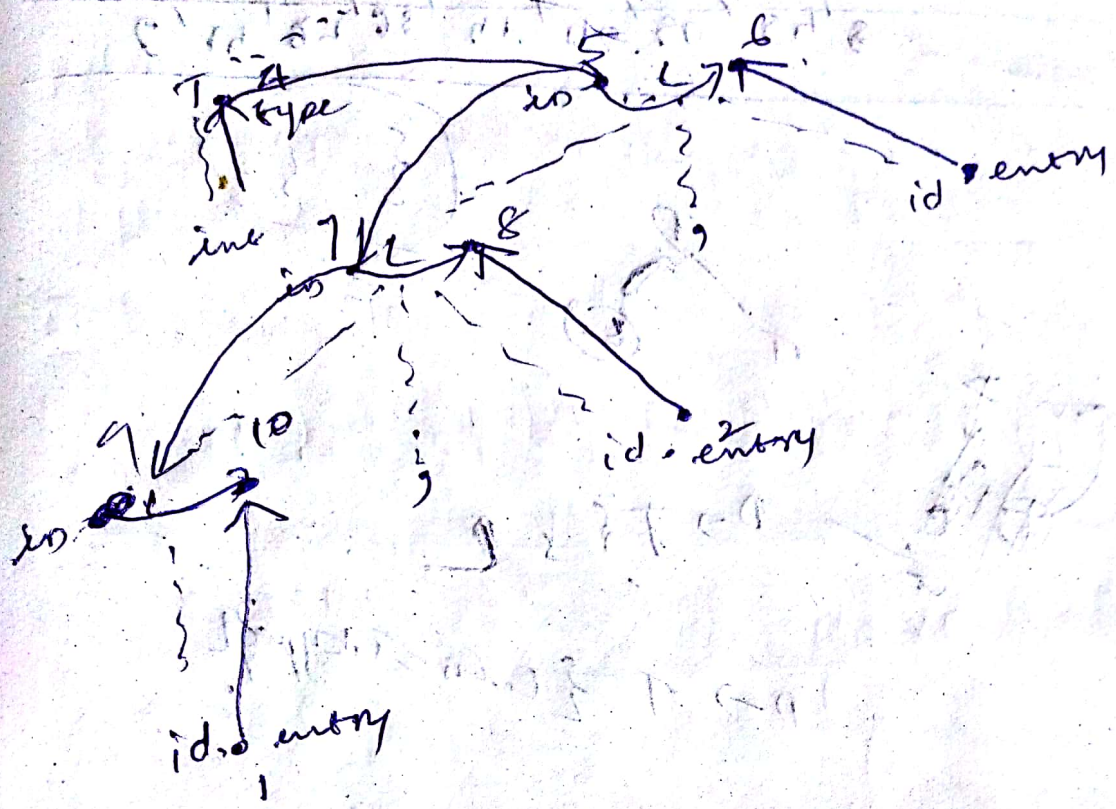
(DAG)



$D \to TL$	L.in = T.type
$T \to$ int	T.type = int
$T \to$ float	T.type = float
$L \to L_1, id$	L_1.in = L.in addType(id.entry, L.in)
$L \to id$	addType(id.entry, L.in)

Feb '23 '09

int a, b, c.



1. Get id_1.entry
2. Get id_2 entry
3. Get id_3. entry
4. T.type = int
5. L.in = T.type.
6. addType(id.entry, L.in)
7. L_1.in = L.in
8. addType(id.entry, L.in)
9. L_1.in = L.in
10. addType(id.entry, L.in).

Scanned by CamScanner

Translation Scheme:

→ As we have seen that the attributes are specified for each grammar symbol of the given production.

→ These attributes are used to evaluate the expression along the process of parsing. And exactly at the same time semantic rules are applied.

→ • During the process of parsing the evaluation of attribute take place by consulting the semantic action enclosed in { } at the right of the grammar symbol.

• This process of execution of code fragment semantic actions from the 'Syntax-directed definition' is called "Syntax-directed translation".

→ Thus the execution of ~~of~~ syntax-directed definition can be done by "Syntax-directed translation scheme".

→ A translation scheme generates the o/p by executing executing the semantic actions in an "ordered manner".

→ This processing is using "depth first traversal".

Scanned by CamScanner

Scheme.

While designing the translation scheme.

- We have to follow one restriction and that is for every semantic action if it refers to some attribute then that attribute value must be computed before that attribute gets referred.
- computation of synthesized attribute is very easy. that

ex: Production Rule — Semantic Action

$E \rightarrow E_1 * T$ — $\{E.val = E_1.val * T.val\}$

- Here val is synthesized attribute.
- For the computation of both the inherited and synthesized attributes.

i) In any semantic action the computation of inherited attribute for the symbol on the right side of the production must be done before that symbol.

That means if there is a production $A \rightarrow XY$ then computation of X.in must be done already.

ii) In the semantic action there should not be any reference to the synthesized attribute of the symbol that is right.

Scanned by CamScanner

That means for the production $A \to B_1 B_2$ the semantic action $\{B_1.s = B_2.s\}$ is invalid as computation of attribute for B_1 is based on computation of the attribute of B_2.

iii) For a computation of synthesized attribute for a non-terminal on the left, first do all the computations of the attributes that refers this non terminal.

Typically the semantic action containing the computation of all ~~sh~~ such attributes is placed at the end of right side of production.

Recursive Evaluation:

→ Recursive evaluation is possible by creating separate functions for each corresponding nonterminal.

→ From the syntax directed definition, the function body for each non-terminal symbol is created.

→ These functions actually help in implementation of syntax directed definition.

Scanned by CamScanner

Ex: Implement the Syntax-directed definition for the set of production rules which is as given below.

$A \rightarrow PQ$

$A \rightarrow XY$

Sol: 1) we will write the semantic rules for the given productions using inherited and synthesized attributes

2) The relationship among synthesized and inherited attributes of each non-terminal is graphically shown.

3) The implementation code is written.

Step 1:

Production rules	Semantic rules
$A \rightarrow PQ$	$P.in := p(A.in)$ $Q.in := q(P.sy)$ $A.sy := f(Q.sy)$
$A \rightarrow XY$	$Y.in = y(A.in)$ $X.in = x(Y.sy)$ $A.sy = f(X.sy)$

Scanned by CamScanner

Step 2:

.A.in
.A.sy

.P.in
.P.sy

Q.in
Q.sy

S - attributed definition

$D \rightarrow D_1, id$ { add_type(id.name, D_1.type)}

| T id {add_type (id.name, T.type), D.type = T.type;}

$T \rightarrow$ int {T.type = int;}

| char { T.type = char;}

SDT to generate three address code

I/P: x = a+b*c

$S \rightarrow id = E$ { gen (id.name = E.place); }

$E \rightarrow E_1 + T$ { E.place = new Temp(); gen(E.place = E_1.place + T.place); }

| T { E.place = T.place }

$T \rightarrow T_1 * F$ { T.place = newtemp(); gen(T.place = T_1.place * F.place); }

| F { T.place = F.place; }

$F \rightarrow id$ { F.place = id.name; }

Scanned by CamScanner

Intermediate code:

→ The task of compiler is to convert the source program into machine program.

→ This activity can be done directly, but it is not always possible to generate such a machine code directly in one pass.

→ Then, typically compilers generates an easy to represent form of source language which is called 'intermediate language'.

→ The generation of an intermediate language leads to efficient code generation.

Benefits of intermediate code Generation

There are certain benefits of generating machine independent intermediate code.

1. A compiler for different machines can be created by attaching different backend to the existing front ends of each machine.
2. A compiler for different source languages (on the same machine) can be created by proving different front ends for corresponding source languages to existing back end.
3. A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

Scanned by CamScanner

Intermediate code:

→ The task of compiler is to convert the source program into machine program.

→ This activity can be done directly, but it is not always possible to generate such a machine code directly in one pass.

→ Then, typically compilers generates an easy to represent form of source language which is called 'intermediate language".

→ The generation of an intermediate language leads to efficient code generation.

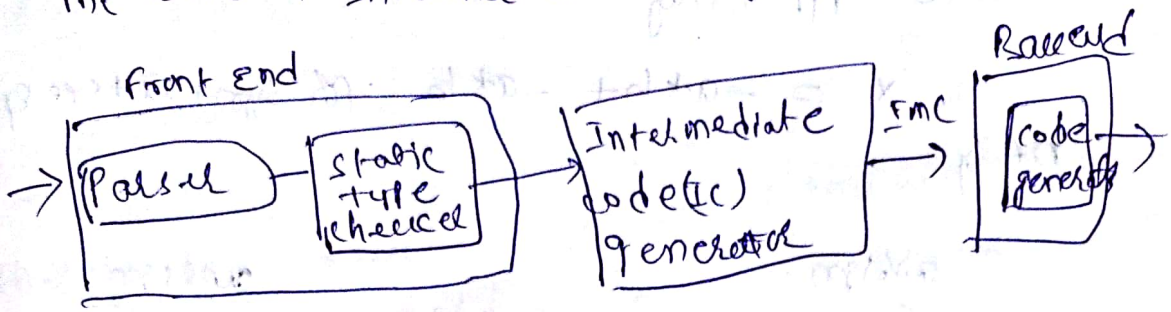
Benefits of intermediate code Generation

There are certain benefits of generating machine independent intermediate code.

1. A compiler for different machines can be created by attaching different backend to the existing front ends of each machine.
2. A compiler for different source languages (on the same machine) can be created by proving different front ends for corresponding source languages to existing back end.
3. A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

Scanned by CamScanner

The role of Intermediate code generator



Forms of Intermediate code

There are mainly 3 types of intermediate code representations:

1. Abstract Syntax tree
2. Polish notation
3. Three address code

These three representations are used to represent the intermediate languages.

Let us discuss with the help of Example

1. Abstract Syntax tree

→ The natural hierarchial structure is represented by Syntax trees.

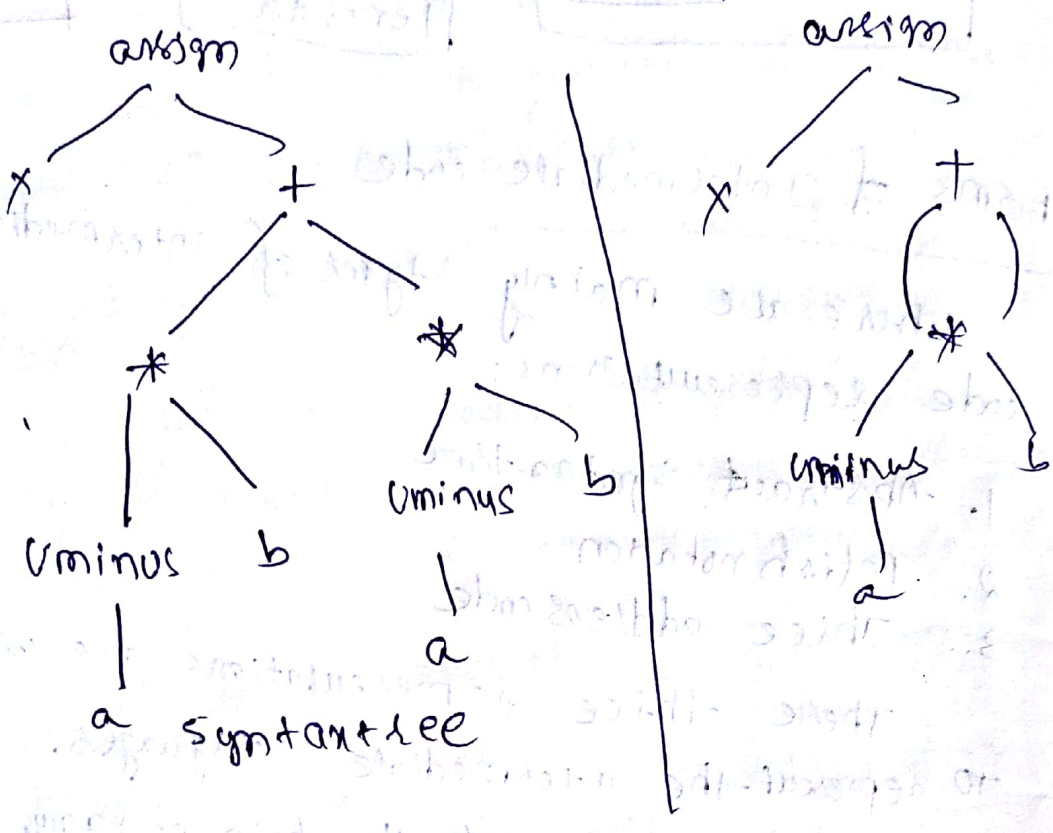
→ DAG is very much similar to Syntax tree but they are in more compact form.

The code being generated as intermediate should be such that the remaining processing of the subsequent phases should be easy.

Scanned by CamScanner

consider I/P string:

$x = -a*b + -a*b$ for syntax tree & DAG.



syntax tree

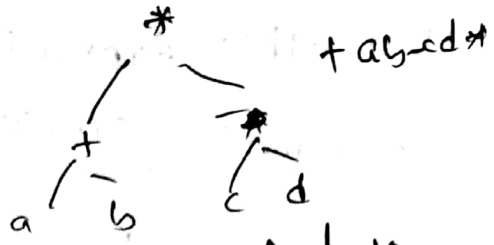
Polish notation

Basically, the linearization of syntax trees in polish notation. In this representation, the operator can be easily associated with the corresponding ~~with the corresponding~~ operands. This is the most natural way of representation in expression evaluation.

The polish notation is also called as prefix notation in which the operator occurs first & then operands are arranged

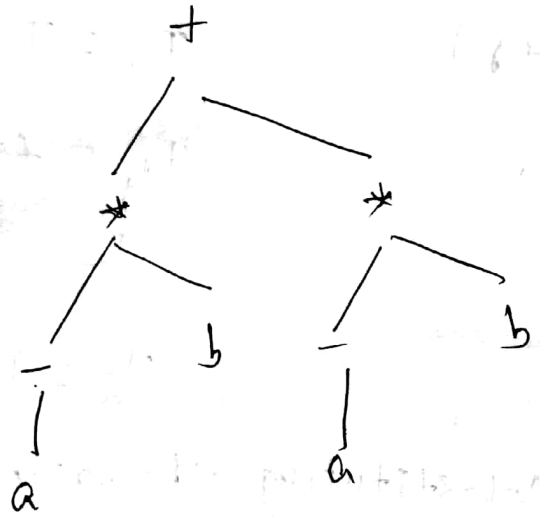
Ex: $(a+b) * (c-d)$

$+ab-cd*$



There is reverse polish notation which is used using post fix or posix representation

Ex: $-a*b+-a*b$

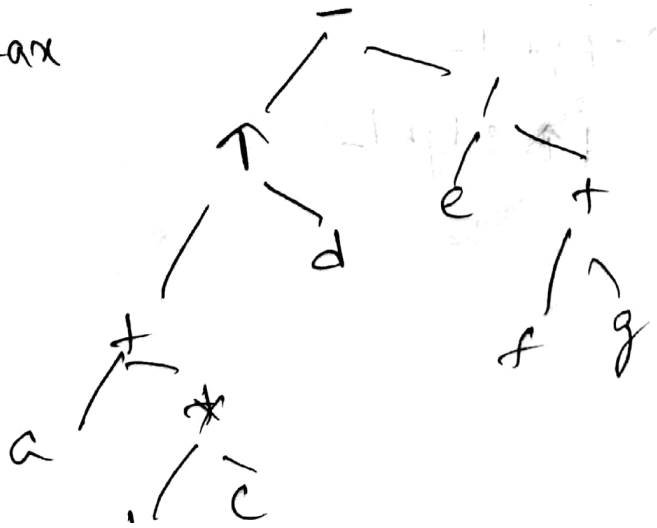


$a-b*a-b*+$

Ex: Construct syntax tree of post fix notation for the following expression

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

Syntax



Scanned by CamScanner

In this Expression the arrangement is operand1, operand2, operator

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

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

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

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

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

$T_3 - T_5$ $\quad T_5 = eT_4/$

T_6 $\quad T_6 = T_3T_5-$

Now backward substituting the values of temporary variables.

