Compiler Design
Subject Code : CS3501

Compilers – Principles, Techniques and Tools
2nd Edition by Alfred V. Aho, Ravi Sethi, Jeffery D. Ullman

Dr. C. Dhaya

Professor

Department of Computer Science & Engineering

Adhiparasakthi Engineering College

Melmaruvathur - 603319

What do we expect to achieve by the end of the course?

- Students are confident that they can use language processing technology for various software developments
- Students are confident that they can design, develop, understand, modify/enhance and maintain compilers

History and Milestones

- Until 1952
  - Most programming in assembly language
- 1952
  - Grace Hopper writes first compiler for the A-0 programming language
- 1957-58
  - John Backus and team writes first Fortran compiler
  - Optimization was an integral component of the compiler

About the Nature of Compiler Algorithms

- Makes practical application of
  - Greedy algorithms - register allocation
  - Heuristic search - list scheduling
  - Graph algorithms - dead code elimination, register
  - allocation
  - Dynamic programming - instruction selection
  - Optimization techniques - instruction scheduling
  - Finite automata - lexical analysis
  - Pushdown automata - parsing
  - Fixed point algorithms - data-flow analysis
  - Complex data structures - symbol tables, parse trees, data
  - dependence graphs
  - Computer architecture - machine code generation

Unit - I
Introduction to Compilers

1. Translators – Compilers and Interpreters
2. Language Processors
3. Phases of a compiler
4. Errors Encountered in different Phases
5. Grouping of Phases
6. Compiler Construction tools
7. Programming Language Basics

Translators



- A translator is a computer program that performs **the translation of a program written in a one programming language into a different programming language**, without losing the functional or logical structure of the original code (ie. the "essence" of original program).

One Programming Language (High Level Languages)

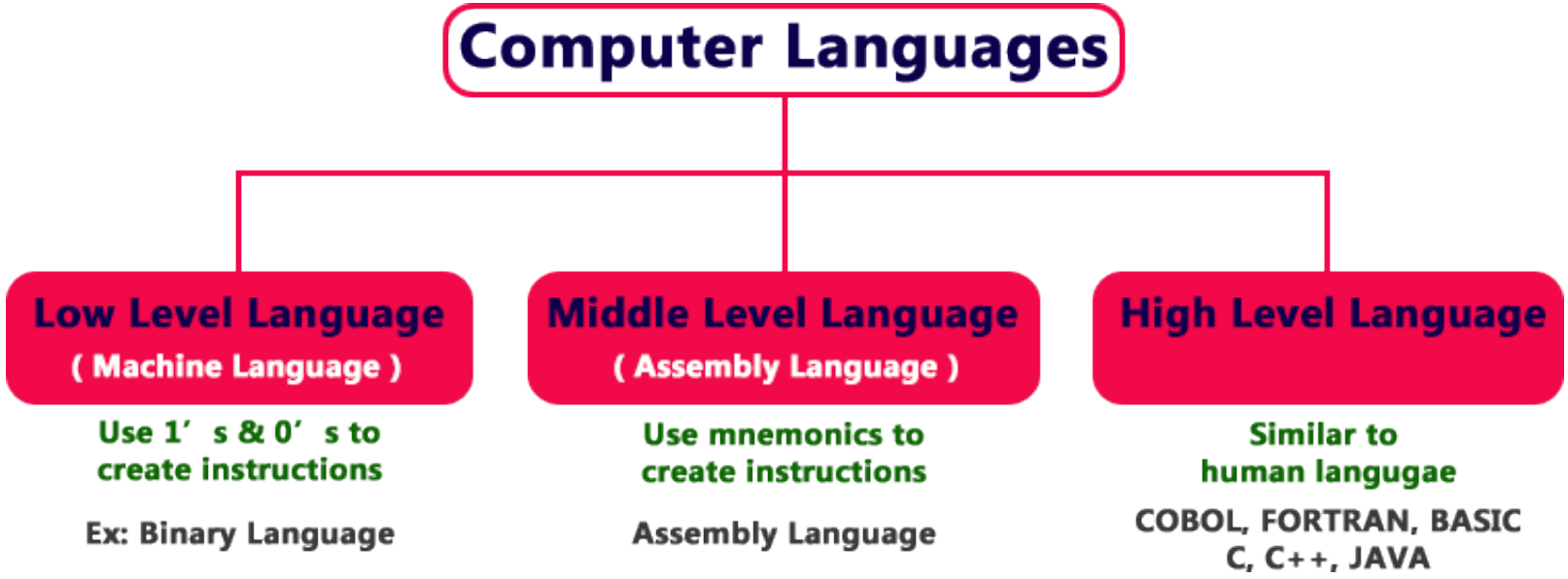
TRANSLATOR

Other Programming language (Low Level Languages)

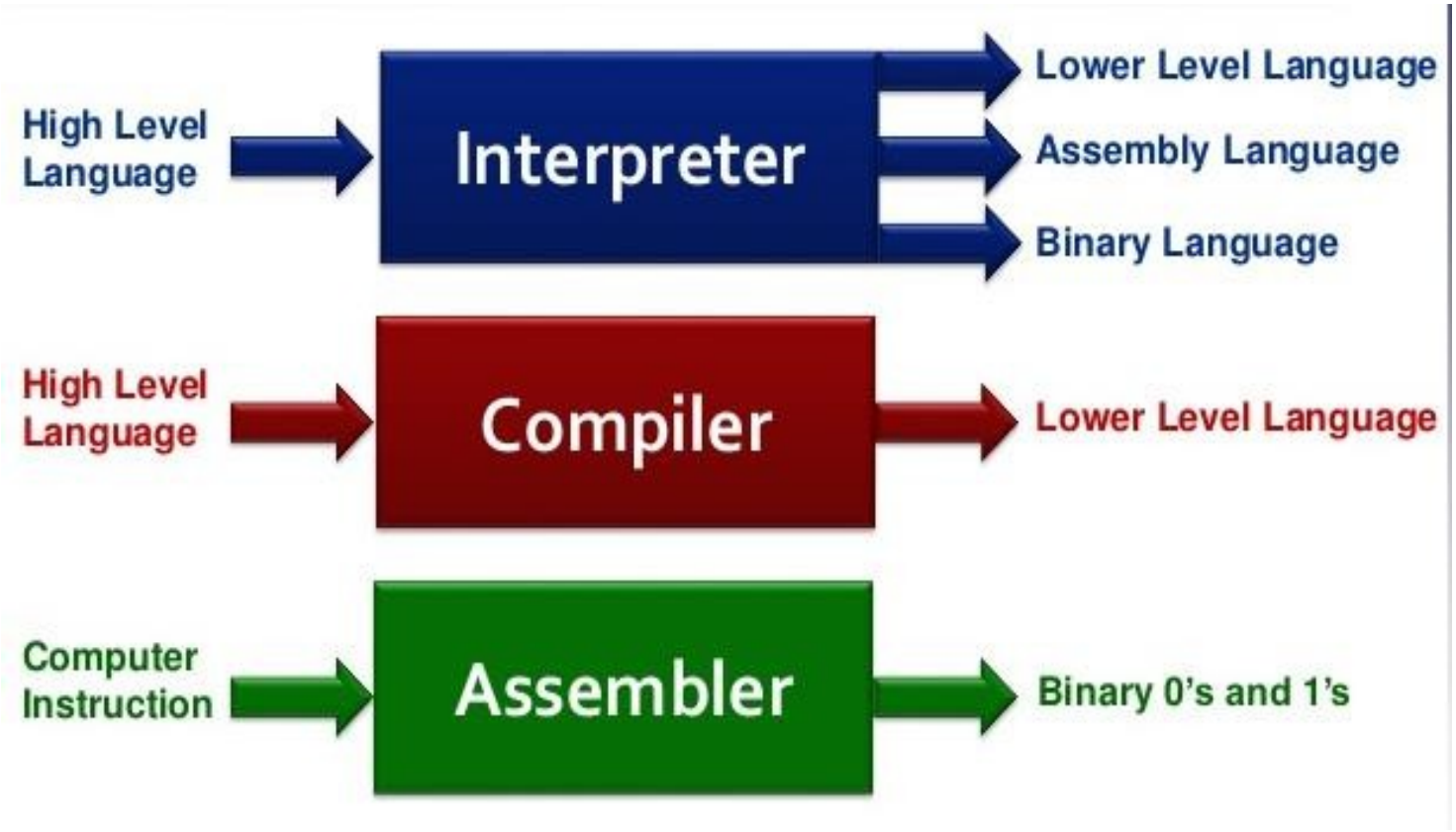


Computer Languages

- Computer languages are the languages through which user can communicate with the computer by writing program instructions.



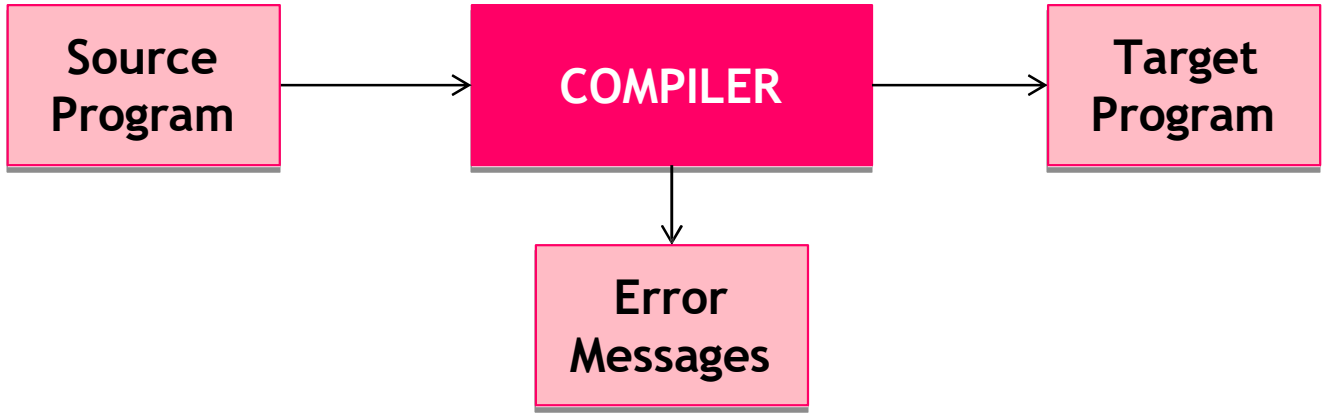
Types of Translators – contd…



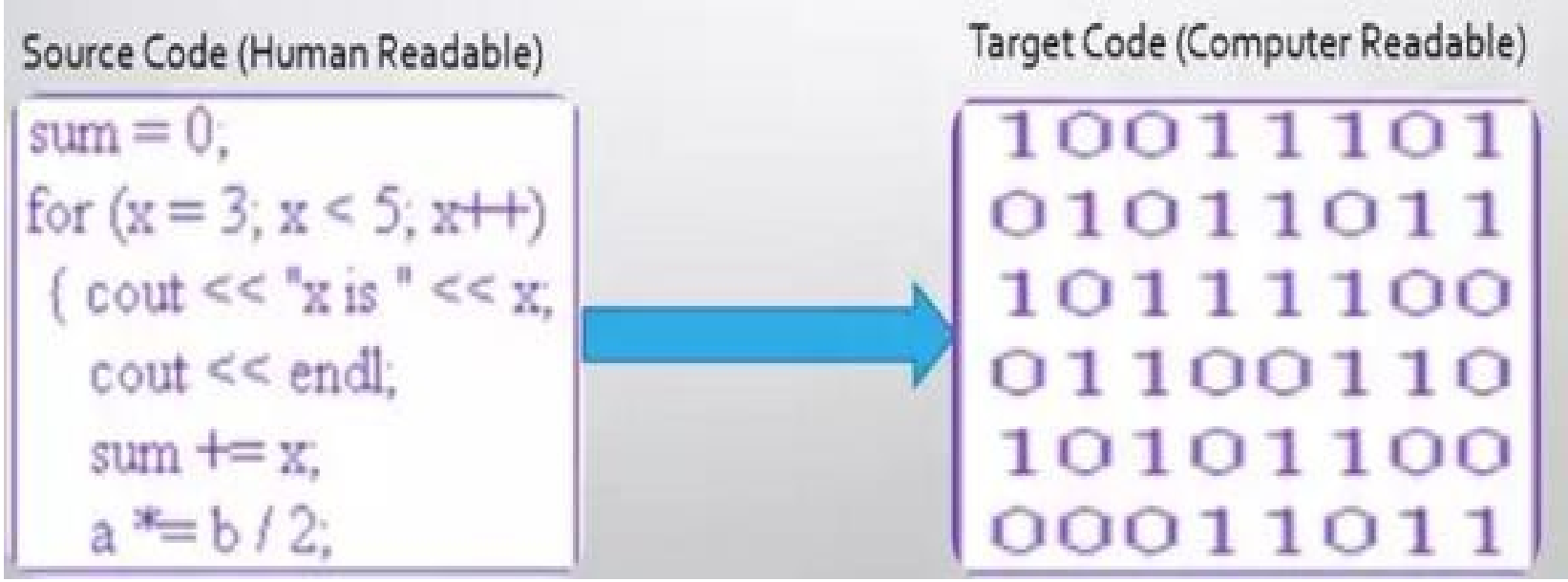
Introduction to Compiling

2 mark

- Compiler is a program that reads a program written in one language **(the source language)** and translates it into another language **(the target language)**.
  - First compiler : started 1950's (took 18 yrs)

