JECRC Foundation

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

SUBJECT- COMPILER DESIGN

SEMESTER- 5TH SEM

VISSION 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.

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	2	1	1	1	1	2	1	3
			L	2. Examine syntax and semantic analyzer and illustrate storage allocation and its organization	3	3	3	3	1	1	1	1	1	2	2	3
			L	3. Analyze symbol table organization, code optimization and code generator	3	3	3	3	2	1	1	1	1	2	2	3
			L	4.Compare and evaluate various compilers and analyzers	3	3	3	3	2	1	1	1	1	2	1	3

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

PSO

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

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

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

LECTURE PLAN:

Subject: Compiler Design (5CS4 – 02) **Year/Sem: III/V**

Unit No./ Total lec. Req.	Topics	Lect. Req.
Unit-1 (6)	Compiler, Translator, Interpreter definition, Phase of compiler	1
	Introduction to one pass & Multipass compilers, Bootstrapping	1
	Review of Finite automata lexical analyzer, Input, buffering,	2
	Recognition of tokens, Idea about LEX:, GATE Questions	1
	A lexical analyzer generator, Error Handling, Unit Test	1
Unit-2 (17)	Review of CFG Ambiguity of grammars, Introduction to parsing	2
	Bottom up parsing Top down Parsing Technique	5
	Shift reduce parsing, Operator Precedence Parsing	2
	Recursive descent parsing predictive parsers	1
	LL grammars & passers error handling of LL parser	1
	Conical LR & LALR parsing tables	3
	parsing with ambiguous grammar, GATE Questions	2
	Introduction of automatic parser generator: YACC error handling in LR parsers, Unit Test	1
Unit 3- (7)	Syntax directed definitions; Construction of syntax trees	1
	L-attributed definitions, Top down translation	1
	Specification of a type checker, GATE Questions	1
	Intermediate code forms using postfix notation and three address code,	2
	Representing TAC using triples and quadruples, Translation of assignment statement.	1
	Boolean expression and control structures, Unit Test	1
Unit 4- (4)	Storage organization, Storage allocation, Strategies, Activation records,	1
	Accessing local and non local names in a block structured language	1
	Parameters passing, Symbol table organization, GATE Questions	1
	Data structures used in symbol tables, Unit Test	1
Unit 5- (6)	Definition of basic block control flow graphs,	1
	DAG representation of basic block, Advantages of DAG,	1
	Sources of optimization, Loop optimization Idea about global data flow analysis, Loop invariant computation, Loop invariant computation, Tutorial	2
	Peephole optimization, GATE Questions, Tutorial	1
	Issues in design of code generator, A simple code generator, Code generation from DAG., UNIT TEST, Revision	1

JECRC Foundation

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

Year & Sem – 3rd Year & 5th Sem

Subject – COMPILER DESIGN

Unit – 1

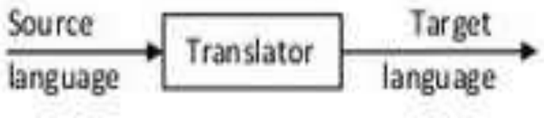
UNIT-1

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.



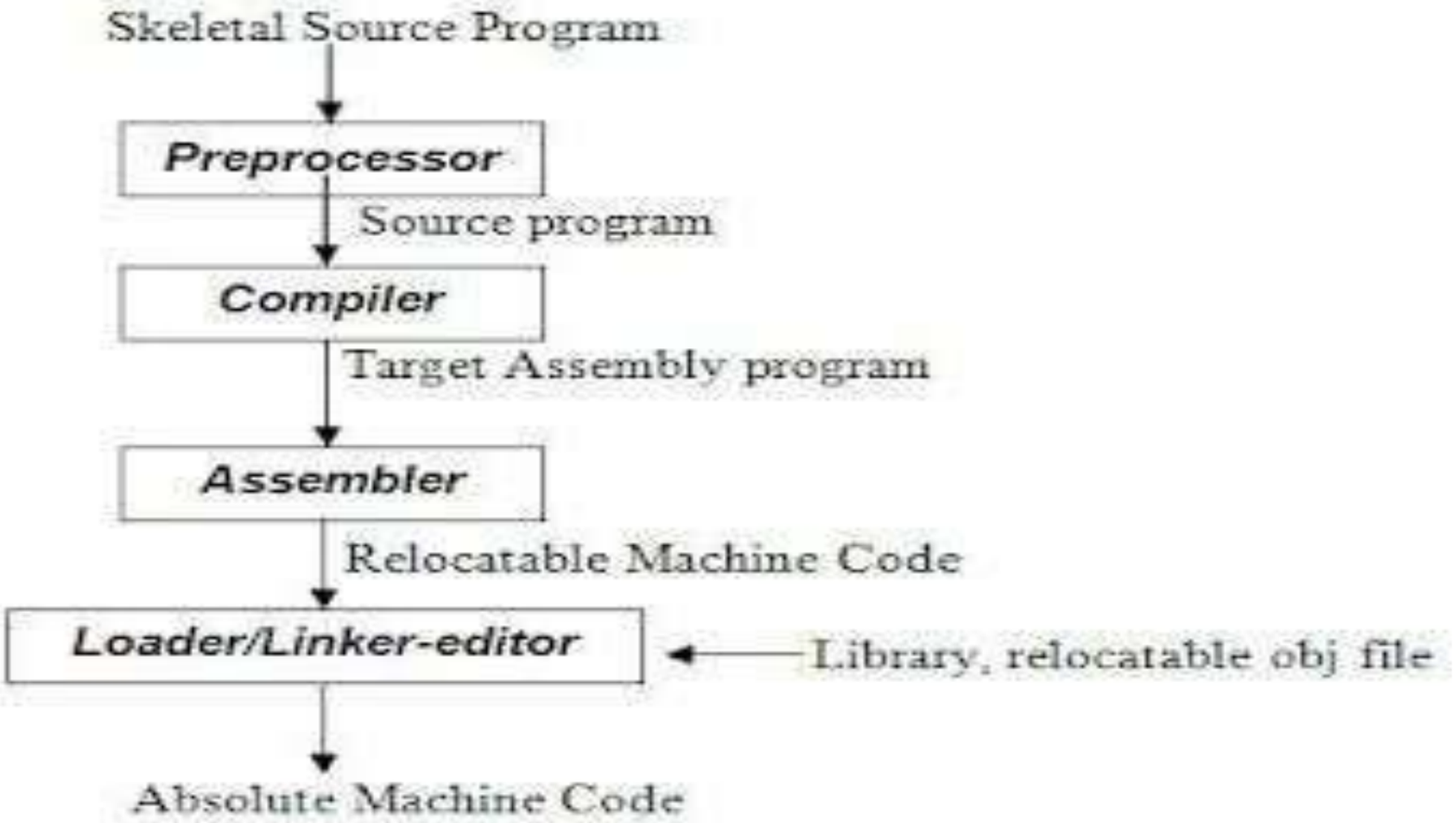


Fig 1.1 Language –processing System

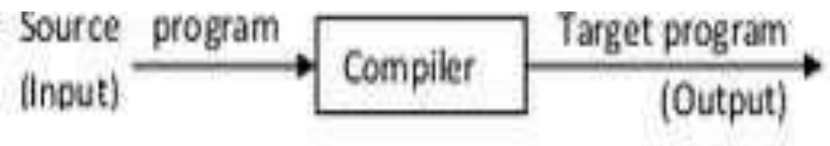
Preprocessor

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*:*

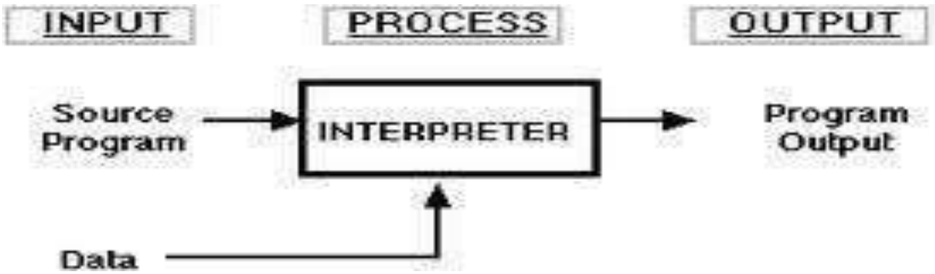
programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler 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.

