310243- System Programming & Operating System

Unit Number: **1**

Unit Name: **Introduction**

Unit Outcomes: To analyze basic System Software and its functionality.

)

Syllabus

- Introduction to Systems Programming
- Need of Systems Programming
- Software Hierarchy
- Types of software: system software and application software
- Machine structure
- **Evolution of components of Systems Programming** Text Editors, Assembler, Macros
- Compiler, Interpreter, Loader, Linker, Debugger Device Drivers, Operating System
- **Elements of Assembly Language Programming**: Assembly Language statements, Benefits of Assembly Language
- A simple Assembly scheme, Pass Structure of Assembler.
- **Design of two pass Assembler**: Processing of declaration statements
- Assembler Directives and imperative statements, Advanced Assembler Directives
- Intermediate code forms, Pass I and Pass II of two pass Assembler.

Introduction to System Programming

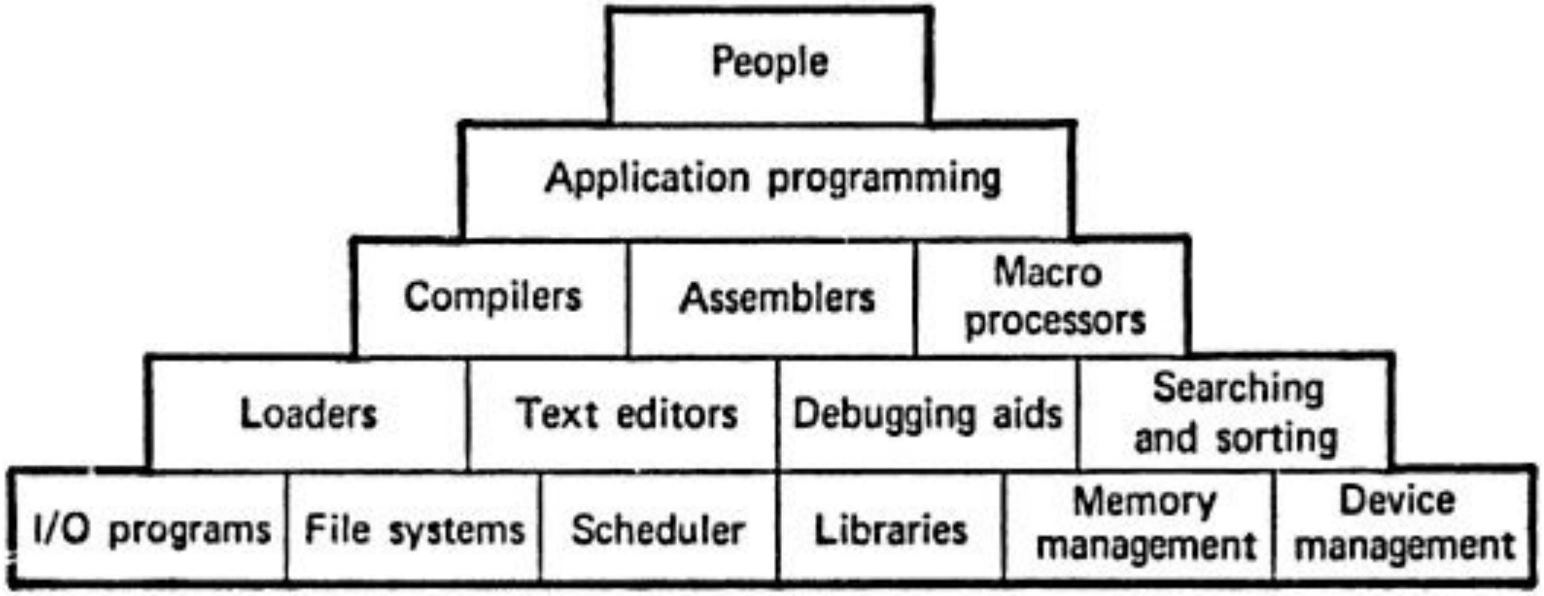
What is systems programming?

- System programs are nothing but the compilers, loaders, macros processor, operating system.
- Computer can not understand the language without aid of system programs.

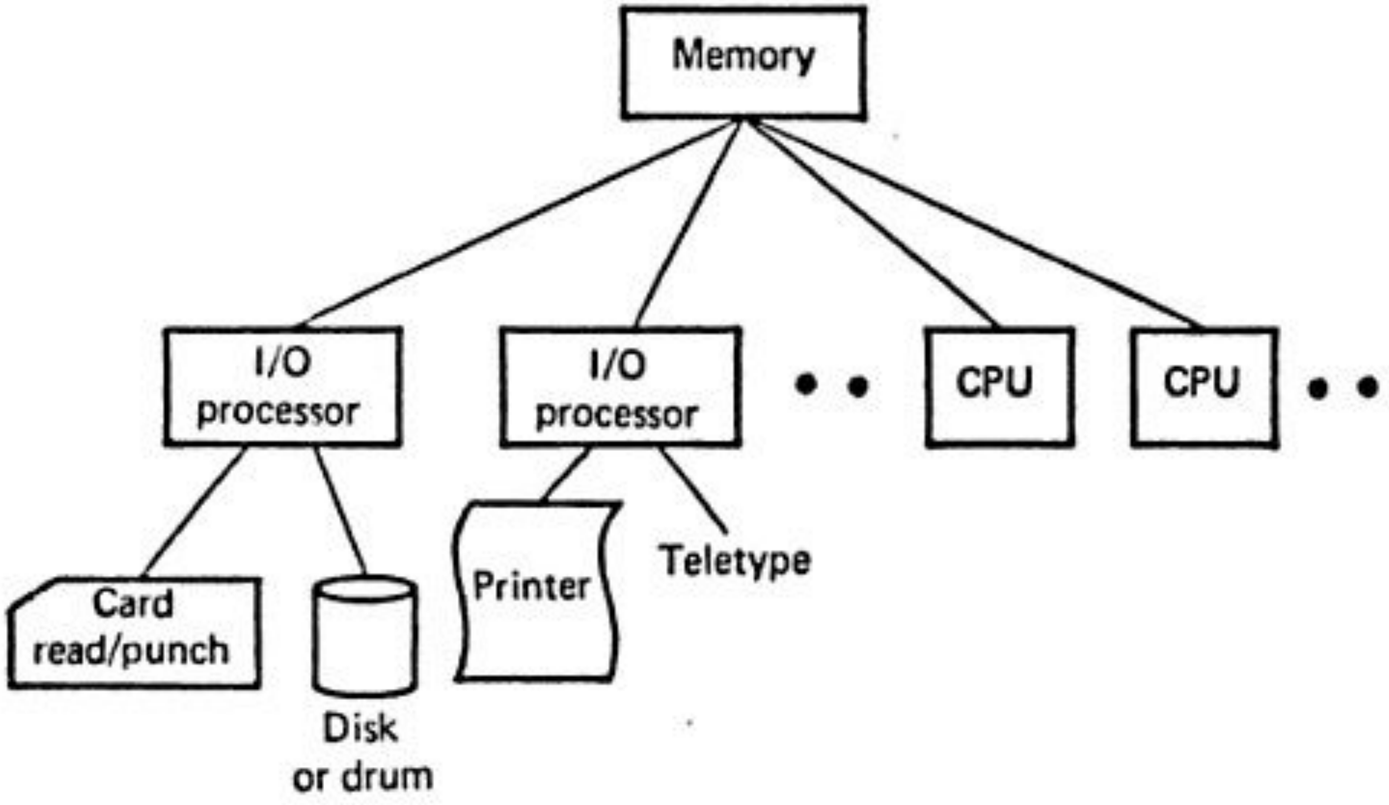
Cont…

- Compilers are system programs that accept people-like language and translate them into machine language.
- Loaders are system programs that prepare machine language programs for execution.
- Macro processors allow programmers to user abbreviations.
- Operating systems and file systems allow flexible storing and retrieval of information.

Foundation of System Programming



General Hardware Organization of Computer System



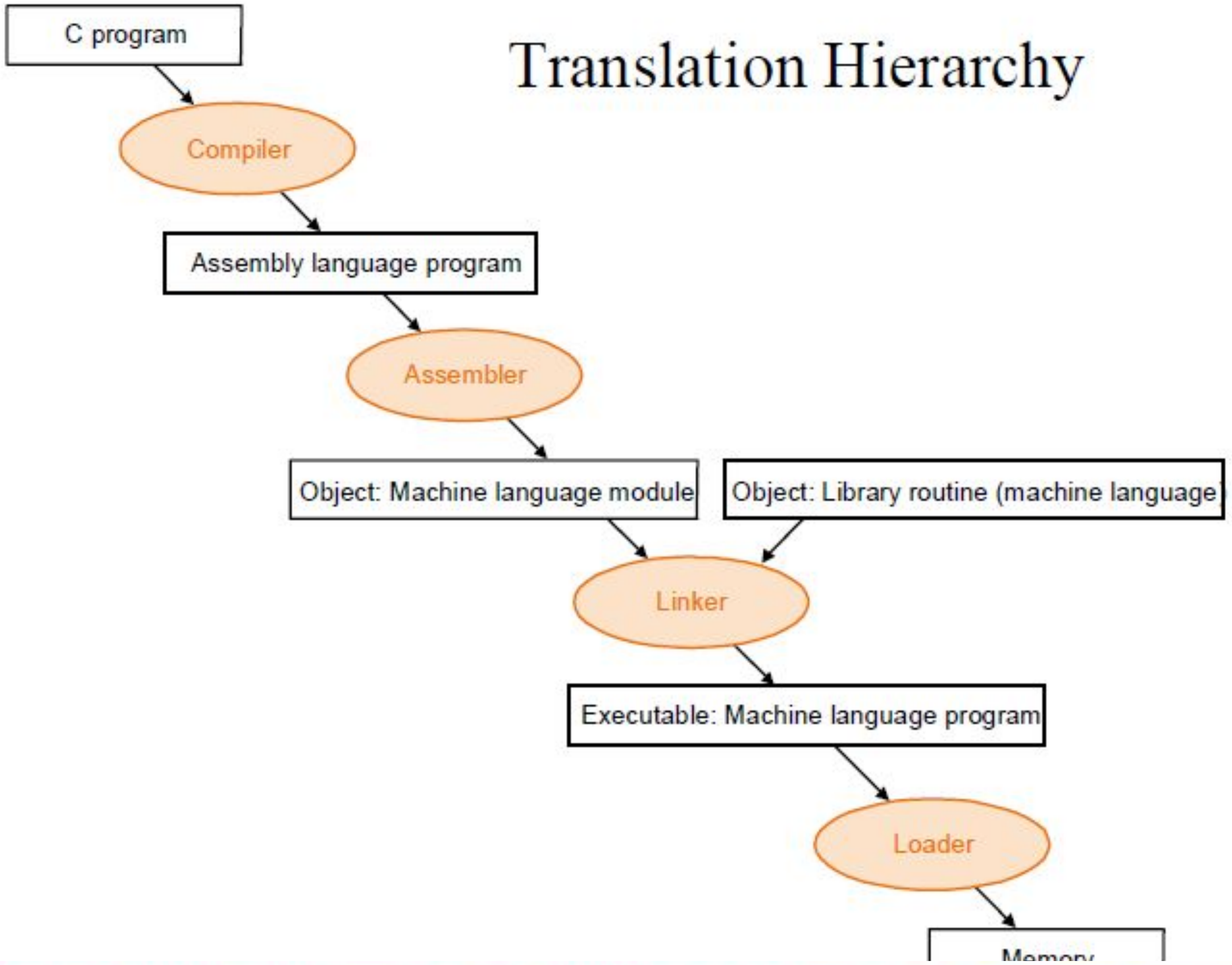
Cont…

- Memory is a device where information is stored.
- A processor is a device that perform a sequence of operations specified by instruction in memory.
- There are two types of processors
  - Input/Output (I/O) Processor concerns with transfer of data between memory and peripheral devices
  - Central Processing Unit (CPU) concerns with the manipulation of data stored in memory

