JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

Year & Sem – 3rd Year & 5th Sem

Subject – COMPILER DESIGN

Unit – 1

Presented by – (Abhishek Dixit, Assistant Prof., Dept of CSE)

VISION AND MISSION OF INSTITUTE

To become a renowned center of outcome based learning and work towards academic, professional, cultural and social enrichment of the lives of individuals and communities

M1: Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.

M2: Identify, based on informed perception of Indian, regional and global needs, the areas of focus and provide platform to gain knowledge and solutions.

M3: Offer opportunities for interaction between academia and industry.

M4: Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

VISION OF THE DEPARTMENT

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

MISION OF THE DEPARTMENT

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.

M2: To provide opportunities for interaction between academia and industry.

M3: To provide platform for lifelong learning by accepting the change in technologies

M4: To develop aptitude of fulfilling social responsibilities

PROGRAM OUTCOMES

Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM EDUCATIONAL OBJECTIVES

1. To provide students with the fundamentals of Engineering Sciences with more emphasis in Computer Science &Engineering by way of analyzing and exploiting engineering challenges.
2. To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.
3. To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues.
4. To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self motivated life-long learning needed for a successful professional career.
5. To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge

PROGRAM SPECIFIC OBJECTIVES

1. PSO1. Ability to interpret and analyze network specific and cyber security issues, automation in real word environment.

2. PSO2. Ability to Design and Develop Mobile and Web-based applications under realistic constraints.

COURSE OUTCOME

CO1: Compare different phases of compiler and design lexical analyzer.

CO2: Examine syntax and semantic analyzer by understanding grammars.

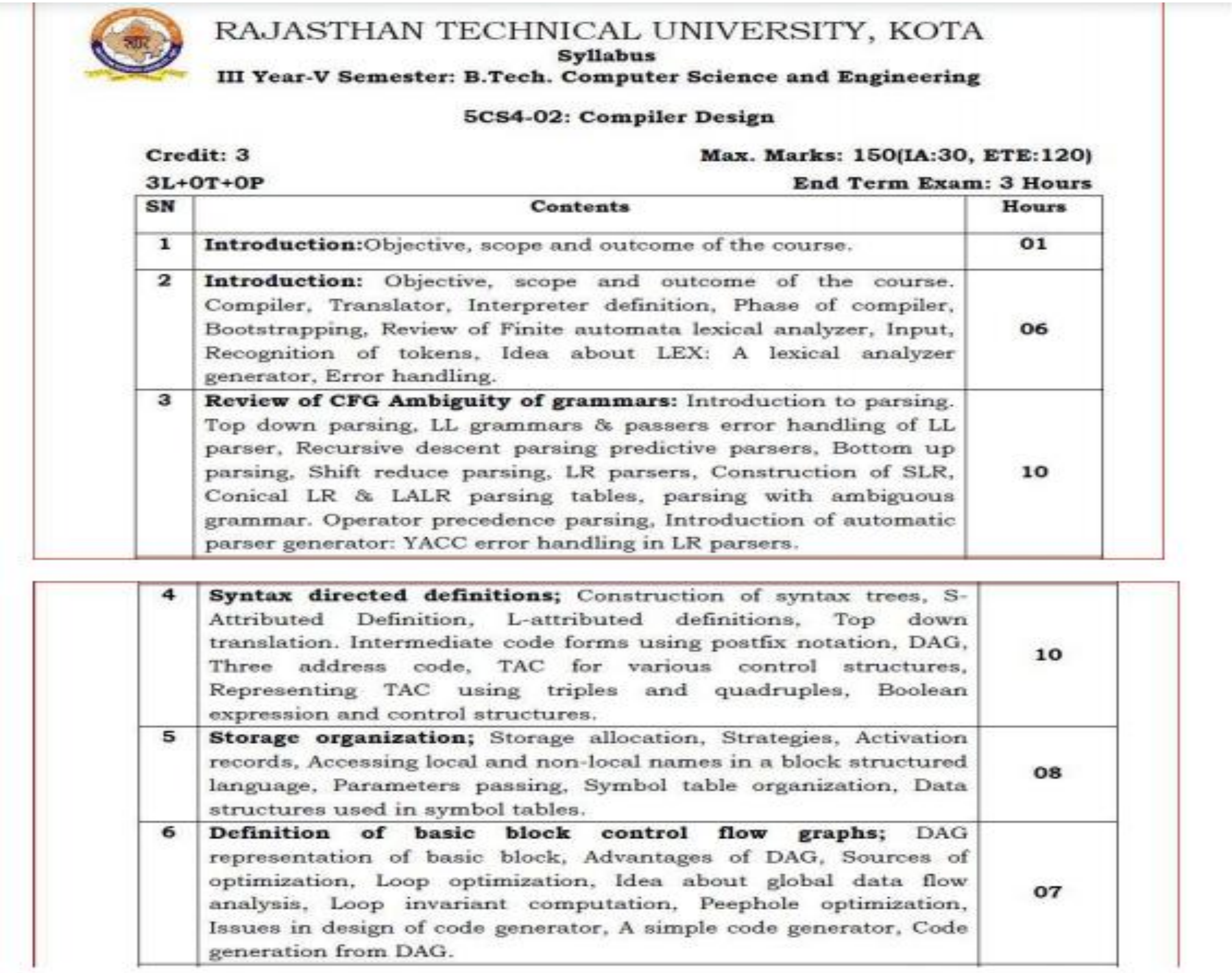
CO3: Illustrate storage allocation and its organization & analyze symbol table organization.

CO4: Analyze code optimization, code generation & compare various compilers.

CO-PO MAPPING

Semester	Subject	Code	L/T/P	CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
V	COMPILER DESIGN	5CS4-02	L	1. Compare different phases of compiler and design lexical analyzer..	3	3	3	3	1	1	1	1	1	2	1	3
			L	2. Examine syntax and semantic analyzer by understanding grammars.	3	3	3	2	1	1	1	0	1	2	1	3
			L	3. Illustrate storage allocation and its organization & analyze symbol table organization.	3	3	2	2	1	1	1	1	1	2	1	3
			L	4.Analyze code optimization, code generation & compare various compilers.	3	3	3	3	2	1	1	1	1	2	1	3