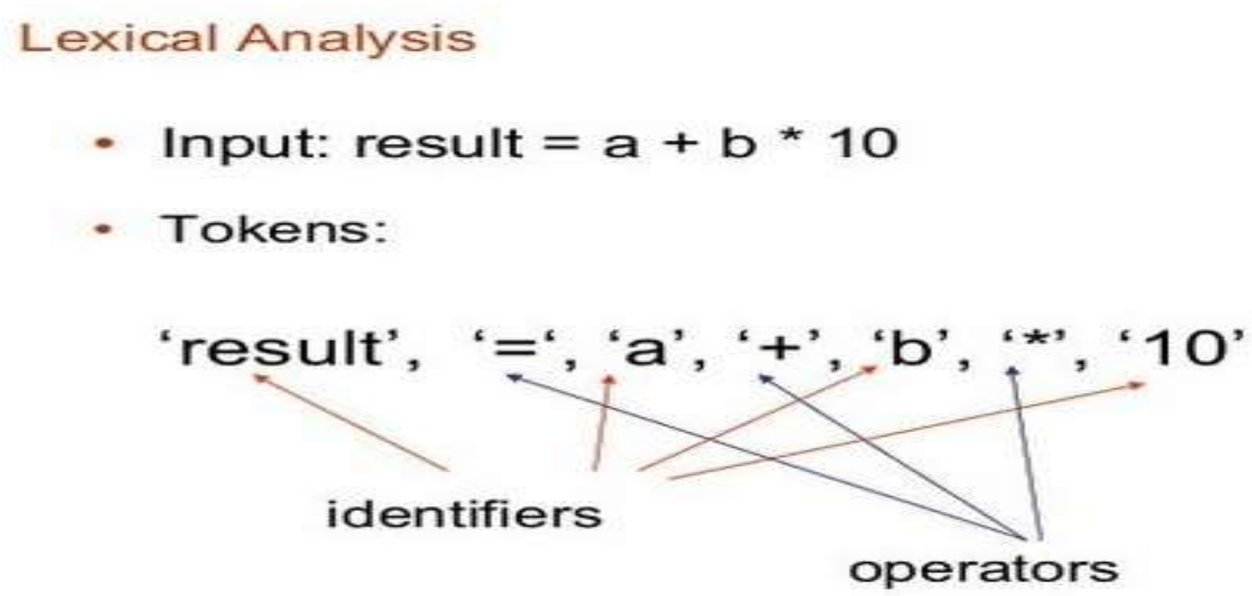
- A Linker resolves external memory address where the code in one file may refer to the code in another file.

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.**



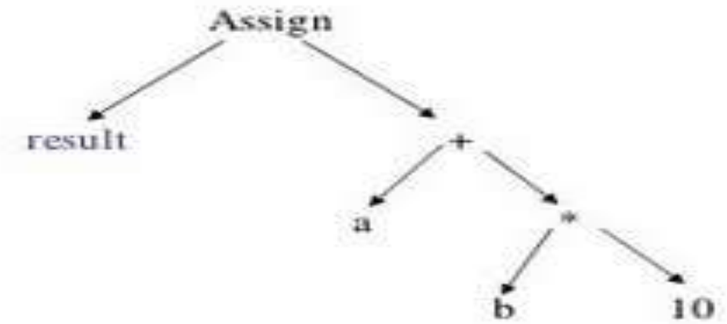
Syntax Analysis:-

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.

- Uncover the structure of a sentence in the program from a stream of tokens.
- For instance, the phrase “x = +y", which is recognized as four tokens, representing “x", “=“ and “+" and “y", has the structure **=(x,+(y))**, i.e., an assignment expression, that operates on “x" and the expression “+(y)".
- Build a tree called a parse tree that reflects the structure of the input sentence.

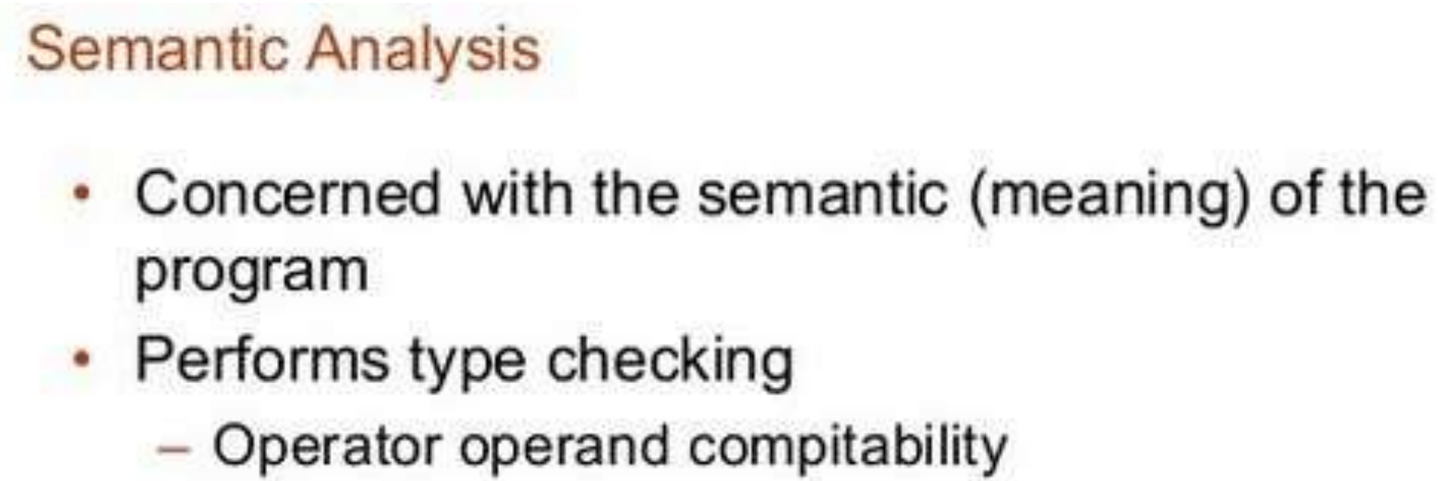
Syntax Tree

Input: result = a + b * 10



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:-

This phase bridges the analysis and synthesis phases of translation.

- Translate Tree structure into intermediate code

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.

Symbol Table

- Records the identifiers used in the source program
  - Collects various associated information as attributes
    - Variables: type, scope, storage allocation
    - Procedure: number and types of arguments method of argument passing
- It's a data structure with collection of records
  - Different fields are collected and used at different phases of compilation

Error Handler

It detects and recover the error occurred in different phases of compiler.

It also provide synchronization among different phases of compiler.

