19/10/23

UNIT-III

SDD (Syntax Directed Definition)

SDD is a context free grammar together with attributes & rules.

Attributes are associated with grammar symbols & rules are associated with productions.

If X is a symbol & 'a' is one of its attribute then we write X.a

Two kinds of attributes for non terminals :-

1) Synthesized attribute
2) Inherited attribute.

Synthesized Attribute :-

A synthesize attribute at node 'N' is defined only in terms of attribute values at the children of 'N' & at 'N' itself.

Inherited attribute :-

An inherited attribute at node 'N' is defined only in terms of attribute values at parent, siblings & itself.

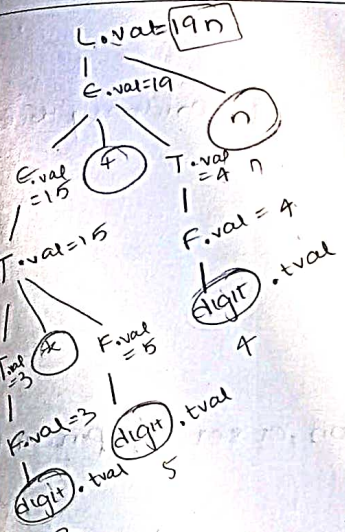
Annotated parse tree :-

A parse tree showing the values of the attributes is called the Annotated parse tree.

(n) → indicates the end of the string

ex:- 3 * 5 + 4n = 19n

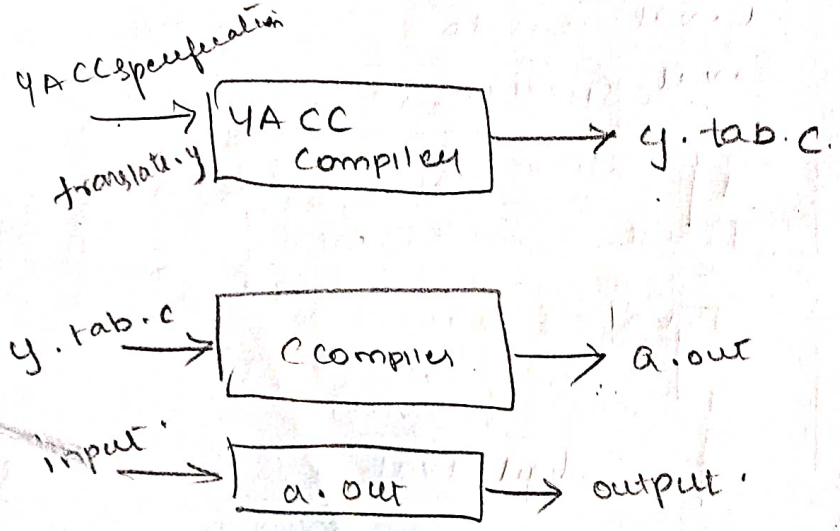
Productions	Semantic rules
L → En	L.val = E.val n
E → E1 + T	E.val = E1.val + T.val
E → T.	E.val = T.val
T → T1 * F.	T.val = T1.val * f.val
T → F	T.val = F.val
F → (E) / digit.	F.val = (E.val) / digit.tval.



Differences between S-attributed & L-attributed.

S-Attributed	L-attributed.
1. Uses only synthesized attribute	1. Uses both synthesized & inherited attribute.
2. Semantic actions are placed at right end of a production A → BC {}	2. Semantic actions are placed at anywhere. ex:- A → {} BC , A → BC{} A → B{} C
3. Attributes are evaluated during bottom up parsing.	3. Attributes are evaluated by translating parse tree depth first.
4. S-attributed grammars is based on LR grammar.	4. L-attributed grammar is based on LL grammar.
	5. These are implemented using

YACC (Yet another Compiler-compiler)



(1) Declaration
(2) Translation rule
(3) Auxiliary procedu

%{ #include<stdio.h> %} ordinary declaration

Translation rules :-

In the part of YACC specification, a set of productions are

<head> → <body1> | <body2> | ... | <bodyn>.

could be written in YACC as

<head> : <body1> {semantic action 1}
| <body2> {semantic action 2}.
|
;

A YACC semantic action is a sequence of 'C' statement. The symbol $$ refers to the attribute-value associated with the non terminal on the left hand side.

$i refers to the value associated with the ith grammar symbol on the right hand side.

E → E+T | T

T → T*F | F

F → (E) | Digit.

```
%%
E : E+T { $$ = $1 + $3 }
  | T
T : T*F { $$ = $1 * $3 }
  | F
F : (E) { $$ = ($2) }
  | digit
%%
```

Auxiliary procedures:-

The third section of YACC specification consists of supporting 'C' routines.

```
%{ #include <ctype.h>
%}
%token DIGIT
%%
expr : expr + term { $$ = $1 + $3 }
     | term.
term : term * factor { $$ = $1 * $3 }
     | factor .
factor : (expr) { $$ = ($2) }
       | digit
%% .
```

```
yylex()
{
  int c;
  c = getchar();
  if(isdigit(c))
  {
    yylval = c - '0';
      return DIGIT
  }
  return c;
}
```

~~YACC~~ to compile :-

```
YACC filename.y
cc y.tab.h
./a.out
```

Type checking :-

Type checking is a methodology to check whether the source language follows both syntactic & semantic conventions of the source language.

Type checking ensures that the errors will be detected & reported.

ICG - Intermediate code generator.

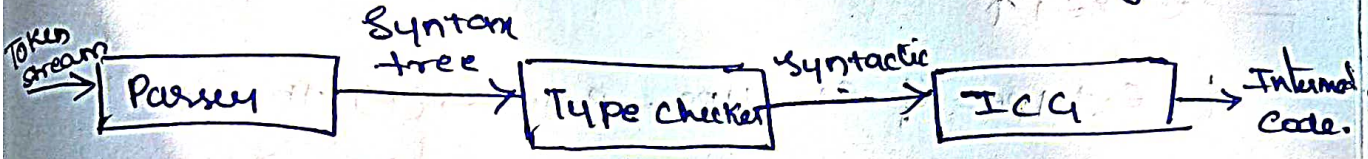


fig :- Position of a type checker.

Type checking can be off 2 types :-

① Static type checking (Compile time)
② Dynamic type checking. (Runtime).

During static & dynamic checking, the following errors can be detected.

① Type-check
② Flow of control checks.
③ Uniqueness checks.
④ Name related checks.

Type checking of expressions :-

$E \rightarrow$ literal { E.type = char }

$E \rightarrow$ num { E.type = integer }

$E \rightarrow$ id. { E.type = lookup(id.entry) }

$E \rightarrow E_1 \bmod E_2$ { E.type = if E_1.type = integer and E_2.type = integer then integer else type-error }

ex:- a[10]

$E \rightarrow E1\,[E2]$.

{ E.type

array(10, integer)

= if $E2$.type = integer and $E1$.type = array(S, t) then t else type-error }.

ex:- $E \rightarrow E1\uparrow$

{ E.type = if E1.type = pointer(t) then t else type-error }.

Type checking of Statements :-

$S \rightarrow id := E$. { S.type = if id.type = E.type then void else type-error }

$S \rightarrow$ if E then S1 { S.type = if E.type = true then S1.type else type error }

$S \rightarrow$ while E do S1 { S.type = if E.type = true then S1.type else type-error }

$S \rightarrow S1; S2$ {S.type = if S1.type = void and S2.type = void then void else type-error }.

***

Type checking of functions :-

$E \rightarrow E1(E2)$ { E.type = if E2.type = t and E1.type = S → t then t else type-error }

Type Conversions:-

Type conversion is the method of converting a variable from its data type to another data type depending on the operation & other operands.

$E \rightarrow num$	$\{E.type = integer\}$
$E \rightarrow num.num$	$\{E.type = float/real\}$
$E \rightarrow id$	$\{E.type = lookup(id.entry)\}$
$E \rightarrow E1\ OP\ E2$	

$E.type =$ if $E1.type = integer$ and $E2.type = integer$ then t
else type-error.

else if $E1.type = integer$ and $E2.type = real$ then t
else type-error.

if $E1.type = real$ and $E2.type = integer$ then real
else if $E1.type = real$ and $E2.type = real$ then real
else type-error.

}

9/11/23

Chomsky hierarchy of languages :-

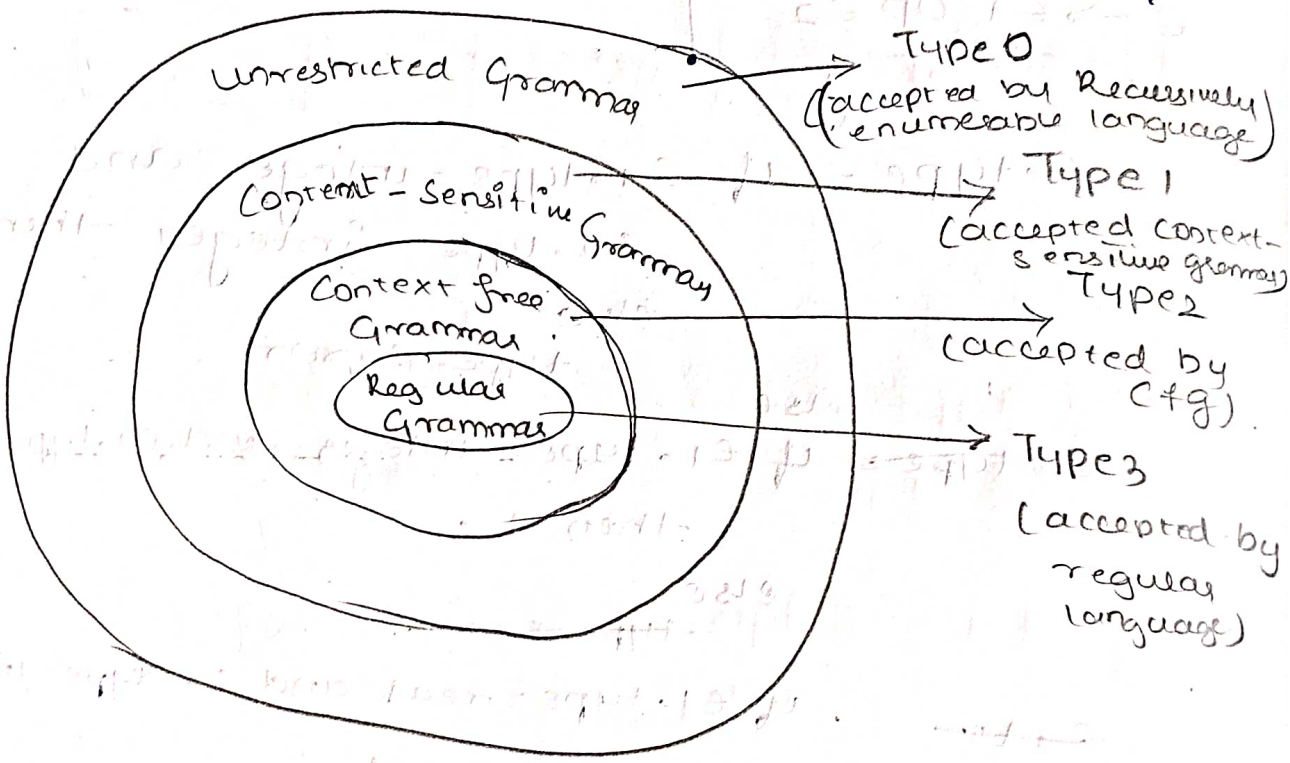
(4) ~~Recursively enumerable language (Type 0)~~

(1) Regular language (Type 3)

(2) Context - free language (Type 2)

(3) Context - sensitive language (Type 1)

(4) Recursively enumerable language (Type 0)



Chomsky hierarchy represents the classes of language that are accepted by diff machines. Therefore, every lang of type 3 is also of type 2, type 1 & type 0. Similarly, type 2 is also of type 1 & type 0. Similarly, type 1 is also of type 0.

Type 3 $\subseteq$ Type 2 $\subseteq$ Type 1 $\subseteq$ Type 0.

Grammar	languages	Recognizing Automata	Production rules	Example
Type-3 Regular grammar	Regular	FSA	$A \to a$ / $A \to aB$	$L=\{a^n / n \geq 0\}$
Type-2 CFG	Context-free	PDA (Push down automata)	$A \to \alpha$	$L=\{a^n b^n / n>0\}$
Type-1 Context sensitive grammar	context-sensitive	LBA (Linear Bounded automata)	$\alpha A \beta \to \alpha \gamma \beta$	$L=\{a^n b^n c^n / n>0\}$
Type-0 unrestricted grammar	Recursively enumerable.	TM (Turing machine)	$\gamma \to \alpha$	$L=\{w / w$ describes a terminating TM$\}$.

* Type 0 Grammar :-

Type 0 grammar is known as unrestricted grammar. There is no restriction on the grammar rules of these type of languages. These languages can be efficiently modeled by turing machine.

* Type 1 Grammar :-

Type 1 grammar is known as context sensitive grammar. It follows the following rules.