Component of System Programming

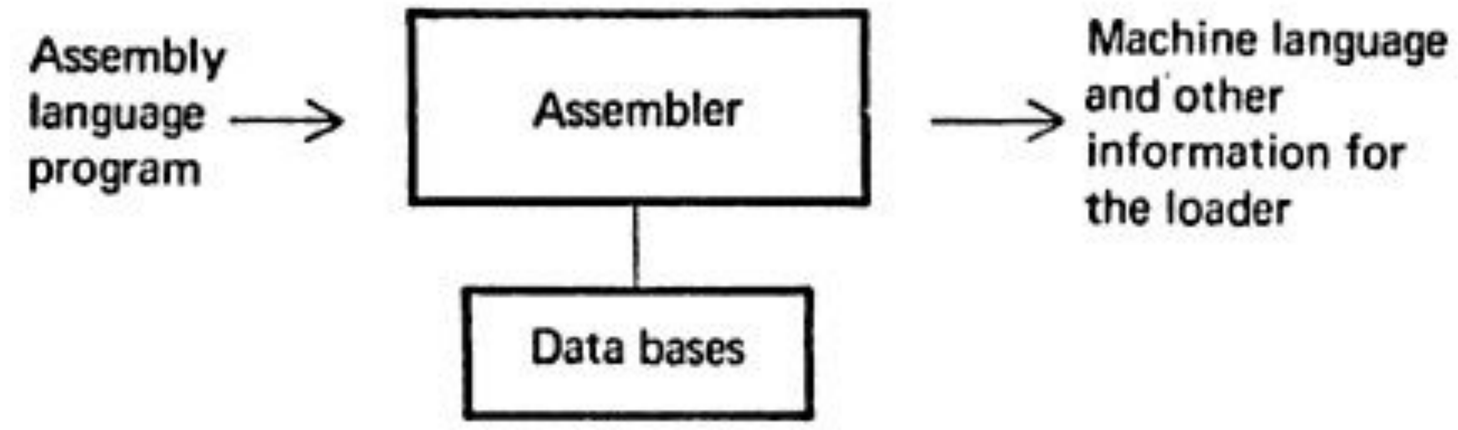
- Assemblers
- Loaders
- Linker
- Macros
- Compilers
- Formal Systems

Translation Hierarchy


C program
Translation Hierarchy
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Assembler

- An assembler is a program that accept an assembly language program and produces it machine language equivalents.


Assembly language program
Assembler
Machine language and other information for the loader
Data bases

Assembly Language

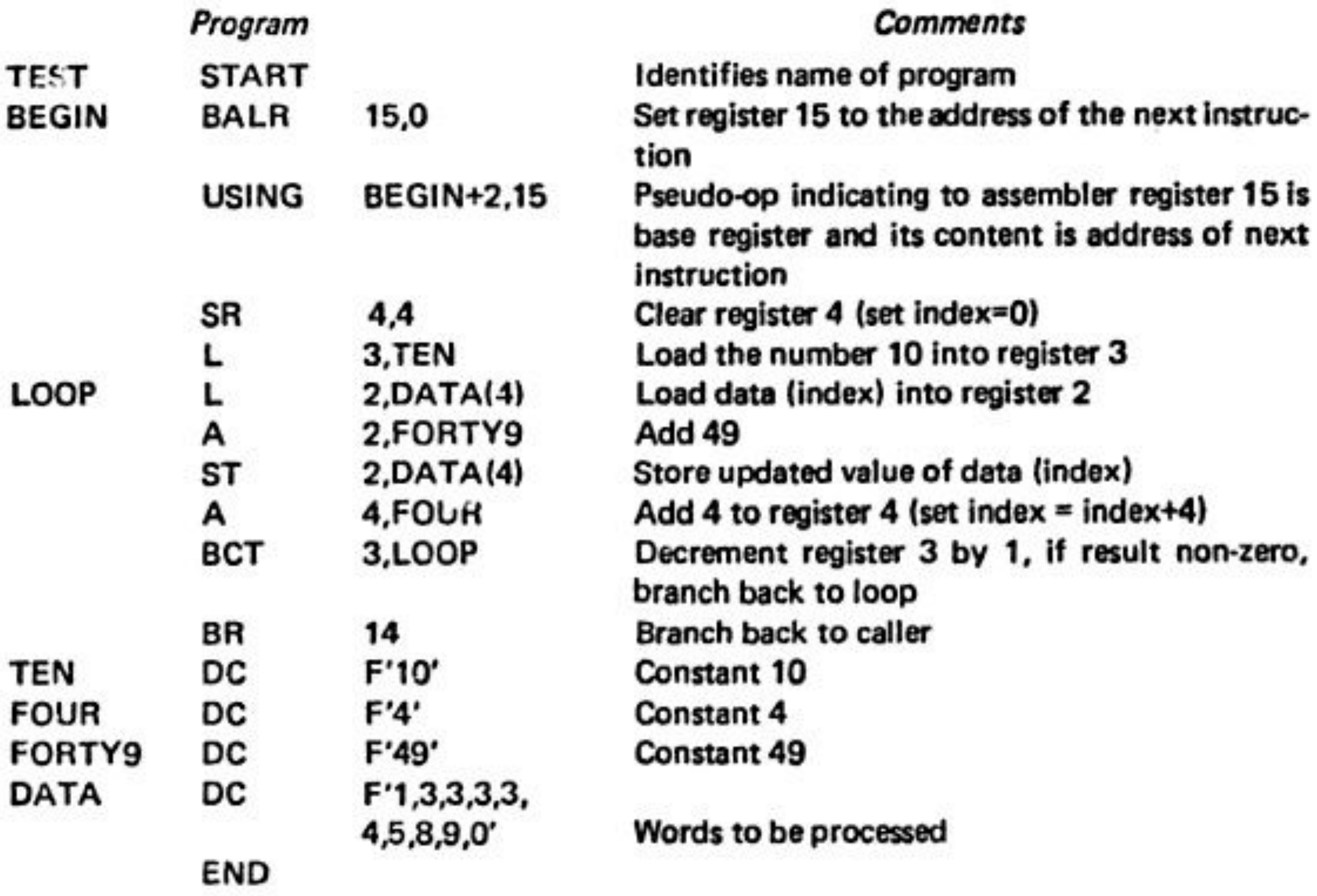
- A spectrum of languages to communicate with the computer

English	Best for Programmer
PL/I, FORTRAN	↑
…	
…	
…	
Assembly Language	
Mnemonic Machine Language	↓
Machine Language	Best for machine

Cont…

- It is the most machine dependant language used by programmer
- A mnemonics are used to understand the machine language.
- Assembly language translate the mnemonics in the machine language.

Example

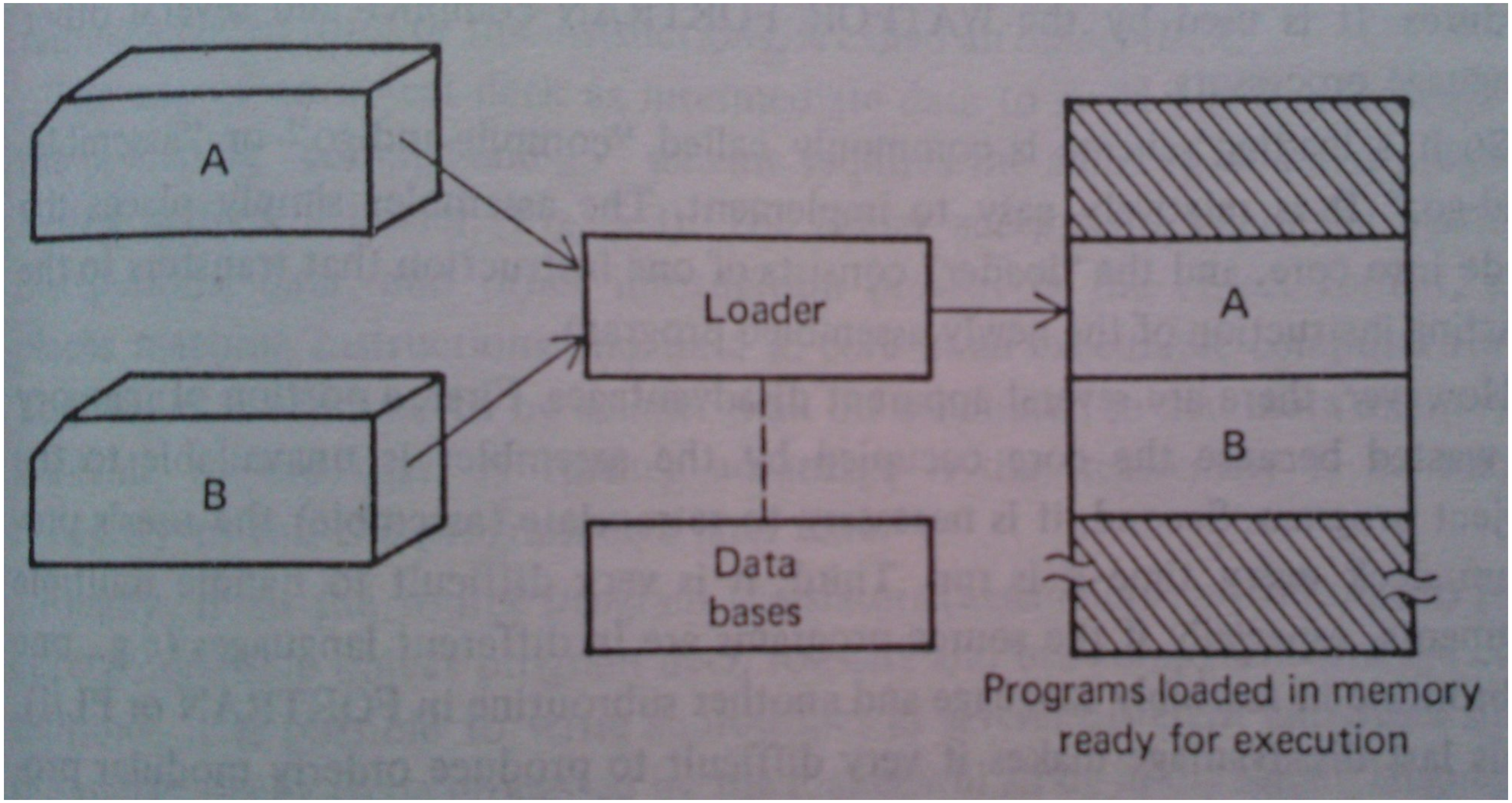


```
                Program                          Comments
TEST      START                          Identifies name of program
BEGIN     BALR      15,0                 Set register 15 to the address of the next instruc-
                                         tion
          USING     BEGIN+2,15           Pseudo-op indicating to assembler register 15 is
                                         base register and its content is address of next
                                         instruction
          SR        4,4                  Clear register 4 (set index=0)
          L         3,TEN                Load the number 10 into register 3
LOOP      L         2,DATA(4)            Load data (index) into register 2
          A         2,FORTY9             Add 49
          ST        2,DATA(4)            Store updated value of data (index)
          A         4,FOUR               Add 4 to register 4 (set index = index+4)
          BCT       3,LOOP               Decrement register 3 by 1, if result non-zero,
                                         branch back to loop
          BR        14                   Branch back to caller
TEN       DC        F'10'                Constant 10
FOUR      DC        F'4'                 Constant 4
FORTY9    DC        F'49'                Constant 49
DATA      DC        F'1,3,3,3,3,
                    4,5,8,9,0'           Words to be processed
          END
```

Loader

- Once the assembler produces an object program, the program must be placed into memory and executed.
- It is the job of the loader to assure that object programs are placed in memory in an executable form.
- There are different types of loaders like compile-and-go, absolute, direct linking, etc.

