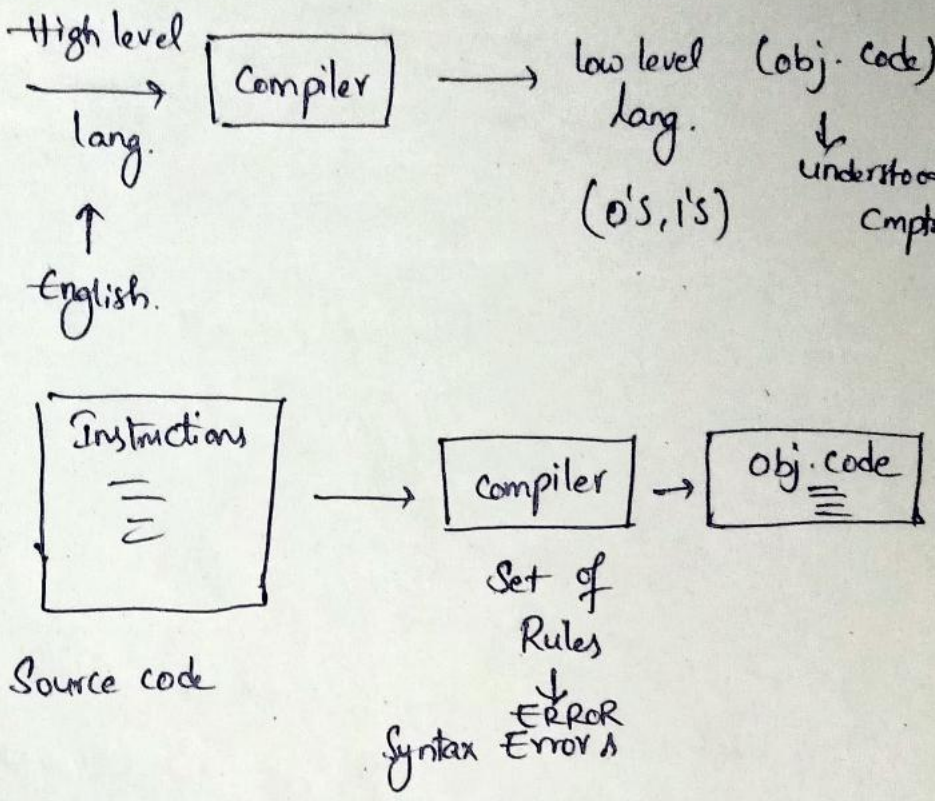
Intrepreter
Line by line one Error

Language procesing System overview:-

Compiler (Design) (Language Translator) - one lang. to another lang.



Steps of executing a program:-

Computer — combination of Software & H/w Compont.

S/w ① App. S/w : user Develop a Pgm then it is app. S/w. eg: + of 2 no.s,

② System S/w : using system S/w we can work with Systems (or) Computers eg: OS, Compilers, Debuggers, unix os, Assembler, Driver files etc.

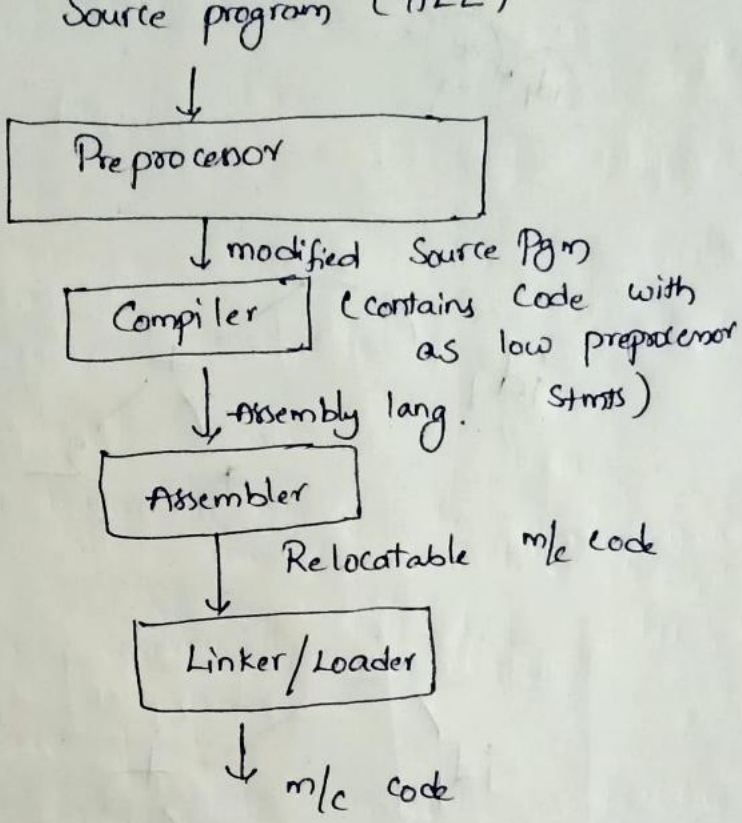
Pgming langs :- ① High Level Pgmin Langs. ② Low " " " → 0's & 1's m/c code. ③ Middle " " " * Develop System S/w

App. S/w { C, C++, Java, C# } Alphabets, digits, symbols

combines low + high. (unix os) eg: C → Develop both high & low level lang

low level lang. Eg: Assembly code (ds, is).

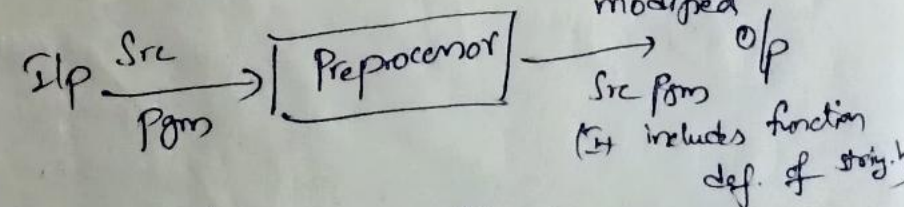
Lang. processing system steps:-



Preprocessor:- The stmts which are processed before compilation are known as Preprocessor stmts.

Eg: Printf(), clrscr(), Strlen()
functions.
stdio.h, conio.h, string.h

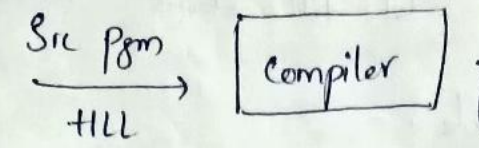
* It Must begins with # function.



(1) file inclusion: A file is included
#include <file.h>
" <stdio.h>
" <conio.h>

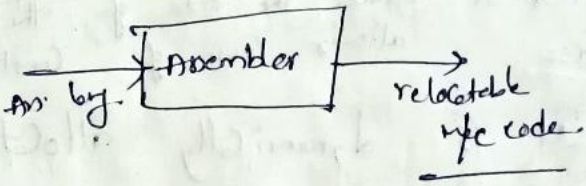
(2) MACRO - Define symbolic constants
#define PI 3.14 — Variable/Constant

Compiler:- Translator (eg: C)



Compiler is a procen of checkin whether a Pgm is following lang. Syntax rules or not. (Error reports to user)

Assembler translates Src Pgm written in Assembly lang. to a tgt Pgm in relocatable m/c code.



{Linker/Loader : It accepts m/c code as i/p & links obj. code of various lib. fnctns to main Pgm. }.

Linker :- It links the relocatable m/c code of Various lib. files to main Pgm and combins them to produce as executable file.

eg: Pf, clrscr, strlen
obj obj obj. code linked main Pgm & makes .exe file.

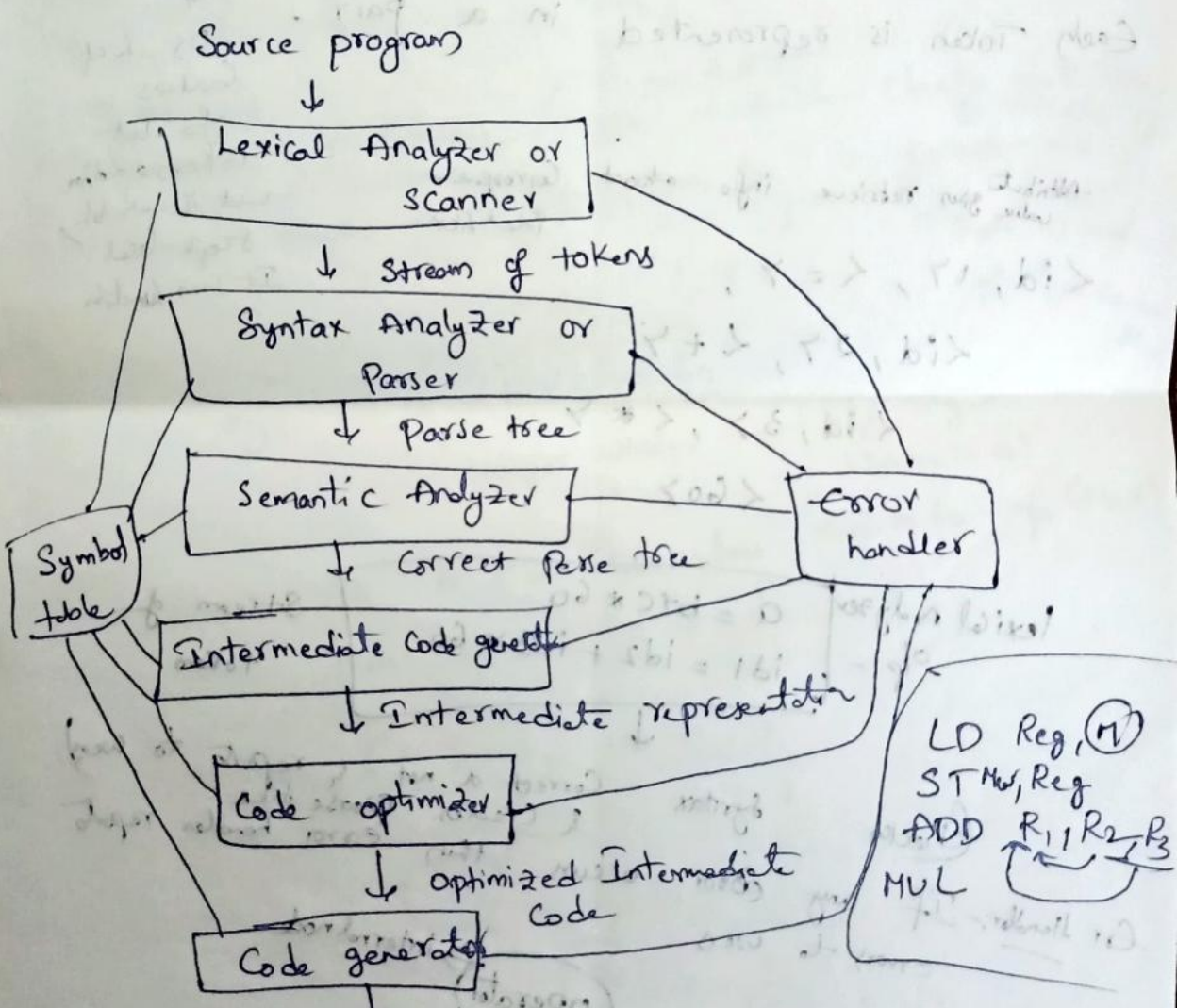
Then Loader loads .exe file to memory for execution. CPU executes our Pgm.

Relocatable m/c Code :-

In OS, Paging & Segmentation Concepts,

1. Addr :- Logical – The Pgm generated addr.
2. Physical – Addr. available in memory (main)
3. offset. { logical + offset } in physical addr.
   relocatable addr.

Phases of a Compiler :-



Lexeme → A Sequence of characters.

a = b + c * 60

Token → A keyword & identifier & variable
operator & Assignment
for while { C, C++, Java Constant

Lexeme	Token
a	identifier
=	Assignment operator
b	"
+	Addition "
c	"
*	Multiplication "
60	Constant

<tokenname, attribute value>

↓ info stored in Symbol Table

↓ DS which Contains info about TOKens & Identifiers what is variable, Scope-local It uses function

Each Token is represented in a pair.

Attribute value gives retrieve info. about Corresponding identifier

<id, 1>, <=>,

<id, 2>, <+>,

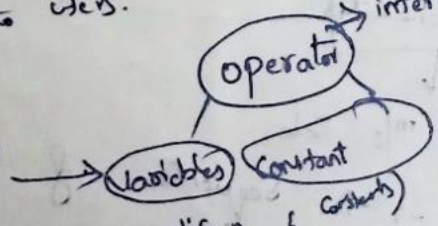
<id, 3>, <*>

<60>

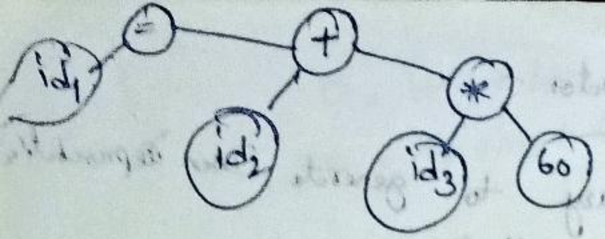
lexical Analyzer o/p:- a = b+C*60 / id1 = id2 + id3 * 60 → Stream of tokens

Checks syntax & Correct a not & reports to user & Creates Parse tree then

Err Handler:- If any error occurs error handler reports errors to users.



out of + & * is * Highest priority.



(operator have & priority)

Parse tree

Syntax Analyzer: It checks whether tokens are following lang. syntax rules ~~then~~ or not.

If not lang. syntax rules then error handler comes into existence.

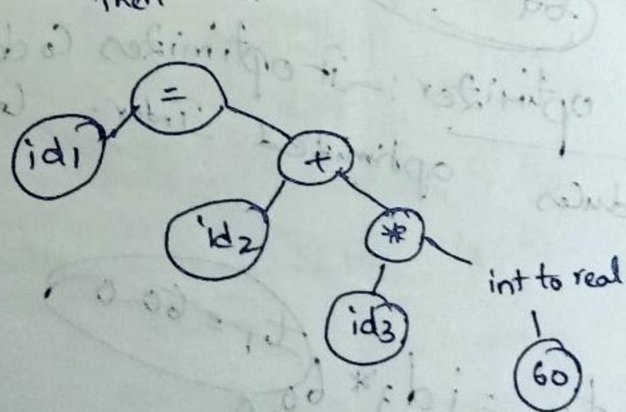
If follows rules, then generates parse tree.

