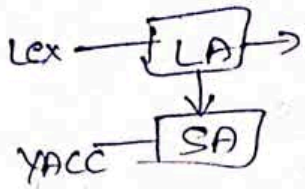
YACC tool - Yet Another Compiler Compiler.

lex - used for Lexical Analyzer gen
YACC - Parser generator.

YACC working

Step 1 -

YACC is a tool which generate LALR Parser.

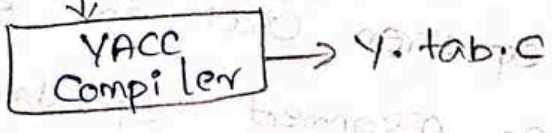


which help the Syntax Analyzer to Produce the Synt

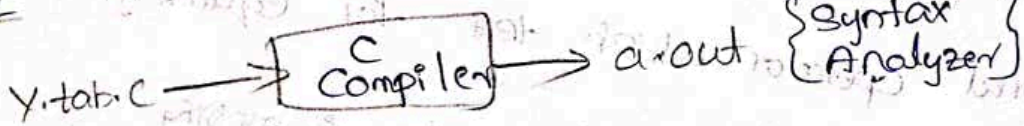
YACC working

Step 1 YACC specification.

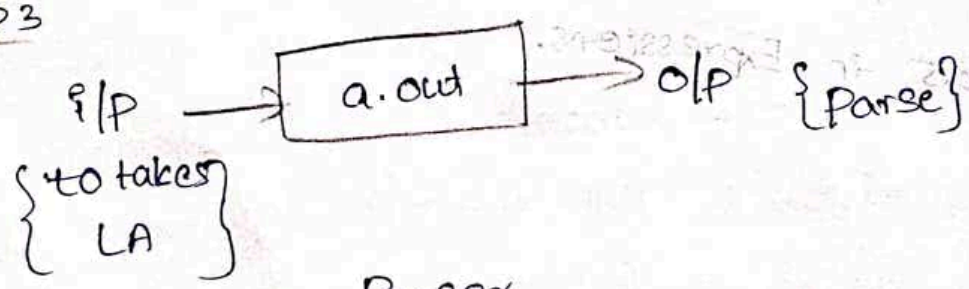
Parser.y



Step 2



Step 3



file containing
desired grammer
in YACC format

Type Conversion (or) Type Casting

A type cast is basically conversion from one type to another.

There are two types of conversion.

1) Implicit Conversion

2) Explicit type Conversion.

Implicit Type Conversion

If a compiler converts one data type into another type of data into automatically is called Implicit type Conversion.

While converting one data type to another data type there is no data loss

Eg: short a=20;

int b=a; // implicit Conversion.

bool → char → short int → int → long → float.

2. Explicit type Conversion

When data of one type is converted explicitly to another type with the help of Predefined function.

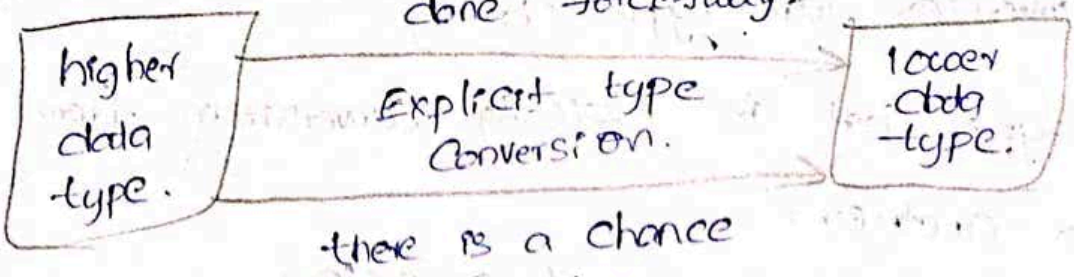
* In explicit type Conversion There is a data loss

Here forcefully data converted.

* Some functions Conversions cannot be made implicit.

int to short

done forcefully.



there is a chance of data loss

Symbol table

Implementation of Symbol Table

The Symbol table can be an implemented in the unordered list if the compiler is used to handle the small amount of data.

A Symbol table can be implemented in one of the following techniques.

* linear list
* Hash table
* Binary Search tree.

Symbol tables are mostly implemented as hash table.

The operations provided by Symbol table are.

* Insert()
* lookup()

Insert(): It is more frequently used in analysis phase. when tokens are identified and names are stored in table.

The insert() function takes the symbol and its value in the form of argument.

eg: `int x;` should be processed by compiler as

insert (x, int)

lookup: It is used to search a name & it determine

* The existence of symbol in the table.
* Declaration of the symbol before it is used.
* check whether the name is used in the scope
* Initialization of the symbol.
* Checking whether the name is declared multiple times.