1) The context sensitive grammar may have more than one symbol on the LHS of production rules.
2) The number of symbols on LHS must not exceed the no. of symbols RHS.
3) $A \to \epsilon$ is not allowed unless A is a start symbol.

Type 2 Grammar :-

Type 2 grammar is known as Context free grammar. Context free languages are the languages which can be represented by the CFG. ($G = V, T, P, S$)

Type 3 Grammar :-

Type 3 Grammar is known as regular grammar. Regular lang's are those lang's which can be described using regular expressions. These lang's can be modelled by DFA & NFA.

Construct SLR / CLR / LALR for following grammar.

$A \rightarrow id \mid B = D$

$B \rightarrow id$

$D \rightarrow a \mid B$

$A \rightarrow id$
$A \rightarrow B = D$
$B \rightarrow id$
$D \rightarrow a$
$D \rightarrow B$

$A' \rightarrow .A$
$A \rightarrow .id$
$A \rightarrow .B = D$
$B \rightarrow .id$
} I_0

goto (I_0, A)

$A' \rightarrow A.$ } I_1

goto (I_0, id)

$A \rightarrow id.$
$B \rightarrow id.$ } I_2

goto (I_0, B)

$A \rightarrow B. = D$ } I_3

goto (I_3, =)

$A \rightarrow B = .D$
$D \rightarrow .a$
$D \rightarrow .B$
$B \rightarrow .id$ } I_4

goto (I_4, D)

$A \rightarrow B = D.$ } I_5

goto (I_4, a)

$D \rightarrow a \cdot \}$ I_6

goto (I_4, B)

$D \rightarrow B \cdot \}$ I_7

goto (I_4, id)

$B \rightarrow id \cdot \}$ I_8

① $A \rightarrow id$ I_2

② $A \rightarrow B = D$ I_6

③ $B \rightarrow id$ I_2

④ $D \rightarrow a$ I_4

⑤ $D \rightarrow B$ I_7

FOLLOW(A) = { $, } FOLLOW(B) = { =, $ }

FOLLOW(D) = { $ }

FIRST(A) = { id, a }

FIRST(B) = { id }

FIRST(D) = { a, id }

State	id	=	a	$	A	B	D
I_0	S2		S4		1	3	
I_1							
I_2				γ_1			
I_3		S4					
I_4	S8		S6			7	5
I_5				γ_2			
I_6				γ_4			
I_7				γ_5			
I_8		γ_3		γ_3			

14/11/23

quivalence of type expressions :-

(1) Structual Equivalence

(2) Name Equivalence.

) Structual Equivalence :-

ructual Equivalence of type expressions tells that
wo type expressions are are structually equivalent
f & only if they are identical i.e the two type
expression should be of same basic type.

) Name Equivalence :-

Name Equivalence views each type name as a distinct
type. So, two type expressions are name equivalent
ff they are identical.

Overloading of functions & operation :-

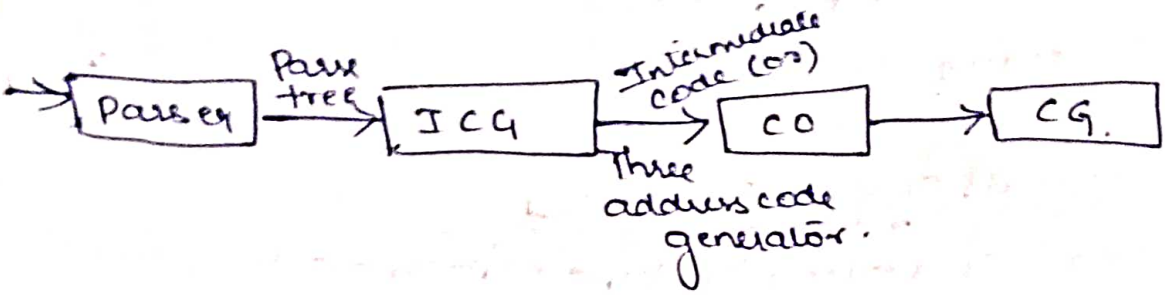
An overloaded symbol has diff meaning depending on
its context. Overloading is resolved when a unique
meaning is determined for each occurrence of a
name.

example :- + operator in java denotes either string
concatenation (or) addition, depending on the type
of its operands.

:- void err() void err(string s)
{ } { }

UNIT-IV

Intermediate Code Generator :-



*** Abstract Syntax Tree :-

~~Abst~~ Syntax Tree is a hierarchical structure of the i/p pgm. In a Syntax Tree the parent nodes are operators & leaf nodes are the symbol of the expression We use the following func's to create the nodes of the syntax tree :-

i) mknode (operator, left, right)

It creates an operator node with label op & two fields containing pointers to left & right.

ii) mkleaf (id, entry)

It creates an identifier node with label id & a field containing entry, a pointer to the symbol table entry for the identifier.

iii) mkleaf (num, value)

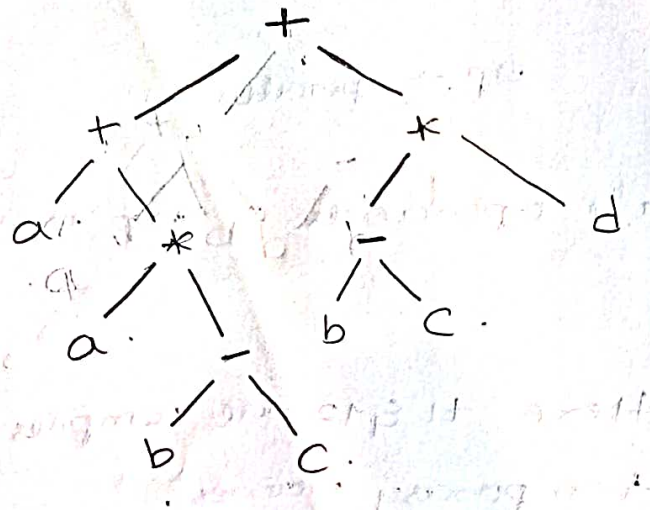
Creates a num node with label num & a field containing val, the value of a number.

ex-

(1) a - 4 + c

p1 = mkleaf (id, a)
p2 = mkleaf (num, 4)
p3 = mknode (-, p1, p2)
p4 = mkleaf (id, c)
p5 = mknode (+, p3, p4)

(5) a+a*(b-c)+(b-c)*d.



p1 = mkleaf(id, b)
p2 = mkleaf(id, c)
p3 = mknode(-, p1, p2)

p4 = mkleaf(id, a)
p5 = mknode(*, p4, p3)
p6 = mkleaf(id, a)
p7 = mknode(+, p6, p5)
p8 = mkleaf(id, b)

p9 = mkleaf(id, c)
p10 = mknode(-, p8, p9)

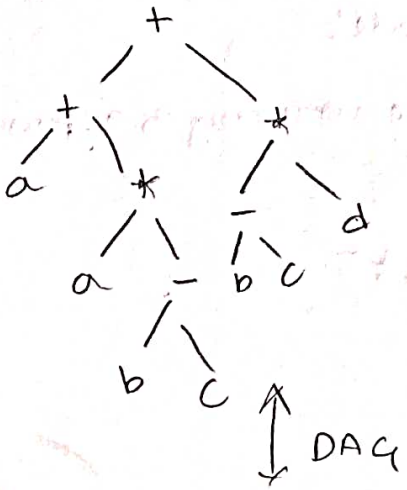
p11 = mkleaf(id, d)

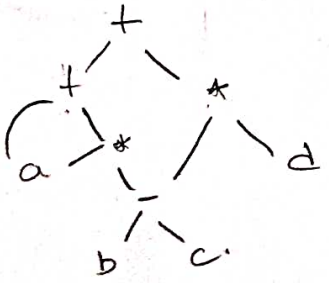
p12 = (*, p10, p11)

p13 = MKnode(+, p7, p12)

V. imp

DAG (Directed Acyclic Graph)





p1 = mkleaf(id, b)
p2 = mkleaf(id, c)
p3 = mknode(-, p1, p2)
p4 = mkleaf(id, a)
p5 = mknode(*, p4, p3)
p6 = mknode(+, p4, p5)
p7 = mkleaf(id, d)
p8 = mknode(*, p3, p7)
p9 = mknode(+, p6, p8)

* Three-address code :-

Three-address code is a sequence of stmts of the gen

form $x = y \, op \, z$ op → operator.

$x = op \, y$

where x, y & z are operands & op is any operator.

ex:- $x + y * z$.

$t1 = y * z$

$t2 = x + t1$

Here t1 & t2 are compiles genera temporary names.

ex:- $a = b * -c + b * -c$.

$t1 = -c$ $t3 = -c$

$t2 = b * t1$ $t4 = b * t3$

$t5 = t2 + t4$. $a = t5$.

Acc dng. Syntax tree

DAG :- $t1 = -c$

$t2 = b * t1$

$t3 = t2 + t2$

$a = t3$.

Implementation of 3 address code :-

Implement the 3 address code using 3 representati

1) Quadruples
2) Triples
3) Indirect triples

Quadruple :-

A Quadruple is a record structure with 4 fields such as operator (op), arg1, arg2, result.

The op field contain an internal code for the operator.

The three address stmt $x = y$ op z. is represented by y in arg1, z in arg2 & x in result.

Triples :- op, arg1, arg2.

To avoid entering temporary name into symbol table refer to a temporary value by the position of the stmt.

Indirect triples :- op arg1, arg2, stmt no

Another representation of 3 address code that has been considered is that testing of pointers to triples rather than using the triples themselves. This implementation is known as indirect triples.

Ex:- Implement the 3addr code for following stmt

$x + y * z$.

$t1 = y * z$

$t2 = x + t1$

(1) Quadruples :-

line no #	op	arg1	arg2	result
1	*	y	z	t1
2	+	x	t1	t2

(2) Triples

(1) $t1 = y * z$

(2) $t2 = x + t1$

Line no #	op	arg1	arg2
1	*	y	z
2	+	x	(1)

(3) Indirect triple

Line no #	Tripleno	op	arg1	arg2
1	101	*	x	z
2	102	+	y	(101)

anything

ex:- $a = b*(-c) + b*(-c)$

(1) $t1 = -c$
(2) $t2 = b * t1$
(3) $t3 = -c$
(4) $t4 = b * t3$
(5) $t5 = t2 + t4$
(6) $a = t5$

(a) Quadruple.

Line no #	op	arg1	arg2	result
1	-	c	null	t1
2	*	b	t1	t2
3	-	c	null	t3
4	*	b	t3	t4
5	+	t2	t4	t5
6	=	t5	-	a

b) Triple

line no #	op	arg1	arg2
1	−	c	
2	*	b	(1)
3	−	c	
4	*	b	(3)
5	+	(2)	(4)
6	=	(5)	

(c) Indirect Triple

line no #	Triple no	op	arg1	arg2
1	101	−	c	
2	102	*	b	(101)
3	103	−	c	
4	104	*	b	(103)
5	105	+	(102)	(104)
6	106	=	(105)	

Ex:- Implement the following using DAG.

$(x+y) * (y+z) + (x+y+z)$.

t1 = x + y.
t2 = y + z
t3 = t1 * t2
t4 = t1 + z.
t5 = t3 + t4.

Quadruple

line no	op	arg1	arg2	result
1.	+	x	y	t1
2	+	y	z	t2
3	*	t1	t2	t3
4	+	t1	z	t4
5	+	t3	t4	t5

Triple

line no #	op	arg1	arg2
1	+	x	y
2	+	y	z
3	*	(1)	(2)
4	+	(1)	z
5	+	(3)	(4)

Indirect triple

line no #	Triple no	op	arg1	arg2
1	101	+	x	y
2	102	+	y	z
3	103	*	(101)	(102)
4	104	+	(101)	z
5	105	+	(103)	(104)

(a) $-(x+y)*(y+z)+(x+y+z)$

$t1 = x+y$

$t2 = -t1$

$t3 = y+z$

$t4 = t2 * t3$

$t5 = t1+z$

$t6 = t4+t5$

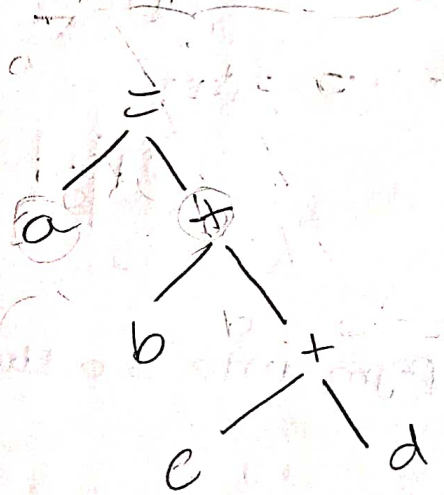
language construct	ICF Intermediate code form	Meaning
Assignment statement	$x = y \text{ op } z$	Binary operation is performed.
assignment statement	$x = \text{op } z$	Unary operation is performed.
copy statement	$x = y$	Here the value of y is assigned to x.
Unconditional jump	goto d.	The control flow goes to the stmt labelled by d.
Conditional jump	if x relop y	If x relational operator is true then it executes the goto L stmt
Array statement	$x = y[i]$	The value at ith index of array y is assigned to x.
Address & Pointer statement	$x = \&y$ $x = *y$	The value of x would be having the address/location of y. The y is a pointer whose value is assigned to x.

