CS304 COMPILER DESIGN

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

AMAL JYOTHI COLLEGE OF ENGINEERING
KANJIRAPPALLY

COURSE TITLE	**COMPILER DESIGN**
COURSE CODE	**CS304**
SEMESTER	**VI**
PROGRAM	**B.TECH**
COURSE DURATION	**MARCH ‘21 – JUNE ‘21**
COURSE LOAD	**3-0-0-3 (L-T-P-C)**
PREREQUISITE	**Nil**

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

AMAL JYOTHI COLLEGE OF ENGINEERING
KANJIRAPPALLY

Outline

- Syllabus & Text books
- History of compilers
- Interpreters
- What are compilers?
- Types of compilers
- Properties of the compiler
- Parts of compilation
- Analysis of the source program

Syllabus

Module 1(13 hrs)

Introduction to compilers – Analysis of the source program, Phases of a compiler, grouping of phases, compiler writing tools – bootstrapping.

Lexical Analysis: The role of Lexical Analyzer, Input Buffering, Specification of Tokens using Regular Expressions, Review of Finite Automata, Recognition of Tokens.

Text Books

1. Alfred V.Aho, Ravi Sethi and Jeffrey D.Ullman, **"Compilers – Principles, Techniques and Tools"**, second edition, Pearson Education, New Delhi, 2006.

2. D. M.Dhamdhare, **System Programming and Operating Systems**, Tata McGraw Hill & Company, 1996

References

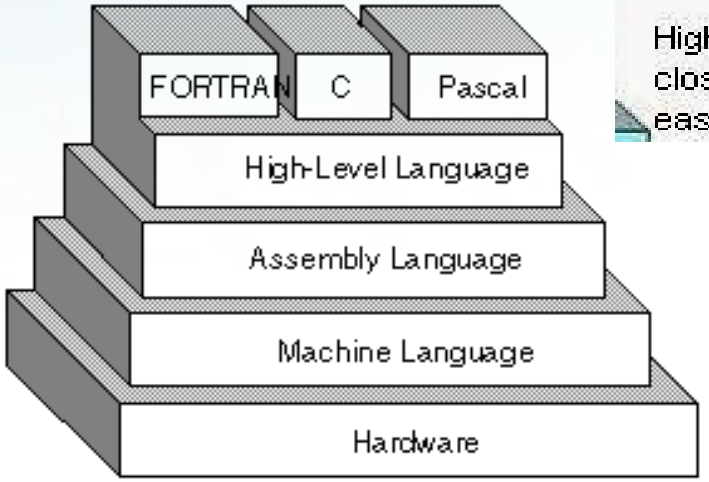
1. Kenneth C. Louden, **Compiler Construction – Principles and Practice**, Cengage Learning Indian Edition, 2006.

2. Tremblay and Sorenson, **The Theory and Practice of Compiler Writing**, Tata McGraw Hill & Company,1984.

High level Language

- Enables a programmer to write programs that are more or less independent of a particular type of computer.
- Such languages are considered high-level because they are closer to human languages and further from machine languages.

Assembly languages use symboles for binary codes



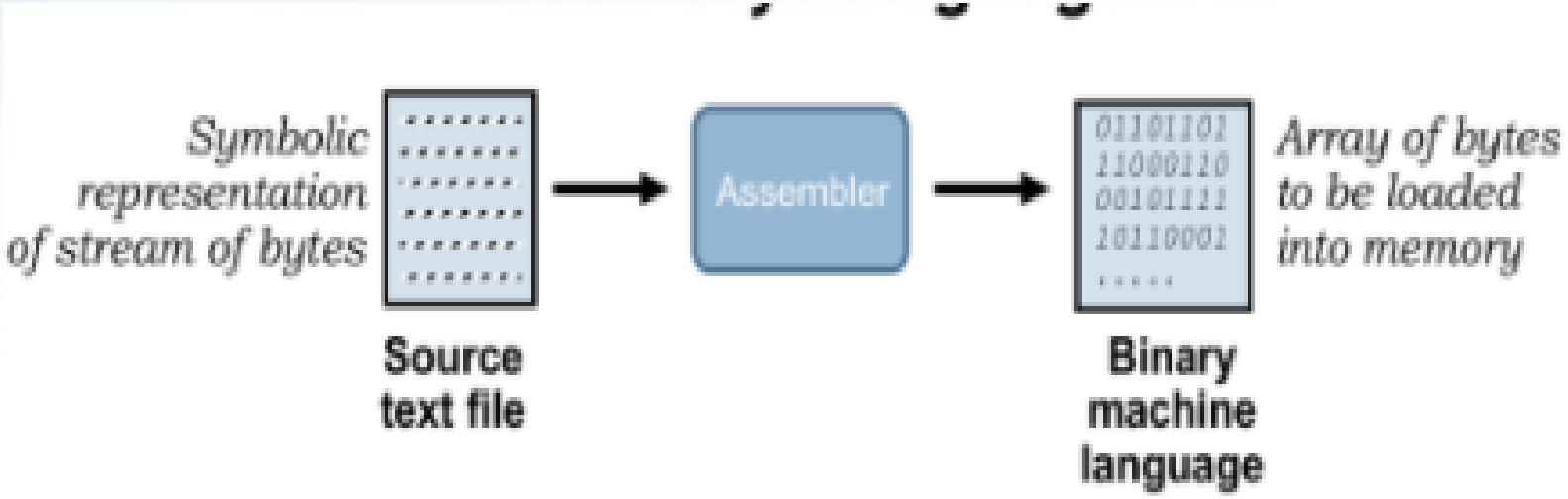
High Level Languages are close to human languages, easy to learn

Native language of computer uses binary digits 0 and 1

Assembly language

Assembly language is a low-level programming language for a computer that uses mnemonic codes to represent machine – Language Instructions.

Assembly language is converted into executable machine code by a utility program referred to as an **assembler** like NASM, MASM, etc.



Levels of Programming Languages

High-level program

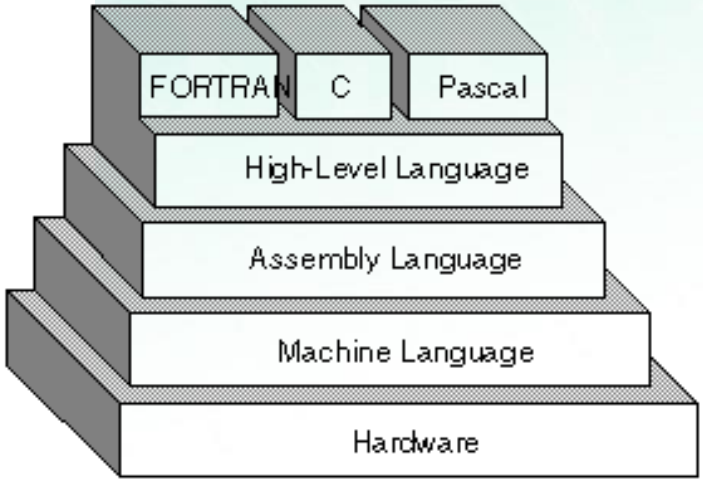
```
class Triangle {
  ...
  float surface()
    return b*h/2;
  }
```