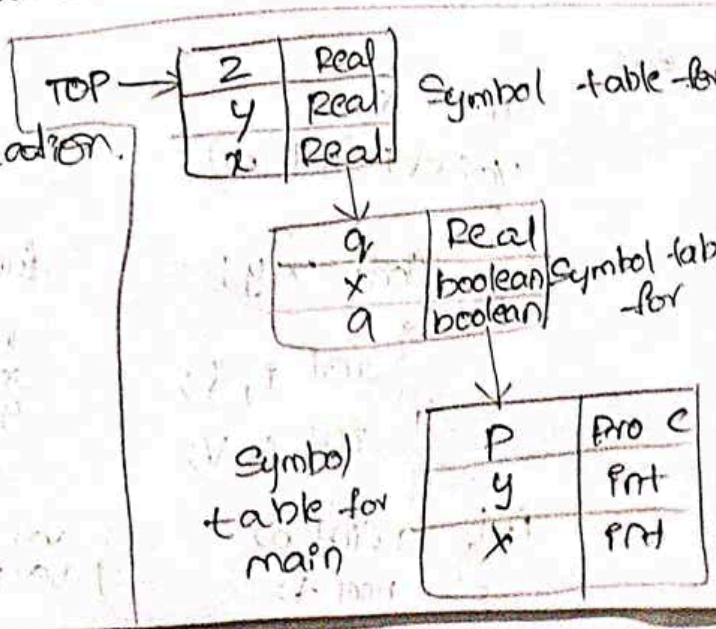
The basic format of lookup() function is

lookup (symbol)

This format is varies according to the programming language.

- Symbol table organization.

```
var x, y : integer;
Procedure P:
var x, a : boolean
procedure q:
var x, y, z : real
begin end.
```



when ever a

Representing Scope Information.

In source program every name possesses a region of validity called the scope of that name.

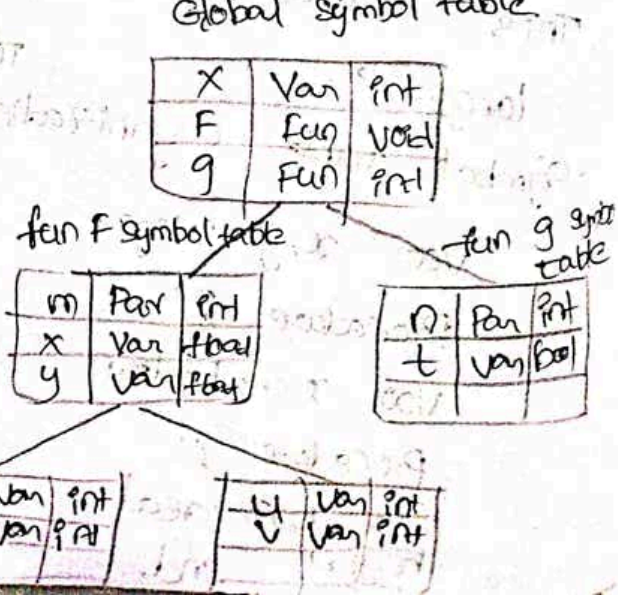
The rules in a block structured language are as follows.

1. If a name declared within blocks B, then it will be valid only within B.
2. If B_1 is nested within B_2 then the names that is valid for B_2 is also valid for B_1 Unless the names identifier is redeclared in B_1
3. The scope rules need a more complicated organization of symbol table than a list of associations between names and attributes.

when ever a new ~~tabl~~ block is entered then a new table is entered onto the stack. The new table holds the name that is declared as local to this blk

Eg:

```
int x;
void F(int m)
{
    float x, y;
    { int i, j;
      int u, v;
    }
int g(int n)
{ bool t;
```



Symbol table datastructure

The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data form that record quickly.

Linked list

* A linear list of record is the easiest way to implement symbol table.
* The new names are added to the symbol table in the order ~~the~~ they arrive.
* when ever a new name is to be added first it searched linear to check it the name is already present in table or not.

Time Complexcity – O(n).

Advantage – less space, additions are simple

Disadvantage – higher access time.

