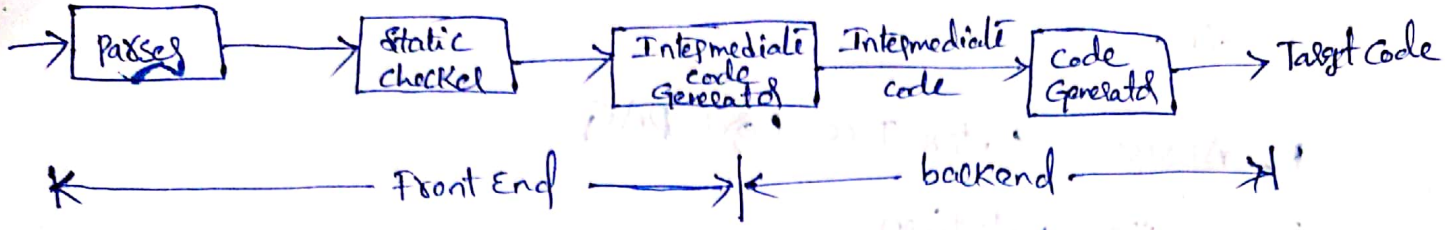
Unit - 5

Intermediate code Generation:



* This chapter deals with Intermediate Representations, Static Type checking, and Intermediate Code Generation.

* Static checking includes typechecking, which ensures that operators are applied to compatible operands. It also includes any syntactic checks that Remain after parsing.

* the process of Translating a program in given source language into code for a given target machine, a compiler may construct a Sequence of Intermediate Representations.

* High level Representations are close to the source language and low-level Representations are close to the Target machine. Syntax trees are high-level; They depict the natural hierarchical structure of the source program and are well suited to tasks like Static Type checking.

* A low-level Representations is suitable for machine-dependent tasks like Register Allocation and Instruction selection. three address code can Range from high-to low-level, depending on the choice of operators.

Intermediate Representations of source program:

There are mainly three types of Intermediate code Representations

They are.

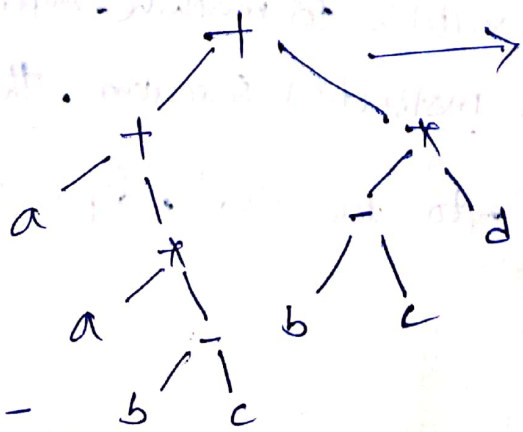
1. Abstract syntax Tree (or) DAG
2. polish notation
3. three address code.

Abstract syntax Tree: the natural hierarchical structure is Represented by syntax Tree. Directed Acyclic Graph (or) DAG is very much similar to syntax trees but they are in more compact form.

* A DAG is a directed Graph with no cycles which gives a picture of how the value computed by each statement in a basic block is used in subsequent statements in the block.

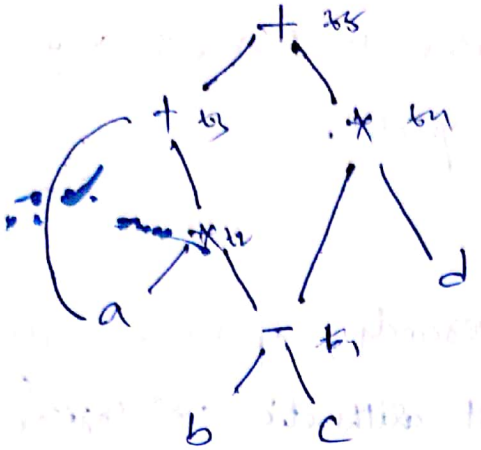
* the difference b/w the syntax Tree and Abstract syntax Tree is, in AbST, the node which is Representing a common sub expression has more than one "parent", whereas, in the syntax Tree, the common sub expression would be Represented as a duplicate subtree.

Ex: Construction of a DAG for the expression, $a + a * (b - c) + (b - c) * c$



→ Syntax Tree

the Abstract DAG for the same is given by



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

2. Polish Notation:- Basically, the linearization of syntax tree is Polish notation. In this Representation, the operator can be easily associated 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 and then operand are observed

Ex:- (a+b) * (c-d) can be written as

+ ab - cd

↓ operator ↓ operands

There is a Reverse Polish Notation which is used using postfix Representation.

Consider that the input expression is

X := -a*b + -a*b

Then the Required postfix form is

X a - b * a - b * + :=

3. Three Address Code: In three address code form at the most three addresses are used to represent any statement and also having at most one operator.

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

where a, b, & c are the operands, op means operators, it can be fixed (or) floating point arithmetic (or) logical operators on Boolean valued data. only single operation at right side of the expression is allowed at a time.

For the expression like $a = b+c+d$ the three address code will be

$t_1 := b+c$

$t_2 := t_1+d$

$a := t_2$.

Here t_1 and t_2 are temporary names generated by compiler.

Types of Three-Address statements:

Assignment statement: $x := y \ op \ z$, where op is binary arithmetic (or) logical operation.

copy statement: $x := y$, where the value of y is assigned to x

conditional Jump: if x relop y goto L

Indexed Assignment: $x := y[i]$ and $x[i] := y$

Address and pointer Assignment: $x := \&y$, $x := *y$ and $*x := y$.

Procedure call: Param x call P
return y:

Implementation of Three-Address Statements:

* There are three types of Representations.

1, Quadruples

2, Triples

3, Indirect Triples.

Quadruples:

Instructions can be implemented as objects (or) as Records with fields for the operator and the operands.

* A quadruple (or just "quad") has four fields, which we call op, arg_1, arg_2, and Result.

* The op field contains an internal code for the operator. For instance, the three-address instruction $x = y + z$ is represented by placing + in op, y in arg_1, z in arg_2, and x in Result.

three-address code for the Assignment $a = b * -c + b * -c$;

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

Quadruples

	op	arg_1	arg_2	Result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

Note: Instructions with unary operators like $x = \text{minus } y$ or $x = y$ do not use arg_2. Note that for a copy statement like $x = y$, op is =, while for most other operations, the assignment operator is implied.

* In above the special operator minus is used distinguish the unary minus operator, as in $-c$, from the binary minus operator, as in $b - c$.

Triples:- A triple has only three fields, which we call op, arg_1, and arg_2. The Result field is used primarily for temporary names.

* Using triples, we refer to the Result of an operation x op y by its position, rather than by an explicit temporary name.

* A triple Representation would Refer to Position (0), instead of the temporary t, in above.

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

	op	arg_1	arg_2
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Indirect Triples:-

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.

* Let us use an array instruction to list pointers to triples in the desired order.
* With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without effecting the triples themselves.
* Indirect triples can save some space compared with quadruples, if the same temporary value is used more than once.

Ex: $A := -B * (C + D)$

$t_1 := -B$

$t_2 := C + D$

$t_3 := t_1 * t_2$

$A := t_3$

Quadruple:

	op	arg_1	arg_2	Result
(0)	uminus	B	-	t_1
(1)	+	C	D	t_2
(2)	*	t_1	t_2	t_3
(3)	:=	t_3	-	A

Triple:

	op	arg_1	arg_2
(0)	uminus	B	-
(1)	+	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

Indirect triple:

	Statement
(0)	(21)
(1)	(22)
(2)	(23)
(3)	(24)

	op	arg_1	arg_2
(21)	uminus	B	-
(22)	+	C	D
(23)	*	(21)	(22)
(24)	:=	A	(23)

Example: Construct triples of an expressions $a*-(b+c)$. (8) (8).

S-1 We will first write the three address code for given expression.

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

S-2 Quadruple:

location	op	arg_1	arg_2	Result
(0)		a		t_1
(1)		b		t_2
(2)	+	t_2	c	t_3
(3)	uminus	t_3		t_4
(4)	*	t_1	t_4	t_5

S-3 Triples:

Location	op	arg_1	arg_2
(0)		a	
(1)		b	
(2)	+	(1)	c
(3)	uminus	(2)	
(4)	*	(0)	(3)

Example: 2 write the Quadruple, triple and indirect triply for the expression $-(a*b)+(c+d)-(a+b+c+d)$

S-1 the three address code can be

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

Assignment Statements:

* In the translation of Assignment into three address code, names can be looked up in the symbol table as follows.