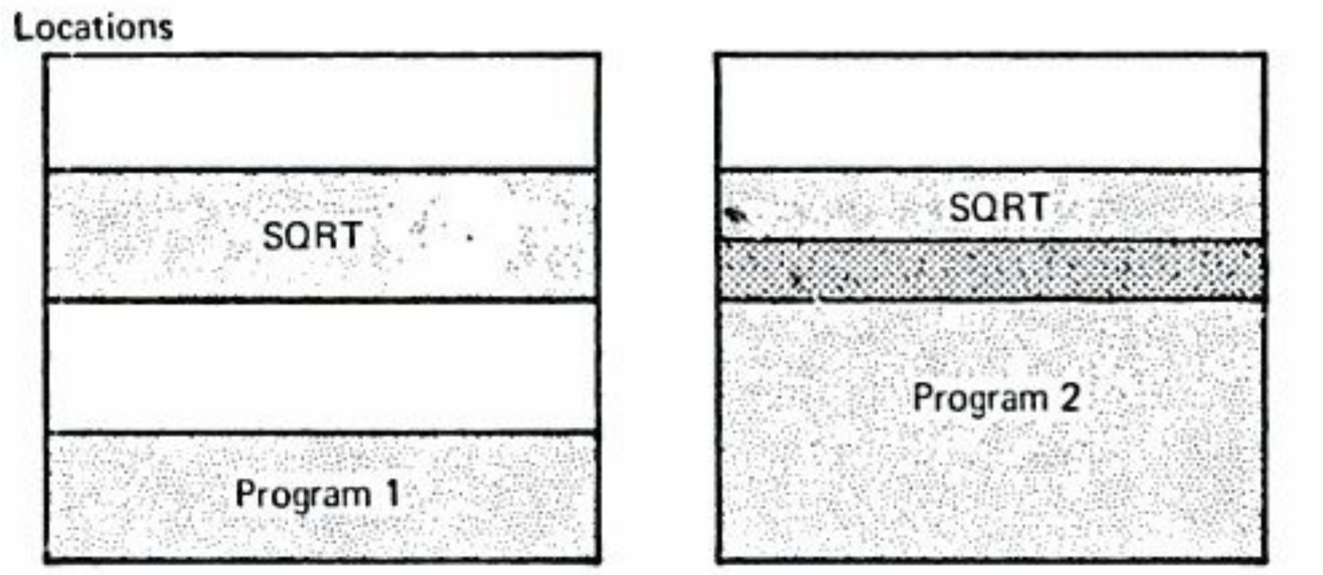
General Loading Scheme



Subroutines

- The user could write a main program that use the several other programs or subroutines.
- A subroutine is a body of computer instructions designed to be use by other routines to accomplish a task.
- There are two types of subroutines
  - Closed subroutines can be stored outside the main routine
  - Open subroutine or macro definition is inserted in main program

Example



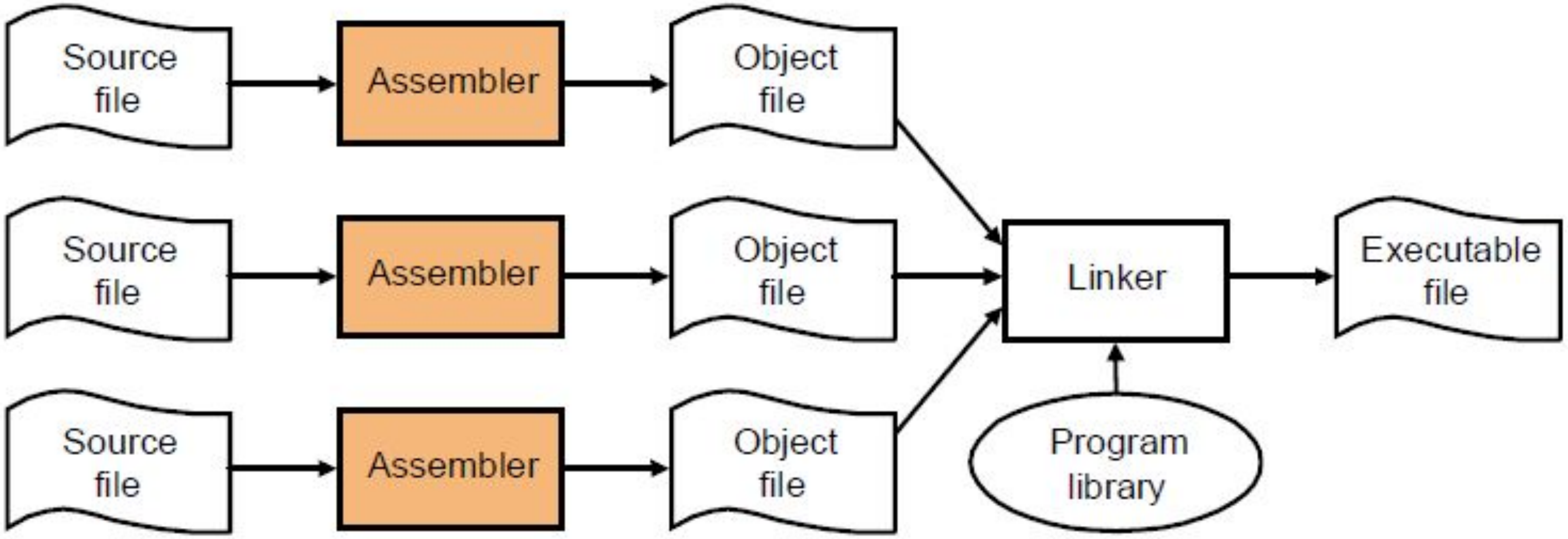
Subroutine and program are assembled together

Subroutine is assembled along with the program

Linker

- Tool that merges object files produced by separate compilation.
- Perform three task
  - Searches the program to find the library routine
  - Determine the memory location.
  - Resolve references among files.

Process for Producing Executable File



Macros

- Macro permits the programmer to define an abbreviation for a part of his program and use the abbreviation in his program

Example

	MACRO	
	INCR	
	A	1, DATA
	A	2, DATA
	A	3, DATA
	MEND	
	…	
	INCR	
	…	
	INCR	
	…	
DATA	DC	F’5’
	…	

Macro in C

```
#include<stdio.h>
#define AND &&
#define ARANGE (a>25 AND a<50)
void main()
{
     int a = 30;
     if(ARANGE)
        printf("within range");
     else
        printf("out of range");
}
```

Compilers

- The high level languages are processed by compilers and interpreters.
- A compiler is a program that accept a program written in a high level language and produces an object program.
- An interpreter is a program that appears to execute a source program.

Formal System

- It consists of an alphabets, a set of words called axioms and a finite set of relations called rules of inference.
- Formal system are used to specify the syntax and semantics of programming language.
- Examples of formal systems are
  - Set theory
  - Boolean algebra
  - Post system
  - Backus normal form

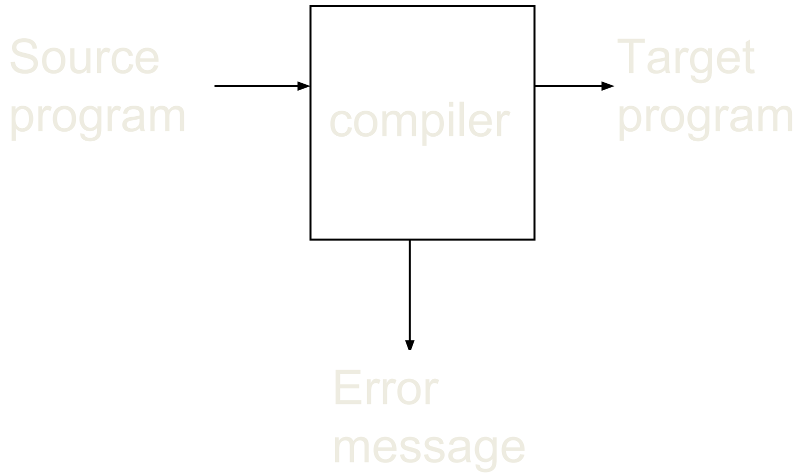
Reference

- John J. Donovan, "*System Programming*", TMH

INTRODUCTION TO COMPILERS

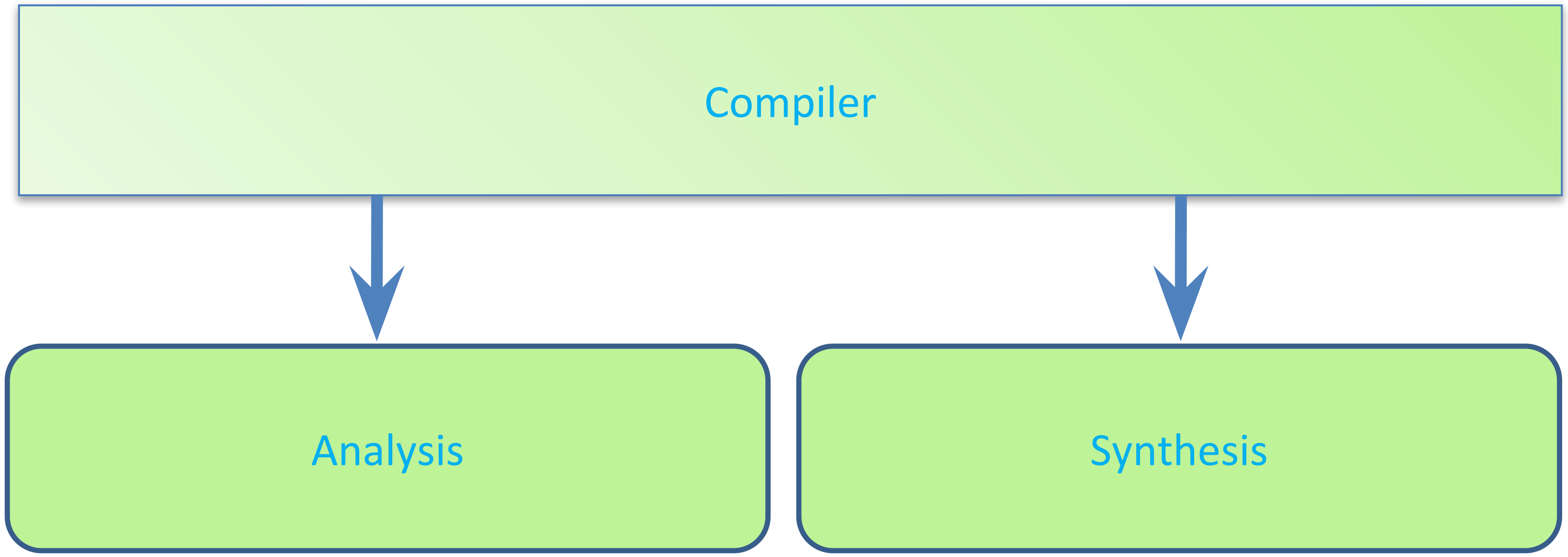
What is a Compiler?

- A **compiler** is a computer program that translates a program in a *source language* into an equivalent program in a *target language*.
- A **source program/code** is a program/code written in the source language, which is usually a high-level language.
- A **target program/code** is a program/code written in the target language, which often is a machine language or an intermediate code.