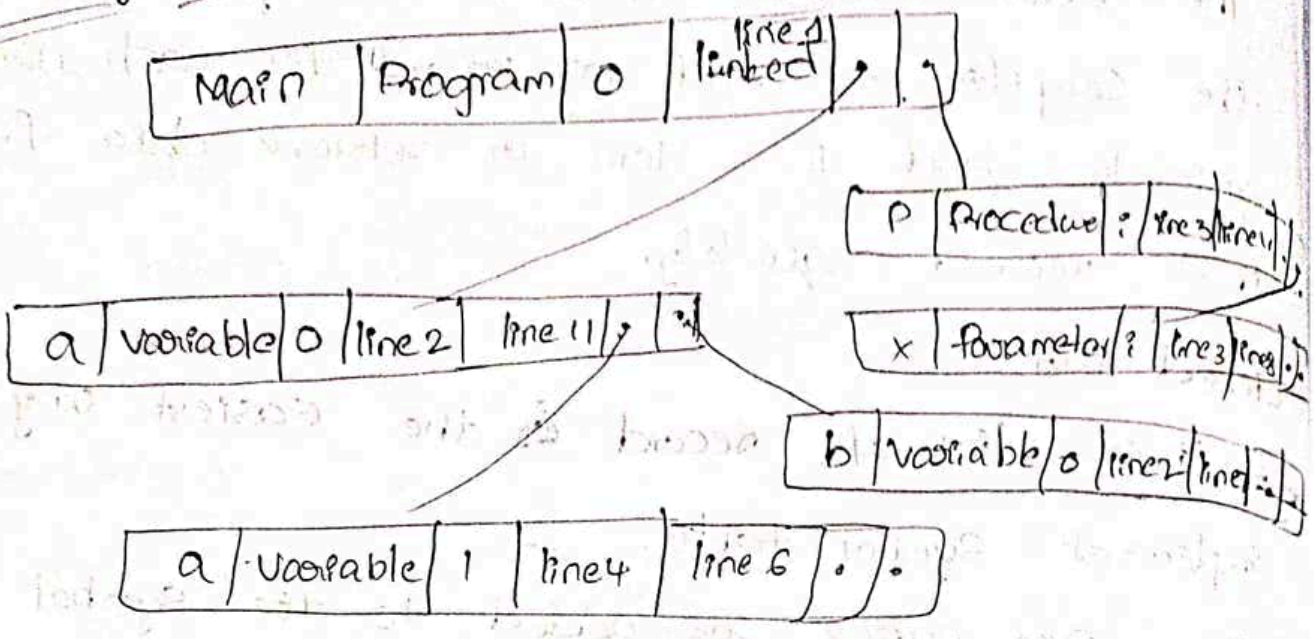
2. Binary tree

Efficient approach for symbol table organization we add two links left & right in each record in the search tree.

If it does not exists then a record for new name is created & added at Proper Position.

This has alphabetical accessibility.

2. Binary tree



3 Hash table

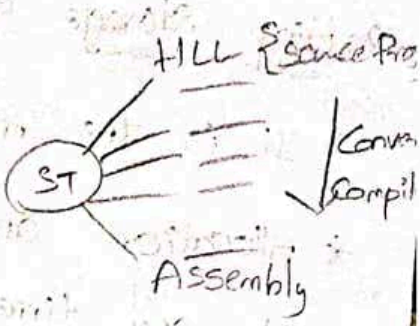
* In hashing scheme two tables are maintained. a hash table and a symbol table.
* A hash table is an array with index range 0 to table size - 1. These entries are pointers pointing to names of symbol table.
* To search for a new name we use hash function that will result in any integer between 0 to table size - 1
* Insertion & lookup can be made very fast - O(1)

Prin

Storage Allocation.

The different ways to allocate Memory are.

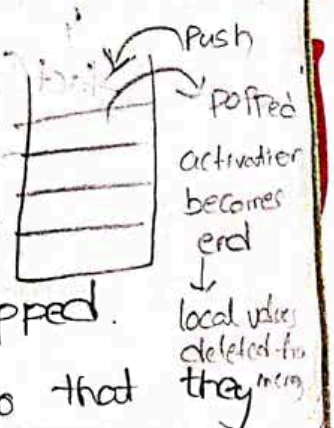
1. Static Storage Allocation.
2. Stack Storage Allocation
3. Heap Storage Allocation.



1. Static Storage Allocation.

* In static allocation, names are bound to Storage locations
* If Memory is created at compile time then the memory will be created in static area at only once
* Static allocation support the dynamic data structure that means, memory is created only at compile time and deallocated after program Completion.
* The draw back with static storage allocation is that the size and position of data objects should be known at compile time.
* Another drawback is restriction of the recursion Procedure.

2. Stack Storage allocation.



* Storage is organized as a stack LIFO
* Activation records are pushed and popped.
* Activation record Contains the locals so that they are bound to fresh storage in each activation record
* The value of locals is deleted when the activation ends
* It works on the basis of LIFO and this allocation

Supports the recursion process.

3. Heap storage allocation {dynamic Nature}

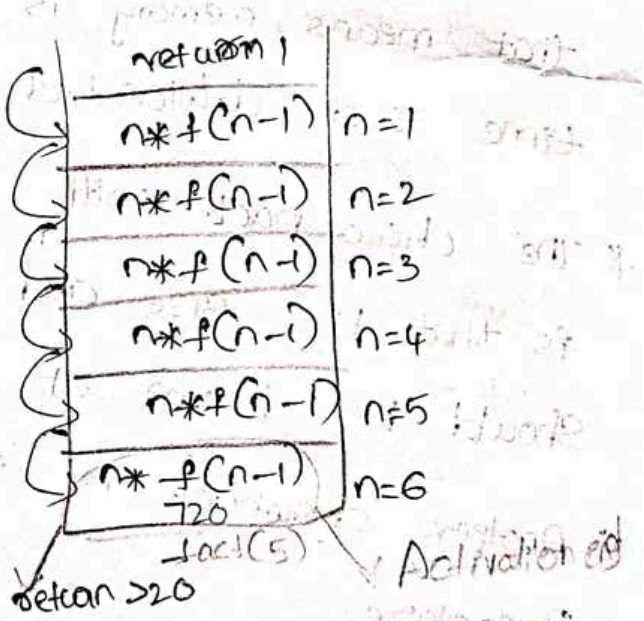
* It is the most flexible allocation scheme.
* Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirements

Advantage

* Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
* Heap storage allocation supports the recursion Process.

Example

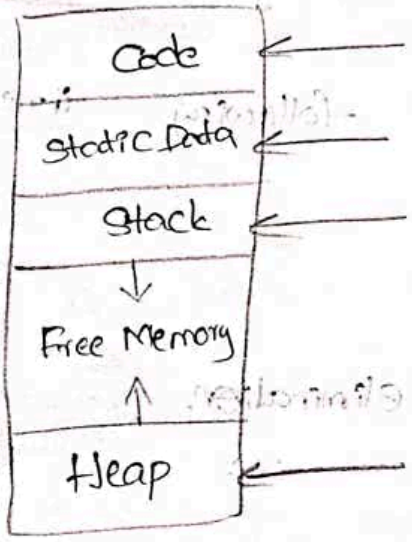
```
fact (int n)
{
  if (n<=1)
    return 1;
  else
    return (n*fact(n-1));
}
fact(6)
```



Storage Organization

* The executing target Program runs in its own logical address space in which each Program value has a location.
* The management and organization of this logical address space is shared between the Compiler, Operating System and target machine.
* The Operating System maps the logical address into physical address, which are usually spread throughout memory.