Translation scheme for Assignments:

$S \rightarrow id := E$	{ p := lookup(id.name); if p ≠ nill then emit (p ':=' E.place) else error }
$E \rightarrow E_1 + E_2$	{ E.place = new temp; emit (E.place ':=' E_1.place '+' E_2.place) }
$E \rightarrow E_1 * E_2$	{ E.place = newtemp; emit (E.place ':=' E_1.place '*' E_2.place) }
$E \rightarrow -E_1$	{ E.place := new temp; emit (E.place ':=' 'uminus' E_1.place) }
$E \rightarrow (E_1)$	{ E.place := E_1.place }
$E \rightarrow id$	{ p := lookup (id.name); if p ≠ nill then E.place := p else error }

Static Single - Assignment form: (SSA)

→ SSA is an intermediate Representation that facilitates certain code optimisations.

→ Two distinctive aspects distinguish SSA from 3-address code

→ The first is that all assignments in SSA are to variables with distinct names; hence the term SSA.

→ Note that subscripts distinguish each definition of variables p and q in the SSA representation.

Ex: 1

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

3-address code

⇒

```
p1 = a+b
q1 = p1-c
p2 = q1*d
p3 = e-p2
q2 = p3+q1
```

SSA.

∵ no of temp variables = 5

no of variables = 5

total variable = (10)

$p_1\ p_2\ p_3\ q_1\ q_2$ — Temp

a, b, c, d, e — variables

Ex: 2

```
a = u-t;
y = a*v;
x = y+w;
y = t-z;
y = a*y;
y = w+y
```

⇒

```
a1 = u-t
y1 = a1*v;
a2 = y1+w
y2 = t-z
y3 = a2*y2
y4 = w+y3
```

5 — temp variable

6 — variable

total = (11)

* The minimum no of variables Required to convert the above code segment to SSA Form is 11

Case Statements:

* The "switch" (or) "case" statement is available in a variety of languages.
* The following, There is a selected expression E, which is to be evaluated, followed by n constant values $V_1, V_2 \ldots V_n$ that the expression might take, perhaps including a default "value", which always matches the expression if no other value does.

```
switch (E)        <- int or char expression.
{
case V1 : S1
case V2 : S2
  . . . . . .
  . . . . . .
  . . . . . .
case Vn-1 : Sn-1

default : Sn
}
```

* E is an integer (or) character expression, after evaluating 'E' the Result is either int (or) char.

→ $V_1, V_2 \ldots V_{n-1}$ are case values

→ $S_1, S_2 \ldots S_{n-1}$ are statement Blocks

→ If expression matches with case V, then the corresponding $S_1 \ldots S_{n-1}$ will be executed, otherwise default block will be executed

Translation of switch-statements: (SDT)

```
    Code to evaluate E into t
        goto test
L1  : code for S1
        goto next
L2  : code for S2
        goto next
        . . . .
Ln-1 : code for Sn-1
        goto next
Ln  : code for Sn
        goto next.
test : if t = V1 goto L1
       if t = V2 goto L2
       . . . .
       . . . .
       if t = Vn-1 goto Ln-1
          goto Ln
next :
```

Boolean Expression: (or) Translation Boolean Expressions:

* Boolean expression it may return either TRUE (or) False
* Boolean expression have two primary purpose
  1) used for computing logical value
  2) used as conditional expressions using if-then else & while-do statements.

Consider a grammar: logical values of Boolean expressions are

1. logical OR 2. logical AND 3. logical NOT.

E → E OR E
E → E AND E
E → NOT E
E → (E)
E → id relop id.
E → TRUE
E → False.

→ production

B → B_1 || B_2 (logical OR) both A & B are False → False

Remaining TRUE

B_1.true ⇒ B.true.
B_1.False ⇒ new label()
B_2.TRUE ⇒ B.TRUE
B_2.False = B.False.

} Semantic Actions

TAC: B.code = B_1.code || label(B_1.False) || B_2.code.

$B \rightarrow B_1 \&\& B_2$ (Logical AND) (∵ if both A & B True → True

Remaining all cases False.

B_1.True → new label()

B_1.False → B.False

B_2.True → B.True

B_2.True → B.False

TAC: B.code = B_1.code || label(B_1.True) || B_2.code.

$B \rightarrow !B_1$ (Logical NOT) (∵ Result of Expr True → False
and False → True)

B_1.True = B.False

B_1.False = B.True

TAC: B.code = B_1.code

B → True

B.code = generate("goto" B.True)

B → False

B.code = generate("goto" B.False)

Procedure call:

A Procedure call like $P(A_1, A_2 \cdots A_n)$ may have the many Addresses for one statement in 3-address code. So its shown as a sequence of n+1 statements.

$P(A_1 A_2 \cdots A_n)$

Param A_1

Param A_2

⋮

Param A_n

Call P, n

→ where P is ~~n~~ name of the Procedure (or) Function. n is integer indicating number of Actual parameters in the call.

→ we use the term Function for Procedure that returns a value.

Suppose that 'a' is an array of integers, and that 'f' is a function from integer to integer. Then assignment statement

$$n = f(a[i])$$

it might Translate into the Following three-address code

1) $t_1 := i * 4$
2) $t_2 := a[t_1]$
3) Param t_2
4) $t_3 := $ call f, 1
5) $n := t_3$
6)

The first two lines compute the value of expression a[3] into temporary t_2. Line 3 makes t_2 an actual parameter for the call on line 4 of p with one parameter. Line 5 assigns the value returned by the function call to t_3. Line 6 assigns the returned value to n.

→ The following productions allow function definitions and function calls. Nonterminal D generates type of procedure T generate Return type (or) Data type, F is formal parameters, S is block of statements, E is expression and A is actual parameters respectively.

$D \rightarrow$ define T id (F) {S}

$F \rightarrow \epsilon \mid T$ id, F

$S \rightarrow$ Return E;

$E \rightarrow$ id(A)

$A \rightarrow \epsilon \mid E, A.$

∴ Adding functions to the source language.

```
Float add()
    (Int a)
    (Int a, Float b)
{
Return add()
}
```

Procedure call for source language:

```
Pnt main()
{
    Pnt x,y;
    Swap (&x, &y);
}
Void Swap (int *a, Pnt *b)
{
    Pnt i;
    i = *b;
    *b = *a;
    *a = i;
}
```

1. call main
2. Param &x
3. Param &y
4. call swap, 2
5. Param i
6. i = *b
7. *b = *a
8. *a = i;
9. Stop
10.
11.

Symbol Table management:

* Symbol Table is a data structure which is used by compiler to keep Track of scope and binding information about names i.e, a Compiler needs to collect and use information about the names appearing in the source program.
* This information is entered into a data structure called a symbol Table during lexical and syntactic analysis.
* The symbol table is searched every time a name is encountered in the source text.
* the information includes the string of characters by which it is denoted as, its type, its form, its location in memory and other attributes depending on the language.
* Each entry in the symbol table is a pair of the form (name, information)

the Symbol Table is used by various phases as follows.

→ lexical analyser : stores the information of the symbols in symbol table

→ Parser : while checking the syntax of the source, makes use of symbol table to verify the information about the tokens

→ Semantic phase : Refers symbol table for type conflict issue.

→ Code Generation : Refers symbol table knowing how much Run time space is allocated? what type of Run-time space is Allocated?

l-value and r-value:

the l and r prefixes come from left and right side Assignment

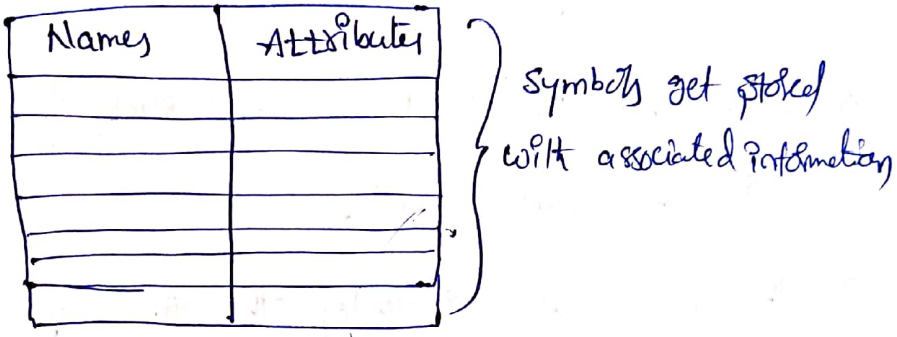
Ex:

$$\boxed{a} := \boxed{i+1}$$

l-value r-value

use of symbol Table:

* To achieve compile time efficiency compiles makes use of symbol Table.
* It Associates lexical names with their attributes.