SYLLABUS



RAJASTHAN TECHNICAL UNIVERSITY, KOTA

Syllabus

III Year-V Semester: B.Tech. Computer Science and Engineering

5CS4-02: Compiler Design

Credit: 3 **Max. Marks: 150(IA:30, ETE:120)**

3L+0T+0P **End Term Exam: 3 Hours**

SN	Contents	Hours
1	**Introduction:**Objective, scope and outcome of the course.	01
2	**Introduction:** Objective, scope and outcome of the course. Compiler, Translator, Interpreter definition, Phase of compiler, Bootstrapping, Review of Finite automata lexical analyzer, Input, Recognition of tokens, Idea about LEX: A lexical analyzer generator, Error handling.	06
3	**Review of CFG Ambiguity of grammars:** Introduction to parsing. Top down parsing, LL grammars & passers error handling of LL parser, Recursive descent parsing predictive parsers, Bottom up parsing, Shift reduce parsing, LR parsers, Construction of SLR, Conical LR & LALR parsing tables, parsing with ambiguous grammar. Operator precedence parsing, Introduction of automatic parser generator: YACC error handling in LR parsers.	10
4	**Syntax directed definitions;** Construction of syntax trees, S-Attributed Definition, L-attributed definitions, Top down translation. Intermediate code forms using postfix notation, DAG, Three address code, TAC for various control structures, Representing TAC using triples and quadruples, Boolean expression and control structures.	10
5	**Storage organization;** Storage allocation, Strategies, Activation records, Accessing local and non-local names in a block structured language, Parameters passing, Symbol table organization, Data structures used in symbol tables.	08
6	**Definition of basic block control flow graphs;** DAG representation of basic block, Advantages of DAG, Sources of optimization, Loop optimization, Idea about global data flow analysis, Loop invariant computation, Peephole optimization, Issues in design of code generator, A simple code generator, Code generation from DAG.	07

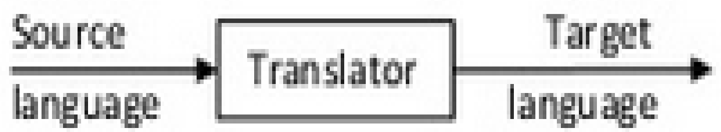
UNIT-1

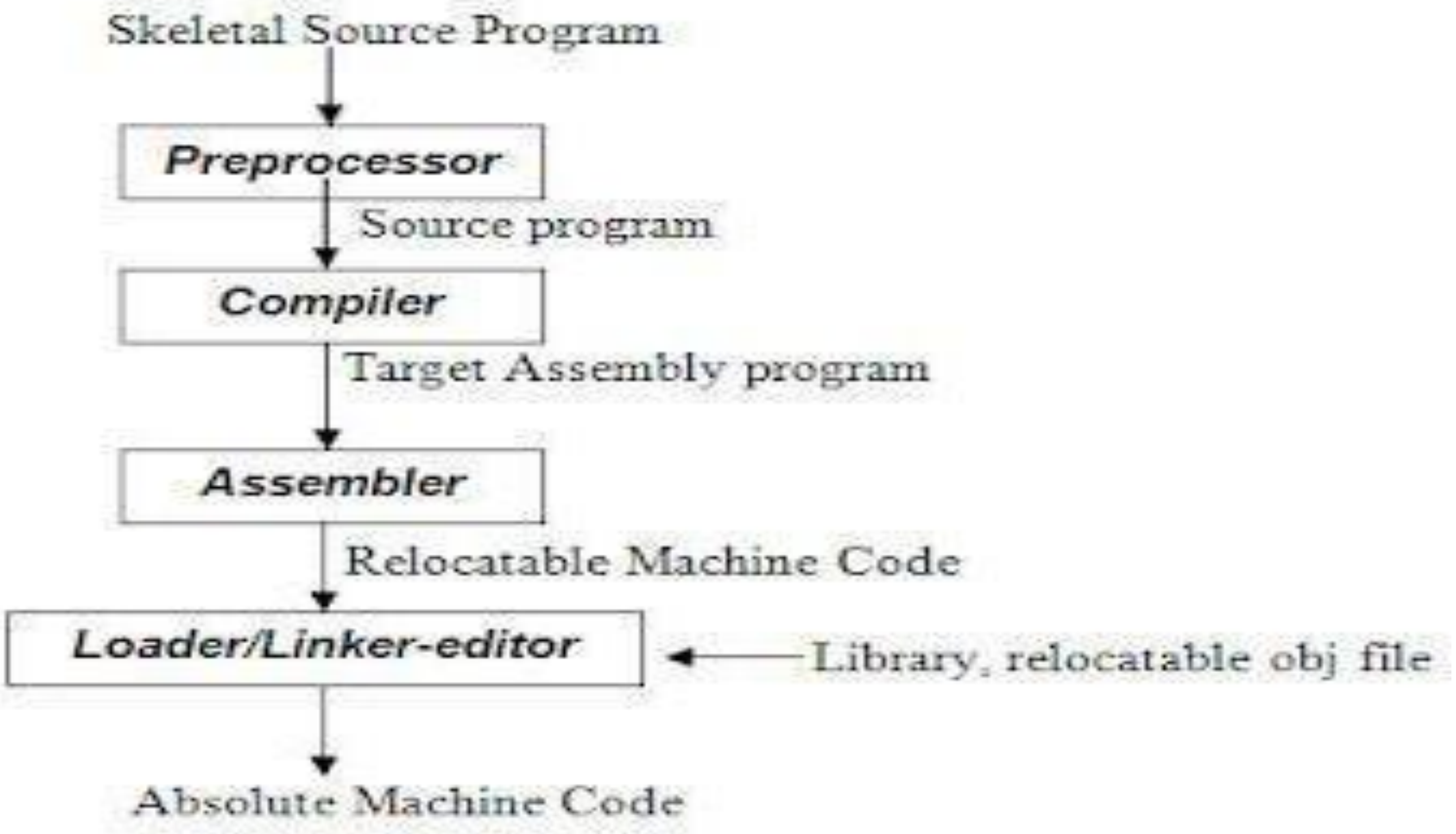
OVERVIEW OF LANGUAGE PROCESSING SYSTEM

TRANSLATOR:

- Translating the high Level language program input into an equivalent machine language program.

•Providing diagnostic messages wherever the programmer violates specification of the High level language.





Skeletal Source Program

Preprocessor

Source program

Compiler

Target Assembly program

Assembler

Relocatable Machine Code

Loader/Linker-editor

Library, relocatable obj file

Absolute Machine Code

Fig 1.1 Language –processing System

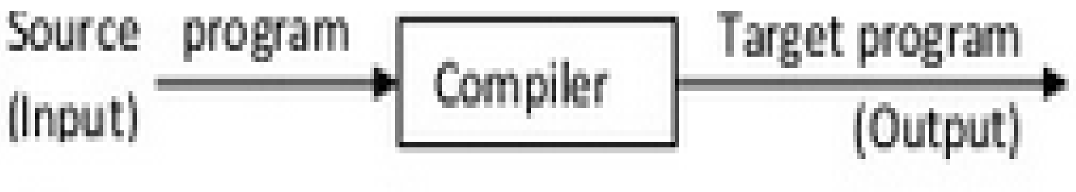
Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

- *Macro processing:* A preprocessor may allow a user to define macros that are short hands for longer constructs.
- *File inclusion:* A preprocessor may include header files into the program text.
- *Rational preprocessor:* these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
- *Language Extensions:* These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