Sub-division of Runtime Memory



Memory location for code are determined at compile time

Locations of static data can also be determine at compile time

Data objects Allocated at Run-time (Activation Records)

Other Dynamically Allocated objects at Run-time for Eg: Malloc Area in C.

* Runtime Storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the four bytes a machine word can form.
* Object of multibyte is stored in consecutive byte and gives the first byte address.
* Runtime Storage can be subdivide to hold the different components of an executing Progra-

1. Generate Executable code
2. Static data objects
3. Dynamic data object - heap
4. Automatic data object - stack.

Loop optimization.

It is most valuable machine - independent optimization because programs inner loop takes bulk of time of a programmer,

If we decrease the number of instructions in an inner loop then the running time of a program may be improved. Even if we increase the amount of code outside the loop.

For loop optimization the following three techniques are important.

1. Code motion
2. Induction variable elimination.
3. Strength reduction.

loop
{
}

1. Code motion

It is used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

Eg: while (i<=limit-2) // stmt does not change limit.

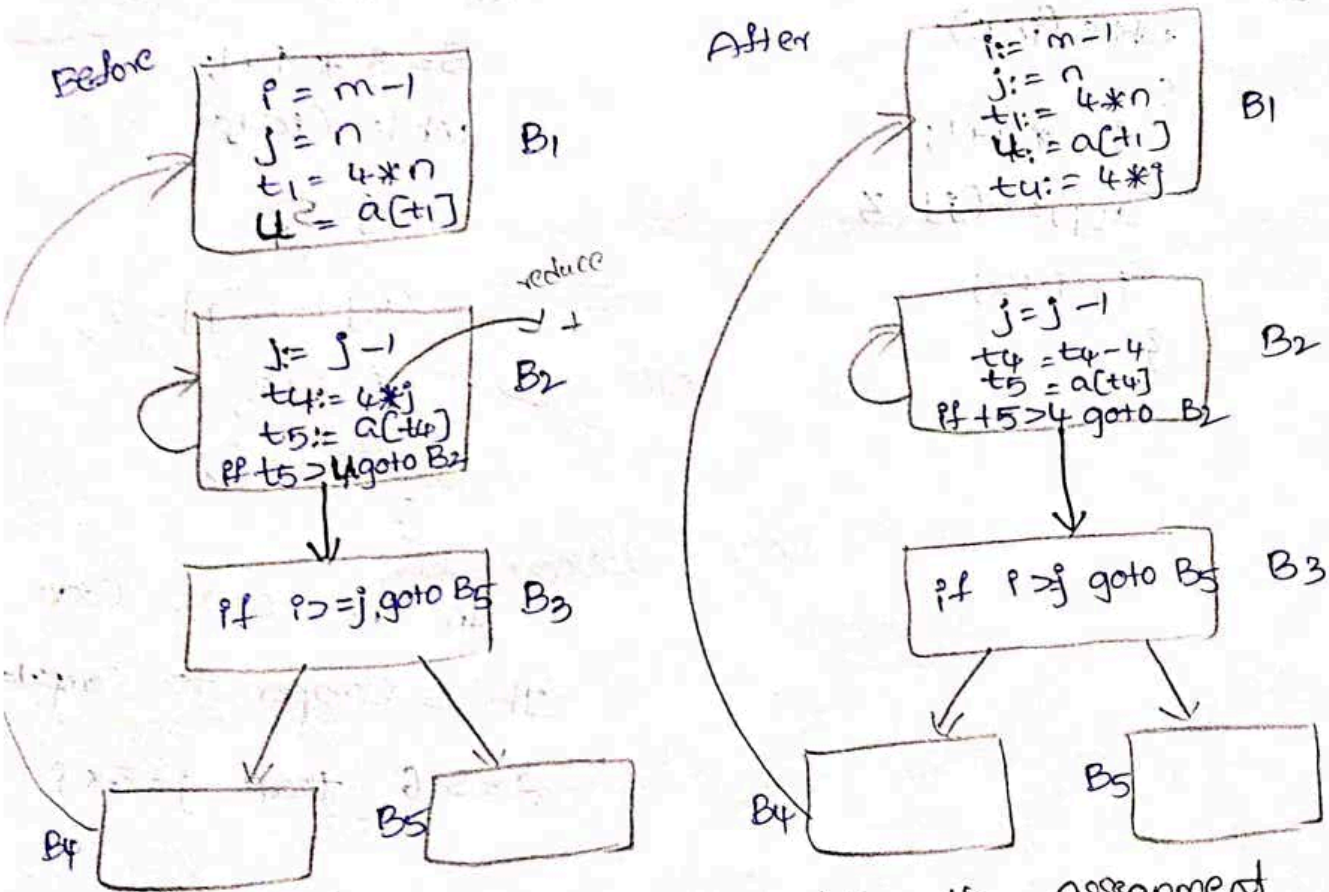
After code motion the result is as follows.

```
a= limit-2;
while (i<=a)   // stmt does not change limit or
```

In this while stmt, the limit-2 equation is a loop invariant equation.