Construct the syntax tree, postfix notation & 3 addr code

for $a = b + c + d$

left right root → post

$a = b + c + d$



Write the 3 address code for the following stmt.

if A<B 1 else 0.

① if A<B goto 2. else goto 3. → optional
② $T_1 = 1$
③ $T_2 = 0$.

Write the 3 address code for the following expression

① if A<B and C<D then t = 1 else t=0.

① if A<B goto (2)
② goto(4)
③ if (C<D) goto(5)
④ $t_1 = 0$.
(5) $t_2 = 1$.

② if A<B or C<D then t=1 else t=0

① if A<B goto (3) else goto (2)
② if C<D goto (3) else goto (4).
③ $t_1 = 1$
④ $t_2 = 0$

Convert the following ~~code~~ pgm into 3 address code.

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

if (a<b) then
    n++;
else
    n--;

(1) if (a<b) goto (2) else goto (3)

(2) n = n+1;

(3) n = n-1.

a) Generate the 3 adds code for the following stmts.

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

if ch=1 goto α_1.
if ch=2 goto α_2.

α_1 $T_1 = a+b$
$c = T_1$
goto last.

α_2 $T_2 = a-b$
$c_2 = T_2$
goto last.

last.

Q) Convert the following pgm into 3 addr code.

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

(1) prod=0 } copy stmts.
(2) i=1
(3) $T_1 = 4 * i$ → int contains 4 bytes
(4) $T_2 = a[T_1]$
(5) $T_3 = 4 * i$
(6) $T_4 = b[T_3]$
(7) $T_5 = T_2 * T_4$
(8) $T_6 = Prod + T_5$
(9) $Prod = T_6$
(10) $T_7 = i + 1$
(11) $i = T_7$
(12) if (i<10) goto(3)

Translation of Simple & Control flow statements :-

21/11/23

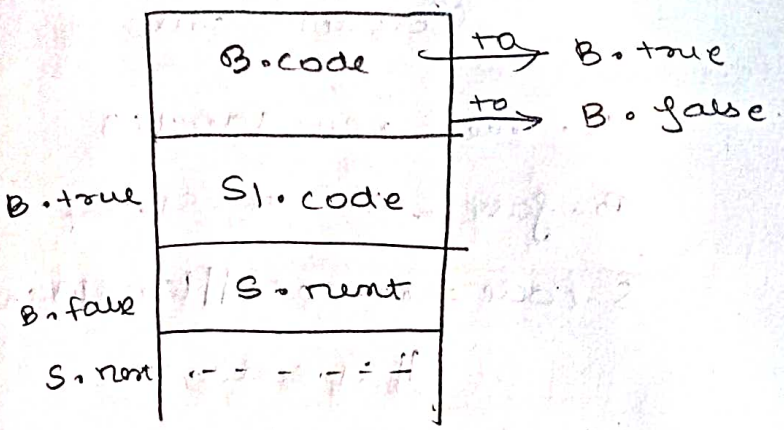
The translation of boolean expressions into 3 address code in the context of stmts such as

$S \rightarrow$ if(B) S_1

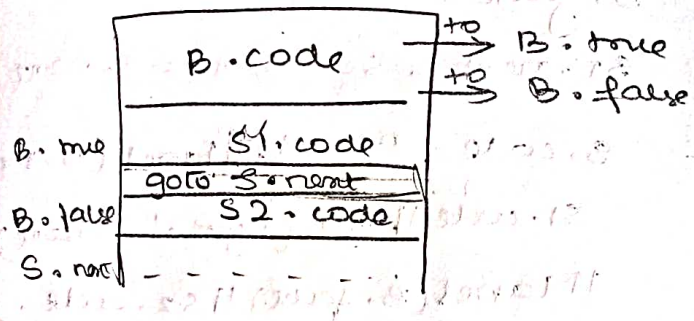
$S \rightarrow$ if(B) S_1 else S_2

$S \rightarrow$ while (B) S_1

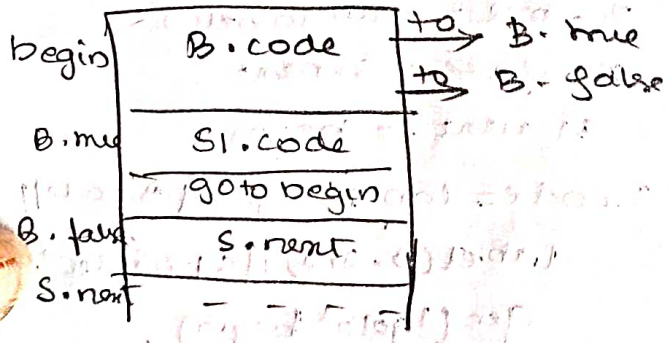
$S \rightarrow$ if (B) S1



$S \rightarrow$ if (B) S1 else S2



$S \rightarrow$ while (B) S1

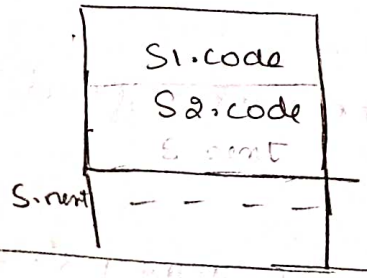
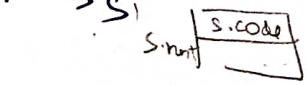


With a boolean expressions B, we associate two labels

i) B.true :- The label to which control flows if B is true.

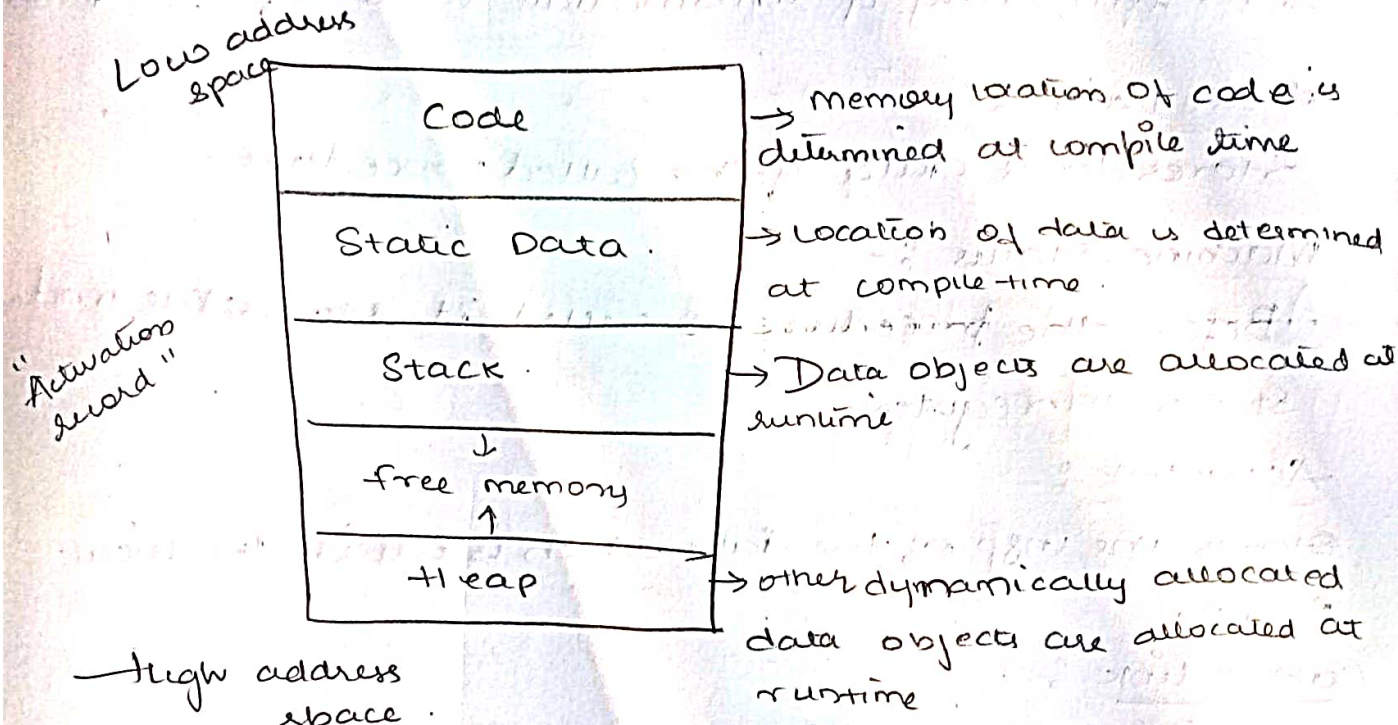
ii) B.false :- The label to which control flows if B is false.

S.next denoting a label for the inst immediately after code for S.

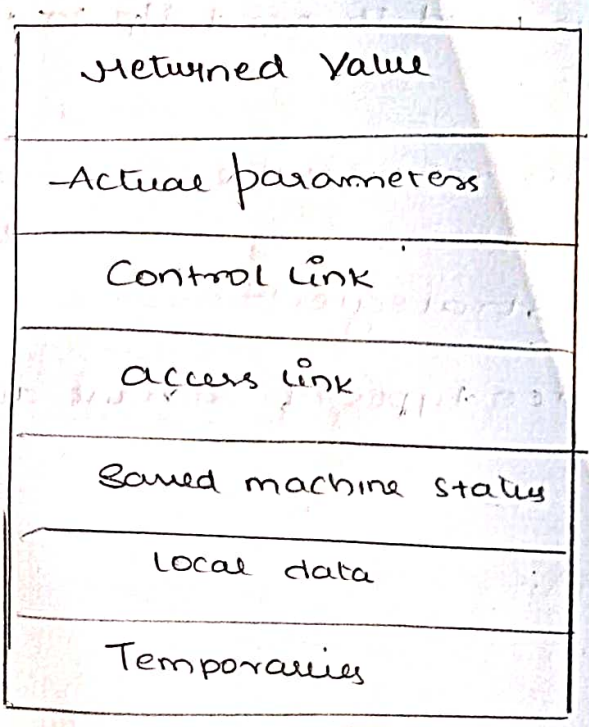
Statement	Semantic rule.
$S \rightarrow if(B)\ S1$	B.true = new label() B.false = S.next = S1.next S.code = B.code || label(B.true) || S1.code.
$S \rightarrow if(B)S1\ else\ S2$	B.true = new label() B.false = new label() S1.next = S2.next = S.next S.code = B.code || label(B.true) || S1.code || gen('goto' S.next) || label(B.false) || S2.code.
$S \rightarrow while(B)\ S1.$	begin = new label() B.true = new label() B.false = S.next S1.next = begin S.code = label(begin) || B.code || label(B.true) || S1.code || gen('goto' begin).
$S \rightarrow S1S2$ 	S.code = S1.code || S2.code
$P \rightarrow S1$ 	S.next = new label() P.code = S.code || label(S.next)
$S \rightarrow assign$	S.code = assign.code.

22/11/23

Storage organization :-



Activation record :-



An activation record is a contiguous block of storage that manages info required by a single execution of procedures.

Temporaries :-

Stores temporary & intermediate values of an expression

Local Data :-

Stores local data of the called procedure.

Machine Status :-

Before the procedure is called it stores the machine status in registers.

Access Link :-

Stores the info of the data which is outside the local scope.

Control Link :-

Stores the address of the activation record of the caller procedure.

Actual parameters :- Stores actual parameters i.e parameters which are used to send i/p to the called procedure.

Returned value :- Stores return value.

24/11/23

Storage allocation Strategies :-

There are mainly three types of storage allocation strategies.

1) static allocation
2) heap "
3) stack "

(1) Static Allocation :- lays out or assign the storage for all the data objects at compile time.

C & C++ use static allocation. The memory will be allocated in a static ~~allo~~ location

Adv:- 1) It is easy to understand.

2) The memory is allocated once only at compile time & remains the same through out the pgm completion.

DisAdv :-

1) Not highly scalable.

2) Static Storage allocation is not very effient.

(2) Heap Allocation

Heap Allocation is used where the stack allocation lacks if we want to retain the values of the local variable after the activation record ends.

LIFO scheme does not work for the allocation & deallocation of the activation record.

C, C++, Python, Java.

Adv:-

1. Heap allocation is the most flexible allocation scheme.
2. Heap allocation is useful where we have data whose size is not fixed & can change during the runtime.

DisAdv

1. Heap allocation is slower as compared to stack allocation
2. there is a chance of memory leaks.

(3) Stack Allocation :-

Stack is commonly known as dynamic allocation. Dynamic allocation means the allocation of memory at runtime.

Stack is a DS that follows the LIFO principle so whenever there is multiple activator record created it will be pushed or popped in the stack as activations begin & ends.