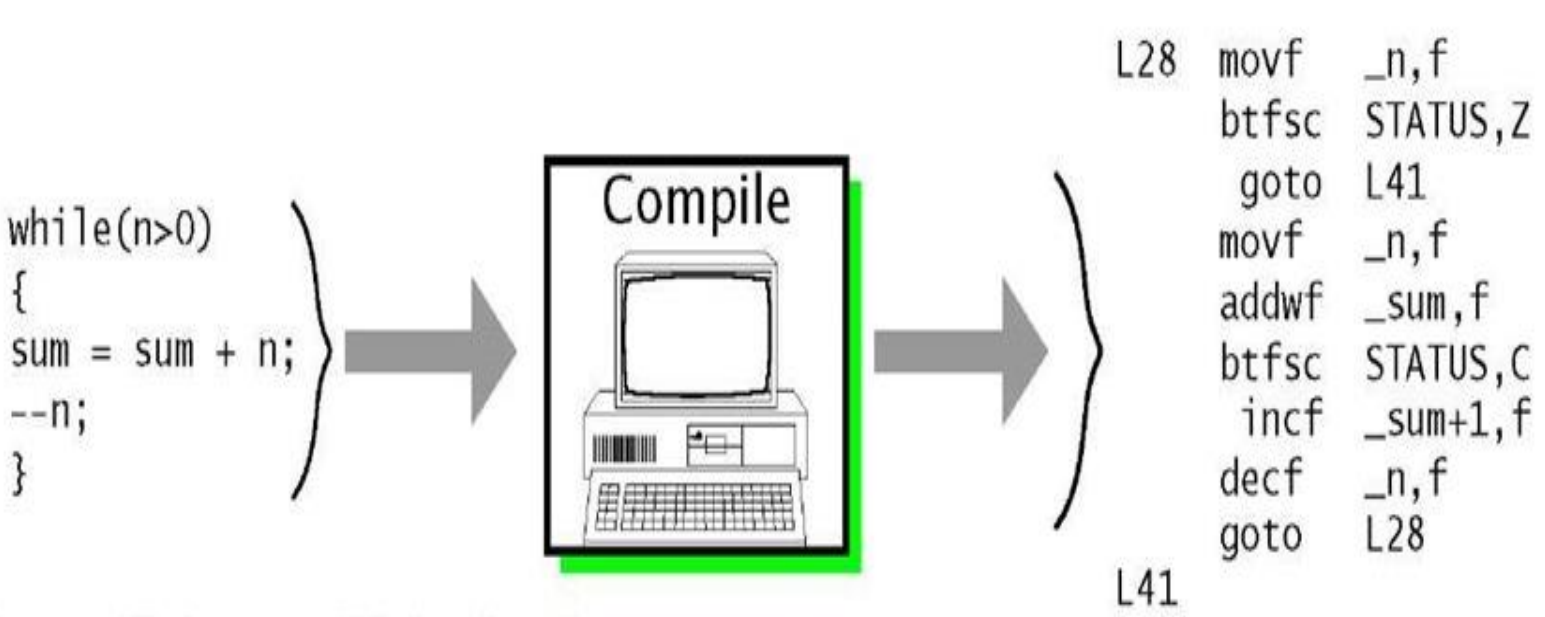
Low-level program

```
LOAD r1,b
LOAD r2,h
MUL r1,r2
DIV r1,#2
RET
```

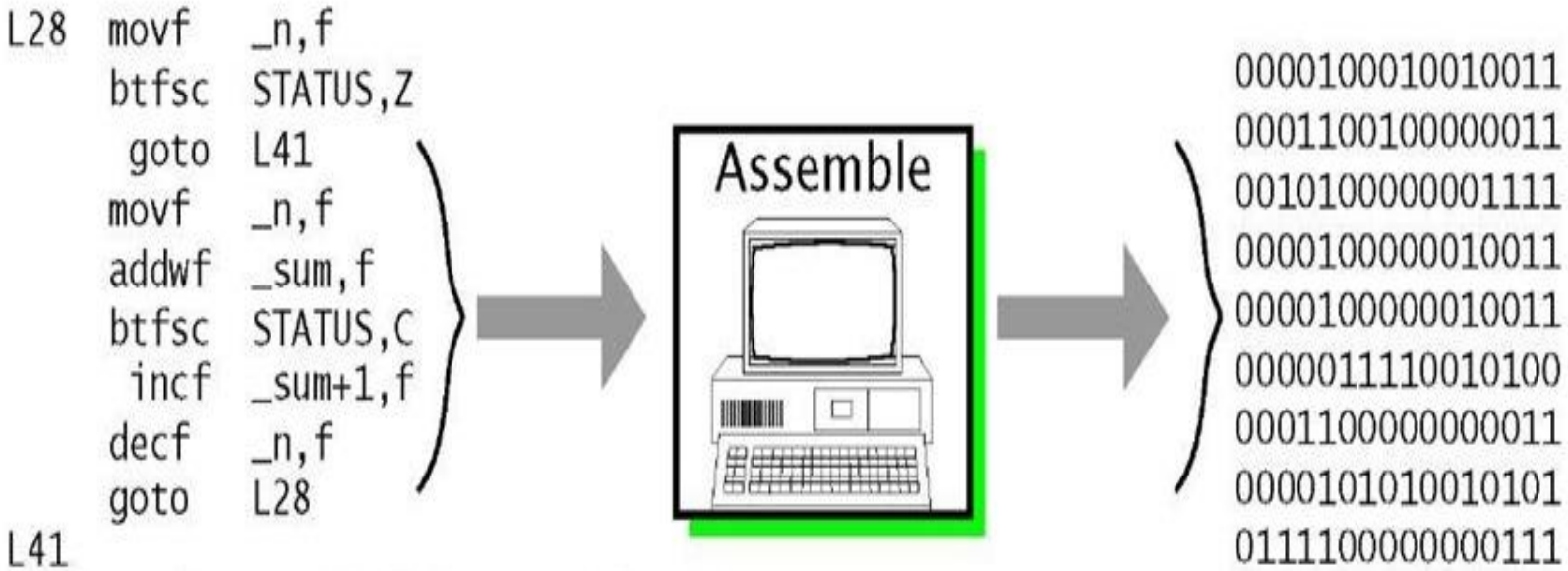
Executable Machine code

```
0001001001000101
0010010011101100
10101101001...
```





(a) First, compile to assembly-level code.

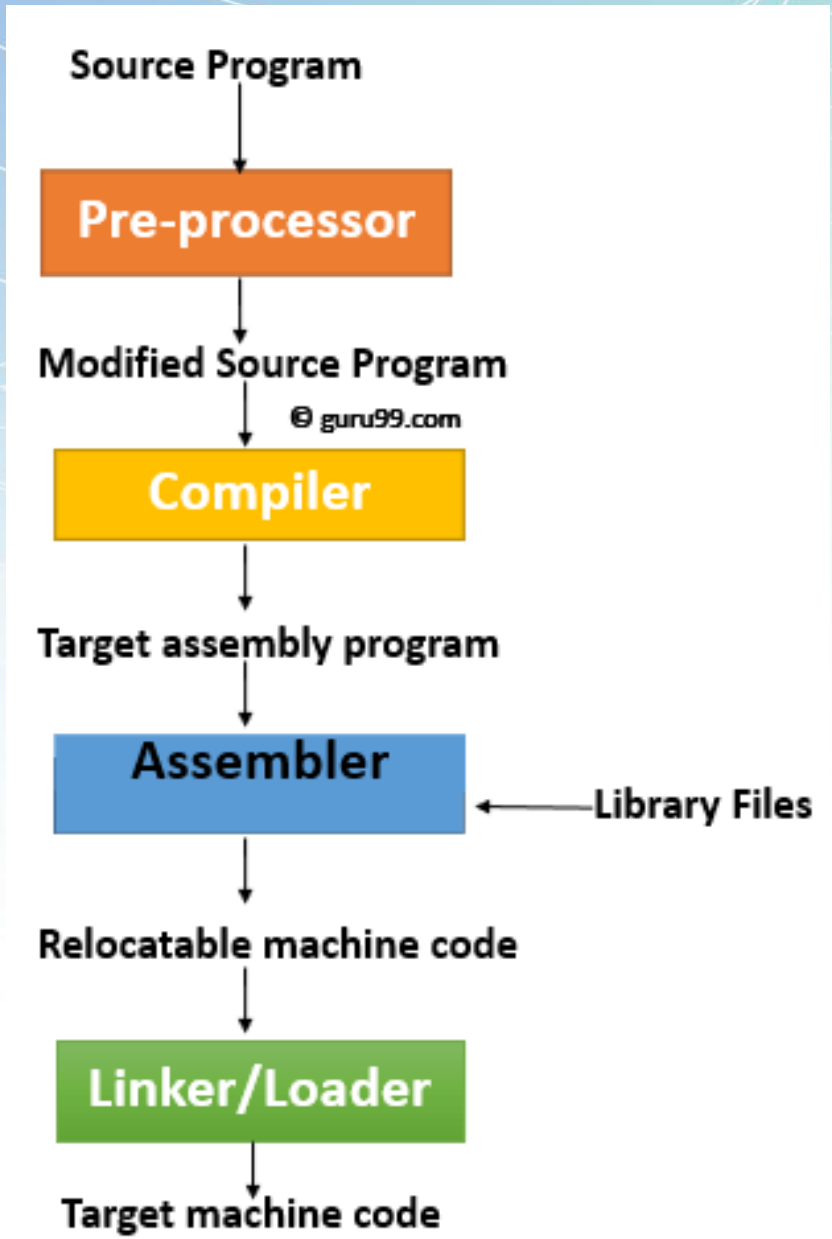


(b) Second, assemble-link to machine code.

- *Machine Level Programming*- Writing programs in the language which machines can understand(0's and 1's)
- *Assembly Level Programming*- Operational codes represented as short words called Mnemonics
- *High Level Programming*- Writing programs much similar to the language used by humans
- Need for language translators-To bridge the gap between HLL and MLL
- Assembler, Compiler, Interpreter

- A compiler is a computer program which helps you transform source code written in a high-level language into low-level machine language.
- It translates the code written in one programming language to some other language without changing the meaning of the code.