(Meaning)
Semantic Analyzer:- (Type checker) Tool or Software ↳ checks data types of variables if is correct or not.
↳ checks meaning of parse tree

There is need of type conversion

eg:- a = b + c * 60 — o/p integer. (stores in a)
integer variables
then * 60 - o/p (float)

If b, c - float values then a is float. So.



Intermediate code generator:-

* It is easy to generate Inter Representation

* Many ways (notations)

Three Address Code:-

(1) = operator. The instr should contain 3 addr

(2) temp. Variable

(3) Some instructions must contain fewer than 3 operands & addr

(1) A three addr code assign instr should have one at most right operator max. of one

at at. hand side.

(2) Compiler must generate a temp. var. to store results.

Inter Representation:-

t_1 = intoreal (60) zero " operator
float one "

t_2 = id_3 * t_1 "

t_3 = id_2 + t_2 zero "

id_1 = t_3

~~t_4 = id_1~~

Code optimizer:- it optimizes code & produces optimized Inter. Code.

~~t_1 = 60.0~~

$t_1 = id_3 * 60.0$

~~t_2~~

$id_1 = id_2 + t_1$

LD Reg, M
ST M, Reg
ADD R_1, R_2, R_3
MUL

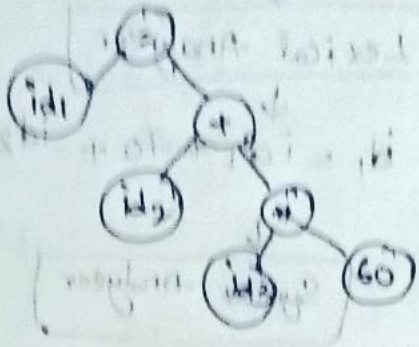
floating point number

LDF R_2, id_3
MULF R_2, R_2, #60.0
LDF R_1, id_2
ADDF R_1, ...

STF id_1, R_1

a = b + c * 60

$id_1 = id_2 + id_3 * 60$



t_1 = into real (60)

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

$id_1 = t_3$

$t_1 = id_3 * 60.0$

$id_1 = id_2 + t_1$

```
LDF   R2 , id3
MULF  R2, R2, 60.0
LDF   R1 , id2
ADDF  R1, R1, R2
STF   id1, R1
```

Q. Consider the following fragment of 'c' code:

```
int i, j;
i = i * 10 + j + 2;
```

write the o/p of all phases of compiler for the above code.

$i = i * 70 + j + 2$

↓

Lexical Analyzer

↓

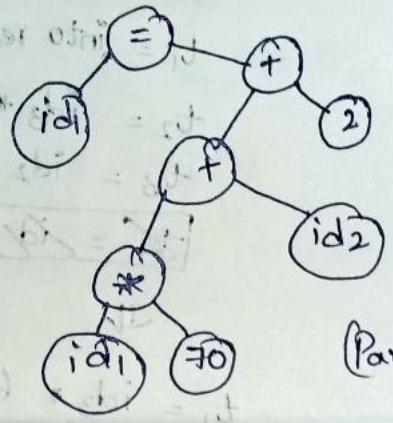
$id_1 = id_1 * 70 + id_2 + 2$ (Tokens)

↓

Symbol table

1	id	...
2	id	...
	...	

Syntax Analyzer

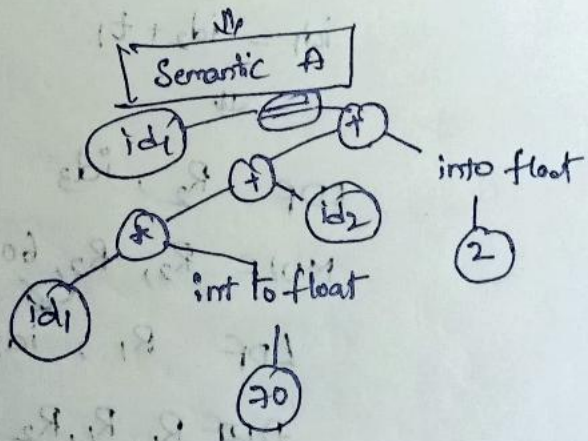


(Parse tree).

Assume i, j - float.

70 - 70.0

2 - 2.0



Intermediate code generator.

(1) max. one operator

(2) Temp_ more results

(3) store instru. < 3 opn

$t_1 = \text{intofloat}(70)$

$t_2 = id_1 * t_1$

$t_3 = id_2 + t_2$

$t_4 = \text{intofloat}(2)$

$t_5 = t_3 + t_4$

$id_1 = t_5$

Code optimizer

↓

$t_1 = id_1 * 70.0$ — R_1

$t_2 = id_2 + t_1$

$t_3 = t_3 + 2.0$

$id_1 = t_3$

↓

```
LDF   R1 , id1
MULF  R1 , R1 , #70.0
LDF   R2 , id2
ADDF  R1 , R1 , R2
LDF   R3
ADDF  R1 , R1 , #2.0
STF   id1 , R1
```

(v) Intermediate Code :-

The code may be kept as an array of text strings, a linked list of structures, or a temp text file based on intermediate Code type.

(vi) Temporary files :-

To store intermediate Code we use these temp files.

Issues in Compiler structure :-

i) Analysis & synthesis :-

There are 2 phases of Compilation.

(a) Analysis : It breaks up the Source Program into Constituent pieces & Create an intermediate representation of the Src Pgm.

(b) Synthesis : It Constructs the desired target Program from the intermediate representation.

(major impacts on it are reliability, efficiency, usefulness & maintainability)

ii) Front & Backend :-

f.t : The operations depends only on the Src lang.

B.t : The operations depends on generating the target Code.

Scanner, Parser & semantic analyzer
front end
code generator back end code optimization
in both intermediate code is the medium.

(iii) Passes:-

It is nothing but the repetitions for
Processing the entire src pgm several times before
generating final code.

for each we end up representation
some data in diff. format.

If a compiler is following one pass
it is less efficient. when compiled to
more than one pass compiler.

Major DS in a Compiler:-

Symbol Table :- It stores info about prm identifier that will be accessed all phases.

* The first 3 phases helps in filling up the Symbol Table.

* In lexical analyzer phase, the first phase interact with Symbol table & Symbol table is created by the lexical analysis.

* A symbol table is a data structure that contains a record for each identifier with fields for the attributes of the identifier.

name of variable

* Symbol table has facilities to manipulate (add/delete) an element.

eg:- x = a + b * 20

Symbol	Category	type	Attribute	memory location
x	identifier	float	id1 assigned	1000
=	operator			
a	id	float	id2 added	1004
+	op			
b	id	float	id3 multiplied	1008
*	op			
20	Constant	int	id4	

Continue to previous page

Scanner / lexical analyzer, Parser & Semantic analyzer helps to fill above table.

ii) Literal table :-

* It is also a ds used in Compiler if it is a part of symbol table.

* It maintains the details of Constants & Strings

literal	Category	attribute	memory	location
60	Const	int	1000	

* It reduces the size of the pgm in memory by allowing reuse of Constants & strings.

Parse tree :

* It is dynamically allocated pointer based structure.

* It describes the syntactic structure of i/p.

* It Parse tree, the terminal nodes represent the tokens & interior node represent the expressions & work (bottom up).

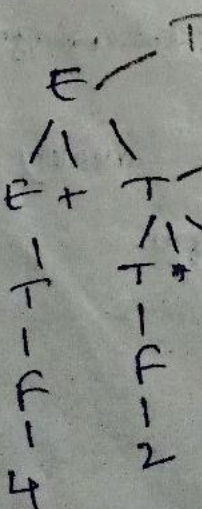
Expression :- 4 + 2 * 3

Grammar

$E \to E+T / E-T / T$

$T \to T * f / f$

$f \to a / (E)$ no. of variable

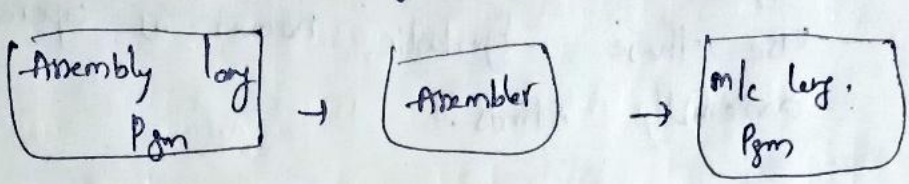


Abstract Syntax Tree

Syntax tree :-

* Compressed version of Parse tree.

Assembler:- It is a system s/w which translate, Pgms written in assembly lang into m/c lang.



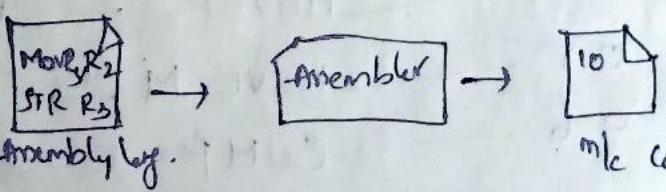
Assembly lang:- It is a kind of low level Pgming lang, which uses symbolic codes or mnemonics as instruction.

* Some examples of mnemonics include ADD, SUB, LDA, & STA that stand for addition, substraction, load accumulator, store accumulator respectively.

Applications of Assembly lang:-

* It is used for H/w manipulation, access to specialized processor instructions, or to address critical performance issues.

* Typical uses are device drivers (CD, HDD), low-level embedded systems (Keyboard, water tank indicator) & real-time systems (Computer, notepad).



Elements of Assembly lang:-

(1) Mnemonic Operation Code: It is used for m/c instructions (also called mnemonic opCodes) are easier to remember & use than numeric operation codes. Eg: ADD, SUB, MOV

(2) Symbolic operands:- A programmer can associate Symbolic names with data or instructions & use these symbolic names as operands in assembly stmts.

Eg:- ADD R_1, R_2, R_3

(3) Data Declaration:- Data can be declared in a variety of notations including the decimal notation.

Eg:- NUM1 03 OR
NUM1 0011H

Instruction op code	Assembly mnemonic Instructions.	Remarks.
00	STOP	Stop execution
01	ADD	Addition
02	SUB	Subtraction
03	MULT	Multiplication
04	MOVER	Move memory to Register
05	MOVE M	Move Register to memory
06	COMP	Comparision
07	BC	Branch on Condition
08	DIV	Division
09	READ	Read memory
10	PRINT	writing to memory.

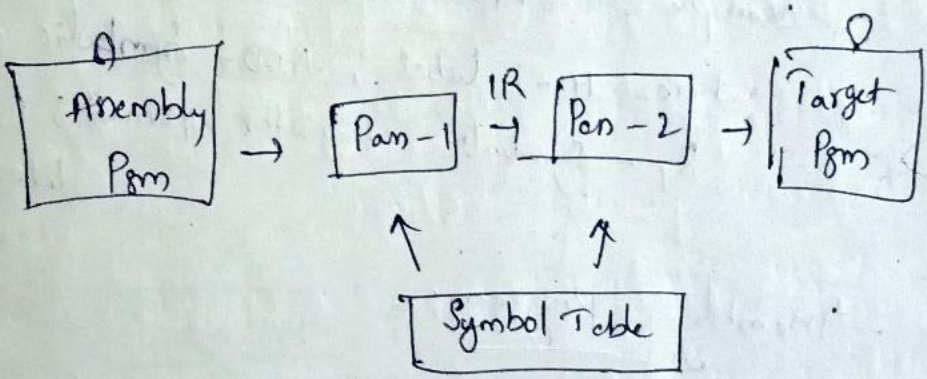
Assembly process :-

Convert .asm file into .obj file.

Pass 1 : Complete Scan .asm file. find all labels, instructions & calculating corresponding address.

Pass 2 : Convert all the instructions into m/c lang. format.

Symbol Table : Stored all the info. of assembly lang. data, variables, instructions, address etc,



Example of Assembly lang. with Assembler Directi

START : This instruction starts the execution of Progra from location 200 & label with START provides name for the Pgm (JOHN is name of Pgm).

MOVER : It moves the content of literal (='3') into registers operand R1.

MOVEM : It moves the content of register into memory operand (x)

MOVER : It again moves the content of literal (='2') into register operand R_2 and its label is specified

LTORG: It assigns address to literals (Current LC Value)

DS (DataSpace): It assigns a data space of 1 to symbol X.

END: It finishes the Pgm execution.

[Label] [op code) [operand).

Example: M ADD R_1, ='3'

where M - label; ADD - symbolic opCode; R_1 - symbolic Register operand; (='3') - literal.

Assembly Programs:-

Label	op_code	operand	LC value (Location counter)
JOHN	START	200	
	MOVER	R_1, ='3'	200
	MOVEM	R_1, X	201
Used	MOVER	R_2, ='2'	202
	LTORG (Director)		203
X	DS	1	204
	END		205

Design of working of Assembler.

(1) Analysis phase (Pass 1 of Assembler):

* To build Symbol Table for Synthesis phase to proceed.
* Determines address of each Symbols called as memory allocation.
* Location Counter used to hold addr. of next inst.
* Isolated label, mnemonic opCode, operands, Constants etc
* Validate meaning & addr. of each stmts.