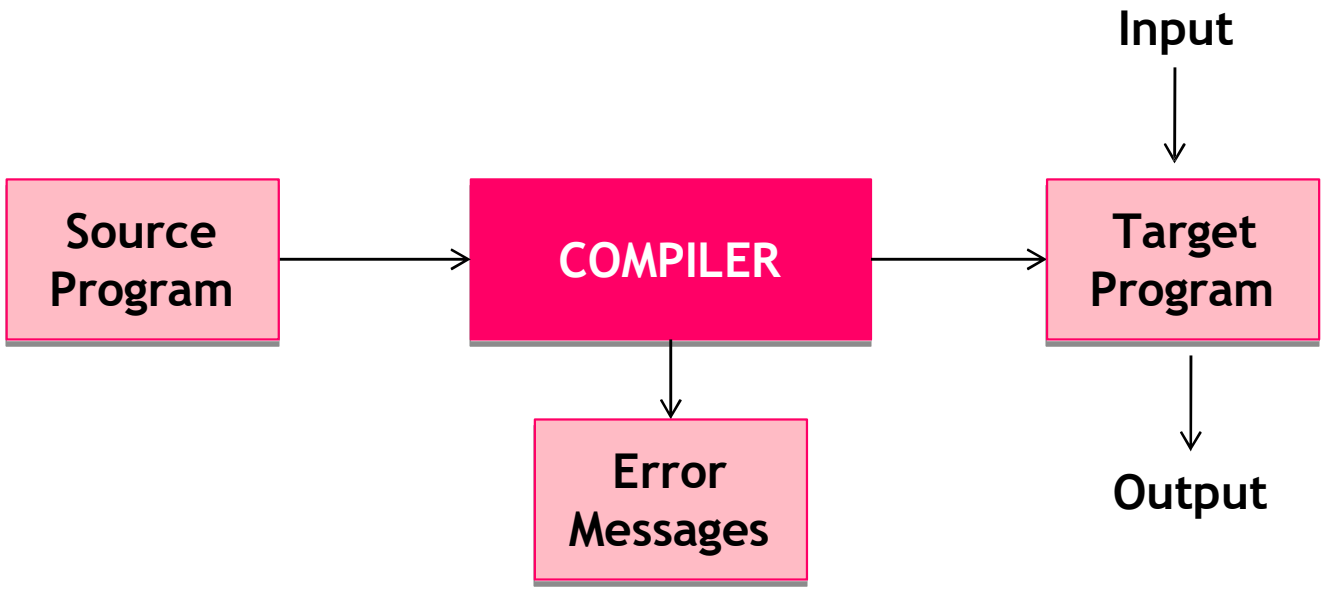


Contd...

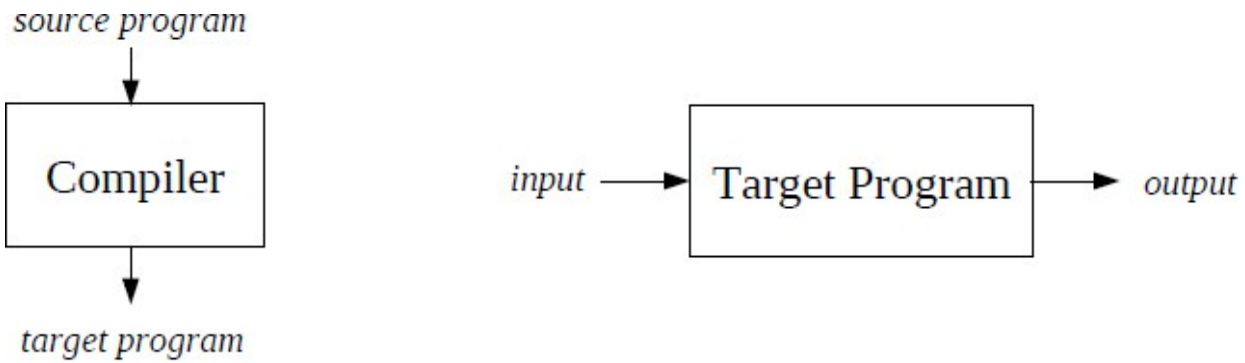


Source Program : Traditional Programming Languages (Fortran and Pascal)
Target Program : Any another programming language (or) the machine language

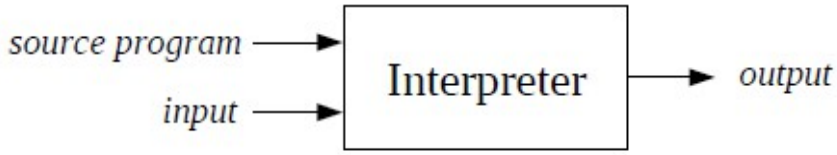
Contd...


Input
Source Program
COMPILER
Target Program
Error Messages
Output

Compiler Vs. Interpreter

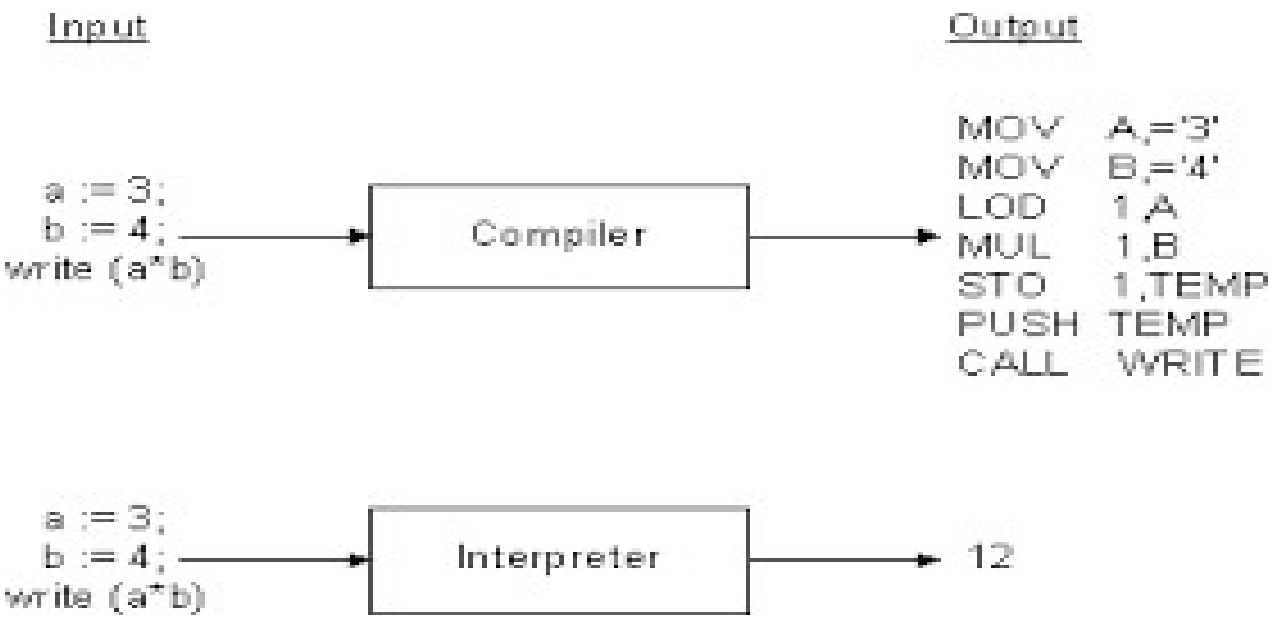


1. Execution of a compiled program



2. Execution of an interpreted program

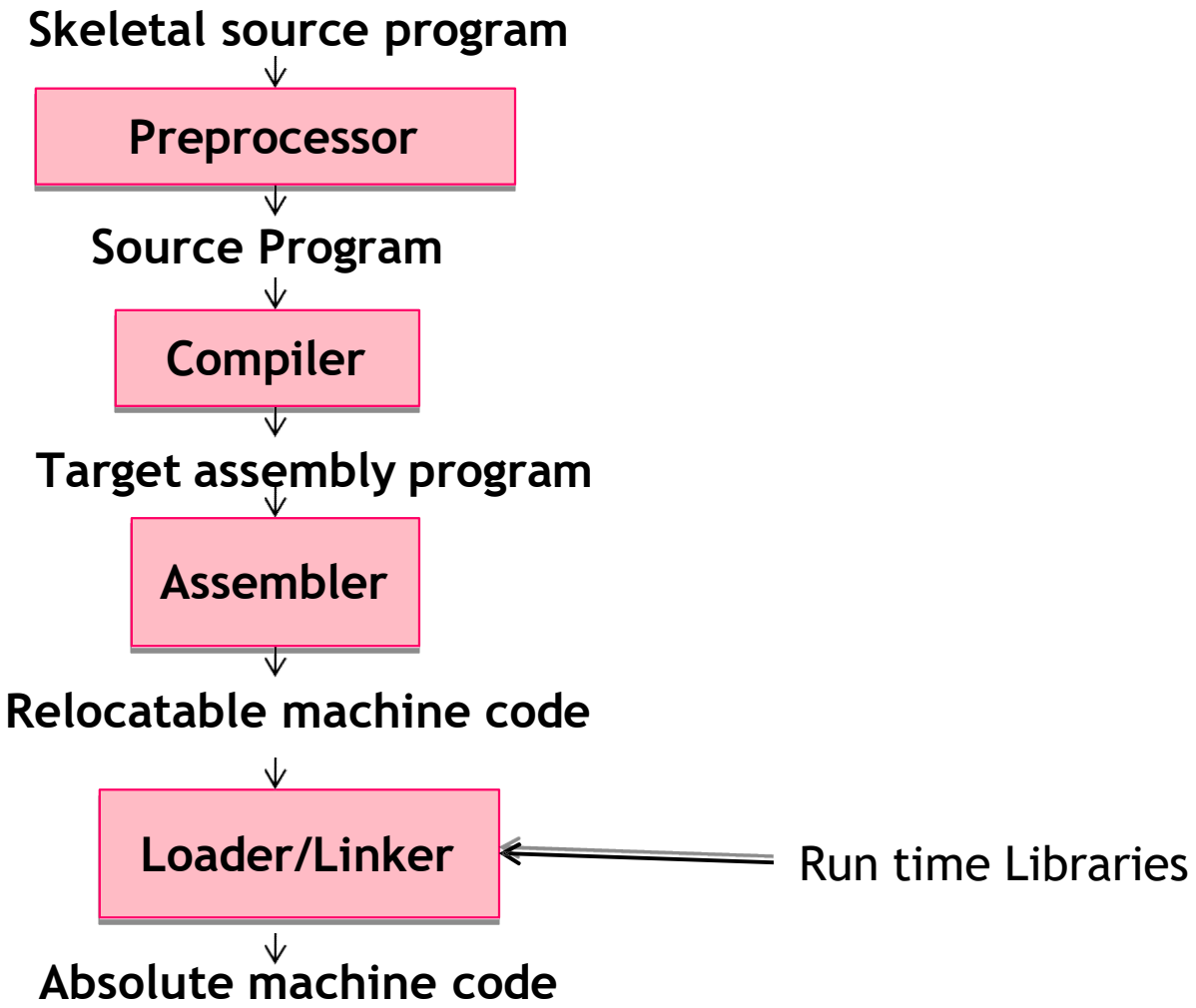
Compiler Vs. Interpreter – Contd…



Compiler	Interpreter
Scans & translates the entire program.	Translates statement by statement.
It takes large amount of time to analyse but the overall execution time is comparatively faster.	It takes less amount of time to analyse but the overall execution time is slower.
Intermediate object code is generated **for linking**, hence requires more memory.	Intermediate object code is **NOT** generated, hence **less memory**.
Debugging is comparatively hard. Generates error message only after scanning the full program.	Debugging is easy. when first error is met, program stops.
Ex: C, C++ use compilers.	Ex: Python, Ruby.

Language Processing system.