The Structure of a Compiler (1)

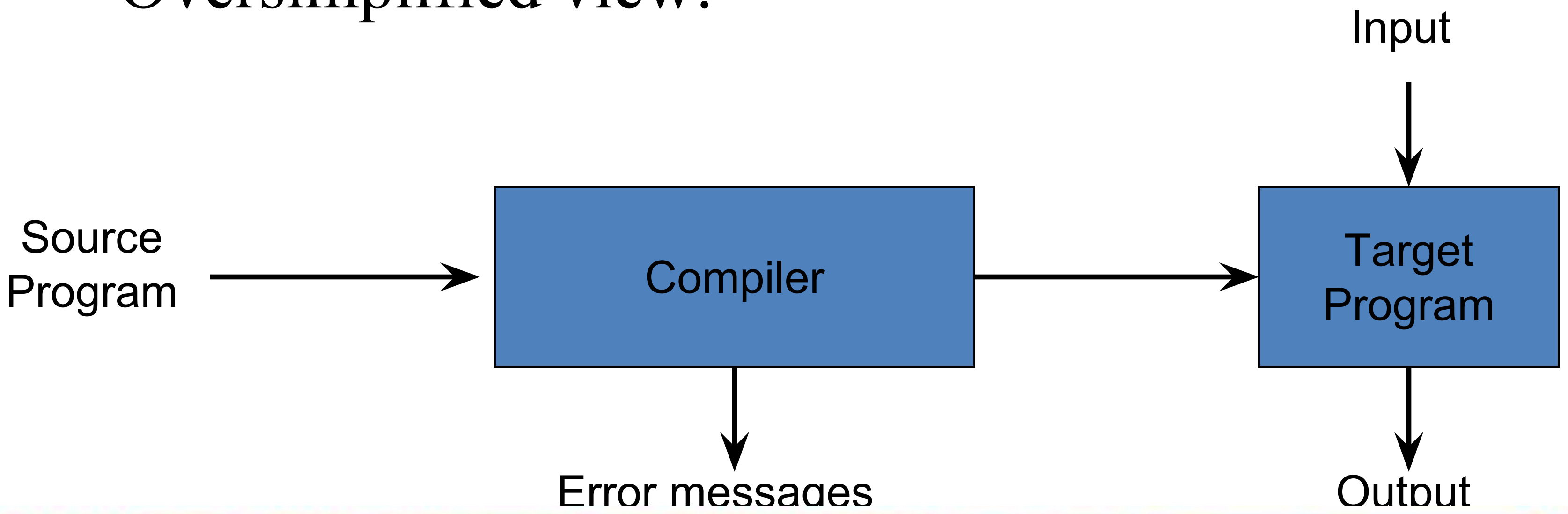
- Any compiler must perform two major tasks



  - ***Analysis*** of the source program
  - ***Synthesis*** of a machine-language program

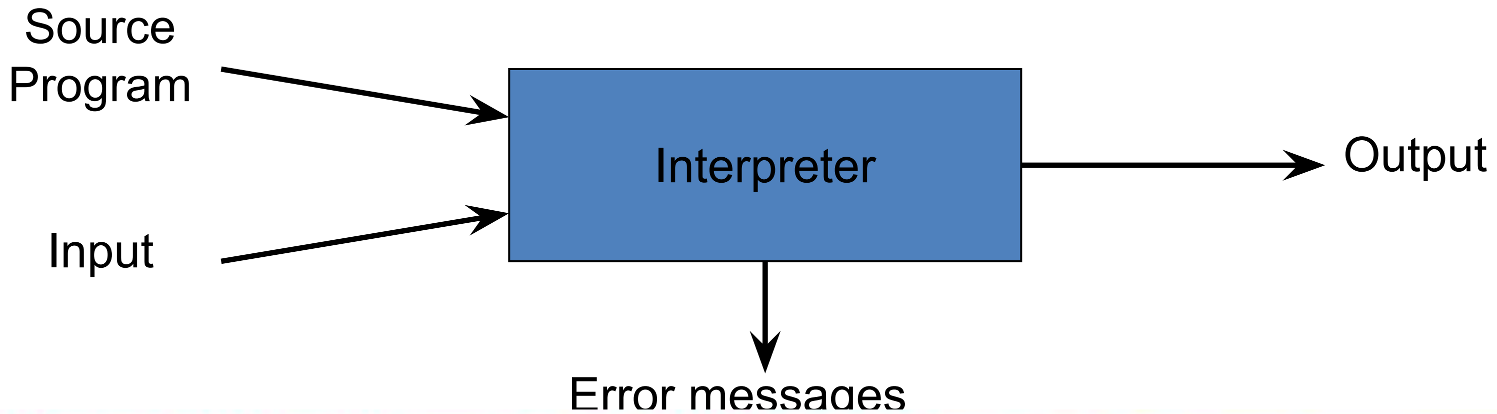
Compilers and Interpreters

- "*Compilation*"
  - Translation of a program written in a source language into a semantically equivalent program written in a target language
  - Oversimplified view:



Compilers and Interpreters (cont'd)

- "*Interpretation*"
  - Performing the operations implied by the source program
  - Oversimplified view:

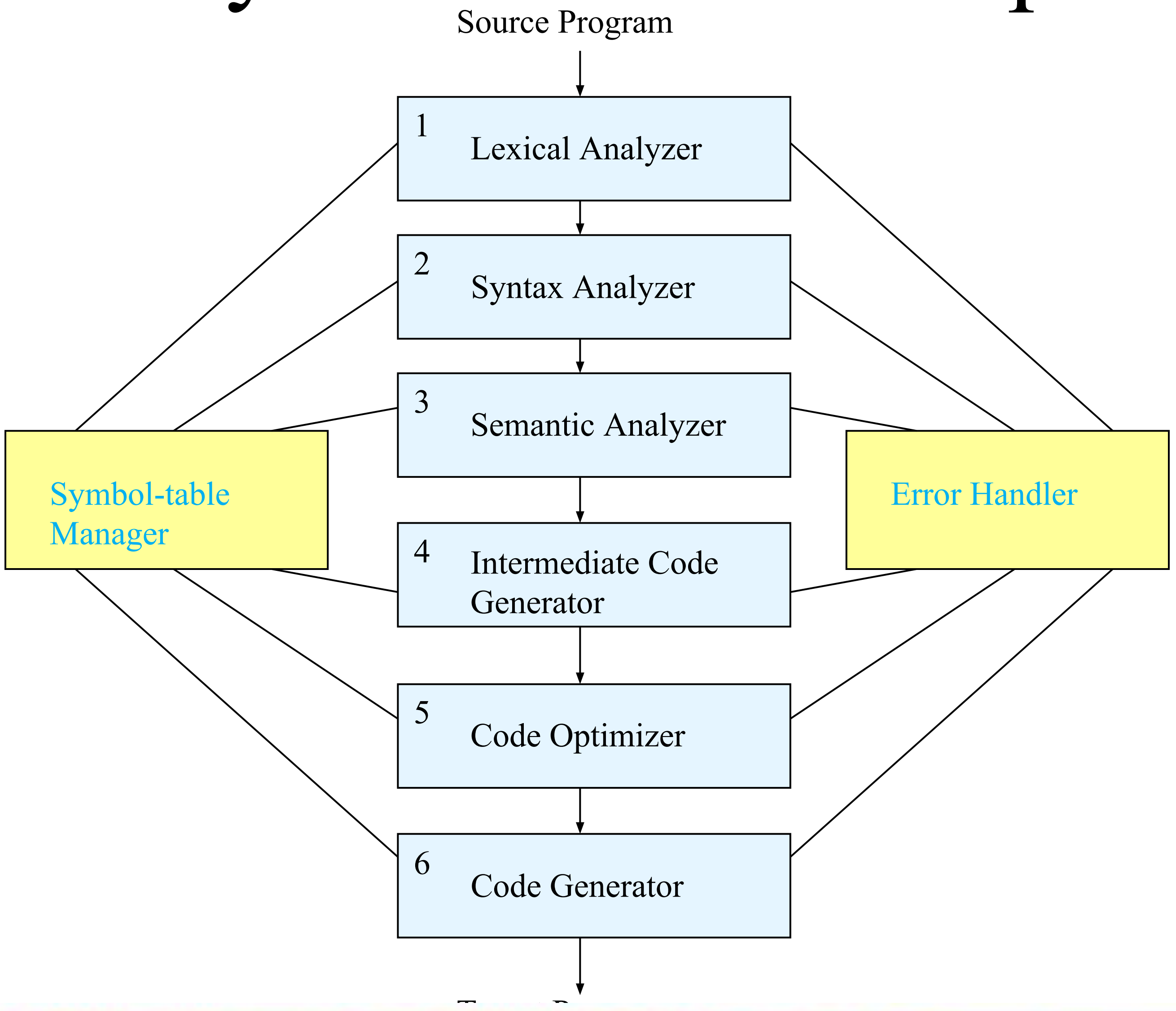


Compilers and Interpreters (cont'd)

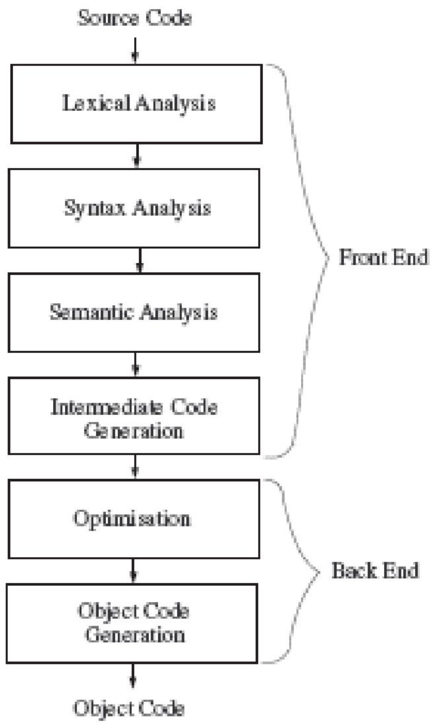
- **Compiler**: a program that translates an *executable* program in one language into an *executable* program in another language

- **Interpreter**: a program that reads an *executable* program and produces the results of running that program