Access to non-local names

1) static — Accesslink, Display.

2) Dynamic — Shallow, Deep.

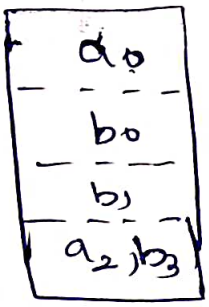
Program:-

```
main
{ int a=0;
  int b=0;
  { int b=1;
    { int a=2;
    } Pf("%d %d\n",a,b);
    { int b=3;
    } Pf("%d %d\n",a,b);
    Pf("%d %d\n",a,b);
  }
  Pf("%d %d\n",a,b);
}
```

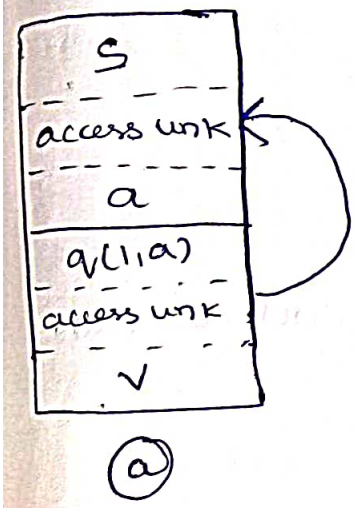
Declarations	Scope
int a=0;	$B_0 - B_2$
int b=0;	$B_0 - B_1$
int b=1;	$B_1 - B_3$
int a=2;	B_2
int b=3;	B_3

O/P:-

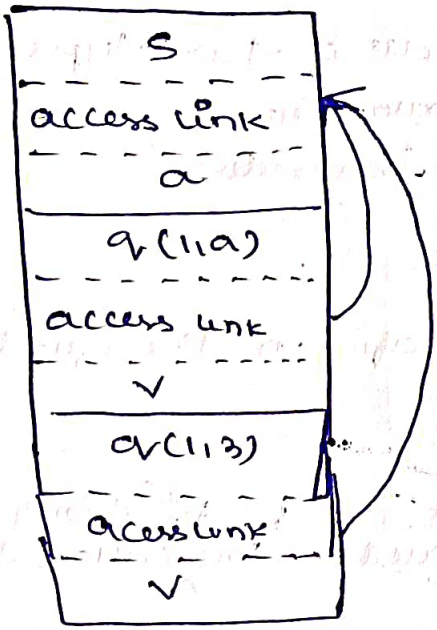
2 1
0 3
0 1
0 0



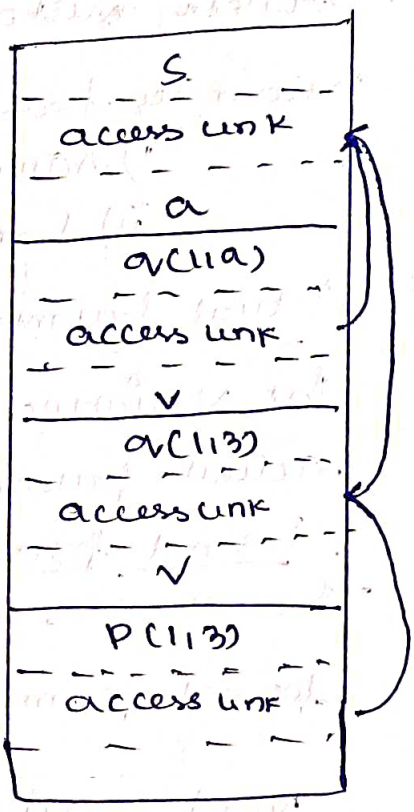
Access link for finding nonlocal data :-

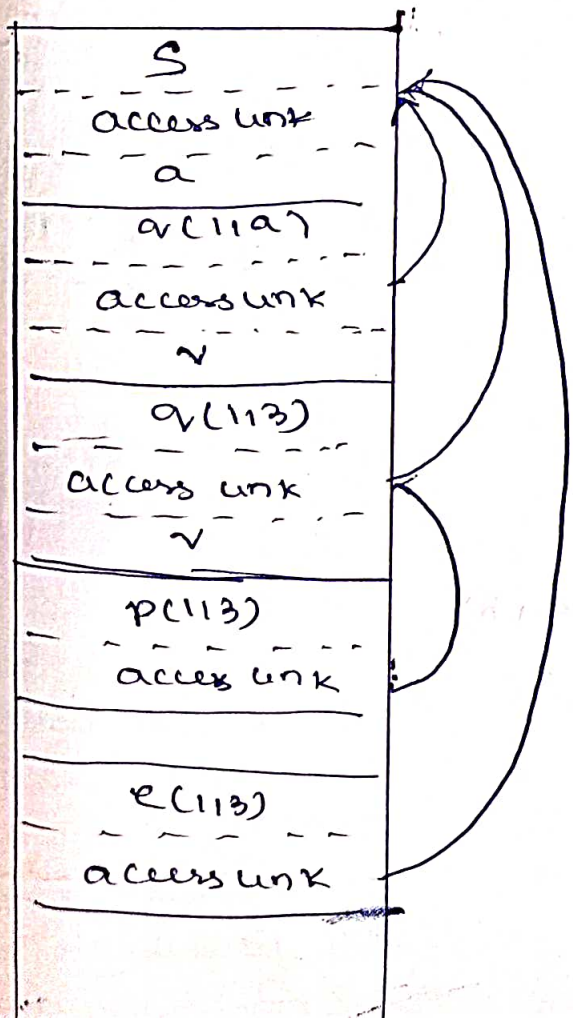


(a)



(b)





6/12/23

Parameter passing techniques

void swap(int x, int y)
{ in

Parameter passing refers to the exchange of information between methods, functions & procedures.

Parameter passing are of two types
i) Actual parameters
ii) Formal parameters.

Actual parameters:-

The variables specified in the function call are called actual parameters.

Formal parameters

The variables declared in the called func is called formal parameters

```
void swap(int x, int y)
{   int t;
    t = x;
    x = y;
    y = t;
}
main()
{ int a=1, b=2;
  swap(a,b);
  pf("a=%d, b=%d", a,b)
}
```

Terminologies in parameter passing :-

l-value :- The l-value of the expression refers to the location in memory. The l-values are placed on the LHS of an assignment operator.

r-value :- The value of an expression is its r-value. They are basically placed on the rhs of the assignment operator.

ex :- a = 1
a → l value
1 → r-value.

Methods for parameter passing :-

Call by value
Call by reference
Call by copy-restore
Call by name.

Call by value :

Caller evaluates the actual parameters. It also passes r-values of actual parameters to formal parameters.

Call by reference :-

Caller evaluates actual parameters. It passes l-values to formal parameters (call by address / call by location).

ex :- swap(&a, &b).

Call by copy-restore :-

It is the hybrid between call by value & call by reference.

Caller evaluates actual parameters & passes r-values to formal parameters.

l-values of actual parameters are determined before the call function.

Call by name :-

Here procedure body is called instead of the procedure names.

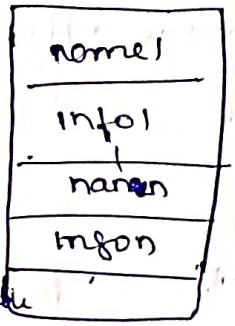
Actual parameters are literally substituted for the formal parameters.

** Symbol table & its implementation :-

<name, information>

allocate
free
lookup
insert
set-attribute
get-attribute

(1) List



(2) Linked List

A compiler need to collect & use information about the names appearing in the source pgm. This info is entered into a data structure called symbol table. Each entry in the symbol table is a pair of the form <name, information>.

ex:- p = i + r * 60 ;

<p, id1>
<=, assignment operator>
<i, id2>
<*, multiplication operator>
<+, addition operator>
<r, id3>
<60, constant>.

Operations of a Symbol table :-

allocate :- To allocate a new empty symbol table.

free :- to remove all entries and free storage of symbol table.

lookup :- to search for a name & return pointer to its entry.

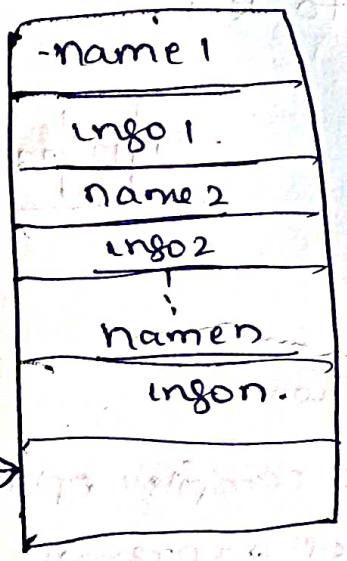
insert :- to insert a name in a symbol table & return to its entry.

set-attribute :- to associate an attribute with a given entry

get-attribute :- to get an attribute associated with a given entry.

Implementation of Symbol Table :-

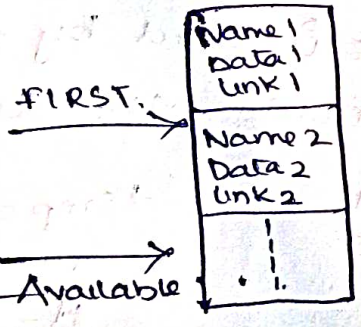
1) List :- In this method an array is used to store names & associated information.



2) Linked List :- Here a link field is added to each record.

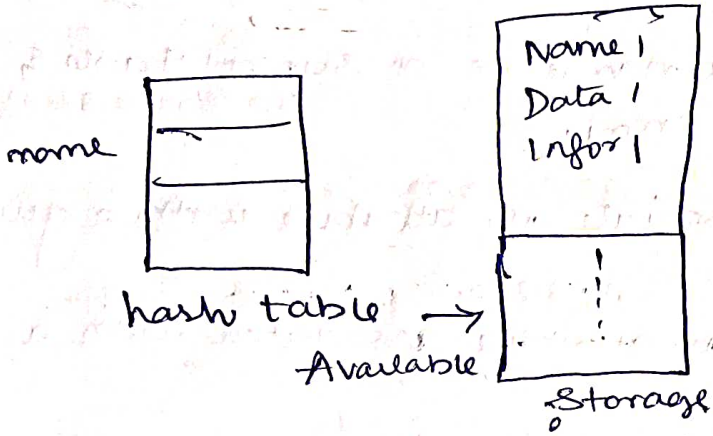
Searching of names is done in order pointed by the link of the link field.

FIRST gives the position of the first record on the linked list.



3) BST :- A more efficient approach to symbol table organization is to add two link fields to each record. These fields to link the records into a binary search tree.

4) Hash table :-



The hash table consists of k words numbered from 0, 1, ..., k-1. These words are pointers into the storage table. To determine whether name is in the symbol table we apply a hash func^c(b) such that h(name) is an integer b/w 0 to k-1.

Code optimization techniques :-

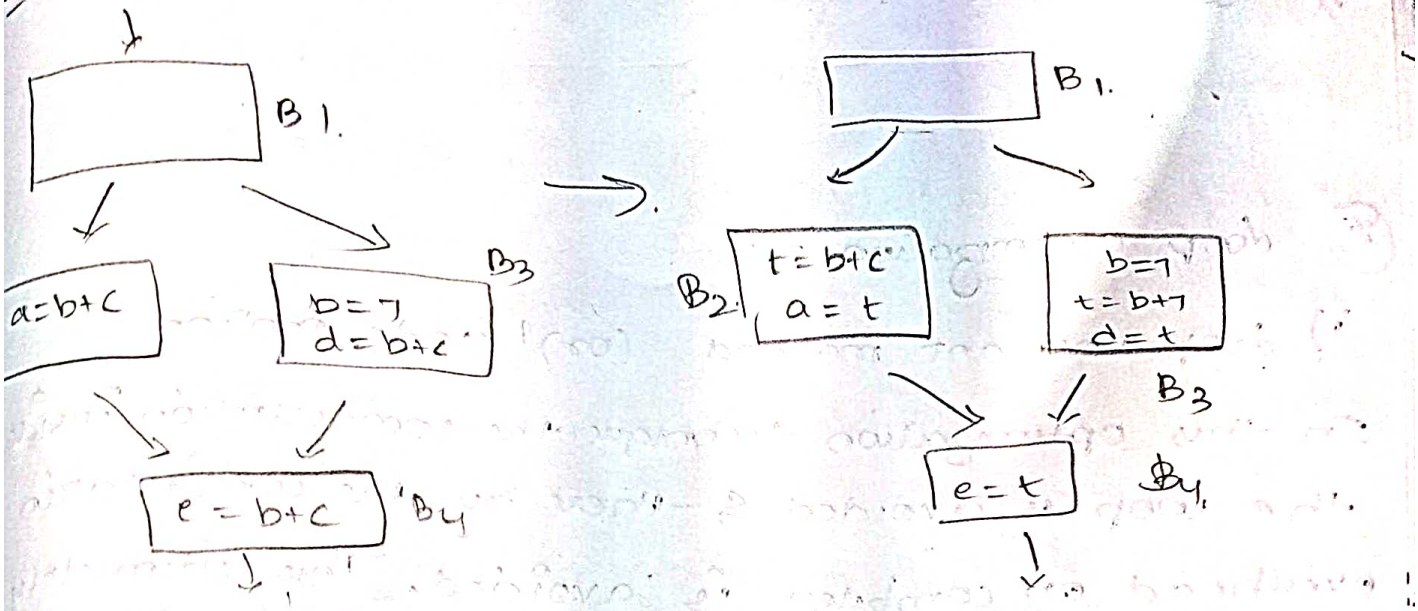
(1) Local optimization techniques :-

i) Common sub expression elimination :-

Common subexpression elimination is a compiler optimization that searches for instances of identical expressions & analyzes whether it is word by replacing them with a single variable holding the compute value.

a = b*c+g
d = b*c*e.

temp = b*c
a = temp + g
d = temp * e.