How actually the compilation process? (6 or 7 mark)



Language Processing system- contd…

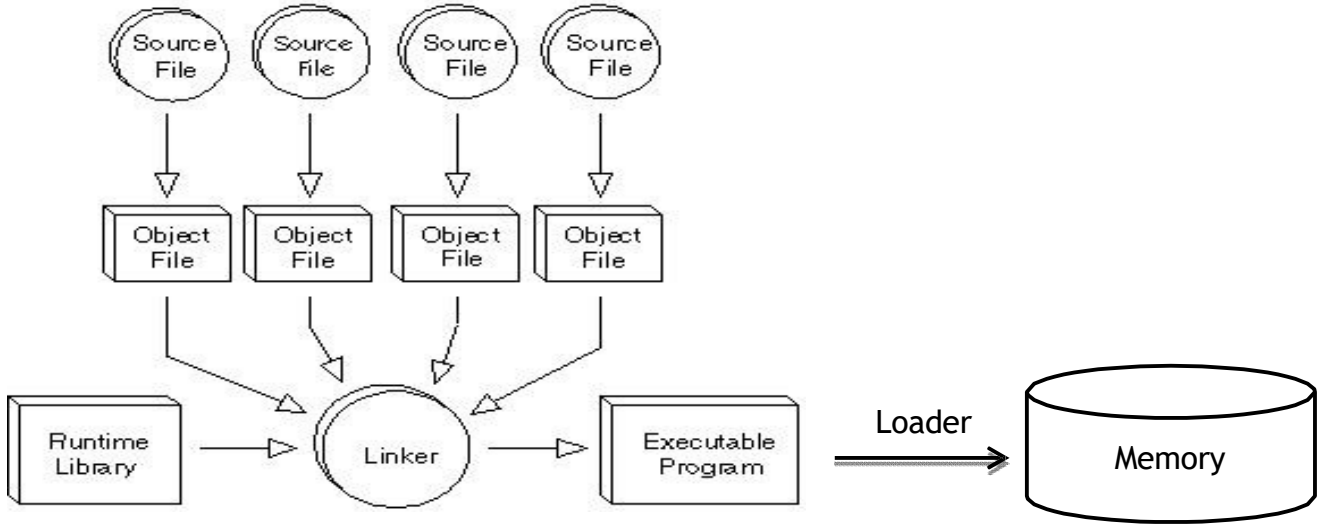
- A computer program is divided into modules. **Pre-processor** takes a module, and processes the skeletal source program before its passed to the compiler. It adds pre-processor directives and produces source code.



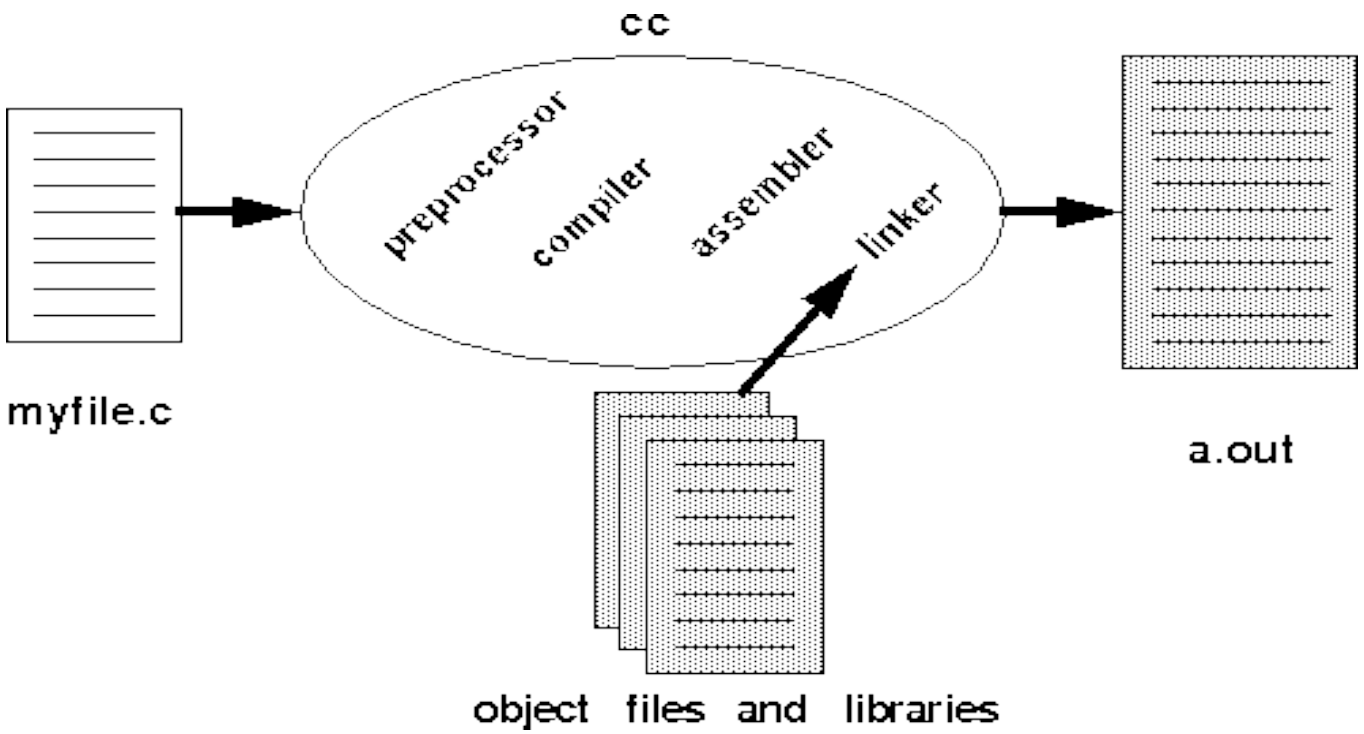
- **Compiler** is a program that takes **source program** as input and produces **assembly language** program as output.
- **Assembler** is a program that converts **assembly language** program into **machine language** program. It produces re-locatable machine code as its output.

Language Processing system- contd…

- The **re-locatable machine code** has to be linked together with other re-locatable object files and library files.
- **The loader** puts together the entire executable object files into memory for execution.

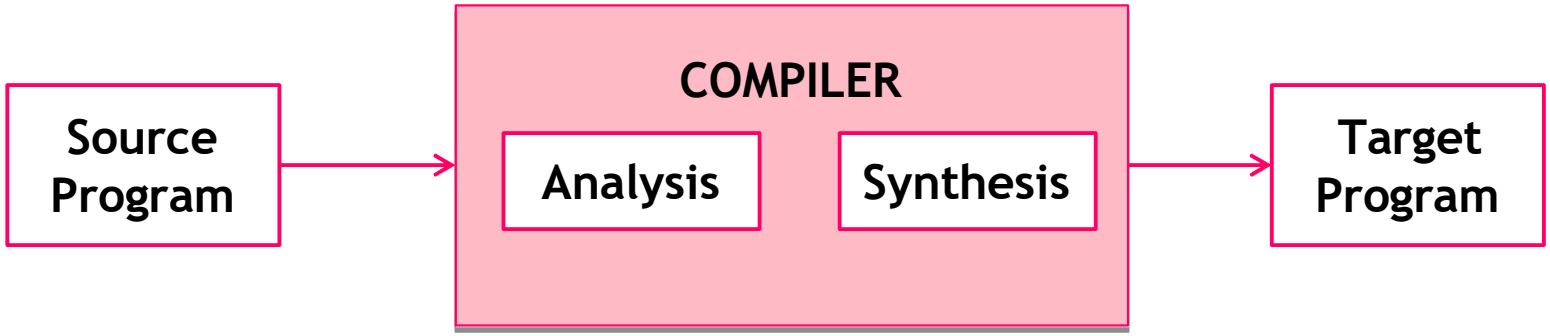


Language Processing system.- contd…



Analysis – synthesis model of Compilation

- 2 parts of compilation



Analysis Phase

(i) **Breaks** the source program into constituent pieces and creates intermediate representation

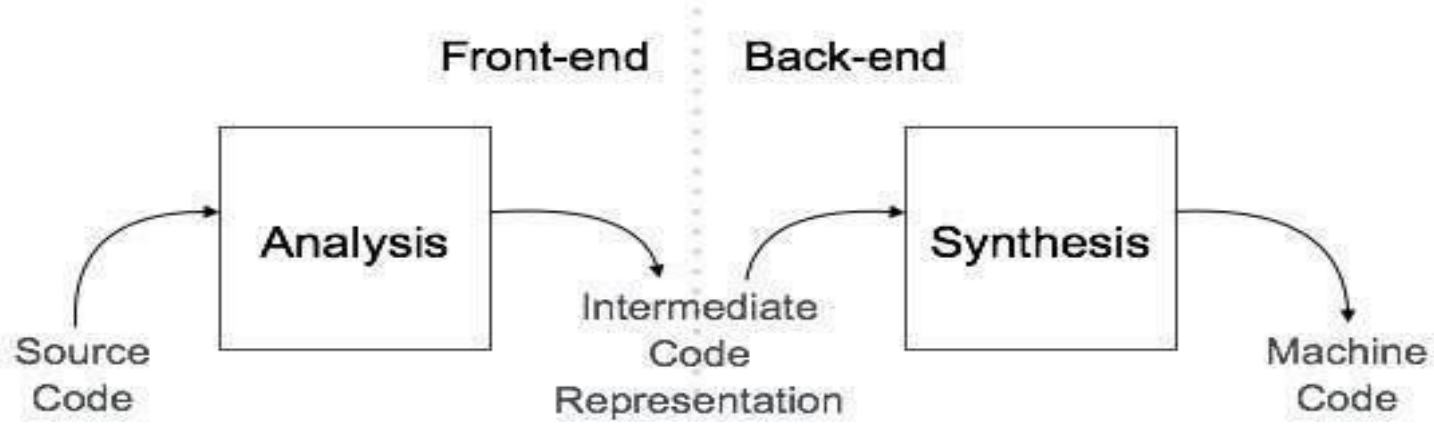
(ii) Front end of the compiler

Synthesis Phase

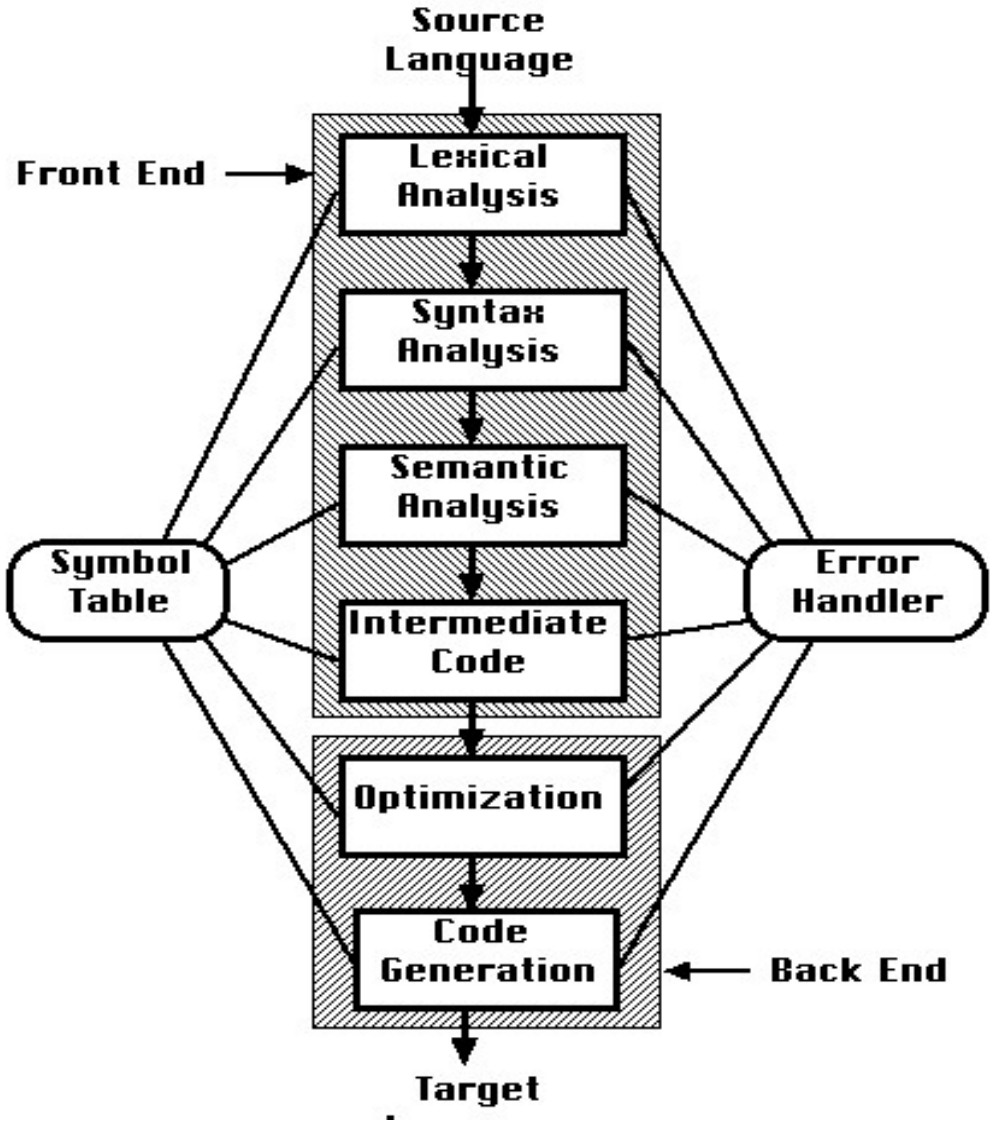
(i) Constructs the desired target program from the intermediate representation

(ii) Back end of the compiler

Analysis – synthesis model of a compiler



Phases of a compiler (13 mark)



Analysis of the source program

- Analysis consists of 3 phases
  - Linear analysis
  - Hierarchical analysis
  - Semantic analysis

Linear analysis

- Called as lexical analysis (or) **SCANNING**
- Read left to right, stream of characters and grouped into **TOKENS**

Token → Sequence of characters having a collective meaning, such as keywords, operators, identifiers etc.