T_6

T_3T_5-

$T_eT_4/-$

$T_3efg+/-$

$T_2d\uparrow efg+/-$

$aT_1d\uparrow efg+/-$

$abc*+d\uparrow efg+/-$

Scanned by CamScanner

Three address code

In three address code ~~form~~ form at the most 3 addresses are used to representation is

$a := b \; op \; c$

a, b, c are operands that can be names, constants, compiler generated temporaries and ~~and~~ op represents the operator.

These operators can be fixed or floating point arithmetic operator or logical or ~~log~~ Boolean valued data.

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

3 address code will be

Ex: $a = b + c + d$

$t_1 = b + c$

$t_2 = t_1 + d$

$a = t_2$

t_1, t_2 are the temporary names generated by compiler. There are at the most 3 addresses allowed.

Hence the name of this representation is 3 - address code.

Scanned by CamScanner

Implementation of 3 address code

3 address code is an abstract form of Intermediate code that can be implemented as a record with the address fields.

There are 3 representation used.

those are

1) Quadruples
2) triples
3) indirect triples

Quadruple representation

The quadruple is a structure with at the most four fields such as op, arg1, arg2, result.

The op is used to represent the internal code for operator,

the arg1 & arg2 represent the two operands used and result field is used to store the result of an expression.

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

		op	Arg1	Arg2	result
t_1 = uminus a	(0)	uminus	a		t_1
$t_2 = t_1 * b$	(1)	*	t_1	b	t_2
$t_3 = -a$	(2)	uminus	a		t_3
$t_4 = t_3 * b$	(3)	*	t_3	b	t_4
$t_5 = t_2 + t_4$	(4)	+	t_2	t_4	t_5
$x = t_5$	(5)	=	t_5		x

Triples

Ind

Scanned by CamScanner

Triples

In the triple representation the use of temporary variables is avoided by referring the pointer in the symbol table

Number	op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

Indirect triples

In the indirect triple representation the listing of triples is been done. And listing pointer are used instead of using statements

Number	op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(11)	b
(2)	uminus	a	
(3)	*	(13)	b
(4)	+	(12)	(14)
(5)	=	x	(15)

Scanned by CamScanner

(0) (11)
(1) (12)
(2) (13)
(3) (14)
(4) (15)
(5) (16)

Merits & Demerits of Quadruple, Triple and Indirect Triples

1. In the qu. rep using temporary names the entries in the symbol table against those temporaries can be obtained
2. The advantage with qu. r is that one can quickly access the value of temporary variables using symbol table.
   use of temporaries introduces the level indirection for the use of symbol table in qua r. rep
3. where as, in triple rep the pointers are used. By using pointers one can access directly the symbol table entry
4. The qua. rep is beneficial for code optimisation.
5. In indirect triple list of all relevant to computations is made separately and stored.

Scanned by CamScanner

Ex 1) $a * -(b+c)$ construct triples of expr

$t_1 = a$
$t_2 = b$
$t_3 = t_2 + c$
$t_4 = \text{uminus } t_3$
$t_5 = t_1 * t_4$

2) Quad all

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

$t_1 = a*b$
$t_2 = \text{uminus } t_1$
$t_3 = c+d$
$t_4 = t_2 + t_3$
$t_5 = a+b$
$t_6 = t_5 + t_3$
$t_7 = t_4 - t_6$

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

Scanned by CamScanner

Types of 3 address stmts.

The form of 3 address code is very much similar to assembly language. Here are some commonly used 3 address code for typical language constructs.

Language construct	intermediate code form	meaning
Assignment stmt	x = y op z	Here binary operation is performed using operator 'op'
Assignment stmt	x = op y	Here the unary operation is perfo the operator 'op' is an unary operator
copy stmt	x = y	Here the value of is assigned to x.
unconditional jump	goto L	The control flow goes t the stmt labeled by L
conditional jump	if x relop y goto L	The relop indicates t relational operators such <, =, >= if x relop is true then it exe goto L stmt
Procedure calls	Param x_1 param x_2 ⋮ param x_n call p, n	Here the parameters $x_1, x_2 \ldots x_n$ are used as Parameters to the procedure P.
	return y	The return stmt indicates the return value y.

Scanned by CamScanner

Array stmt	$x = y[i]$	The value at i^{th} index of array y is assigned to x.
	$x[i] = y$	The value of identifier y is assigned at the index i of the array x.
Address and pointer assignments	$x := \&y$ $x := *y$	The value of x will be the address of (or) location of y. The y is a pointer whose value is assigned to x.
	$*x = y$	The r-value of object pointed by x is set by the l-value of y.

Declarative stmt

In the declarative stmts the data items along with their data types are declared.

Ex:

	{offset := 0}
$S \rightarrow D$	
$D \rightarrow id : T$	{ enter_tab (id.name, T.type, offset); offset = offset + T.width)}
$T \rightarrow$ integer	{ T.type = integer ; T.width = 4 }
$T \rightarrow$ real	{T.type = real ; T.width = 8 }
$T \rightarrow$ array [num] of T_1	{T.type = array(num.val, T_1.type) T.width = num.val × T_1.width}
$T \rightarrow *T_1$	{T.type = pointer(T_1.type) T.width = 4 }

Scanned by CamScanner

Assignment stmts

→ The assignment stmt mainly deals with the Expressions.

→ The Expressions may be of type integer, real, array and record.

Ex: $S \to id := E$

$E \to E_1 + E_2$

$E \to E_1 * E_2$

$E \to -E_1$

$E \to (E_1)$

$E \to id$

Sol:

Production Rule	Semantic actions
$S \to id := E$	{ id_entry = look_up (id.name); if id_enty ≠ nil then append (id_entry = E.place) else error; /*id not declared*/ }
$E \to E_1 + E_2$	{ E.place = newtemp ( ); append (E.place = E_1.place + E_2.place }
$E \to E_1 * E_2$	{ E.place = newtemp ( ); append (E.place = E_1.place * E_2.place) }
$E \to -E_1$	{ E.place = newtemp ( ); append (E.place = uminus E_1.place) }

Scanned by CamScanner

$E \rightarrow E_1$ $\{E.place = E_1.place\}$

$E \rightarrow id$

```
{ id_entry = look-up(id.name);
  if id_entry ≠ nil then
  append(id_entry = E.place)
  else Error; /* id not declared */
}
```

→ The look up returns the entry for id.name in the symbol table if it exists there.

→ The function append is for appending the 3-address code to the o/p file otherwise an error will be reported.

→ new temp() is function for generating new temporary variables.

→ E.place is used to hold the value of E.

Consider the assignment stmt

$$X = (a+b) * (c+d)$$

we will assume all the identifiers are of the same type. Let us have a bottom up parsing method.

Production Rule	Semantic action for Attribute evaluation	O/P
$E \rightarrow id$	$E.place = a$	
$E \rightarrow id$	$E.place = b$	
$E \rightarrow E_1 + E_2$	$E.place = t_1$	$t_1 = a+b$
$E \rightarrow id$	$E.place = c$	
$E \rightarrow id$	$E.place = d$	
	$E.place = t_2$	$t_2 = c+d$

Scanned by CamScanner

$E \to E_1$	$\{E.place = E_1.place\}$

$E \to id$	{ id_entry = look-up(id.name);
	if id_entry ≠ nil then
	append(id_entry = E.place)
	else error; /* id not declared */
	}

→ The look up returns the entry for id.name in the symbol table if it exists there.

→ The function append is for appending the 3-address code to the o/p file otherwise an error will be reported.

→ new temp() is function for generating new temporary variables.

→ E.place is used to hold the value of E.

consider the assignment stmt

$$X = (a+b) * (c+d)$$

we will assume all the identifiers are of the same type. Let us have a bottom up parsing method.

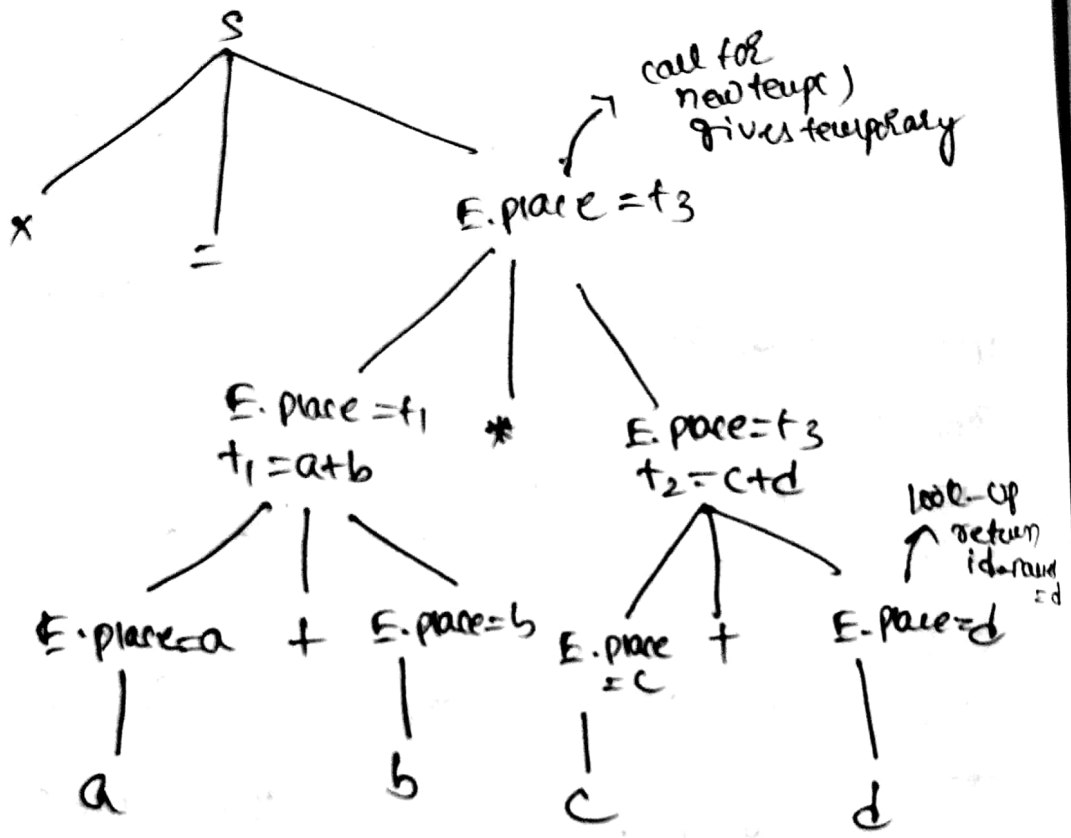
Production Rule	Semantic action for Attribute evaluation	o/p
$E \to id$	$E.place = a$	
$E \to id$	$E.place = b$	
$E \to E_1 + E_2$	$E.place = t_1$	$t_1 = a+b$
$E \to id$	$E.place = c$	
$E \to id$	$E.place = d$	
$E \to E_1 + E_2$	$E.place = t_2$	$t_2 = c+d$

Scanned by CamScanner

$E \to E_1 * E_2$ $\quad$ $E.place = t_3$ $\quad$ $t_3 = (a+b) * (c+d)$

$S \to id := E$ $\quad$ $x = t_3$

parse tree



Scanned by CamScanner

Back patching

Ex:1 if (a < b) then t=1 else t=0

i) if a<b goto (i+3)

i+1) t=0

i+2) goto (i+4)

i+3) t=1

i+4) ...

leaving the table as empty and filling then later is called Back patching

Ex2: $a \overset{t_1}{<} b$ and $c \overset{t_2}{<} d$ or $e \overset{t_3}{<} f$

100) if (a< b) goto 103

101) $t_1 = 0$

102) goto 104

103) t1 = 1

104) if (c < d) goto 107

105) $t_2 = 0$

106) goto 108

107) $t_2 = 1$

108) if (e < f) goto 111

109) $t_3 = 0$

110) goto 112

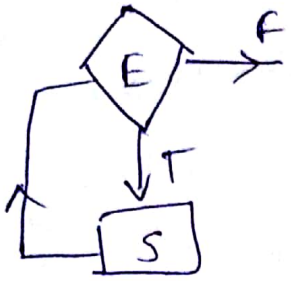
111) $t_3 = 1$

112) $t_4 = t_1$ and t_2

113) $t_5 = t_4$ or t_3

Scanned by CamScanner

* while E do S



L: if (E==0) goto L1
S
Goto L
L1:

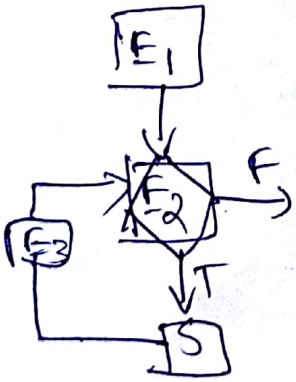
(or)

L: if (E) goto L1
goto Last
L1: S
goto L
Last: ___

Ex: while (a<b) do
x = y+z.

L: if a<b goto L1
goto Last
L1: t = y+z
x = t
goto L
Last: ___

* for (E1 ; E2 ; E3)
E1 → initialization
E2 → condition
E3 → incr/decre.
S



Scanned by CamScanner

Ex: for (i=0 ; i<10 ; i++)

a=b+c

Sol:

```
i=0
L: if (i<10) goto L1
   goto Last

L1: t1 = b+c
    a = t1
    t = i+1
    i = t
    goto L
Last:
```

* switch (i+j)

{

case(1) : a = b+c ;
break ;

case (ii) : p = q+r ;
break ;

~~case~~ default : x = y+z ;
break ;

}

Sol:

```
t = i+j
goto test

L1: t1 = b+c
    a = t1
    goto last

L2: t2 = q+r
    p = t2
    goto last

L3: t3 = y+z
    x = t3
    goto last

test: if (t==1) goto L1
      if (t==2) goto L2
      goto L3
Last:
```

Scanned by CamScanner

Boolean Expressions

Normally there are two types of Boolean Expressions used:

1. For computing the logical values.
2. In conditional Expressions using if-then-else & while-do

Consider the Boolean Expression generated by following grammar.

E → E OR E

E → E AND E

E → NOT E

E → (E)

E → id relop id

E → TRUE

E → FALSE

Here the relop is denoted by <=, >=, ≠, <, >

the OR and AND are left associative.

highest precedence is to NOT then AND and lastly OR.

Scanned by CamScanner

Numerical Representation

Production Rule	Semantic Rule
$E \rightarrow E_1$ OR E_2	{ E.place = newtemp() append (E.place = E_1.place OR E_2.place) }
$E \rightarrow E_1$ AND E_2	{ E.place = newtemp() append (E.place = E_1.place 'AND' E_2.place) }
$E \rightarrow$ NOT E_1	{ E.place = newtemp() append (E.place = 'NOT' E_1.place) }

$E \rightarrow$ id, relop id2	{ E.place = newtemp()
	 append ('if' id1.place relop id2.place 'goto' next_state + 3);
	 append (E.place = '0');
	 append ('goto' next_state + 2);
	 append (E.place = '1')
	 }

$E \rightarrow$ TRUE	{ E.place = newtemp(); append (E.place = '1') }
$E \rightarrow$ FALSE	{ E.place = newtemp(); append (E.place = '0') }

Scanned by CamScanner

The function append generates the 3 adress code and new temp ( ) is for generation of temporary variables. For the semantic action for the rule.

E → id1 relop id2 contains next-state which gives the index of next 3 address stmt in the o/p sequence.

Let us take an example and generate the 3-address code using above translation scheme.

P > q AND r < s OR U > V

100 : if P > q goto 103

101 : $t_1 = 0$

102 : goto 104

103 : $t_1 = 1$

104 : if r < s goto 107

105 : $t_2 = 0$

106 : goto 108

107 : $t_2 = 1$

108 : if U > V goto 111

109 : $t_3 = 0$

110 : goto 112

111 : $t_3 = 1$

112 : $t_4 = t_1$ AND t_2

113 : $t_5 = t_4$ OR t_3

→ In this three adress code we have used goto to jump on some specific stmts.

→ This method of evaluation is called "short-circuit". The AND has higher precedence over OR hence at location 112 then ANDing is performed and then in the immediate next stmt. ORing is performed.