- the items to be stored in symbol to table are: (or) Attributes of S.T

1, Variable names
2, Constants
3, Procedure names
4, Function names
5, Literal constants and strings
6, Compiles generated temporaries
7, Labels in source language.

- Compiles uses following types of information from symbol-table

1, Data type
2, Name
3, Declaring procedure
4, offset in storage
5, For parameters, whether parameter is passing is by value (or) Reference?
6, Number and type of arguments passed to the function
7. Base address.

Types of Symbol Tables:

The Organization of symbol table for non block structured languages is by ordered (or) unordered manner.

Ordered Symbol Table: In this table the entries of variables is made in alphabetical manner.

* the searching of ordered symbol table can be done using lineal and binary search.

Advantages: 1. The searching of particular variable is efficient.

2. The Relationship of particular variable with another can be established very easily.

Disadvantages: 1. Insertion of element in ordered list is costly if there are large number of entries in symbol table.

Unordered Symbol Table: As variable is encountered, then its entry is made in symbol table. These entries are not in sorted manner. Each time, before inserting any variable in the symbol table, a lookup is made to check whether it is already present in the symbol table (or) not.

Advantages:

1. Insertion of variable in symbol table is very efficient

Disadvantages: 1. searching must be done using linear search.

How to store Names in symbol Table?

There are two types of name Representation.

1. Fixed_length name
2. Variable length name.

1. **Fixed_length name:** A fixed space for each name is allocated in symbol table. In this Type of storage if name is too small then there is wastage of space.

Ex:

Name	Attribute
calculate	
sum	
a	
b	

the name can be refferred by pointer to symbol table entry.

2. **Variable length name:** the amount of space required by string is used to store the names. the name can be stored with the help of starting index and length of each name

Ex:

Name		Attribute
Starting Index	length	
0	10	
10	4	
14	2	
16	2	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
c	a	l	c	u	l	a	t	e	$	s	u	m	$	a	$	b	$

Need for Symbol Table: The symbol tables are required for

(i) For Quick insertion of identifies and Related information

(ii) For Quick searching of identifier

Organization for Block structure languages:

* The block structured language is a kind of language in which section of source code is within some matching pair of delimiters as "{" and "}" or begin and end.
* And such a section gets executed as one unit (or) one procedure (or) a function (or) it may be controlled by some conditional stmt (if, while, do-while).

Following are data structures that are used for organization of block structured languages.

1. Arrays (Linear list)
2. List (self organising list)
3. Hashing
4. Tree structure representation of scope information (binary search

Arrays: (Linear list)

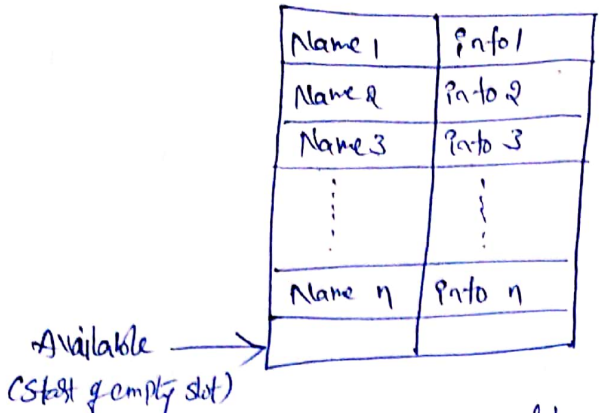
→ Linear list is a simplest kind of mechanism to implement the symbol table.

→ In this method an array is used to store names and associated information.

→ New names can be added in the order as they arrive.

→ The pointer 'available' is maintained at the end of all stored Records.

The following figure for list data structure using Arrays



→ To Retrive the information about some name we start from beginning of array and go on searching up to available pointer. if we reach at pointer available without finding a name we get an error " use of undeclared name".

→ while inserting a new name we should ensure that it should not be already there. if it is already present there then another error occurs i.e, " multiple defined name".

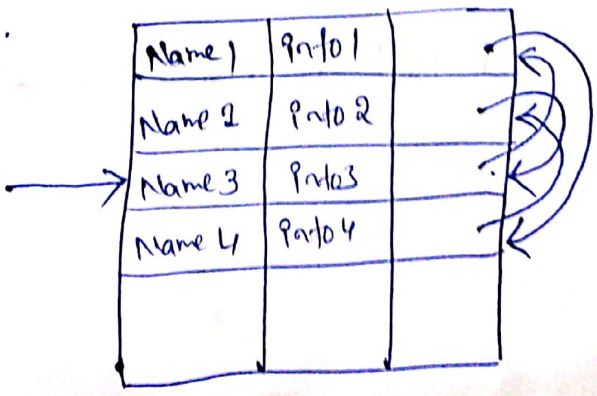
→ The advantage of list organisation is that it takes minimum amount of space.

List (self organising list):-

→ this symbol table implementation is using linked list. A link field is added to each Record.

→ we search the Records on the nodes pointed by link of the link field.

→ A pointer " First" is maintained to point to first Record of the symbol table.



→ The Reference to these names can be Name 3, Name 4, Name 2, Name 1. (3)

(21)

→ when the name is referenced or created it is moved to the front of the list.

→ The most frequently referred names will tend to be front of the list. Hence access time to most frequently referred names will be the least.

Hashing: Hashing is an important technique used to search the Records of symbol table. This method is superior to list organization

→ In hashing scheme two tables are maintained a hash table and symbol table.

→ The hash table consists of K entries from 0, 1 to K-1. These entries are basically pointers to symbol table pointing to the names of symbol table.

→ To determine whether the 'name' is in symbol table, we use a hash function 'h' such that h(name) will Result any integers between 0 to K-1. We can search any name by,

Position = h(name)

→ using this position we can obtain the exact locations of name in symbol table

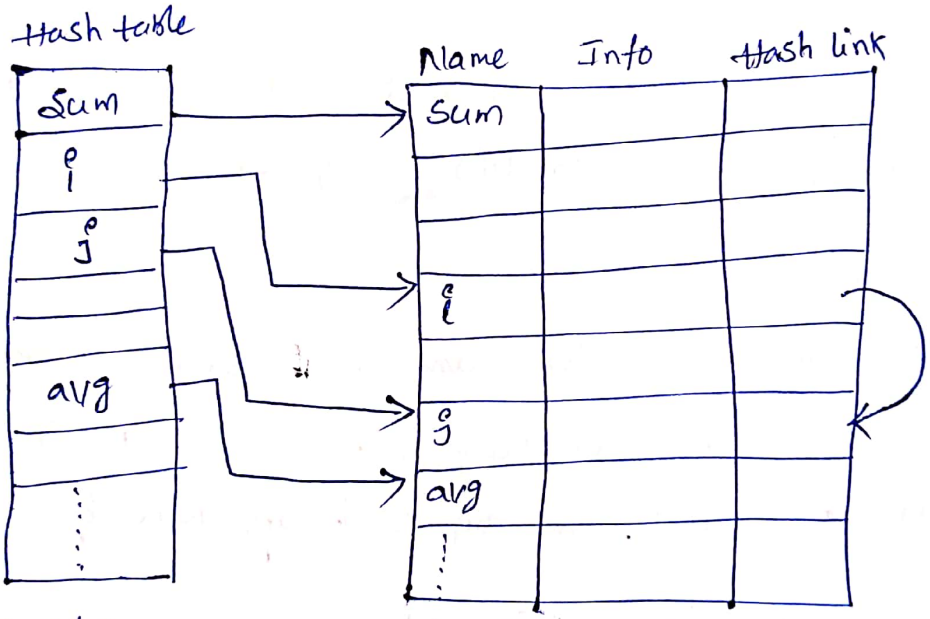
→ The has function should Result in uniform distribution of names in symbol table.

→ the hash-function should be such that there will be minimum (4) number of collision. (28)

→ Collision is such a situation where hash-function results in same location for storing the names.

→ Various collision Resolution techniques are open addressing, chaining, Rehasing.

→ the advantage of hashing is quick search is possible and Disadvantage is that hashing is complicated to implement. Some extra space is required. obtaining scope of variable is very difficult.

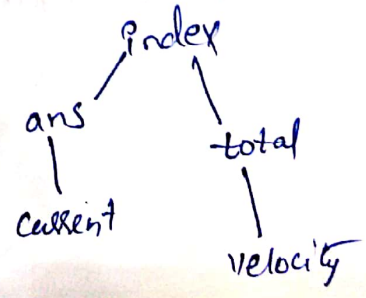


Binary Tree search:

A binary tree is build lexicographic ordering of tokens. the typical data structure is,

Left child	Name of symbol	Mode information	Right child.