2. Induction - variable Elimination.

It is used to replace variable from inner loop It can reduce the number of additions in a loop. It improves both code space and runtime Performance.



In this figure we can replace the assignment $t4 := 4*j$ by $t4 := t4-4$. The only problem which will be arose that t4 does not have a value when we enter block B2 for the first time. So we place a relation $t4 = 4*j$ on entry to the block B2.

3. Reduction in Strength

* Strength reduction is used to replace the expensive operation by the cheaper once on the target machine.
* Addition of constant is cheaper than a multiplication. So we can replace multiplication with an addition with the loop.
* Multiplication is cheaper than exponent. So we can replace exponent with multiplications with in the loop

Ex > mul > add.

Eg:

Before

```
while (i<10)
{
  j = 3*i+1;
  a[j] = a[j] - 2;
  i = i+2;
}
```

After Strength reduction the code will be:

```
S = 3*i+1;
while (i<10)
{
  j = S;
  a[j] = a[j] - 2;
  i = i+2;
  S = S+6;
}
```

In the above code It is cheaper to compute S = S+6, than j = 3*i

Principle Sources of optimization

Optimization is classified into two types

1. Machine independent
2. Machine dependent.

1. Machine independent optimization

Machine Independent are program transformations that improve target code, without taking into consideration any properties of target MC

2. Machine dependent optimization

This optimization based on : register allocation. utilization of special machine instruction sequence

Code optimization techniques

1. Compile time Evaluation.
2. Variable Propagation
3. dead code elimination
4. Code motion.
5. Induction variable and strength reduction

1. Compile time Evaluation.

(a) $z = \{5 \times (45.0 / 5.0) \times r\}$

Execute at Compile time

(b) $x = 5.7$

$y = x / 3.6$

The compile time evaluate $\left\{\frac{5.7}{3.6}\right\}$

2. Variable propagation

Before Optimization.

```
c = a x b
x = a
till
d = x * b + 4
```

After optimization

```
c = a x b
x = a
till
d = a * b + 4
```

3. Dead Code elimination.

Before elimination.

```
c = a * b
x = b  => dead state
till
d = a x b + 4
```

After elimination

```
c = a x b
till
d = a x b + 4
```

4. Code Motion.

* It Reduces the evaluation frequence of expression.
* It brings loop invariant Stmt out of the loop

Eg:

```
a = 200;
while (a > 0)
{
    b = x + y;
    If (a % b == 0)
    Printf("%d", a);
}
```

```
a = 200;
b = x + y;
while (a > 0)
{
    If (a % b == 0)
    Printf("%d", a);
}
```

5. Induction variables & Strength reduction.

Strength reduction.
To Replace high Strength Operator by low Strength Operator.

Induction variable.

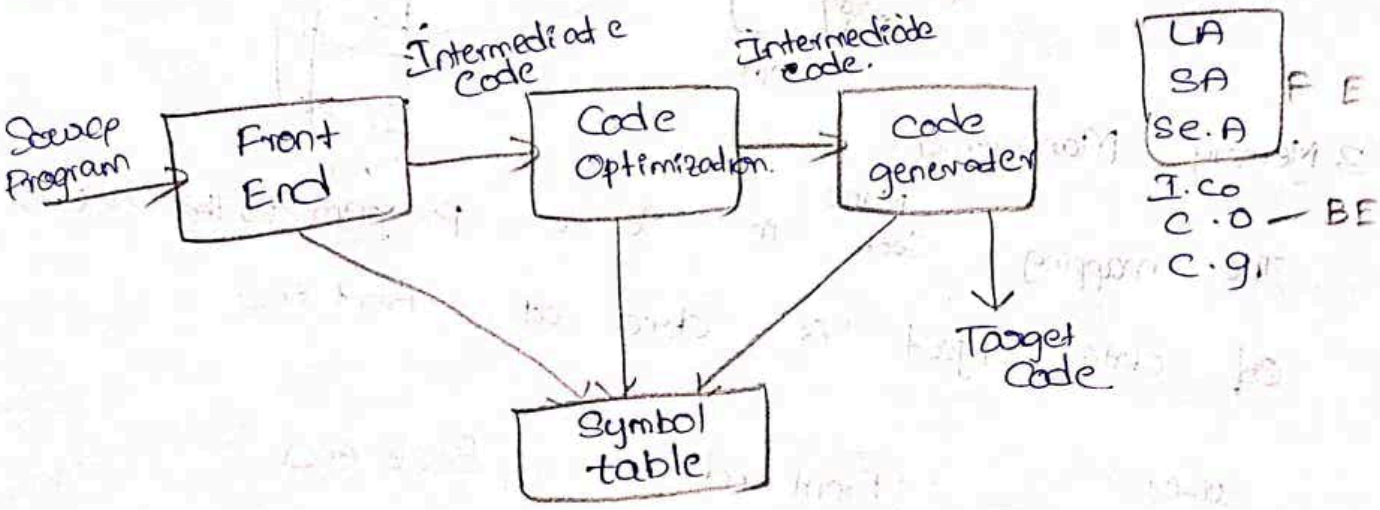
Eg:

Before reduction Code.	After reduction.

i=1;
while (i<10)
{
y = i * 4;
}
	i=1;
	t=4;
	while (t<10)
	{
	y= t;
	t = t+4;
	}

*>+

Issues in the Design of Code generator.