Lexeme → Instance of the Token

Pattern → Describes the rule that the lexeme takes

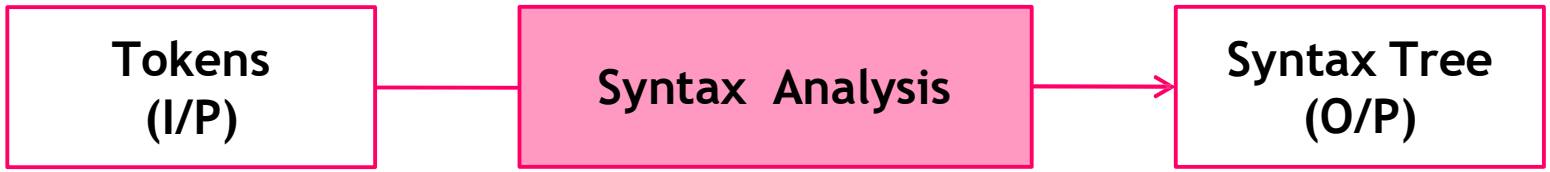
Contd…



- Token Template : <token-name, attribute-value>
- Hence, **<id,1><=><id,2><+><id,3><*><60>**

Syntax analysis

- Hierarchical analysis (or) PARSING
- Converts tokens produced by Lexical analyzer into tree like representation called PARSE TREE.



Parse tree for position := initial + rate * 60

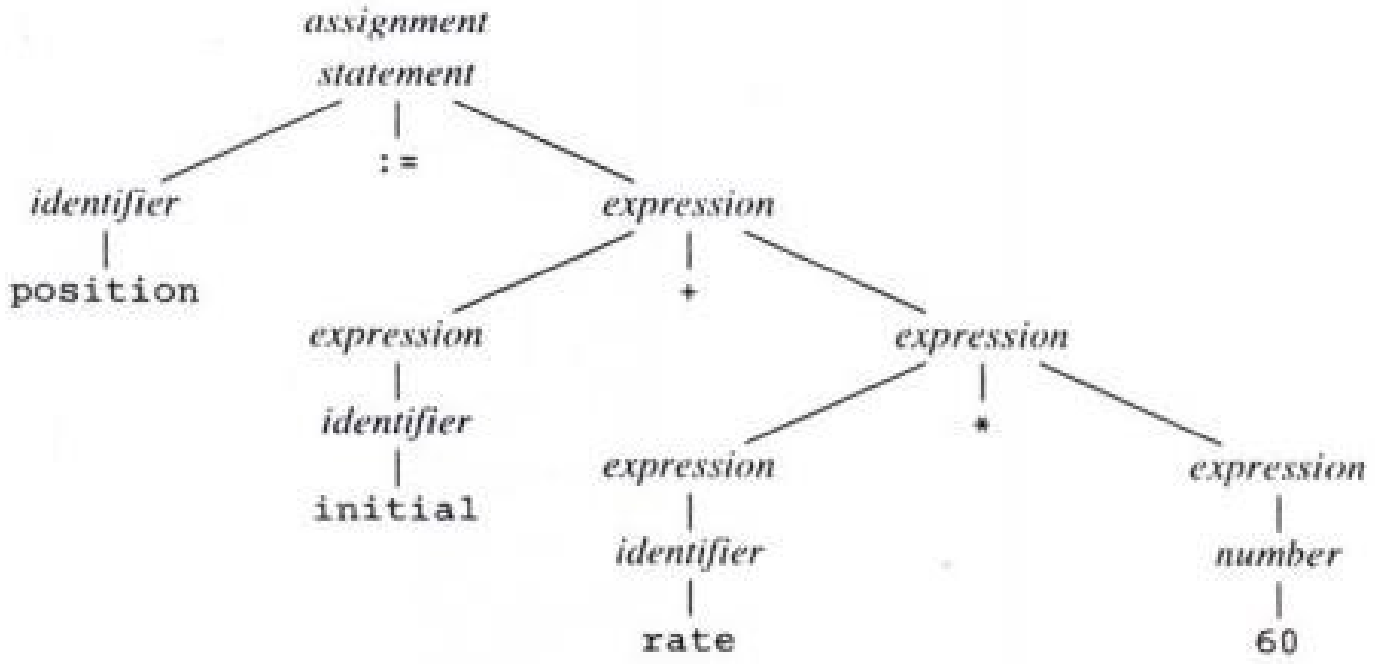


Figure 4. Parse Tree for Position := initial + rate * 60

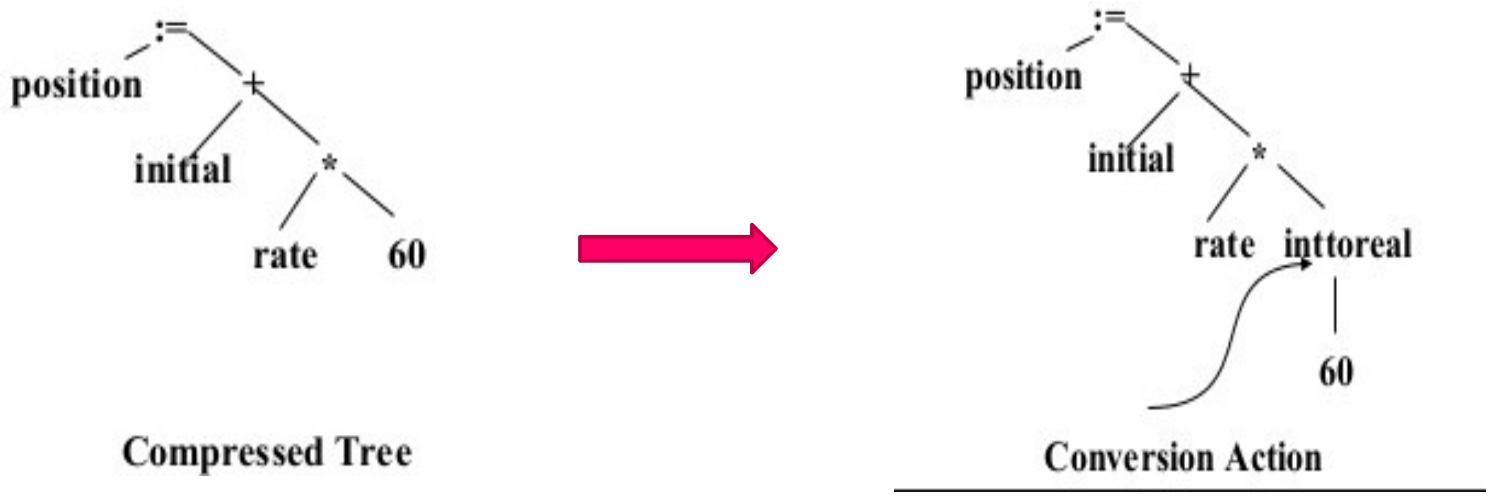
Contd…

- Characters grouped are recorded in a table – Symbol table

- Parse tree – describe the syntactic structure of the input.
- Syntax tree – compressed representation of parse tree.

Semantic analysis

- Semantic analysis checks for semantic errors and gathers information for code generation phase.
  - Type checking – checks each operator has 2 operands



Intermediate code generation

- Produces intermediate representation of the source program
  - Properties of Intermediate code generation
    - Easy to produce
    - Easy to translate into target program

- Various formats for representing Intermediate code generation are
    - Postfix notation
    - Three address code – most popular and most widely used
    - Quadruple & Triple

Contd - Three address code

- Sequence of instructions, each of which has at most 3 operands.
  1. temp:= inttoreal(60)
  2. temp2 := id3 * temp1
  3. temp3 := id2 + temp2
  4. id1 := temp3

- Properties of three address code
  - Has at most 1 operator in addition to assignment
  - Generate temporary name to hold the intermediate value
  - Some instructions, have operands < 3 (example 1 & 4)

Code optimization

- Improve the intermediate code so that object programs runs faster and takes lesser space.

- Optimization may be
  - Detection and removal of dead(unreachable) code)
  - Calculation of constant expressions and terms
  - Loop unrolling
  - Moving codes outside loops
  - Removal of unnecessary temporary variables

Code generation

- Final phase of the compiler
- Once optimizations are completed, the intermediate code is mapped into sequence of machine instructions.

- This involves
  - Allocation of registers and memory
  - Generation of correct references
  - Generation of correct types
  - Generation of machine code

Symbol table management

- **Symbol table** : Used to store identifiers(variables) used in the program
- Information about attributes of each identifier.
  - Scope
  - Information about storage allocated
- Example:
  - int a
  - char b
  - float c;

Symbol Name	Type	Address
a	int	1000
b	char	1002
c	float	1003

Why to use symbol table?

- To track the record of each identifier and retrieve quickly.

Error detecting and handling

- The errors identified, is reported to **error handler**

Handles error

- Errors → normally in the form of message
- During,
  - Lexical analysis → when they do not form token
  - Syntactical analysis → syntax errors
  - Semantic analysis → type mismatch

Statement translation

SYMBOL TABLE

1	position	···
2	initial	···
3	rate	···
4		

```
position := initial + rate * 60
```

Lexical Analyzer

$id_1 := id_2 + id_3 * 60$

Tokens

Syntax Analyzer



Syntax tree

Semantic analyzer

Semantic Tree



Intermediate Code Generator

Non-optimized Intermediate Code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Code Optimizer

Optimized Intermediat Code

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Code Generator

Target machine code

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

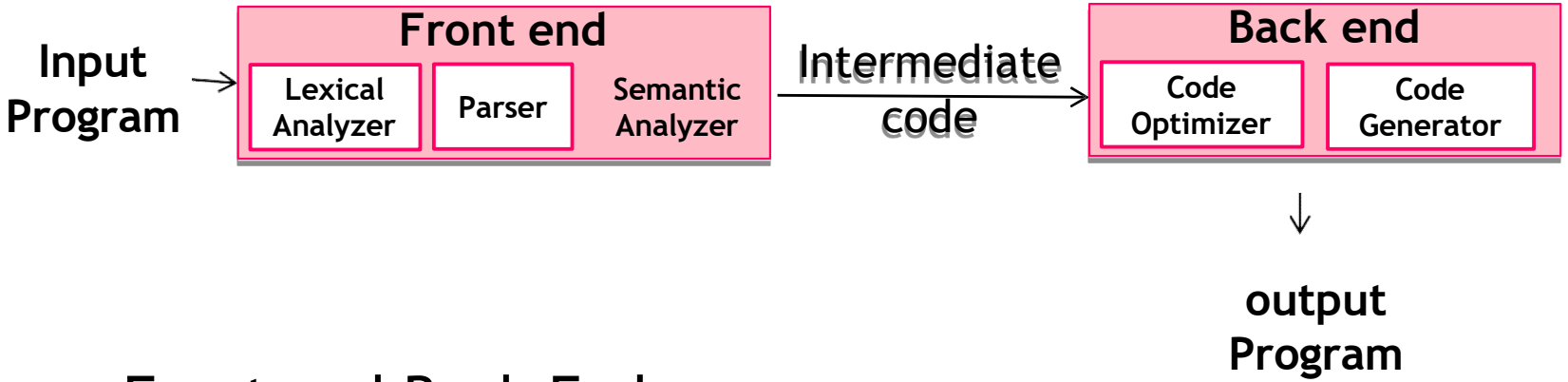
F → Floating point number
1st operand → **source**
2nd operand → **destination**

Assignment

- Translate the statement for
  - p = i + r * 60
  - a = b + c + 50
  - si = p * n * r /100

Grouping of phases

- Activities from more than one phase can be grouped together.



- Front and Back Ends
- Passes
- Reducing the number of passes

Grouping of phases – contd…

Front and Back Ends:

- **The front end phases depends more on source language and is independent of the target machine.**
  - These normally include lexical and syntactic analysis, semantic analysis and intermediate code generation.
  - In addition it includes symbol table management & error handling.

- **The back end phases depends more on target machine and is independent of the source language**
  - This includes code optimization and code generation.
  - In addition it includes symbol table management & error handling.

Why grouping of phases is advantageous?

- Keeping same front end and attaching different back ends (ie. Same source program for different machines)
- Keeping different front ends and same back end.

Grouping of phases – contd…

Passes:

- One complete scan of source program - PASS
  - It includes reading an input file and writing an output file.