Ex: int index, total, velocity, current, ans;

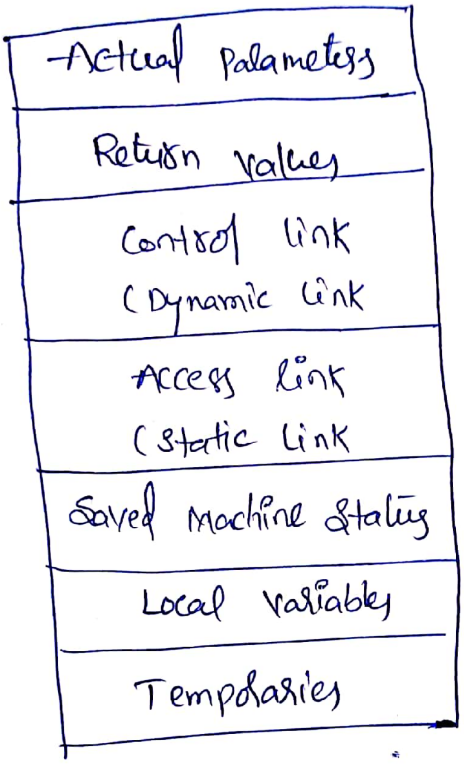


Activation Record:

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

→ whenever a function (or) procedure call occurs then activation Record is created. Activation Record information is pushed on to the top of the stack.

Model of Activation Record:



Actual Parameters: It holds the Actual parameters of calling function

Return values: to store the Result of function call

control link: It points to the Activation Record of calling function

Access Link: It refers to the non local data in other Activation Record

Saved machine Registers: stores - Address of next instruction (6)
to be executed. (or) status of machine
is just before the function is called. (30)

Local variables: these variables are local to a function.

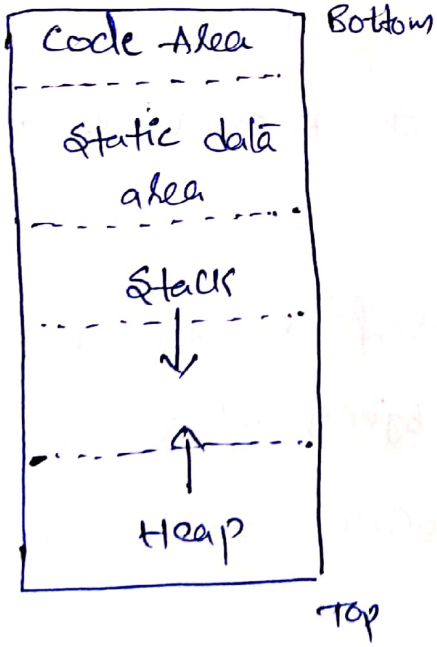
-temporary variables: Needed during expression Evaluation.

Runtime Environment:

* 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 Runtime Storage.

* The Runtime storage is subdivided to hold code and data such as.
  i) The generated Target code
  ii) Data objects
  iii) Information which keeps track of procedure activation

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



* The amount of 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. (32) (2)

* Compiler prefers to place the Data objects in the statically determined area because these data objects then can be compiled into target code.

* for example in FORTRAN all the data objects are allocated statically. hence the static data area is on the top of code area.

* the counterpart of control stack is used to manage the active procedures. managing of active procedures means that when a call occurs then execution of activation is interrupted and information about status of the stack is saved on the stack.

* when the control returns from the call this suspended activation is Resumed after storing the values of relevent Registers. Also the program counter is set to the point immediately after the call.

* this information is stored in the stack area of Runtime Storage. Some data objects which are contained in this activation can be allocated on the stack along with the Relevant Information.

* the heap area is the area of Runtime 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.

* Pascal and C need the Runtime stack.

Storage Organization:

Runtime storage is divided into

1. Code area
2. Static data area
3. Stack area
4. Heap area.

there are three 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

let us discuss each of these strategies in detail

→

Static Allocation:

→ The size of data objects is known at compile time. The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static allocation.

→ The binding of name with the amount of storage allocated do not change at runtime. Hence the name of this allocation is static allocation.

→ At compile time compiler can fill the addresses at which the target code can find the data it operates on.

→ FORTRAN uses the static allocation strategy.

Limitations of static allocation:

→ Recursive procedures are not supported by this type of allocation

→ The static allocation can be done only if the size of data object is known at compile time.

→ The data structures can not be created dynamically.

Stack Allocation:

→ Stack Allocation strategy, in which the storage is organized as stack. This is also called control stack.

→ The activation begins the activation Records are pushed onto the stack and on completion of this activation the corresponding

activation Records can be popped.

→ the locals are stored in the each activation Record. Hence locals are bound to corresponding activation Record on each fresh activation.

→ the data structures can be created dynamically for stack allocation.

Limitations of Stack Allocation:

* the memory addressing can be done using pointers and Index Registers. Hence this type of allocation is slower than static allocation.

Heap Allocation:

→ if the values of non local variables must be Retained even after the activation Record then such a Retaining is not possible by stack allocation.

→ this limitation of stack allocation is because of its Last In FIRST Out (LIFO) nature.

→ the heap allocation allocates the continuous block of memory when Required for storage of activation Records. this allocated memory can be deallocated when activation ends. this deallocated (free) space can be further Reused by heap manager.

the efficient heap management can be done by:

(i) Creating a linked list for the free blocks and when any memory is deallocated that block of memory is appended in the linked list.

(ii) Allocate the most suitable block of memory from the linked list. i.e, use best fit technique for allocation of block.

The Principal Sources of optimization:

* The code optimization phase attempts to improve the Intermediate code, so that faster-running machine code will Result.

* The term "code optimization" refers to techniques a compiler can employ in an attempt to produce a better language program than the most obvious for a given source program.

* A Transformation of a program is called local if it can be performed by looking only at the statements in a basic block, otherwise it is called global.

→ Many Transformations can be performed at both the local and global levels. Local transformations are usually performed first.

→ There are a number of ways in which a compiler can improve a program without changing the function it computes.

1, Function-preserving transformations (Local optimization)

(or) Basic Block optimization

2, Loop optimization

1, ~~Com~~ optimization of Basic Blocks:

a) Common sub-expression elimination

b) Copy propagation

c) Dead code elimination

d) Constant folding

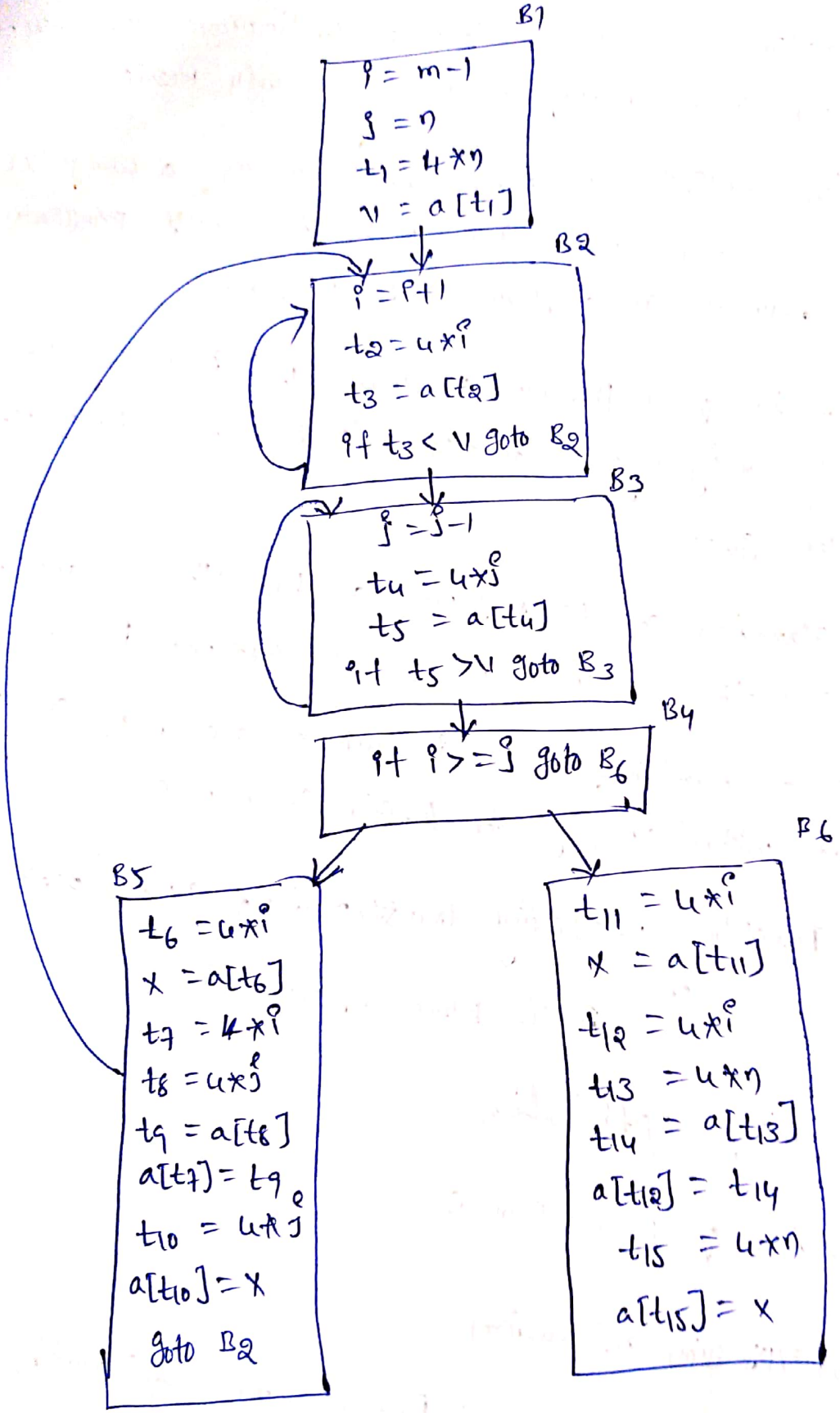
e) Renaming temporary variables

f) Interchanging statements

g) Algebraic Transformation

h) Strength Reduction

Consider the flow graph as shown in fig.



Common Subexpression elimination:

An occurrence of an Expression E is called a common sub-Exprs, if E was previously computed and the values of variables in E have not changed since the previous computation, we can avoid Recomputing the exprs, if we can use the previously computed value.

Ex: The Assignment to t_7 and t_{10} have the common sub-exprs $4*i$ and $4*j$ Respectively, on the Right side. they have been eliminated, by using t_6 instead of t_7 and t_8 instead of t_{10}

(1) * After eliminating t_7 and t_{10}, B_5 is

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

(2) * Same like After eliminating t_{12} and t_{15} B_6 is

```
t11 = 4*i
x = a[t11]
t13 = 4*n
t14 = a[t13]
a[t11] = t14
a[t13] = x
```

(3) * After eliminating t_6 and t_8, B_5 is

```
x = a[t2]
t9 = a[t4]
a[t2] = t9
a[t4] = x
goto B2
```

(4) * After eliminating t_{11} and t_{13}, B_6 is

```
x = a[t2]
t14 = a[t1]
a[t2] = t14
a[t1] = x
```

(5) After eliminating t_9, B_5 is

```
x = a[t2]
a[t2] = t5
a[t4] = x
goto B2
```

Copy propagation: It means use of one variable instead of another.

This statement $f := g$ called copy statement or copying.

Ex:

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = x$
goto B_2

$\Rightarrow$

$x := t_3$
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2.

Dead code elimination: A variable is live at a point in a program if its value can be used subsequently, otherwise it is dead at that point. i.e, statements that compute values that never get used.

Ex: 1

$x = t_3$ → Dead code
$a[t_2] = t_5$
$a[t_4] = x$
goto B_2

$\Rightarrow$

$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2.

Ex: 2

```
sum = 0
if (sum)  ⟶ Dead code.
printf ("%d", &sum)
```

Constant Folding: Replacing an expression which can be computed at compile time by its value.

Ex: 1

$x = 2+3*4+d+f$
$x = 14+d+f$

Ex: 2

$z = 2+9+k$
$z = 11+k$

Renaming temporary variables.

$t_1 = b+c$
$t_2 = a-t_1$
$t_1 = t_1 * d$
$d = t_2+t_1$

$\Rightarrow$

$t_1 = b+c$
$t_2 = a-t_1$
$t_3 = t_1 * d$
$d = t_2+t_3$

Interchange of statements:

-the states depending on operation

$t_1 = b+c$
$t_2 = a-t_1$
$t_3 = t_1 * d$
$d = t_2+t_3$

$\Rightarrow$

$t_1 = b+c$
$t_3 = t_1 * d$
$t_2 = a-t_1$
$d = t_2+t_3$.

Algebraic Transformation:

(:. Refer unit-6 peephole optimization)

Ex) $t_1 = a-a$ $\quad t_1 = 0$
$t_2 = b-t_1$ $\quad t_2 = b$
$t_3 = t_2 * 2$ $\quad t_3 = t_2 << 1$ (left shift.

Strength Reduction:

(:. Refer unit-6 peephole optimization)

$a^2 = a * a$ (Power operator Replaced by multiplication)
$2a = a+a$ (multiplication Replaced by addition)

Loop optimisation Techniques:

* the code optimisation can be significantly done in loops of the program. specially inner loop is a place where program spends large amount of time.
* hence if number of instructions are less in inner loop then the Running time of the program will get decreased to a large extent.
* hence loop optimisation is a Technique in which code optimisation performed on inner loops.

a) Code motion
b) Induction variable
c) Loop Invariant method
d) Loop unrolling
e) Loop fusion.

Code motion: it is a technique, which moves the code outside the loop if it won't have any difference if it execute inside (or) outside loop.

Ex:

```
for (i=0; i<n; i++)
{
  x = y+z;
  a[i] = 6*i;
}
```

⇒

```
x = y+z
for (i=0; i<n; i++)
{
a[i] = 6*i;
}
```

Ex-2

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

⇒

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

Induction Variables: A variable x is called an Induction variable of loop L. if the value of variable gets changed every time. it is either decremented (or) incremented by some constant.

Ex:

```
i = i+1
t1 = 4*j
t2 = a[t1]
if t2 < 10 goto B,
```

Here i, t1 are Induction variables,

loop Invariant method: the stmts in the loop whose Results of the Computations do not change over the iteration.

Ex:

```
a = 10;
b = 20;
c = 30;
For (i=0; i<5; i++)
 {
  c = a+b;
  d = a-b;
  e = a*b;
  s = s+i;
 }
```

(∵ Here a, b, & values are assigned above "For loop" do not depend on 'i') simply we can eliminate these variables.

loop unrolling: loop overhead can be Reduced by Reducing number of iterations and Replacing the body of For loop.

Ex: 1
```
for(i=0; i<100; i++)
    add();
```
⇒
```
for(i=0; i<50; i++)
{
add();
add();
}
```
(: 50*2 = 100.

Function call 100 times

Ex: 2
```
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++;
}
```

→ In this method the number of jumps and test can be Reduced by writing the code two times.

Loop Fusion: In loop fusion method several loops are merged to one loop (or) Adjacent loops can be merged in to one loop to Reduce loop overhead and improve the performance.

Ex: 1
```
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
```

Ex: 2
```
for (i=0; i<10; i++)
a[i] = a[i]+10;
for (i=0; i<10; i++)
b[i] = b[i]+10
```
⇒
```
for (i=0; i<10; i++)
{
a[i] = a[i]+10
b[i] = b[i]+10
}
```

Directed Acyclic Graphs (DAG's):

* A DAG is a directed graph with no cycles which gives a picture of how the value computed by each statement in a basic block is used in subsequent statements in the block.

* A DAG for an expression identifies the common subexpressions in the expression.

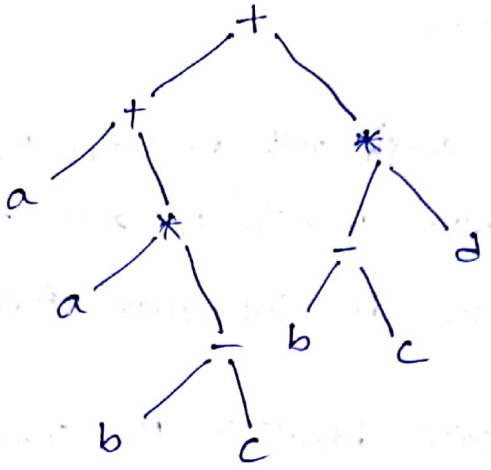
For Example: In (a+b)*c + (a+b), the sub expression (a+b) is Repeated This can be easily identified using DAG.