COMPILER

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



ASSEMBLER*:*

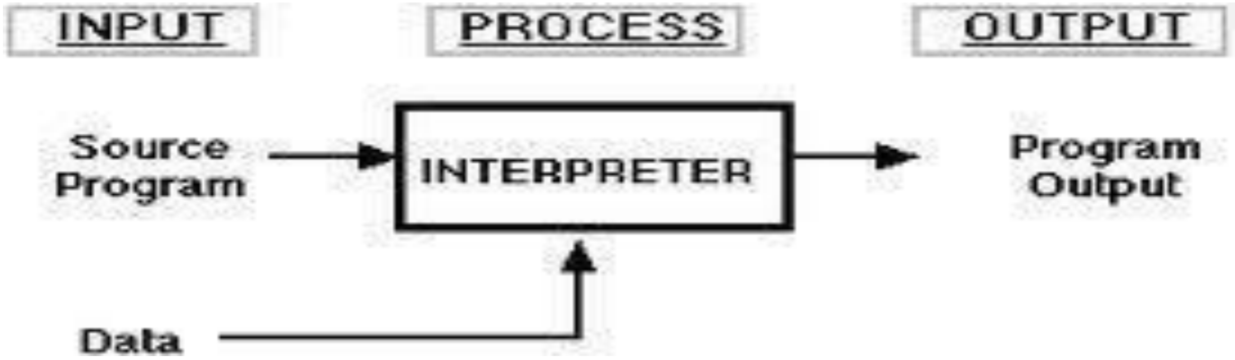
programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

Ex: MOV A,B

INTERPRETER:

An interpreter is a program that appears to execute a source program as if it were machine language.

Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter.



Advantages*:*

- Modification of user program can be easily made and implemented as execution proceeds.
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it machine independent.

Disadvantages:

- The execution of the program is *slower*.
- Memory consumption is more.

DIFFERENCE BETWEEN COMPILER AND INTERPRETER

•A compiler converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.
•Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.
•List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.
•An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.
•The compiler produce object code whereas interpreter does not produce object code. In the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. hence interpreter is less efficient than compiler.

Examples of interpreter: A *UPS Debugger* .
example of compiler: *Borland c compiler* or Turbo C compiler compiles the programs written in C or C++.

Loader and Linker

•A loader is a program that places programs into memory and prepares them for execution.

•A Linker resolves external memory address where the code in one file may refer to the code in another file.

Phases of a compiler

A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation.

Lexical Analysis:-

LA or Scanners reads the source program one character at a time, carving the source program into a sequence of atomic units called **tokens.**

Syntax Analysis:-

The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

Semantic Analyzer:

It uses syntax tree and symbol table to check whether the given program is **semantically** consistent with language definition. It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.

Intermediate Code Generations:-

An intermediate representation of the final machine language code is produced.

This phase bridges the analysis and synthesis phases of translation.

Code Optimization :-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-

The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

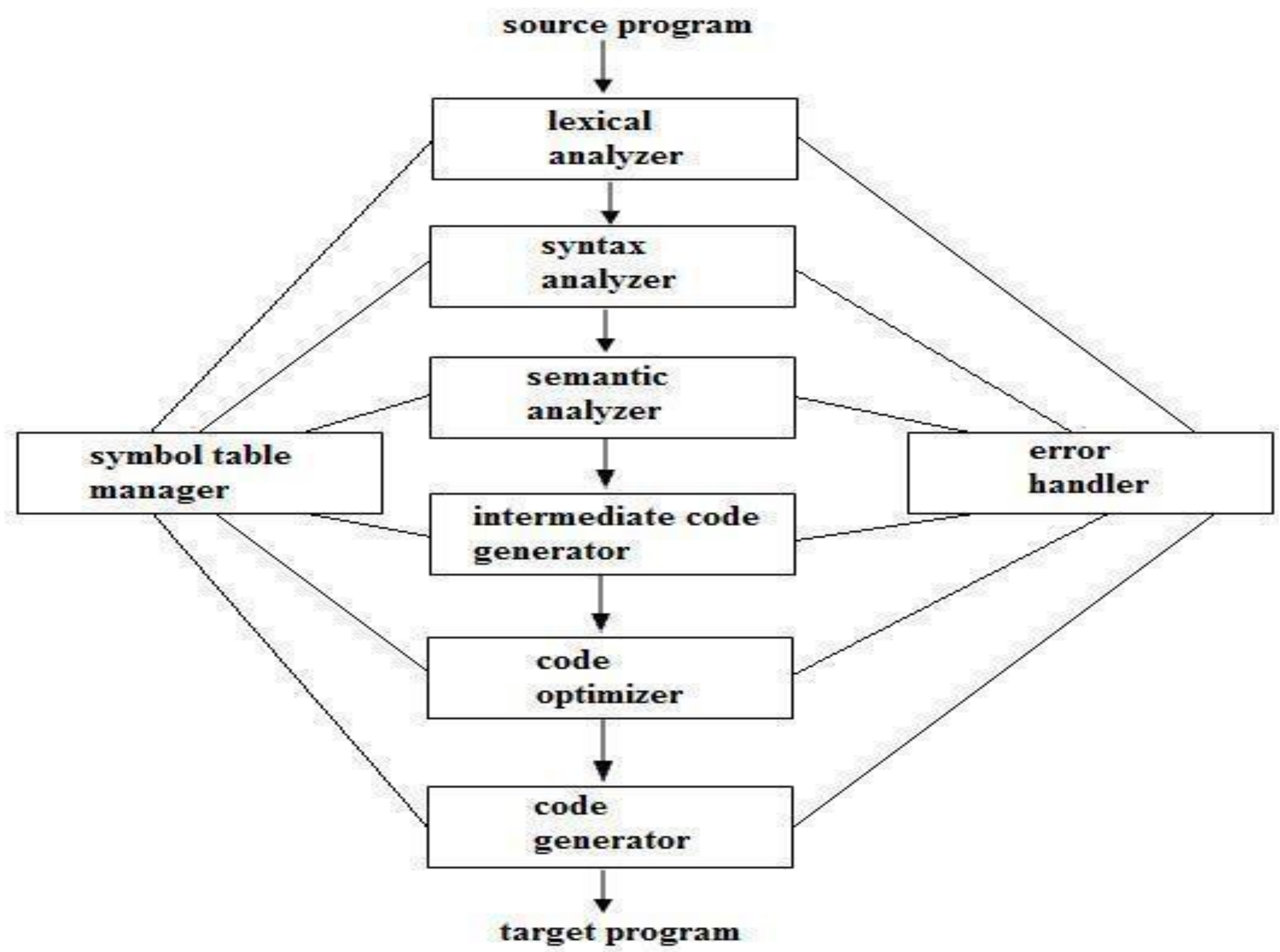
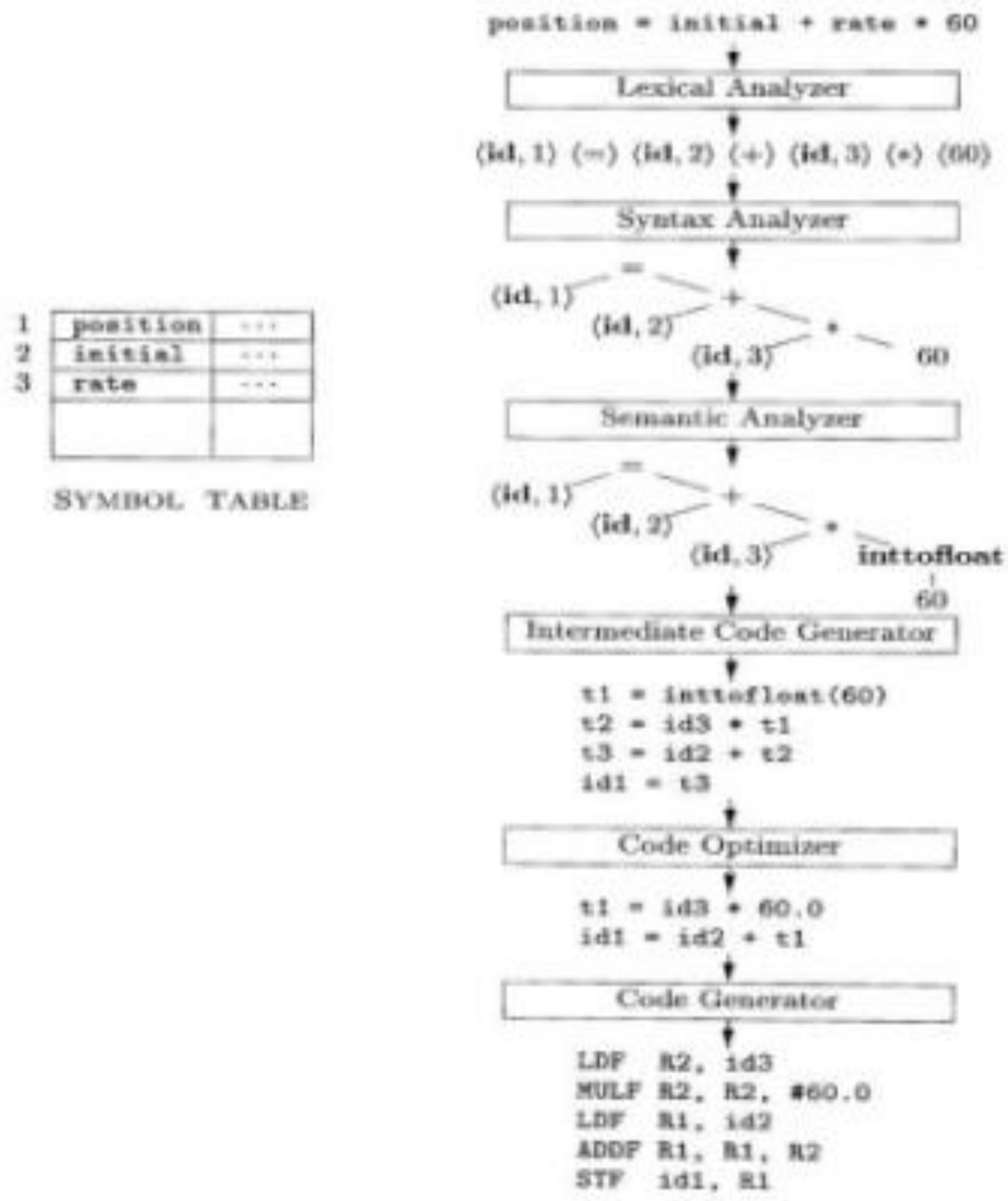