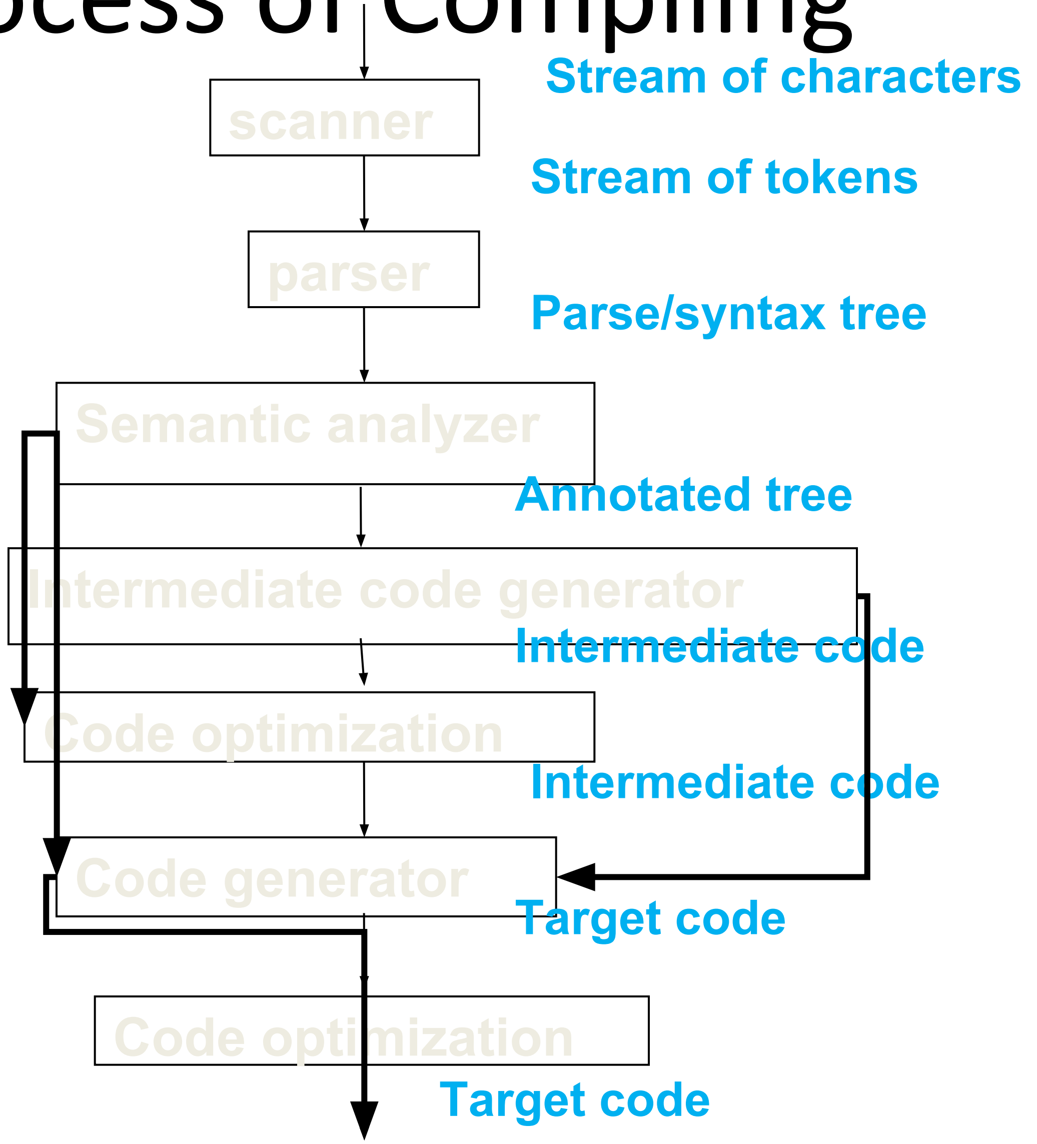
The Many Phases of a Compiler



Grouping of phases

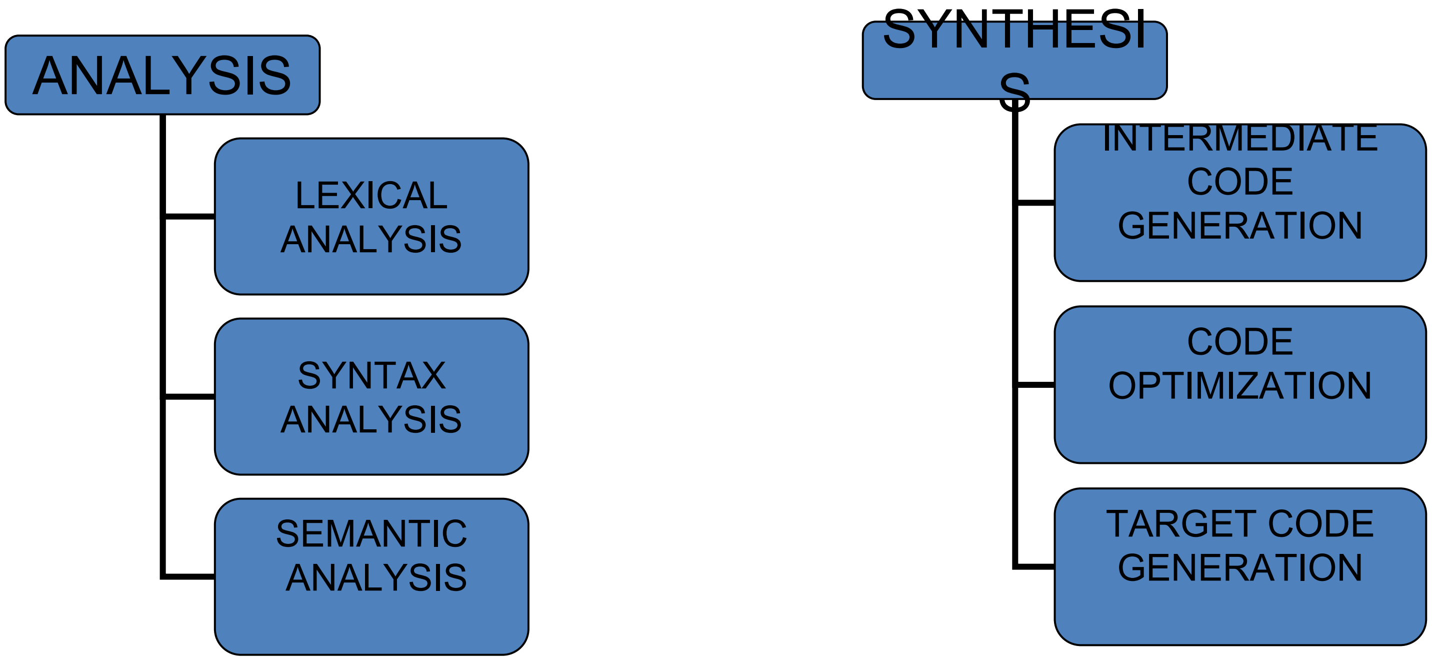


Process of Compiling



PHASE OF A COMPILER:

- Analysis of Language1
- Synthesis of Language 2



Analysis

- In compiling, analysis has three phases:

- **Linear analysis**: stream of characters read from left-to-right and grouped into *tokens*; known as **lexical analysis** or **scanning**

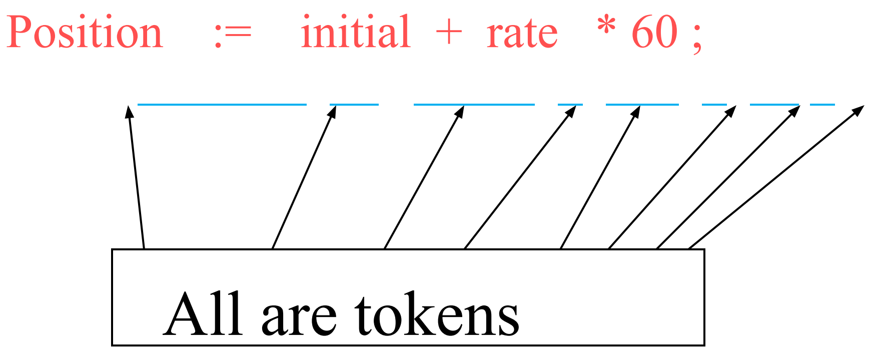
- **Hierarchical analysis**: tokens grouped hierarchically with collective meaning; known as **parsing** or **syntax analysis**

- **Semantic analysis**: check if the program components fit together meaningfully

Phase 1. Lexical Analysis

Lexical Analysis involves scanning of the source program from left to right and separating them into tokens. A token is a sequence of characters having collective meaning.

For Example:



Blanks, Line breaks, etc. are scanned out

LEXICAL ANALYZER:

Lexical Analyzer or Linear Analyzer breaks the sentence into tokens. For Example following assignment statement :-

position = initial + rate * 60

Would be grouped into the following tokens:

1. The identifier **position**.
2. The terminal symbol **=**.
3. The identifier **initial**.
4. The terminal symbol +.
5. The identifier **rate**.
6. The terminal symbol *.
7. The literal 60.

Example: Scanner

- **Input is sequence of characters.**
  - **If x>100 then y :=1 else y:=2;**

- **Output is tokens (lexemes).**
  - $\boxed{\text{if}}\ \boxed{\text{x}}\ \boxed{>}\ \boxed{\text{100}}\ \boxed{\text{then}}\ \boxed{\text{y}}\ \boxed{:=}\ \boxed{\text{1}}\ \boxed{\text{else}}\ \boxed{\text{y}}\ \boxed{:=}\ \boxed{\text{2}}\ \boxed{;}$

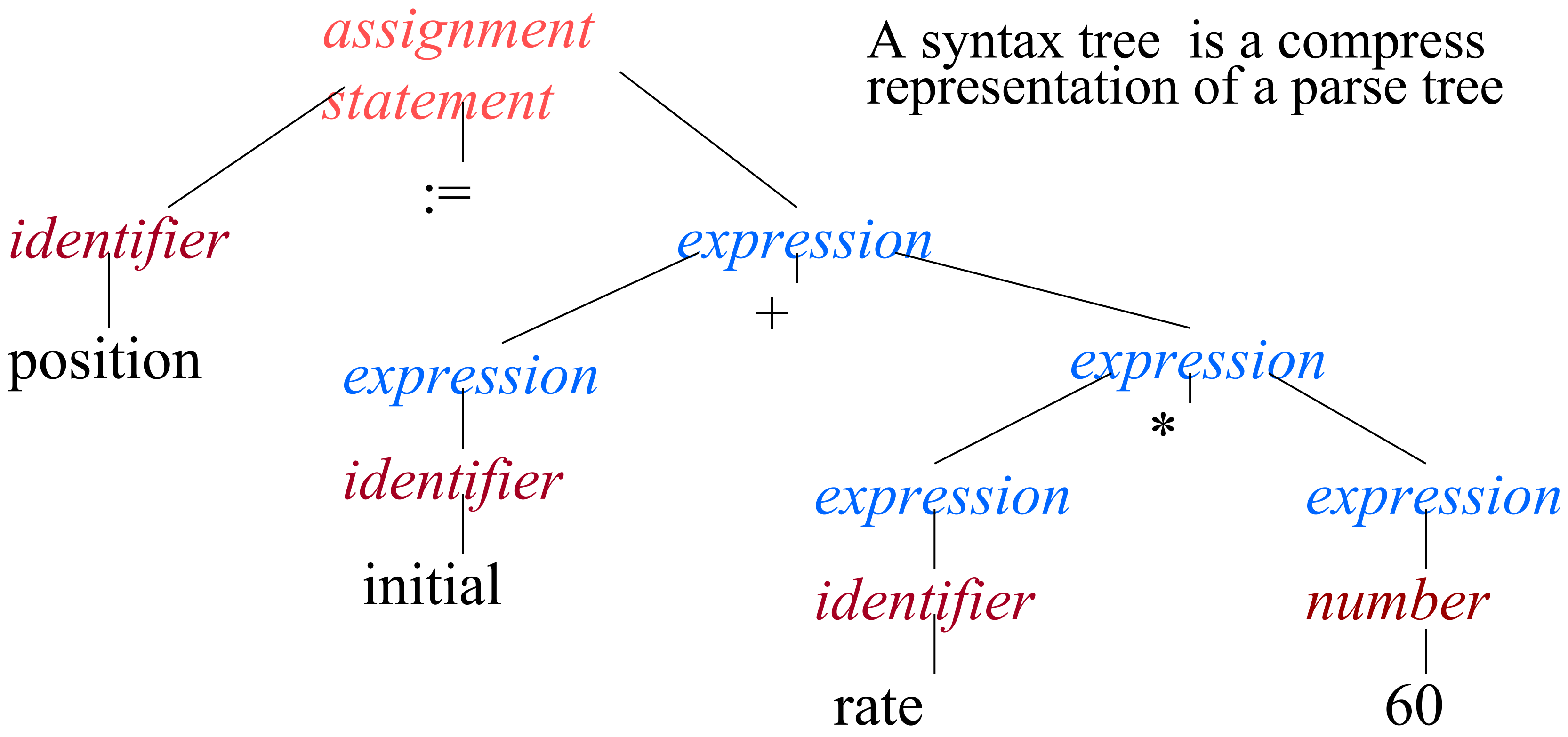
  - Terminal symbol table is a fixed table for any computer.
  - Literal table is generated during lexical analysis.
  - Symbol table is generated during lexical analysis.
  - Uniform symbol table is generated during lexical analysis.

Phase 2. Hierarchical Analysis

Parsing or Syntax Analysis

Usually a source statement is represented using a parse tree.

A syntax tree is a compress representation of a parse tree



Nodes of tree are constructed using a grammar for the language

SYNTAX ANALYSIS:

Syntax analysis is also called PARSING. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. It checks the code syntax using CFG : i.e. the set of rules .For example: if we have grammar of the form:

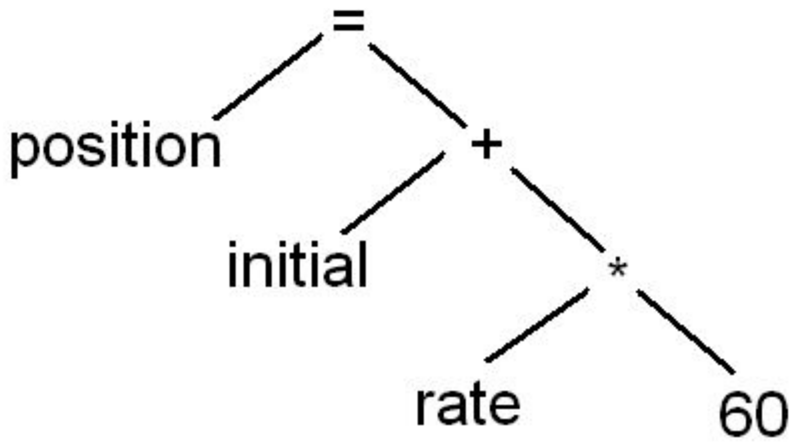
- E = E
- E = E + E
- E = E * E
- E = const.