Language Processing System



- The high-level language is converted into binary language in various phases.
- A **compiler** is a program that converts high-level language to assembly language.
- Similarly, an **assembler** is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

- User writes a program in C language (high-level language).
- The C compiler, compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).
- A linker tool is used to link all the parts of the program together for execution (executable machine code).
- A loader loads all of them into memory and then the program is executed.

Preprocessor

- A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers.
- It deals with macro-processing, augmentation, file inclusion, language extension, etc.

Interpreter

- An interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence.
- If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

Difference between Compiler and Interpreters

COMPARISON	COMPILER	INTERPRETER
Input	It takes an entire program at a time.	It takes a single line of code or instruction at a time.
Output	It generates intermediate object code.	It does not produce any intermediate object code.
Working mechanism	The compilation is done before execution.	Compilation and execution take place simultaneously.
Speed	Comparatively faster	Slower
Memory	Memory requirement is more due to the creation of object code.	It requires less memory as it does not create intermediate object code.
Errors	Display all errors after compilation, all at the same time.	Displays error of each line one by one.
Error detection	Difficult	Easier comparatively
Pertaining Programming languages	C, C++, C#, Scala, typescript uses compiler.	Java, PHP, Perl, Python, Ruby uses an interpreter.

Linker

- Linker is a computer program that *links and merges various object files together* in order to make an executable file.
- All these files might have been compiled by **separate assemblers**
- The main task of a linker is to search for called modules in a program and to find out the memory location where all modules are stored.

Loader:

- The loader is a part of the OS, which performs the tasks of loading executable files into memory and run them

Why study compilers?

- Application of a wide range of theoretical techniques
  - Data Structures
  - Theory of Computation
  - Algorithms
  - Computer Architecture
- Good SW engineering experience
- Better understanding of programming languages

Where are compilers used?

Implementation of programming languages

C, C++, Java, Lisp, Prolog, SML, Haskell, Ada, Fortran

Document processing

DVI → PostScript,
Word documents → PDF

Natural language processing

NL → database query language → database commands

Hardware design

silicon compilers, CAD data → machine operations, equipment lists

Report generation

CAD data → list of parts,

All kinds of input/output translations

various UNIX text filters, . . .

ANALYSIS OF THE SOURCE PROGRAM

- In compiling, analysis consists of three phases:
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis

- In a compiler linear analysis is called lexical analysis or scanning.
- The lexical analysis phase reads the characters in the source program and grouped into **tokens** that are sequence of characters having a collective meaning.

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

This is grouped into tokens →

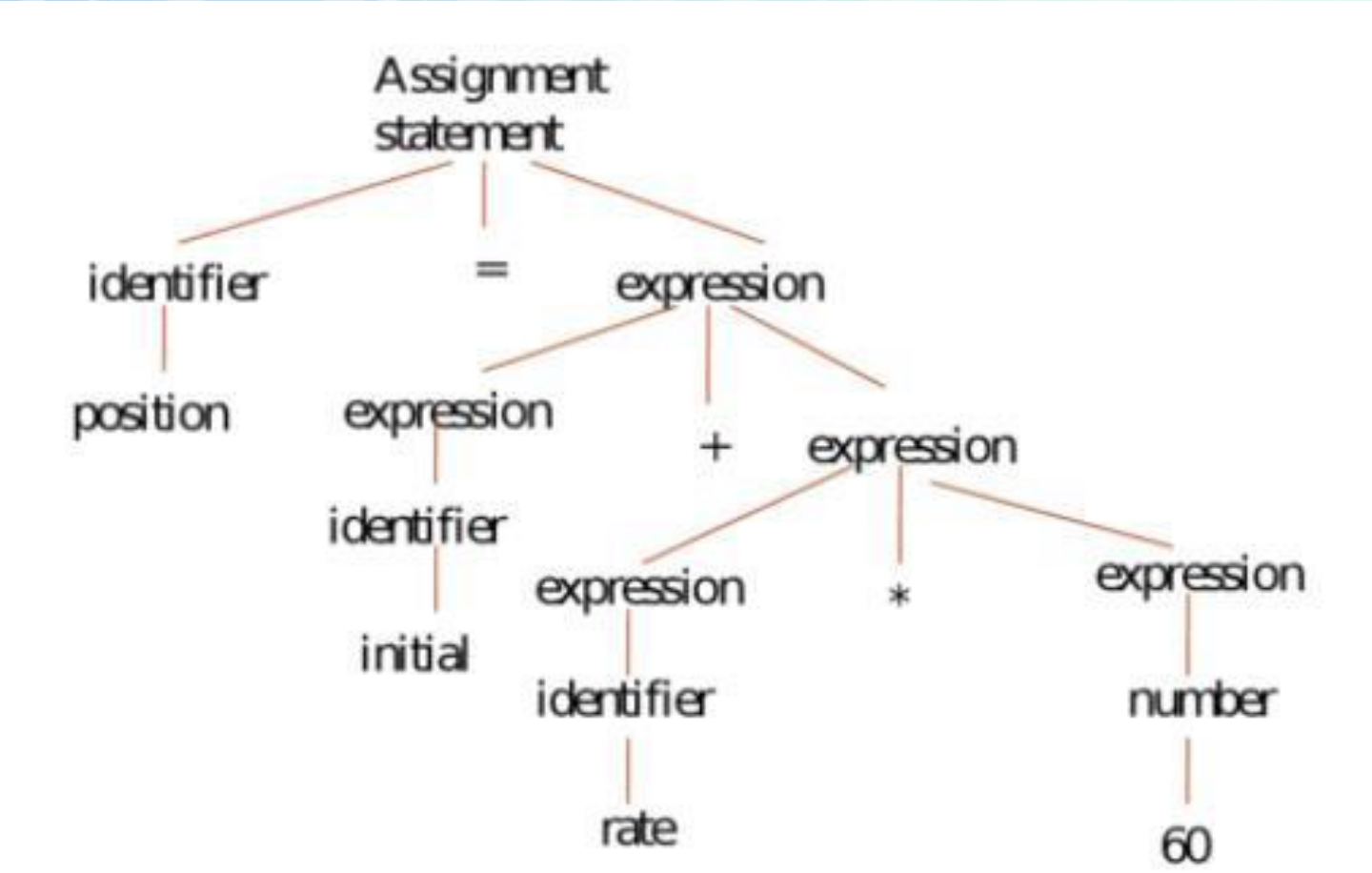
1. The identifier position.
2. The assignment symbol : =
3. The identifier initial
4. The plus sign
5. The identifier rate
6. The multiplication sign
7. The number 60

- Blanks separating characters of these tokens are normally eliminated during lexical analysis.

Syntax Analysis /parsing

- Hierarchical Analysis is called parsing or syntax analysis.
- It involves grouping the tokens of the source program into grammatical phrases that are used by the complier to synthesize output.
- They are represented using a syntax tree.

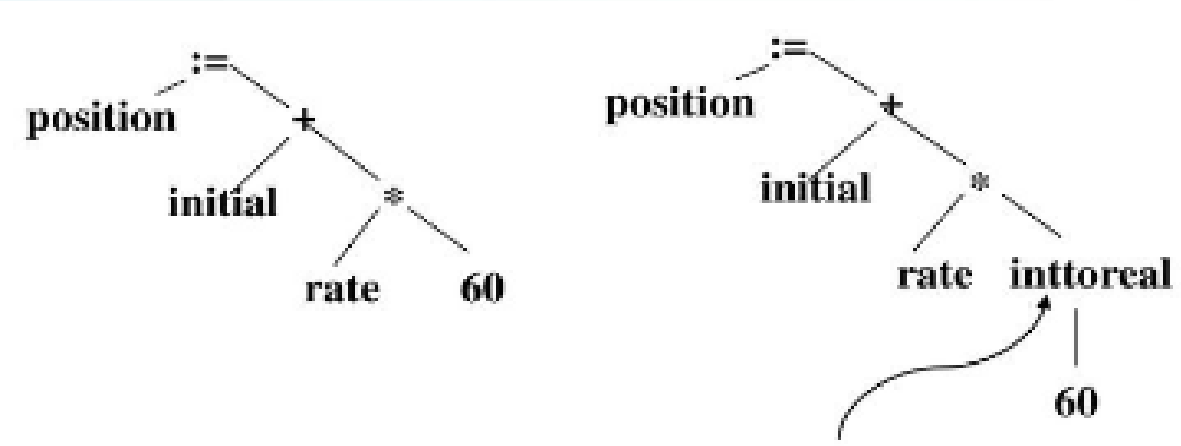
- A syntax tree is a compressed representation of parse tree
- Operators appear as **interior nodes, operands** of an operator are children of the node for that operator