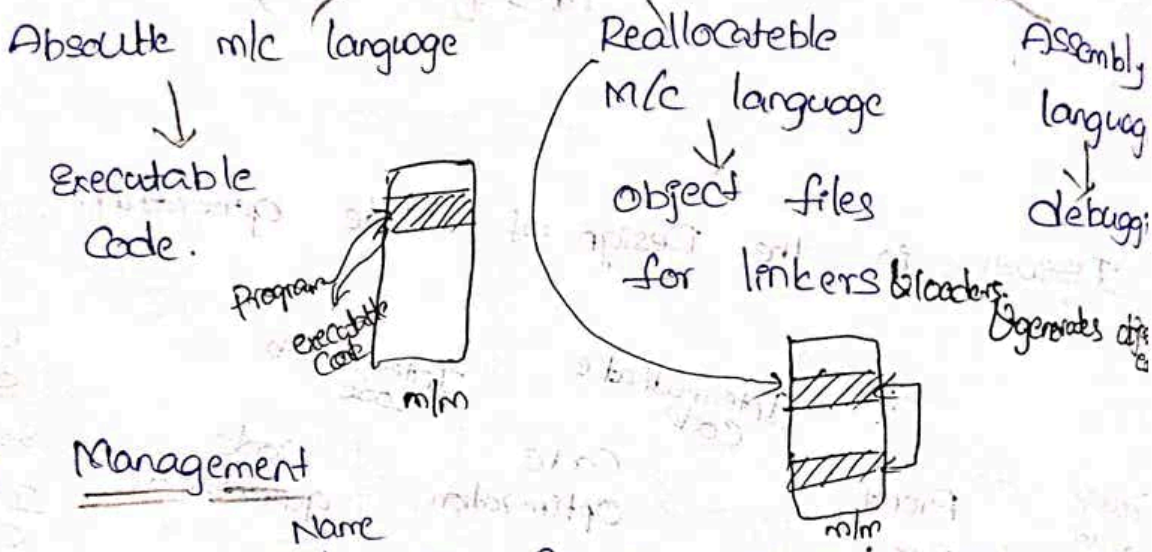
1. Input to Code generator
2. Target Program
3. Memory - Management
4. Instruction Selection.
5. Register allocation issues.
6. Evaluation order

1. Input to Code generator : You will get {Intermediate}

* It is linear representation like : Post-fix Notation and TAC or DAG (Direct Acyclic Graph)
* The I/P is free of errors by performing the type checking.

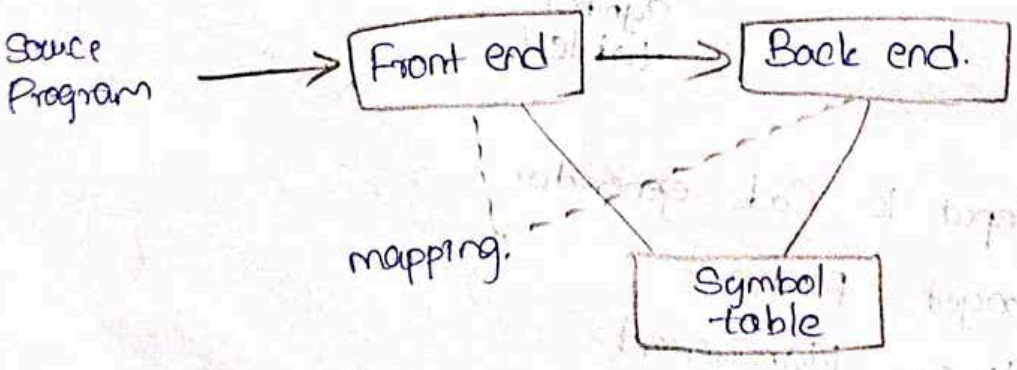
2. Target Program {output}

The obtput may be: Many forms.



3 Memory Management

The mapping Name in Source Program to the addres of data object is done at front end.



with the help of relative address we can find the data

All the Information Stored in Symbol -table.

4. Instruction selection

Code generator takes Intermediate code as i/p and Convert into target machine instruction set.

It is the responsible for code generator to choose appropriate instruction.

The quality of the generator code is determined by its speed and size.

Eg: $x = y + z$

```
LD  R0, y
ADD R0, R0, z
ST  x, R0
```

a = b+c	a=b+c
MOV b, R1	MOV b, R1
ADD C, R1	MOV C, R2
MOV R1, a	ADD R1, R2
	MOV R2, a

} increases the execution time & memory required

5. Register Allocation

what value to hold in ; what register? → key problem

Instruction involving

1. register operands — fast
2. Memory operands — larger and slow.

The Register is subdivided into two sub problems.

Register Allocation	Register Assignment
during which we select set of variables that will resides in register at a point in program	during which we pick specific Register that a variable will reside in SP

6. Evaluation order

The order in which the computations are performed can affect the efficiency of the target code. When instructions are independent that evaluation order can be changed.