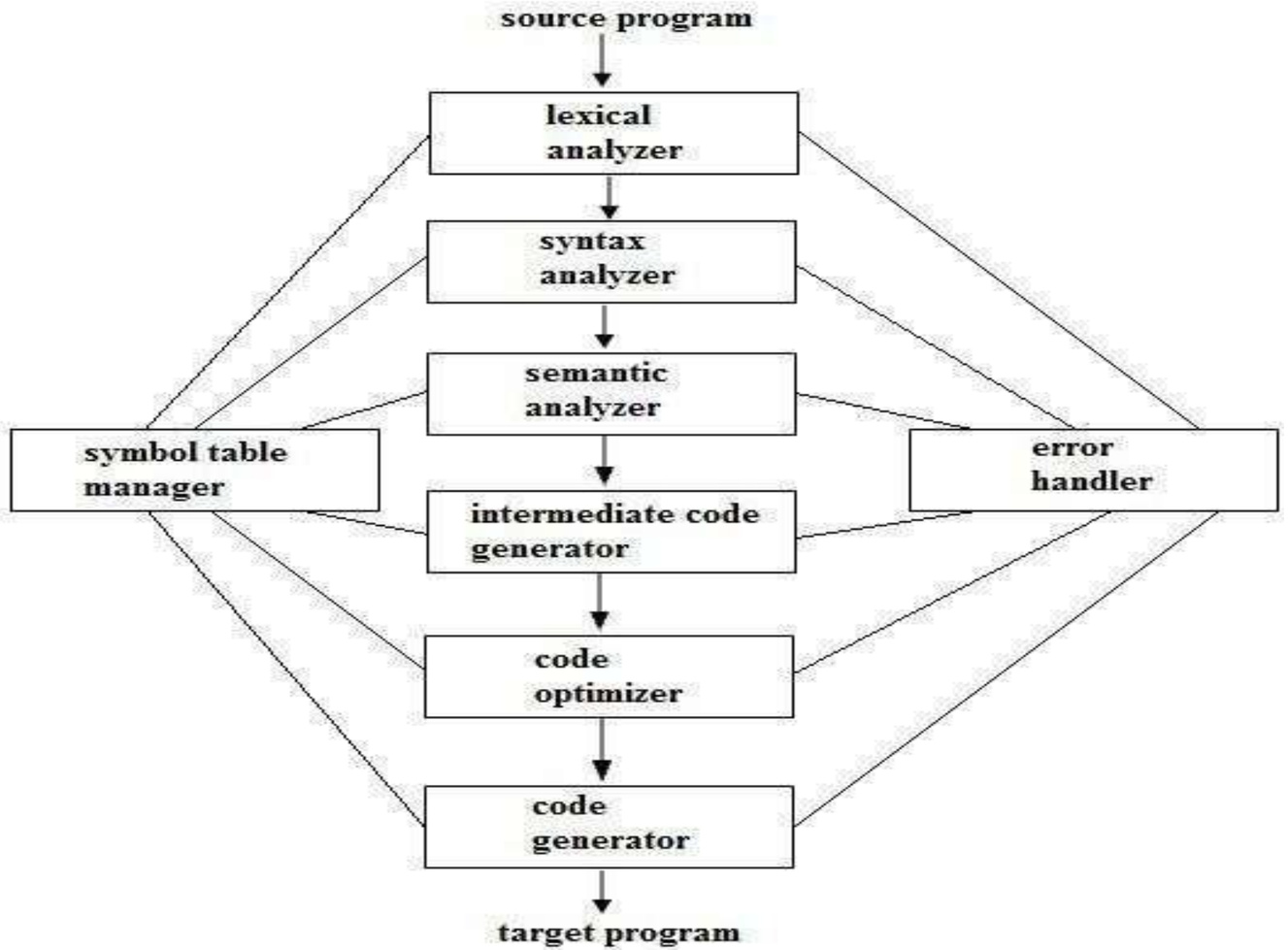
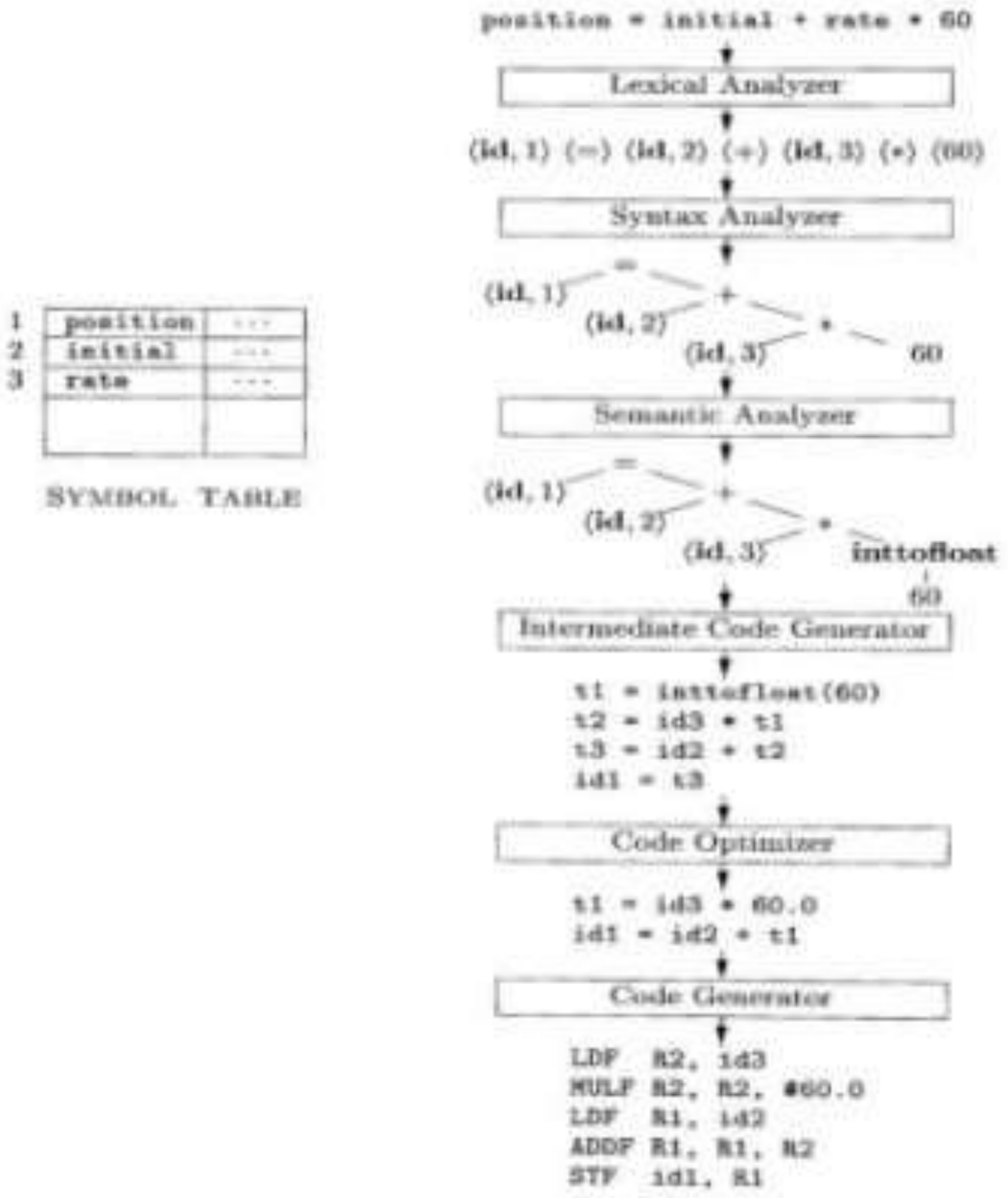