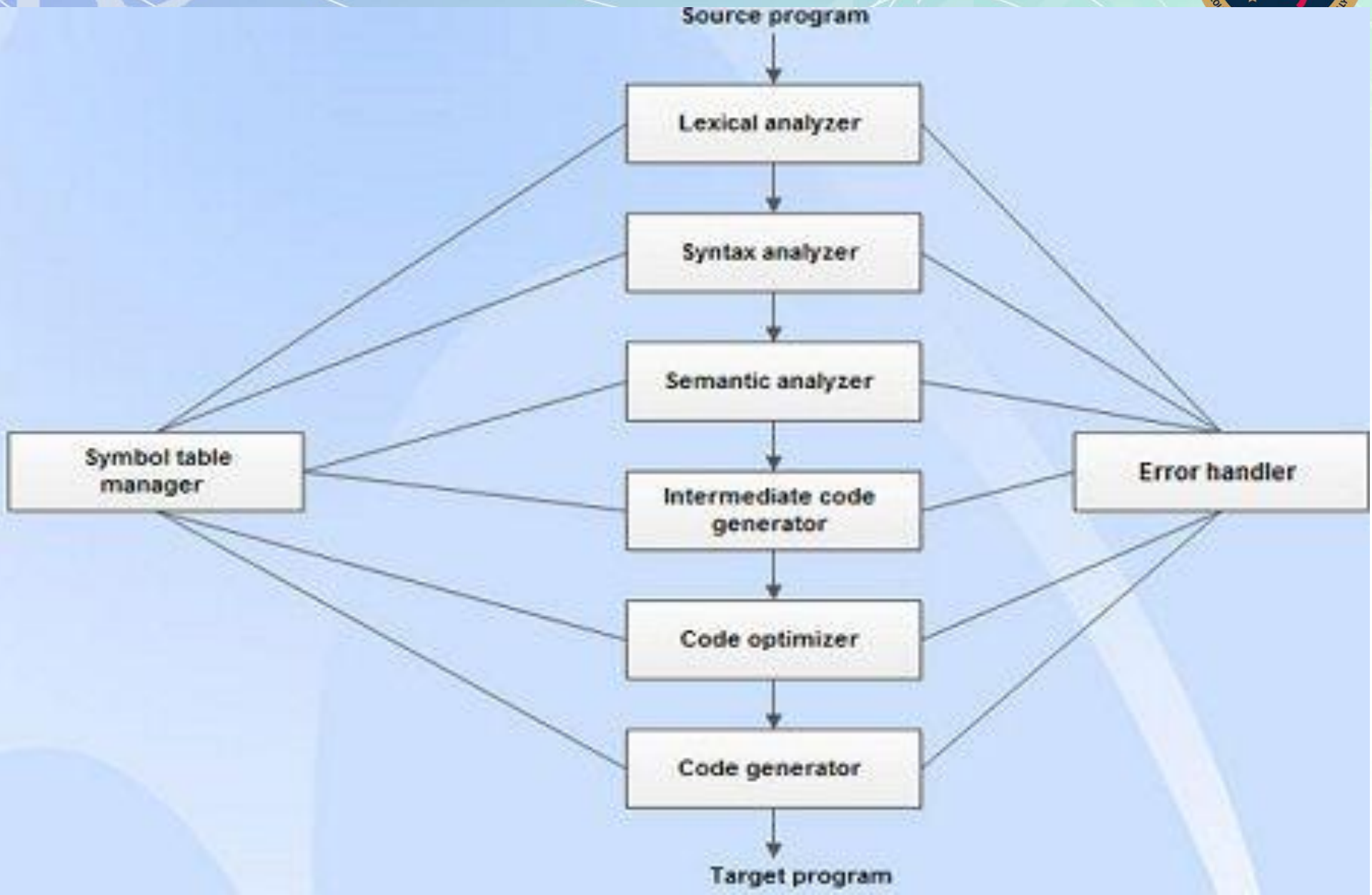
Semantic Analysis

- This phase checks the source program for semantic errors and gathers type information for subsequent code generation phase.
- It uses hierarchical structure determined by syntax-analysis phase to identify operators and operands of expressions
- An important component of semantic analysis is type checking.
- Here the compiler checks that each operator has operands that are permitted by the source language specification.

- Suppose for eg: position := initial + rate * 60
  - All identifiers have been declared to be reals and 60 by itself is assumed to be an integer
- Type checking reveals that * is applied to a real, rate and an integer
- Hence the general approach is to convert integer to real
- This is achieved by creating an extra node for the operator **inttoreal** that converts int to real



PHASES OF COMPILER



I. Lexical analysis

- Reads characters in the source program and group them into meaningful pieces called tokens
- In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as **tokens.**
- The character sequence forming a token is called as lexeme for the token(It is nothing but an instance of a token.)
- Blank spaces separating the characters of these tokens are normally eliminated

Basic Terminologies

- A **lexeme** is a sequence of characters that are included in the source program according to the matching pattern of a token.

  It is nothing but an instance of a token.

- The **token** is a sequence of characters which represents a unit of information in the source program.

- A **pattern** is a description which is used by the token.

In the case of a keyword which uses as a token, the pattern is a sequence of characters.

eg : **new = old+10**

Lexeme	**Token**
new	id1
=	assignment operator
old	id2
+	plus operator
10	Constant

Suppose we pass a statement through lexical analyzer –

a = b + c ; It will generate token sequence like this:

id=id+id; Where each id refers to it's variable in the symbol table referencing all details

Exercise 1:

Count number of tokens :

int max(int i);

Answer:

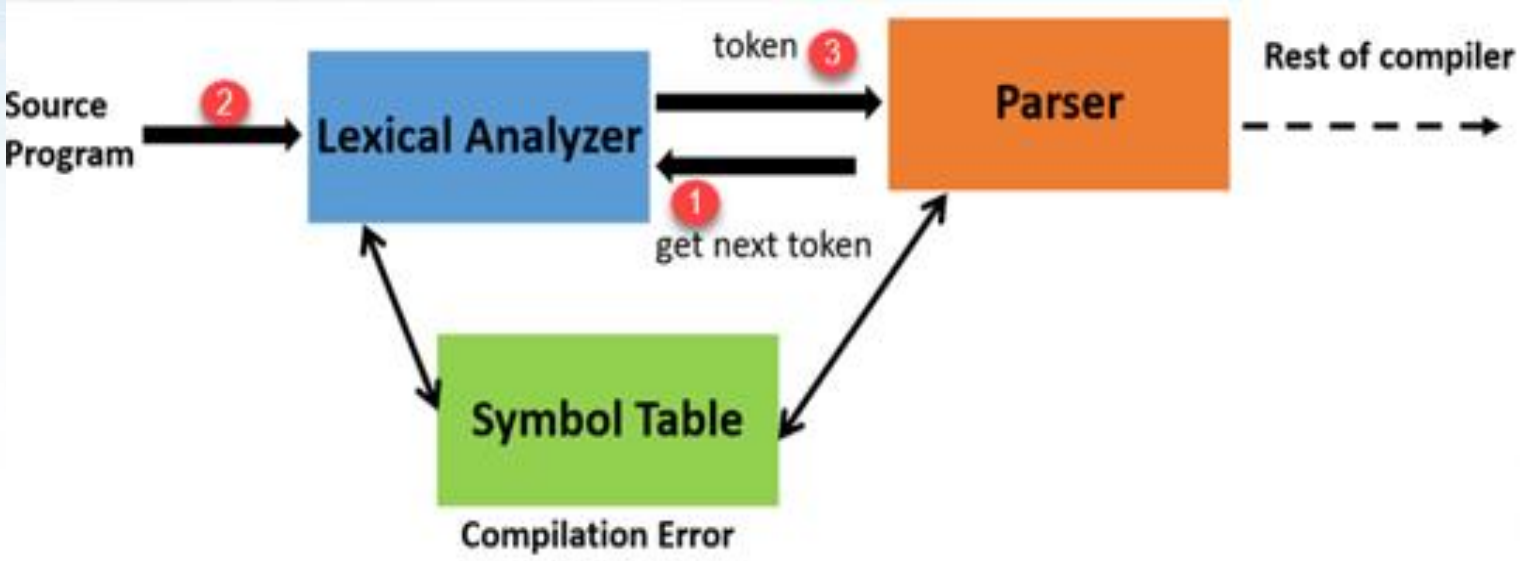
Total number of tokens 7: int, max, (,int, i,), ;

- Lexical analyzer first read int and finds it to be valid and accepts as token
- max is read by it and found to be a valid function name after reading (
- int is also a token , then again i as another token and finally ;

Lexical Analyzer Architecture: How tokens are recognized

- The main task of lexical analysis is to read input characters in the code and produce tokens.
- Lexical analyzer scans the entire source code of the program.

  It identifies each token one by one. Scanners are usually implemented to produce tokens only when requested by a parser.