Eg: $(a+b) - (c+d) * e$

reorder

Three Address Code:

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

$\Rightarrow$

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

```
MOV  a, R0        MOV  c, R0
ADD  b, R0        ADD  d, R0
MOV  R0, t1       MOV  e, R1
ADD  d, R1        MUL  R0, R1
MOV  e, R0        MOV  a, R0
MUL  R1, R0       ADD  b, R0
MOV  t1, R1       SUB  R1, R0
SUB  R0, R1       MOV  R0, t4
```

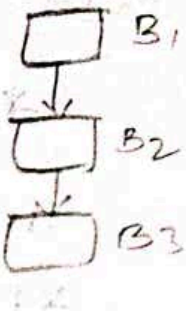
Register allocation

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

```
MOV  a, R1        MOV  a, R1
MOV  b, R2        ADD  b, R1
ADD  R1, R2       MUL  d, R1
MOV  R2, t1  ->   MOV  R1, t1
MUL  d, t1        optimized one
```

Global Data flow Analysis

It collects the Information about entire prog and distributed this information to each block in the flow graph.



⇒ A typical data flow equation.

$$out[S] = gen[S] \cup [in[S] - kill[S]]$$

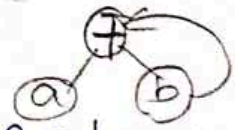
out[S] → Definitions that reach B's exist.

gen[S] → definition within Block that reach the end of Block.

in[S] → definition that reaches B's entry.

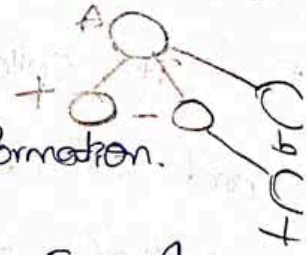
kill[S] → definition that Never reach the end of Block.

DAG Representation



* DAG Stands for Directed Acyclic Graph
* Syntax tree and DAG both are graphical representation Syntax tree does not find the Common Sub expression where as DAG can.
* Another Usage of DAG is the application of optimization technique in the basic block.
* To apply optimization technique on basic block DAG is Constructed three address code. which is the output of an intermediate code generation.

Algorithm for Construction of DAG : Basic Block



Input - It Contains a basic block.

Output - It Contains the following information.

* Each Node Contains a label. For leave the label is an identifier.
* Each node can contains a list of attached identifiers to hold the computed values

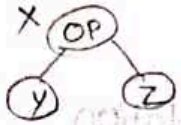
Case(i) $x := y \text{ op } z$

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

Case (iii) $x := y$

Step 1 : 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) and left child is node (y)

$X = Y \text{ OP } Z$

For Case (ii), check whether there is node(op) with one child node (y) $X = \text{OP } Y$

For Case (iii), node n will be node (y) $X = Y$

Output: For node (x) delete x from the list of identifiers

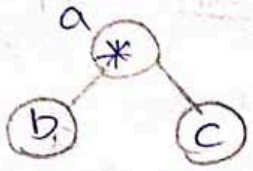
Append x to attached identifiers list for the node n found in Step 2. finally set

node (x) to n

DAG Example

Example
$a = b * c$
$d = b$
$e = d * c$
$b = e$
$f = b + c$
$g = f + d$ ~~d~~ ~~f~~

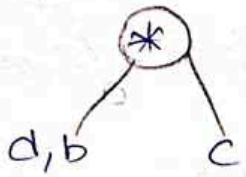
Step 1 : Consider the first statement $a = b * c$



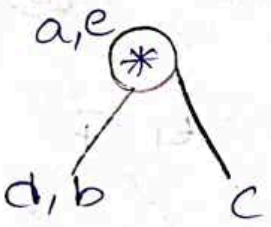
Append Result variable a to node c

Step 2 : Take 2nd statement append the d value.

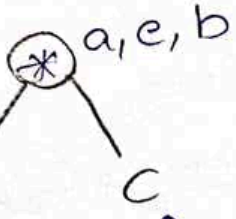
$d = b$



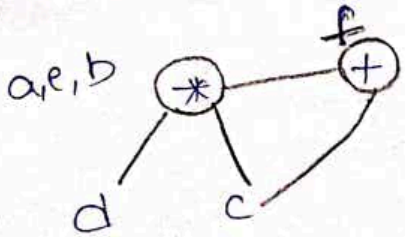
Step 3 : The Node $d * c$ already created.



Step 4 : $b = e$ append the b node to e

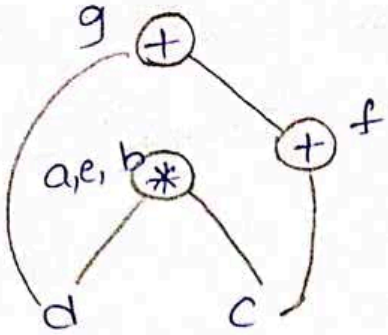


Step 5:



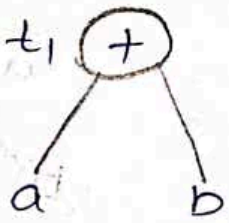
$f = b + c$

Step 6: g = d + f



Example 2 (a+b) * (a+b+c)