) Copy propagation :-

ssignment of the form u=v is called copy statement

(Same example as above).

i) Deadcode elimination :-

A variable is alive at a point in a program if it's value can be used subsequently. otherwise it is dead at the point.

iv) Reduction in Strength :-

The transformation of replacing an expensive operation such as multiplication by cheaper one as addition is known as strength reduction.

```
for i=1 to 10 do
    .
    .
    .
    x=i*5
    .
    .
    .
end.
```

=>

```
temp=5;
for i=1 to 10 do
    x=temp
    .
    .
    .
    temp=temp+5
    .
    .
    .
end.
```

v) Code motion.

(ii) Loop optimization :-

i) loop invariant method (or) code motion :-

In this optimization technique the computation inside the loop is avoided & thereby the computation overhead on compiler is avoided. This ultimately optimizes code generation.

```
for i=0 to 10 do begin
    k = i+a/b;
    sop(k);
end;
```

=>

```
t = a/b.
for i=0 to 10 do begin
    k = i+t
end.
```

```
while(i<=limit-2)  => t=limit-2
                      while(i<=t)
```

ii) loop fusion :-

In loop fusion method several loops are merged into one loop.

ex:-

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

=>

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

iii) loop unrolling :-

In this method the no. of jumps & tests can be reduced by writing the code 2 times

Eg:
```
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++;
```

Basic Block :-

A basic block contains a sequence of stmts. The flow of control enters at the beginning of the stmt & leave at the end without any halt.

The following sequence of three address stmts form a basic block.

```
t1 = x*x
t2 = x*y
t3 = 2*t2
t4 = t1+t3
t5 = y*y
t6 = t4+t5
```

Basic Block Construction :-

Algorithm :-

partition into basic blocks.

Input :- It contains the sequence of 3 address stmts

Output :- It contains a list of basic block with each three address stmts in exactly one block.

Method :- First identify the leader in a code.

The rules for finding leaders are as follows :-

i) The first stmt is a leader.

ii) Statement 'L' is a leader if there is an conditional (or) unconditional goto stmts.

iii) Instruction 'L' is a leader if it immediate follows a goto or conditional goto stmts.

For each leader, its basic block consists of the leader & all stmts in 3 address code. It doesn't include the next leader or end of the pgm.

```
begin
prod=0;
 i=1;
do begin
prod:=prod+a[I]*b[I];
    i=i+1;
    end
    while i<=10
        end
```

```
prod=0
 i=1
T1=4*i
T2=a[i]
T3=4*i
T4=b[T3]
T5=T2*T4
T6=Prod*T5
Prod=T6
 T7=i+1
 i=T1
if (i<=10) goto 3.
```

8/12/23

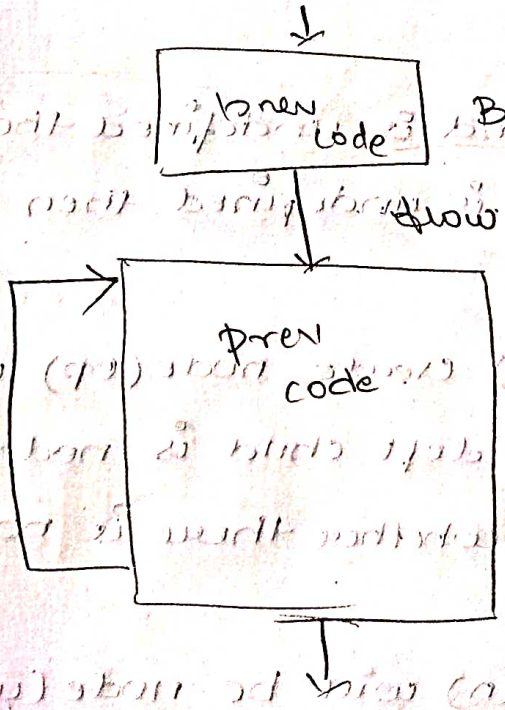
B1
1) prod=0
2) i=1

B2
3) $t_1 = 4 * i$
4) $t_2 = a[t_1]$
5) $t_3 = 4 * i$
6) $t_4 = b[t_3]$
7) $t_5 = t_2 * t_4$
8) $t_6 = Prod + t_5$
9) $Prod = t_6$
10) $t_7 = i + 1$
11) $i = t_7$
12) if $i <= 10$ goto (3)

Flow Graph :- Flow graph is a directed graph. It contains the flow of control info for the set of basic block.

Once an intermediate code prgm is partitioned into basic blocks we represent the flow of control btw them by a flow graph. The nodes of the flow graph are the basic blocks.

If there is an edge from block B to block C iff only it is possible for the instruction in block C to immediately follow the last inst in Block B. Therefore 'B' is the predecessor of 'C' & 'C' is a successor of 'B'.



DAG representation for Basic Block :-

A DAG for basic block is a directed acyclic graph with the following labels on nodes.

1) The leafs of the graph are labelled by a unique identifier & that identifier can be variables names or constants.

2) Interal nodes of the graph is labelled by an

3) DAG provides a good way to determine the common sub expression.

Algorithm for construction of DAG :-

Input :- It contains a basic block.

Output :- It contains the following information

→ Each node contains a label. for leafs the label is an identifier.

→ Each node contains a list of attached identifiers to hold the computed values.

Case (i) $x = y \text{ op } z$

Case (ii) $x = \text{op } y$

Case (iii) $x := y$

Method :-

Step1 :- If y operand is undefined then create node(y)

If z operand is undefined then for case (i).

Create node(z)

Step 2 :- for case (i) create node(op) whose right child is node(z) & left child is node(y).

for case (ii) check whether there is node (op) with one child node (y).

for case (iii) node (n) will be node (y)

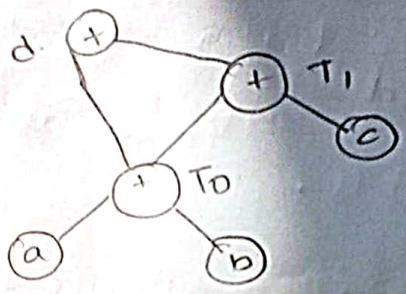
Construct DAG for the following expressions :-

$T_0 = a + b$

$T_1 = T_0 + c$

$d = T_0 + T_1$

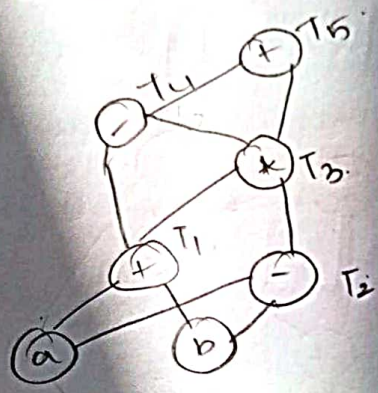
d.



e) $T_1 = a + b$ $\quad T_4 = T_1 - T_3$

$T_2 = a - b$ $\quad T_5 = T_4 + T_3$

$T_3 = T_1 * T_2$



Write in steps.

Construct DAG for the following 3 address stmt :-

(1) $S_1 = 4 * i$

(2) $S_2 = a[S_1]$.

(3) $S_3 = 4 * i$

(4) $S_4 = b[S_3]$

(5) $S_5 = S_2 * S_4$

(6) $S_6 = prod + S_5$

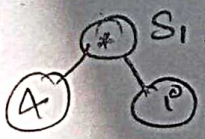
(7) $prod = S_6$

(8) $S_7 = i + 1$

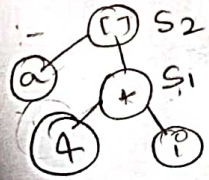
(9) $i = S_7$.

(10) if $i <= 20$ goto(1)

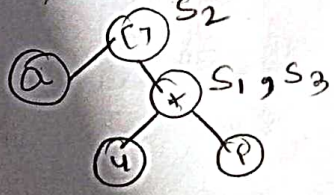
Soln Step1 :-



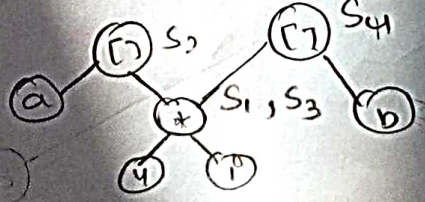
Step2 :-



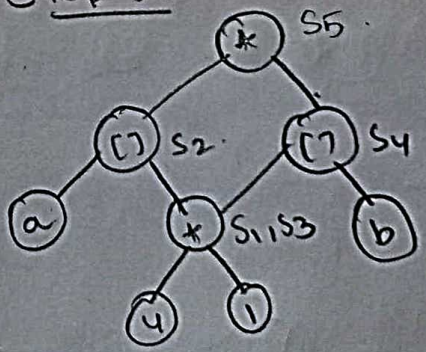
Step3



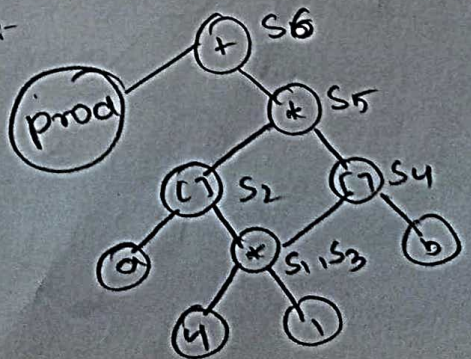
Step4



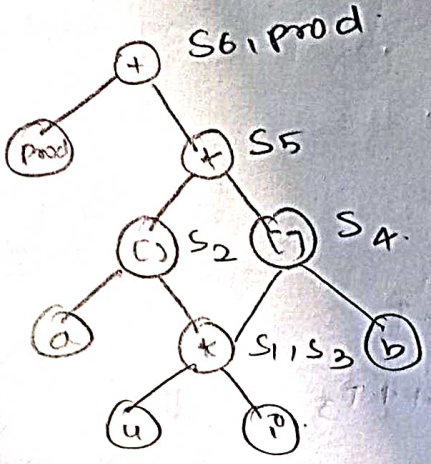
Step5 :-



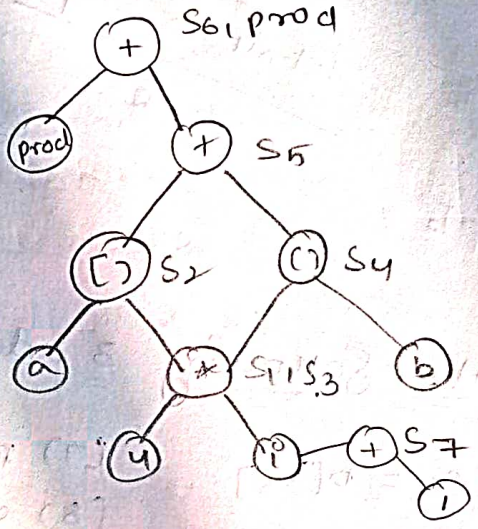
Step6 :-



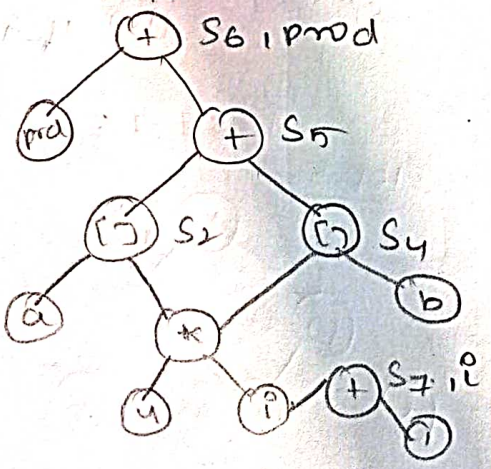
Step 7:- prod = S_6



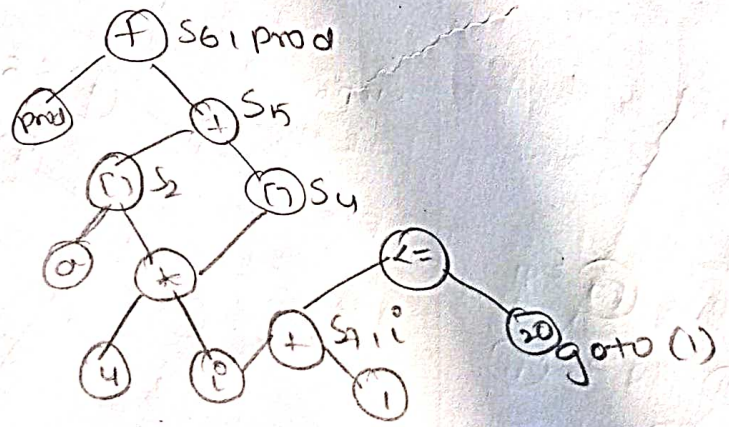
Step 8:-



Step 9:- i = S_7.