1. "Get next token" is a command which is sent from the Parser to the lexical analyzer.
2. On receiving this command, the lexical analyzer scans the input until it finds the next token.
3. It returns the token to Parser.

- Lexical Analyzer skips whitespaces and comments while creating these tokens.
- If any error is present, then Lexical analyzer will correlate that error with the source file and line number.

Roles of the Lexical analyser

Lexical analyser performs below given tasks:

- Helps to identify token into the symbol table
- Removes white spaces and comments from the source program
- Correlates error messages with the source program
- Helps to expands the macros if it is found in the source program
- Read input characters from the source program

Consider the following code that is fed to Lexical Analyzer

```
#include <stdio.h>
 int maximum(int x, int y)
{
 // This will compare 2 numbers

if (x > y)
        return x;
 else
        {
                return y;
        }
 }
```

Examples of Tokens

Lexeme	Token
int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
Y	Identifier
)	Operator
{	Operator
If	Keyword

Examples of Non-tokens

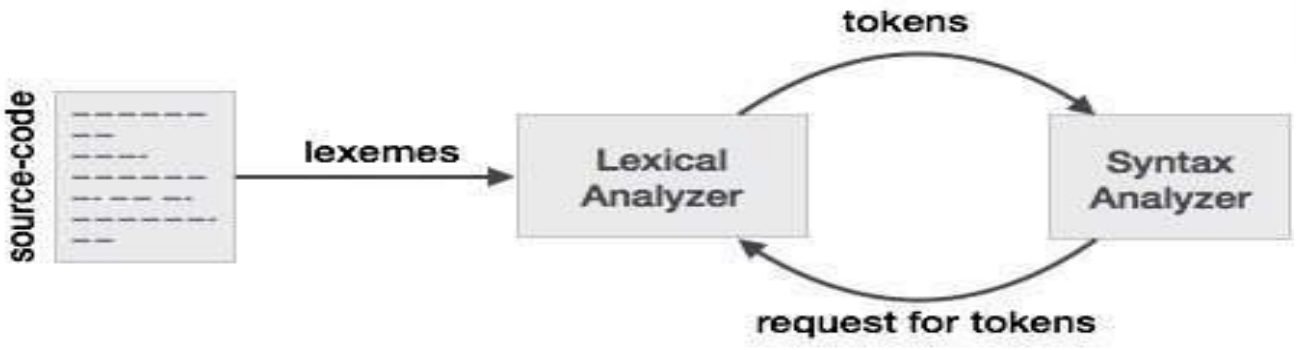
Type	Examples
Comment	// This will compare 2 numbers
Pre-processor directive	#include <stdio.h>
Pre-processor directive	#define NUMS 8,9
Macro	NUMS
Whitespace	/n /b /t

- For each lexeme, the lexical analyzer produces as output a token of the form

  < token- name, attribute-value >

- In the token,

  1. the first component token- name is an abstract symbol that is used during syntax analysis, and
  2. the second component attribute-value points to an entry in the symbol table for this token.



- Information from the symbol-table entry 'is needed for semantic analysis and code generation.

- Consider the assignment statement

position = initial + rate * 60

- The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. **position** is a lexeme that would be mapped into a token **<id, 1>**,
   - where id is an abstract symbol standing for identifier
   - 1 points to the symbol table entry for position.

The symbol-table entry for an identifier holds information about the identifier, such as its name and type

position = initial + rate * 60

2. = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. For notational convenience, the lexeme itself is used as the name of the abstract symbol.
3. “initial” is a lexeme that is mapped into the token (id, 2), where 2 points to the symbol-table entry for initial.
4. + is a lexeme that is mapped into the token (+).
5. “rate” is a lexeme mapped into the token (id, 3), where **3** points to the symbol-table entry for rate.
6. * is a lexeme that is mapped into the token (*) .
7. 60 is a lexeme that is mapped into the token (60)

- The representation of the assignment statement

position = initial + rate * 60

after lexical analysis as the sequence of tokens as:

< id, l > < = > <id, 2> <+> <id, 3> < * > <60>

Blanks separating the lexemes would be discarded by the lexical analyzer.

summary of lexical analyser

- Remove comments and white spaces (or scanning)
- Macros expansion
- Read input characters from the source program
- Group them into lexemes
- Produce as output a sequence of tokens
- Interact with the symbol table
- Correlate error messages generated by the compiler with the source program

Syntax Analysis

- The second phase of the compiler is syntax analysis or parsing.
- The parser uses the first components of the tokens produced by the lexical analyzer to create a *tree-like intermediate representation* that depicts the grammatical structure of the token stream.
- A typical representation is a **syntax tree** in which each
  - interior node represents an operation
  - the children of the node represent the arguments of the operation.

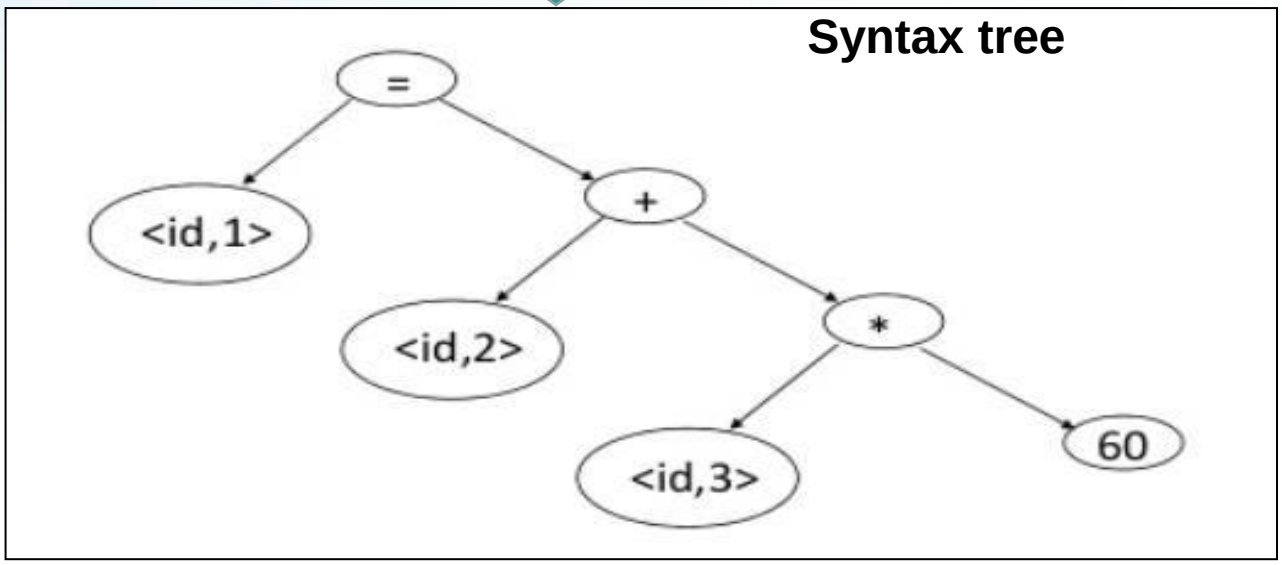
position = initial + rate * 60

Lexical Analysis

Sequence of Tokens

< id, l > < = > <id, 2> <+> <id, 3> < * > <60>

Syntax Analysis

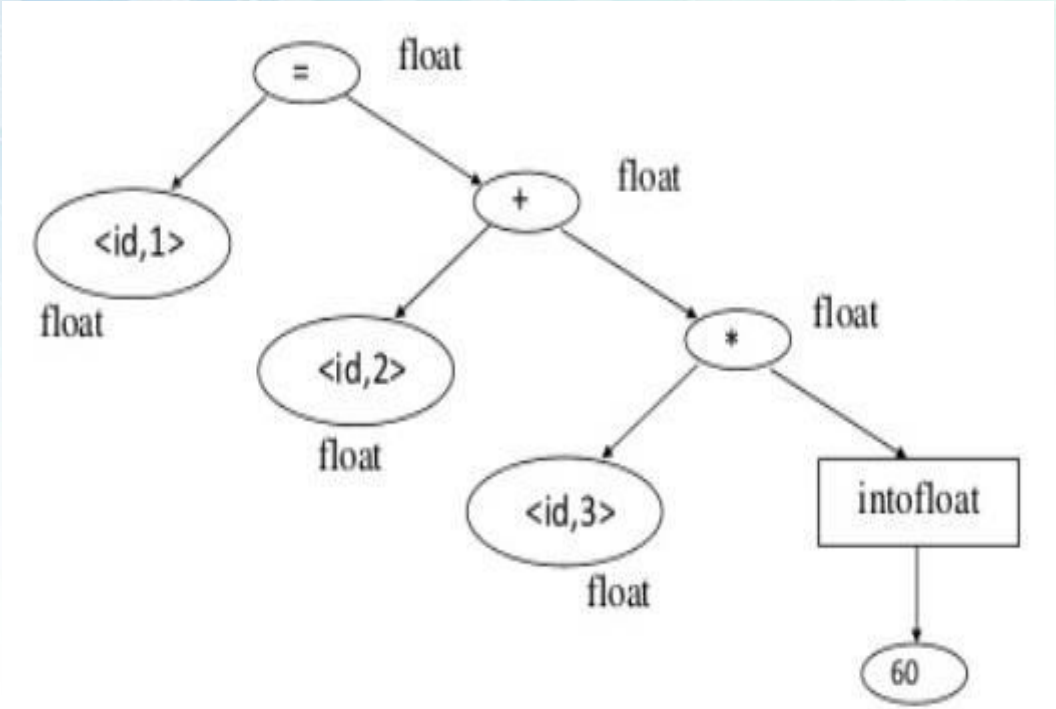


Semantic Analysis

- The Semantic analyzer uses the **syntax tree** and the **information in the symbol table** to check the source program for semantic consistency with the language definition.
- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate –code generation
- An important part of semantic analysis is **type checking**, where the compiler checks that each operator has matching operands.
- For example, many programming language definitions require an array index to be an integer; the *compiler must report an error* if a floating-point number is used to index an array.