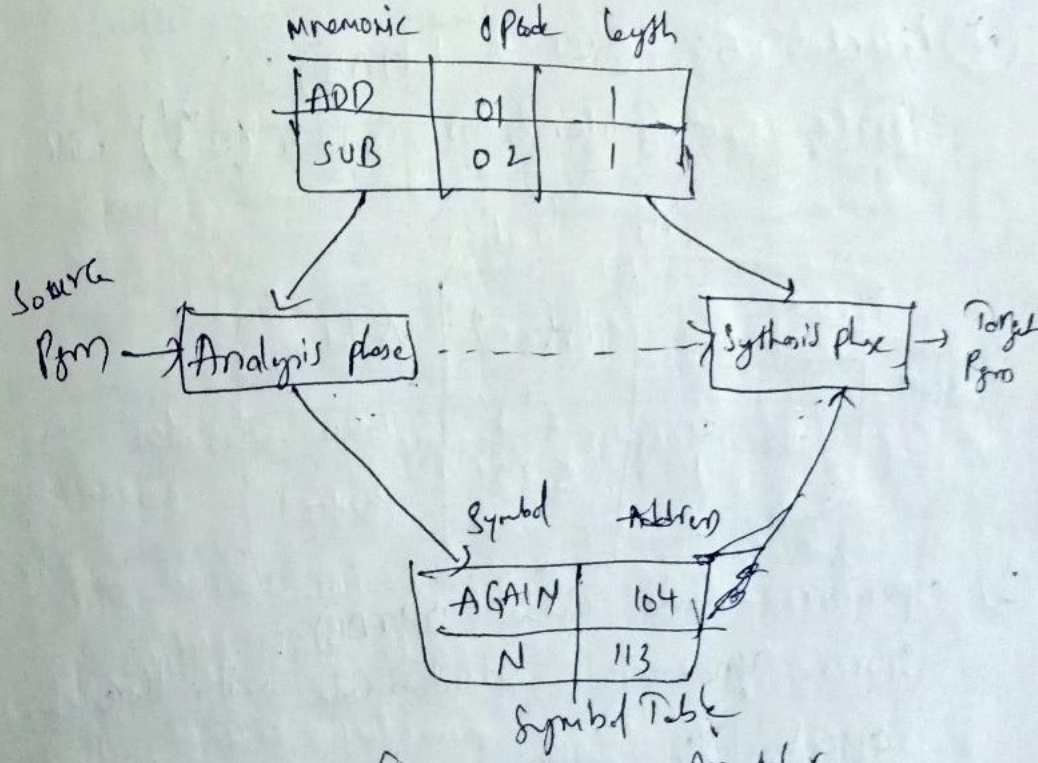


fig: Design of Assembler.

(2) Synthesis phase: (Pass 2 of Assembler):

* To build m/c inst.'s for every assembly stmts as per mnemonic code & three addr. allocation as per Src Code.
* Synthesis is m/c instr.

Pass I of Assembler (Analysis phase)

DS in Assembly lg.

(1) Symbol Table (ST or SYMTAB):

Store value of address assign to the label

Label	Addr.
JOHN	200
L1	202
X	204

(2) Literal Tab (LT or LITAB):

Store each literals or Constants (='3') with its location.

Starts with 0

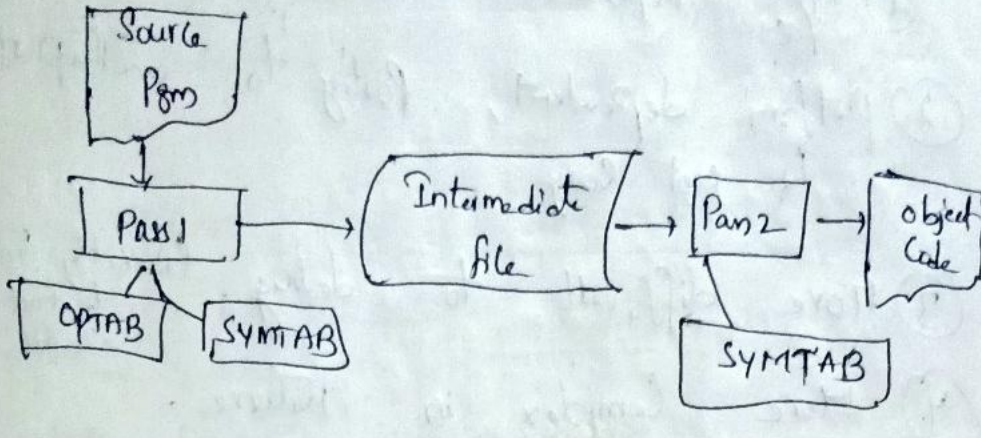
Index	Literal	Address
0	='3'	200
1	='2'	202

(3) Operation Code Table (OPTAB):

Stores Mnemonic operation code with their opcodes & length.

Inst.	opcode	length (Bytes)
MOVER	3	2 - int
MOVEM	X	1 - char
MOVER	2	2 - int

PASS 2 of Assembler (Synthesis phase)



* In the Second pass the Inst.'s are again read & are assembled using the Symbol table.
* Basically, the assembler goes through the block of Pgm & generates m/c code for that inst.
* Then the assembler proceeds to the next inst.
- In this way, the entire m/c Code Pgm is Created
* Convert mnemonic operation codes, Symbolic Table operands with there equivalent m/c code.
* Convert data Constants to Internal machine representations.
* Convert Complete Pgm into .obj file.

Advs:-
1. H/w oriented.
2. Increases readability by using DS.
3. Useful in embedded systems.
4. Less resources, Managing size & Code.
5. Allows H/w devices & system code easily as compared to High level

Disadv:-

(1) M/c Dependent - (H/w)

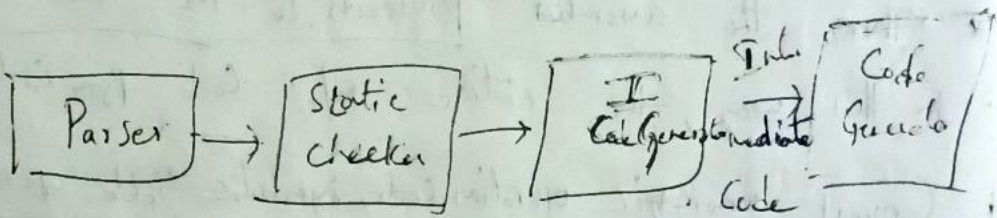
(2) platform dependent, Porting to another platform is not easy.

(3) More difficult to debug. (Directly in contact with H/w)

(4) More Complex in nature.

Intermediate Code generation:-

I.C is used to translate the source code into the M/c code. It lies b/w HLL & M/c lang.



* If the Compiler directly translates Src Code into m/c Code without generating intermediate code then a full native Compiler is required for each new m/c.

* Intermediate code generator receives i/p from its predecessor phase & Semantic analyzer phase. It takes i/p in the form of an annotated Syntax tree.

* using intermediate code, the Second phase of the Compiler is changed according to the target m/c. Intermediate code can be represented in 2 ways.

High level Intermediate code

It can be represented as Src Code.

low level I Code.

It is close to the target m/c. It is used for m/c dependent optimization.

The different forms of Intermediate Code :-

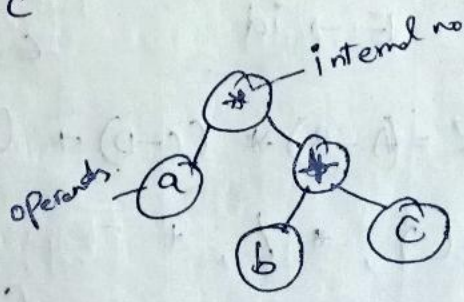
(1) Abstract Syntax tree.
(2) Polish Notation
(3) Three Address Code.

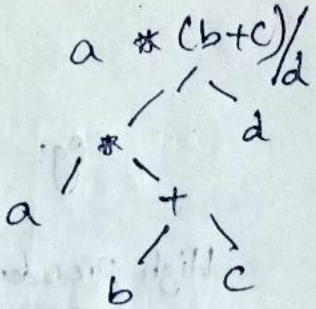
(1) Abstract Syntax Tree :-

These are compact than a parse tree & can be easily used by compiler.

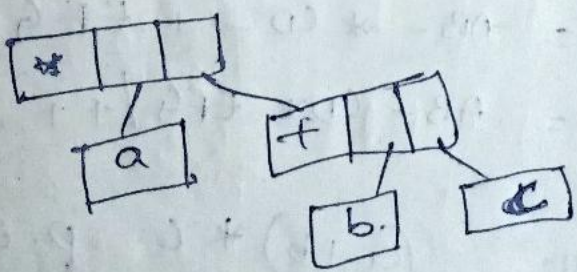
eg :- a * b + c

for arithmetic operators, associativity from left to right





It is represented as



② Polish Notation:- It is of 3 forms.

Infix (a+b)

Prefix (+ab)

Postfix (ab+) Notation:- It is also called as Suffix or Reverse polish Notation.

It is a Linear representation of syntax tree

eg:- x+y ⇒ xy+

Productions	Semantic Rule (SDG)
$E \to E_1$ op E_2	$E.code = E_1.code \| E_2.code \|$ op
$E \to (E_1)$	$E.code = E_1.code$
$E \to id$	$E.code = id$

eg:- $Z = (A-B) * (C-D) + [E+(F/G)]$

High precedence: (), *, /, +, -

$Z = AB- * CD- + [E+(F/G)]$

$= AB- * CD- + EFG/+$

$= AB-CD* EFG/++$

eg (a+b) * c P.E is ab+c*

a+(b*c) P.E is abc*+

(a-b) * (c/d) P.E is ab-cd/*

Three Address Code:-

* It is an intermediate code. It is used by the optimizing compilers.
* In 3-address code, the given expression is broken down into several separate instructions. These inst.'s can easily translate into Assembly lang.
* Each 3-address code instruction has at most 3 operands. It is a combination of assignment & a binary operator.

In TAC, there is at most one operator on the right side of an instruction.

eg:- $x + y * 2$

t_1, t_2 -

$t_1 = y * 2$

$t_2 = x + t_1$

Quadruple

Triple

Indirect Triple

eg: $x + y * 3$ (inst)

$t_1 = y * 3$

$t_2 = x + t_1$

operator
Argument 1
Argument 2
Result

each inst. consists of 4 fields.

highest priority unary minus

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

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$t_3 = b * t_1$

$t_4 = t_2 + t_3$

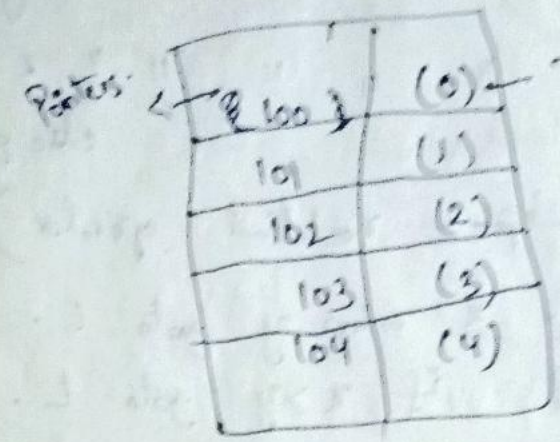
Addr	op	arg1	arg2	Result
(0)	-	c		t_1
(1)	*	b	t_1	t_2
(2)	-	c		
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5

Too many temp variables ie, stored in assembly table so that it takes more time. 3-address code. Hence we are going for

	op	arg1	Arg2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)

With less memory, we can represent same representation.

Indirect Triple. pointer to the triple.



(1) Assignment instructions of the form $x = y \text{ op } z$, where op is a binary operator.

Eg:- $x = y + z$, $x = y * z$, $x = y \&\& z$

(2) Assignments of the form $x = \text{op } y$ where op is unary operator such as unary -, inc, dec, logical NOT.

eg:- $x = -y$, $x = !y$

$x = ++y$

(3) Copy instructions of the form $x = y$ where value of y is assigned to x.

```
eg:- y = 10;
     x = y;
```

(4) Unconditional jump goto L.

```
eg:- goto L;
     L: stmts to be executed.

eg:- go to 1000;
```

(5) (a) Conditional Jumps of the form If x relop y goto L.

(b) " " " " " " " x goto L_1 else goto L_2.

(relop – relational operator)

(a) if x relop y goto L.

eg:- if $x > y$, goto L.

checks

(b) if x goto L_1 else goto L_2.

↓

Contains value 0, 1 (F) (T)

1 — L_1

0 — L_2.

(6) Procedure (function) Calls & returns are implemented using following Parameter x for y = call p, n (Pointer no. of arguments)

return y

(7) Address & Pointer assignments of the form

$x = \&y$ (Address of op)

$x = *y$ (The reference op) value at address

$*x = y$

1000 | 10 | *(1000)

(8) Indexed Copy Instructions of the form.

$x = y[i]$ (indexed).

$x[i] = y$

Q: Construct Quadruples, Triples, Indirect Triples for the statements $(a*b) * (c+d) - (a+b+c)$

Sol:- Three Addr. Code:-

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

Quadruple:-

	op	Arg1	Arg2	Result
(0)	+	a	b	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	+	t_1	c	t_4
(4)	-	t_3	t_4	t_5

Triple :-

	op	Arg1	Arg2
(0)	+	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	+	(0)	c
(4)	-	(2)	(3)

Pointers	Triples
100	(0)
101	(1)
102	(2)
103	(3)
104	(4)

Literal & Symbol Tables:-

Literals and Constants:

Literals: Literal is an operand with syntax

='<value>'

* Its location can't be specified in assembly Pgm. & its not known to assembly lang. Pgmer. So its value can't change during execution of Pgm.

eg:-

ADD AREG, ='5' this means

ADD AREG, FIVE

....

....

....

FIVE DC '5'

Constant:

* Its value can be changed during exe.
* eg: ADD AREG, 5

Is translated into an instruction with 2 operands. AREG and the value 5 as an immediate operand.

Consider the following Assembly lang. Program:-

Source program — Variant I — Variant II

```
START   100
        READ    N
LOOP    MOVER   AREG, N
        SUB     AREG, ='1'
        BC      GT, LOOP
        STOP
A       DS      1
        END
```

Symbol Table

	Symbol	Address	Length
1)	N	—	01
2)	LOOP	101	1
3)	A	105	1

Literal Table

Literal	Address
='1'	106

Pool Table

Literal No.
#1

Instruction opcode	Assembly Mnemonic
00	STOP
01	READ
02	MOVER
03	SUB
04	BC

Declarative	Short
DS	01

ASSEMBLER	DIRECTIVE
START	01
END	02

Write intermediate code form using Variant I & II.