Fig 1.5 Phases of a compiler



1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE

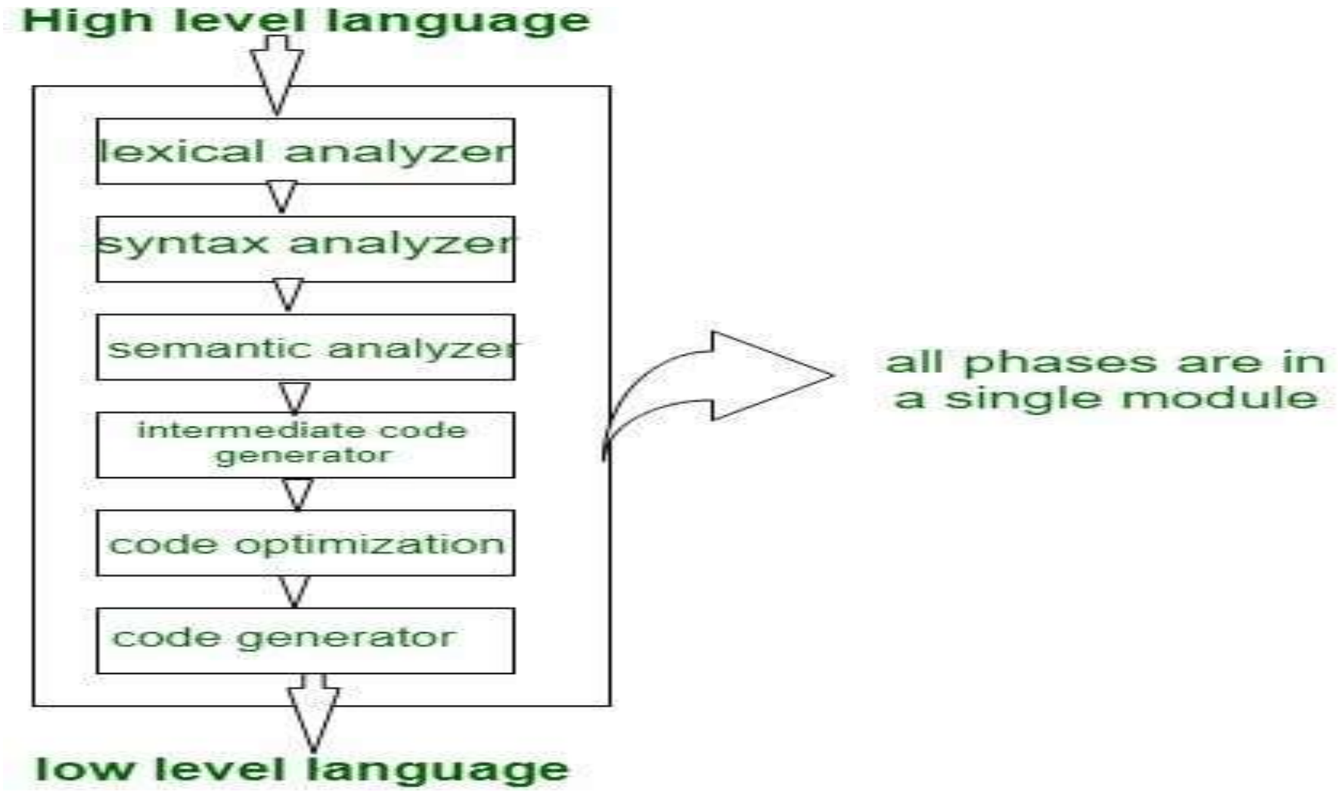
Single pass, Multi pass Compilers

- **Pass : A pass refers to the traversal of a compiler through the entire program.**

- **Phase : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.**

Single Pass Compiler

If we combine or group all the phases of compiler design in a **single** module known as single pass compiler.



Some Points about single pass compiler

1.A one pass/single pass compiler is that type of compiler that passes through the part of each compilation unit exactly once.

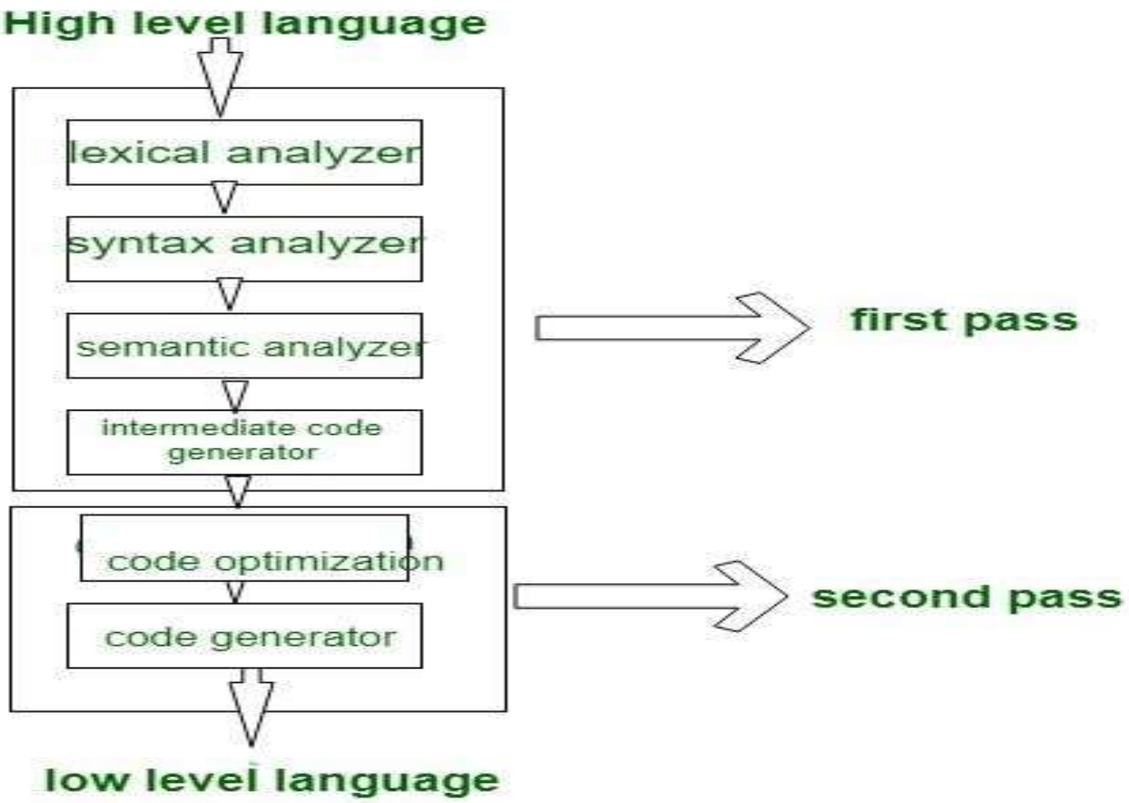
2.Single pass compiler is faster and smaller than the multi pass compiler.

3.As a disadvantage of single pass compiler is that it is less efficient in comparison with multipass compiler.

4.Single pass compiler is one that processes the input exactly once.

Two Pass compiler *or* Multi Pass compiler:

A Two pass/multi-pass Compiler is a type of compiler that processes the *source code* or abstract syntax tree of a program multiple times. In multipass Compiler we divide phases in two pass as:



First Pass: is refers as

(a) Front end
(b) Analytic part
(c) Platform independent

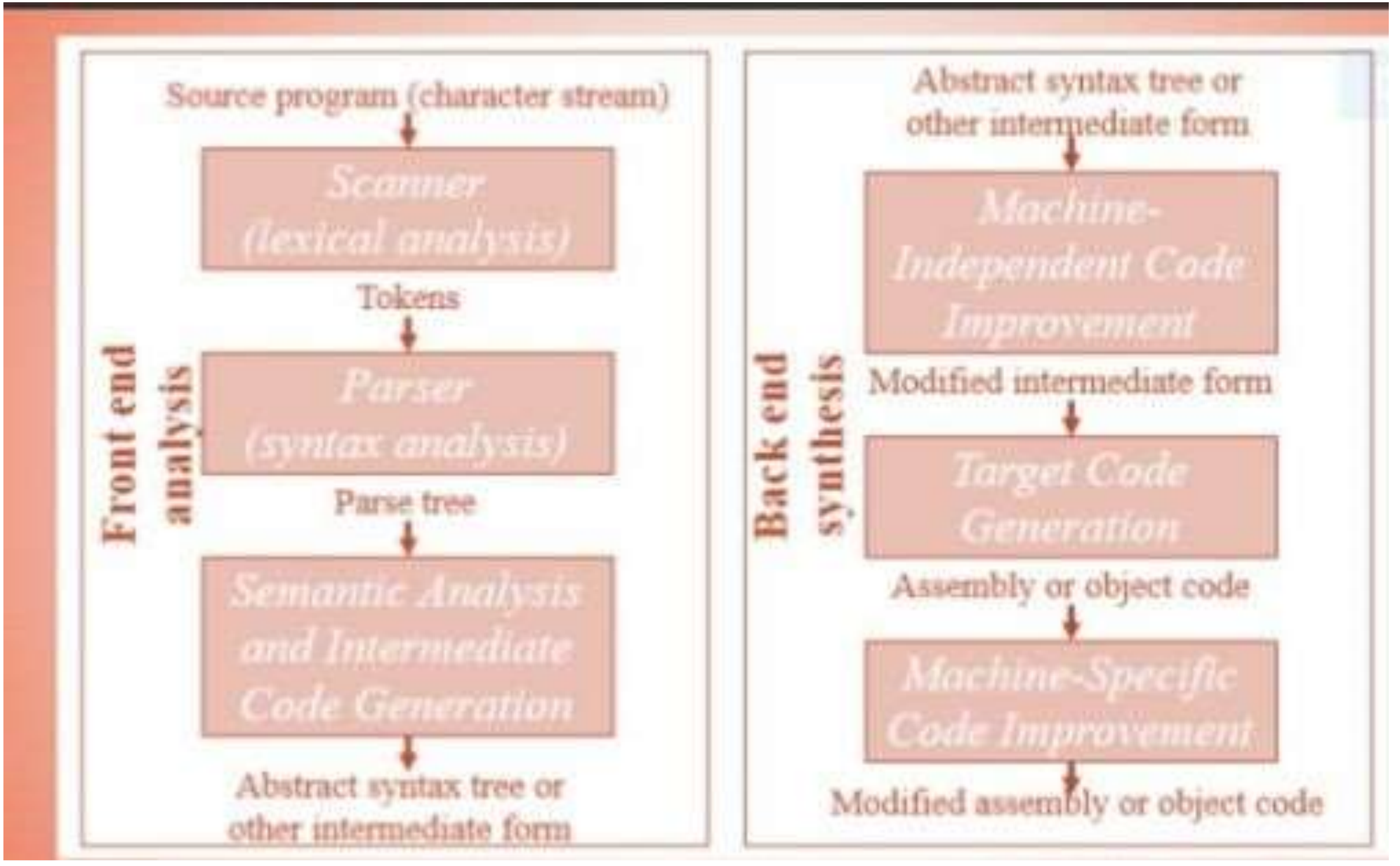
In first pass the included phases are as Lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator are work as front end and analytic part means all phases analyze the High level language and convert them **into intermediate code** and first pass is platform independent

The output of first pass have requirement of the **code optimization and code generator phase** which are comes to the second pass.

Second Pass: is refers as

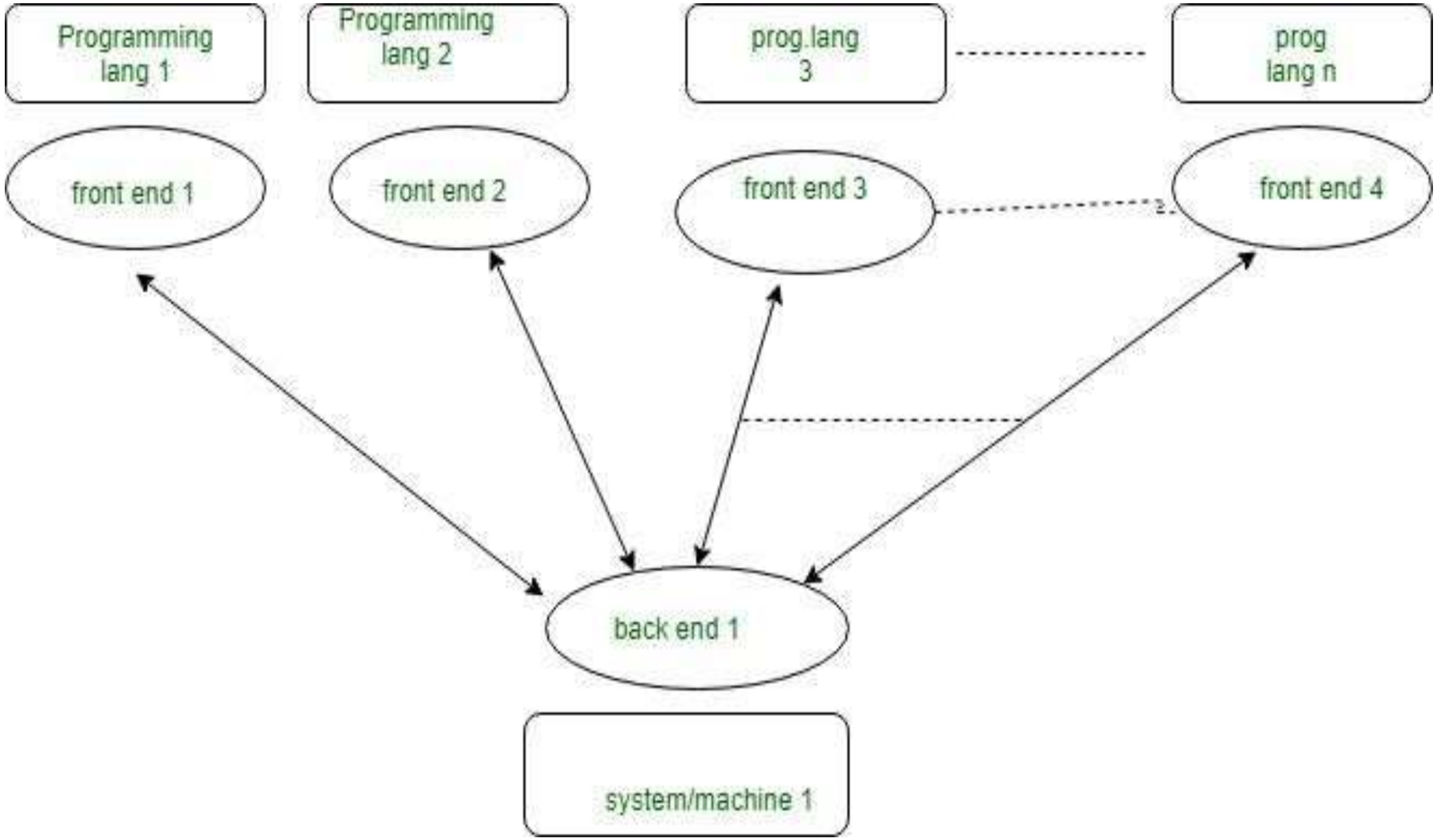
(a) Back end
(b) Synthesis Part
(c) Platform Dependent

In second Pass the included phases are as Code optimization and Code generator are work as back end and the synthesis part refers to taking input as three address code(Intermediate code) and convert them into Low level language/assembly language and second pass is platform dependent because final stage of a typical compiler converts the intermediate representation of program into an executable set of instructions which is dependent on the system.