Then corresponding parse tree derivation is:

E□ E = E□id = E+E□id = id + E*E□id = id + id*60

Parser thus consumes these tokens .If any token is left unconsumed, the parser gives an error /warning.

Following is the parse tree for the taken equation:-



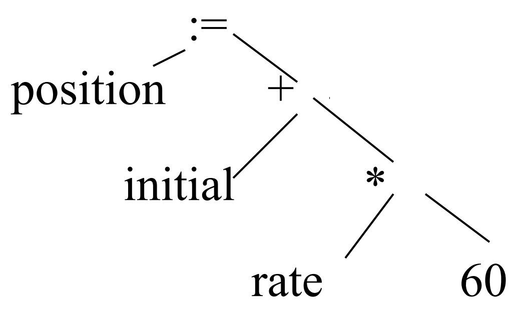
Parser parses the tree such that –if all the tokens are consumed by the parse tree, no non-terminal should be left to be expanded

In syntax tree, operators appears as internal nodes and operand as leaf nodes.

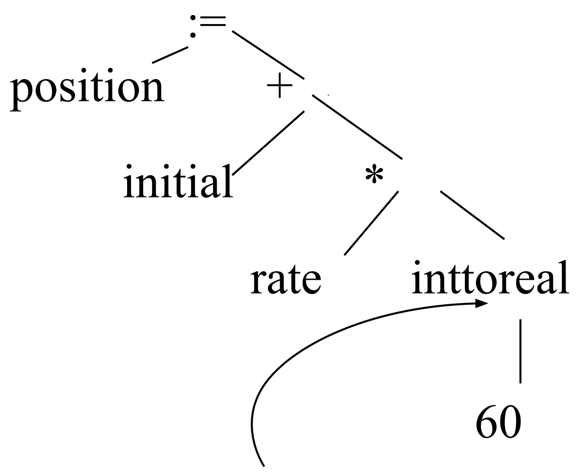
Phase 3. Semantic Analysis

- Find More Complicated Semantic Errors and Support Code Generation

- Parse Tree Is Augmented With Semantic Actions



Compressed Tree



Conversion Action

Example of semantic Errors

- 5*”ABC” (Multiplication of int and str is not permitted)
- Multiplication of two pointer p1 and p2. p1*p2 is not allowed.
- Many languages do not allow mixing of integer and real number in an expression.

SEMANTIC ANALYSIS

The semantic analysis phase checks source program for semantic errors and gathers type information for the subsequent code-generation phase . In this checks are performed to ensure that the components of a program fit together meaningfully.

For example: we have a sample code:

```
 int a; int b;
char c[ ];
a=b + c;    (Type check is done)
```

SYNTHESIS PHASE OF COMPILATION:

Phase 4 INTERMEDIATE CODE GENERATION:

- ❖ Before generation of the machine code, the compiler create an intermediate form of the source program.

- ❖ Intermediate code separates machine-independent phases (lexical, Syntax, Semantic) of a compiler form the machine-dependent phases(code generation) of a compiler.

- ❖ The intermediate representation can have a variety of form.

These form include:

1. Three address code
2. Quadruple
3. Triple
4. Postfix Notation
5. Syntax tree

e.g. z=x+y*A

1. Three address code: Each instruction has maximum 3 operands.
- temp1 = y * A
- Temp2 = x + temp1
- Z = temp2

2. Quadruple representation: Quadruple representation format

Operator	Operand 1	Operand 2	Result

Operator	Operand 1	Operand 2	Result
*	Y	A	Temp1
+	X	Temp 1	Temp 2
=	Temp 2	-	z

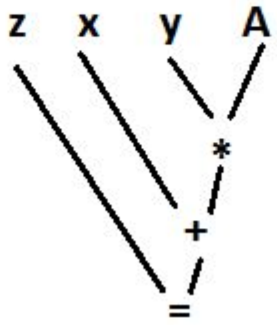
3. Triple representation:

Operator	Operand 1	Operand 2
*	Y	A
+	X	(1)
=	z	(2)

4. Postfix Notation:

Z=x+y*A is given by zxyA*+=

5. Syntax tree:



❖Phase 5 CODE OPTIMIZATION

The code optimization phase attempts to improves the intermediate code, so that faster-running machine code result. Some optimization are trivial. So the final code for example above will be:-

```
temp1 =y*A      // removed unnecessary
z=x+temp1     //variables
```

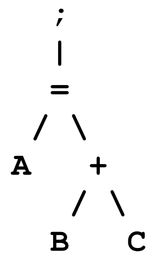
In "optimizing compilers" ,a significant amount of time is spent on this phase. How-ever ,there are simple optimizations that significantly improve the running time of the target program with out slowing down the compilation too much.

❖Phase 6 CODE GENERATION

The Final phase of the compiler is the generation of the target code, consisting normally of the relocatable machine code or assembly code. Compilers may generate many types of target codes depending on M/C while some compilers make target code only for a specific M/C. Translation of the taken code might become:

```
  Intermediate code          Assembly code
1. temp1 = y*A           MOVER    AREG, y
                     MUL          AREG, A
                     MOVEM    AREG, temp1
2. z= x + temp1                MOVER     AREG, x
                     ADD          AREG, temp1
                     MOVEM    AREG, z
```

The Phases of a Compiler

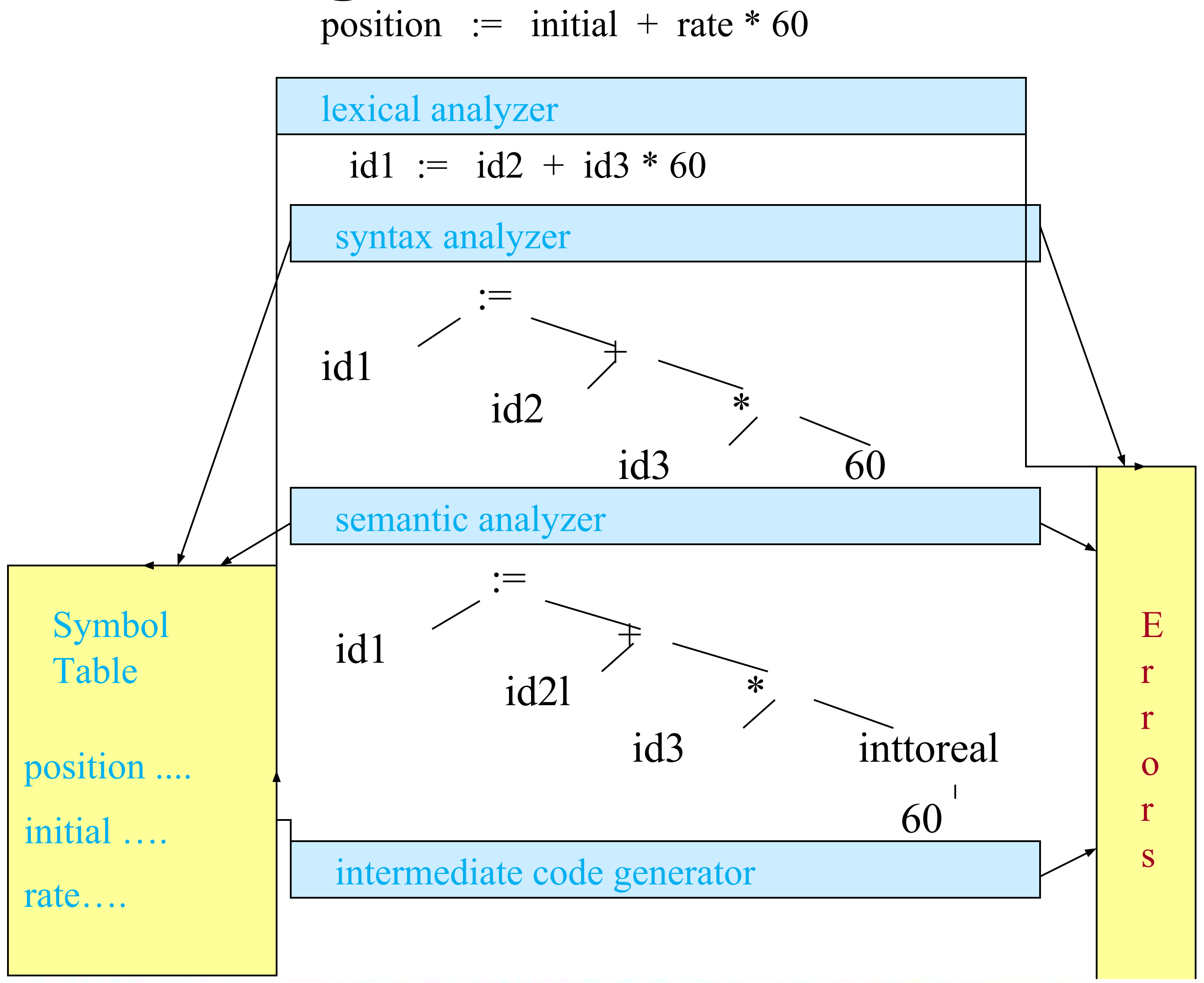
Phase	**Output**	**Sample**
Programmer (source code producer)	Source string	`A=B+C;`
Scanner (performs *lexical analysis*)	Token string	`'A', '=', 'B', '+', 'C', ';'` And *symbol table* with names
Parser (performs *syntax analysis* based on the grammar of the programming language)	Parse tree or abstract syntax tree	
Semantic analyzer (type checking, etc)	Annotated parse tree or abstract syntax tree	

Intermediate code generator	Three-address code, quads, or RTL	`int2fp B t1`
		`+ t1 C t2`
		`:= t2 A`

Optimizer	Three-address code, quads, or RTL	`int2fp B t1`
		`+ t1 #2.3 A`

Code generator	Assembly code	`MOVF #2.3,r1`
		`ADDF2 r1,r2`
		`MOVF r2,A`

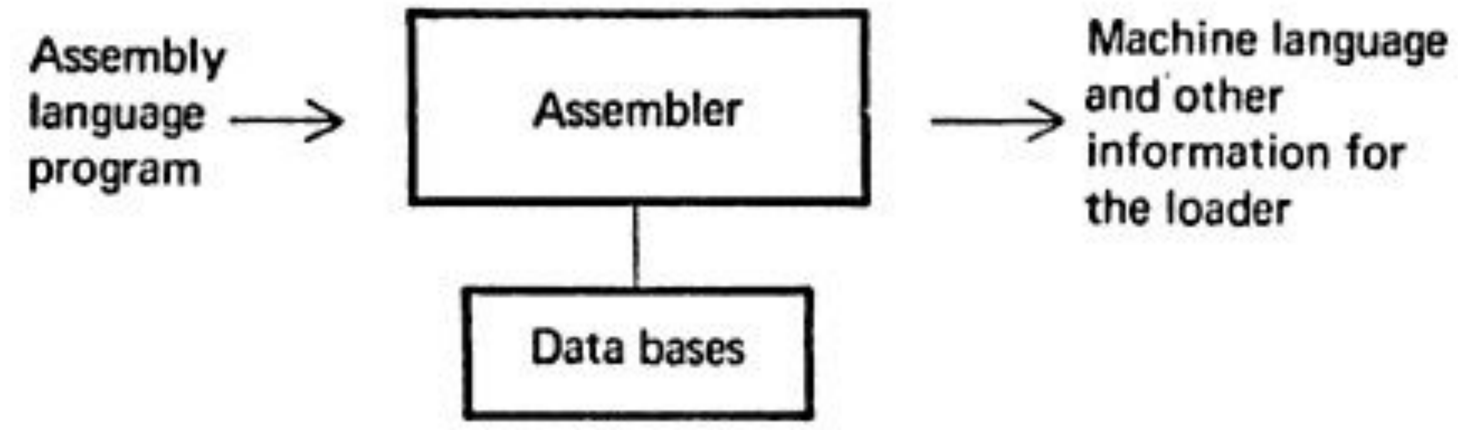
Reviewing the Entire Process



Assemblers

Introduction

- An assembler is a program that accept an assembly language program and produces it machine language equivalents


Assembly language program
Assembler
Machine language and other information for the loader
Data bases

Assembly Language

- An assembly language is a machine dependent, low level programming language which is specific to certain computer system
- Compared to machine languages, it provides three basic features
  - Mnemonic operation code
  - Symbolic operation
  - Data declaration

Cont…

- An assembly language statement has the format

  [label] <Opcode> <operand spec> [<operand spec>…]

- For example,

SR. NO	LABEL	OPCODE	OPERAND
1		MOVER	BREG, ONE
2	ONE	DC	‘1’

- A label is associated as a symbolic name

Cont…

- In assembly language, each statement has two operands,
  - First operand is always register (AREG, BREG, CREG, DREG, etc)
  - Second operand refers to a memory word using symbolic name.