Location Counter	V - I
100	(AD, 01) (C, 100)
	(IS, 01) (S, 01)
101	(IS, 02) (1) (S, 01)
102	(IS, 03) (1) (L, 01)
103	(IS, 04) (4) (S, 02)
104	(IS, 00)
105	(DL, 01) (C, 1)
	(AD, 02)

No change → Reg code

V - II

(AD, 01) (C, 100)

(IS, 01) N

(IS, 02) AREG, N

(IS, 03), AREG, (L, 01)

(IS, 04) GT, LOOP

(IS, 00)

(DL, 01) (C, 1)

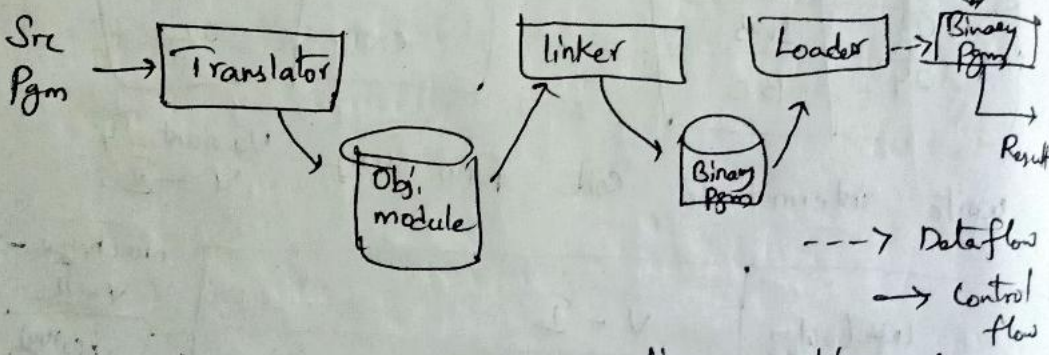
(AD, 02)

-1 < = -3 = -5 > -4

Relocation & linking Concepts:-

Execution of a Pgm written in a language L involves following steps:

* Translation of the Pgm by translator of language L
* Linking of the Pgm with other Pgms needed for its execution by a linker.
* Relocation of the Pgm to execute from the specific memory area allocated to it by a linker.
* Loading of the Pgm in the memory for the purpose of execution by a loader.



Translated, linked and load time addresses:-

Translation Time (or translated) address: Address assigned by translator.

Linked Address: Address assigned by the linker.

Load Time (or load) Address: Address assigned by Loader.

Translated Origin : Addr. of the origin assumed by translator which is specified by the Progammer in an ORIGIN Stmt.

Linked Origin : Addr. of the origin assigned by the linker while producing a binary Pgm.

Load time (or load) Address :- Addr. of the origin assigned by the loader while loading the Pgm for executing.

Topics :-

a) Program Relocation
   . Performing "

b) Linking
   - EXTERN & ENTRY stmt
   - Resolving external references
   - Binary programs.

c) Object Module

Pgm Relocation :-

Addr. Sensitive Pgm :-

(1) - AA - Set of Absolute Addresses, may be inst. addr. or data addr.

(2) - AA ≠ ϕ ; means inst. or data occupy memory words with specific addr.

(3) Such a Pgm is called Addr. Sensitive Pgm which contains

(1) Addr. Sensitive instruction : an instr. which contains addr. $a_i \in AA$.

(2) Addr. Constant : a data word which contains an address $a_i \in AA$.

Important: Addr. Sensitive Pgm 'P' can execute correctly only if the start addr. of the memory area allocated to it is the same as its translated origin.

Start Addr. == Translated Addr.

Thus, to execute correctly from any memory area, addr. used in each addr. sensitive instruction of P must be 'Corrected'.

Def :- Pgm Relocation is the process of modifying addr. used in the address sensitive instructions such that the pgm can execute correctly from the designated area of memory.

→ Linker performs relocation if Linked origin ≠ Translated Origin

→ Loader performs relocation if Load Origin ≠ Linked Origin

→ In general: Linker always performs relocation, where as some loaders do not.

- Absolute loaders do not perform relocation, then

load origin = Linked origin.

Thus, load origin & linked Origins are used interchangeably.

Performing Relocation:-

* Relocation factor (RF) of P is defined as

$$[\text{relocation\_factor}_P = l\_origin_P - t\_origin_P]$$

where, Rf is the relocation factor

* Symbol is operand

$$[t_{symb} = t\_origin_P + d_{symb}]$$

where, d_{symb} = offset

t_{symb} = translation time addr.

l_{symb} = link time addr.

t_origin_P = translation origin

$$[l_{symb} = l\_origin_P + d_{symb}]$$

where, l_origin_P = linked origin,

* $L_{symb} = t_{origin} + relocation\_factor_P$

$\rightarrow + d_{symb}$

$= t_{symb} + relocation\_factor_P$

IRR: Inst. Requiring Relocation.

It's set of insts that Perform relocation in Pgm P.

Steps:

* Calculate Relocation factor (RF)
* Add it to Translation Time Addr (es) for every inst. which is member of IRR.

eg:- RF = 900 - 500 = 400

IRR Contains translation addresses 540 & 538.

(inst : Read A)

Addr. is changed to 540 + 400 = 940 &

538 + 400 = 938.

Linking :-

AP is an App. Pgm Consisting of a set of Pgm units SP = {Pi}

A Pgm unit Pi interacts with another Pgm unit Pj using insts & addresses of Pj.

for this it must have.

Public Def.: A symbol pub_symb defined in Pgm unit which may be referenced in other Pgms.

External Reference: A reference to a symbol ext_symb which is not defined in Pgm unit containing the reference.

who will handle these 2 things?

EXTRN & ENTRY Stmts:-

* The ENTRY Stmt lists the public def. of Progcam unit.
- ie, it list those symbols defined in Pgm unit which may be referenced in other Pgm units.
* The EXTRN Stmt list the symbols to which external references are made in the Pgm unit.

Eg:- (1) TOTAL is ENTRY Stmt. (public def.)

(2) MAX & ALPHA are EXTRN Stmts. (external reference)

* Assembler don't know addr. of EXTRN symb & so puts 0's addr. field of inst. wherever these symb are found.

* what if we don't refer these var with EXTRN stmts?

* Assembler gives errors.

* Thus requirement arises for resolving external references.

Resolving External Reference :-

* Before AP executes, every external reference should be bound to correct link time addres.

* who will do this binding?

— Linker.

* Linking : It is the process of binding an external reference to Correct link time addr.

* External reference is said to be unresolved until linking is performed & resolved once linking is completed.

Binary Progeams.

- Is a m/c lang. Pgm comprising set of Progeam units Sp such that for all $P_i \in SP$,
  1. P_i relocated at link origin
  2. linking is performed for each external reference in P_i.

Linker Command :-

→ linker < link origin > < object module name
→ [, < execution start address >
→ linker Converts obj. module into → the set of Pgm units (Sp) into → a binary Pgm.
→ If link addr = load addr., loader simply loads binary Pgm into appropriate memory are a for execution.

object Module :-

→ OM contains all the info. necessary to relocate & link the Pgm with other Pgms.
→ OM consist of four Components.

(1) -Header: Contains -translated Origin, size & execution start addr. of P.

(2) Program: Contains m/c lang. Pgm Correspondg P.

(3) Relocation Table: (RELOCTAB) describing IRRp (Instruction Requiring Relocation). Each entry contain field: 'Translated Address' - of address sensitive instruction.

(4) Linking Table: (LINKTAB) Contains info. concerning Public def. & external reference. Has three fields:

Symbol: symbolic name

Type: PD/EXT ie, public def. or external reference.

Translated Addr: If PD, addr. of first memory word allocated. If EXT, addr. of memory word containing reference of symbol.

Example:

→ for previously given assembly program, object module contains following info:

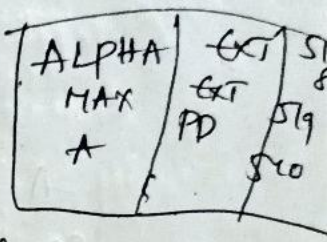
(1) Translated Origin: 500, Size: 42,
Exe. Strt. Add = 500

(2) Machine Language instructions

(3) Relocation Table:

500
538

(4) Linking Table:

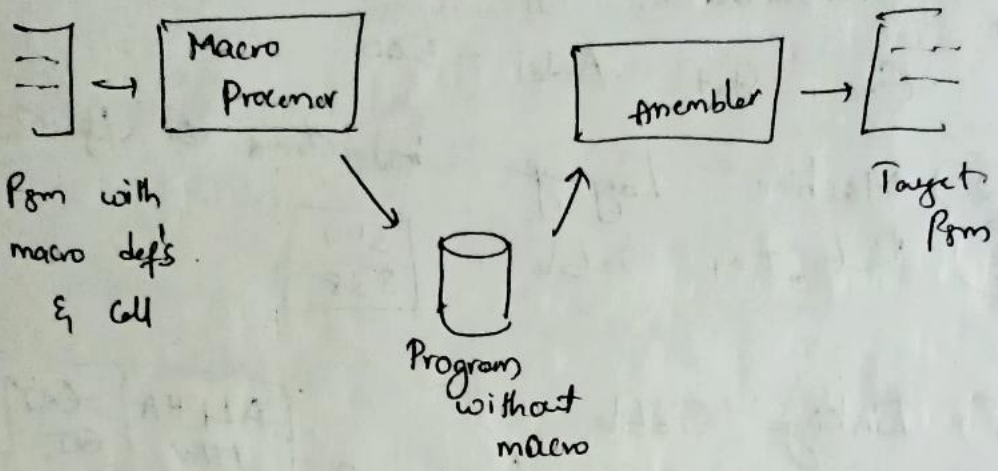


- Loop is not here as it is not PD.

← Public Def.

Macro Processors & Loaders
2. unit

Macro instructions & features :-



A schematic of a macro preprocessor.

eg:- Macro : PRINT_MSG (msg) expands to printf ("%s", msg);

Why Macro ? Code: PRINT_MSG ("Hello") becomes printf ("%s", "Hello");

In C pgming : Concept of function.

* function is block of stmts related to such task which we want to execute repeatedly in program.
* The function is defined once & can be repeatedly call whenever necessary.
* Suppose function call 100 times, Compiler toans fer control call to definition & definition to call 100 times.
* It waste lots of time.

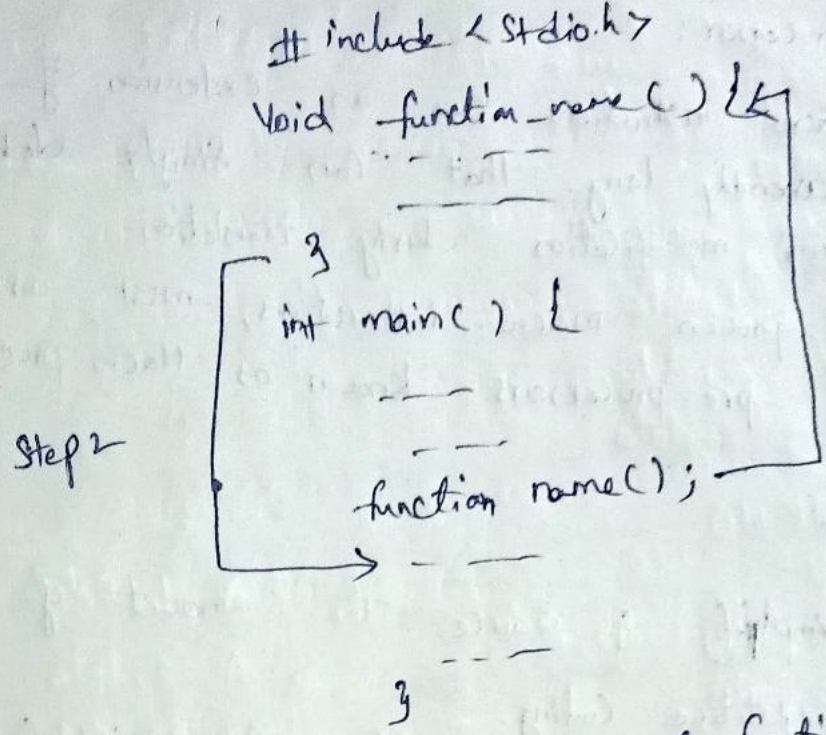


fig: working of functions

Macro :-

It is sequence of instructions assign by name which can be used at any location in the progeam.

* Macro instructions are single line abbreviations for sequential instructions.
* Using a Macro, Programmer can define a single "instruction" to represent block of code.

```
MACRO   ... Start of definition
INCR    ... Macro name
A1, DATA
A2, DATA    ... Sequence of instructions to be abbreviated
A3, DATA
MEND    ... END of Definition.
```

Micro Processor:

Macro instructions are an extension of the basic assembly lang. that can simply debugging & pgm modification during translation.

* To process macro instructions, most assembler use pre-processors known as Macro processors.

Adv:-

1) Simplify & reduce the amount of repetitive coding.
2) Reduce the possibility of errors caused by repetitive coding.
3) Make an assembly pgm more readable.

Disadv:- The macro is the size of pgm (Low level HLL)

The reason is, the pre-processor will replace all the macros in the pgm by its real definition prior to the compilation process of the progeam.

Macro Definition & Call

A macro definition is enclosed b/n the name of a macro header and a macro end statement.

It consists of:-

* A macro prototype stmt
* One or more model stmt.
* Macro Processor stmt.