Step1

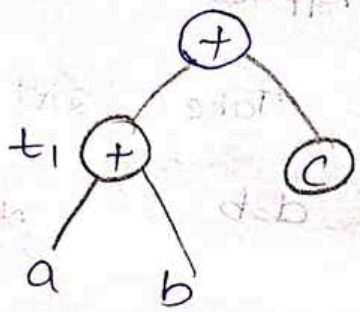


TAC ⇒ $t_1 = a+b$

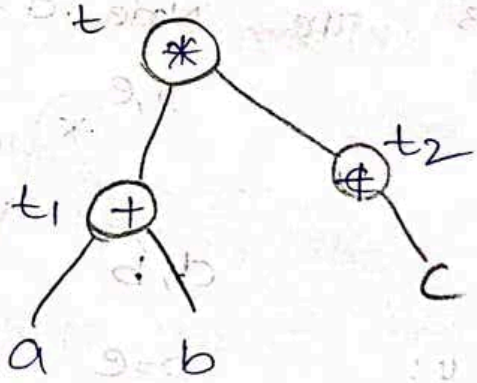
$t_2 = t_1 + c$

$t_3 = t_1 * t_2$

Step2



Step 3



Peephole Optimization

* This technique works locally on source code to transform it into optimized code.
* The peep hole optimization is a short sequence of target instruction that can be replaced by shorter or faster sequence instruction.
* It examine at most a few instruction transforming instruction into other less expensive ones such as turning multiplication of x by 2 into an addition of x with itself.

$$x \times 2 \Longleftrightarrow x + x$$

$$2x \Longleftrightarrow 2x$$

Characteristics of Peephole optimization.

1. Redundant inst elimination
2. Unreachable Code.
3. Flow of Control optimization.
4. Algebraic Simplifications

1. Redundant instruction elimination.

The Source Code level following can be done by the User.

1)
```
int add_ten(int x)
{ int y, z;
  y = 10;
  z = x + y;
  return z;
}
```

2)
```
int add_item(int x)
{
  int y;
  y = 10;
  y = x + y;
  return y;
}
```

3)
```
int add_item(int x)
{
  int y=10; → its constant
  return x+y;
}
```

4)
```
int addfitem(int
{
  return x+10;
}
```

2. Unreachable Code

It is a part of Program Code that is never accessed because of Program Constructs.

Programmers may have accidently written a piece code that can never be reached.

Eg:
```
void add_item(int x)
{
  return x+10;
  Printf("value of x is %d", x);
}
```

In this Stmt Printf Stmt will never execcded as Program Control returns back before it execcde, hence Printf can be removed.

3. Flow of Control Optimization

These are instances in a code where the Program Control jumps back & forth without Performing any significant task, these Jumps can be removed.

eg: MOV R1,R2

Goto L1

- - - -

L1 : goto L2 ×

L2 : INC R1

- - - -

- - - -

Unconditional jump

In this example code L1 can be removed as it passes the control to L2. So, instead of jumping to L1 & then to L2, the control directly reach L2.

```
MOV R1,R2
Goto L2
- - - -
L2 : INC R1
```

4. Algebraic Simplifications

There are occasions where algebraic expression can be made simple.

for eg: a = a + 0 // can be replaced by a itself

a = a + 1 // can simply by replaced by INC a

⇓

INC a ⇒ a = a + 1

As better write ←

A simple code generator Algorithm.

- It generates target code for a sequence of instructions
- It uses a function getReg() to assign registers to variables.
- It uses 2 data structures.
  1. Register Descriptor
  2. Address Descriptor.

1. Register Descriptor: Used to keep track of which variable is stored in a register. Initially all registers are empty.

2. Address Descriptor: Used to keep track of location where variable is stored. location may be register, memory address, stack .... etc.

The following actions performed by Code generator for an instruction $x = y \text{ OP } z$ assumes that L is the location where the output of y OP z is to be saved.

1. Call the function getReg() to get the location of L
2. Determine the present location of 'y' by consulting Address descriptor of y. If y is not present in location 'L' then generate the instruction.

   mov y', L be Copy value of y to L

3. One Present location of z is determine using step2 & the instruction is generated as OP z', L

4. Now L contains the value of y OP z i.e Assigned to x. So, If L is a register then update its descriptor that it contains is stored in 'L'

5. If y,z have no future use, then update the descriptors to remove y & z.

Eg: d=(a-b)+(a-c)+(a-c)

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

Statement	Codegeneration.	Register Description.	Address Descriptor
$t_1 = a-b$	mov a, R0 Sub b, R0	R0 Contains t_1	t_1 in R0
$t_2 = a-c$	mov a, R1 Sub c, R1	R0 Contains t_1 R1 Contains t_2	t_1 in R0 t_2 in R1
$t_3 = t_1 + t_2$	Add R1, R0	R0 Contains t_3 R1 Contains t_2	t_3 in R0 t_2 in R1
$d = t_3 + t_2$	add R1, R0 mov R0, d	R0 Contains d	d in R0

Heap Management

* Heap is used in Dynamic memory Allocation.
* Memory manager allocates two basic functions
  * Allocation - If A[5] is 5 ()
  * Deallocation - A[100] is used 5 95 is wasted

Properties of Memory Manager.

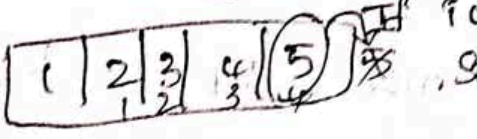
* Space efficiency.
  minimize the Heap space required by a Program
* Program efficiency.
  Better use of Program to run faster using less memory
* Low overhead.
  Allocation & Dellocation should be efficient.

Advantages of Heap.

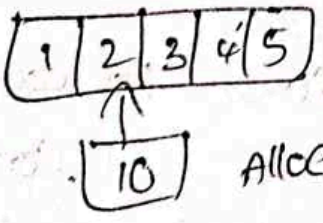
Dynamic.

Alloc deAlloc

dynamic.



Heap.



Allocate order node & link it

Use the Pointers