Scanned by CamScanner

Flow of control stmts

The control statements are if-then-else and while-do.

The grammar for such statements is as shown below.

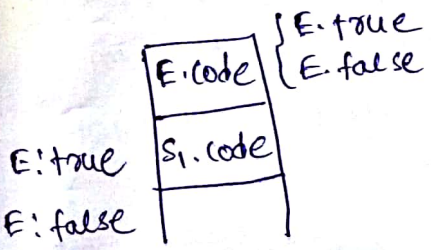
$S \rightarrow$ if E then S_1

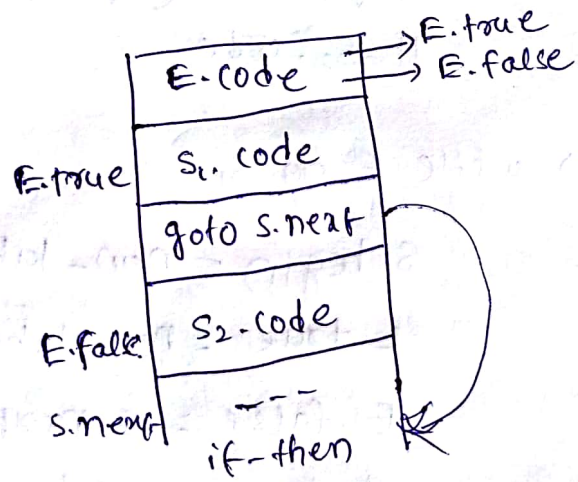
| if E then S_1 else S_2

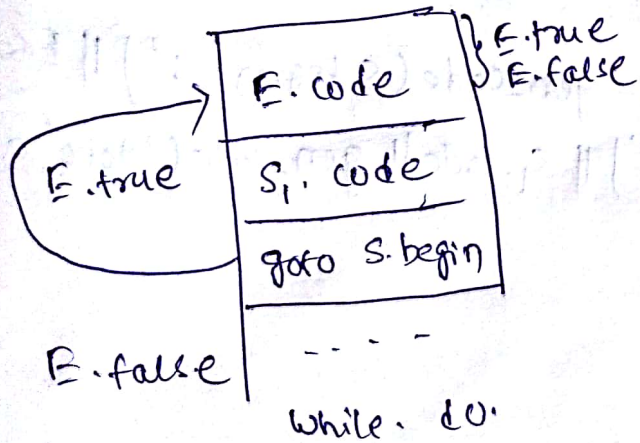
| while E do S_1

while generating 3 address code.

- To generate new symbolic label the function new_label() is used
- with the expression E.true & E.false are the tables associated.
- S.code and E.code is for generating 3 address code.







Scanned by CamScanner

$S \rightarrow$ if E then S_1

E.true = new_label()

E.false = S.next

S_1.next = S.next

S.code = E.code || gen_code(E.true ':') || S_1.code

In the above translation scheme || is used to concatenate the strings. The function gen_code is used to evaluate the non quoted arguments passed to it and to concatenate complete string.

$S \rightarrow$ if E then S_1 else S_2

E.true = new_label()

E.false = new_label()

S_1.next = S.next

S_2.next = S.next

S.code = E.code || gen_code(E.true ':') || S_1.code || gen_code('goto', S.next) || gen_code(E.false ':') || S_2.code.

$S \rightarrow$ while E do S_1

S.begin = new_label()

E.true = new_label()

E.false = S.next

S_1.next = S.begin

S.code = gen_code(S.begin ':') || E.code || gen_code(E.true ':') || S_1.code || gen_code('goto', S.begin)

Scanned by CamScanner

Ex: Generate 3 address code for

```
while (i<10)
{
  x=0;
  i=i+1;
}
```

Sol:

```
100: L1: if i<10 goto L2
101: goto Lnext
102: L2: ~~i=i+1~~, x=0
103: i=i+1
104: goto L1
105: Lnext:
```

Here we use translation scheme for while do

Ex: if [(a<b) and ((c>d) or (a>d))] then

z = x + y * z

else z = z+1

Sol: Three address code

```
100: if a<b goto 102
101: goto 110
102: if c<d goto 106
103: goto 104
104: if a>d goto 106
105: goto 110
106: t1 = x+y
107: t2 = t1 * z
108: z = t2
109: goto 112
110: t3 = z+1
111: z = t3
112: stop
```

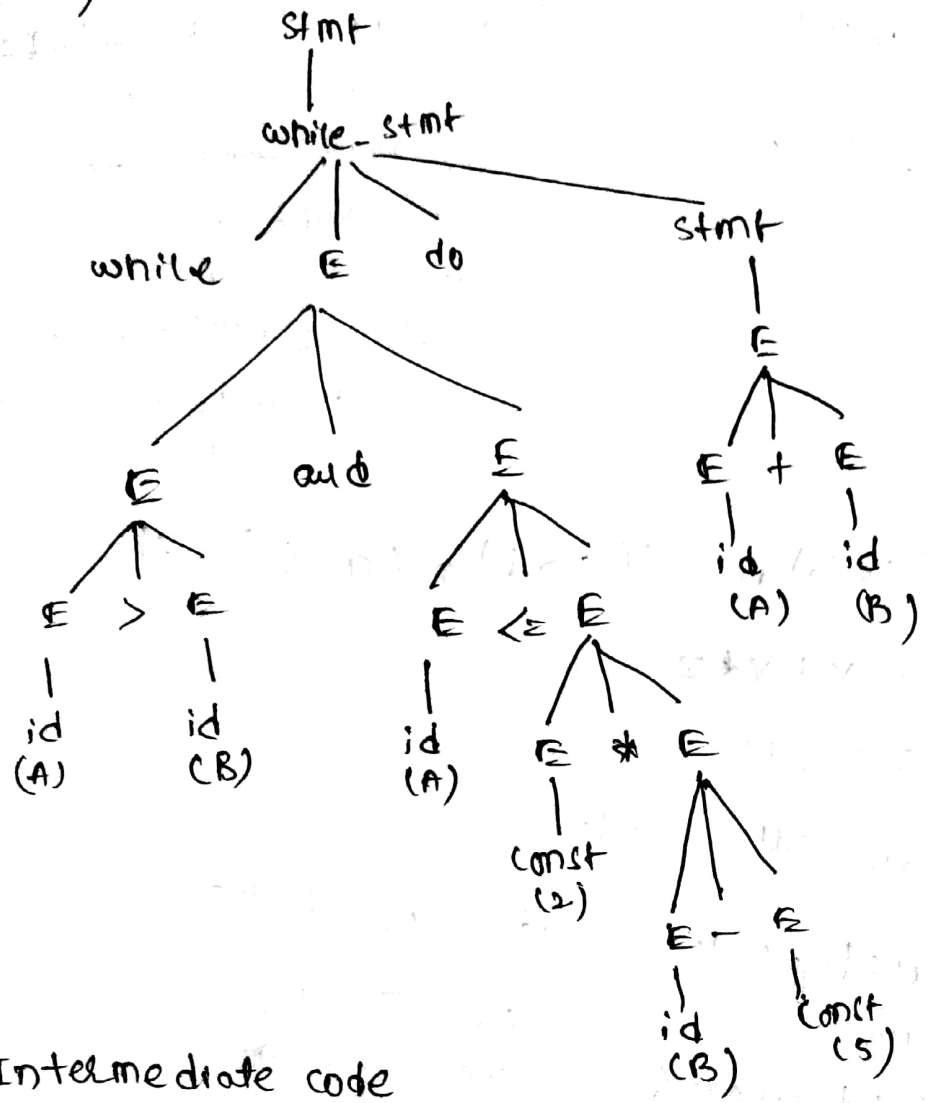
Ex: consider the following while-stmt - while A>B and A<= 2*B-5 do A: A+B;

i) construct the parse tree for the above given while stmt

ii) write the intermediate code for while-stmt.

Scanned by CamScanner

Sol: i) Parse tree



ii) Intermediate code

L1: if A>B goto L2

goto Lend

L2: t_1 = 2*B

t_2 = t_1 - 5

if A <= t_2

goto t_4

L4: A = A+B

goto t_1

Lend: stop.

Scanned by CamScanner

Ex:3 - address code

while (A<C and B>D) do

if A=1 then C=C+1 else while A<=D do

A = A+B

Sol:

100 : if A<C then goto 102

101 : goto 115

102 : if B>D then goto 104

103 : goto 115

104 : if A=1 then goto 106

105 : goto 109

106 : t_1 = C+1

107 : C = t_1

108 : goto 100

109 : if A<=D then goto 111

110 : goto 100

111 : t_2 = A+B

112 : A = t_2

113 : goto 109

114 : goto 100

115 : stop

Ex.

```
main()
{
int i;
int a[10];
i=1
while ( i<=10)
    {a[i]=0,
        i=i+1;
    }
}
```

Scanned by CamScanner

Case stmts

The Syntax for switch - case stmt can be as shown below :

```
switch expression
  {
    case value : stmt
                 break ;
    case value : stmt ; break
      :
    default : stmt
  }
```

Production Rule

```
switch E
 {
 case v1: S1
 case v2: S2
   :
 case vn-1 : Sn-1
 default : Sn
```

Semantic Action

```
Evaluate E into t such
 that t=E
 goto check
L1 : code for S1
     goto last
L2 : code for S2
     goto last
  :
Ln : code for Sn
     goto last
check: if t=v1 goto L1
       if t=v2 goto L2
       if :
       if t=vn-1 goto Ln-1
       goto Ln

last:
```

Scanned by CamScanner

Generate the 3 address code for

```
switch (ch)
{
case 1: c=a+b;
        break;
case 2: c = a-b;
        break;
}
```

Sol:

```
100:  if ch=1  goto L1
101:  if ch=2  goto L2
102:  L1: t1 = a+b
103:      c = t1
104:      goto Last
105:  L2: t1 = a-b
106:      c = t1
107:      goto last
108:  last:
```

Procedure calls

Procedure or function is an important programming construct which is used to obtain the modularity in the user program.

Consider the grammar for a simple procedure call

S -> call id(L)

L -> L, E

L -> E

S is nonterminal of statement

L denotes the list of parameters

And E denotes the expression it could be id as well

Scanned by CamScanner

The translation Scheme

Production Rule	Semantic Action
S → call id(L)	{ for each item p in queue do append ('param' p); append ('call' id.place) }
L → L, E	{insert E place in the queue}
L → E	{initialize queue and insert E.place in the queue}

→ The data structure queue is used to hold the various parameters of the procedure.

→ The keyword param is used to denote the list of 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 value of parameter which is inserted in the queue

→ For L→E the queue gets Empty and a pointer to the Symbol table is obtained. This pointer denotes the value of E.

Scanned by CamScanner

Back patching:

Implementation of s.d.d using two passes is the most convenient method. If we decide to generate the 3 address code for a given Syntax d.d using single pass only, then the main problem that occurs is the decision of address lablels.

If we use two passes instead of one pass then ~~in one pass~~ then in one pass we can leave these ~~address~~ addresses unspecified and in the second pass this incomplete information can be filled up. To overcome the problem of processing the incomplete information in one pass the backpatching technique is used.

Definition: Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.

In this section we will use

- Boolean Expression
- Flow of control stmt

To generate code using b.p.t. the following functions are used

1. mklist(i) creates the newlist. The index 'i' is passed as an argument to this function

2. merge_list(P1, P2) - this function concatenates two lists pointed by P1 & P2. it returns the pointer to the concatenated list.

3. backpatch(P, i): inserts 'i' as target label for the stmt pointed by pointer p.

Backpatching using Boolean Expressions

consider, the grammar for Boolean expression

$E \rightarrow E_1$ OR M E_2

$E \rightarrow E_1$ AND M E_2

$E \rightarrow$ NOT E_1

$E \rightarrow (E_1)$

$E \rightarrow id_1$ relop id_2

$E \rightarrow$ TRUE

$E \rightarrow$ FALSE

$M \rightarrow \epsilon$

→ Here a new nonterminal M is inserted as a marker nonterminal.

→ The purpose of M is to mark the exact point where the semantic action is picked up.

→ TList & FList are used to generate the jumping code of Boolean expression.

True – E.TList will be generated

False – E.FList " "

→ M is used to ~~store~~ store the address of the stmt. It is denoted as M.State.

Scanned by CamScanner

→ The 'next state' will point to the next stmt.

Production Rule	Semantic Action
$E \to E_1$ OR $M E_2$	{ backpatch (E_1.Flist, M.state) E.Tlist = merge(E_1.Tlist + E_2.Tlist) E.Flist = E_2.Flist }
$E \to E_1$ AND $M E_2$	{ backpatch (E_1.Tlist, M.state); E.Tlist = E_2.Tlist E.Flist = merge (E_1.Flist, E_2.Flist) }
$E \to$ NOT E_1	{ E.Tlist = E_1.Flist; E.Flist = E_1.Tlist; }
$E \to (E)$	{ " }
$E \to id_1$ relop id_2	{ E.Tlist = mklist (nextstate) E.Flist = mklist(nextstate+1); append ('if' id_1.place relop.op id_2.place 'goto _') append ('goto _'); }
$E \to$ TRUE	{ E.Tlist = mklist(nextstate); append('goto _'); }
$E \to$ FALSE	{ E.Flist = mklist (nextstate); append ('goto _'); }
$M \to \epsilon$	{ M.state = nextstate; }

Scanned by CamScanner

Ex: A<B OR C<D AND P<Q

A<B matches the rule $E \to id_1 \text{ relop } id_2$

100 if A<B goto
101 goto

} as semantic actions are append('if' id_1.place relop id_2.place goto _)
append ('goto')

Similarly, for the remaing part of Expr

102 if C<D goto _
103 goto _

104 if P<Q goto _

105 goto _

if A<B OR C<D AND P<Q we will give AND operation first then OR, as AND has higher precedence over OR.