Macro prototype Start :-

Syntax:

< macro name > [< formal Parameters >]

eg:-

formal par	Actual par
&par1	REG1
&par2	A
&par3	B

- Macro definition (logic of macro)

```
- MACRO
  MNAME  &par1, &par2, &par3
  MOVER  &par1, &par2
  ADD    &par1, &par3
  MOVEM  &par1, &par2
  MEND
```

Macro call

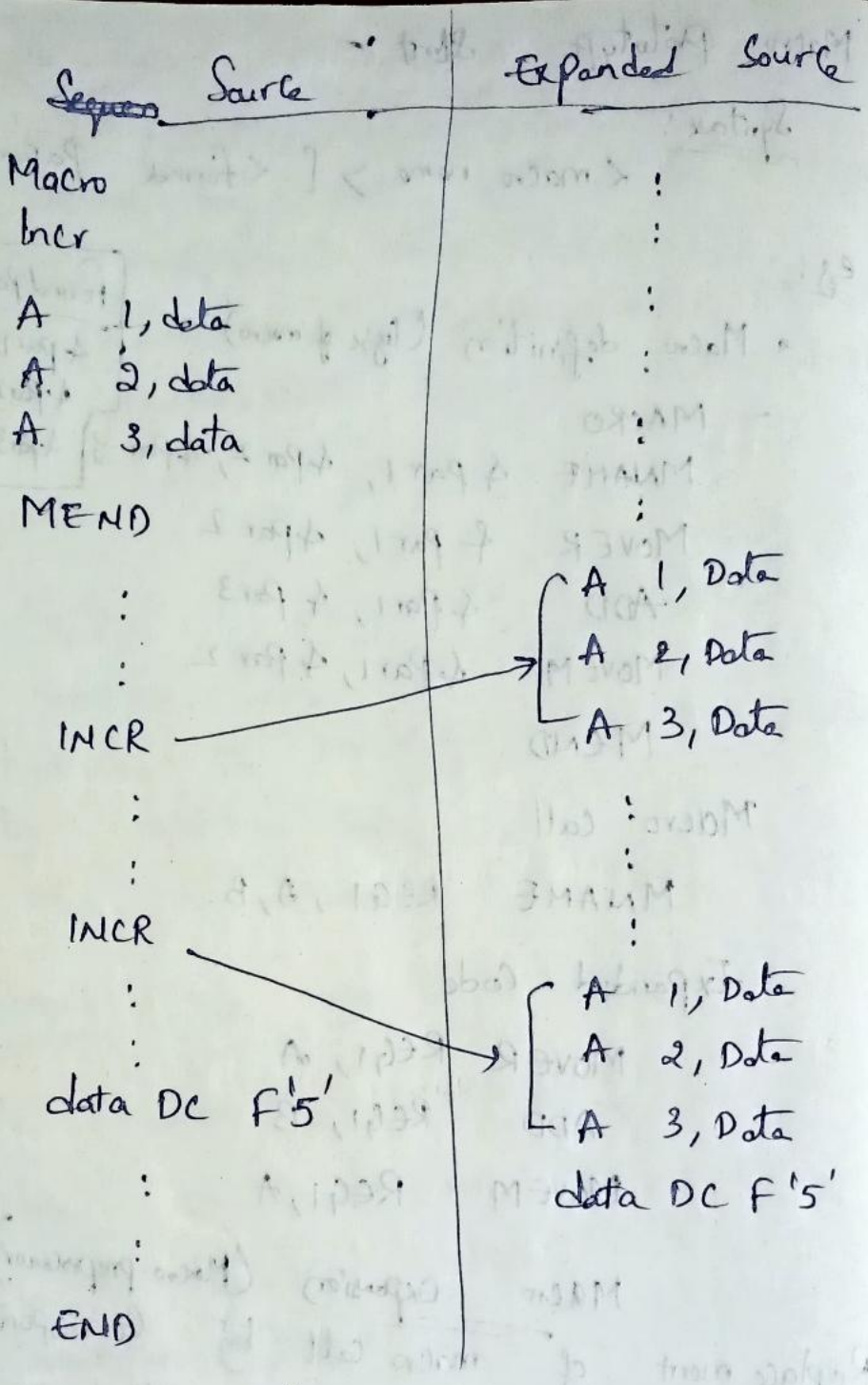
```
MNAME  REG1, A, B
```

Expanded Code

```
MOVER  REG1, A
ADD    REG1, B
MOVEM  REG1, A
```

Macro expansion (Macro preprocessor):-

Replacement of macro call by Corresponding Sequence of instructions is called as Macro expansion.



Macro Arguments :-

A) Positional argument (Actual Re)

Argument are matched with dummy arguments according to order in which they appear.

INCR A, B, C

A replaces first dummy arg.
B " Second " "
C " third " "

B) Keyword arguments (formal Par)

* This allows reference to dummy args by name as well as by position.

Eg:-

INCR &arg1 = A, &arg3 = C, &arg2 = B

eg:

INCR &arg1 = &arg2 = A, &arg = 'C'

Nested Macro:-

In a macro, Model statement can constitute a call on any other macro. These calls are called as nested macro.

eg:-

Source Code

Def. of Macro ADDI

```
MACRO
ADDI  &arg
L     1, &arg
A     1, =F'10'
ST    1, &arg
MEND
```

Def. of Macro ADDS

```
MACRO
ADDS  &arg1, &arg2,
      &arg3
ADDI  &arg1
ADDI  &arg2
ADDI  &arg3
MEND
```

Source Code | Expanded Code (Level 1) | E.C (L2)

```
:
MACRO
ADD1 &arg
 L   1, &arg
 A   1, =f'10'
 ST  1, &arg
MEND
 MACRO
ADDS &arg1, &arg2, &arg3
 ADD1 &arg1
 ADD1 &arg2
 ADD1 &arg3
  MEND
      :
      :
ADDS data1, data2, data3 →  ADD1 data1
      :                     ADD1 data2
      :                      ADD1 data3
data1  Dcf'5'
data2  Dcf'6'
data3  Dcf'7'
 END
```

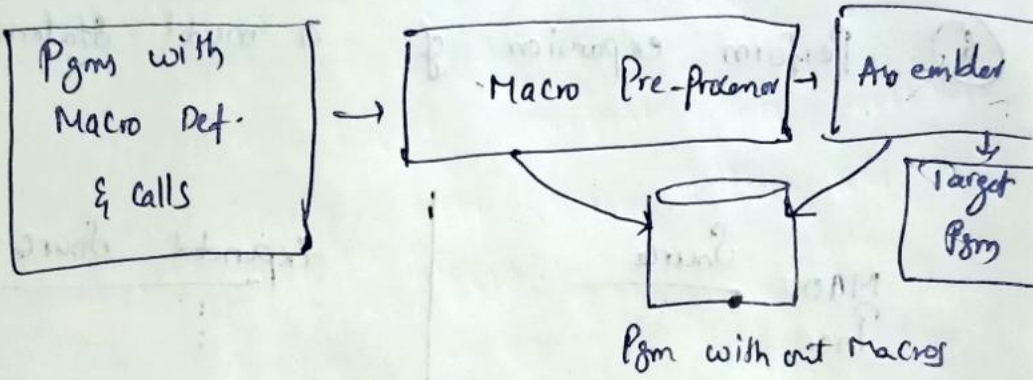
Expansion of ADDS

```
:
:
:
:
:
data Dcf'5'
data Dcf'6'
data Dcf'7'
 END.
```

Macro preprocessor :-

It takes assemble lang. Pgms which contains macros as i/p & translate into assembly lang. Pgm without macros.

The o/p of macro preprocessor can be given to assembler to convert it into target lang. Pgm.

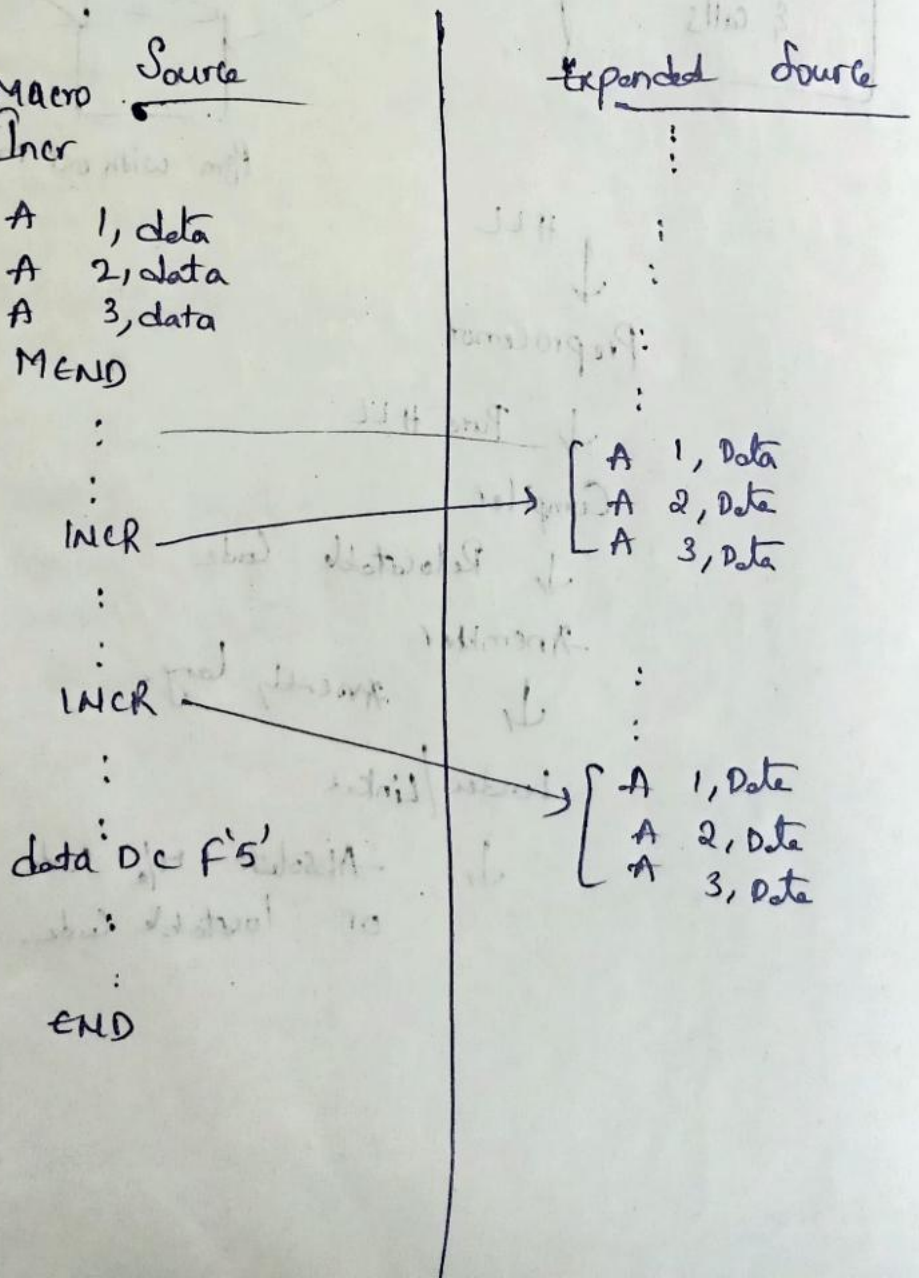


HLL
↓
Preprocessor
↓ Pure HLL
Compiler
↓ Relocatable Code
Assembler
↓ Assembly lang.
Loader / Linker
↓ Absolute M/c code or locatable code

Design working of Macro preprocessor.

Design overview:

1. Identify macro calls in the Pgm.
2. Determine the values of formal Parameters.
3. Maintain the values of expansion time variables declared in a macro.
4. Organize expansion time control flow.
5. Determine the values of sequencing symbol.
6. Perform expansion of a model statement.



Data structures :-

- Macro Name Table (MNT)
- Parameter Name Table (PNTAB)
- EV Name Table (EVNTAB)
- SS " " (SSNTAB)
- Keyword Parameter Default Table (KPDTAB)
- Macro Definition Table (MDT)
- Actual Parameter Table (APTAB)
- EV Table (EVTAB)

UNIT –III

Scanning, Parsing, and Compilers

Language grammars and ambiguity, Lexical analysis – regular expressions, token generation, Syntax analysis – parsing techniques (top-down, bottom-up), Semantic analysis and intermediate code generation, Code optimization techniques – constant folding, dead code elimination

It focuses on the **front-end phases of a compiler**, which involve **analyzing the structure of a program** written in a high-level language and converting it into an intermediate form that can be optimized and translated into machine code.

It mainly covers the **process of reading and understanding source code** — from recognizing words and symbols (scanning/lexical analysis) to understanding the grammatical structure (parsing/syntax analysis) and producing intermediate code for further compilation.

Language Grammars and Ambiguity

Defines the formal structure (syntax) of programming languages using **context-free grammars (CFGs)**.

Discusses **ambiguous grammars**, where a statement can be parsed in more than one way, and how to resolve ambiguity through grammar refinement or precedence rules.

· **Lexical Analysis (Scanning)**

The **first phase** of compilation.

Converts the **stream of characters** from the source code into **tokens** using **regular expressions** and **finite automata**.

Handles tasks such as removing comments and whitespace, recognizing identifiers, keywords, literals, and operators.

Token Generation

Defines how tokens are created and categorized (e.g., IDENTIFIER, NUMBER, OPERATOR).

Uses **regular expressions** to specify token patterns.

Implements **longest-match rules** and **priority handling** to manage overlapping patterns.

Syntax Analysis (Parsing)

The **second phase** of compilation.Uses **parsing techniques** (top-down and bottom-up) to analyze the sequence of tokens and check whether they follow the language grammar.

Builds **parse trees or syntax trees** representing program structure.

Semantic Analysis & Intermediate Code Generation

Checks for **semantic correctness** (e.g., type compatibility, variable declarations).

Produces an **intermediate code representation (IR)** — a bridge between high-level source code and low-level machine code.

Code Optimization Techniques

Improves efficiency without changing the meaning of the program.

Examples: **constant folding**, **dead code elimination**, **strength reduction**.

What is a Language Grammar?

A **grammar** defines the **syntactic structure** (rules) of a programming language — i.e., how valid statements and expressions are formed.
It is a formal description of how tokens (like identifiers, numbers, and operators) can be combined to form valid programs.

Components of a Grammar (Context-Free Grammar - CFG)

A **Context-Free Grammar (CFG)** is defined as a 4-tuple:

G=(V,T,P,S)G = (V, T, P, S)G=(V,T,P,S)

Where:

V (Variables / Non-terminals): Symbols that represent groups or categories of syntactic structures (e.g., E for Expression).

T (Terminals): Actual symbols or tokens of the language (e.g., id, +, *, (,)).

P (Productions): Set of rules that describe how terminals and non-terminals can be combined.

S (Start Symbol): A special non-terminal that represents the whole program (e.g., E).

Example of a Grammar:Let's define a grammar for arithmetic expressions:

E → E + EE → E * EE → (E)E → id

Here:

Non-terminal: E (Expression)

Terminals: id, +, *, (,)

Start symbol: E

This grammar defines expressions like:

id + id
id * (id + id)
(id + id) * id

Derivation and Parse Tree

A **derivation** shows how a string is produced using the grammar rules.
A **parse tree** visually represents the structure of the derivation.

Example:

Input: id + id * id

Possible derivations:

E → E + E → id + E → id + E * E → id + id * id

The parse tree shows how operators combine operands.

What is Ambiguity?

A grammar is **ambiguous** if **a single string** can have **more than one parse tree (or derivation)**.

That means the same expression can be understood in more than one way — leading to confusion in meaning or order of execution.

Example of Ambiguity

Consider the grammar above:

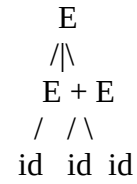
E → E + EE → E * EE → id

For the input:

id + id * id

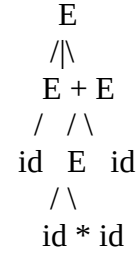
There are **two different parse trees** possible:

(a) + evaluated first



Meaning: (id + id) * id

(b) * evaluated first



Meaning: id + (id * id)

Hence, the grammar is **ambiguous**, because it does not specify operator **precedence** or **associativity**.