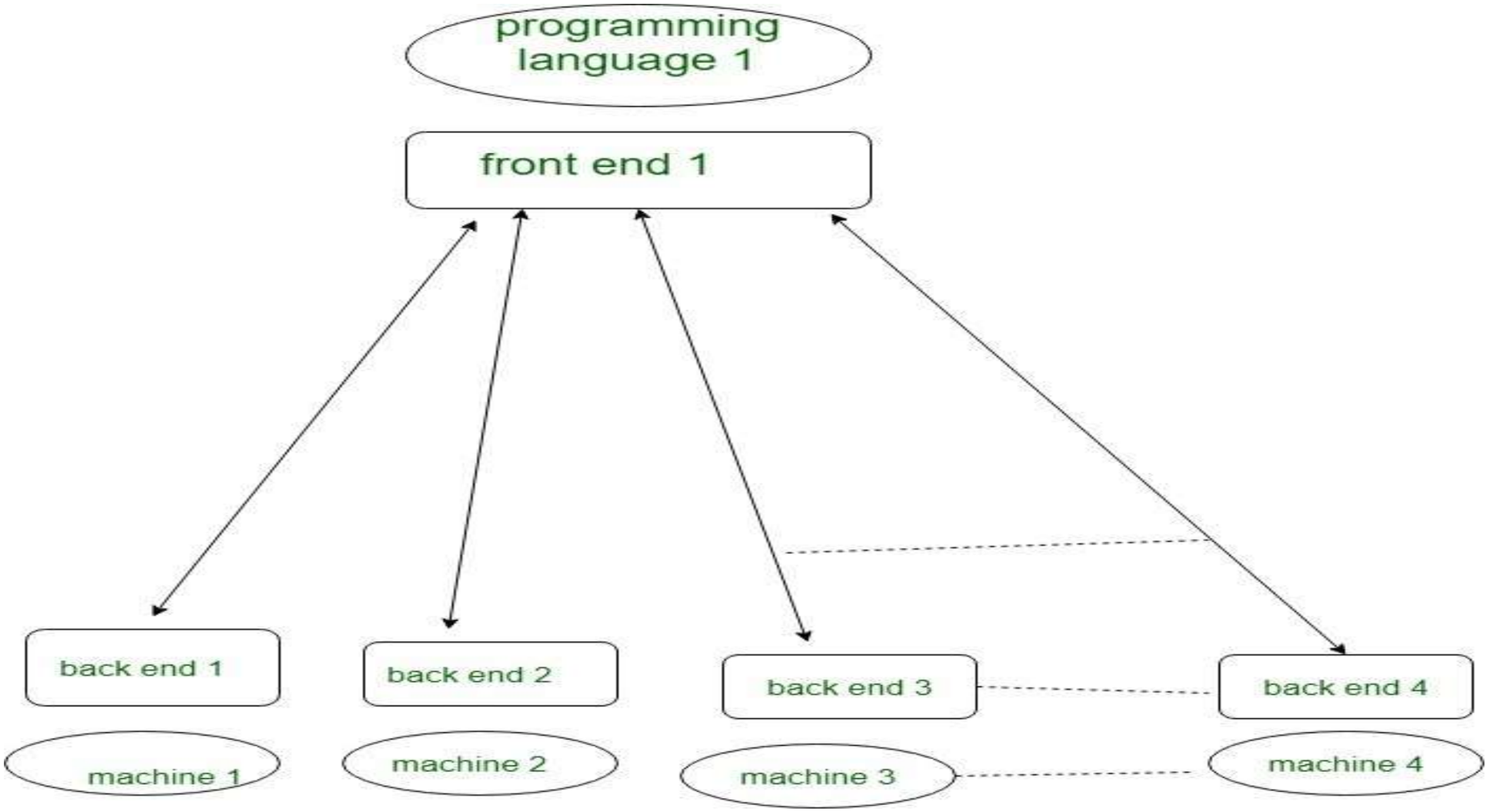

Source program (character stream)
Scanner (lexical analysis)
Tokens
Parser (syntax analysis)
Parse tree
Semantic Analysis and Intermediate Code Generation
Abstract syntax tree or other intermediate form
Front end analysis
Abstract syntax tree or other intermediate form
Machine-Independent Code Improvement
Modified intermediate form
Target Code Generation
Assembly or object code
Machine-Specific Code Improvement
Modified assembly or object code
Back end synthesis

With multi-pass Compiler we can solve these 2 basic problems:

1.If we want to design a compiler for different programming language for same machine. In this case for each programming language there is requirement of making Front end/first pass for each of them and only one Back end/second pass as shown in diagram:


Programming lang 1
Programming lang 2
prog.lang 3
prog lang n
front end 1
front end 2
front end 3
front end 4
back end 1
system/machine 1

2.If we want to design a compiler for same programming language for different machine/system. In this case we make different Back end for different Machine/system and make only one **Front end** for same programming language as:


programming language 1
front end 1
back end 1
back end 2
back end 3
back end 4
machine 1
machine 2
machine 3
machine 4

Differences between Single Pass and Multipass Compilers

PARAMETERS	SINGLE PASS	MULTI PASS
Speed	Fast	Slow
Memory	More	Less
Time	Less	More
Portability	No	Yes

Types of compiler

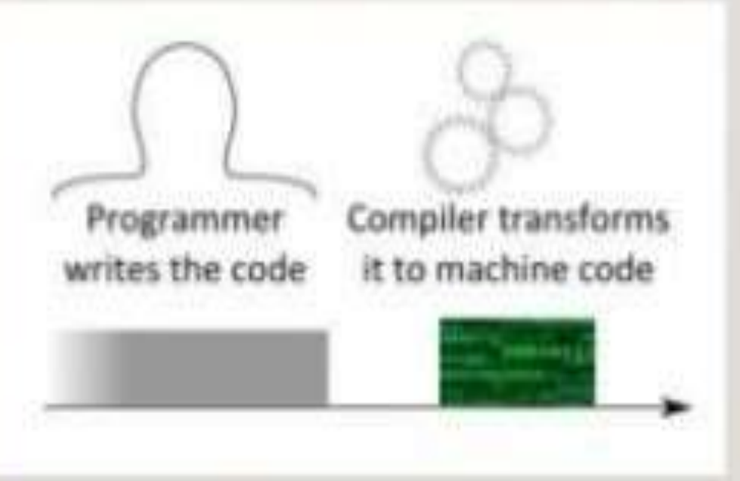
- Incremental Compiler
- Cross Compiler
- Load & Go Compiler
- Threaded Code Compiler
- Stage Compiler
- Just – in – time (JIT) Compiler
- Parallelizing Compiler

According to it's Pass structure

- One Pass Compiler
- Multi Pass Compiler

INCREMENTAL COMPILER

- Incremental compiler is a compiler which performs the recompilation of only modified source rather than compiling the whole source program.

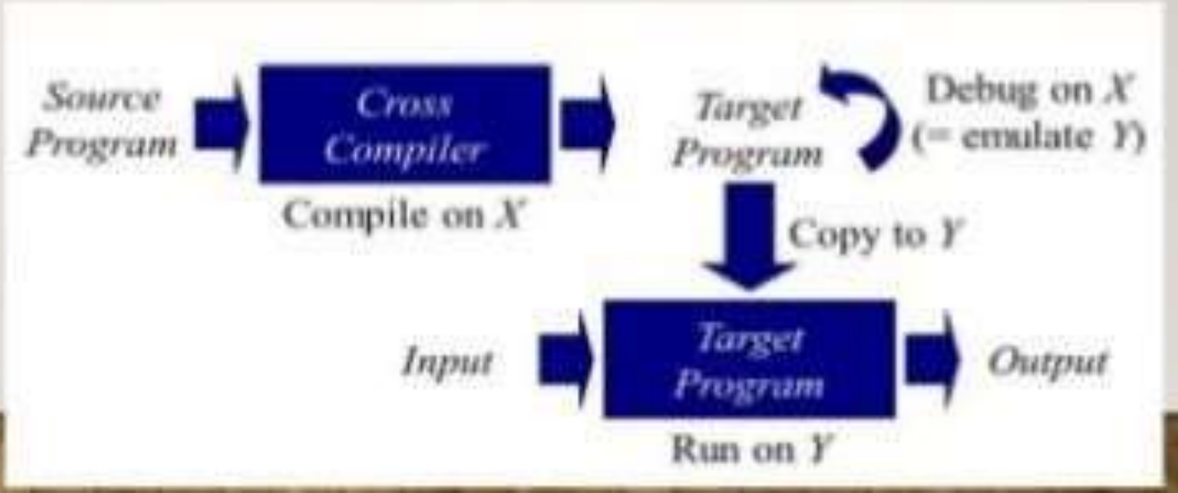


Features of Incremental Compiler

- It tracks the dependencies between output and the source program.
- It produces the same result as full recompile.
- It performs less task than recompilation.
- The process of incremental compilation is effective for maintenance.