- For example, if the operator is applied to a floating point number and an integer, the compiler may convert the integer into a floating point number.
- In our example, suppose that position, initial, and rate have been declared to be floating- point numbers, and that the lexeme 60 by itself forms an integer.
- The semantic analyzer discovers that the operator * is applied to a floating-point number rate and an integer 60.
- In this case, the integer may be converted into a floating-point number.

position = initial + rate * 60



First 3 phases.. So far

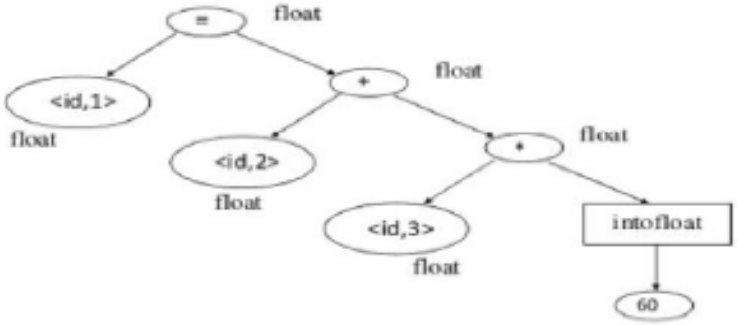
position = initial + rate * 60

LEXICAL ANALYZER

< id, 1 > < = > <id, 2> <+> <id, 3> < * > <60>

SYNTAX ANALYZER

SEMANTIC ANALYZER



INTERMEDIATE CODE GENERATOR

Intermediate Code Generation

- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.
- Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.
- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation
- This intermediate representation should have two important properties:
  - It should be simple and easy to produce
  - It should be easy to translate into the target machine

- In our example, the intermediate representation used is three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction.

Eg : id1 = id2 + id3 * 60

t1 = inttofloat(60)

t2 = id3 * t1

t3 = id2 + t2

id1 = t3

Code Optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- The objectives for performing optimization are:
  - faster execution, shorter code, or target code that consumes less power.
- In our example , id1 = id2 + id3 * 60 , the optimized code is:

  t1 = id3 * 60.0

  id1 = id2 + t1

- A significant fraction of the time of compilers is spent on this phase

Code Generator

- Last phase of compiler consisting relocatable machine code or assembly code
- The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.

- In our example, the code generated is:

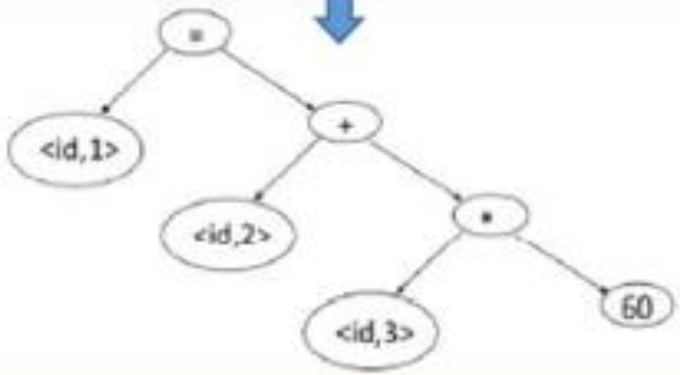
```
LDF   R2, id3
MULF  R2, #60.0
LDF   R1, id2
ADDF  R1, R2
STF   id1, R1
```

- The first operand of each instruction specifies a destination.
- The F in each instruction tells us that it deals with floating-point numbers.
- The above code loads the contents of address id3 into register R2, then multiplies it with floating-point constant 60.0.
- The # signifies that 60.0 is to be treated as an immediate constant.
- The third instruction moves id2 into register Rl and the fourth adds to it the value previously computed in register R2.
- Finally, the value in register Rl is stored into the address of idl , so the code correctly implements the assignment statement **position = initial + rate * 60.**

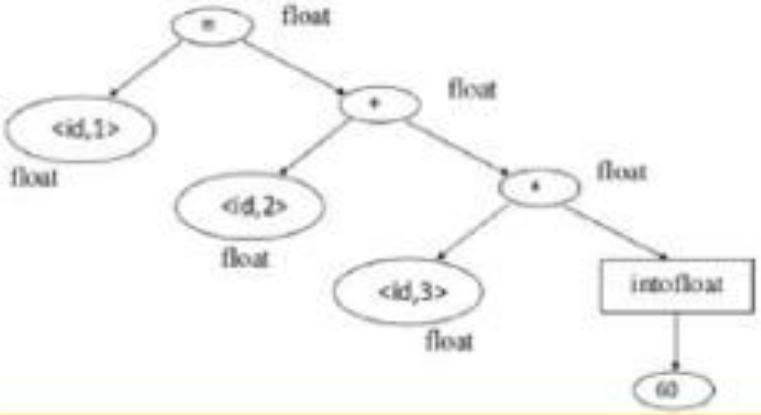
LEXICAL ANALYZER

< id, 1 > < = > <id, 2> <+> <id, 3> < * > <60>

SYNTAX ANALYZER



SEMANTIC ANALYZER



INTERMEDIATE CODE GENERATOR

INTERMEDIATE CODE GENERATOR

t1 = inttofloat(60)

t2 = id3 * t1

t3 = id2 + t2

id1 = t3

CODE OPTIMIZER

t1 = id3 * 60.0

id1 = id2 + t1

CODE GENERATOR

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

Symbol Table Management

- A symbol table contains a record for each identifier with fields for the attributes of the identifier.
- This component makes it easier for the compiler to search the identifier record and retrieve it quickly.
- The symbol table also helps you for the scope management.
- The symbol table and error handler interact with all the phases and symbol table update correspondingly.

GROUPING OF PHASES

- The process of compilation is split up into following phases:
  - Analysis Phase
  - Synthesis phase

Analysis Phase

- Analysis Phase performs 4 actions namely:

  a. Lexical analysis

  b. Syntax Analysis

  c. Semantic analysis

  d. Intermediate Code Generation

- The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them.
- It then uses this structure to create an intermediate representation of the source program.
- If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.
- The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

Front and Back Ends:

– The phases are collected into front end and back end.

– The **front end** consists of phases or part of phases that depends primarily on **source language** and is largely independent of the target machine.

• These normally include lexical and syntactic analysis, the creation of symbol table, semantic analysis intermediate code generation, some code optimization and error reporting.

– The **back end** includes portions of the compiler that depend on the **target machine** and independent of source language.