E_1.TList

102 if (C<D) goto _ ← backpatch(E_1.TList, 104)
M.state = nextstate (104)

103 goto _

104 if P<Q goto _

105 goto _

⇒ E.TList = {104}
E.FList = {103, 105}

Hence

102 if C<D goto 104

103 goto _

104 if P<Q goto _

105 goto _

Scanned by CamScanner

Now consider

A<B OR C<D AND P<Q

E_1.Tlist

100 if A<B goto

101 goto — ← backpatch (E1.Flist, 102)

E.Tlist (100, 104)

E.Flist

102 if C<D goto —

103 goto

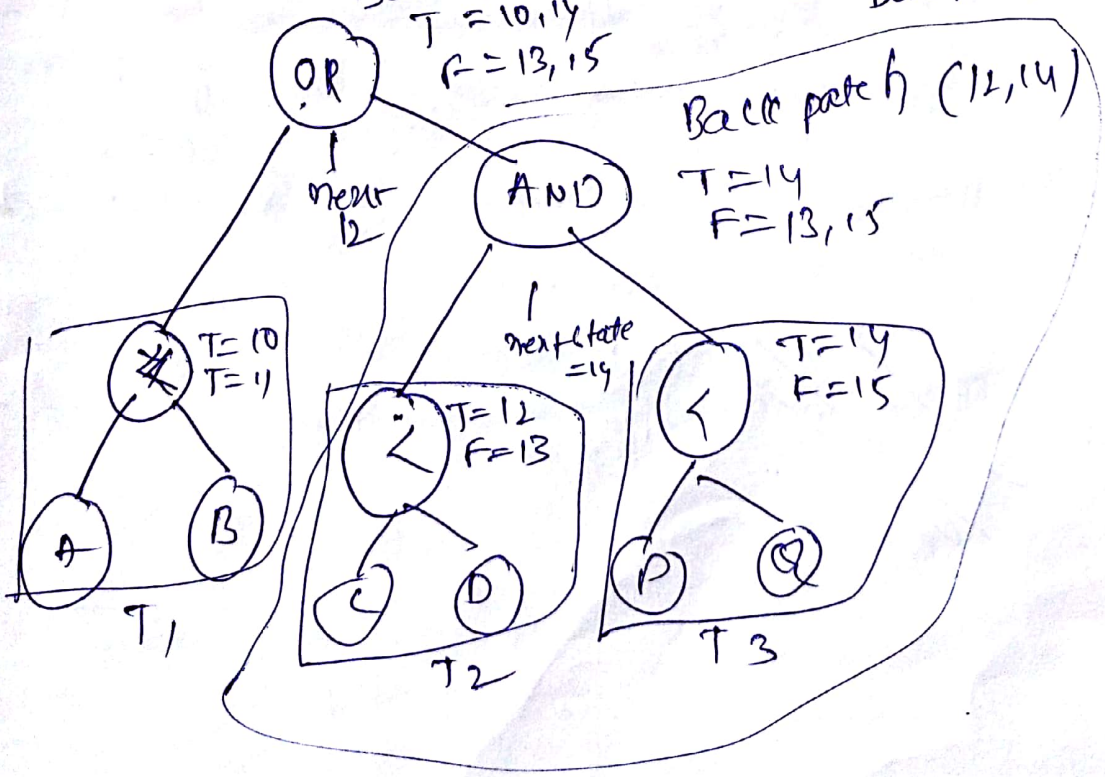
104 if P<Q goto —

105 goto —

E_2.Tlist

T_2	T_3	AND
0	0	0
0	1	0
1	0	0
1	1	1

— backpatch



Scanned by CamScanner

~~Each~~

→ generate 3 address code for

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

2)

```
for (i=1; i<=10; i++)
{
  a[i] = x*5;
}
```

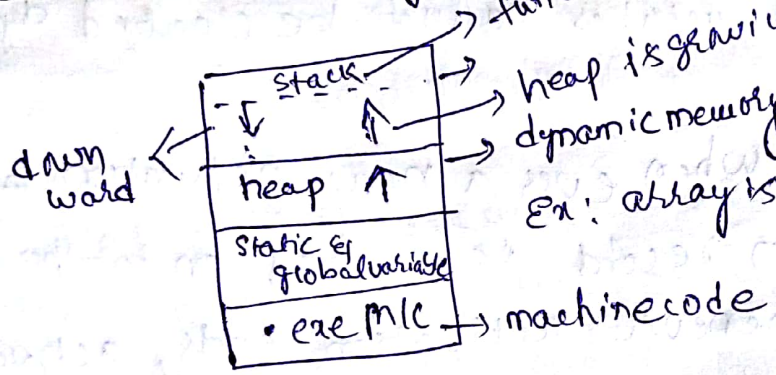
Scanned by CamScanner

Runtime environments: when you the program what is the support that you want from operating System.

How a program is loaded into O.S and what is the support that your program needs

Current State : process

→ Executing a program is called as process and process is managed by O.S that is called O.S process management.



Recursion stack : in that activation records will be pushed.

→ initially they are using only one memory location

→

Scanned by CamScanner

Storage allocation strategies

1. Static:
   1) Allocation is done at Compile time
   2) Bindings do not change at run time
   3) one activation record per procedure

Dis: 1) Recursion is not supported
2) Size of data objects must be known at compile time
3) Data structures cannot be created dynamically

2. Stack: when ever a new activation begins, activation record is pushed on to the stack and whenever activation ends, activation record is popped off.

Local variables are bound to fresh storage

Dis: Local variables cannot be retained once Activation end

3) Heaps: Allocation and deallocation can be in any order

Dis: Heap memory management is overhead

Scanned by CamScanner

1) Source language issues:

There are various languages features that affect the organization of memory. The organization of data is determined by answering the following question.

The source language issues are

1) Does the source lauage allow recursion?

while handling the recursive calls there may be several instance of recursive procedure that are active simultaneously. memory allocation must be needed to ~~procedure~~ store each instance with its copy of local variables and parameters passed to that recursive procedure. But the no. of active instances is determined by run time.

2) How the parameters are passed to the procedure?

There are two methods of parameter passing: call by value & call by reference. The allocation strategies for each of these methods are different.

Some languages support passing of procedure itself as parameter or a return value of the procedure itself could be a procedure.

3) Does the procedure refer nonlocal names? How?

Any procedure has a access to its local names. But a language should support the method of accessing non-local names by procedures.

Scanned by CamScanner

- Does the language support the memory allocation & deallocation dynamically?
  The dynamic allocation & deallocation of memory brings the effective utilization of memory.

Storage organization

- The compiler demands for a block of memory to operating system.
- The compiler utilizes this block of memory for running (executing) the compiled program.
  This block of memory is called 'run time storage'.
- The runtime storage is subdivided to hold code and data such as
  1) The generated target code
  2) Data objects
  3) information which keeps track of procedure activations.

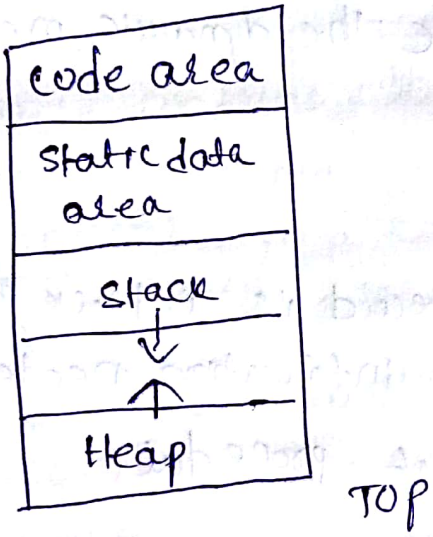
→ The size of the generated code is fixed. Hence the target code occupies the statically determined area of memory.
  Compiler place the target code at the lower end of the memory. compiler places the target code at the lower end of the memory.

Scanned by CamScanner

→ The amount of ~~data~~ memory required by the data objects is known at the compiled time and hence data objects also can be placed at the statically determined area of the memory.

→ The static data area is on the top of code area.

The subdivision of run time memory as shown by following figure



→ stack is used to manage the active procedures.

→ The heap area is the area of run time storage in which the other information is stored.

→ The size of stack and heap is not fixed it may grow or shrink interchangeably during the program execution

Scanned by CamScanner

There are 3 different storage allocation strategies based on this division of runtime storage.

1. Static allocation: The static allocation is for all the data objects at compile time
2. Stack allocation: In the stack allocation a stack is used to manage the runtime storage.
3. Heap allocation: In heap allocation the heap is used to manage the dynamic memory allocation.

Activation Record

- The activation record is a block of memory used for managing information needed by a single execution of procedure.

- FORTRAN uses the static data area to store the activation record whereas in PASCAL & C the activation record is situated in stack area. The contents of activation record are show in following fig.

Return value
Actual parameters
Control link [control optional link]
(Dynamic link)
~~control~~ control Access link (Static link)
Saved machine status
Local variable
Temporaries

Scanned by CamScanner

1. Temporary values: The temporary variables are needed during the evaluation of expressions. such variables are stored in the temporary field of activation record.

2. Local variables: The local data is a data that is local to the execution of procedure is stored in this field of activation record.

3. Saved machine registers: This field holds the information regarding the status of machine just before the procedure is called. This field contains the machine registers & program counter.

4. control link: This field is optional. It points to the activation record of the calling procedure. this link is also called as dynamic link.

5. Access link: This also optional. it refers to the non local data in other activation record. This is also called as static link field.

6. Actual parameters: This fields holds the information about the actual parameters. These actual parameters are passed to the called procedure.

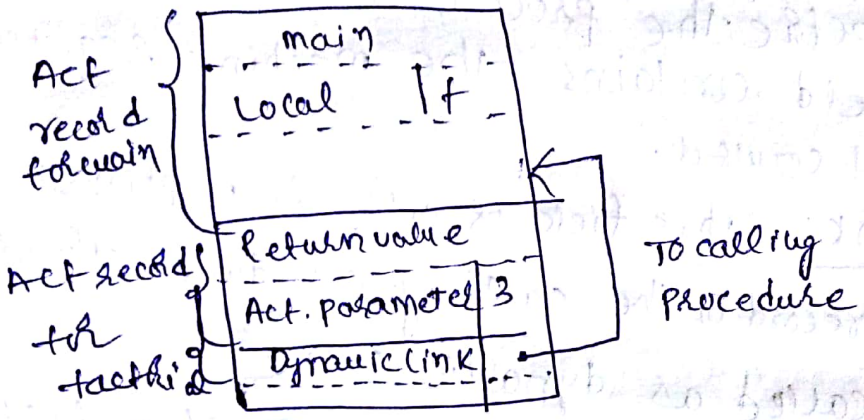
7. Return values: This field is used to store the result of a functional call.

- The size of each field of activation record is determined at the time when a procedure is called.

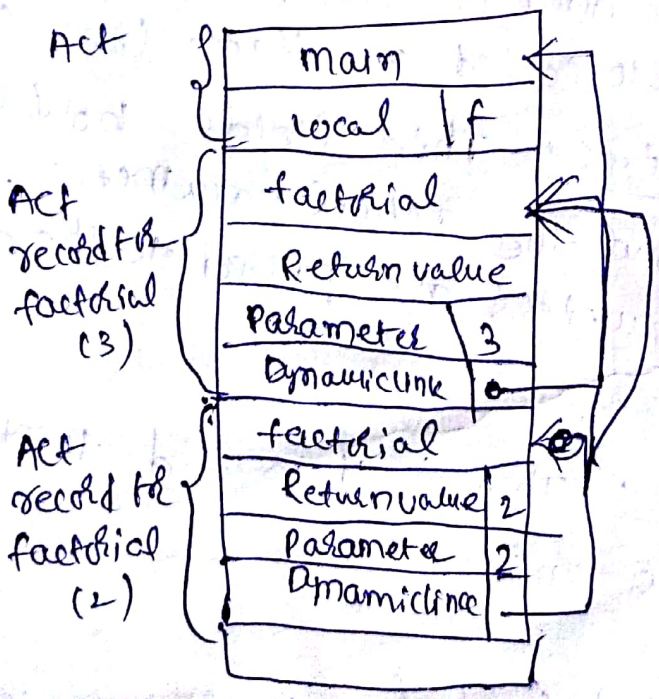
Scanned by CamScanner

Ex:

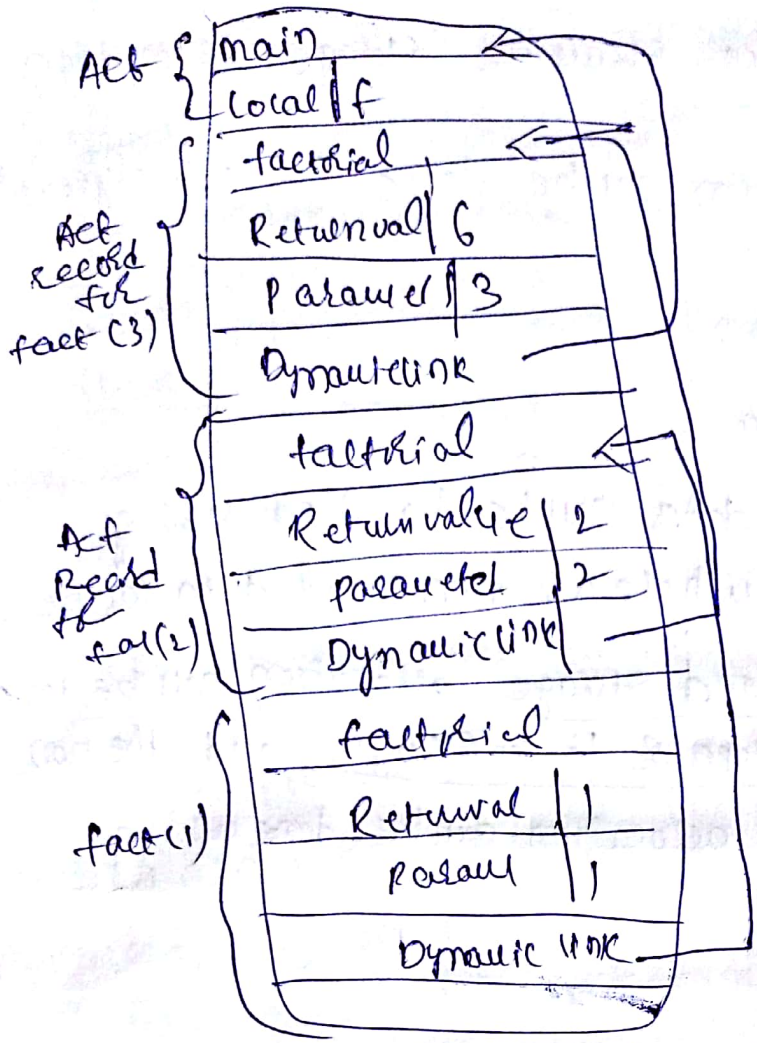
```
main()
{
  int f;
  f= factorial(3);
}

int factorial(int n)
{
  if(n==1)
    return 1;
  else
    return (n* factorial(n-1));
}
```



First call



Scanned by CamScanner



Third call

Variable Length Data

A procedure call sequence can be implemented by caller & callee procedures.

The caller is procedure which calls another produre where as the callee is a procedure which get called by the another procedure.

Scanned by CamScanner

Block & Non Block Structure Storage Allocation

The storage allocation can be done for two types of data variables.

1. Local data
2. Non local data.

→ The local data can be handled using activation record whereas non local data can be

→ The block structured storage allocation can be done using static scope & lexical scope and the non block structured allocation can be done using dynamic scope

Local data

→ The local data can be accessed with the help of activation record.

→ The offset relative to base pointer of an activation record points to local data variables within an activation record.

Reference to any variable x in procedure

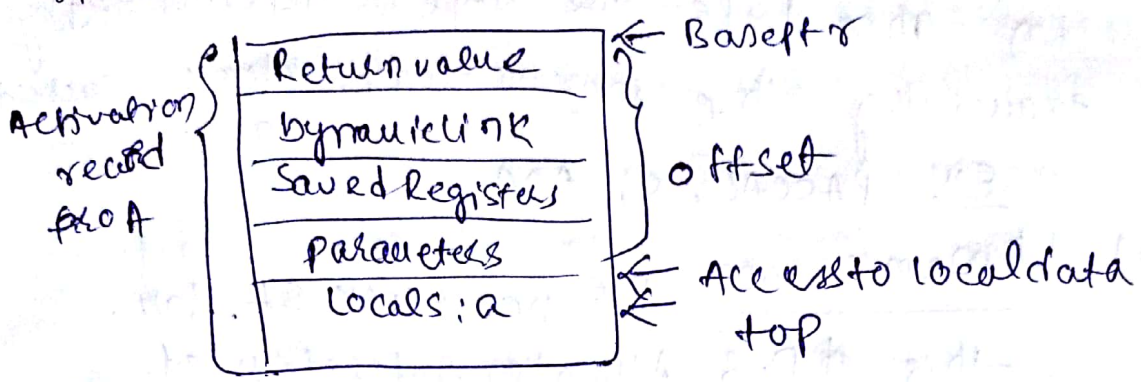
= Base pointer pointing to start of procedure + offset of variable x from base pointer.

ex:

```
procedure A
    int a;
    Procedure B
        int b;
        body of B;
    body of A;
```

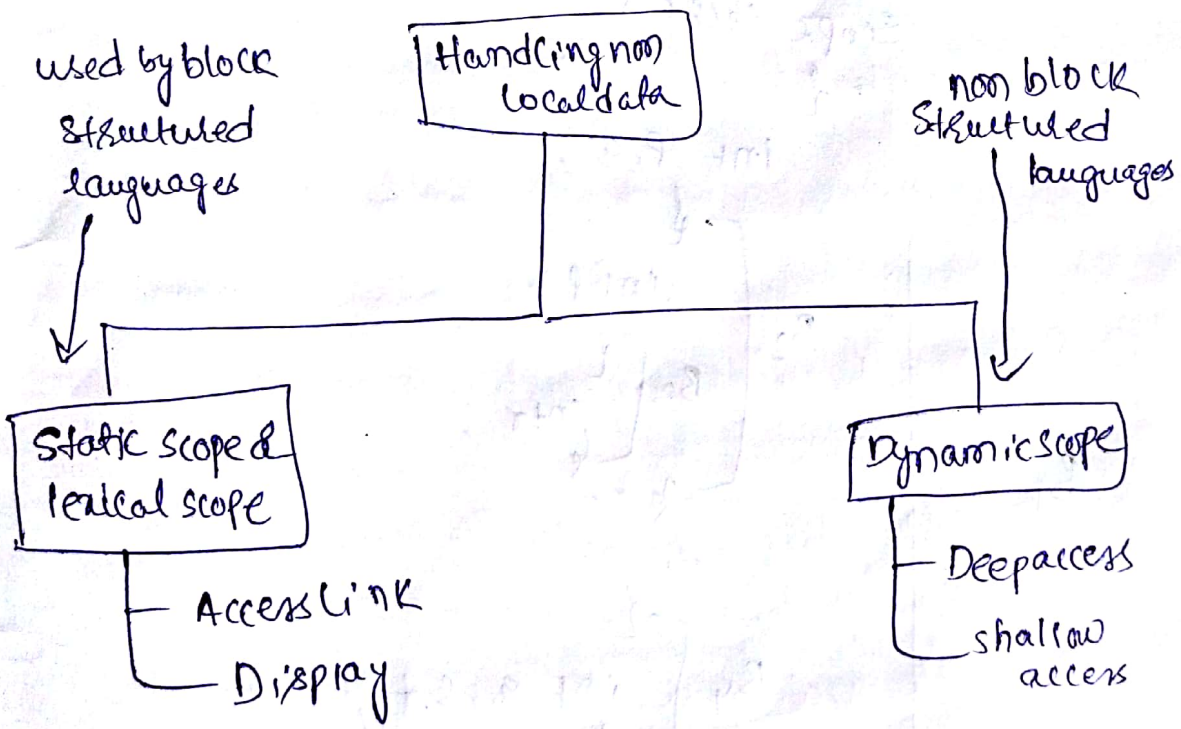
Scanned by CamScanner

The contents of stack along with base pointer and offset are as shown below.