* Like a syntax tree, a DAG has a node for every subexpression of the expression. An interior node Represents an operator and its children Represents its operands.

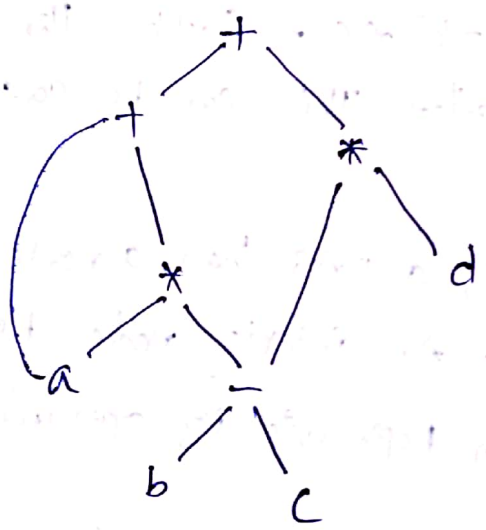
* The difference between the DAG and syntax tree is, in DAG, the node which is Representing a common subExpression has more than one "parent", where as, in the syntax Tree, the common subExpression would be Represented as a Duplicate sub Tree.

→ Construction of a DAG for the expression, a+a*(b-c)+(b-c)*d

The parse tree is given by:



The DAG for the same is given by.



This shows that the common subexpressions 'a' and (b-c) which are repeated in the expression are retained only once in the DAG.

* The nodes which were representing the duplicate of these are eliminated in DAG. Thus the node for 'a' has two parent nodes i.e, + and *, similarly, the common sub expression (b-c) has node * and * as its parent. It should be taken care that b and c are not Repeated, but the subexpression (b-c) is Repeated.

The sequence of instructions for constructing this DAG is (47)

P_1 := mkleaf (id, a);
P_2 := mkleaf (id, a);
P_3 := mkleaf (id, b);
P_4 := mkleaf (id, c);
P_5 := mknode ('-', P_3, P_4);
P_6 := mknode ('*', P_2, P_5);
P_7 := mknode ('+', P_1, P_6);
P_8 := mkleaf (id, b);
P_9 := mkleaf (id, c);
P_{10} := mknode ('-', P_8, P_9);
P_{11} := mkleaf (id, d);
P_{12} := mknode ('*', P_{10}, P_{11});
P_{13} := mknode ('+', P_7, P_{12});

Note:

1, leaves of a node are labeled by unique identifiers. This can be either variable name (or) some constant value.

2, The label of interior nodes is an operator symbol.

3, Nodes are also optionally labeled by a list of identifiers.

Applications of DAG:-

1, Determining the common sub-expression (expression computed more than once).

2, Determining which names are used inside the block and computed outside the block.

3, Determining which statements of the block could have their computed values outside the block.

4, Simplifying the list of Quadruples by eliminating the common sub-expression and not performing the assignment of the form $x := y$ unless and until it is a must.

Construction of DAG:

A DAG for a block is constructed from the three-address statement by processing each statement in turn. The Resultant DAG contains the following information.

1, Each node is given a label leaves are labeled by an identifier and constant and interid nodes are labeled by an operator symbol.

2, Each node maintain a list of identifiers attached to it.

In the construction of a DAG, a function node(id) Returns the most Recently created node for the identifier Represented by 'id'.

An algorithm for constructing a DAG consists of three steps which are executed for each statement of the block. we consider three cases,

(i) $a := b \; op \; c$

(ii) $a := op \; b$

(iii) $a := b$

initially it is assumed that there are no nodes Therefore. node is undefined for all arguments.

1) if there is no node for b i.e, node b is undefined, There create a leaf node b and let node b is Represented by by that leaf in case (i) if there is no node for c i.e, node c is undefined then create a leaf node 'c' and let node 'c' is Represented by that leaf.

2) In case (i), look for the node Representing an operator 'op' with two children where node b being left child and node c being right child. if such a node does not exist, then create it and let the newly created (or) existing node to be n.

- In case (ii), look for the node representing an operator 'op' with only a single child i.e, node b. if such a node does not exist, then create it and let the newly created (or) existing to be n.
- In case (iii), let n Represents the node b.

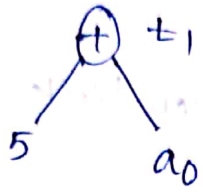
3) From the list of identifiers for node a delete a and add it to the list of identifiers for the node n and set node (a) = n.

Ex:- Consider a sequence of three-address code for a block,

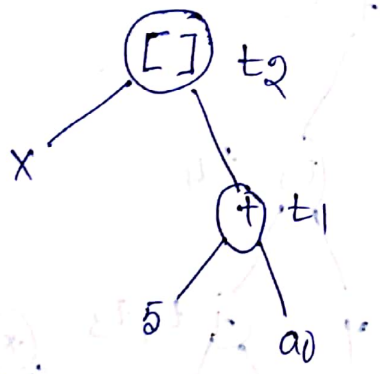
I_1 : $t_1 := 5 + a$
$t_2 := x[t_1]$
$t_3 := 5 + a$
$t_4 := y[t_3]$
$t_5 := t_2 + t_4$
$t_6 := b * t_5$
$b := t_6$
$t_7 := a * 3$
$a := t_7$
if $a < 10$ goto I_1

A DAG for this block is constructed as follows. take the statement $t_1 := 5 + a$. we first create two leaves one labeled by constant value 5 and another labeled

by an identifier a_0 (the subscript 0 indicates the initial value of a). Next we create a node for an operator + and finally we attach the identifier t_1 to the newly created node.



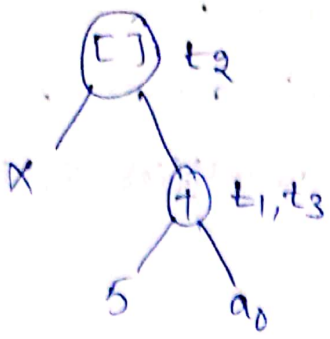
$t_2 := x[t_1]$



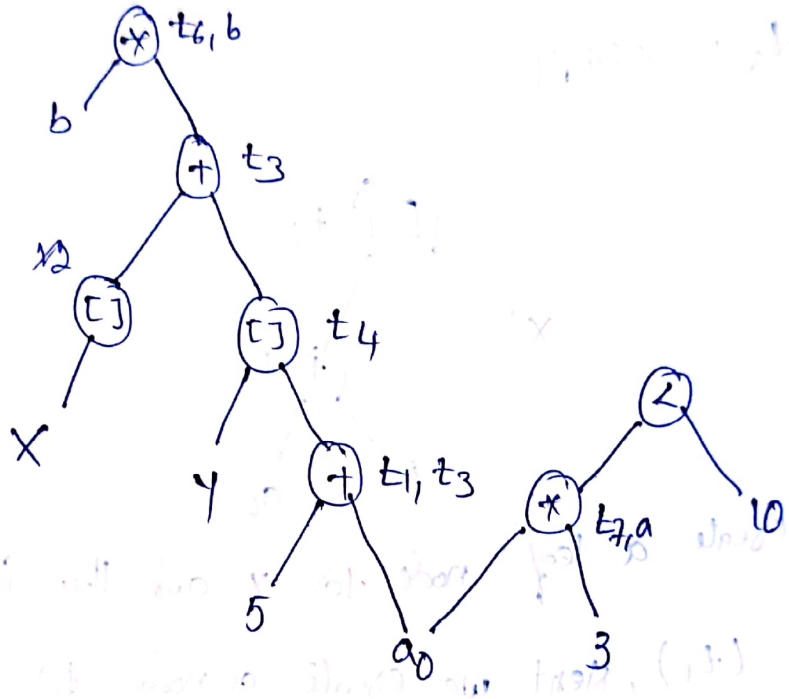
we first create a leaf node for x and then find a node for t_1 as node (t_1). Next we create a node for an operator [] and attach to it the nodes x and t_1 as its children. The node for [] is labeled as t_2.

$t_3 := 5 + a$

Now consider the third statement, since there is already a node for + whose left child is node (5) and right child is node (a), we don't create a new node for the operator rather we simply add the label t_3 on the list of identifiers for node t_1.



The complete DAG for the given block is shown in follows



Ex:- DAG for a three-Address statements.