Mnemonic Operations

Instruction Op-code	Assembly Mnemonic	Remarks
00	STOP	Stop Execution
01	ADD	
02	SUB	
03	MULT	
04	MOVER	Move memory to register
05	MOVEM	Move register to memory
06	COMP	Set condition code
07	BC	Branch on condition
08	DIV	Analogous to SUB
09	READ	
10	PRINT	

Cont…

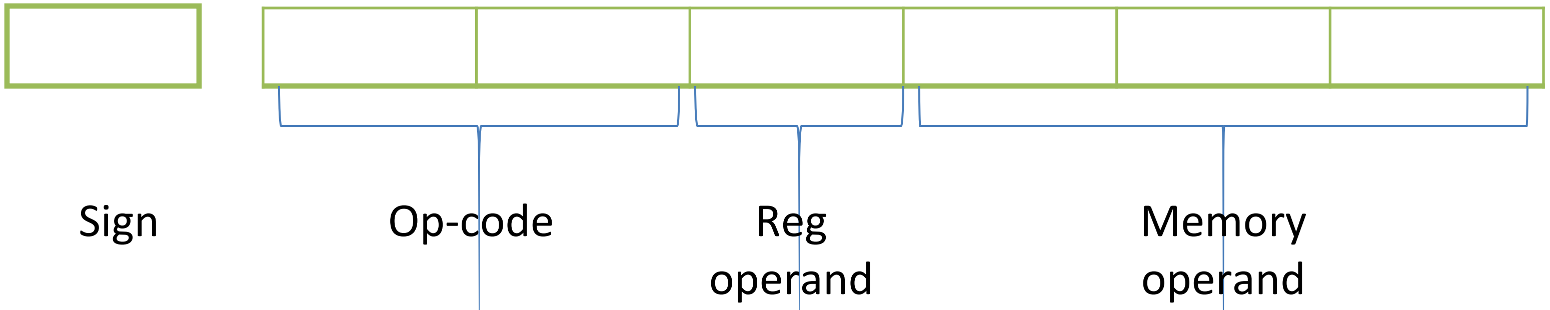
- The MOVE instruction move a value between a memory word and a register.
- All arithmetic is performed in a register
- A comparison instruction sets a condition code analog to subtract code.
- The condition code can be tested by a BC instruction. Its format is

  BC <condition code>, <memory address>

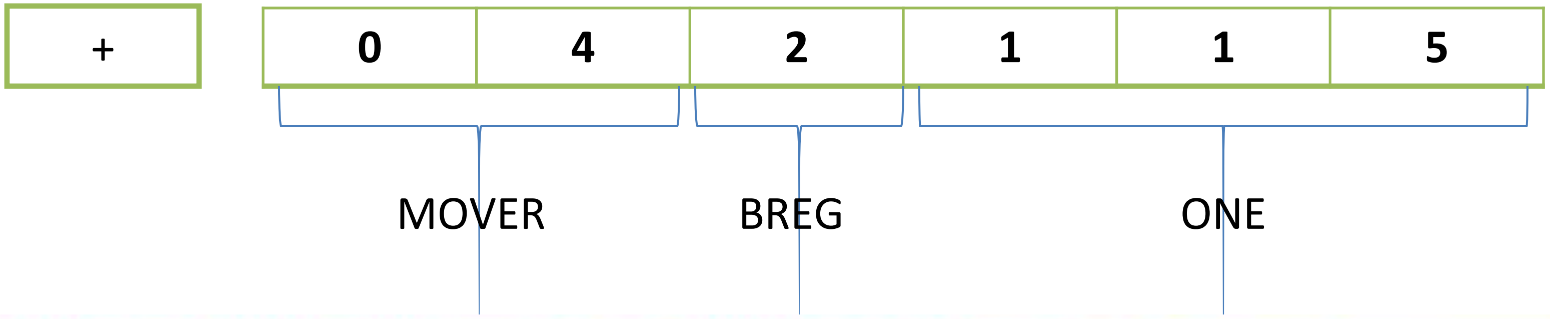
  where , condition code is a simple character string like LT, LE, EQ, GT, GE and ANY

Machine Instruction Format

- The machine instruction format is



- For example



Cont…

- The register operands are

Instruction Op-code	Register
1	AREG
2	BREG
3	CREG
4	DREG

A Simple Assembly Language

	START	**101**		
	READ	N	101	+ 09 0 112
	MOVER	BREG, ONE	102	+ 04 2 114
	MOVEM	BREG, TERM	103	+ 05 2 115
AGAIN	MULT	BREG, TERM	104	+ 03 2 115
	MOVER	CREG, TERM	105	+ 04 3 115
	ADD	CREG, ONE	106	+ 01 3 114
	COMP	CREG, N	107	+ 06 3 112
	BC	LE, AGAIN	108	+ 07 2 104
	MOVEM	BREG, RESULT	109	+ 05 2 113
	PRINT	RESULT	110	+ 10 0 113
	STOP		111	+ 00 0 000
N	DS	1	112	
RESULT	DS	1	113	
ONE	DC	'1'	114	+ 00 0 001

Assembly Language Statement

- An assembly language program contains three kind of statements
  - Imperative statement
  - Declarative statement
  - Assembler directives

Imperative Statement

- It indicates the actions to be performed.
- For example,

Sr. No.	Op-code	Operand	Meaning
1	MOVER	BREG, ONE	Move memory to register
2	MOVEM	BREG, TERM	Move register to memory
3	MULT	BREG, TERM	Perform multiplication
4	ADD	CREG, ONE	Perform addition
5	COMP	CREG, N	Comparison statement
6	BC	LE, AGAIN	Conditional statement
7	PRINT	RESULT	Printing operation

Declarative Statement

- The syntax of declarative statement is as

  [label] DS <constant>

  [label] DC '<value>'

- For example

Sr. No.	Label	Op-code	Operand	Meaning
1	N	DS	1	Reserve a memory of 1 word
2	RESULT	DS	1	Reserve a memory of 1 word
3	ONE	DC	'1'	Memory word store constant
4	TERM	DS	200	Reserve a memory of 200 word

Cont…

- An assembly program can use constants in two ways
  - As immediate operand:
    - This feature is provided by the target machine
    - For example, ADD   AREG, 5
  - As literals:
    - A literal is an operand with the syntax =‘<value>’
    - For example, ADD   AREG, =‘5’

Example

	START	**200**	
	MOVER	AREG, ='5'	200
	MOVEM	AREG, A	201
LOOP	MOVER	AREG, A	203
	…		
	LTORG		
		='5'	211
		='1'	212
	…		
	BC	LT, BACK	215
	…		
BACK	EQU	LOOP	
B	DS	1	218
	END		

Assembler Directives

- Assembler directives provides some directives to the assembler to perform certain actions.
- For example,

  START <constant>

- This directive indicates, assembler should be start with memory address <constant>

Advantages of Assembly Language

- It is mnemonic
- Reading is easier
- Ease of programming than machine language

Disadvantage of Assembly Language

- The disadvantage of assembly language is that it requires the use of an assembler to translate a source program into object code

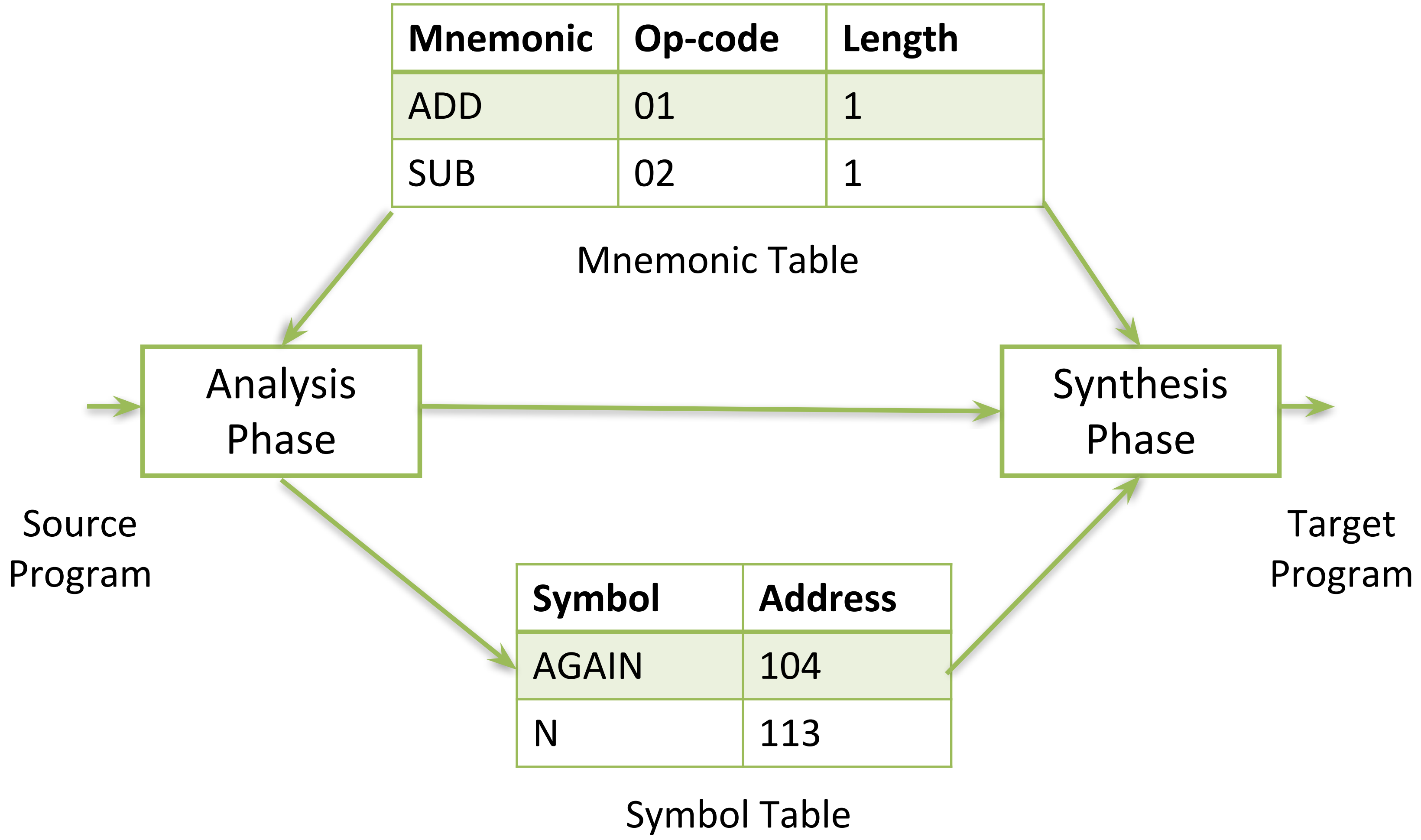
Design Specification of An Assembler

- It is four step approach to develop a design specification for an assembler
  - Identify the information to perform a task
  - Design a suitable data structure to record information
  - Determine processing to <u>obtain and maintain the information</u>
  - Determine the processing to <u>perform the task</u>.