Access to Non Local Names

A procedure may sometimes refer to variables which are not local to it. Such variables are called as non local variables. For the non local names there are two types of scope rules that can be defined : Static & dynamic



Access to non local data.

Scanned by CamScanner

1) Static Scope Rule

in This type the scope is determined by examining the program text. block structured lang

Ex: PASCAL, C, ADA

2) Dynamic scope For non block St. lan.

-this D.S. Allocation rules are used.

The dynamic scope rule determines the scope of declaration of the names at runtime by considering current activation.

Ex: LISP & SNOBOL

Ex: obtain the static scope of the declaration made in the following piece of code

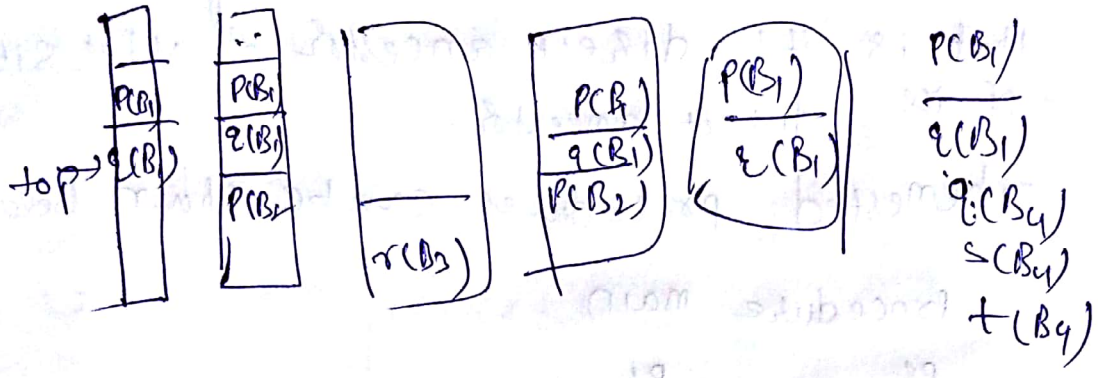
```
scope test()
{
    int p, q;
    {
        int p;
        {
            int r;
        }
    }
    ----
    {
        int q, s, t;
    }
}
```

B_1 B_2 B_3 B_4

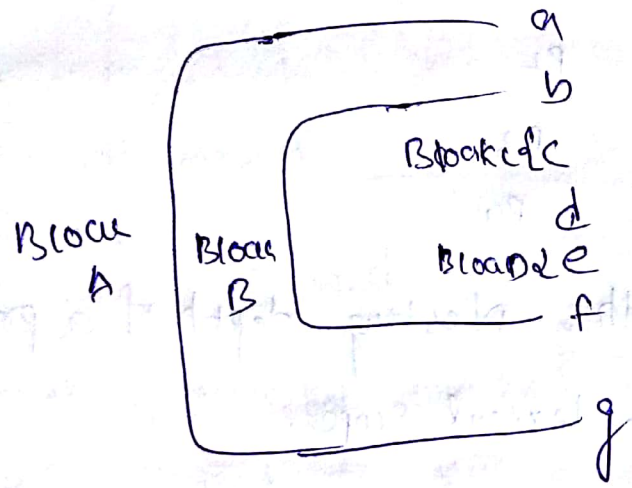
Scanned by CamScanner

Sol: The storage can be allocated for a complete procedure body at one time.

The storage for the names corresponing to particular block can be as shown below.



Ex:



Scanned by CamScanner

Lexical Scope for Nested procedure

- Nested procedure is a procedure that can be declared with in another program.
- A procedure P_i, can call any procedure that is its direct ancestor or older siblings of its direct ancestor.
- The nested procedures can be shown below

```
procedure main
  P2   "    P1
       "    P2
       "    P3
       "    P4
```

- Nesting depth: Nesting depth of a produre is used to implement lexical scope.

The nesting depth can be calculated as follows

i) The nesting depth of main program is 1.

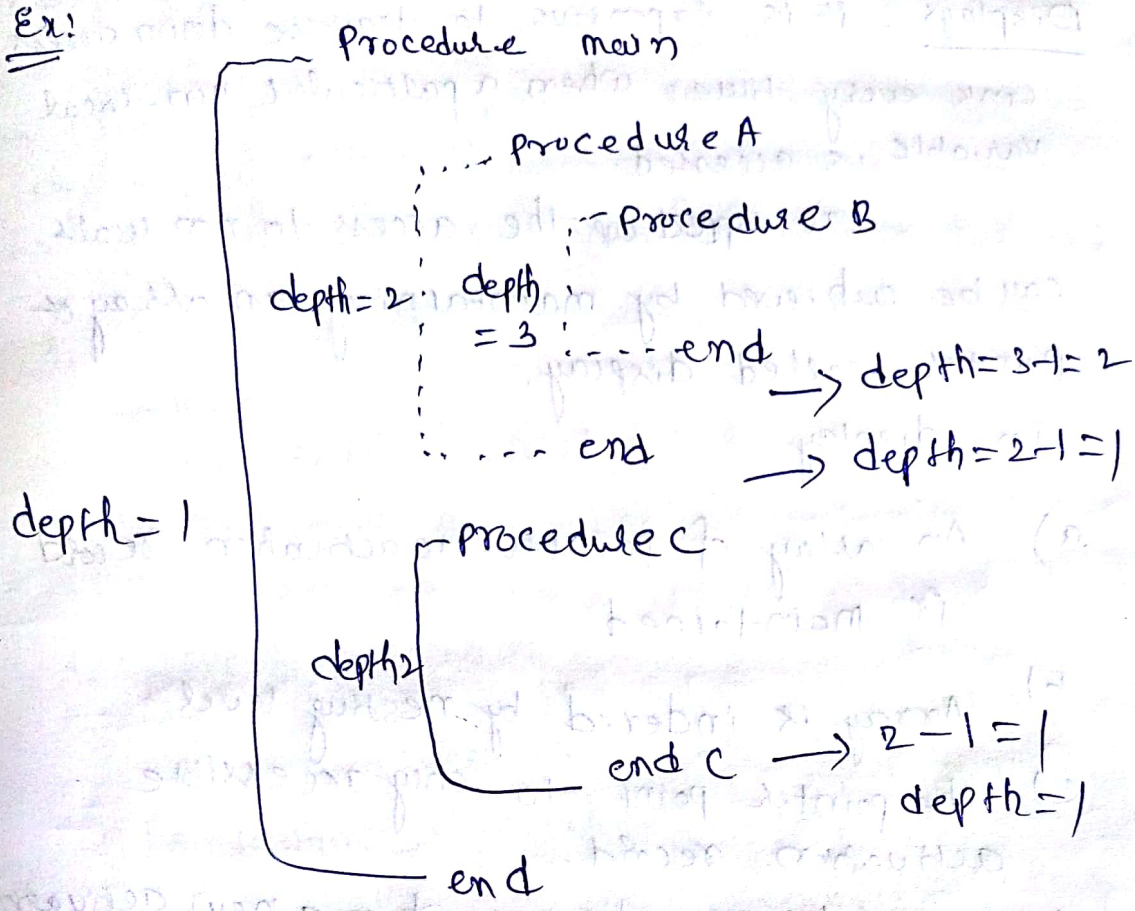
ii) Add 1 to depth each time when a new procedure begins.

iii) subtract 1 from depth each time when you exit from a nested procedure.

iv) The variable declared in specific procedure is associated with nesting depth.

Scanned by CamScanner

Ex:



- The lexical scope can be implemented using access link and displays.

1. <u>Access link</u> : The implementation of lexical scope can be obtained by using pointer to each activation record. These pointers are called access links. If procedure P is nested within a procedure q then access link of P points to access link of most recent activation

Scanned by CamScanner

Displays : it is expensive to traverse down access link every time when a particular non-local variable is accessed.

To speed up the access to non locals can be achieved by maintaining an array or pointer called display.

In display

a) An array of pointers to activation record is maintained

b) Array is indexed by nesting level

c) The pointer point to only accessible activation record.

d) The display changes when a new activation occurs and it must be reset when control returns from the new activation.

Comparision b/w

1. Access link take more time to access non local variables especially when non local variables are at many nested levels away.

2. Display is used to generate code efficiently

3. Display requires more space at the run time than access links.

Scanned by CamScanner

Language Facilities for Dynamic Storage Allocation

→ The data under program control can be allocated dynamically.

→ This allocation is done from heap memory

→ There are two approaches used to allocate data -

- The explicit allocation
- Implicit allocation

Explicit Allocation

E.A and deallocation is done with the help of some functions such as 'new' and 'dispose'.

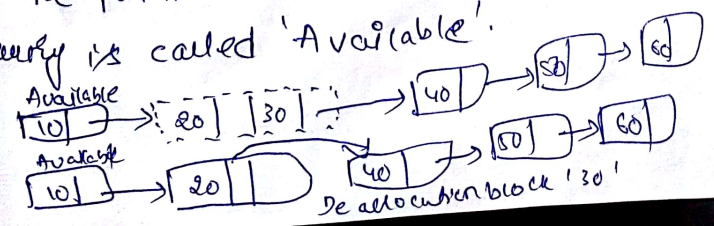
→ implicit A. is done by compiler using runtime support packages.

E A for Fixed Size Blocks

→ In this free list is used. 'Free list' is a set of free blocks.

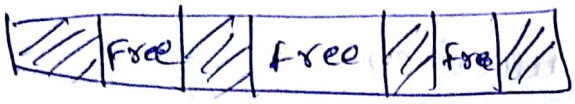
→ This list is observed when we want to allocate memory. If some memory is de-allocated then the free lists gets appended.

→ The pointer which points to first block of memory is called 'Available'.

→ 

Explicit Allocation of variable sized blocks

- Due to frequent memory allocation and deallocation the heap memory becomes fragmented. That means
→ That means heap may consist of some blocks that are free & some that are allocated.

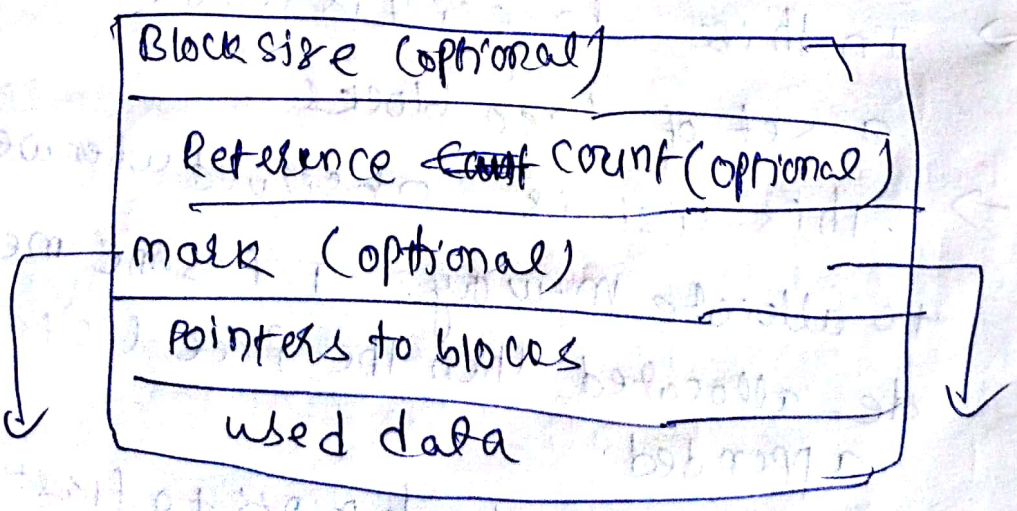


→ For allocating variable sized blocks some strategies such as firstfit, worst fit and best fit are used.

1st fit size of block > procedure

Implicit Allocation

- The implicit allocation is performed using user program and runtime packages.
- The runtime packages is required to know when the 'storage block' is not in use.
- Format of storage block is



Scanned by CamScanner

Problem

→ recognizing block boundaries

→ which block is in use.

Approaches

1) Reference count: If any block is referred by some another block then its reference count incremented by one.

These R.C are best used when pointers blocks never appear in cycle.

2) marking technique:

→ In this method, the user program is suspended temporarily and frequently pointers are used to mark the blocks that are in use.

Garbage

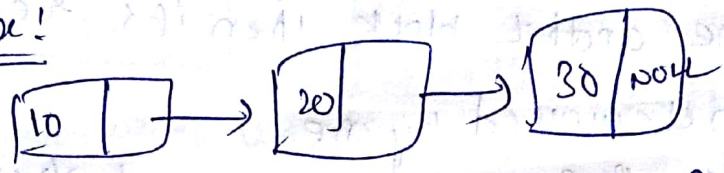
is a substantial amount of memory which gets allocated dynamically but is unreachable memory. If this memory unreacheble then it becomes a wastage of memory.

→ Hence a new technique called 'garbage collection' is introduced. Ex: Java, USP

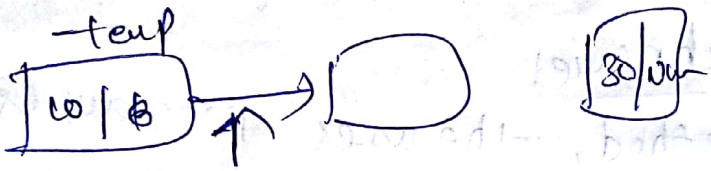
Scanned by CamScanner

Dangling reference is a kind of complication that occures due to explicit deallocation of memory. The dangling reference is a kind of reference that gets deallocated which is referred to

Ex:



Before dangling reference situation



This is a dangling reference

(After dangling)

→ D.R & garbage are two interdependent concepts.

→ That means before reaching to the last node if dangling reference occurs then the last node remains unreachable and garbage exists.