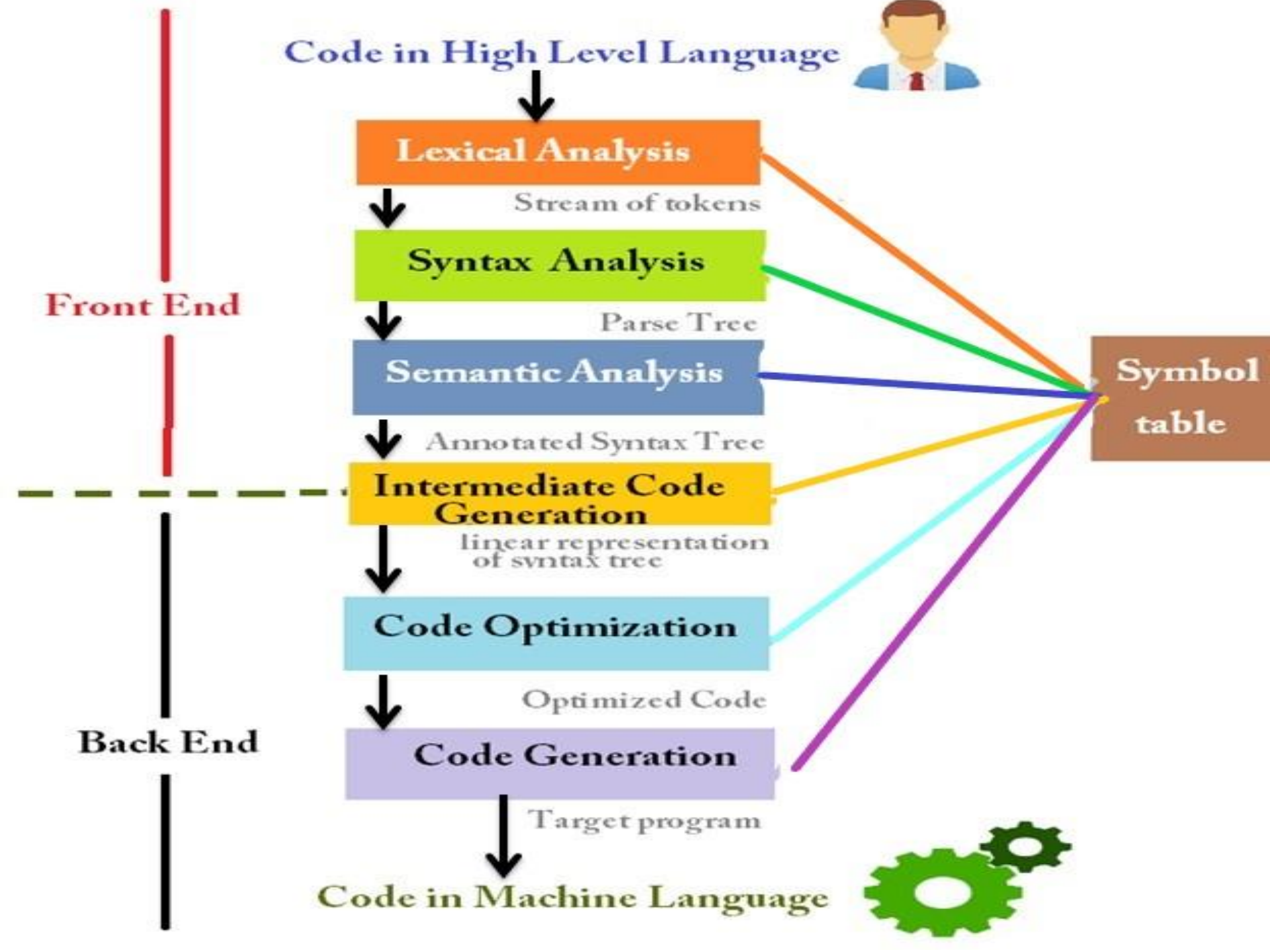
• This includes code optimization, code generation and symbol table operations.

Symbol table

- It is a data structure containing a *record for each identifier*, with fields for the attributes of the identifier.

- The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

- When an identifier in the source program is detected by the lexical Analyzer, the identifier is entered into the symbol table if it is not duplicate


Code in High Level Language
Lexical Analysis
Stream of tokens
Syntax Analysis
Parse Tree
Semantic Analysis
Annotated Syntax Tree
Intermediate Code Generation
linear representation of syntax tree
Code Optimization
Optimized Code
Code Generation
Target program
Code in Machine Language
Front End
Back End
Symbol table

Error Detection and Reporting

- Each phase can encounter some errors.
- After detecting an error, a phase must deal with that error, so that compilation can proceed allowing further errors in the source program to be detected.
- Syntax and semantic phase handle large fraction of errors

Eg: i) Lexical Errors : (Don't form any token of that language)

"1bcd, switc etc.,"

ii) Syntax Errors: (Token stream violates the structure rules of the language). " Missing of parenthesis, braces etc.,"

iii) Semantic Errors:(No meaning to the operation involved)

"a is not used";

Error Handling Routine

In the compiler design process error may occur in all the below-given phases:

- Lexical analyzer: Wrongly spelled tokens
- Syntax analyzer: Missing parenthesis
- Intermediate code generator: Mismatched operands for an operator
- Code Optimizer: When the statement is not reachable
- Code Generator: Unreachable statements
- Symbol tables: Error of multiple declared identifiers

COMPILER WRITING TOOLS

- Compiler writers use software development tools and more specialized tools for implementing various phases of a compiler. Some commonly used compiler construction tools include the following.
  - Parser Generators
  - Scanner Generators
  - Syntax-directed translation engine
  - Automatic code generators
  - Data-flow analysis Engines

Scanner Generators

- Input : Regular Expression description of the token of a language
- Output: Lexical Analyzers
- These automatically generate Lexical Analyzers, normally from a specification based on Regular Expression
- Eg : lex (Lexical Analyzer Generator)
- Lex facilitate lexical analysis, the processing of character sequences such as source code to produce symbol sequences called tokens for use as input to other programs such as parsers.

- **Parser Generators**
  - Input: CFG; Output: Syntax Analyzers.
  - They produce syntax analyzers from CFG.
  - Use powerful parsing algorithms.
  - Syntax analysis phase is highly complex and consumes manual and compilation time, these parser generator are highly useful.
  - Parser generator takes the grammatical description of a programming language and produces a syntax analyzer
  - Ex: Yacc

- Syntax directed translation engines

  Input: Parse tree

  Output: Intermediate code.

  - Collection of routines that traverse the parse tree and generate intermediate code
  - Translations are associated with nodes of the parse tree

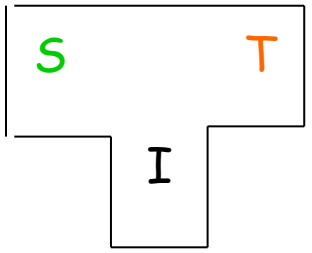
- Data Flow Engines
  - Much needed for code optimization
  - Help to analyze the flow of data and transformations

- Automatic Code Generators
  - Contains rules for translating intermediate code into machine language code
  - Capable of handling different access methods for data, register variables, stack data
  - Intermediate code statements are replaced with templates that represents sequence of machine instructions