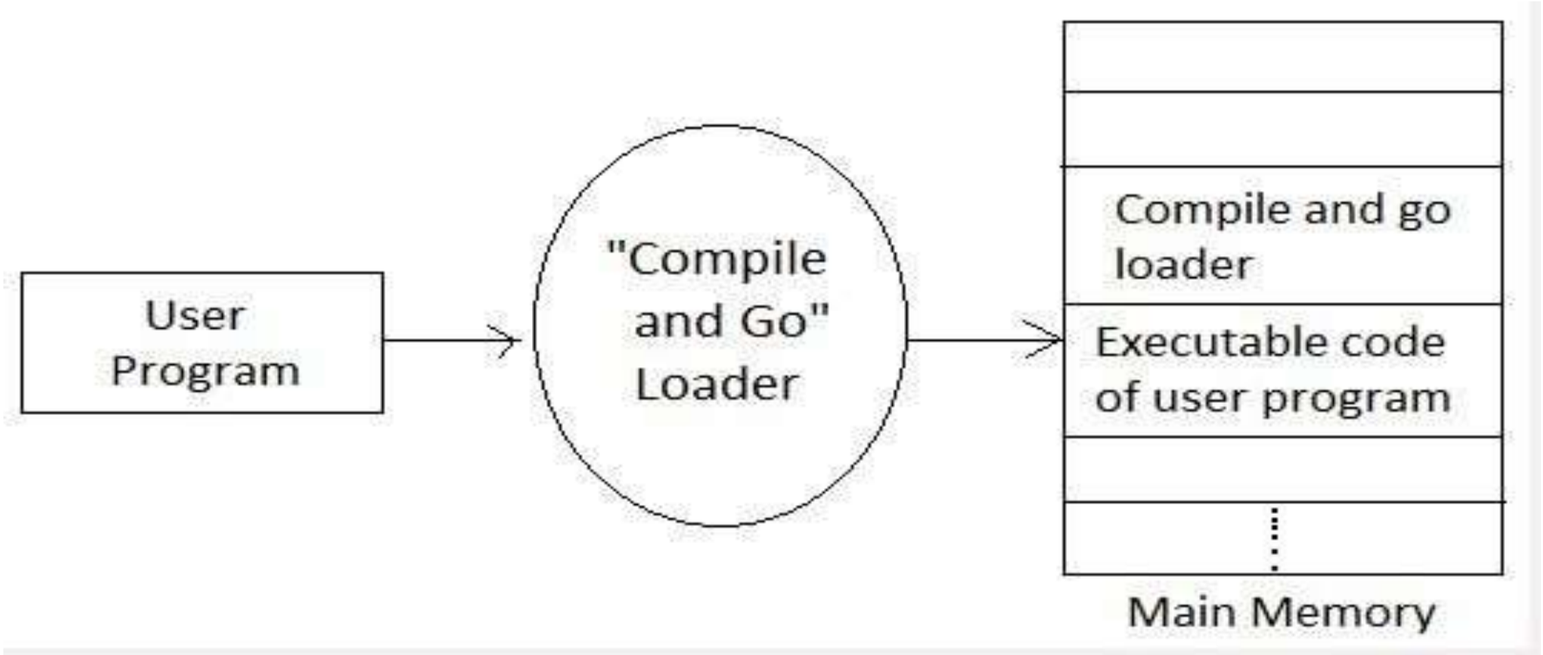
CROSS COMPILER

- A compiler which runs on one machine and produces the target code for another machine. Such compiler is called Cross Compiler.
- Compiler runs on platform X and target code runs on platform Y.



Load & Go System

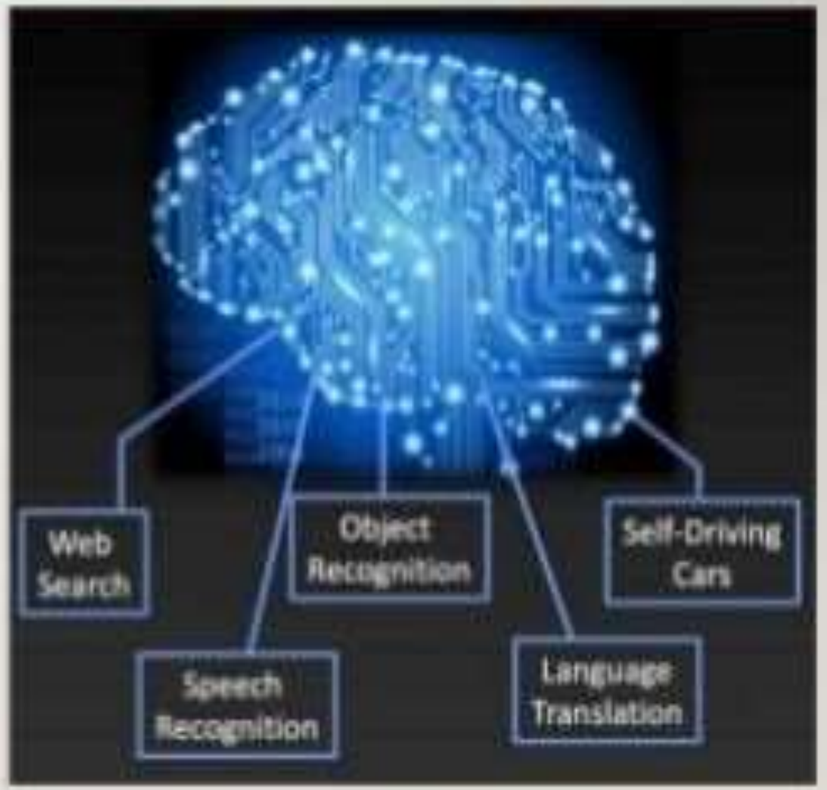
Load and **go** system is a programming language processor in which the **compilation**, assembly, Loader, linker steps are not separated from program Execution.EX: FORTRAN



Threaded Code Compiler
where each op-code in the virtual machine **instruction code** is the
address of some (lower level) code to perform the required operation. This kind of virtual machine can be implemented efficiently in **machine code** on most processors by simply performing an indirect jump to the address which is the next instruction to be executed.

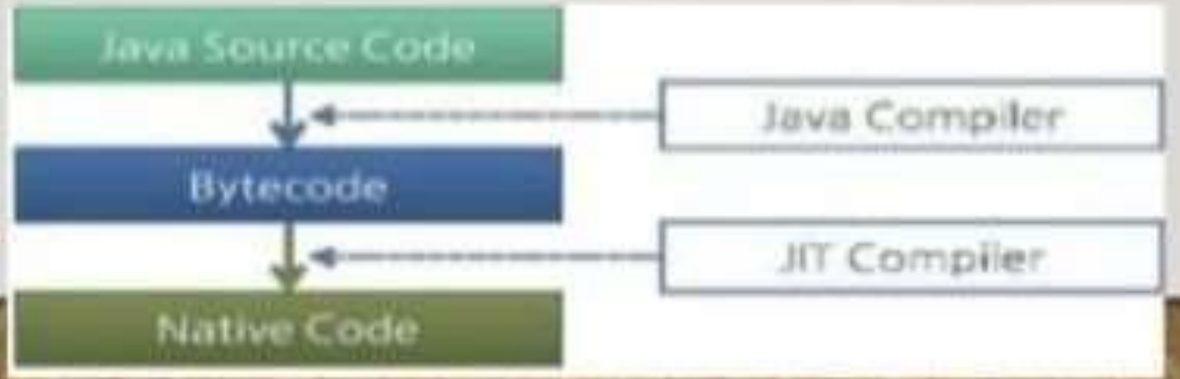
STAGE COMPILER

- Stage compiler compiles Prolog implementations of theoretical machine to its assembly language.
- Prolog is a general-purpose logic programming language associated with artificial intelligence and computational linguistics.
- Prolog is declarative : the program logic is expressed in terms of relations, represented as facts and rules.