Dynamic Storage Allocations are
- Explicit Allocation
- Implicit Allocation

Scanned by CamScanner

Code optimization

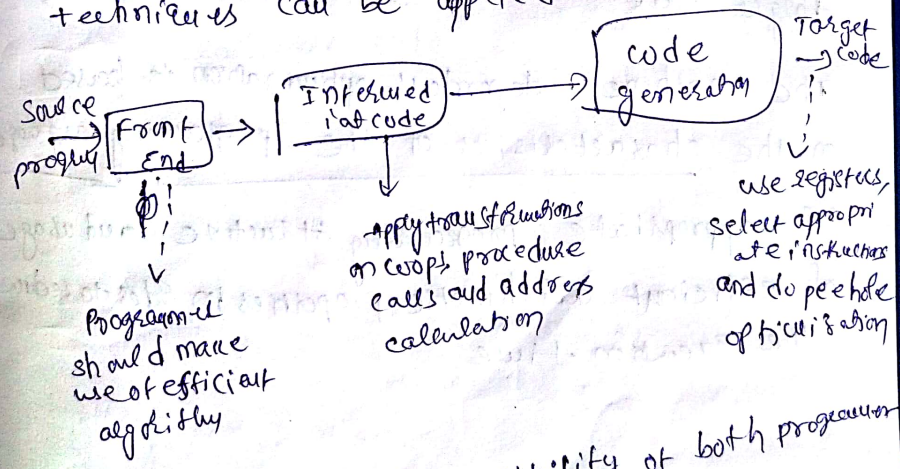
→ The code optimization is required to produce an efficient target code.

→ There are two important issues that need to be considered while applying the techniques for code optimization

1) The semantic equivance of source program must not be changed.

2) The improvement over the program efficiency must be achieved without changing the algorithm of the program.

Block diagram which itself suggests us the places where exactly code optimization techniques can be applied.



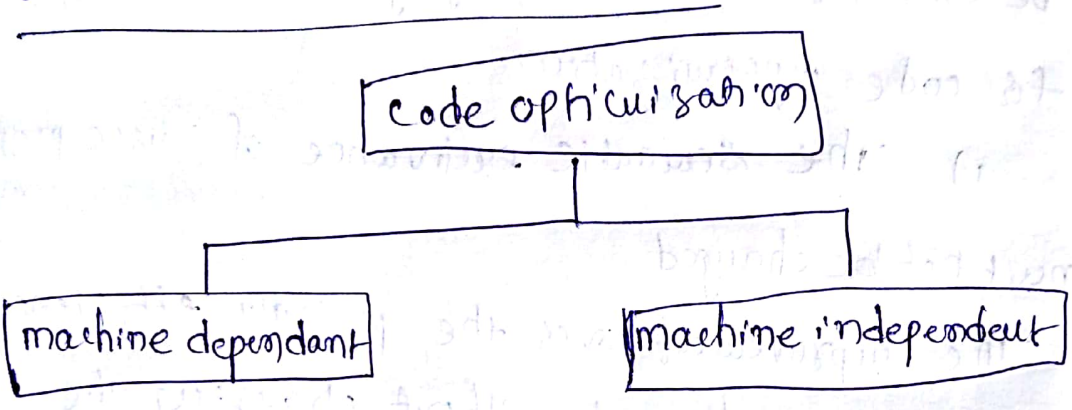
→ it is the responsibility of both programmer and compiler to generate efficient code using proper usage of target code.

Scanned by CamScanner

Advantages

1. The intermediate code can be optimized to produce efficient objectcode
2. The code op. allows ~~ret~~ faster execution of source program by making minor.

Classification of optimization



The machine dependent optimization is based on characteristics of the target machine for the instruction set used and addressing modes used for the instructions to produce the efficient target code.

The machine independent optimization is based on the characteristics of the programming language for appropriate programming structure and usage of efficient arithmetic properties in order to reduce the execution time

Scanned by CamScanner

The machine dependant optimization can be achieved using following criteria.

1. Allocation of sufficient no. of. resources to improve the execution efficiency of program.
2. using immediate instructions wherever necessary
3. The use of intermix instructions.

The machine. I. op. can be achieved

1. The code should be analyzed completely and use alternative equivalent sequence of source code that will produce a minimum amount of target code.
2. use appropriate program structure in order to improve the efficiency of target code.
3. From the source program eliminate the unreachable code.

Scanned by CamScanner

Principle sources of optimization

1. Compile time evaluation : means shifting of computations from runtime to compilation time. There are two methods used to obtain it

1. Folding : the calculation of constant is done at compile time instead of execution time.

Ex: length = (22/7)*d

Here folding is implied by performing the calculation of 22/7 at compile time instead of execution time

2. Constant propagation:

the value of variable is replaced and calculation of an expression is done at the compilation time.

Ex: pi = 3.14
r = 5
Area = pi * r * r

pi is replaced by 3.14 & r by 5 then calculation 3.14 * 5 * 5 is done during compilation

Scanned by CamScanner

common sub expression Elimination

is appearing repeatedly in the program which is calculate previously. ~~then if the operands of the~~

Ex:

$t1 = 4 * i$

$t_2 = a[t_1]$

$t_3 = 4 * j$

$t_4 = 4 * i$

$t_5 = n$

$t_6 = b[t_4] + t_5$

optimized code

$t_1 = 4 * i$

$t_2 = a[t_1]$

$t_3 = 4 * j$

$t_5 = n$

$t_6 = b[t_1] + t_5$

t_4 is eliminated as it is calculation is alread in t_1 and value of i is not been changed

Scanned by CamScanner

Variable propagation

V.P means use of one variable instead of another.

Ex: x = pi;

....

area = x*r*r;

The optimization using variable propagation can be done as follows.

area = pi*r*r

x is eliminated

Code movement

two basic goals of code movement

1. To reduce the size of the code i.e to obtain the space complexity.
2. To reduce the frequency of execution of code. i.e to obtain the time complexity

Ex:

```
for (i=0; i<=10; i++)
{
    x = y*5;
    :
    k = (y*5)+50;
}
```

This can be optimized as

```
temp = y*z;
for (i=0; i<=10; i++)
{   x = z;
```

Scanned by CamScanner

```
k = z+50;
}
```

Loop invariant computation

The loop invariant optimization can be obtained by moving some amount of code outside the loop and placing it just before entering in the loop.

This method is called code motion

Ex:

```
while(i<= max-1)
{
  sum= sum+a[i]
}
```

This can be written as

```
n= max-1
while(i<=n)
{
  sum= sum+a[i];
}
```

Strength Reduction

The strength of certain operations is higher than other.

Ex: * is higher than +.

In this technique the higher strength operators can be replaced by lower strength operators

Ex:

```
for (i=1; i<=50; i++)
{...
 count = i*7
 ...
}
```

can be written as

```
temp = 7
for (i=1; i<=50; i++)
{...
 count = temp;
 temp = temp + 7;
}
```

Dead code Elimination

A variable is said to be live in a program if the value contained into it is used subsequently.

if the value is dead means the value ~~not~~ contained into it is never been used.

The code containing such a variable supposed to be a dead code.

Ex:

```
i = j
...
x = i+10
```

The optimization can be performed by eliminating the assignment stmt i=j.

This assignment stmt is called "dead assignment"

Scanned by CamScanner

Ex: 2

```
i = 0
if(i=1)
{
a = x+5
}
```

→ if is dead code. because it is never satisfied.

Optimization

machine Independent

1) loop optimizations
   a) code motion (or) frequency reduction
   b) loop unrolling
   c) loop jamming
2) folding
   - constant propagation
3) - Redundancy elimination
3) strength reduction

machine dependent

1) Register allocation
2) use of addressing modes
3) Peephole optimization
   a) Redundant = load/store
   b) flow of control optimization
   c) strength reduction
   d) use of machine code idioms.

Scanned by CamScanner

Loop optimizations

→ To apply loop optimization, we must first detect loops.

→ For detecting loops we use control flow analysis (CFA) using program flow graph (PFG)

→ To find PFG, we need to find basic blocks

A Basic block is a sequence of 3-address stmts where controls enters at the begining and leaves only at the end without any jumps or halts.

Finding the Basic blocks

→ In order to find the basic blocks, we need to identify the 'leaders' in the program.

→ Then a Basic block will start from one leader to the next leader but not including next leader

Identifying leaders in a basic block

1) 1st stmt is a leader
2) stmt that is target of conditional or unconditional goto stmt is a leader
3) stmt that follows immediately a conditional or unconditional goto stmt is leader.

Scanned by CamScanner

Ex:

```
fact(x)
{
int f=1
for (i=2; i<=x; i++)
    f= f*i
return f;
}
```

Sol: Three address code

1) f = 1
2) i = 2
3) if(i>x) goto 9
4) t_1 = f * i
5) f = t_1
6) t_2 = i+1
7) i = t_2
8) goto(3)
9) goto

Loop unrolling: In this method the no. of jumps and tests can be reduced by writing the code two time

Ex:

```
int i=1
while(i<=100)
{
a[i]=b[i];
i++;
}
```

⇒

```
int i=1;
while(i<=100)
{
a[i]=b[i];
i++;
a[i]=b[i];
i++;
}
```

Scanned by CamScanner

Loop fusion | Frequency reduction

method Several loops are merged to one loop

Ex:

```
for i=1 to n do
for j=1 to n do
  a[i,j]=10
```

→

```
for i=1 to n*n do
  a[i]=10
```

Basic blocks

The basic block is a Sequence of consecutive stmts in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching.

Leader algorithm Finding basic blocks

Ex:

```
for (i=1 to n)
{
    j=1
    while (j ≤ n)
    {
        A = B*C/D
        j = j+1
    }
}
```

Scanned by CamScanner

Three address code

1)

3)

* 1) i = 1
* 2) if i ≤ n then goto 4
* 3) goto 9
* 4) j = 1
* 5) if j ≤ n then goto 7
* 6) i = i+1 goto 2
* 7) A = B*C/D
  j = j+1
* 8) goto 5
* 9) stop

2)

```
prod = 0
i = 1
do {  prod = prod + a[i] * b[i];
      i = i+1;
   } while(i<=10);
```

3 address code

* 1) prod = 0
  2) i = 1
* 3) L1: t1 = a[i] * b[i]
  t2 = prob + t1
  prob = t2
  i++;
  4) if i<=10 then goto 3
* 5) stop.

Scanned by CamScanner

Optimization of Basic blocks

Two types of optimization that can be done on basic blocks

1. Structure preserving transformations
2. use of algebraic identities

1. Structure preserving transformation

→ by applying some principle techniques such as common sub expression elimination, variable and constant propagation, code movement, dead code elimination.

→ The structure preserving transformation is a DAG based transformation. That means a DAG is constructed for the basic block then the above said transformation applied

→ The common subexpression can be easily detected by observing the DAG for corresponding basic block.

Ex :-

$m = n * p$

$n = m + q$

$p = n * p$

$q = m + q$

if we assume the values $n = 1$, $p = 2$, $q = 3$ &

then the expressions can becomes

$m = 1 \times 2 = 2$

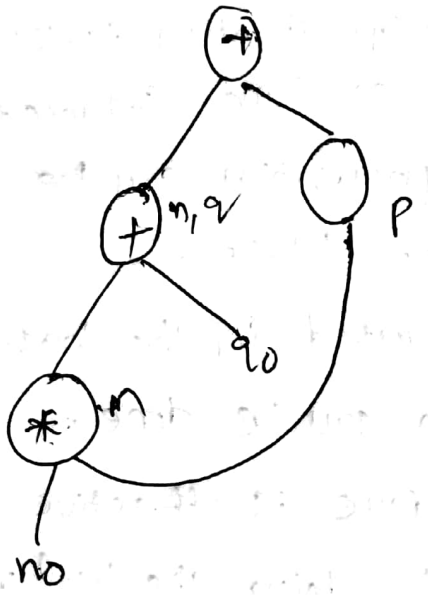
$n = 2 + 3 = 5$

$p = 5 \times 2 = 10$

$q = 2 + 3 = 5$

Scanned by CamScanner

n*p, m+q are common sub expressions, but n is value varies, m+q is not changing, so n=q



use

use of algebraic identities

- Algebraic identities are used in peephole optimization techniques. Also some simple transformations can be applied in order to optimize the code.

Ex: a+0=a, a*1=a, a/1=a

→ a*a ⟹ a+a

→ a/2 → a*0.5

Scanned by CamScanner

Local optimization

→ The local optimization is done at restricted scope

→ In other words local optimization is a kind of optimization that can be done on some sequence of stmts.

→ Typically local optimization can be done within the specific basic blocks.

→ A DAG is constructed for the basic block and then optimization can be done.

→ Peephole technique is effective method of optimization for doing the local optimization.

```
a=b+c
d=e*f
p=b+c
q=e*f
```

DAG Representation of Block

→ The directed acyclic graph is used to apply transformations on the basic block.

→ To apply the transformations on basic block a DAG is constructed from three address stmt.

A DAG can be constructed for the following type of labels on nodes.

1) Leaf nodes are labeled by identifiers or variable names or constants. Generally leaves represent r-values.

2.) Interior nodes store operator values

Scanned by CamScanner

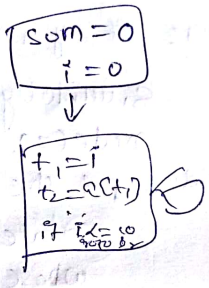
Each node of the flow graph can be represented by DAG because each node of the flow graph is a basic block.

Ex: consider

```
sum=0
for(i=0; i<=10; i++)
    sum=sum+a[i]
```

1) sum = 0
2) i = 0
3) $t_1 = i$
4) $t_2 = a[t_1]$
5) $t_3 = sum + t_2$
6) $sum = t_3$
7) $t_4 = i+1$
8) $i = t_4$
9) if $i <= 10$ goto(3)

basic block



t_1
|
i

Algorithm for construction of DAG

We assume the 3address stmt could of following types.

Case (i) $x = y$ op z

Case (ii) $x =$ op y

Case (iii) $x = y$

with the help of following steps the DAG can be constructed

Step 1: if y is undefined then create node(y). Similarly if z is undefined create a node(z).

Step 2: For the case (i)

Create a node (op), whose left child is node y & node z will be the right child. Also check for any common sub expressions. For the the case (ii) determine whether is a not labeled op, such node will have a child node(y).

in case (iii) node 'n' will be node(y).

Step 3: Delete x from list of identifiers for node(x). Append x to the list of attached identifiers for node n found in 2.

Application of DAG: used in

1. Determining the common sub-expressions
2. Determining which names are used inside the block and computed outside the block.
3. Determining which stmts of the block could have their computed value outside the block.
4. Simplifying the list of quadruples by eliminating the common sub-expressions and not performing the assignment of the form $x = y$ unless and until it is a must.

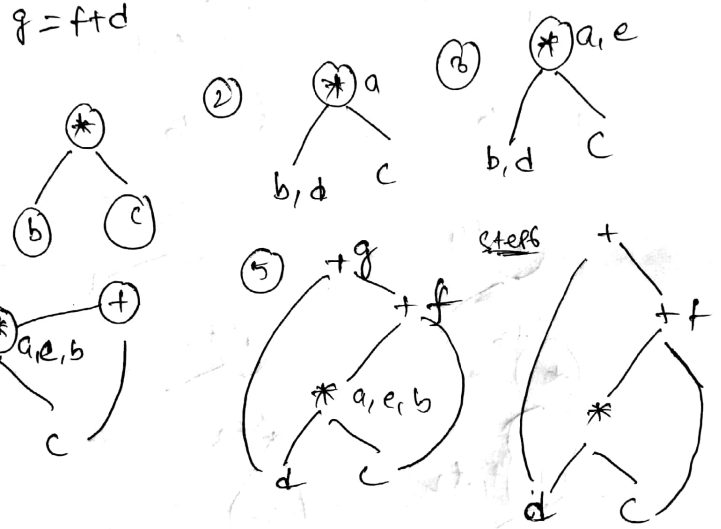
Scanned by CamScanner

DAG based local optimization

Dead code elimination can be applied for

Ex: $a = b * c$
$d = b$
$e = d * c$
$b = e$
$f = b + c$
$g = f + d$

Step: 1

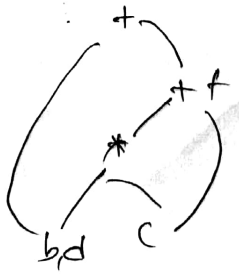


The optimized code can be generated by traversing the DAG.