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

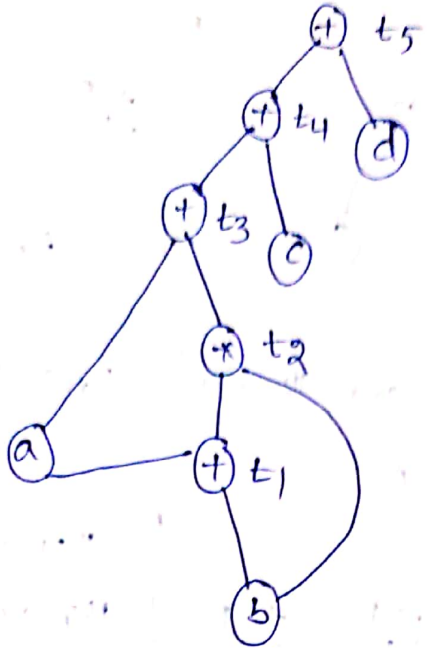
$t_1 := a + b$

$t_2 := b * t_1$

$t_3 := a + t_2$

$t_4 := t_3 + c$

$t_5 := t_4 + d$



*

Examples

a) Construct the DAG for the following basic blocks.

D := B*C

E := A+B

B := B+C

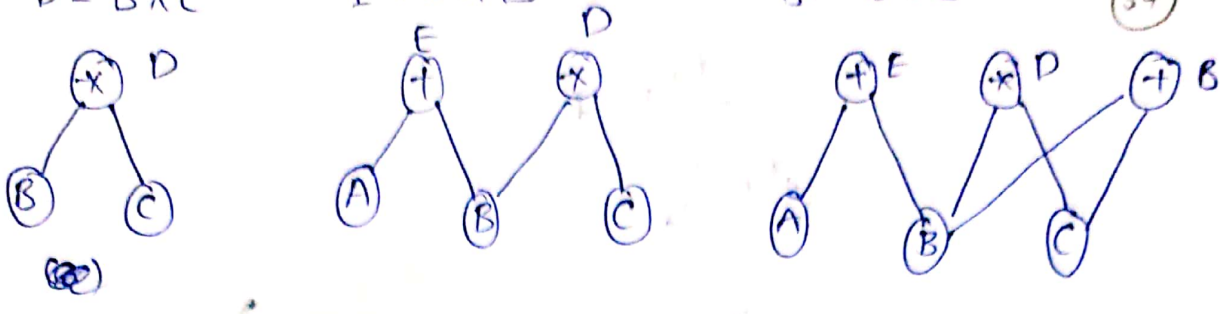
A := E-D

b) what are the legal evaluation orders and names for the values at the nodes for the DAG of problem (a)

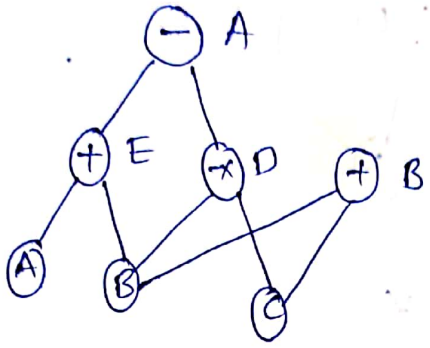
i) Assuming A, B and C are alive at the end of the basic block

ii) Assuming only A is alive at the end.

a) D = B * C E := A + B B := B + C

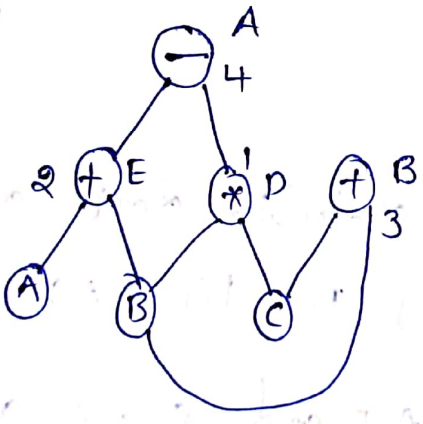


The last statement A := E - D and the final DAG is

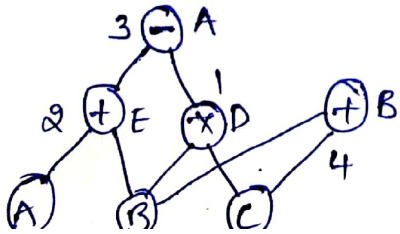


B = B + C, do not effect the evaluations of A.

b) i) A variable is said be alive at the end its value is Required outside the block. Given A, B and C, are alive at the end of the basic block, there is no change in the evaluation order (i.e, the evaluation order for the original DAG is used). DAG with the evaluation order and names is shown



ii) if only A is alive i.e, its new computed value is used outside the basic block, thus the first is computed then E and then A, B is computed at the end as it does not effect the evaluation of A. The DAG with the evaluation order and names is shown.



Ex: Explain how following expression can be converted in a DAG (4)

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

Sol: we will build the three address code for given expression

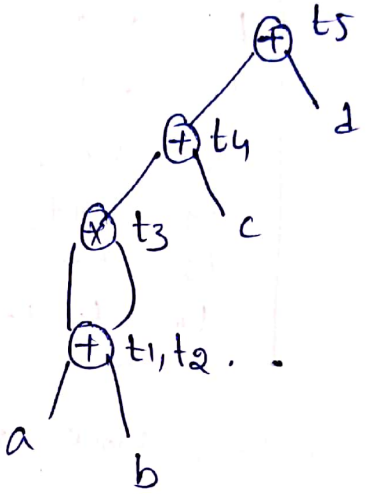
$t_1 := a+b$;

$t_2 = a+b$

$t_3 = t_1 * t_2$

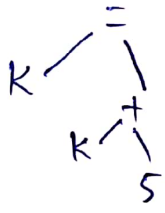
$t_4 = t_3 + c$

$t_5 = t_4 + d$.



3 Ex: 1 $K = K+5$

Syntax tree



DAG

2) Consider the following Grammar.

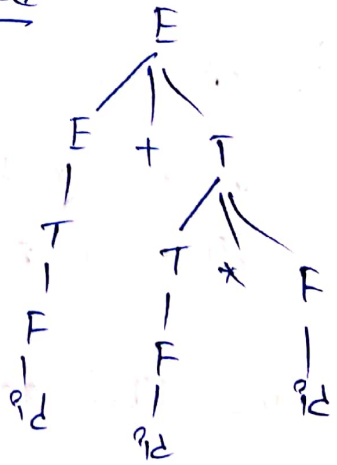
$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

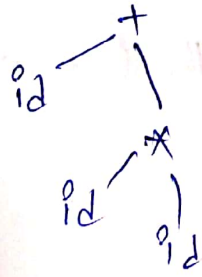
$F \rightarrow (F) \mid id$

String = id+id*id.

Parse Tree



Syntax Tree:



DAG

Basic Blocks and Flow Graphs:

* A 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 except at the end. for example, the following sequence of three address statements forms a basic block.

```
t1 = a*a
t2 = a*b
t3 = 2*t2
t4 = t1+t3
t5 = b*b
t6 = t4+t5
```

→ Basic Block.

* A name in a basic block is said to be live at a given point if its value is used after that point in the program. i.e; in another basic block..

Algorithm for partitioning into Blocks:

This algorithm is used to partition a sequence of three address statements into basic blocks.

Input: A sequence of three address statements

output: A list of basic blocks with each three address statement in exactly one block.

Method: we first determine the set of leaders, the first statement of basic blocks. The Rules used are.

1) The first statement is a ~~leader~~ Leader

2) Any statement that is the target of a conditional (or) unconditional goto is a Leader

3) Any statement that immediately follows a goto (or) conditional goto statement. is a leader.

For each leader, its basic block consists of the leader and all stmts upto but not including the next leader (or) the end of the program.

Note → Conditional (or) unconditional goto stmts do not appear as intermediate in a block.