Step 10:-



UNIT-5

12/12/23

Code Generation Algorithm

It Generates target code for a sequence of three address stmts.

for each operator in a stmt - there is a corresponding target lang operator.

Register & Address Descriptors

Register Descriptors

It keeps track of what is currently in each register

Initially all the registers are empty.

Address Descriptors.

It keeps track of the location where the current value of the name can be found.

The location may be a register or memory address.

Algorithm :-

for a 3 address instruction $x = y + z$ do the following:

1. Use getreg($x = y + z$) to select registers for x, y, & z. call these as R_x, R_y, R_z.
2. If y is not in R_y then issue an instruction LD Ry y' (Load Ry, y'), where y' is one of the memory location for y.
3. Similarly if z is not in R_z issue an instruction LD R_z, z' where z' is a memory location for z.
4. Issue the instruction at R_x, R_y, R_z

$t = a - b$

$u = a - c$

$v = t + u$

$a = d$

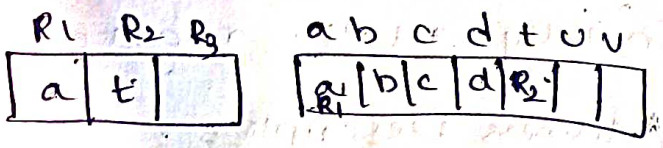
$d = v + u$

Here t, u, v are the temporary variables. a, b, c, d are the normal variables.

Step 1 :- Initially all registers are empty.

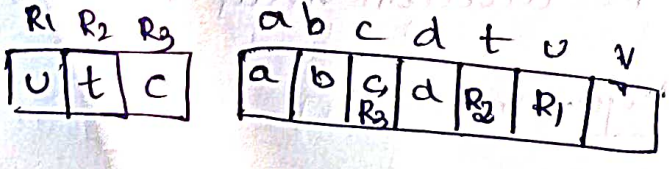
Step 2 :- $t = a - b$

```
LD R1, a
LD R2, b
SUB R2 R1, R2
```



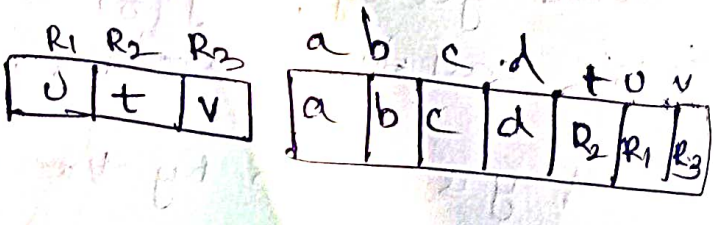
Step 3 :- $u = a - c$.

```
LD R3, c.
SUB R1 R1, R3.
```

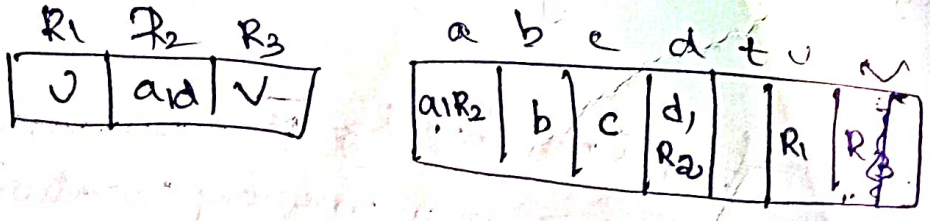


Step 4 :- $v = t + u$

```
ADD R3 R2, R1
```



Step 5 :- $a = d$.



Step 6 :- $d = v + u$

```
Add R1 R3, R1
ST a, R2
ST d, R1
```

8) perform code generation algorithm for the given expression :-

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

first convert into three address code.

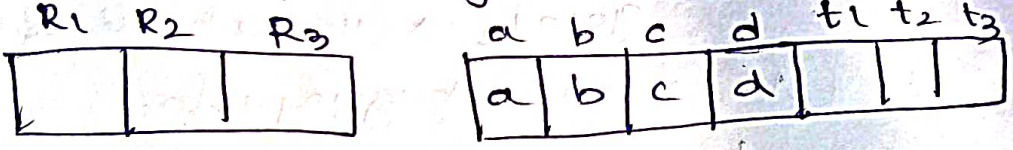
$t_1 = a - b$

$t_2 = a - c$.

$t_3 = t_1 + t_2$

$d = t_3 + t_2$.

step 1:- Initially all registers are empty

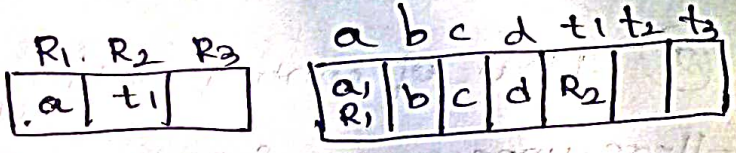


step 2:- $t_1 = a - b$.

LD R1, a

LD R2, b.

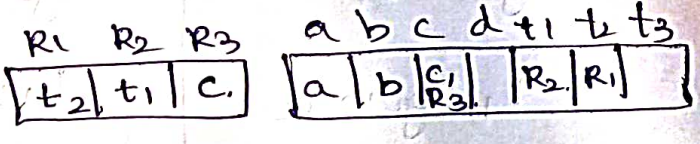
SUB R2, R1, R2



step 3:- $t_2 = a - c$.

LD R3, c.

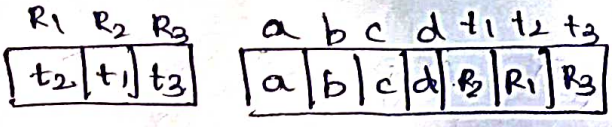
SUB R1 R1, R3



step 4:- $t_3 = t_1 + t_2$.

~~ADD R3 t1, t2~~

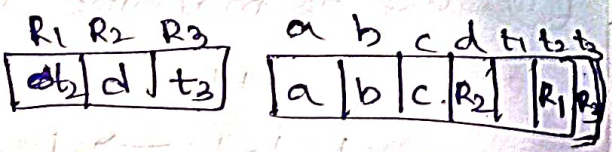
ADD R3 R2, R1



step 5:- $d = t_3 + t_2$.

~~LD R2 d~~

ADD R2 R3, R1.



ST. d, R2

14/12/23

Peephole Optimization Techniques :-

It replaces short sequence of target instructions by a shorter or faster sequence.

(1) Redundant instruction elimination :-

(a) Eliminating redundant load & Store

ex:- LD R0, a } eliminate one
ST a, R0

(b) Eliminating unreachable code.

ex:- if debug == 1 goto L_1
goto L_2 × (unreachable code)
L_1 - print debugging info.
L_2.

(2) Flow of control optimization :-

Here, unneccessary jumps can be avoided.

goto L_1	goto L_2.
L_1 : goto L_2	L_1 : goto L_2
L_2 : a = b	L_2 : a = b.

(3) Algebraic ~~express~~ simplifications :-

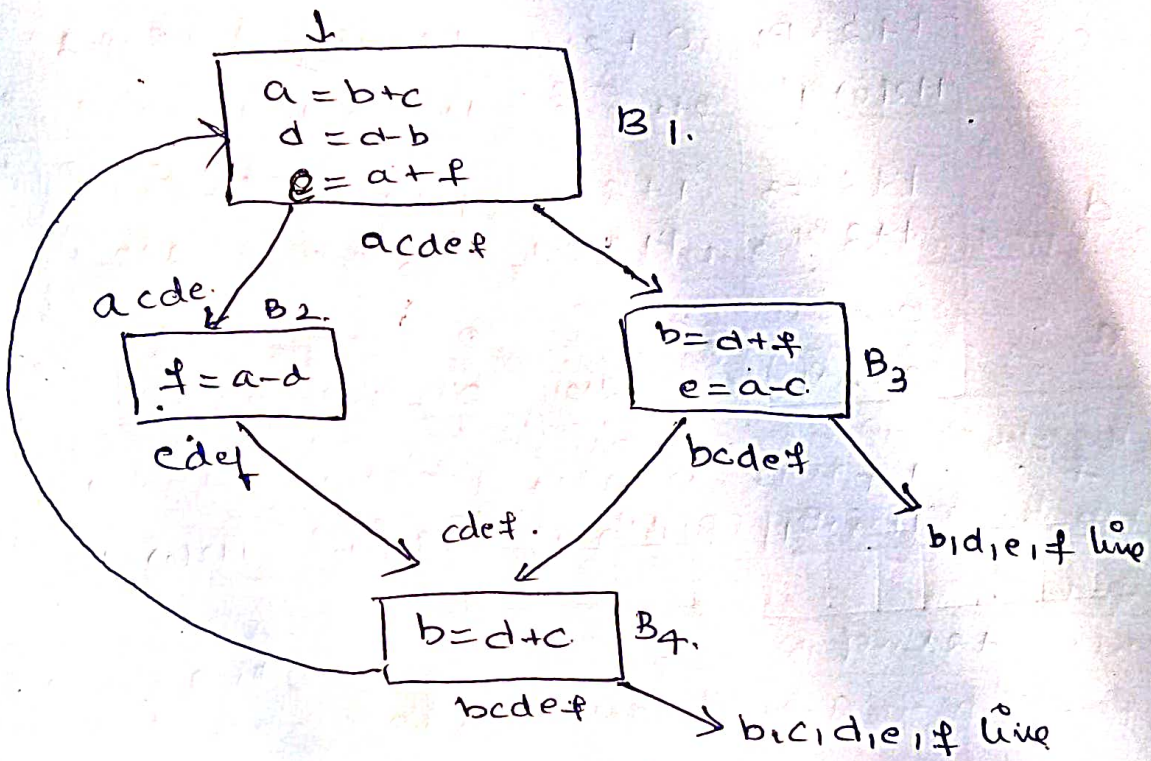
$x = x + 0$ $x = x * 1$

(4) Use of machine idioms,

$i = i + 1 \rightarrow$ Inc i
$i = i - 1 \rightarrow$ Dec i.

Register Allocation & assignment :-

1. Global register allocation
2. Usage count
3. Register assignment for outer loops
4. Register allocation by graph coloring



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

loop = L Register = x.

use(x, B) is no. of times x used & not preceeded by an assignment to x in same block.

& live(x, B) is 1 if 'x' is live on exit, otherwise it is zero.

	B_1	B_2	B_3	B_4	Total
a	$0+2*1=2$	$1+2*0=1$	$1+2*0=1$	$0+2*0=0$	4
b	$2+2*0=2$	$0+2*0=0$	$0+2*1=2$	$0+2*1=2$	6
c	$1+2*1=2$ $1+2(0)=1$	$10+2*1=2$ $0+2*0=0$	$1+2*1=3$ $1+2*0=1$	$1+2*1=3$ $1+2*0=1$	11 3
d	$1+2*1=3$ $1+2*1=3$	$1+2*1=3$ $1+2*0=1$	$1+2*1=3$ $1+2*0=1$	$1+2*1=3$ $1+2*0=1$	12 6
e	$0+2*1=2$ $0+2*1=2$	$0+2*1=2$ $0+2*0=0$	$0+2*1=2$ $0+2*1=2$	$0+2*1=2$ $0+2*0=0$	8 4
f	$1+2*1=3$ $1+2*0=1$	$0+2*1=2$ $1+2(0)=1$	$1+2*1=3$ $1+2(0)=1$	$0+2*1=2$ $1+2(0)=1$	10 4

$wx(n,B)+2*live(n,B)$

$a = b+c$ — a → defined variables | b, c → used variables..

d, c, f, e, b, a

$R_1, R_2, R_3, R_4, R_5, R_6$

du chains - definition used chains.

R_1	R_2	R_3	R_4	R_5	R_6
d	c	f	e	b	a

a - 4
b - 6
c - 3
d - 6
e - 4
f - 4

b, d | a, e, f | c
6 | 4 | 3

R_1 R_2 R_3

Global register Allocation :- To save some of these stores & corresponding loads, we might arrange to assign registers to frequently used variables & keep this registers consistent across block boundaries.

15/12/23

③ Register assignment for outer loop:-

If an outer loop L_1 contains an inner loop L_2 the names allocated registers in L_2 need not be allocated in $L_1 - L_2$. If 'x' is allocated to register in L_2 then load x on entrance of L_2 & store 'x' on exit from L_2.