The local optimization on the above block can be done

1. $e = d * c$ actually $b * c$ (since $d = b$) is eliminated
2. dead code $b = e$ is eliminated

$a = b * c$
$d = b$
$f = a + c$
$g = f + d$



Scanned by CamScanner

2) Explain how following expression can be converted in a DAG $a+b*(a+b)+(c+d)$

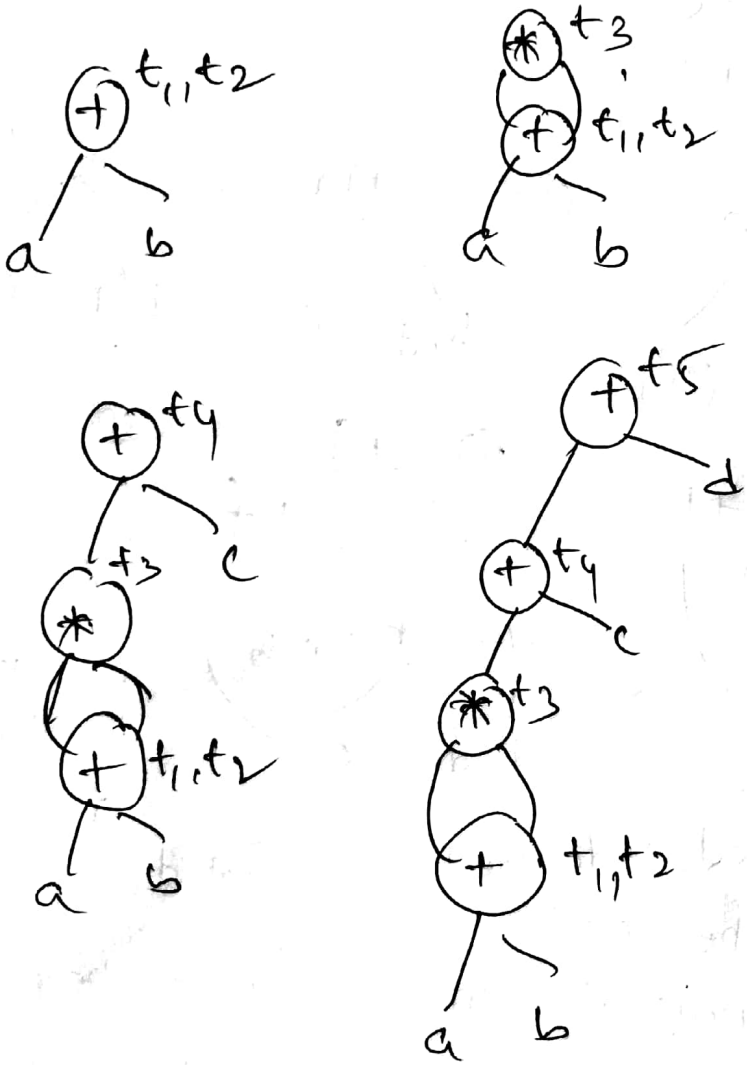
$t_1 = a+b$

$t_2 = a+b$

$t_3 = t_1 * t_2$

$t_4 = t_3 + c$

$t_5 = t_4 + d$



Scanned by CamScanner

DAG

Rule 1: In a DAG

→ Leaf nodes represents identifiers, names or constants

→ Interior nodes represent operators

Rule 2: while constructing DAG, there is a check made to find if there is an existing node with the same children.

A new node is created only when such a node does not exist. This action alows us to detect ~~only~~ common sub expressions and eliminate the re-computation of the same.

Rule 3: The assignment of from $x = y$ must not be performed until and unless it is a must

Ex:

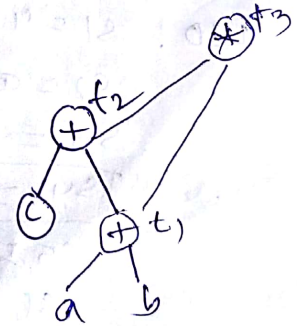
$(a+b) * (a+b+c)$

Sol: Three address code

$t_1 = a+b$

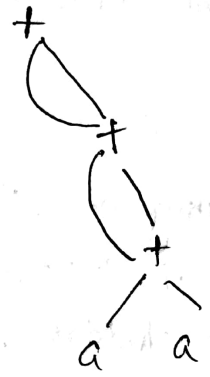
$t_2 = t_1 + c$

$t_3 = t_1 * t_2$



Scanned by CamScanner

2) $(((a+a)+(a+a))+((a+a)+(a+a)))$



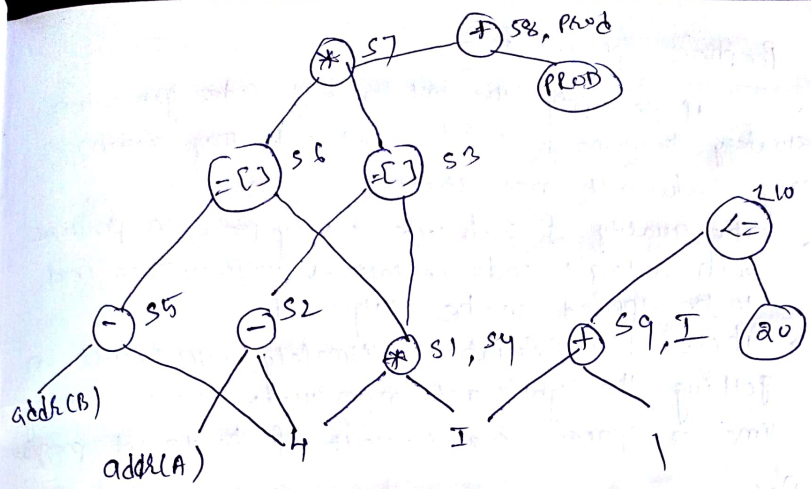
3) ~~Example in DAG optimization~~

~~Sol: 1) First construct the DAG~~

Ex: consider the following basic block.
Draw the DAG representation of block and identify local common sub expressions.
Eliminate the common expressions and rewrite the basic blocks.

L10: S1 = 4*I
S2 = addr(A) - 4
S3 = S2[S1]
useless $\boxed{S4 = 4*I}$
S5 = addr(B) - 4
S6 = S5[S4]
S7 = S3*S6
S8 = PROD+S7
PROD = S8

S9 = I+1
I = S9
IF I < = 20 goto L10



-> 4*I is already calculated so, S4 is eliminate
S1=S4

-> S8 = PROD+S7 -> PROD = PROD+S7
PROD = S8

-> S9 = I+1 ---> I = I+1
I = S9

After eliminating S4, S8, & S9 we get

L10: S1 = 4*I
S2 = addr(A) - 4
S3 = S2[S1]
S5 = addr(B) - 4
S6 = S5[S1]
S7 = S3*S6
PROD = PROD + S7
I = I+1
If I <= 20 goto L10

Scanned by CamScanner

Peephole optimization

→ If we apply the stmt by stmt code generation strategy then the generated target code may contain many redundant instructions.

→ The quality of such code is very poor. To optimize such a target code certain transformations need to be applied on the target code.

→ These transformations ultimately will result in getting the significant improvement over the running time or space requirement of the target program.

Definition : Peephole optimization is a simple and effective technique for locally improving target code.

→ This technique is applied to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter or faster sequence.

Characteristics of Peephole optimization

The peephole optimization can be applied on the target code using following characteristics.

1. Redundant instruction elimination
2. Flow of control optimization
3. Algebraic simplification
4. use of machine idioms

Scanned by CamScanner

1. Redundant instruction elimination

- Especially the redundant loads and stores can be eliminated in this type of transformations.

Ex: MOV R0, X
MOV X, R0

we can eliminate the second instruction X is already in R0.

But if (mov X, R0) is a label stmt then we cannot remove it.

- we can eliminate the unreachable instructions.

Ex:
```
sum = 0
if(sum)
  Print("%d", sum)
```
it eliminated

2. Flow of control optimization

using peephole optimization unnecessary jumps or jumps can be eliminated.

Ex:
```
goto test
....
test: goto done
....
done:
```
⇒
```
goto done
...
test: goto done
...
done:
```

Thus we reduce one jump by this transformation

Exam:
```
if a<b goto test
....
test: goto done
done ; --
```
⇒
```
if a<b goto done
....
test: goto done
...
done: --
```

Scanned by CamScanner

3. Algebraic Simplification

Peephole optimization is effective technique for algebraic simplification.

The stmt such as

$x = x + 0$ or $x = x * 1$ can be Eliminated by Peephole optimization

Machine idioms

→ The target instructions have equivalent machine instructions for performing some operations

→ Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.

→ Ex: Some machines have auto-increment (or) auto-decrement addressing modes that are used to perform add & subtract operations.

Scanned by CamScanner

Exam

1) write S-attribute grammar to convert the given grammar with infix operators to prefix operator.

$L \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow E-T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow T/F$

$T \rightarrow F$

$F \rightarrow F \uparrow P$

$F \rightarrow P$

$P \rightarrow (E)$

$P \rightarrow id$

2) Consider the CFG.

$E \rightarrow TR$

$R \rightarrow +TR \mid -TR \mid \epsilon$

$T \rightarrow num$

with translation schema to generate Postfix expression equivalent to the given infix expression which is recognized by above grammar. All actions in the translation should be at the end of each production.

Scanned by CamScanner

Flow graphs:

A flow graph is a directed graph in which the flow control information is added to the basic blocks.

- The nodes to the flowgraph are represented by basic blocks
- The block whose leader is the first stmt is called initial block.
- there is a directed edge from block B_1 to block B_2 immediately follows B_1 in the given sequence.
- B_1 predecessor of B_2.

Loop:

→ loop is collection of nodes in the flow graph.

i) All nodes are strongly connected. That means always there is a path from any node to any other node within that loop

ii) The collection of nodes has unique entry. That mean there is only one path from a node outsid the loop to the node inside the loop.

iii) The loop that contains no other loop is called inner loop.

Scanned by CamScanner

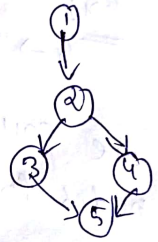
Loops in Flow graphs

1. **Dominators:** a node 'd' dominates 'n' if every path to node n from initial node goes through d only.

This can be denoted as 'd dom n'.

- Every initial node dominates all the remaining nodes in the flowgraph.
- every node dominates itself.

Ex:



Node 1 is initial node and it dominates every node as its a initial node.

Node 2 dominates 3, 4, 5 as there exists only one path from initial node to node 2 which is going through 2.

Node 3, 4 dominates itself

Node 5 dominates no nodes.

Natural loops:

Loop in flow graphs can denoted by $n \to d$ such that 'd dom n'

These edges are called 'back edges' and for a loop there can be more than one back edges.

Scanned by CamScanner

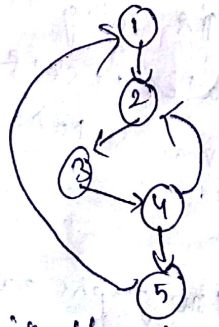
If there is $p \to q$ then q is head and p is a tail.

The natural loop can be defined by a back edge $n \to d$ such that there exists a collection of all the nodes that can reach to 'n' without going through d and at the same time d also can be added to this collection

Ex:

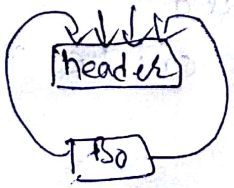
$6 \to 1$ is a natural loop because we can reach to all the remaining nodes from 6.

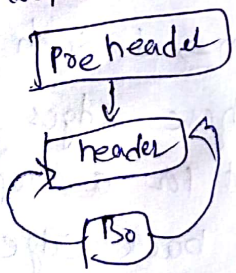
3. Inner loop : The inner loop is a loop that contains no other loop.



$4 \to 2$ is the inner loop.

4) Pre-header





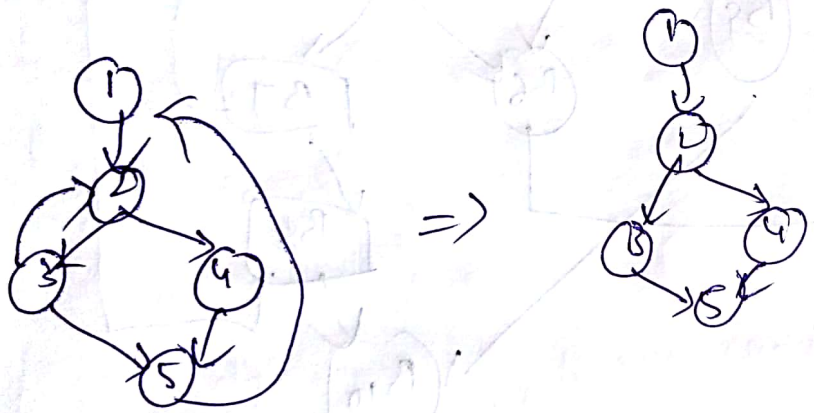
Scanned by CamScanner

The pre-header is a new block created such that successor of this block is the header block. All the computations that can be made before the header block can be made before the pre-header block.

Reducible flowgraph

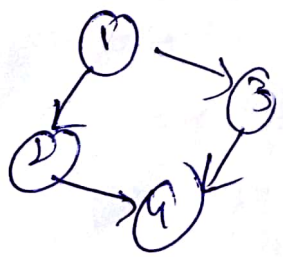
is a flowgraph in which there are two types of edges forward edges and backward edges.

1) The forward edge ~~from~~ an acyclic graph
2) The backedges are such edges whose head dominates their ~~head~~ tails.



Non-reducible flowgraph

1) There are no back edges
2) Forward edges may produces cycle in the graph.

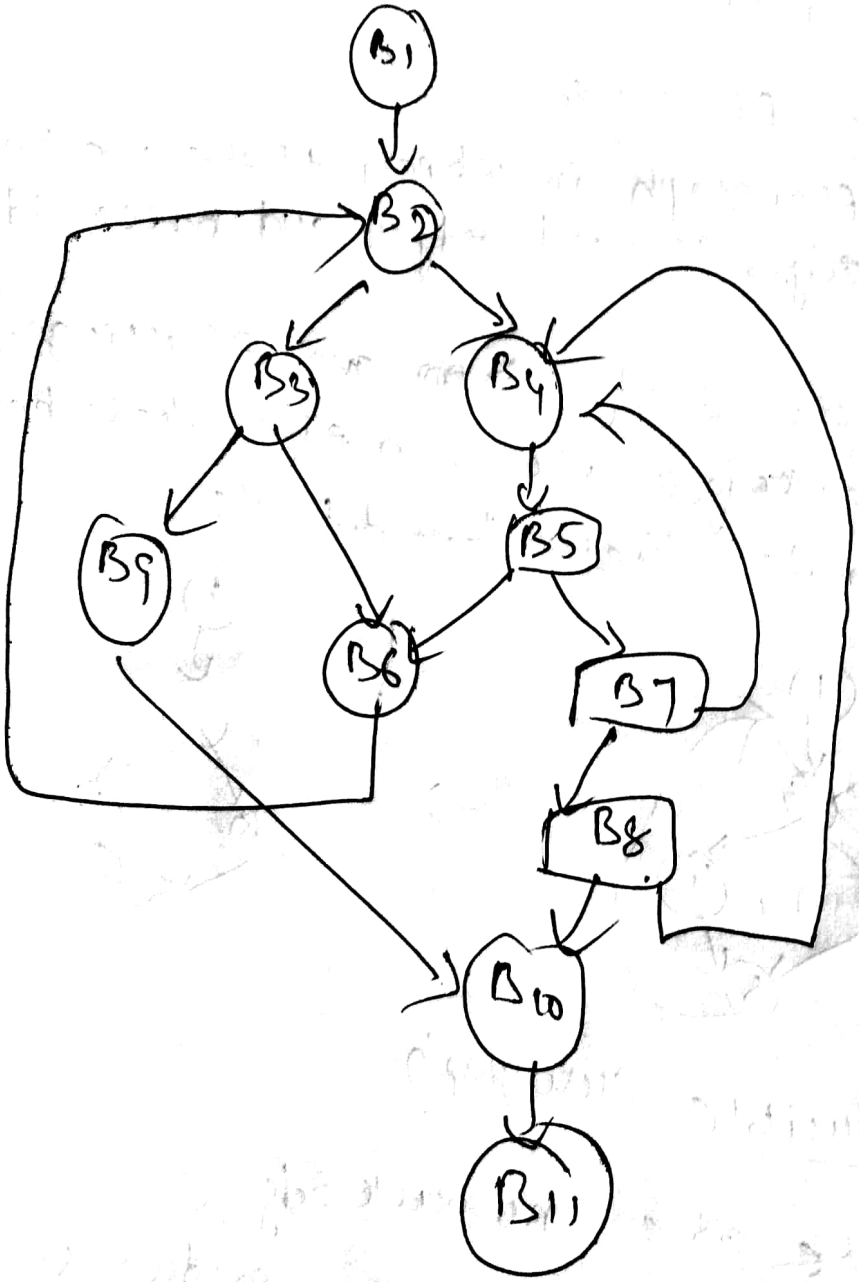


Scanned by CamScanner

Ex: consider flowgraph find

a) Dominators

b) Detect all the loops, find back edge and header block



Scanned by CamScanner

Global optimisation

→ G.O is applied over a 'broad scope' such as procedure or function body.