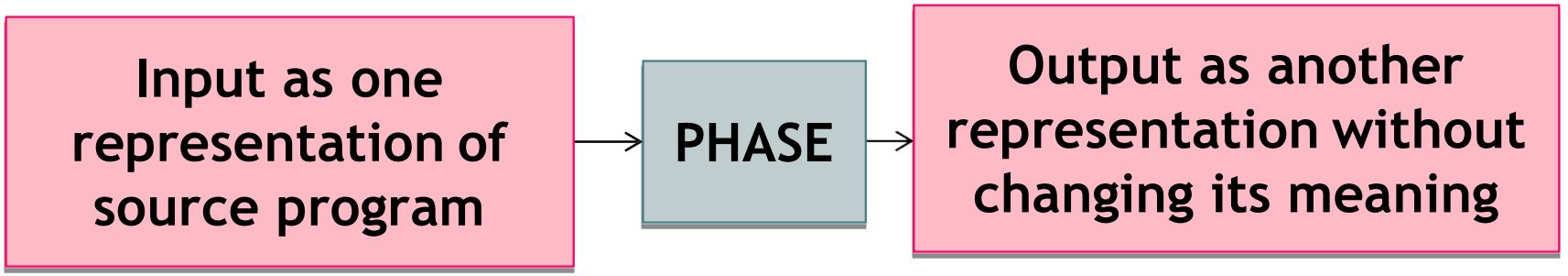
Grouping of phases – contd…

Reducing the number of passes:

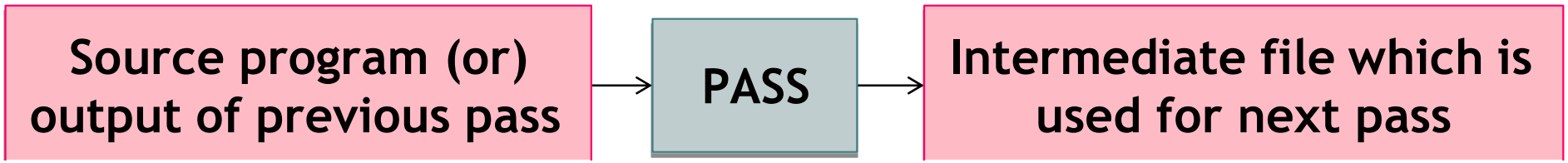
- Takes time to read and write intermediate files.
- Grouping of several phases into one pass, may force the entire program in memory, because one phase may need information in a different order than previous phase produces it.
- Compiler requires
  - 1st pass – scanning and parsing
  - 2nd pass – semantic analysis
  - 3rd pass – code generation and target code optimization
- C, PASCAL → 1 pass
- MODULA -2 → 2 passes
- Intermediate code and code generation are often merged into one pass using a technique called backpatching.

Differences between phases and pass of the compiler

- Phase



- Pass
  - Portions of 1 or more phases combined into a module called pass



Phases and Passes

Pass

1) Pass is a physical scan over a source program. The portions of one or more phases are combined into a module called pass.

2) Requires an intermediate file between two passes

3) Splitting into more no. of passes reduces memory.

4) Single pass compiler is faster than two pass .

Phase

1) A phase is a logically cohesive operation that takes i/p in one form and produces o/p in another form.

2) No need of any intermediate files in between phases.

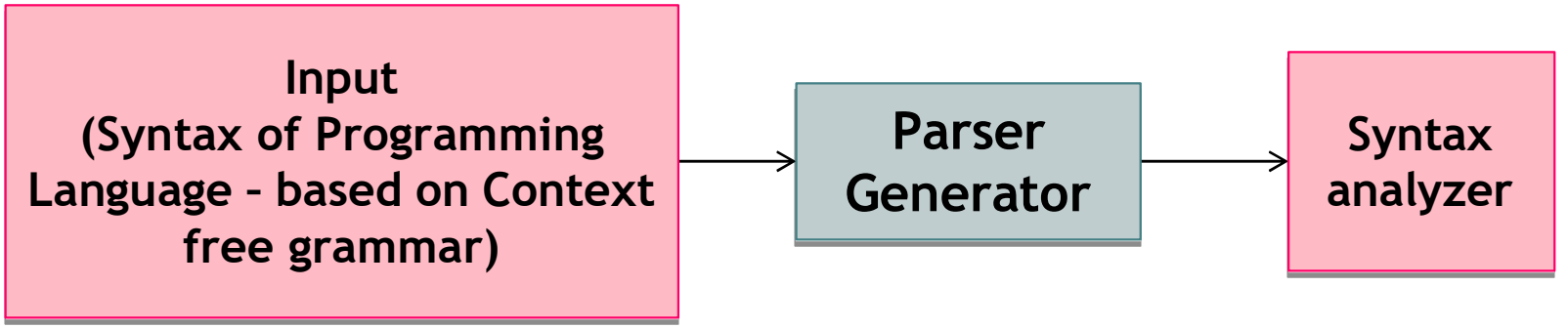
3) Splitting into more no. of phases reduces the complexity of the program.

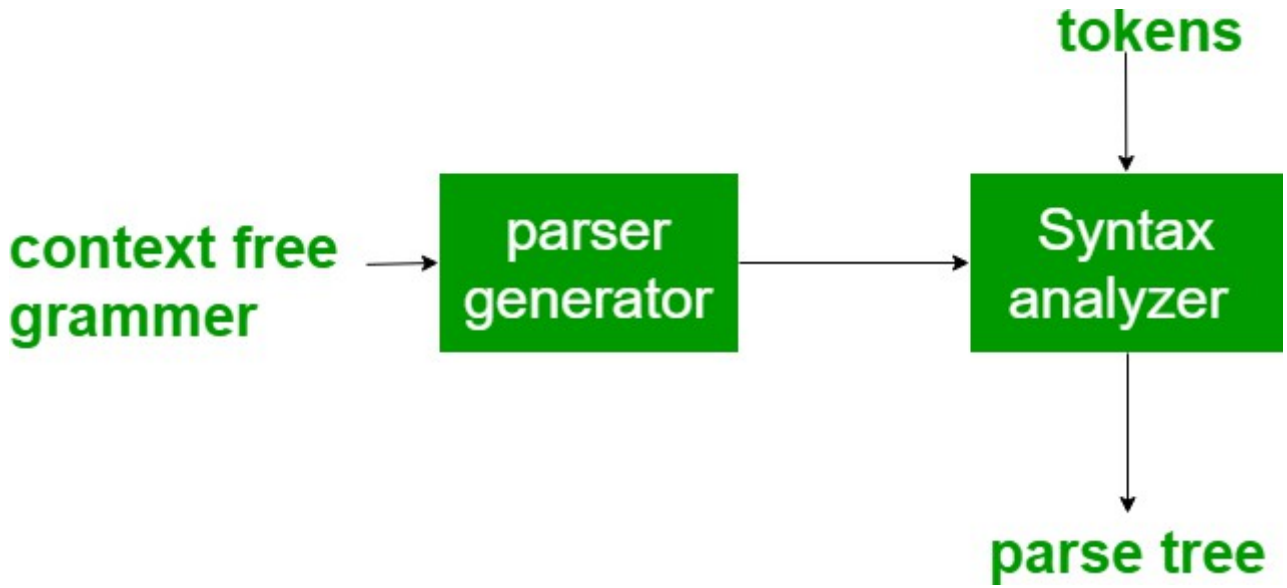
4) Reduction in no. of phases, increases the execution speed

Compiler construction tools

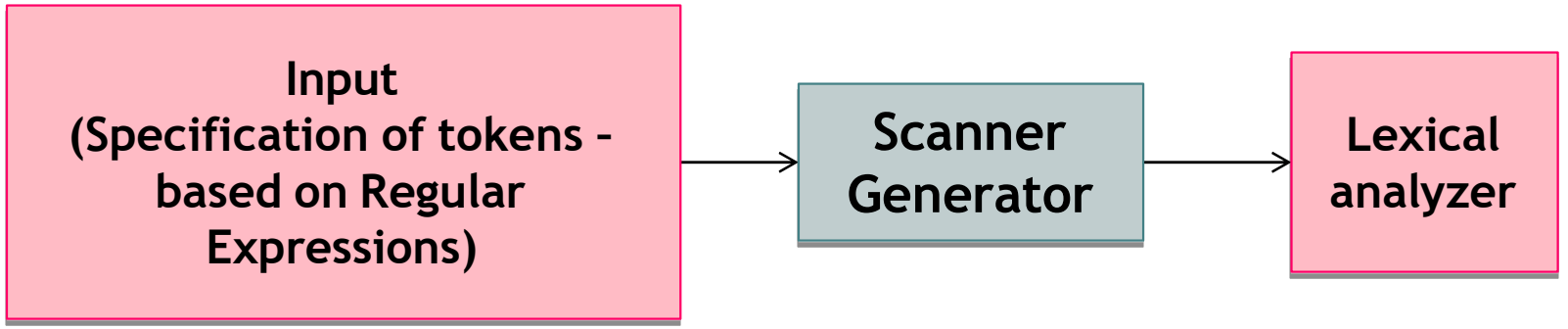
- Tools that help in the design of compilers are
  - Parser generator
  - Scanner generator
  - Syntax directed translation engines
  - Automatic code generators
  - Data flow engines

Parser Generator(Example:PIC, EQM)

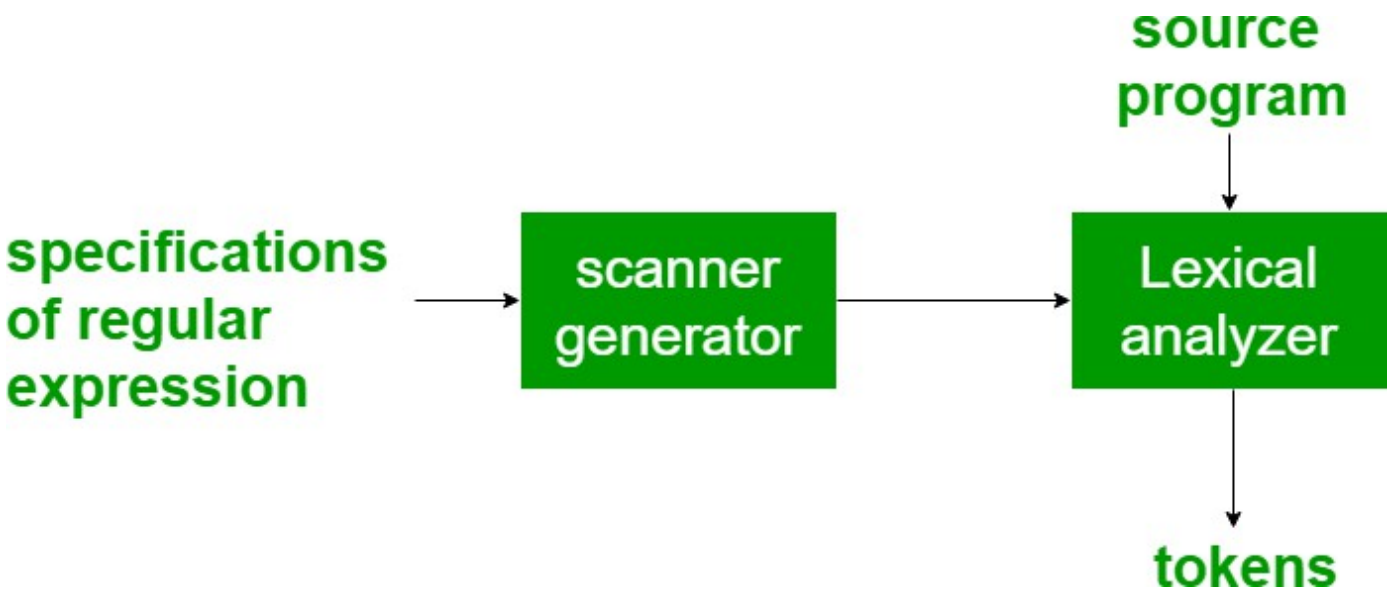




Scanner Generator



- UNIX has LEX. LEX has various regular expressions for representing tokens

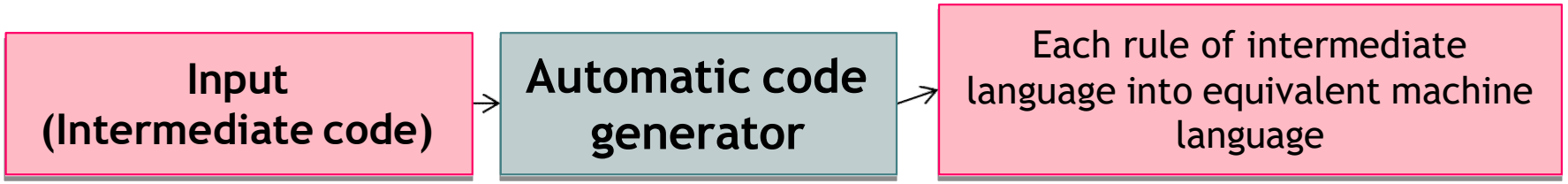

source
program
specifications
of regular
expression
scanner
generator
Lexical
analyzer
tokens

Syntax directed translation engine



- Translation is done for each node.