④ Register allocation by graph coloring :-

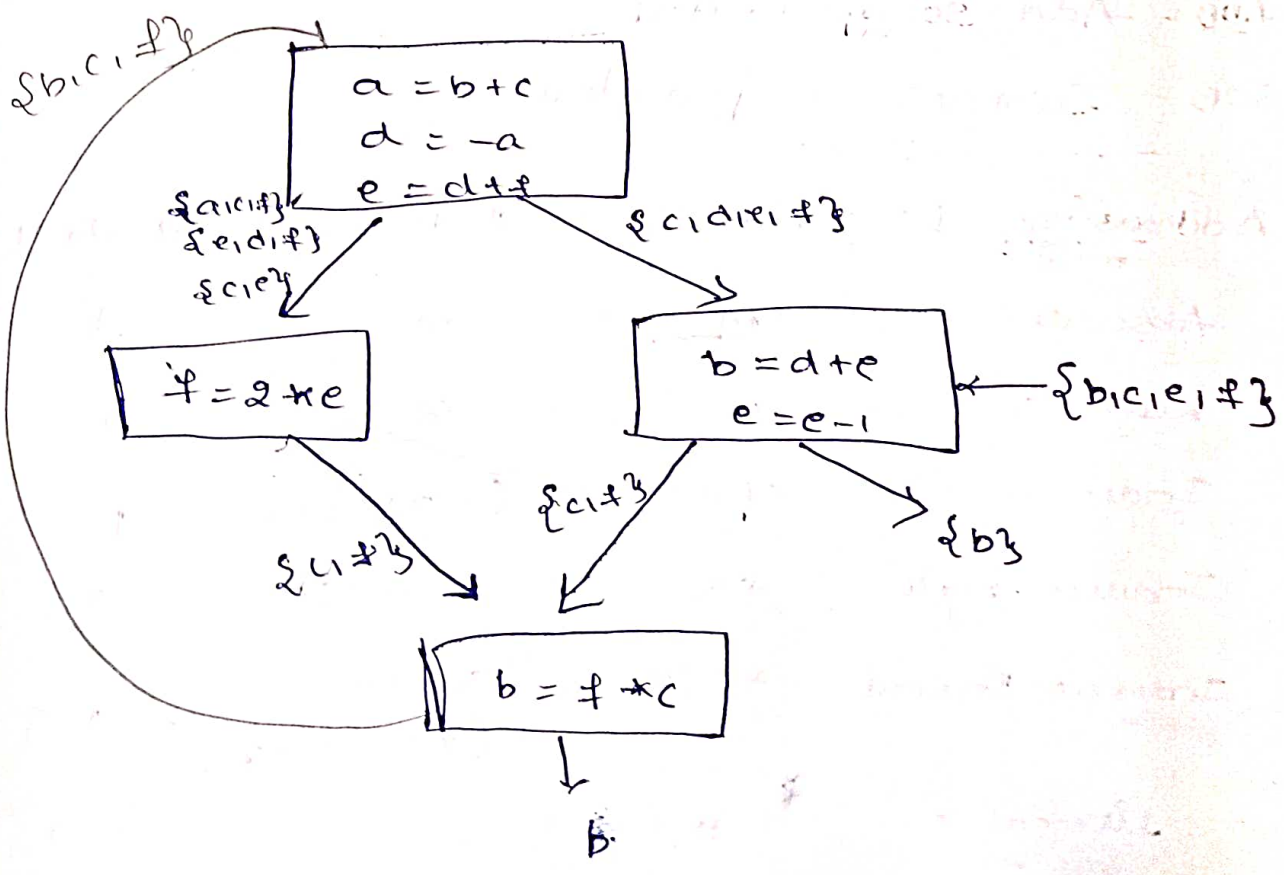
When a register is needed for computation but all registers are in use the contents of one register must be stored into memory location.

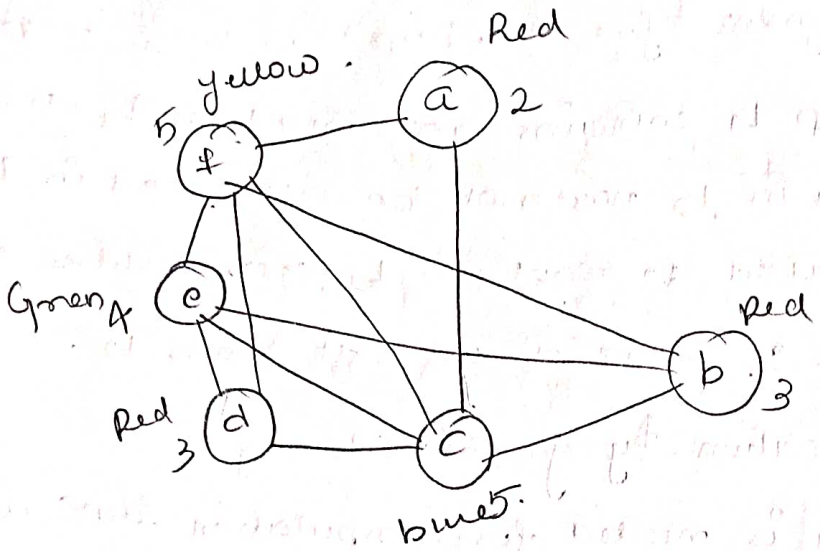
Here 2 passes are used :-

1) Target machine instructions are selected.

2) A register interference graph is constructed, nodes are symbolic registers & h connects ~~2~~ nodes.

A graph is said to be colored if each node has been assigned a color in such a way that no two adj nodes have the same color.





a = 2
b = 3
c = 5
d = 3
e = 4
f = 5

c, f = 5
e = 4
b, d = 3
a = 2

Introduction to assembly language instructions :-

Address in the target code :

There are n-general purpose registers R0, R1, ... Rn-1.

The two address instruction of the form op source destination

Where op is any operator code, source & dest are data fields.

MOV - Moves from src to dest.

ADD - Add source to dest

SUB - Subtracts src from dest.

Addressing mode	form	Address	Added cost.
Absolute	m	m	1
Register	R	R	0
Indexed	C(R)	(+contents(R))	1
Indirect-register	*R	Content(R)	0
Indirect Indexed	*C(R)	Contents (C+contents(R))	1
literal	#C	C	1

Instruction cost = 1 + cost of the src & des Addressing mode.

Consider the following code sequence :-

1) MOV B,R0
ADD C,R0.
MOV R0,A.

2) MOV B,A
ADD C,A.

3) MOV a,R0
ADD b,R0
MOV d,R1
ADD C,R1
SUB e,R1.
SUB R1,R0
MOV R0,x.

Calculate the cost of the above instructions :-

(1) MOV B,R0 = 1+1+0=2
ADD C,R0 = 1+1+0=2
MOV R0,A = 1+0+1=2
6.

(2) MOV B,A = 1+1+1=3
ADD C,A. = 1+1+1=3
6.

(3) MOV a,R0 = 1+1+0=2
ADD b,R0 = 1+1+0=2
MOV d,R1 = 1+1+0=2
ADD C,R1 = 1+1+0=2
SUB e,R1 = 1+1+0=2
SUB R1,R0 = 1+0+0=1
MOV R0,x = 1+0+1=2

(4) MOV R0,R1 = 1+0+0 = 1

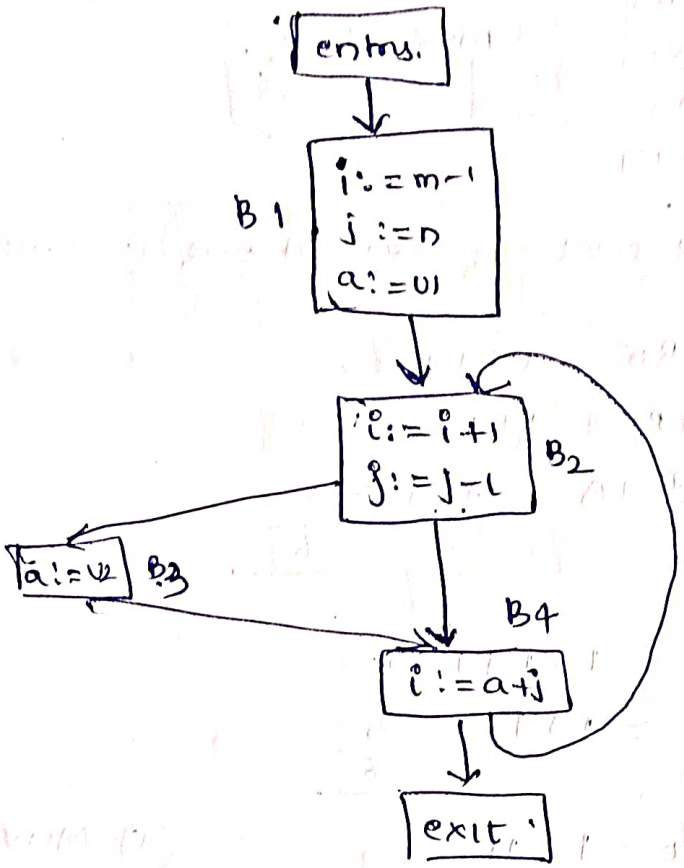
(5) MOV R0,M
= 1+0+1=2
ADD #1,R0
= 1+1+0=2
SUB 2(R0),48(R1)
1+2(1)+

Live Variable Analysis

→ The variable $OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$

$IN[B] = USE[B] \cup (OUT[B] - DEF[B])$

$IN[B] = \phi$, for all B (initialization only)



Step 1:-

$OUT[B4] = IN[B2] = \phi$

$IN[B4] = use[B4] \cup [\phi - i]$

$IN[B4] = \{a, j\}$

Step 2:- $OUT[B3] = IN[B4] = \{a, j\}$

$IN[B3] = USE[B3] \cup (OUT[B3] - DEF[B3])$

$IN[B3] = \{u_2\} \cup [\{a, j\} - \{a\}]$

$IN[B3] = \{u_2, j\}$

ep3:- $OUT[B_2] = IN[B_3] \cup IN[B_4]$

$= \{v_2, j\} \cup \{a, j\} = \{v_2, a, j\}$

$IN[B_2] = use[B_2] \cup [OUT[B_2] - DEF[B_2]]$.

$= (i, j) \cup \{\{v_2, a, j\} - \phi\}$.

$IN[B_2]. = \{i, j, a, v_2\}$

step4:- $OUT[B_1]. = IN[B_2] = \{i, j, a, v_2\}$.

$IN[B_1] = USE[B_1] \cup (OUT[B_1] - DEF[B_1])$

$= \{m, n, v_1\} \cup (\{i, j, a, v_2\} - \{i, j, a\})$

$= \{m, n, v_1\} \cup (\{v_2\})$

$= \{m, n, v_1, v_2\}$.

Step 5:- $OUT[B_4] = IN[B_2] = \{a, i, j, v_2\}$.

$IN[B_4] = use[B_4] \cup [OUT[B_4] - DEF[B_4])$

$= \{a, j\} \cup [\{a, i, j, v_2\} - \{i\}]$

$= \{a, j, v_2\}$.

Step 6:- $OUT[B_3] = IN[B_4] = \{a, j, v_2\}$

$IN[B_3] = USE[B_3] \cup (OUT[B_3] - DEF[B_3])$

$IN[B_3] = \{v_2\} \cup (\{a, j, v_2\} - \{a\})$

$= \{j, v_2\}$.

Step7:- $OUT[B_2] = IN[B_3] \cup IN[B_4]$

$= \{j, v_2\} \cup \{a, j, v_2\}$

$= \{a, j, v_2\}$.

$IN[B_2] = USE[B_2] \cup [OUT[B_2] - DEF[B_2]]$

$= (i, j) \cup [\{a, j, v_2\} - \{\phi\}]$

$= \{i, j, a, v_2\}$

Data flow Analysis

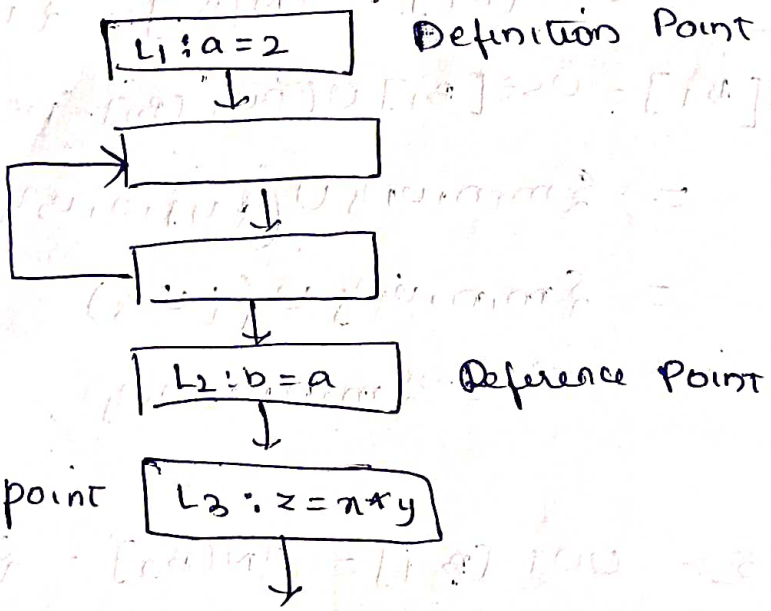
Basic Terminologies :-

Definition point - a point in a pgm containing some definition.