Fig 1.5 Phases of a compiler



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

SYMBOL TABLE

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

Lexical Analyzer

⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩

Syntax Analyzer

Semantic Analyzer

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

Lexical Analysis

- Lexical analysis is the first phase of compiler which is also termed as scanning.
- Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.

.

Lexical Analysis

- Lexical analysis is the first phase of compiler which is also termed as scanning.
- Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.

Token: Token is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc.

- **Lexeme:** Lexeme is instance of a token i.e., group of characters forming a token. ,
- **Pattern:** Pattern describes the rule that the lexemes of a token takes. It is the structure that must be matched by strings.

.

Syntax Analysis

• Syntax analysis is the second phase of compiler which is also called as parsing.

• Parser converts the tokens produced by lexical analyzer into a tree like representation called parse tree.

• A parse tree describes the syntactic structure of the input.

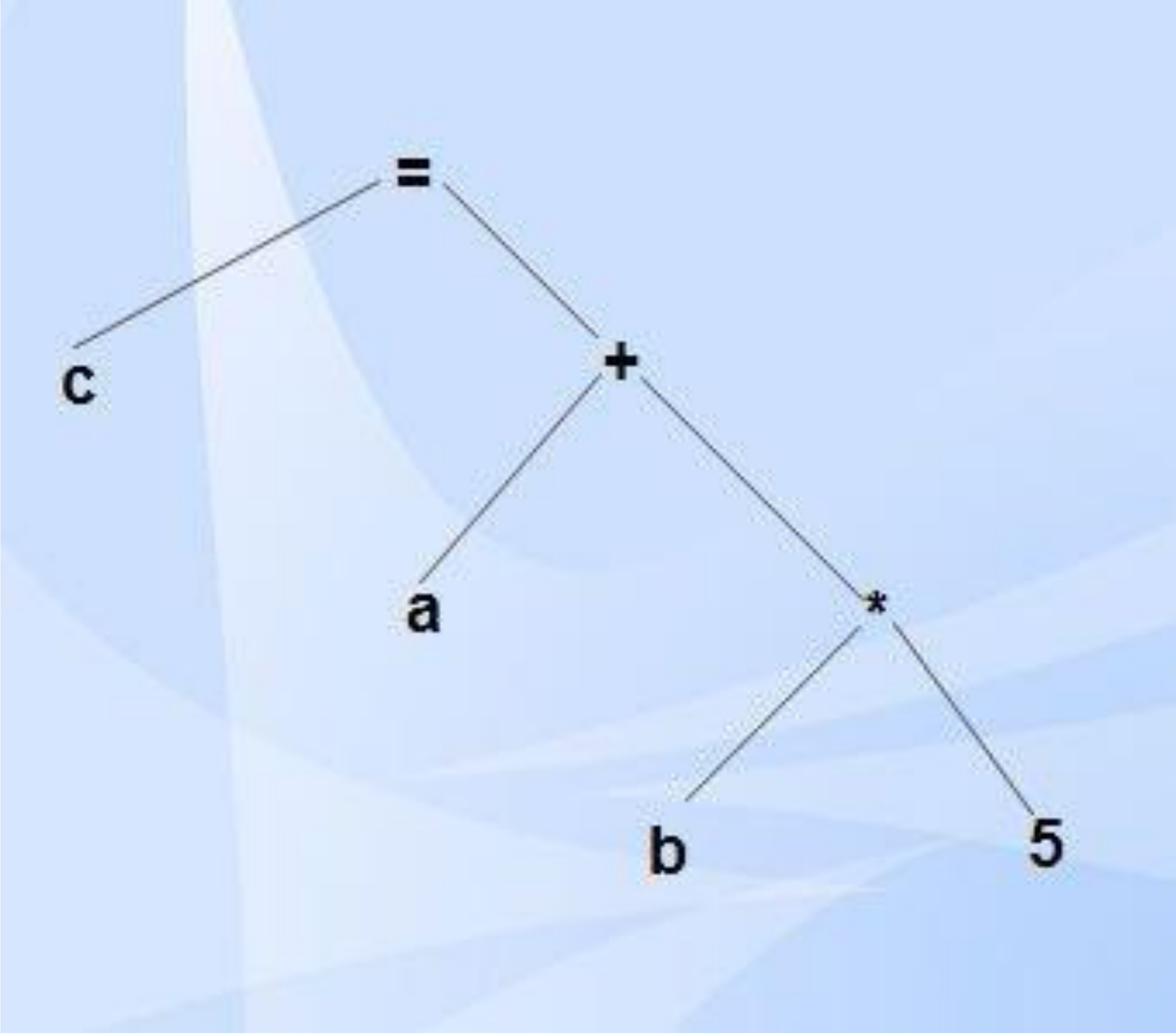
• Syntax tree is a compressed representation of the parse tree in which the operators appear as interior nodes and the operands of the operator are the children of the node for that operator.