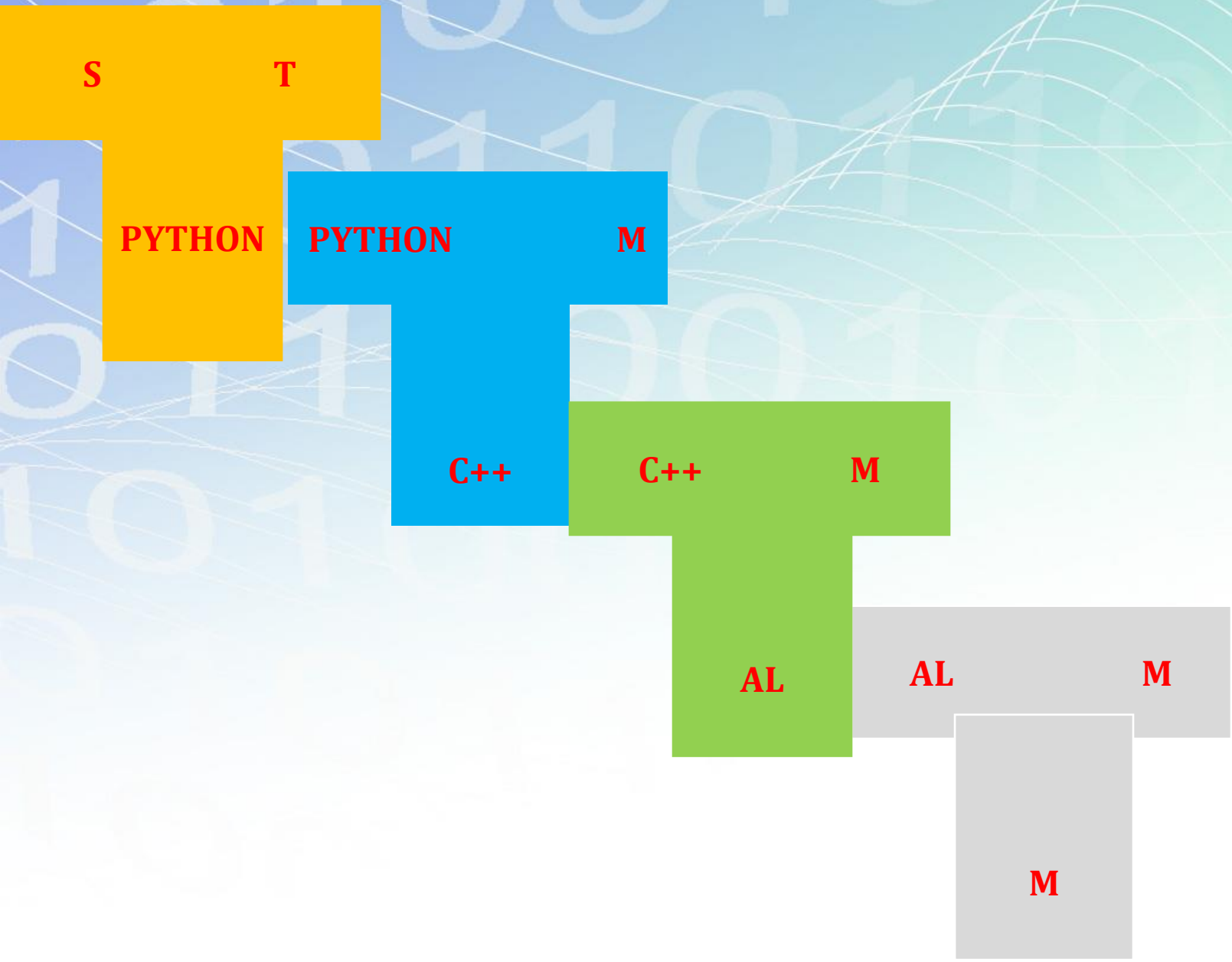
Bootstrapping

- A compiler can be characterized by three languages: the source language (S), the target language (T), and the implementation language (I)
- The three language S, I, and T can be quite different.
- This is represented by a T-diagram as:
- In textual form this can be represented as

$S_I T$



CONSIDER AN EXAMPLE Bootstrapping



MACHINE CODE

Bootstrapping:

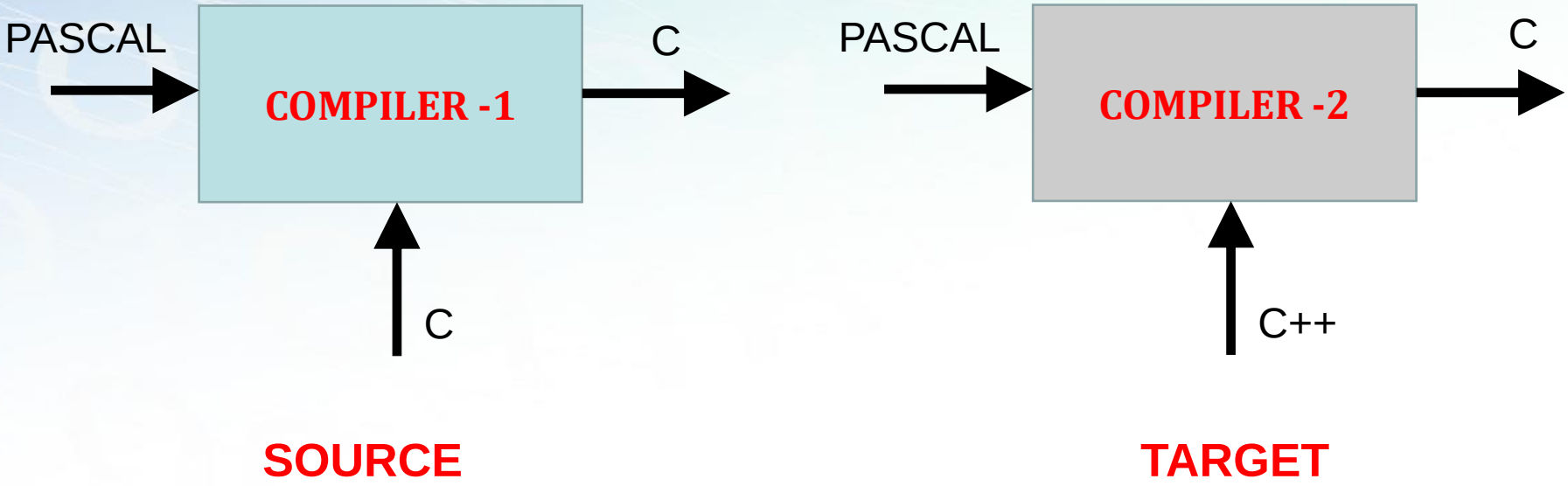
- The process of writing a compiler (or Assembler) in the target programming language which has to be compiled is known as "Bootstrapping"
- A compiler leads to a self-hosting compiler by applying Bootstrapping technique
- Many compilers for many programming languages are bootstrapped
- Examples: BASIC, ALGOL, C, PASCAL, JAVA, PYTHON, etc.

- **Advantages of Bootstrapping**

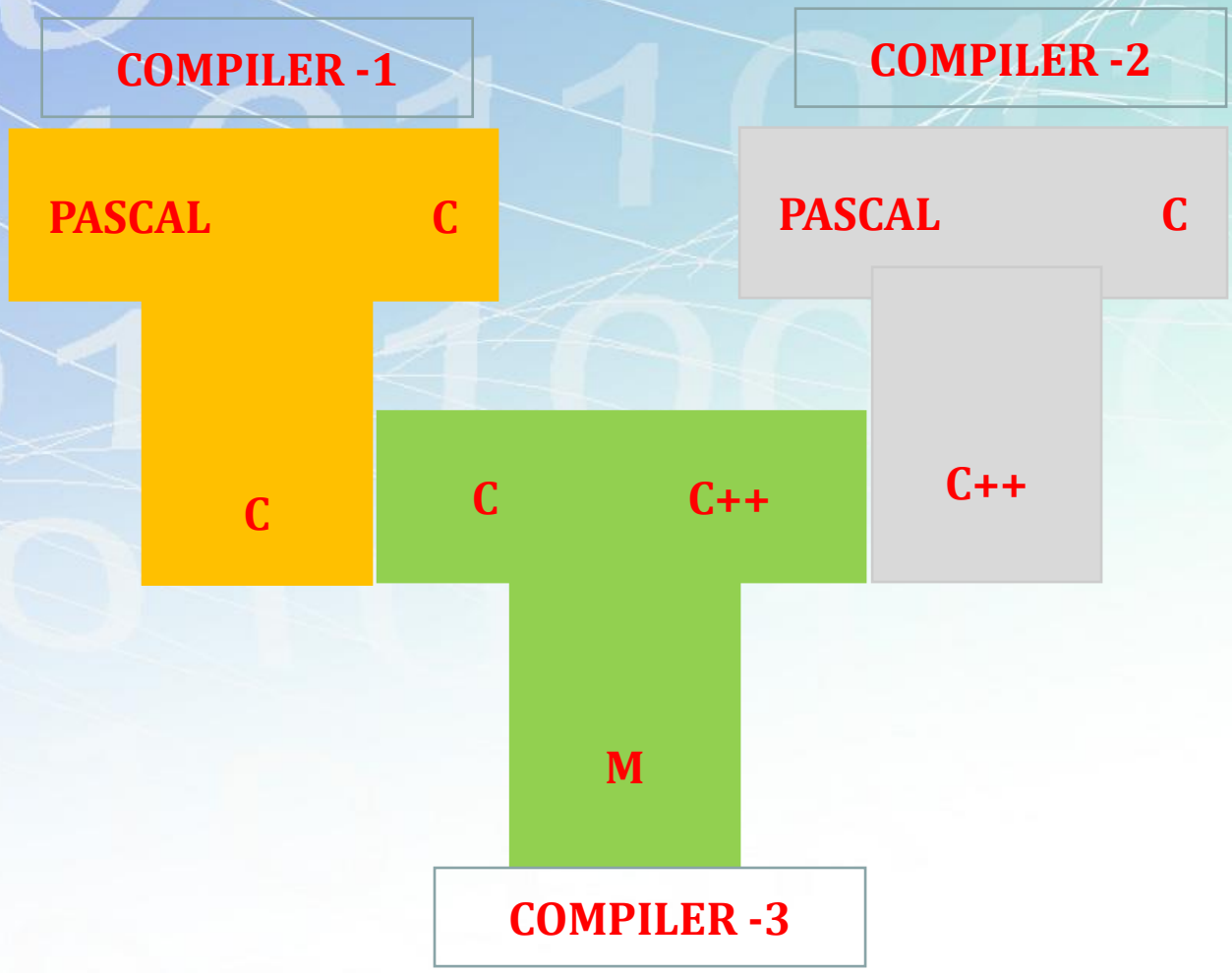
  It is a non-trivial test of the language being compiled
- Compiler developers only need to know the language being compiled
- Compiler development can be done in the higher level language being compiled
- Improvements to the compiler's back-end improve not only general purpose programs but also the compiler itself
- It is a comprehensive consistency check as it should be able to reproduce its own object code

Cross Compiler:

- A compiler, that run on one machine and produce the target code for another machine. Such a compiler is called **cross compiler**
- Source language L, the target language N gets generated, which runs on machine M



Cross Compiler




Thank
You