Reference point - a point in a pgm containing a reference to a data item

Evaluation point :- a point in a pgm containing evaluation of expression

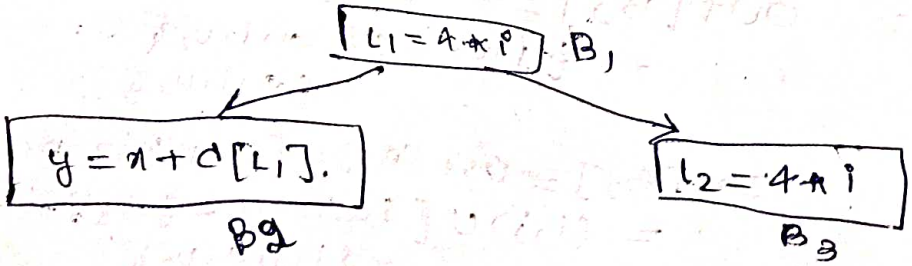
Dataflow Analysis a technique used to analyze how data flows through a pgm.



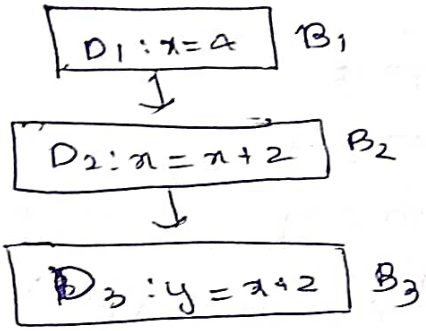
Data flow Properties :-

1) Available Expression
2) Reaching Definitions
3) Live variable Analysis
4) Busy expression

Available Expression :- An expression is said to be available at pgm point x, if along path it reaches to x. An expression is available at its evaluation point.

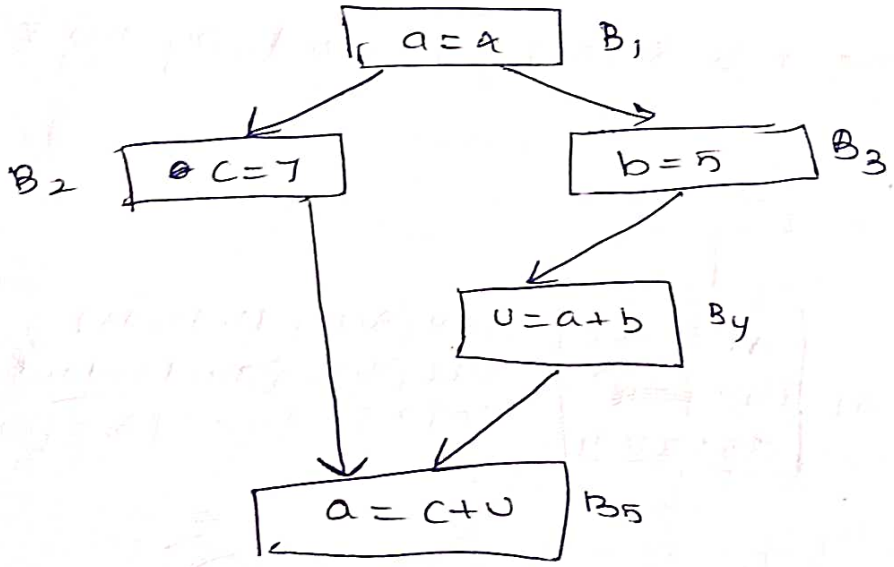


Reaching Definition :- A definition D reaches a point x if there is path from D to x in which D is not killed, i.e not defined.



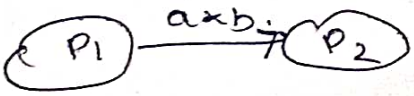
D_1 is reaching definition for B_2 but not for B_3 since it is killed by D_2.

Live Variable :- A var is said to be live at some point p if from p to end the variable is used before it is redefined else it becomes dead.



a is live at block B_1, B_3, B_4 but killed at B_5

Busy Expression :- An expression is busy along a path if its evaluation exists along that path & none of its operand definition exists before its evaluation along the path.



Data flow Equation :-

Input :- kill(B) and gen(B) for every basic block B

Output :- in(B) and out(B) for every basic block B.

For each B repeat

out(B) := gen(B)

while changes to any out(B) occur repeat -

$in(B) := \bigcup_{B' \in pred(B)} out(B')$

$out(B) := gen(B) \cup (in(B) - kill(B))$

where

Out[B] is the information at the end of the statement B.

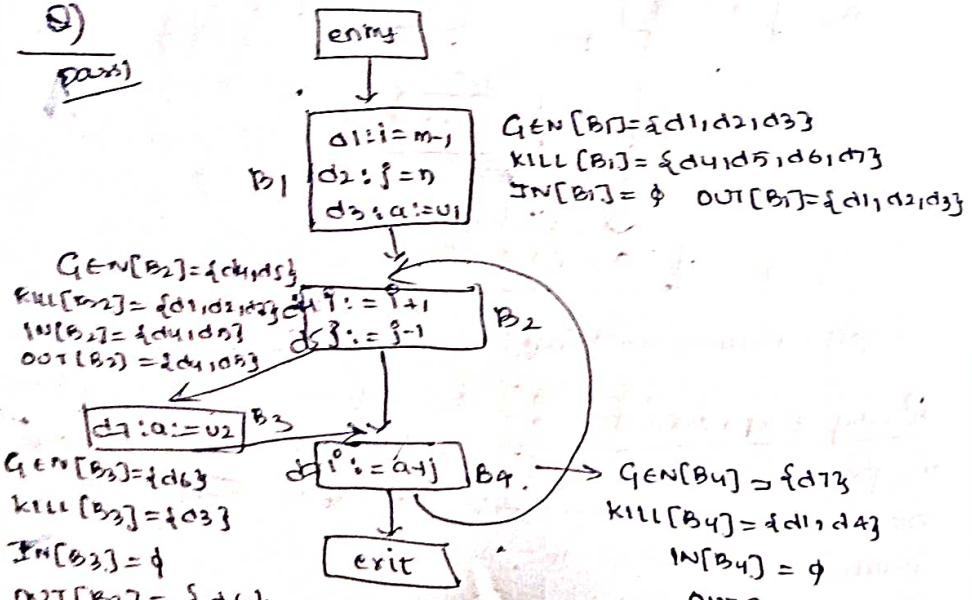
gen[B] is the info generated by the block.

In[B] is the info at the begining of the block.

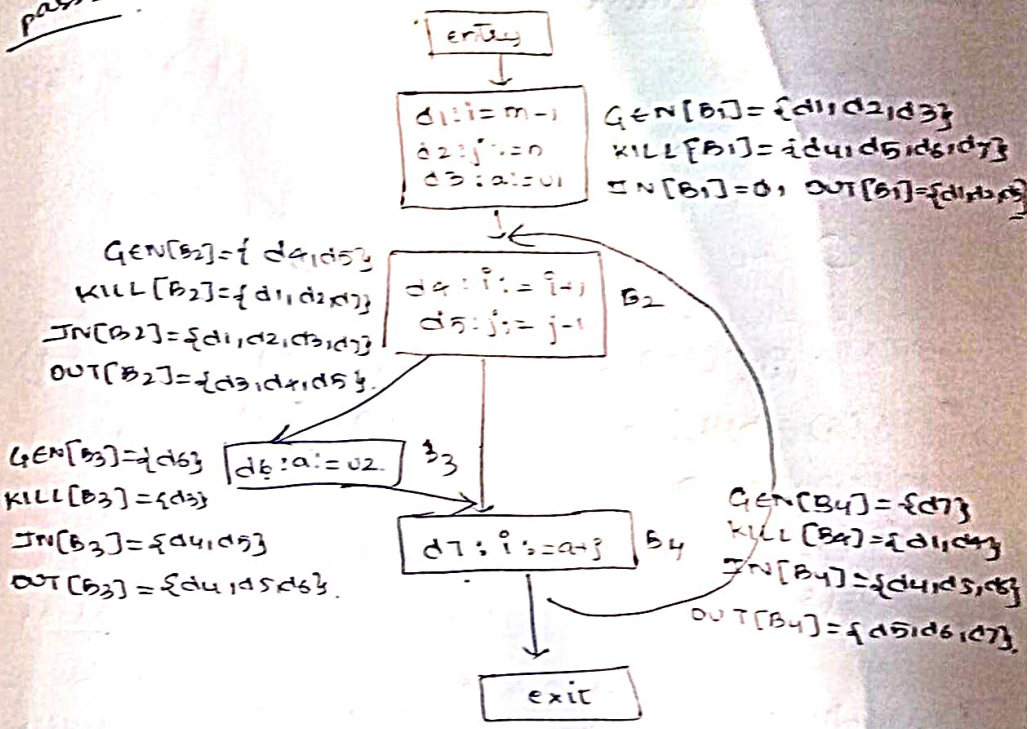
Kill[B] is the info killed / removed by the block.

a)

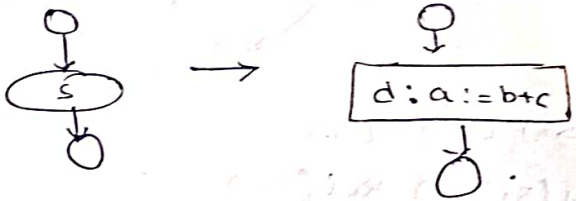
pass1



pass 2



Dataflow Equations for an Assignment :-

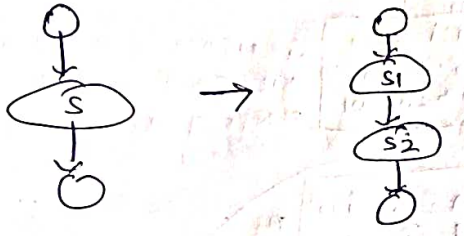


gen(S) = {d}

kill(S) = D_a - {d}

out(S) = gen(S) ∪ (in(S) - kill(S))

Data-flow equations for a sequence of statements :-



$gen(S) = gen(S_2) \cup (gen(S_1) - kill(S_2))$

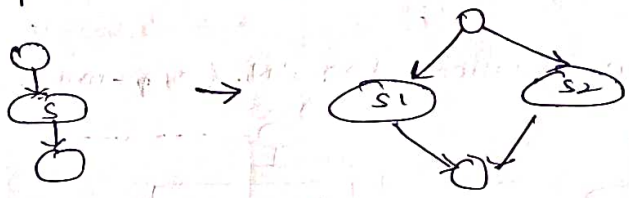
$kill(S) = kill(S_2) \cup (kill(S_1) - gen(S_2))$

$in(S_1) = in(S)$

$in(S_2) = out(S_1)$

$out(S) = out(S_2)$

Data flow equations for an alternative :-



$gen(S) = gen(S_1) \cup gen(S_2)$

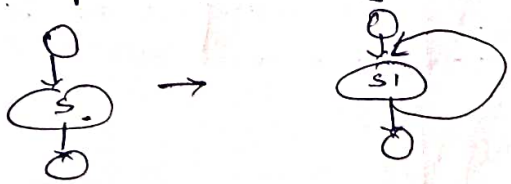
$kill(S) = kill(S_1) \cap kill(S_2)$

$in(S_1) = in(S)$

$in(S_2) = in(S)$

$out(S) = out(S_2) \cup out(S_1)$.

Data flow equations for a while loop :-



$gen(S) = gen(S_1)$

$kill(S) = kill(S_1)$

$in(S_1) = in(S) \cup gen(S_1)$

$out(S) = out(S_1)$.

Object code forms (op of Target generator)

21/12/23

1. Absolute code
2. Relocatable code
3. Assembly code

Absolute Code :-

Absolute code is a machine code that contains references to actual addresses within pgm addrs space. The generated code can be placed directly in the memory & execution starts immediately. Generally small pgms can be compiled & executed quickly.

Relocatable code :-

Producing a relocatable machine lang pgm as o/p allows sub pgms to be compiled separately. A set of relocatable obj modules can be linked together & loaded with the help of the linking loaders.

Assembly code :-

Producing an assembly lang as o/p makes the process of code generation. But generating an assembler code as o/p makes code generation process slower.

Algorithm – copy propagation

Input :- A flow graph containing used definition i.e ud chains reaching to block B.

Output:- A graph after applying copy propagation transformation.

Method:- for each stmt $x = y$ perform following

Steps :-

Step1:- determine the stmts in which x is used. This stmt shld be reachable from def of x.

Step 2:- There shld not be any def of x or y occur prior to use(x) in the block containing stmt s.

Step3:- If 's' satisfies the cond mentioned in Step2 then remove 's' & replace all uses of x dounding step1 by y.