Input: *Tokens*

Output: *Syntax tree*

Syntax Analysis

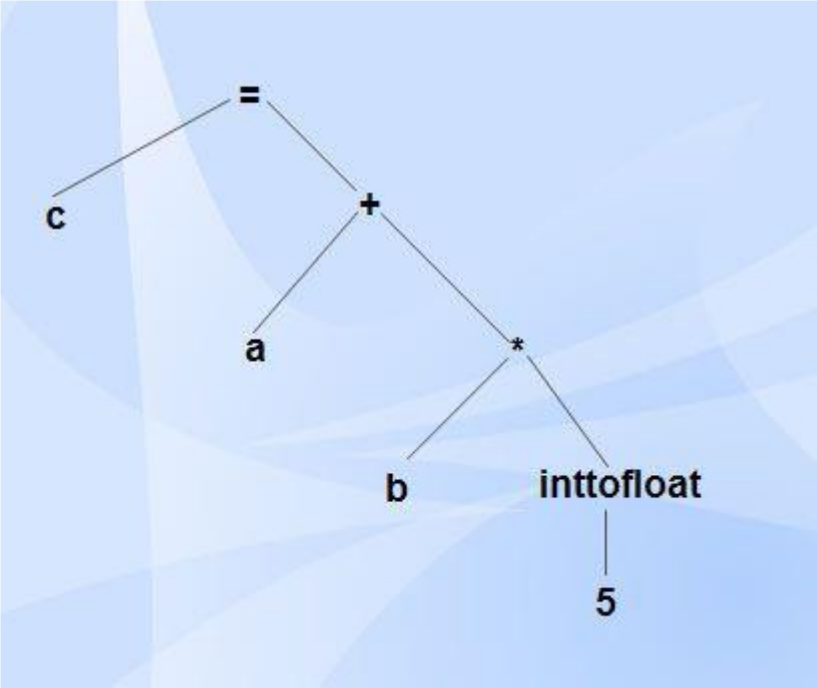
.



Semantic Analysis

- Semantic analysis is the third phase of compiler.
- It checks for the semantic consistency.
- Type information is gathered and stored in symbol table or in syntax tree.
- Performs type checking.

.



Intermediate Code Generation

- Intermediate code generation produces intermediate representations for the source program which are of the following forms:
  - o Postfix notation
  - o Three address code
  - o Syntax tree

Most commonly used form is the three address code.

t_1 = inttofloat (5)
t_2 = id_3* tl
t_3 = id_2 + t_2
id_1 = t_3

Properties of intermediate code

- It should be easy to produce.
- It should be easy to translate into target program.

.

Code Optimization

- Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output.
- It results in faster running machine code.
- It can be done by reducing the number of lines of code for a program.
- This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves
  - o Deduction and removal of dead code (unreachable code).
  - o Calculation of constants in expressions and terms.
  - o Collapsing of repeated expression into temporary string.
  - o Loop unrolling.
  - o Moving code outside the loop.
  - o Removal of unwanted temporary variables.

$$\mathbf{t_1 = id_3 * 5.0}$$
$$\mathbf{id_1 = id_2 + t_1}$$

Code Generation

- Code generation is the final phase of a compiler.
- It gets input from code optimization phase and produces the target code or object code as result.
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
  - o Allocation of register and memory.
  - o Generation of correct references.
  - o Generation of correct data types.
  - o Generation of missing code.

LDF R_2, id_3

MULF R_2, # 5.0

LDF R_1, id_2

ADDF R_1, R_2

STF id_1, R_1

.

Symbol Table Management

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows finding the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

.

Bootstrapping

Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.

Writing a compiler for any high level language is a complicated process. It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages. to clearly understand the **Bootstrapping** technique consider a following scenario.

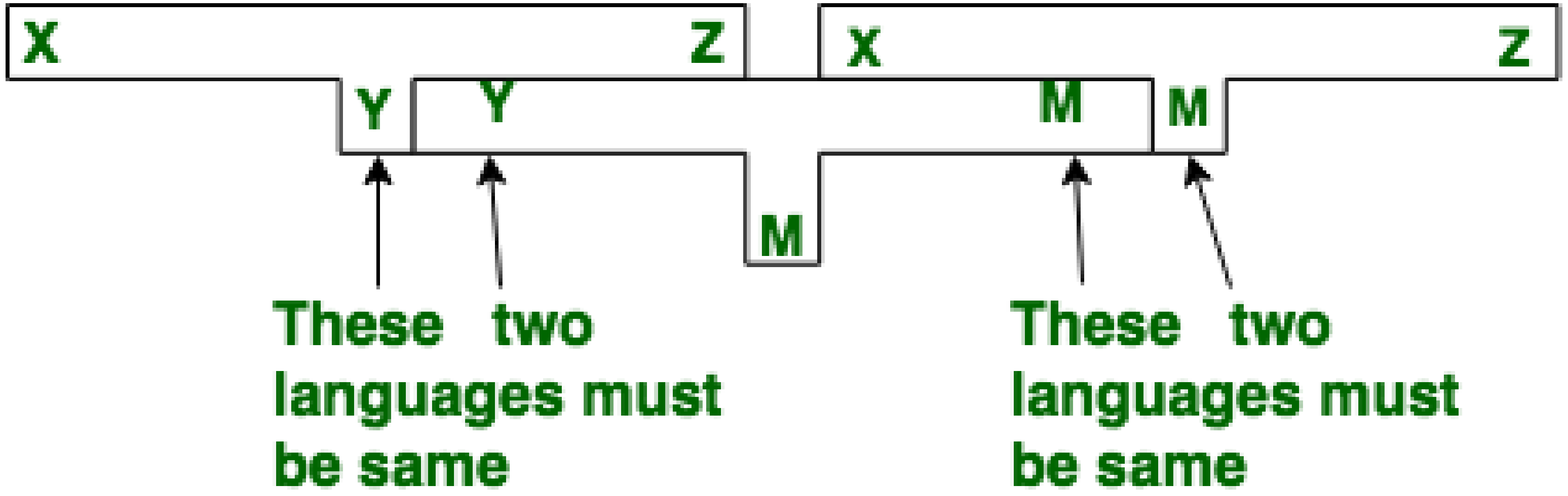
Bootstrapping

Suppose we want to write a cross compiler for new language X. The implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create XYZ. Now if existing compiler Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ. That means a compiler for source language X that generates a target code in language Z and which runs on machine M.