What is Lexical Analysis?

Lexical Analysis is the **first phase of a compiler**.
It reads the **source code (characters)** and converts it into a sequence of **tokens**.

 Think of it like a translator that breaks a sentence into individual words before understanding its meaning.

A **token** is the **smallest meaningful unit** in a program.

Examples of tokens:

Code	**Tokens Generated**
int num = 10;	int, num, =, 10, ;
a = b + c;	a, =, b, +, c, ;

Each token has a **type** and a **value** (lexeme):

int → (KEYWORD, "int")

num → (IDENTIFIER, "num")

10 → (NUMBER, "10")

= → (OPERATOR, "=")

; → (SYMBOL, ";")

Role of the Lexical Analyzer

The **Lexical Analyzer (Scanner)**:Reads source code character by character.

Groups characters into **tokens** using patterns called **regular expressions**.

Removes spaces, tabs, and comments.

Passes the tokens to the **Parser** (next phase)

What are Regular Expressions?

A **regular expression (regex)** is a pattern that describes how certain strings (like identifiers, numbers, or keywords) are formed.

In lexical analysis, **regular expressions are used to define the rules for each token type.**

Common Regular Expression Patterns

Token Type	**Regular Expression**	**Example**
Identifier	[A-Za-z_][A-Za-z0-9_]*	sum, _total, count1
Integer Constant	[0-9]+	5, 123, 99
Floating Point Number	[0-9]+\.[0-9]+	3.14, 10.0
Keyword	`if	else
Operator	`+	-
Whitespace	[\t\n]+	(space, tab, newline)
String Literal	`"([^"\]	\.)*"`

---

6. Example: Step-by-Step Tokenization

Input Code:

int total = price + tax;

Process:

Step	**Input Characters**	**Matched Token**	**Token Type**
1	int	int	KEYWORD
2	space	ignored	—
3	total	total	IDENTIFIER
4	space	ignored	—

Step	Input Characters	Matched Token	Token Type
5	=	=	OPERATOR
6	space	ignored	—
7	price	price	IDENTIFIER
8	space	ignored	—
9	+	+	OPERATOR
10	space	ignored	—
11	tax	tax	IDENTIFIER
12	;	;	SYMBOL

Output Tokens:

(KEYWORD, "int")
(IDENTIFIER, "total")
(OPERATOR, "=")
(IDENTIFIER, "price")
(OPERATOR, "+")
(IDENTIFIER, "tax")
(SYMBOL, ";")

Important Rule – Longest Match Rule (Maximal Munch)

When two patterns can match the same string,
The **longest match** is chosen.

Example:
For input ==,

= (operator) and == (relational operator) both match,

but == is **longer**, so it's chosen.

Keywords vs Identifiers

Both match the same pattern [A-Za-z_][A-Za-z0-9_]*,
but if the word is a **reserved keyword** (like if, for, int),
the lexer labels it as a **keyword** instead of an identifier.

Example with Regular Expressions

Let's match some tokens:

Input	Matched Regex	Token
sum1	[A-Za-z_][A-Za-z0-9_]*	IDENTIFIER
123	[0-9]+	NUMBER
while	while	KEYWORD
+	\+	OPERATOR

Input	Matched Regex	Token
"Hello"	`"([^"\]	\.)*"`

What is Token Generation?

After lexical analysis (scanning), the compiler **groups characters** into meaningful units called **tokens**.
This process is known as **token generation**.

Each **token** represents a logical category such as **identifier**, **number**, **operator**, or **keyword**.

Structure of a Token

A token usually has 2 or 3 parts:

Part	Meaning	Example
Token Type	The category of the token	IDENTIFIER, NUMBER, KEYWORD, OPERATOR
Lexeme	Actual string from the source code	"count", "=", "10"
Attribute (optional)	Extra info (like value or symbol table index)	value = 10

Example:

Input:

int sum = a + b;

Generated Tokens:

Token Type	Lexeme	Attribute (if any)
KEYWORD	int	—
IDENTIFIER	sum	symbol table index
OPERATOR	=	—
IDENTIFIER	a	symbol table index
OPERATOR	+	—
IDENTIFIER	b	symbol table index
SYMBOL	;	—

How Tokens Are Generated

Regular expressions define patterns for each token type.
Example:Identifier → [A-Za-z_][A-Za-z0-9_]*

Number → [0-9]+

Operator → \+ | - | =

Lexical analyzer (scanner) reads the source code **character by character**.

Syntax Analysis (Parsing Techniques)

What is Syntax Analysis?

After token generation, the **parser** checks if the token sequence follows the grammar rules of the language.
This phase is called **Syntax Analysis** or **Parsing**.

 The parser builds a **parse tree** or **syntax tree** showing the grammatical structure of the source code.

Example

Grammar:

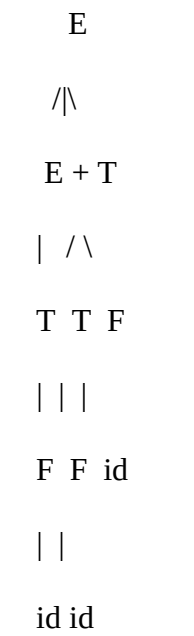
E → E + T | T

T → T * F | F

F → (E) | id

Input Tokens:
id + id * id

The parser checks whether these tokens follow the grammar and builds a tree like:



When a sequence of characters matches a regex pattern,
→ the **token is created** and sent to the **parser**.

The analyzer also ignores whitespaces and comments.

Example – Step-by-Step

Input:

total = 100 + tax;

Process:

total → matches identifier pattern → **(ID, "total")**

= → matches operator pattern → **(OP, "=")**

100 → matches number pattern → **(NUM, "100")**

+ → **(OP, "+")**

tax → **(ID, "tax")**

; → **(SYMBOL, ";")**

Output Token Stream:

(ID, total), (OP, =), (NUM, 100), (OP, +), (ID, tax), (SYMBOL, ;)

Syntax Analysis (Parsing Techniques)

What is Syntax Analysis?

After token generation, the **parser** checks if the token sequence follows the grammar rules of the language.
This phase is called **Syntax Analysis** or **Parsing**.

The parser builds a **parse tree** or **syntax tree** showing the grammatical structure of the source code.

Example

Grammar:

E → E + T | TT → T * F | FF → (E) | id

Input Tokens:
id + id * id

 The parser checks whether these tokens follow the grammar and builds a tree like:

```
    E
   /|\
  E + T
 |   / \
 T  T  F
 | | |
 F  F  id
 | |
 id id
```

Types of Parsing Techniques

There are two major types of parsing:

 1. Top-Down Parsing

Starts **from the start symbol (S)** and tries to **derive the input tokens**.

It expands non-terminals into possible productions until the input string is produced.

Works like **prediction** → "Can this rule produce the input?"

Common Methods:

Recursive Descent Parsing

LL(1) Parsing

Example (Recursive Descent):
Grammar:

E → T + E | TT → id

Input: id + id

Step-by-step:

Start with E.

Expand E → T + E.

Match T → id.

Input id matched.

Match +.

Expand remaining E → T.

Match T → id.
 Accepted — the input is valid.

Advantages:

Simple to implement by hand.

Easy to understand.

Disadvantages:

Can fail with **left-recursive** grammars (e.g., E → E + T).

Not suitable for complex grammars.

2. Bottom-Up Parsing

Starts **from the input tokens** and tries to **reduce** them back to the start symbol.

Works like **construction in reverse** → "Can we reduce this string to S?"

Common Methods:

Shift-Reduce Parsing

LR(0), SLR(1), LALR(1) Parsers

Example (Shift-Reduce):
Grammar:

E → E + T | TT → id

Input: id + id

Steps:

Shift: read id → stack: id

Reduce: id → T

Reduce: T → E

Shift: read +

Shift: read id

Reduce: id → T

Reduce: E + T → E
 Reduced to start symbol E → Accepted!

Advantages:

Very powerful, handles most programming language grammars.

Used in modern parser generators like **YACC**, **Bison**.

Disadvantages:

More complex to understand and implement.

Top-Down vs Bottom-Up (Comparison Table)

Feature	**Top-Down Parsing**	**Bottom-Up Parsing**
Direction	From start symbol to input	From input to start symbol
Method	Expands productions	Reduces productions
Common Types	Recursive Descent, LL(1)	Shift-Reduce, LR(1), LALR
Implementation	Simple, hand-written	Complex, automated
Grammar Type	Cannot handle left recursion	Can handle left recursion
Use	Simpler languages	Real-world languages

Semantic Analysis

After parsing, the compiler knows the structure (syntax) of the program —
but now it must check whether the **meaning** of that structure is **correct**.
That process is called **semantic analysis**.

Purpose:Semantic analysis checks for logical or meaning-related errors that are not caught by syntax rules.

Main Tasks of Semantic Analysis

Task	**Example**	**What Happens**
Type Checking	int a; a = "hello";	Error — assigning string to int
Variable Declaration Check	x = 10;	Error — x not declared
Function Parameter Check	sum(5); but function sum(int a, int b)	Error — missing parameter
Scope Checking	Using a variable outside its scope	Error — undefined variable
Array Bounds Check	arr[10] for array size 5	Error — index out of range (runtime check)

How It Works

Semantic Analyzer uses a **Symbol Table**, which stores:

Variable names

Data types

Scope (global/local)

Function info (parameters, return type)

When analyzing, it:

Looks up each identifier in the symbol table.

Checks if usage matches declaration.

Ensures expressions are type-compatible.

Example

```
int a, b;float c;
a = 5;
b = a + 2;
c = a + b;
```

All assignments are **type-consistent** (int + int = int, which can be stored in float).

If you wrote:

```
a = "text";
```

Semantic Error → Type mismatch (cannot assign string to int).

Intermediate Code Generation

Once syntax and semantics are correct, the compiler converts the source code into an **intermediate form**, called **Intermediate Code (IC)**.

This IC is not machine code, but it's easier to **optimize** and **translate** to any target machine.

Purpose:Acts as a bridge between source code and machine code.

Makes the compiler **machine-independent**.

Simplifies optimization.

Common Intermediate Representations

Type	Example	Description
Three-Address Code (TAC)	t1 = a + b t2 = t1 * c	Each instruction has at most 3 addresses (operands)
Quadruples	(op, arg1, arg2, result)	Explicit representation for each operation
Triples	(op, arg1, arg2)	Uses positions instead of names for results

Example

Source Code:

a = b + c * d;

Intermediate Code (Three-Address Form):

t1 = c * dt2 = b + t1a = t2

Quadruple Representation:

Op	Arg1	Arg2	Result
*	c	d	t1
+	b	t1	t2
=	t2	—	a

Code Optimization Techniques

What is Code Optimization?

After generating intermediate code, the compiler tries to make it **faster** and **more efficient**, without changing what the program does.

Goal: **Better performance**, **less memory usage**, and **smaller code**.

Types of Code Optimization

Type	Description
Machine-Independent	Performed on intermediate code (works for all CPUs)
Machine-Dependent	Performed after code generation (specific to hardware)

We'll discuss two common **machine-independent** techniques:

(a) Constant Folding

Definition:
If an expression involves **only constants**, the compiler can **evaluate it at compile time** instead of runtime.

Example:

int a = 5 * 10;

Constant Folding result:

int a = 50;

Another example:

int b = (2 + 3) * 4;

→ Compiler changes to:

int b = 20;
It Saves runtime computation — compiler already knows the value.

(b) Dead Code Elimination

Definition:
Removes code statements that **never affect the program's output** or are **never executed**.

Example 1 – Unreachable Code:

```
if (false) {
   printf("This will never run");
}
```

 Compiler removes this block — **dead code**.

Example 2 – Unused Variable:

```
int a = 10;
a = 20;
a = 30;printf("%d", a);
```

Only the last assignment matters → previous ones are **dead** and removed.

Example 3 – After return statement:

```
return x;
y = x + 1;  // Dead code (never executes)
```

Compiler deletes y = x + 1.

Benefits of Optimization

Reduces program size
Increases execution speed

Decreases memory and CPU usage
Improves battery life (for mobile devices)

Quick Comparison Table

Concept	Purpose	Example
Semantic Analysis	Checks meaning and type correctness	int a; a = "hi"; □
Intermediate Code Generation	Converts source to machine-independent code	t1 = a + b
Constant Folding	Pre-computes constant expressions	2 + 3 → 5
Dead Code Elimination	Removes unnecessary code	Code after return

UNIT IV

Linkers, Debuggers, and Shell Programming

Symbol resolution and relocation, Linking (static vs dynamic), relocation records,

Debugging techniques and breakpoints, Unix/Linux shell environment, Shell

commands, variables, redirection, pipes, control statements, Shell script functions and script-based automation

What is *linking* (big picture)?

A *linker* (link editor) combines compiled object files (.o) and libraries (.a, .so) into a final program (an executable or a shared library).
Its two main jobs are:

> **Symbol resolution** — match every *reference* (use of a variable or function) to a *definition* (where it is defined).
>
> **Relocation** — fix up addresses in the code/data so each reference points to the correct final memory address.

Linking can happen in different phases:

> **Link-time** (when building the executable — static linking resolves most symbols then)
>
> **Load-time / run-time** (dynamic linking may resolve some symbols when the program starts or on first use)

Symbol resolution

A symbol is a name representing code or data: function names, global variables, static/local symbols, etc.

Symbol kinds

> **Defined symbol**: an object file provides its location (e.g., int x = 10; or void foo(){...}).
>
> **Undefined symbol**: a file references it but does not define it (e.g., extern int x; or a call to printf).
>
> **Local vs Global**: local symbols are internal to object file; global symbols are visible for resolution across files.
>
> **Strong vs Weak**: weak symbols allow a linker to prefer a strong definition if one exists (useful for default implementations).

How resolution proceeds (conceptual algorithm)

The linker collects all object files and libraries given on the command line.

It builds a global table of **defined symbols** and **undefined references**.

For each undefined symbol, the linker searches the defined symbols (in object files and then in libraries).

If a definition is found, the undefined reference is **resolved** to that definition.

If not found → **link error**: "undefined reference to foo".

If there are multiple definitions of the same global symbol:

Two *strong* definitions → **error** (multiple-definition).