Automatic Code Generator



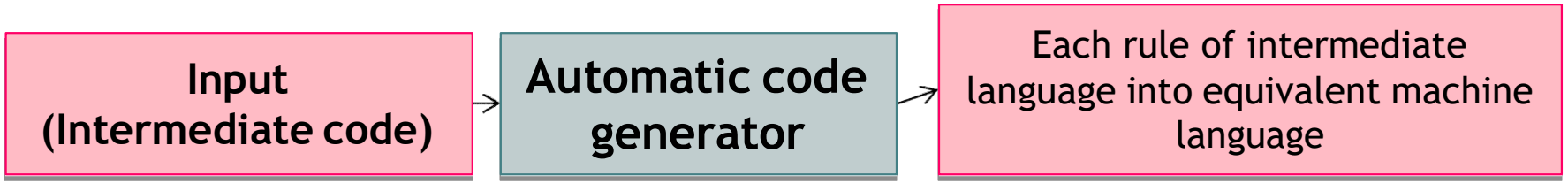
- Template matching technique is done. Intermediate code statements are replaced by “templates”

Syntax directed translation engine



- Translation is done for each node.

Automatic Code Generator



- Template matching technique is done. Intermediate code statements are replaced by “templates”

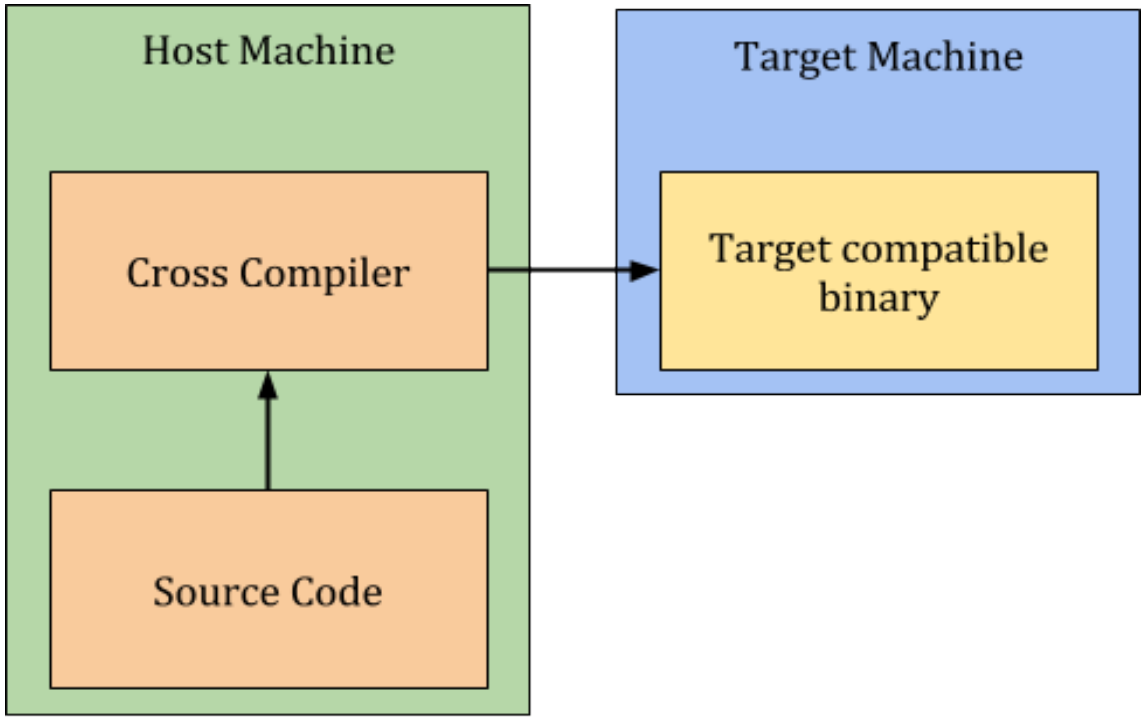
Data flow engines

- It is used to perform good code optimization
- It’s just gathering information about how values are transmitted from one part of program to another.

Types of compiler

- **Incremental compiler**
  - Performs recompilation of only modified source rather than compiling the whole program.
- **Cross compiler**
  - Compiler that runs on one machine and produces target code for another machine

Cross compiler



Cross Compiler operation

Incremental Compiler

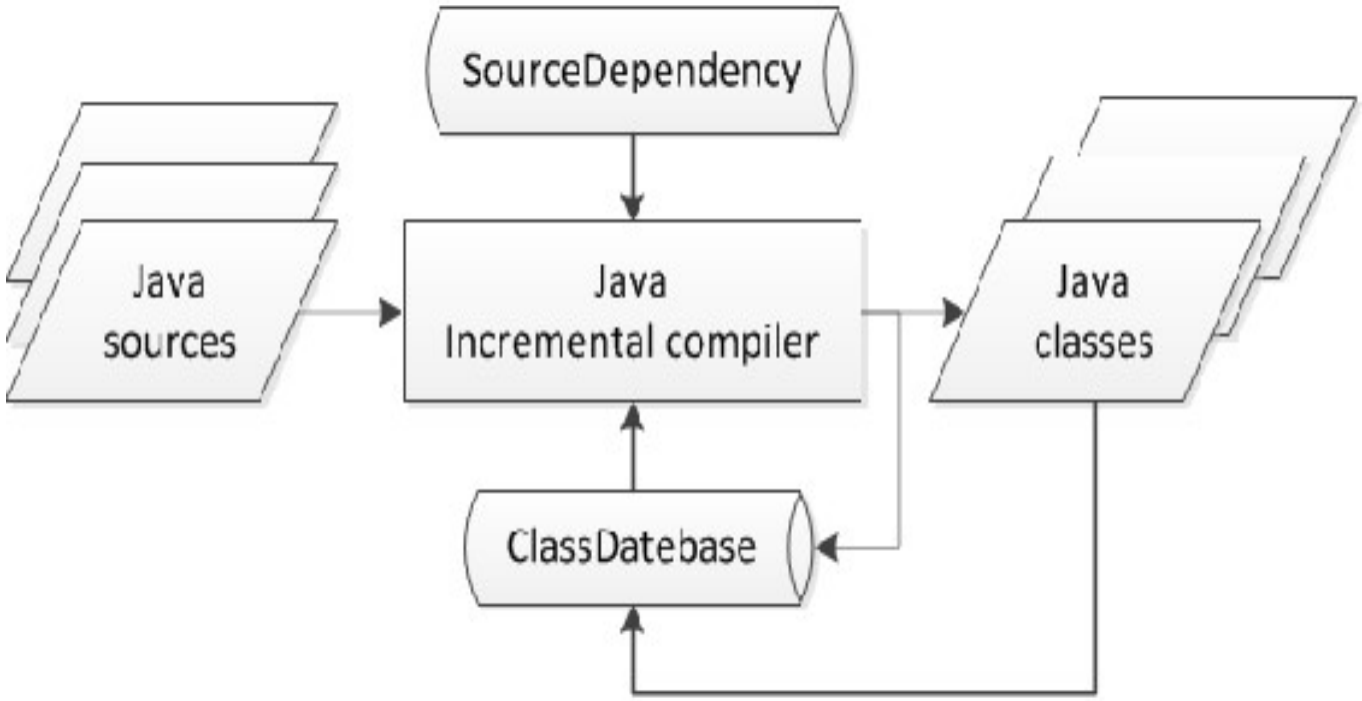


Figure 1. Java Incremental Compiler

Tools that perform Analysis

- Structure Editors
- Pretty Printers
- Static Checkers
- Interpreters
- Text Formatters
- Silicon Compilers
- Query Interpreters

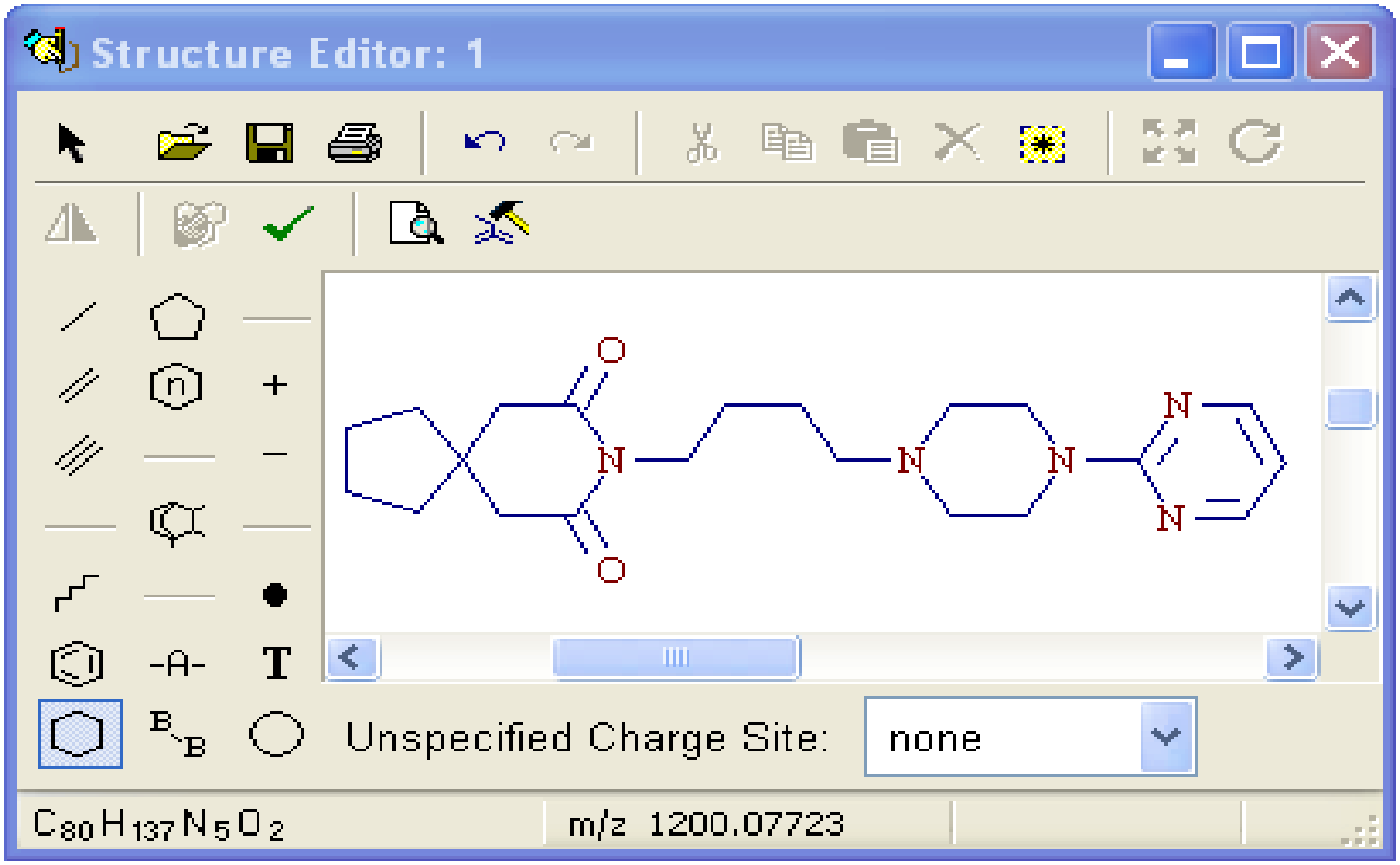
Structure Editors

- ***Ordinary text editor*** - text creation and modification functions
- ***Structure Editor*** - Analyze the program text by putting an appropriate hierarchical structure on the source program



- Example:
  - While – check for matching do
  - Begin – end
  - Left parenthesis – right parenthesis