Cont…

- There are two phases in assembly language programming
  - Analysis Phase:
    - Isolate the labels, mnemonics and operands
    - Generate symbol table (SYMTAB)
    - Validate mnemonic through mnemonic table
  - Synthesis Phase:
    - Obtain the machine code from mnemonic table
    - Obtain the address of operand from symbol table

Cont…



Mnemonic	Op-code	Length
ADD	01	1
SUB	02	1

Mnemonic Table

Symbol	Address
AGAIN	104
N	113

Symbol Table

Example

	START	**101**		
	READ	N	101	+ 09 0 112
	MOVER	BREG, ONE	102	+ 04 2 114
	MOVEM	BREG, TERM	103	+ 05 2 115
AGAIN	MULT	BREG, TERM	104	+ 03 2 115
	MOVER	CREG, TERM	105	+ 04 3 115
	ADD	CREG, ONE	106	+ 01 3 114
	COMP	CREG, N	107	+ 06 3 112
	BC	LE, AGAIN	108	+ 07 2 104
	MOVEM	BREG, RESULT	109	+ 05 2 113
	PRINT	RESULT	110	+ 10 0 113
	STOP		111	+ 00 0 000
N	DS	1	112	
RESULT	DS	1	113	
ONE	DC	'1'	114	+ 00 0 001

Cont…

Mnemonic	Op-code	Length
ADD	01	1
SUB	02	1
MULT	03	1
MOVER	04	1
MOVEM	05	1
COMP	06	1
BC	07	1
DIV	08	1
READ	09	1
PRINT	10	1

Mnemonic Table

Symbol	Address
AGAIN	104
N	113
RESULT	114
ONE	115
TERM	116

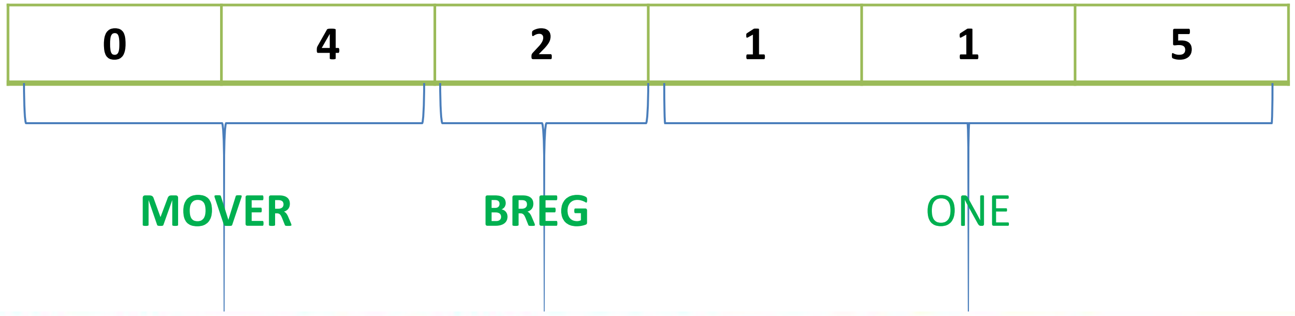
Symbol Table

Synthesis Phase

- Consider the assembly statement

  MOVER BREG, ONE

- To synthesize the machine instruction we must have
  - Address of the memory word of symbol ONE
  - Machine operation code of MOVER



Analysis Phase

- The primary function performed by the analysis phase is the building of the symbol table.

Symbol Table Generation

- The symbol table has two fields

Symbol	Address

- To determine the address of symbols, first fix the address of all program element. This is called as memory allocation.
- The memory allocation is done by the help of LC

Cont…

- To allocate the memory the LC is used
- LC always contains the address of the next memory word
- LC is initialize with the START statement
- LC is updated by the length of the instruction
- The Length of the instruction is obtained from the mnemonic table.

Example

	START	101		
	READ	N	101	+ 09 0 112
	MOVER	BREG, ONE	102	+ 04 2 114
	MOVEM	BREG, TERM	103	+ 05 2 115
AGAIN	MULT	BREG, TERM	104	+ 03 2 115
	MOVER	CREG, TERM	105	+ 04 3 115
	ADD	CREG, ONE	106	+ 01 3 114
	COMP	CREG, N	107	+ 06 3 112
	BC	LE, AGAIN	108	+ 07 2 104
	MOVEM	BREG, RESULT	109	+ 05 2 113
	PRINT	RESULT	110	+ 10 0 113
	STOP		111	+ 00 0 000
N	DS	1	112	
RESULT	DS	1	113	
ONE	DC	'1'	114	+ 00 0 001

Cont…

Symbol	Address
AGAIN	104
N	113
RESULT	114
ONE	115
TERM	116

Symbol Table

Pass Structure of Assembler

- Generally, there are two types of assemblers
  - Single pass assembler
  - Two pass assembler

Single Pass Assembler

- It passes over the source file only once
- During the pass, it performs all operations like collecting the labels, address and literals.
- The analysis and synthesis of program is done in single pass.

Cont…

- The problem of the single pass assembler is to tackle the forward references
- For example,

	START	**101**		
	…			
	MOVER	BREG, ONE	102	+ 04 2 114
	MOVEM	BREG, TERM	103	+ 05 2 115
	…			
	…			
ONE	DC	‘1’	114	+ 00 0 001
TERM	DS	1	115	

Two Pass Assembler

- Two pass assembler can handle forward references easily.
- LC processing is performed in the first pass and symbols defined in the program are entered in the symbol table.
- In effect, the first pass perform the analysis of program while the second pass perform the synthesis.

Design of A Two-Pass Assembler

- Task performed by the two pass assembler are
  - Pass I:
    - Separate the symbol, mnemonic and operand fields
    - Build the symbol table
    - Perform LC processing
    - Construct intermediate representation
  - Pass II:
    - Synthesize the target program

Advanced Assembler Directives

- ORIGIN
  - Syntax: ORIGIN   <address spec>
  - Set LC to the address given by <address spec>
- EQU
  - Syntax: <symbol>EQU   <address spec>
  - Define the symbol to represent address given by <address spec>
- LTORG
  - It permit a programmer to specify where literals should be placed

Example

	START	**200**	
	MOVER	AREG, ='5'	200
	MOVEM	AREG, A	201
LOOP	MOVER	AREG, A	202
	…		
	LTORG		
		='5'	211
		='1'	212
	…		
	ORIGIN	LOOP+2	
	MULT	CREG, B	204
	…		
BACK	EQU	LOOP	
B	DS	1	218
	END		

Pass-I of the Assembler

- Pass-I uses
  - Machine Operation Table(OPTAB)
  - Symbol Table(SYMTAB)
  - Literal Table(LITTAB)

Cont…

Mnemonic op-code	Class	Mnemonic info

OPTAB

Sybmol	Address	Length

SYMTAB

Literal	address

LITTAB

Pass-I Algorithm

```
1.  loc_cntr := 0; (default value)
    pooltab_ptr :=1; POOLTAB[1]:=1;
    littab_ptr:=1;
2.  While next statement is not an END statement
    a)  If label is present then
            this_label:= symbol in label field;
            Enter(this_label, loc_cntr) in SYMTAB.
    b)  If an LTORG statement then
        i.    Process literals
              LITTAB[POOLTAB[pooltab_ptr]…LITTAB[lit_tab_ptr-1] to
              allocate memory and put the address in the address field. Update
              location counter  accordingly.
        ii.   pooltab_ptr := pooltab_ptr +1;
        iii.  POOLTAB[pooltab_ptr]:=littab_ptr;
```

Cont…

```
c) If START or ORIGIN statement then
        loc_cntr := value specified in the operand field;
d) If an EQU statement then
    i.   this_addr := value of <address_spec>;
    ii.  Correct the symbtab entry for this_label to
         (this_label,this_addr)
e) If a declaration statement then
    i.   code:= machine opcode from OPTAB;
    ii.  size := size of memory are required by DC/DS
    iii. loc_cntr := loc_cntr + size;
    iv.  Generate IC ‘(DL, code)…’
```

Cont…

```
f) If an imperative statement then
   i.   code:= machine opcode from OPTAB;
   ii.  loc_cntr := loc_cntr + instruction length from
        OPTAB;
   iii. If operand is a literal then
        i.   this_literal := literal in operand field;
        ii.  LITTAB[littab_ptr]:= this_literal;
        iii. littab_ptr= littab_ptr +1;
   iv.  else (i.e. operand is a symbol)
        i.   this_entry := SYMTAB entry number of operand
        ii.  Generate IC '(IS,code)(S,this_entry)';
```

Cont…

3. Processing of END statement
   a) Perform step 2(b)
   b) Generate IC
   c) Go to pass II

Example

	START	**200**	
	READ	A	200
	READ	B	201
	MOVER	AREG, =‘5’	202
	MOVER	AREG, A	203
	ADD	AREG, B	204
	SUB	AREG, =‘6’	205
	MOVEM	AREG, C	206
	PRINT	C	207
	LTORG		
	MOVER	AREG, =‘15’	210
	MOVER	AREG, A	211
	ADD	AREG, B	212
	SUB	AREG, =‘16’	213
	DIV	AREG, =‘26’	214

Cont…

Sybmol	Address	Length
A	216	1
B	217	1
C	218	1

Literal	address
=‘5’	208
=‘6’	209
=‘15’	220
=‘16’	221
=‘26’	222

Pass-II Algorithm

```
1.  code_area_address := address of code_area;
    Pooltab_ptr :=1;
    Loc_cntr:=0;
2.  While next statement is not an END statement
    a)  Clear machine_code_buffer;
    b)  If an LTORG statement
        i.   Process literals in
             LITTAB[POOLTAB[pooltab_ptr]]…LITTAB[POOLTAB[pooltab_ptr+
             1]]-1 similar to processing of constants in a DC statement i.e. assemble
             the literals in machine_code_buffer.
        ii.  size := size of memory area required for literals;
        iii. pooltab_ptr:= pooltab_ptr +1;
```

Cont…

c) If a START or ORIGIN statement then
   i. loc_cntr := value specified in operand field;
   ii. size:=0;

d) If a declaration statement
   i. If a DC statement then
      Assemble the constant in machine_code_buffer.
   ii. size: = size of memory area required by DC/DS;

f) If an imperative statement
   i. Get operand address from SYMTAB or LITTAB.
   ii. Assemble instruction in machine_code_buffer.
   iii. size: = size of instruction;

Cont…

f) if size not equal to 0 then
   i. Move contents of Machine_code_buffer to the address code_area_address + loc_cntr;
   ii. loc_cntr := loc_cntr + size;

4. (Processing of END statement)
   a) Perform steps 2(b) and 2(f).
   b) Write code_area into output file.

Example

	START	200		
	READ	A	200	+09 0 216
	READ	B	201	+09 0 217
	MOVER	AREG, ='5'	202	+04 1 208
	MOVER	AREG, A	203	+04 1 216
	ADD	AREG, B	204	+01 1 217
	SUB	AREG, ='6'	205	+02 1 209
	MOVEM	AREG, C	206	+05 1 218
	PRINT	C	207	+10 0 218
	LTORG			
	MOVER	AREG, ='15'	210	+04 1 220
	MOVER	AREG, A	211	+04 1 216
	ADD	AREG, B	212	+01 1 217
	SUB	AREG, ='16'	213	+02 1 221
	DIV	AREG, ='26'	214	+08 1 222

Exercise

- For give source program generate symbol table, literal table

	START	**100**	
A	DS	3	100
L1	MOVER	AREG, B	103
	ADD	AREG, C	104
	MOVEM	AREG, D	105
D	EQU	A+1	
L2	PRINT	D	106
	ORIGIN	A-1	
C	DC	‘5’	099
	ORIGIN	L2+1	
	STOP		107

References

- D. M. Dhamdhere, “System Programming and Operating System”, TMH