→ G.O → control flow analysis
G.O → Data flow analysis

con.f. & Data F'A

→ The C.F determines the information regarding 'arrangement of graph nodes' (basic block), presence of loops, nesting of loops. and nodes visited befor execution of a specific code.

→ In ~~CF~~ this Analysis is made on the flow of control.

→ In D.F Analysis, the analysis is made on the dataflow.

Scanned by CamScanner

consider flowgraph code

```
if (i>0)
  {
  som = B[0];
  i=0
L1: if(A[i] < B[i])
    {
      j=1;
L2:   if(B[i]>=0)
      {
        som = som+B[j];
      }
      j=j+1
      if(j<N) goto L2
    }
    //i=i+1
    }
    i=i+1;
    if(i<N) goto L1;
  }
  P.f("som=%d\n",som);
```

Draw a controlflowgraph, find Dominat etc.

Build the control flow graph

```
L: a=k+2
   c=d-b
   d=a+b
   if(d>i) goto E
   f=b-d
   k=d-2
   b=a+f
   if(d<i) goto B
   d=b*2
   g=2*d
B: i=i+1
   b=d-1
   goto L
E: k=a-e
   f=e+k
   c=d+b
```

Apply dead

→ d

L

B1 c is u

B3:

Scanned by CamScanner

Apply dead code Elimi

→ d = a+b

L: a = k+2

c = d-b

b = a+b

if(d>i) goto E

f = b-d

K = d-2

b = a+f

goto L

E: f = e+K

c = d+b

return C

B1 C is used for the

B3:

Scanned by CamScanner

Code Generation

→ C.G is a final activity of compiler.

→ Basically C.G is a process of creating assembly language / machine language stmts which will perform the operations specified by the source program when they run.

Properties

→ Correctness - It should produce a correct code and donot alter the purpose of source code.

→ High quality - It should produce a high quality object

→ Efficient use of resources of target machine -

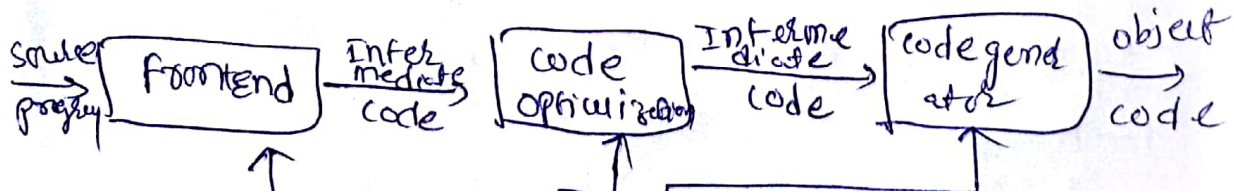
The C.G phase can make an efficient use of resources of the target machine.

Ex: memory allocation & deallocations, arithmetic logic unit operations.

Quick code generation: It is necessary that the code generation phase should produce the code quickly after compiling the source program.

Machine Dependent code Generation

C.G takes intermediate code as an i/p and generates target machine code as o/p.



Scanned by CamScanner

The o/p of C.G is an object code or machine code.

This code normally comes in following forms

1. Absolute code
2. Relocatable machine code
3. Assembler code

1. Absolute code: A.C is a machine code that contains reference to actual addresses within program's address space.

→ The generated code can be placed directly in the memory and execution starts immediately.

→ Generally small programs can be compiled and executed quickly because of absolute code generation.

2. Relocatable code: producing a relocatable machine language program as o/p allows subprograms to be compiled separately.

→ A set of relocatable object modules can be linked together and loaded for execution with the help of a linking loader.

→ The advantage of generating relocatable machine language code is flexibility in in being able to compile subroutines separately and to call other previously compiled program from an object module.

Scanned by CamScanner

3. Assembler code: Producing an assembly language program as o/p makes the process of code generation some what easier

→ we can generate symbolic instructions and use the macro facilities of the assembler to help in generation of code.

But generating assembler code as an o/p makes C.G process slower. because of assembling, linking and loading

Issues in code generation

1) I/P to the code generator.

2) Target programs

3) memory management

4) Instruction selection

5) Register allocation.

6) Choice of Evaluation order

7) Approaches to code generation.

Scanned by CamScanner

Target machine

For designing the good code generator it is necessary to have prior knowledge of target machine and instruction set used for this target machine.

- we will assume that in the target computer addresses are given in bytes and four bytes form a word.
- There are 'n' general purpose registers
  R0, R1, ... Rn-1.
- The two address instruction is of the form
  op source destination
  op is opcode
  source & destination are data fields.
  Ex: MOV x, y
- The addressing modes used as follows

Addressing mode	form	Address	Added cost
absolute	M	M	1
register	R	R	0
indexed	c(R)	c+ contents(R)	1
indirect register	*R	contents (R)	0
indirect indexed	*c(R)	contents (c+contents(R))	1
literal	#c	c	1

Scanned by CamScanner

→ If we have absolute (or) register addressing mode we can use R1 or register for source & destination

Ex: MOV ~~R1, M~~ R1, M ~~M, R1~~ stores the contents of R1 into M

→ For indexed addressing mode the address offset 'c' from the value of register R0 can be written as

MOV 7(R1), M

Means ~~com~~ contents(7+contents(R1)) to the memory location M

→ The last two addressing modes represent the indirect addresses indicates by '*'.

MOV *7(R0), M

contents (content(7+contents(R0)) to M

→ In the literal addressing mode the source becomes constant

Ex: MOV #5, R0

constant 5 into Register R0.

cost of

The

as one plus

destination

Instruction

MOV R0, R1

MOV R1, M

SUB 5(R0)

Ex: comput

MOV

Add

MOV

Ex:

Scanned by CamScanner

cost of instruction

The instruction cost can be computed as one plus cost associated with the source and destination addressing modes given by "added cost"

Instruction	Interpretation	cost
MOV R0, R1	cost of register mode $+1 = 1$ $= 0+1 = 1$	1
MOV R1, M	Memory variable $+1$ $= 1+1 = 2$	2
SUB 5(R0), 10(R1)	first constant + offset second constant $+1$	= 3

Ex: compute the cost of following set of instructions

MOV a, R0	2
Add b, R0	2
MOV R0, C	2
	cost = 6

Ex:

MOV *R1, *R0	= 1
ADD *R2, *R0	= 1
	2

Scanned by CamScanner

Ex

i) MOV B, R0 ⇒
ADD C, R0
MOV R0, A

ii) MOV B, A
ADD C, A

MOV B, R0

MOV → cost of instruction (1)

B → cost of Memory location (1)

R0 → cost of Register (0)

So: MOV B, R0 → cost = 1+1+0 = 2

cost = 1+1+0 = 2

1+0+1 = 2

6

ii) MOV B, A

MOV → cost of instruction 1

B → cost of Memory location 1

A → cost of Memory location 1

= 3

ADD C, A ⇒ 3

6

Storage for Temporaries ⇒ :- For each temporary each time a space gets allocated. To have optimization

a = c+d mapping to $r_1 = r_2 + r_3$

e = a + b mapping $r_1 = r_1 + r_4$

f = e − 1 $r_1 = r_1 - 1$

- In the process of code gen. we can pack two temporaries into the same location if they are not live simultaneously.

Register A

→ The w
memory o
→ There a
how to
are to

• The r
register
specific

For
regis

• The
design
becom
varia

1. Gl
2. C
3.
4.

Scanned by CamScanner

Register Allocation and Assignment

→ The use of register operands instead of memory operands is always the faster & shorter.

→ There are various strategies for deciding how to use these registers and what values are to be stored in the registers.

- The most commonly used strategy to register allocation & assignment is to assign specific values to specific registers.

For base addresses separate set of registers can be assigned.

- The advantage of this approach is that the design process of compiler for code generation becomes simplified

Various strategies used in register allocation

1. Global register allocation
2. Usage count
3. Register assignment for outer loop
4. Graph coloring for Register assignment.

Scanned by CamScanner

Global. R.A : while generating the code the

→ registers are used to hold the values for the duration of single block.

→ All the live variable are stored at the end of each block.

→ For the variable that are used consistently we can allocate specific set of registers.

→ Hence allocation of variables to specific registers that is consistent across the block boundaries is called "global register allocation"

Strategies

1. The global register allocation has a strategy of storing the most frequently used variables in fixed register throughout the loop.
2. The Active values in each inner loop can be stored in fixed no. of global register.
3. The register not already allocated may be used to hold values local to one block
4. In certain language like C or BLISS programmer can do the register allocation by using register declaration.

Scanned by CamScanner

usage count

-> The usage count is the count for the use of some variable x in some register used in any basic block.

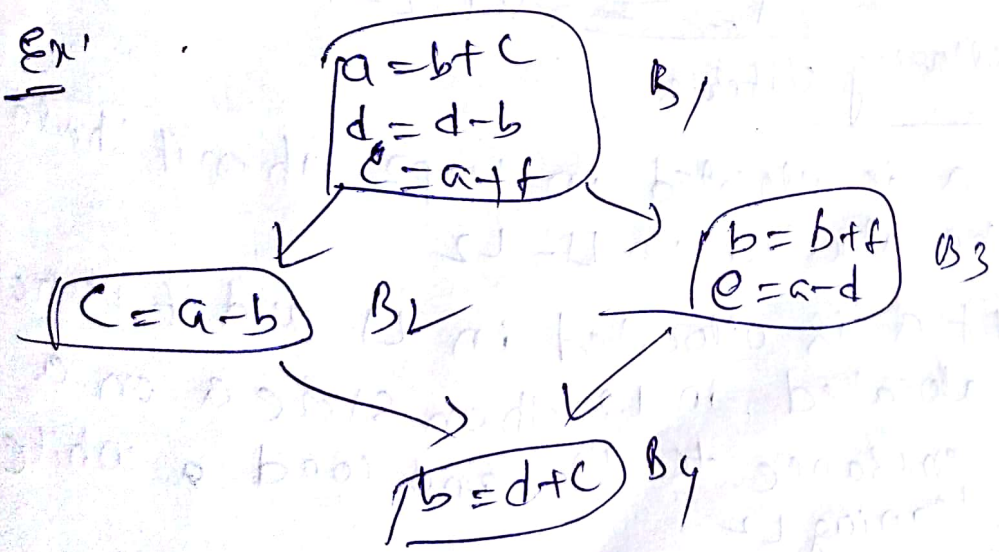
-> The usage count gives the idea about how many units of costs can be saved by selecting a specific variable for global register allocation.

-> The count can be calculated

$$\sum_{\text{block } B \text{ in } L} (use(x,B) + 2 * live(x,B))$$

where use(x, B) is no. of times x used in block B prior to any definition of x and live(x, B) = 1 if x is live on exit from B otherwise live(x) = 0

Ex:



Scanned by CamScanner

B1, a

use(a, B1) = 0 ∴ a is defined in B1 before use

2* live(a, B1) = 2 ∴ a is live on Exit of B1 hence live(a, B1) = 1

use(a, B1) + 2* live(a, B1) = 2

B2, B3 → a

use(a, B2) = 1 use(a, B2) = 1

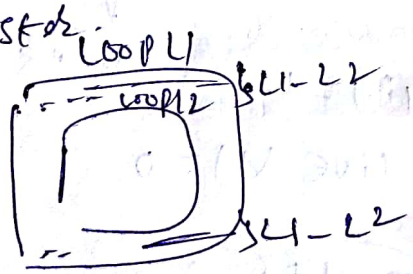
use(a, B4) = 0, & use(a, B) = 4

B1 B4

Register allocation Assignment for outer loop

Consider two loop L1 is outer loop and L2 is an inner loop.

allocation of variable 'a' is to be done to some register



Following criteria

1) If a is allocated in loop2 then it should not be alocated in L1-L2

2) If a is allocated in L1 and if is not alocated in L2 then store a on a entrance to L2 and load a while leaving L2

3) If a is allocated in L2 and not in L1 then load a on entrance of L2 and store a on exite from L2.

Graph color

note

Some ti

Reusability

→ whi

a big que

to sol

technique

In the

The graph

1. In the

is sele

For e

is a

2. In the

graph is

In le

is a sym

two nodes

other is

3. then a

for this

K-colo

to be

→ In regis

each node

so that

interfere

Scanned by CamScanner

Graph coloring for Register assignment

not to adcent nodes have same color.

Sometimes some registers free for reusability.

→ which register is can be freed is a big question

To solve this problems a graph coloring technique is used.

In the first phase

The graph coloring is work is two passes.

1. In the first pass machine instruction is selected for register allocation.

   For each variable a symbolic register is allocated

2. In the second pass the register interence graph is prepared.

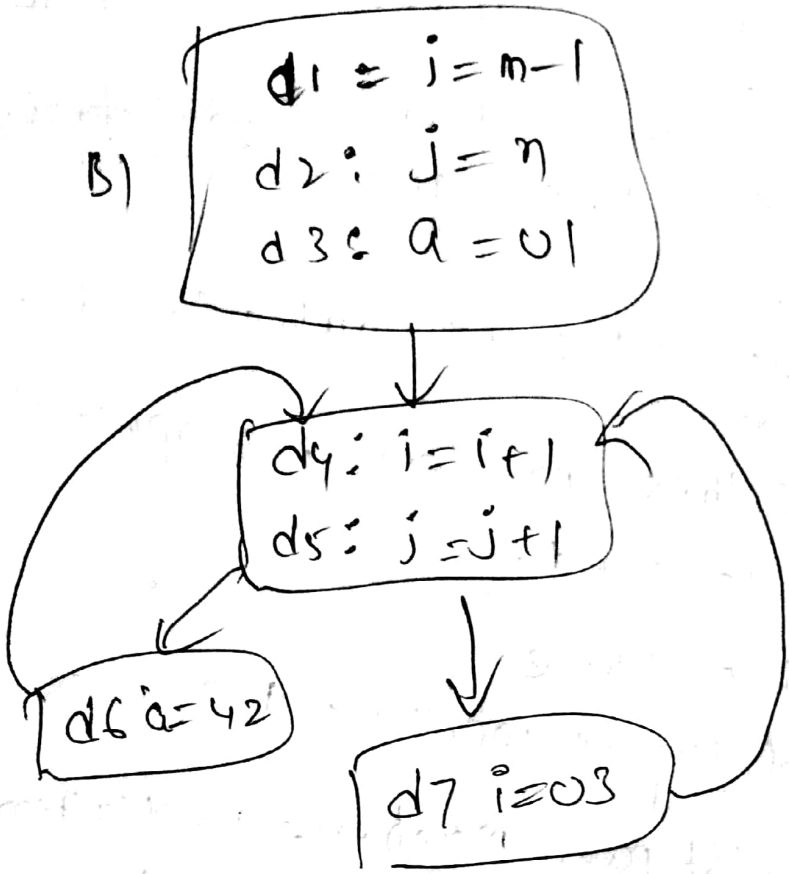
   In register interence graph each node is a symbolic registers and an edge connects two nodes where one is live at a point other is defined.

3. Then a graph coloring technique is applied for this register inference graph using K-color. The K-color can be assumed to be no. of assignable registers.

→ In register interence graph using principle each node is assigned the symbolic registers so that no two symbolic registers can interfere with each other

Scanned by CamScanner

B)



GEN(B) = set of definitions generated by B

KILL(B) = " " " that can not reach the end of B.

~~GEN(B) =~~

Block	GEN(

Scanned by CamScanner