Structure Editors

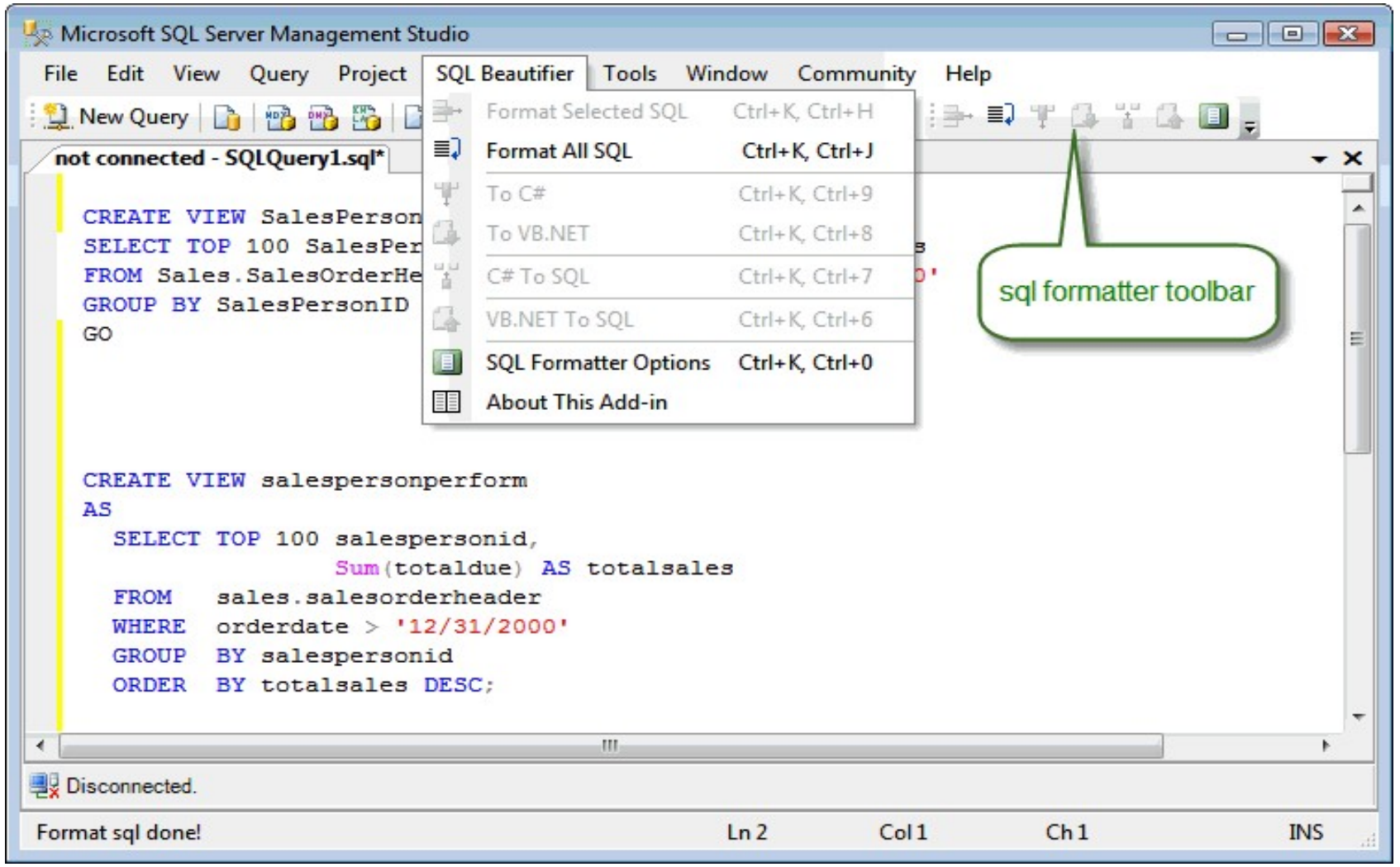


Pretty printer

- A Pretty printer analyses a program and prints it in such a way that the structure of the program becomes clearly visible.

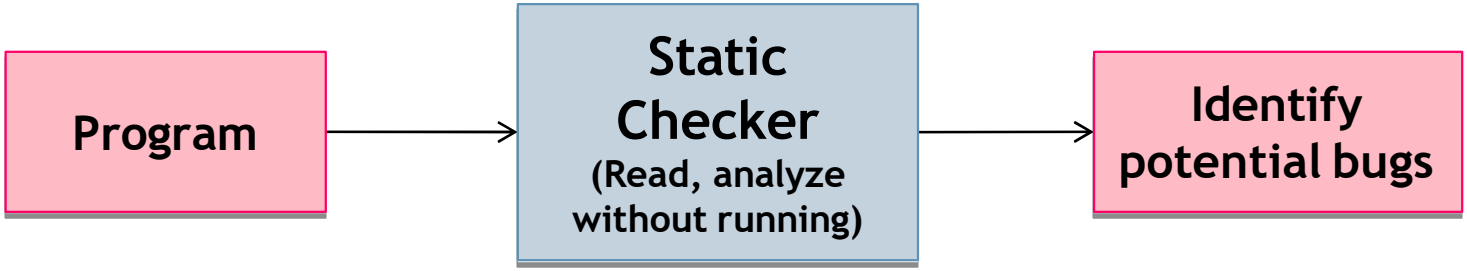
- Example
  - Comments in special font
  - Indentation proportional to the depth – nesting of the program

Pretty printer



Static checker

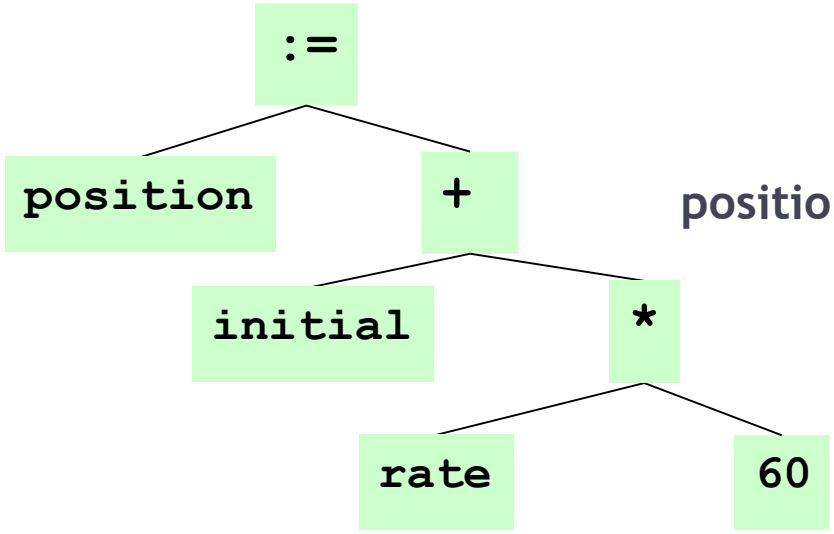
- A static checker reads a program, analyses it and attempts to discover potential bugs with out running the program.



- Example
  - Detect parts that never be executed
  - Variable defined but not used
  - Variable used but not defined

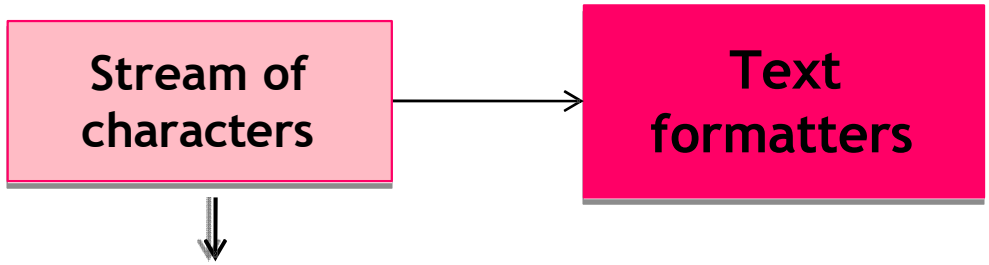
Interpreters

- Instead of producing a target program as a translation, an interpreter performs the operations implied by the source program.
  - For example, for an assignment statement an interpreter might build a tree and then carry out the operations at the nodes as it “walks” the tree.



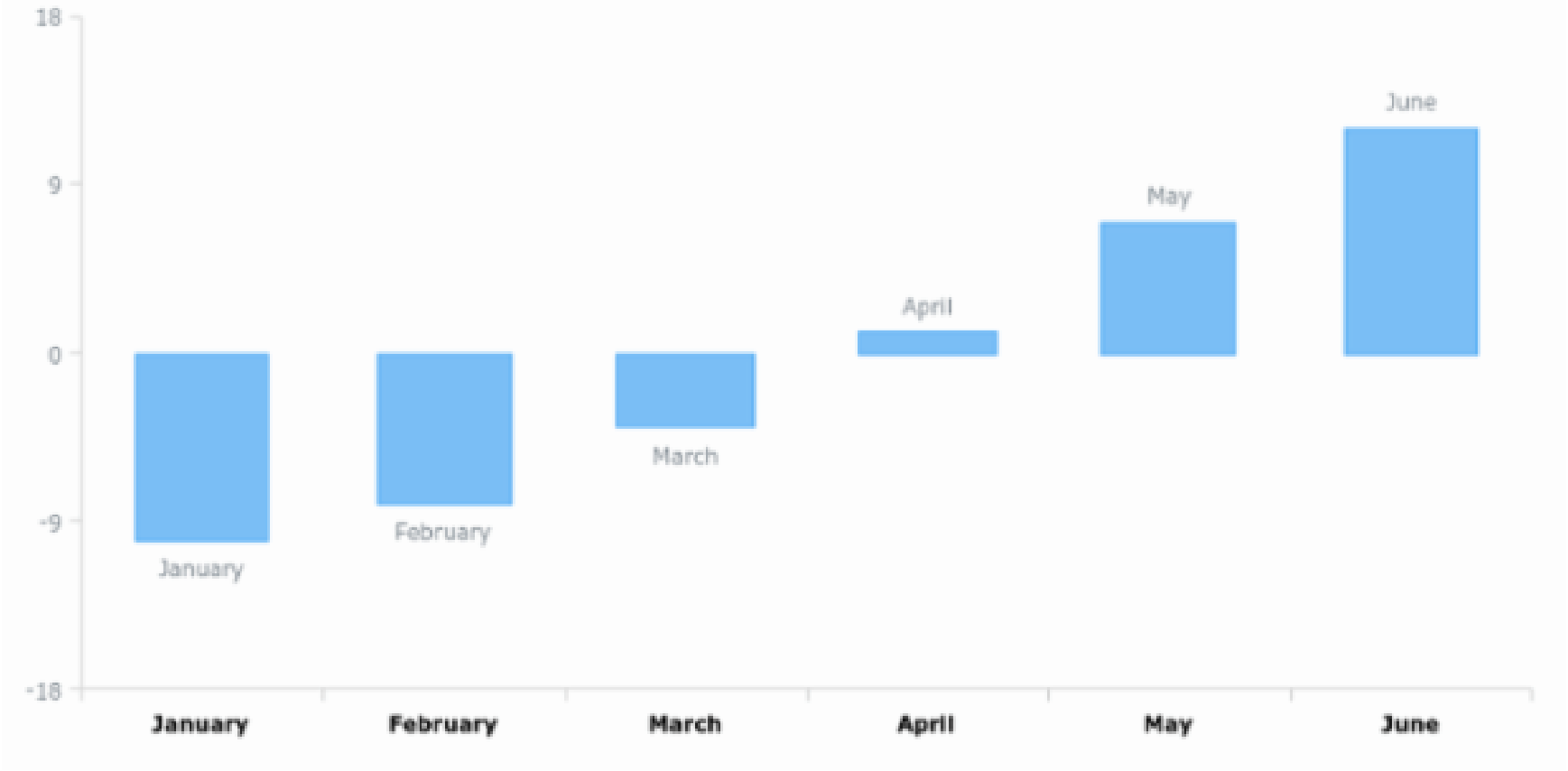
position := initial + rate * 60

Text Formatters



- Text
- Commands to include paragraphs
- Figures
- Subscripts and superscripts

Text Formatters



Silicon compilers



Query interpreters



Cousins of compiler



- To help the compilation process, various translators used are,
  - Preprocessors
  - Assemblers
  - Loaders
  - Link editors

} **Cousins of compiler**

Preprocessor

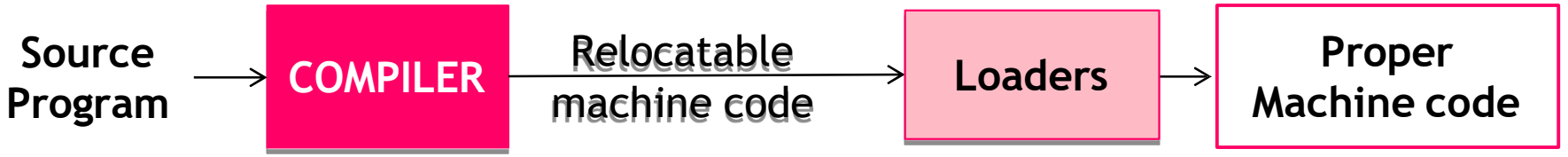


Functions performed by preprocessor are

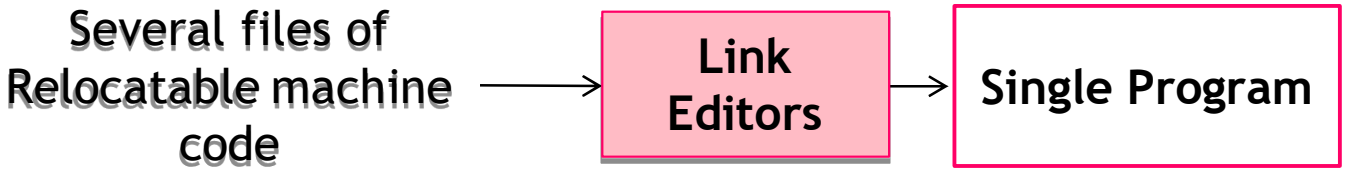
- Macro processing
  - Allow user to define macros
- File inclusion
  - Header files can be included example : #include <global.h>
- Rational preprocessor
  - Augment older languages with modern flow and data structuring facilities
- Language extensions
  - Add more capabilities to the language
  - Example: Database query language EQUAL is embedded in C

Loaders

- Taking relocatable machine code, altering relocatable addresses with proper locations.



Link editors



- If files are used together, there may be external references (one file refers to another file data)
- The relocatable machine code retains all the information in the symbol table.