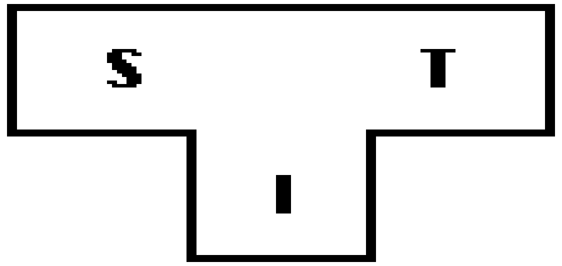
Following diagram illustrates the above scenario.

Example:

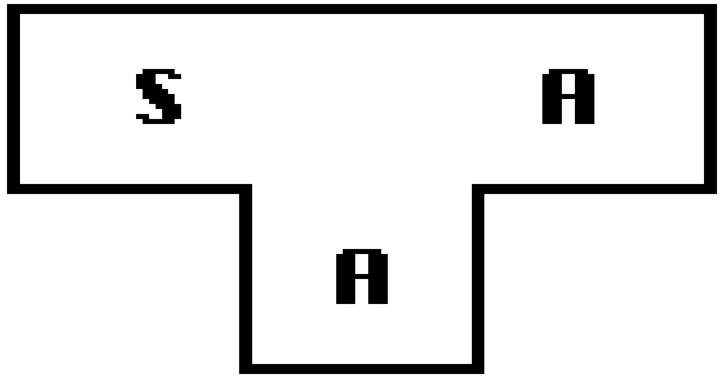
We can create compiler of many different forms. Now we will generate.


X
Z
Y
Y
X
Z
M
M
M
These two languages must be same
These two languages must be same

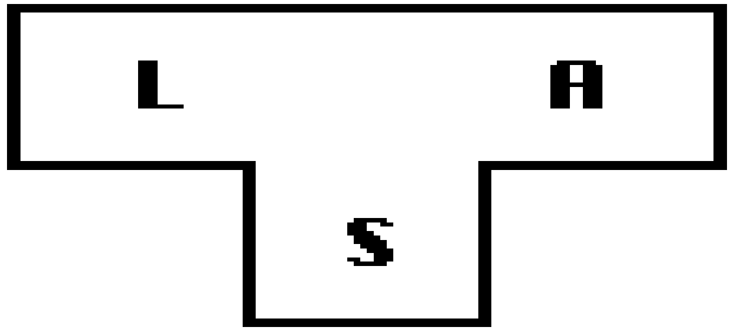
- The T- diagram shows a compiler ${}^{S}C_{I}^{T}$ for Source S, Target T, implemented in I.



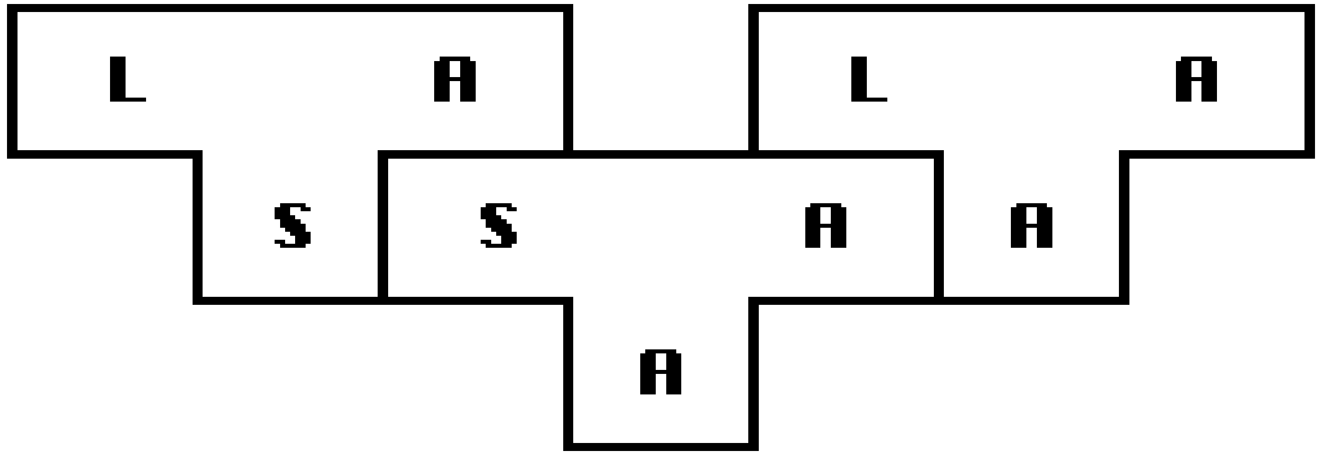
- Follow some steps to produce a new language for machine **A**
- Create a compiler ${}^{S}C_{A}^{A}$ for subset, S of the desired language, L using language "A" and that compiler runs on machine A.



- Create a compiler ${}^{L}C_{S}^{A}$ for language L written in a subset of L.



- Compile ${}^{L}C_{S}^{A}$ using the compiler ${}^{S}C_{A}^{A}$ to obtain ${}^{L}C_{A}^{A}$. ${}^{L}C_{A}^{A}$ is a compiler for language L, which runs on machine A and produces code for machine A



Review of Finite Automata

•Finite Automata(FA) is the simplest machine to recognize patterns.

•Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly.

•Finite automata is a recognizer for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted.

The mathematical model of finite automata consists of:

Finite set of states (Q)
Finite set of input symbols (Σ)
One Start state (q0)
Set of final states (qf)
Transition function (δ)

Finite Automata Construction

Let L(r) be a regular language recognized by some finite automata (FA)

- **States** : States of FA are represented by circles. State names are written inside circles.
- **Start state** : The state from where the automata starts, is known as the start state. Start state has an arrow pointed towards it.
- **Intermediate states** : All intermediate states have at least two arrows one pointing to and another pointing out from them.
- **Final state** : If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows are one greater than even, i.e. **odd = even+1**.
- **Transition** : The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows points to the destination state. If automata stays on the same state, an arrow pointing from a state to itself is drawn.

Input Recognition of Tokens

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

The patterns for the tokens are described using regular definitions.