Ex:

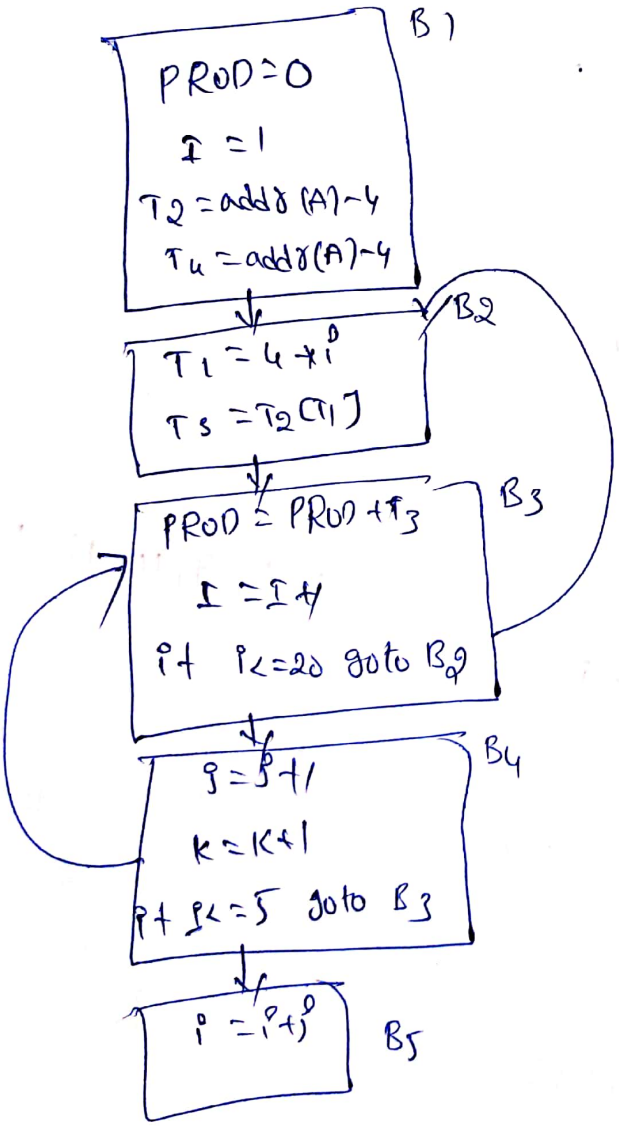
1. PROD = 0
2. I = 1
3. T_2 = addr(A) - 4
4. T_4 = addr(B) - 4
5. T_1 = 4 * I
6. T_3 = T_2[T_1]
7. PROD = PROD + T_3
8. I = I + 1
9. if I <= 20 goto (5)
10. J = J + 1
11. K = K + 1
12. if J <= 5 goto (7)
13. I = I + J

∴ 1, 5, 7, 10, 13 - Leaders.

Flow Graph: Flow graph is a directed graph which gives the flow of control information to the set of blocks making up a program. The nodes of the flow graph are the basic block.

One node is distinguished as initial; it is the block whose leader is the first statement. There is a directed edge from block B_1 to B_2, if B_2 can immediately follow B_1 in some execution sequence. That is if,

→ there is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2.



Construct DAG Fd Following expressions!

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

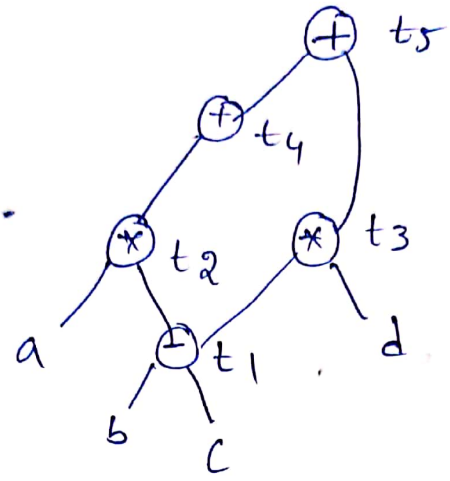
$t_1 := b-c$

$t_2 := a*t_1$

$t_3 := t_1*d$

$t_4 := a+t_2$

$t_5 := t_4+t_3$



2) $((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$

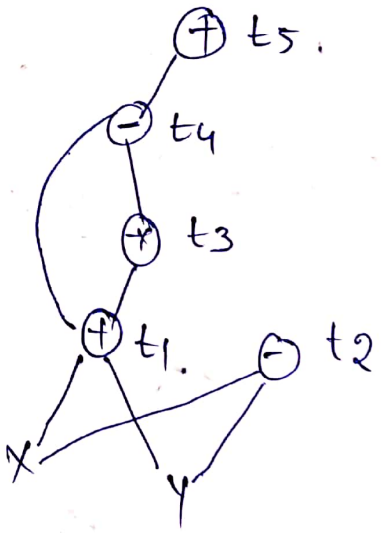
$t_1 = x+y$

$t_2 = x-y$

$t_3 = t_1*t_2$

$t_4 = t_1-t_3$

$t_5 = t_4+t_3$



3) $a=10$

$b=4*a$

$t_1=i*j$

$c=t_1*b$

$t_2=15*a$

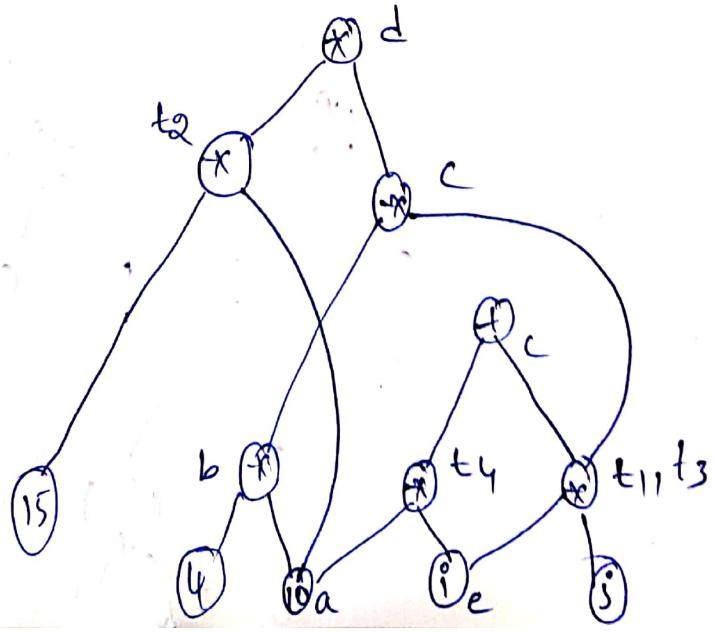
$d=t_2*c$

$e=i$

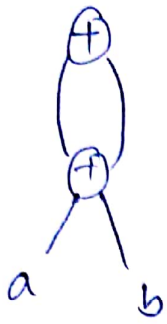
$t_3=e*j$

$t_4=i*a$

$c=t_3+t_4$

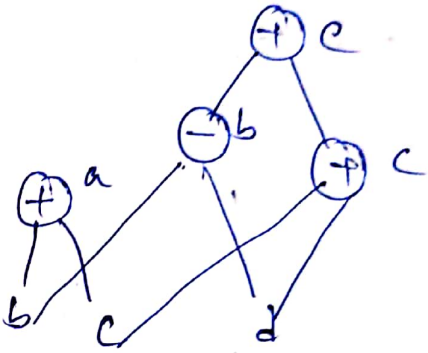


4) (a+b) + (a+b)

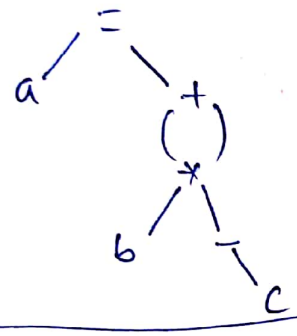


6) a+a

9) a=b+c b=b-d
c=c+d e=b+c

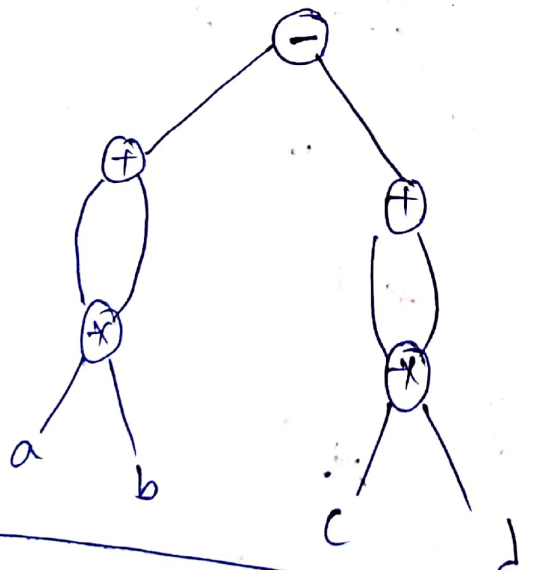


10) a=b*-c+b*-c

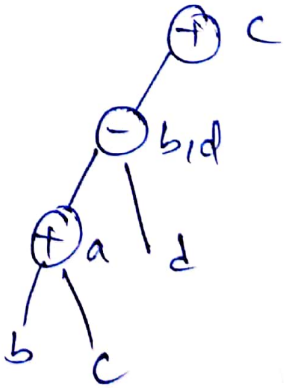


5) (a*b)+a

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



8) a=b+c b=a-d
c=b+c d=a-d.



11) a=(a*b+c)-(a*b+c)