Strong + weak → strong wins.

For static archives (.a), the linker pulls in only those object modules from the archive that define currently unresolved symbols; order matters.

Quick example

main.c

```
extern int x;int main() { return x; }
```

def.c

```
int x = 42;
```

Commands:

```
gcc -c main.c    # -> main.o (has undefined symbol x)
gcc -c def.c     # -> def.o (defines x)
gcc -o prog main.o def.o   # linker resolves x -> def.o
```

Result: prog has symbol x resolved to the data in def.o.

Notes about archives and order

With libstuff.a, the linker scans archive members and **adds only those** .o **files that satisfy unresolved symbols**. If symbols appear later on the command line, they may remain unresolved — so ordering matters or use --start-group.

Symbol visibility & versioning

Compilers/linkers can hide or expose symbols (visibility attributes) and symbol versioning is used in shared libraries to manage API/ABI changes. These affect how symbol resolution behaves at runtime.

Relocation

Why relocation?

Object files are generated with internal addresses that are **relative** or placeholders. The final executable or shared object will be loaded at particular memory addresses; the linker/loader must **patch** instructions/data to point to real addresses.

What a *relocation entry* contains

For each address in the code/data that depends on a symbol, the object file contains a relocation record including:

offset (where to patch)

type (absolute vs PC-relative, etc.)

symbol index (which symbol this relocation refers to)

addend (extra constant)

Two common relocation types (conceptual)

Absolute relocation: final value = address_of(symbol) + addend → used for global data pointers.

PC-relative (relative) relocation: final value = address_of(symbol) + addend − place → used for relative jumps/calls (helps with position independency).

What the linker does (static linking)

For each relocation entry, the linker computes the final address of the referenced symbol and **writes (patches)** the computed value into the output section at the relocation offset.

It also merges/concatenates sections from multiple .o files into one output section (e.g., .text, .data) and adjusts offsets accordingly.

What the loader/dynamic linker does (dynamic linking)

Shared libraries may be loaded at addresses determined at load time (ASLR, address conflicts). The *dynamic linker* (ld.so on Linux) will process relocation entries in the shared objects and executable, patching addresses either:

At load time (eager relocation) — all relocations fixed when loading, or

Lazily — function addresses are fixed on first call (via PLT/GOT mechanism).

Position-Independent Code (PIC) reduces the number of relocations needed for code by using relative addressing and indirection through the GOT.

Example of relocation: from main.o

If main.o has an instruction like call foo but the object file stores a placeholder, relocation tells the linker: "at offset 0x20 in .text there's a call instruction that needs to be updated to call symbol foo's final address + addend".

PIC, GOT and PLT — dynamic linking avoids heavy relocation

Position-Independent Code (PIC)

Compiled with -fPIC on Unix-like systems.

Uses relative addressing so the same code can run regardless of absolute load address (necessary for shared libraries).

Reduces or eliminates relocations inside .text.

GOT (Global Offset Table)

A table of addresses used by PIC code to access global data and function addresses. The code loads addresses from GOT entries rather than embedding absolute addresses.

PLT (Procedure Linkage Table):A trampoline area used to call external functions. The PLT entry initially jumps to the dynamic linker resolver. On first call, the resolver finds the real address, writes it into the GOT, and subsequent calls go directly to the function.

Enables **lazy binding** (resolve a function only when it is first called).

Static linking vs Dynamic linking

Static Linking

All used library code is copied into the final executable at link time. The executable contains everything it needs.

link with static libraries (e.g., libfoo.a) or use gcc -static to prefer static libc.

Pros:

Single self-contained binary — no external dependencies at runtime.

Predictable behavior (no surprise changes when system libraries update).

Sometimes slightly faster startup (no runtime relocation of libraries).

Cons:

Large executable size (duplicate code if many programs statically link the same library).

Cannot share memory pages of the library code across processes.

Harder to patch a bug in a library (need to rebuild/redeploy all apps).

Licensing issues (some libraries/OSS licenses restrict static linking).

Dynamic Linking (Shared Libraries)

Executable contains references to shared libraries (.so / .dll) and the dynamic loader resolves and loads those libraries at program start (or earlier/later).

compile shared objects (gcc -fPIC -shared -o libmylib.so mylib.o) and link with them (gcc main.o -L. -lmylib).

Pros:

Smaller executables (shared code stays in .so files).

Multiple processes share the same in-memory library pages.

Libraries can be updated independently — a bugfix in a .so benefits all apps that use it.

Cons:

Programs can break if incompatible library changes are installed (dependency hell / ABI mismatch).Slight runtime overhead: loader work and potential relocations at load-time, plus possible lazy resolution overhead on first call.

Less deterministic if system libraries change.

Practical example commands

Create object files:

```
gcc -c main.c      # main.o
gcc -c lib.c       # lib.o
```

Create a shared library:

```
gcc -fPIC -shared -o libmylib.so lib.o
gcc -o prog main.o -L. -lmylib   # links to libmylib.so (dynamic)
```

Create a statically linked program:

gcc -static -o prog_static main.o lib.o# or link with static archive libmylib.a

Runtime linking details & tools

Dynamic loader (ld.so / ld-linux.so) is responsible for locating shared libs (via rpath, RUNPATH, LD_LIBRARY_PATH, standard system paths) and performing relocations.

dlopen/dlsym allow explicit runtime loading and symbol lookup.

Tools to inspect/linking artifacts:

ld / gcc (linker driver)

readelf -s / nm (inspect symbol tables)

ldd (shows dynamic dependencies of an executable)

objdump (view relocations and sections)

Edge cases & important behaviors

Multiple definitions: Multiple strong definitions → linker error. Some platforms permit symbol interposition where a symbol in the executable overrides one in a shared library.

Archives (.a): Linker only extracts needed object modules, so order of libraries matters.

Lazy vs Eager resolution: Lazy reduces startup cost but moves cost to first call; eager resolves everything at startup.

ASLR & relocations: Address Space Layout Randomization moves libraries around; PIC + GOT/PLT help make shared libraries relocatable without modifying code pages.

Stripping symbols: strip reduces file size but removes symbol info used for debugging

What is Relocation?

When you compile a program, each .c or .cpp file becomes an **object file** (.o), which contains **machine code**, **symbols**, and **relocation information**.

However, the **final memory addresses** of functions and variables are **not yet known** — because the linker still needs to **combine** multiple object files and libraries into one executable.

 So, **Relocation** means **adjusting addresses** in the code and data after linking.

We Need Relocation

Each object file assumes it starts at **address 0**.

When files are combined, they are placed at **different memory addresses**.

Therefore, **addresses in instructions and data** must be **updated** to point to their actual (final) locations.

Example:

Symbol	Object File	Original Address	Final Address
main()	main.o	0x0000	0x1000
sum()	sum.o	0x0000	0x2000

If main() calls sum(), the call instruction must be **relocated** from address 0x0000 to 0x2000.

What Are Relocation Records?

A **relocation record** (or relocation entry) is **metadata** stored in an object file that tells the linker:

"At this place (offset) in the code, you need to insert or adjust the address of a symbol."

It contains **information for the linker** to fix the addresses.

Contents of a Relocation Record

Each relocation record typically includes:

Field	Description
Offset	The location in the section (e.g., .text, .data) where a fix-up is required.
Type	The type of relocation (absolute, PC-relative, etc.).
Symbol Index	Which symbol this relocation refers to.
Addend	An additional constant value to add.

 Example

Let's say we have two files:

main.c

```
extern int sum(int, int);int main() {
  return sum(2, 3);
}
```

sum.c

```
int sum(int a, int b) {
  return a + b;
}
```

After compiling:

```
gcc -c main.c   # main.o
gcc -c sum.c    # sum.o
```

main.o has:

A **symbol table** entry for the external function sum.

A **relocation record** like:

Offset: 0x14Type: R_X86_64_PC32Symbol: sumAddend: -4

This tells the linker:

"At offset 0x14, replace the placeholder with the final address of sum."

 During Linking

When the linker (ld) combines the files:

It assigns final addresses to all functions and variables.

It goes through each relocation record.

It updates the instructions in .text and data in .data to the correct absolute or relative addresses.

After relocation → the executable (a.out) has **fixed addresses** that the CPU can execute directly.

Relocation Records

Concept	**Explanation**
Purpose	Adjust symbol addresses after linking.
Stored In	Object files (.o) under sections like .rela.text.
Used By	Linker and Loader.
Contains	Offset, type, symbol, addend.

Concept **Explanation**
Example Tool readelf -r main.o shows relocation entries.

Debugging Techniques and Breakpoints

Debugging is the process of **finding and fixing errors (bugs)** in a program so it runs correctly.

A **debugger** is a special tool that allows developers to:

Run programs step by step,

Examine variable values,

Set **breakpoints**,

Watch memory/registers,

And control program execution interactively.

Common Debugging Tools

Platform	**Debugger Tool**
Linux / Unix	gdb (GNU Debugger)
Windows	Visual Studio Debugger
macOS	lldb
IDEs	Code::Blocks, Eclipse, VSCode (internally use gdb/lldb)

 Common Debugging Techniques

1 Print (or Trace) Debugging

Add printf() or cout statements in code to check values and flow.

```
printf("x = %d\n", x);
```

Simple but manual; not suitable for large programs.

Using a Debugger

Compile the program with debug info:

gcc -g myprog.c -o myprog

Run it inside gdb: gdb ./myprog

Breakpoints

A **breakpoint** is a marker that tells the debugger:

> "Stop running the program here, before executing this line."

You can inspect variable values, memory, and flow from that point onward.

In gdb**:**

```
(gdb) break main        # set breakpoint at start of main()
(gdb) run               # run the program
(gdb) print x           # see the value of variable x
(gdb) next              # go to next line
(gdb) continue          # resume execution
(gdb) delete 1          # remove breakpoint number 1
```

Step Execution

step → go into functions line by line

next → go to next line (skip inside functions)

continue → run until next breakpoint

Watchpoints

A **watchpoint** stops the program whenever a **variable's value changes**.

(gdb) watch x

 Useful to detect where a variable becomes incorrect.

Backtrace

Shows the sequence of function calls (stack trace) that led to a crash or current line.

(gdb) backtrace

Core Dumps

If a program crashes, the OS can produce a **core file** (snapshot of memory).
You can open it with gdb:

gdb ./a.out core

and inspect the state of the program at the moment of crash.

Memory and Logic Checks

Tools like:**Valgrind** → finds memory leaks and invalid accesses.

AddressSanitizer (ASan) → runtime check for out-of-bounds or use-after-free.

Unix/Linux Shell Environment — Overview

The **Shell** is a **command-line interface (CLI)** between the user and the **Unix/Linux operating system**.
It lets you **run commands, write scripts, automate tasks, and manage the system**.

Common shells:

Bash (Bourne Again Shell) – default on most Linux systems

Sh, **Zsh**, **Csh**, **Ksh**, **Fish**

Shell Environment

The shell environment consists of all **settings, variables, and configurations** that define how the shell behaves for a user.

It includes:

Environment variables

The working directory

Command history

Aliases and functions

Example Commands:

echo $HOME # shows user's home directoryecho $PATH # shows directories searched for commandspwd # print working directorywhoami # shows current user

Key Point:

The shell environment defines the **user's working context** when using the command line or running shell scripts.

Shell Commands

Shell commands are **instructions** entered in the terminal to perform actions like viewing files, copying data, or running programs.

There are two main types:

Internal (built-in) – handled by the shell itself (e.g., cd, echo, exit)

External – separate executable programs (e.g., /bin/ls, /usr/bin/grep)

Common Examples:

Command	Purpose
ls	List files and directories
cd folder	Change directory
pwd	Show current directory
cp file1 file2	Copy files
mv file1 file2	Move or rename files
rm file	Delete file
cat file	View file contents
man command	Show manual for a command

Example:

ls -l /home

→ Lists all files in /home in long format.

Shell Variables:

Variables store **data or values** that can be used later in commands or scripts.

Types:

User-defined variables – created by the user in scripts or terminal.

Environment variables – predefined system variables used by the shell.

Examples:

name="Sona"echo "Hello $name" # prints Hello Sona
echo $HOME # predefined environment variable
PATH=/usr/bin:/bin # set pathexport PATH # make it available to child processes

Key Point:

Variables help store information temporarily and make scripts dynamic.

Redirection:Redirection allows you to **change where input or output comes from or goes to**.
By default:

Input → keyboard

Output → screen

You can **redirect** these to files.

Operators:

Operator	Meaning	Example
>	Redirect output (overwrite)	ls > files.txt
>>	Redirect output (append)	echo "Hi" >> notes.txt
<	Take input from a file	sort < data.txt
2>	Redirect error output	gcc file.c 2> error.txt
&>	Redirect both output & error	command &> all.txt

Example:

cat file1 file2 > combined.txt

Combines two files and writes output to combined.txt.

Pipes (|)

A **pipe** connects the **output of one command** directly as the **input to another** — without using intermediate files.

Used to **chain commands** together for efficient data processing.

Syntax:

command1 | command2 | command3

Examples:

ls | wc -l

Counts the number of files in the current directory.

ps aux | grep firefox

Filters the list of running processes to show only “firefox”.

Key idea:
Pipes help you combine simple commands to perform complex tasks.

Control Statements (in Shell Scripting)

Control statements manage the **flow of execution** in shell scripts — similar to loops and conditions in programming languages.

Types and Examples:

(a) If-Else Statement

```
if [ $a -gt $b ]then
  echo "a is greater"else
  echo "b is greater"fi
```

(b) For Loop

```
for i in 1 2 3do
  echo "Number $i"done
```

(c) While Loop

```
count=1 while [ $count -le 5 ]do
  echo "Count = $count"
  count=$((count + 1))done
```

(d) Case Statement

```
case $choice in
  1) echo "Start";;
  2) echo "Stop";;
  *) echo "Invalid";;esac
```

Shell Script Function:-

A **function** is a **block of reusable code** inside a shell script.
Functions let you **group commands together** and **call them multiple times**, which makes scripts **shorter, cleaner, and easier to manage**.

Basic Syntax

```
function_name() {
   # commands
}
# orfunction function_name {
   # commands
}
```

We can **call the function** by writing its name:

```
function_name
```

Functions can **take arguments** like scripts do.