digit -->[0,9]

letter-->[A-Z,a-z]

id -->letter(letter/digit)*

if --> if

then -->then

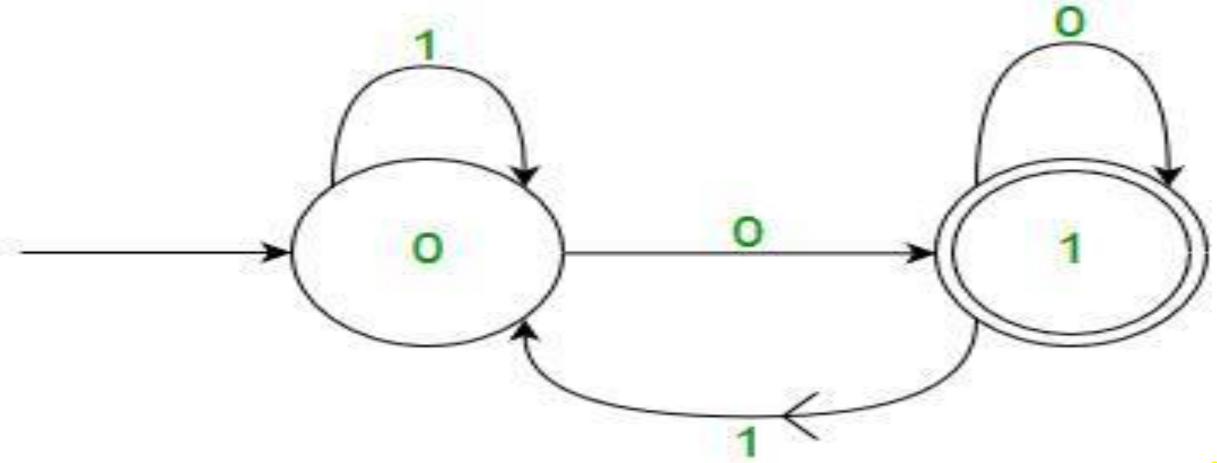
else -->else

relop--> </>/<=/>=/==/< >

1) Deterministic Finite Automata (DFA)

DFA consists of 5 tuples {Q, Σ, q, F, δ}.
Q : set of all states.
Σ : set of input symbols. (Symbols which machine takes as input)
q : Initial state. (Starting state of a machine)
F : set of final state.
δ : Transition Function, defined as δ : Q X Σ --> Q.

In a DFA, for a particular input character, the machine goes to one state only. A transition function is defined on every state for every input symbol. Also in DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character.
For example, below DFA with Σ = {0, 1} accepts all strings ending with 0.



2) Nondeterministic Finite Automata(NFA)

NFA is similar to DFA except following additional features:

1. Null (or ε) move is allowed i.e., it can move forward without reading symbols.
2. Ability to transmit to any number of states for a particular input.

However, these above features don't add any power to NFA. If we compare both in terms of power, both are equivalent.

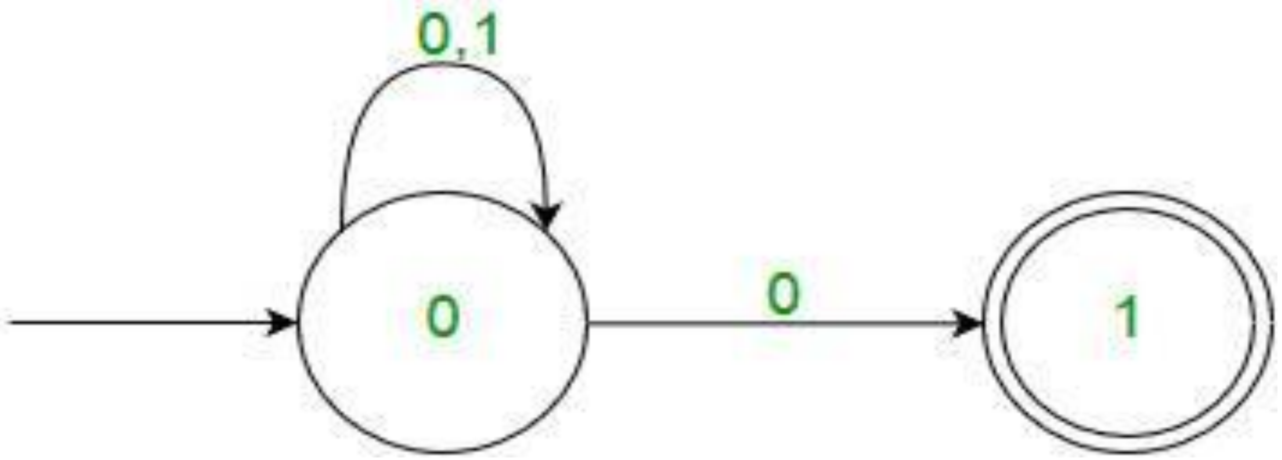
Due to above additional features, NFA has a different transition function, rest is same as DFA.

δ: Transition Function

δ: Q X (Σ U ε) --> 2 ^ Q.

As you can see in transition function is for any input including null (or ε), NFA can go to any state number of states.

For example, below is a NFA for above problem



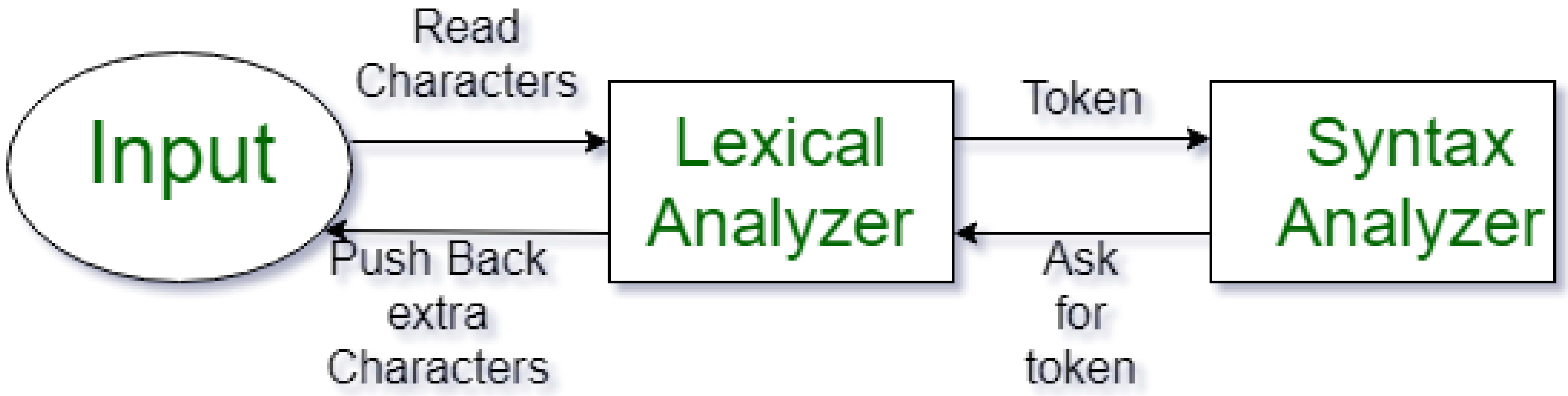
Lexical Analysis & its Role

Introduction of Lexical Analysis:

- Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens.

- Lexical Analysis can be implemented with the Deterministic finite Automata.

The output is a sequence of tokens that is sent to the parser for syntax analysis



What is a token?

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

Type token (id, number, real, . . .)