Programming Language Basics

- The terminologies related with the programming languages are discussed here. The concepts related with the language like C, C++, C#, Java are discussed here.
  - Static/Dynamic Decision Policy
  - Variable scope
  - Mapping of Names to Values
  - Scope of the Data Members
  - Block Structure
  - Access Specifiers
  - Dynamic Scope

Static/Dynamic Decision Policy

- During designing a compiler, a policy should be followed to decide on issues ie. following static policy or dynamic policy.
  - If languages uses Static policy → then issues are decided at compile time
  - If languages uses Dynamic policy → then issues are decided at run time

Variable scope

- The scope of the variable declaration ”d” is the region of the program in which the use of “d” is referred.
  - **Static scope** → scope is determined at compile time(C & Java)
  - **Dynamic scope** → scope is determined at run time

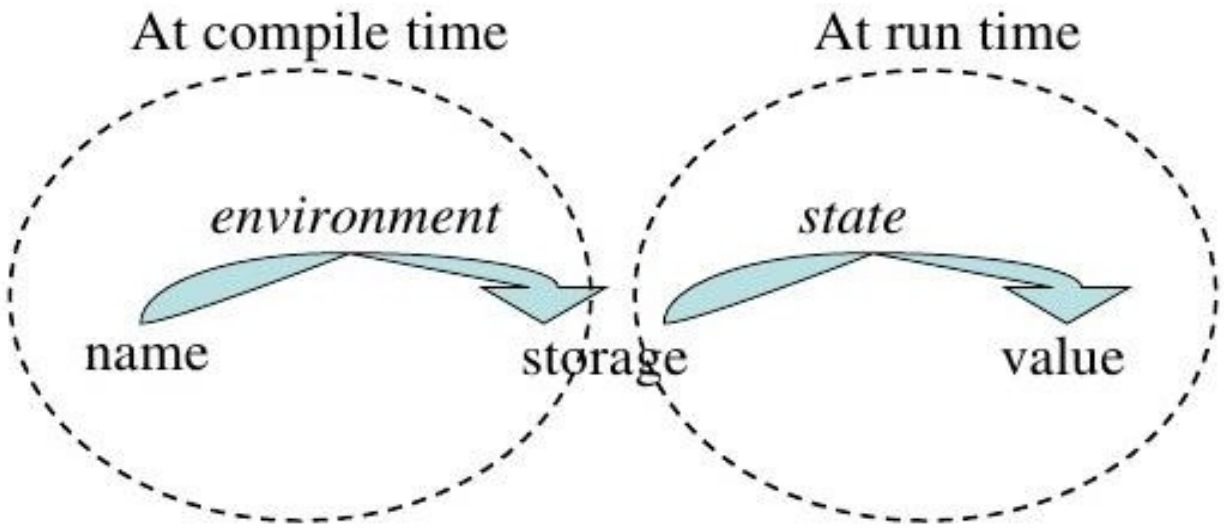
Mapping of Names to Values

- How a change in the program affect the values of data elements in the program.
  - **Environment:** mapping from (variable) names to store addresses
  - **State:** mapping from addresses to their values
    - Both mappings can be static or dynamic




```
var i;
…
i := 0;
…
i := i + 1;
```

Mapping of Names to Values – Contd…




```
var i;
…
i := 0;
…
i := i + 1;
```

Block Structure

- Block is the sequence of variable declarations and member function definitions.
  - In C, C++ language → block is represented by "{" and "}"
  - In Algol → block is represented by the keywords 'begin' and 'end'.

```
#include<iostream.h>
#include<conio.h>
void main()
{
        private int a,b;
        void display()
        {
        cout<<"No Pain; No Gain";
        }
}
```

Block B1

Block B2

Access Specifiers

- The access specifiers are used to include the scope of the variables, in and out of the class.
- In C++ and Java, access specifiers are public, private & protected.

```
Class Apple
{
        private:
                int a,b;
        public:
                int c;
        protected:
                int d;
}
```

Dynamic Scope

- If the scope of the variables are decided at run time, then its called as Dynamic Scope.
- The Dynamic scope of the variable is decided by parameter passing mechanisms

- Parameters are of two types
  - Actual Parameters
  - Formal Parameters

Parameter Passing

- When the execution control is transferred from calling function to called function it may carry one or more number of data values. These data values are called as parameters.
  - ***Parameters*** are the data values that are passed from calling function to called function.
-

Types of parameters

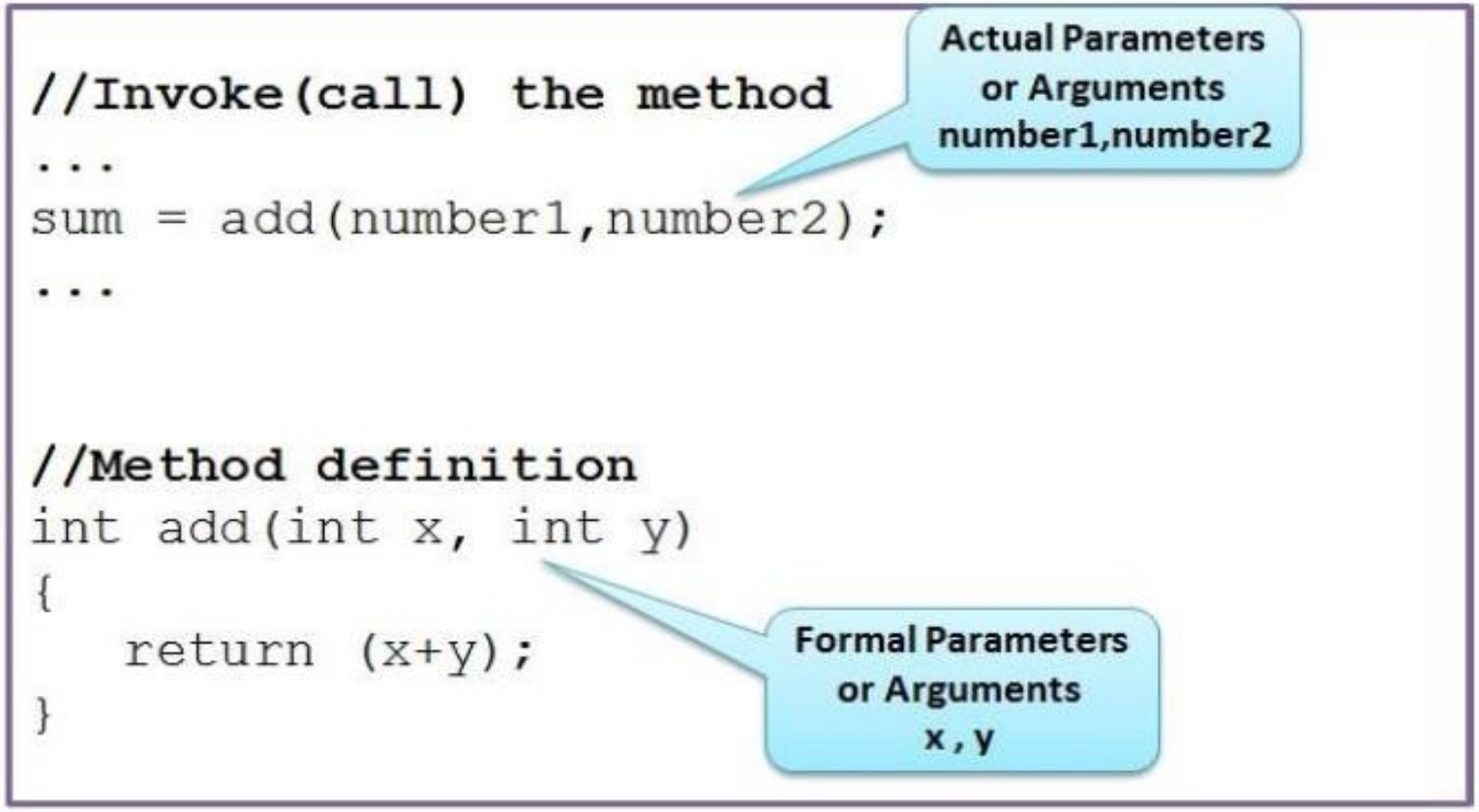
- In C, there are two types of parameters and they are as follows:
- **Actual Parameters**
  - The actual parameters are the parameters that are specified in calling function.
- **Formal Parameters**
  - The formal parameters are the parameters that are declared at called function.
- When a function gets executed, the copy of actual parameters is copied to the formal parameters.

Two types of parameters

- Actual Parameters
  - The actual parameters are the parameters that are specified in calling function.
- Formal Parameters
  - The formal parameters are the parameters that are declared at called function.

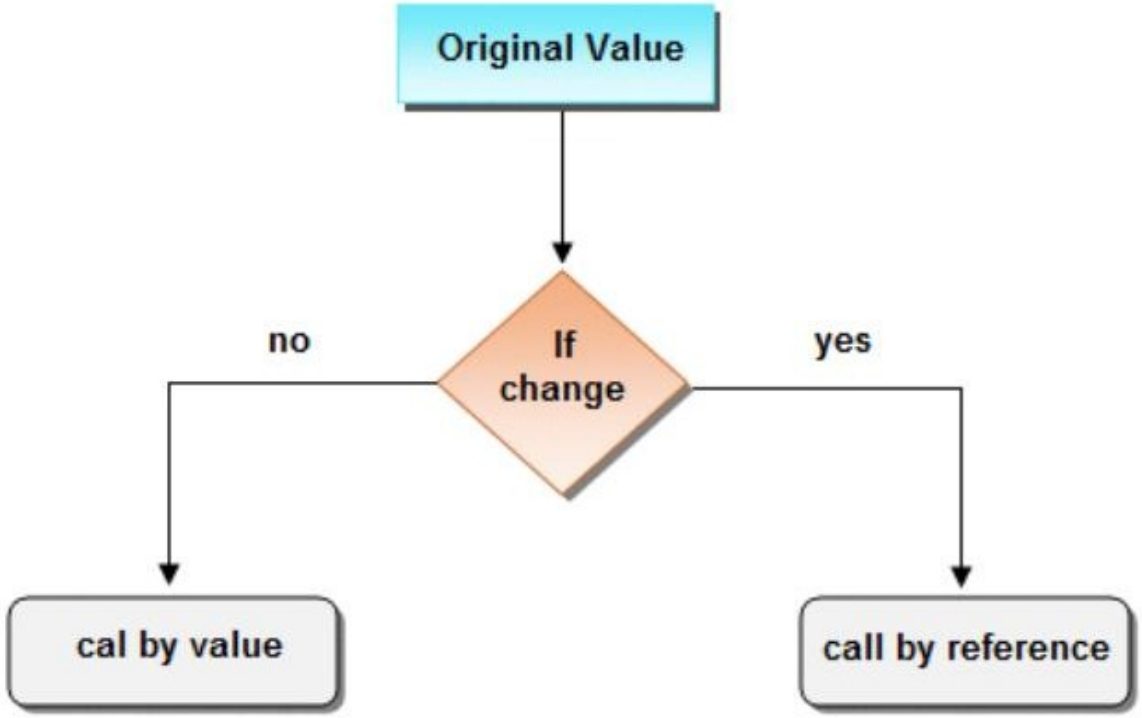
- When a function gets executed, the copy of actual parameters is copied to the formal parameters.

Contd...


//Invoke(call) the method
...
sum = add(number1,number2);
...
//Method definition
int add(int x, int y)
{
return (x+y);
}
Actual Parameters or Arguments number1,number2
Formal Parameters or Arguments x , y

Parameter passing mechanisms

- Call by Value (or) Pass by value
- Call by Reference (or) Pass by Reference



Call by Value

- In call by value parameter passing method, the **copy of actual parameter values are copied to formal parameters**.

- The changes made on the formal parameters have **NO EFFECT** on the actual parameters.
  - That means, after the execution control comes back to the calling function, the actual

parameter values remains same.

```c
void main()
{
  int num1, num2 ;
  void swap(int,int) ; // function declaration
   num1 = 10 ;
   num2 = 20 ;

   printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
   swap(num1, num2) ; // calling function
   printf("\nAfter swap: num1 = %d\nnum2 = %d", num1, num2);
}

void swap(int a, int b)  // called function
{
   int temp ;
   temp = a ;
   a = b ;
   b = temp ;
}
```

Execution:

Output:

```
Before swap: num1 = 10, num2 = 20
After swap: num1 = 10, num2 = 20
```

Call by Reference

- In Call by Reference parameter passing method, **the address of the memory location is copied to formal parameters**. The address of the actual parameters is passed to the called function and is received by the formal parameters (pointers).
  - Hence changes made on the formal parameters **have effect** on the actual parameters also.

```c
#include <stdio.h>
void main()
{
   int num1, num2 ;
   void swap(int*,int*) ; // function declaration
   num1 = 10 ;
   num2 = 20 ;

   printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
   swap(&num1, &num2) ; // calling function
   printf("\nAfter swap: num1 = %d\nnum2 = %d", num1, num2);
}

void swap(int*a, int*b)  // called function
{
   int temp ;
   temp = *a ;
   *a = *b ;
   *b = temp ;
}
```

Execution:

Output:

```
Before swap: num1 = 10, num2 = 20
After swap: num1 = 20, num2 = 10
```

UNIT – I Completed