Using Arguments

```
greet() {
   echo "Hello, $1!"  # $1 = first argument
}
```

```
greet "Sona"
```

Output:

```
Hello, Sona!
```

$1, $2, … $n → positional parameters for function arguments

$@ → all arguments

return → returns an exit status (0–255)

echo → can return text for assignment

Example: Reusable Function

```
# Function to check if a file existscheck_file() {
   if [ -f "$1" ]; then
      echo "File $1 exists."
   else
      echo "File $1 does not exist."
   fi
}

check_file file.txt
check_file data.csv
```

Advantages:

Avoid repeating the same code multiple times

Makes scripts **organized and readable**

Supports **modular scripting**

Script-Based Automation

Automation means using **scripts** to **perform tasks automatically** without human intervention.

Shell scripts can automate:

File management (backup, delete, move)

System monitoring (disk usage, CPU load)

Software installation or updates

Scheduled jobs using **cron**

Use Automation

Saves time on repetitive tasks

Reduces human error

Ensures tasks run consistently

Useful for **system administration** and **DevOps**

Basic Automation Example

Backup Script

```
#!/bin/bash# Backup /home/user/Documents to /home/user/Backup
backup() {
   src="/home/user/Documents"
   dest="/home/user/Backup"
   timestamp=$(date +%Y%m%d_%H%M%S)
   mkdir -p "$dest"
   cp -r "$src" "$dest/Documents_$timestamp"
   echo "Backup completed at $timestamp"
}

backup  # call the function
```

Can be scheduled using **cron**:

```
crontab -e# Add line to run every day at 2 AM
0 2 * * * /home/user/scripts/backup.sh
```

Output: Automatically creates timestamped backups every day.

Tips for Script Automation

Always test scripts manually before scheduling.

Use **functions** for modular tasks (like backup, logging).

Use **variables** for paths, filenames, and parameters.

Add **logging** for tracking success or failure:

```
echo "$(date) Backup completed" >> backup.log
```

UNIT V

Unix/Linux System Programming

Introduction to system-level programming in C, File I/O system calls (open, read, write, close), Process creation using fork(), exec(), wait(), Inter-process communication (pipes, FIFO), Signal handling and POSIX threads (pthread_create, pthread_join), Case studies: background processes, daemon creation, mini shell

Introduction to System-Level Programming in C

System-level programming refers to writing programs that interact closely with the **operating system** and **hardware resources**. Unlike high-level application programming, system programming allows you to:

Access files, directories, and devices directly.

Control processes and memory.

Perform network communications.

Use system calls provided by the OS.

In **Unix/Linux**, system programming is mostly done in **C**, because C provides:

Low-level memory access.

Direct mapping to system calls.

Efficient execution.

Key characteristics:

Programs interact with **kernel services** through **system calls**.

Examples: File management, process control, signal handling, inter-process communication.

File I/O in Unix/Linux

File input/output (I/O) in Unix/Linux can be done in two ways:

Standard I/O – Using functions like fopen(), fread(), fprintf().

High-level, buffered.

Easier for general programming.

System-level I/O – Using **system calls** like open(), read(), write(), close().

Low-level, unbuffered.

Directly interacts with the kernel.

Provides more control and efficiency.

We will focus on **system-level file I/O**.

Common File I/O System Calls in Unix/Linux

(a) open()

Used to **open a file** and obtain a **file descriptor**.

Syntax:

```
#include <fcntl.h>
int open(const char *pathname, int flags, mode_t mode);
```

pathname – Path to the file.

flags – Access mode (read/write/append) and other options.

mode – File permission (used when creating a new file).

Common flags:

Flag	Description
O_RDONLY	Open file for reading only
O_WRONLY	Open file for writing only
O_RDWR	Open file for reading and writing
O_CREAT	Create file if it doesn't exist
O_TRUNC	Truncate file to zero length
O_APPEND	Append data to end of file

Example:

```
int fd = open("file.txt", O_RDWR | O_CREAT, 0644);
if (fd == -1) {
   perror("open");
   return 1;
}
```

Returns a **file descriptor** (integer ≥ 0) if successful.

Returns -1 on error.

(b) read()

Reads data from a file descriptor into a buffer.

Syntax:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

fd – File descriptor obtained from open().

buf – Buffer to store data.

count – Number of bytes to read.

Example:

```
char buffer[100];
ssize_t bytesRead = read(fd, buffer, sizeof(buffer));
if (bytesRead == -1) {
    perror("read");
}
```

Returns the number of bytes actually read.

Returns 0 if end-of-file (EOF) is reached.

Returns -1 on error.

(c) write()

Writes data from a buffer to a file descriptor.

Syntax:

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

fd – File descriptor.

buf – Buffer containing data to write.

count – Number of bytes to write.

Example:

```
char *msg = "Hello, Unix!\n";
ssize_t bytesWritten = write(fd, msg, strlen(msg));
```

```
if (bytesWritten == -1) {
   perror("write");
}
```

Returns the number of bytes written.

Returns -1 on error.

(d) close()

Closes an open file descriptor.

Syntax:

```
#include <unistd.h>int close(int fd);
```

fd – File descriptor to close.

Frees kernel resources associated with the file.

Always close files after use to avoid leaks.

Example:

```
if (close(fd) == -1) {
   perror("close");
}
```

Complete Example of File I/O in C

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
int main() {
   int fd = open("example.txt", O_CREAT | O_WRONLY, 0644);
   if (fd == -1) {
      perror("open");
      return 1;
   }

   char *msg = "Hello, Unix/Linux System Programming!\n";
   if (write(fd, msg, strlen(msg)) == -1) {
      perror("write");
      close(fd);
      return 1;
   }

   close(fd);
```

```
    // Reading the file
    fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    char buffer[100];
    ssize_t bytesRead = read(fd, buffer, sizeof(buffer)-1);
    if (bytesRead == -1) {
        perror("read");
        close(fd);
        return 1;
    }

    buffer[bytesRead] = '\0'; // Null terminate
    printf("File content:\n%s", buffer);

    close(fd);
    return 0;
}
```

Explanation:

Open a file example.txt for writing (O_CREAT | O_WRONLY).

Write a message to the file.

Close the file.

Reopen the file for reading (O_RDONLY).

Read the contents into a buffer.

Print the content.

Close the file.

Process Creation in Unix/Linux

In Unix/Linux, a **process** is an instance of a program in execution. **Creating processes** is fundamental in system programming. The main system calls involved are:

fork() – Create a new process.

exec() – Replace the current process image with a new program.

wait() / waitpid() – Synchronize parent and child processes.

(a) fork() – Creating a new process

fork() is used to create a **child process**. The child is a copy of the parent, sharing almost everything initially, except:

Process ID (PID) is unique.

Parent PID (PPID) points to the parent.

File descriptors are shared but maintain separate offsets.

Syntax:

```
#include <unistd.h>
pid_t fork(void);
```

Return value:

Returns 0 in the **child process**.

Returns the **child PID** in the **parent process**.

Returns -1 on **failure**.

Example:

```
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
    } else if (pid == 0) {
        printf("Child process, PID = %d\n", getpid());
    } else {
  printf("Parent process, PID = %d, Child PID = %d\n", getpid(), pid);
    }

    return 0;
}
```

Explanation:

fork() duplicates the process.

Both parent and child execute the code **after fork()**.

PID helps distinguish parent from child.

(b) exec() – Replacing a process image

The **exec()** family of functions replaces the **current process image** with a **new program**.
There are several versions: execl(), execp(), execv(), etc.

Syntax (example with execl):

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ..., NULL);
```

Example:

```
#include <stdio.h>
#include <unistd.h>
int main() {
   pid_t pid = fork();

   if (pid == 0) {
      // Child executes a new program
      execl("/bin/ls", "ls", "-l", NULL);
      perror("execl failed"); // Only executes if execl fails
   } else {
      wait(NULL); // Parent waits for child
      printf("Parent process finished\n");
   }

   return 0;
}
```

Explanation:

Child process runs ls -l.

Parent waits until child finishes.

(c) wait() – Synchronizing parent and child

wait() makes the **parent process wait** until any child terminates.

Syntax:

```
#include <sys/wait.h>
pid_t wait(int *status);
```

Returns **child PID** on success.

Returns -1 if no children exist.

status contains termination information.

Example:

```
int status;
pid_t child_pid = wait(&status);
if (WIFEXITED(status)) {
   printf("Child exited with code %d\n", WEXITSTATUS(status));
}
```

Inter-Process Communication (IPC)

Processes can communicate using **shared memory, signals, pipes, FIFOs, and sockets**. We'll focus on **pipes** and **FIFO**.

(a) Pipes – Anonymous Communication

Pipes are **unidirectional communication channels** between **related processes** (parent-child).

Data written by **write()** can be read by **read()**.

Created using pipe().

Syntax:

```
#include <unistd.h>
int pipe(int fd[2]);
```

fd[0] → read end

fd[1] → write end

Example:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main() {
   int fd[2];
   pipe(fd); // Create pipe
   pid_t pid = fork();

   if (pid == 0) {
      // Child writes to pipe
      close(fd[0]); // Close read end
      char msg[] = "Hello from child";
      write(fd[1], msg, strlen(msg)+1);
      close(fd[1]);
   } else {
      // Parent reads from pipe
      close(fd[1]); // Close write end
      char buffer[100];
      read(fd[0], buffer, sizeof(buffer));
```

```
        printf("Parent received: %s\n", buffer);
        close(fd[0]);
    }

    return 0;
}
```

Explanation:

Pipe allows **one-way communication** from child → parent.

Always close unused ends to avoid deadlocks.

(b) FIFO – Named Pipes

FIFO (First In First Out) is a **named pipe**, can communicate between **unrelated processes**.

Created using mkfifo() or mknod().

Accessed like a regular file using open(), read(), write(), close().

Syntax:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Example:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
int main() {
    char *fifo = "/tmp/myfifo";
    mkfifo(fifo, 0666); // Create FIFO

    pid_t pid = fork();
    if (pid == 0) {
        // Child writes
        int fd = open(fifo, O_WRONLY);
        write(fd, "Message through FIFO", 20);
        close(fd);
    } else {
        // Parent reads
        int fd = open(fifo, O_RDONLY);
        char buffer[100];
        read(fd, buffer, sizeof(buffer));
        printf("Parent received: %s\n", buffer);
```

```
        close(fd);
    }

    unlink(fifo); // Delete FIFO
    return 0;
}
```

Explanation:

FIFO persists in the filesystem.

Can be used by **unrelated processes** for communication.

unlink() removes the FIFO.

Signal Handling in Unix/Linux

Signals are **asynchronous notifications** sent to a process to notify it of events like:

Interrupt from keyboard (Ctrl+C)

Illegal memory access

Child termination

Timer expiry

Common signals:

Signal	**Description**
SIGINT	Interrupt (Ctrl+C)
SIGKILL	Kill process (cannot be caught)
SIGTERM	Termination request
SIGCHLD	Child process terminated
SIGALRM	Timer alarm

Handling signals

We can **catch signals** using signal() or sigaction().

Syntax:

```
#include <signal.h>
void signal(int sig, void (*handler)(int));
```

Example – Catching Ctrl+C:

```
#include <stdio.h>
```

```
#include <signal.h>
#include <unistd.h>
void handle_sigint(int sig) {
   printf("\nCaught signal %d. Exiting safely.\n", sig);
   _exit(0);
}
int main() {
   signal(SIGINT, handle_sigint);
   while (1) {
      printf("Running...\n");
      sleep(1);
   }
   return 0;
}
```

Explanation:

When the user presses Ctrl+C, handle_sigint() executes.

Provides safe termination or cleanup.

POSIX Threads (pthreads)

Threads are lightweight processes that share the **same memory space**. POSIX threads allow **multithreading in C**.

Key Functions:

pthread_create() – Create a new thread

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
```

thread → thread ID

start_routine → function executed by thread

arg → argument to thread function

Returns 0 on success

pthread_join() – Wait for a thread to finish

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

Blocks the calling thread until the target thread terminates.

Example – Thread creation and join

```
#include <stdio.h>
#include <pthread.h>
void* print_message(void* arg) {
   char* msg = (char*) arg;
   printf("%s\n", msg);
   return NULL;
}
int main() {
   pthread_t tid;
   pthread_create(&tid, NULL, print_message, "Hello from thread");
   pthread_join(tid, NULL);
   printf("Thread finished execution\n");
   return 0;
}
```

Explanation:

Thread prints a message.

Main thread waits for it to complete.

Case Studies

(a) Background Processes

Background processes run **without blocking the terminal**.

Achieved by appending & in shell or using fork() + setsid() in C.

Example:

```
pid_t pid = fork();
if (pid == 0) {
   // Child process running in background
   printf("Child running in background\n");
   sleep(10);
   _exit(0);
} else {
   printf("Parent continues without waiting\n");
}
```

Parent doesn’t wait; child runs independently.

(b) Daemon Process Creation

Daemon: **Background service process** with no controlling terminal.

Steps to create a daemon:

fork() and exit parent.

setsid() → create a new session.

Optional second fork() to prevent reacquisition of terminal.

Close standard file descriptors (stdin, stdout, stderr).

Run background tasks.

Example Skeleton:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
int main() {
   pid_t pid = fork();
   if (pid > 0) exit(0); // Parent exits

   setsid(); // New session
   chdir("/"); // Change working directory

   // Close standard file descriptors
   close(STDIN_FILENO);
   close(STDOUT_FILENO);
   close(STDERR_FILENO);

   // Daemon main loop
   while (1) {
      // Background tasks
      sleep(10);
   }
   return 0;
}
```

(c) Mini Shell Implementation

A **mini shell** allows basic command execution. Core steps:

Display prompt.

Read input (fgets()).

Parse command and arguments.

fork() → create child process.

execvp() → execute command in child.

wait() → parent waits for child.

Example:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
int main() {
   char cmd[100];
   while (1) {
      printf("mini-shell> ");
      fgets(cmd, sizeof(cmd), stdin);
      cmd[strcspn(cmd, "\n")] = 0; // Remove newline

      if (strcmp(cmd, "exit") == 0) break;

      pid_t pid = fork();
      if (pid == 0) {
         char *args[] = {cmd, NULL};
         execvp(args[0], args);
         perror("exec failed");
         exit(1);
      } else {
         wait(NULL);
      }
   }
   return 0;
}
```

Supports simple commands like ls, pwd.

Extensible for pipes, redirection, and background processes.