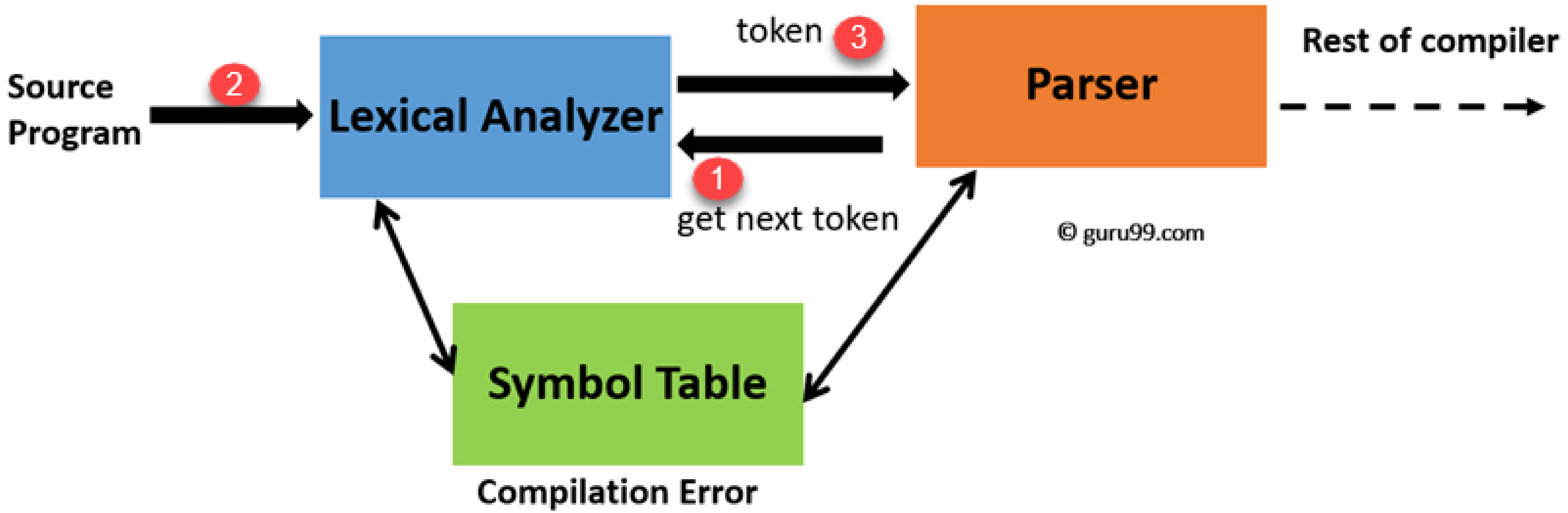
Punctuation tokens (IF, void, return, . . .)

Alphabetic tokens (keywords)

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

Role of the Lexical Analyzer

- Remove comments and white spaces
- 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


Source Program
2
Lexical Analyzer
token
3
Parser
Rest of compiler
1
get next token
© guru99.com
Symbol Table
Compilation Error

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.

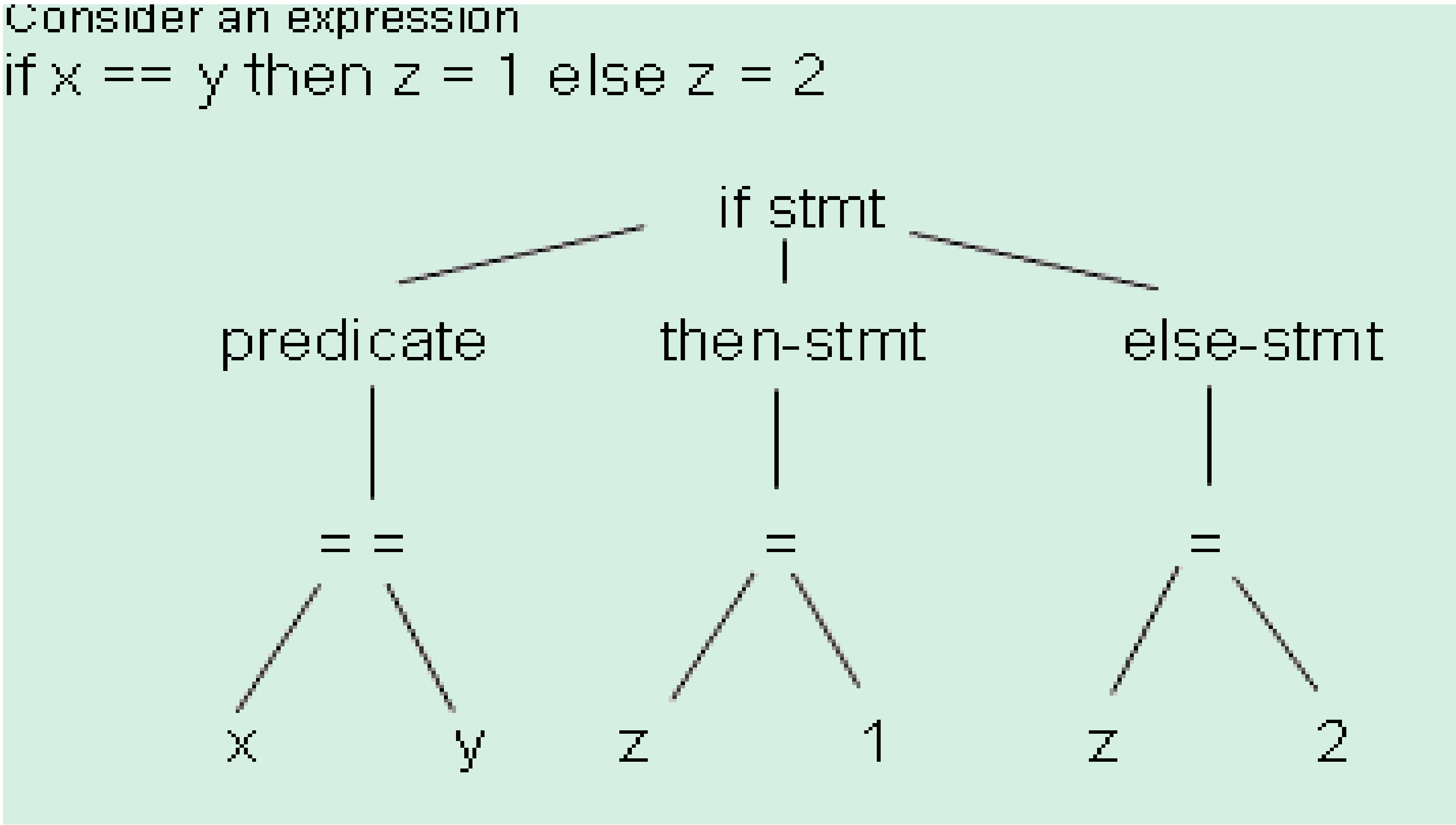
1. "Get next token" is a command which is sent from the parser to the lexical analyzer.
2. On receiving this command, the lexical analyzer scans the input until it finds the next token.
3. It returns the token to Parser.

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.

LECTURE CONTENTS WITH A BLEND OF NPTEL CONTENTS

Parsing

Just like a natural language, a programming language also has a set of grammatical rules and hence can be broken down into a parse tree by the parser. It is on this parse tree that the further steps of semantic analysis are carried out. This is also used during generation of the intermediate language code. Yacc (yet another compiler compiler) is a program that generates parsers in the C programming language.



Consider an expression

if x == y then z = 1 else z = 2

REFERENCES/BIBLIOGRAPHY

1 https://nptel.ac.in/courses/106/104/106104072/

2 https://www.slideshare.net/appasami/cs6660-compiler-design-notes

3 http://www.brainkart.com/article/Recognition-of-Tokens_8138/

Thank you!