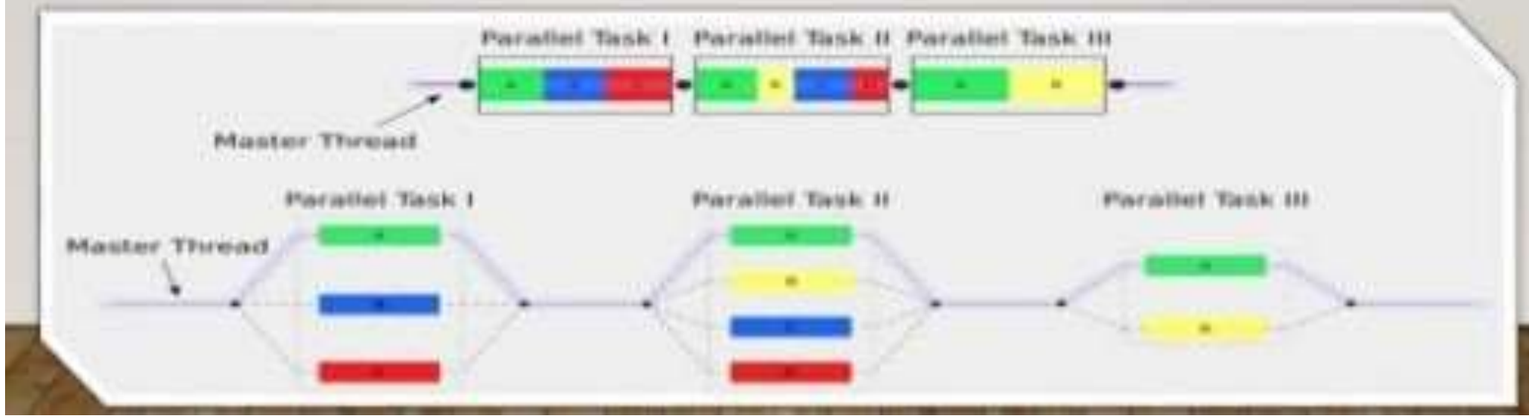
JUST – IN – TIME (JIT) COMPILER

- In this type of compiler applications are delivered in byte code, which is compiled to native machine code just prior to execution.
- The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecodes of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it.



PARALLELIZING COMPILER

- Parallelizing compiler converts a serial input program into a form suitable for efficient execution on a parallel computer architecture.
- The goal of automatic parallelization is to relieve programmers from the hectic and error-prone manual parallelization process.

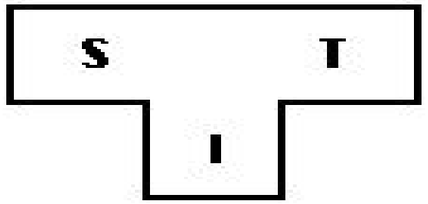


Bootstrapping

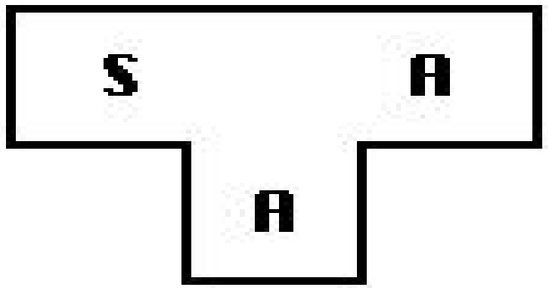
Bootstrapping is widely used in the compilation development. Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code. Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler to compile everything else as well as future versions of itself.

The process described by the T-diagrams is called bootstrapping.

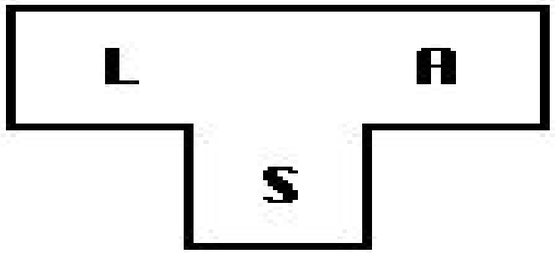
- The T- diagram shows a compiler ${}^{S}C_{I}^{T}$ for Source S, Target T, implemented in I.



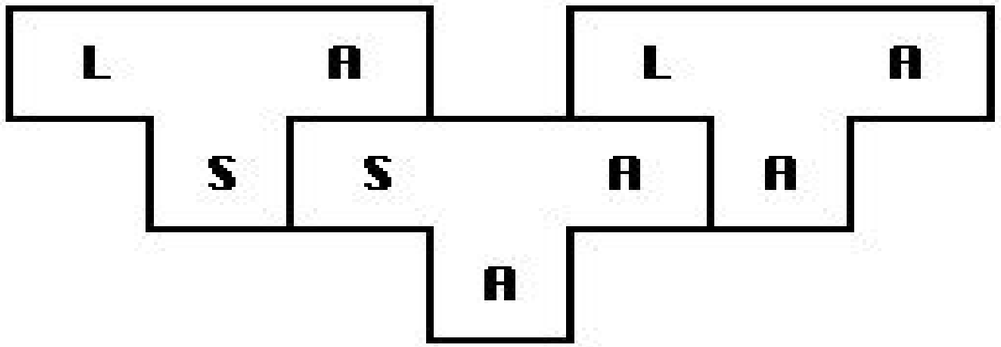
- Follow some steps to produce a new language for machine **A**
- Create a compiler ${}^{S}C_{A}^{A}$ for subset, S of the desired language, L using language "A" and that compiler runs on machine A.



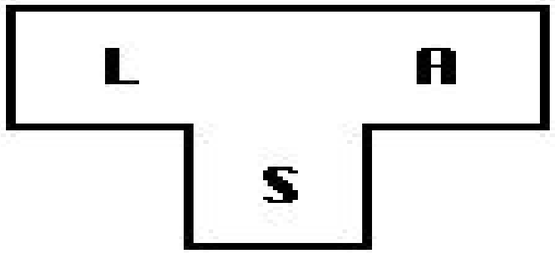
- Create a compiler ${}^{L}C_{S}^{A}$ for language L written in a subset of L.



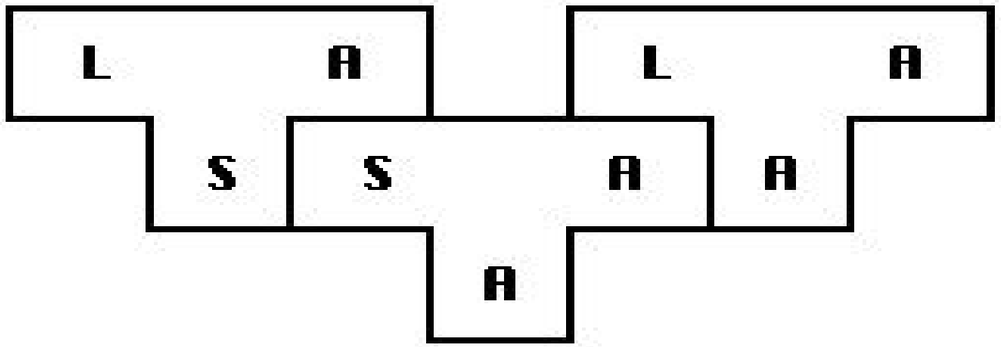
- Compile ${}^{L}C_{S}^{A}$ using the compiler ${}^{S}C_{A}^{A}$ to obtain ${}^{L}C_{A}^{A}$. ${}^{L}C_{A}^{A}$ is a compiler for language L, which runs on machine A and produces code for machine A



- 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 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

An automaton with a finite number of states is called a Finite Automaton (FA) or Finite State Machine (FSM).

A finite automata can be represented by a 5-tuple $(Q, \Sigma, \delta, q0, F)$, where:

- Q is a finite set of states.
- Σ is a finite set of symbols, called the alphabet of the automaton.
- δ is the transition function.
- $q0$ is the initial state from where any input is processed ($q0 \in Q$).
- F is a set of final state/states of Q ($F \subseteq Q$).

Deterministic Finite Automaton (DFA)

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.

- **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.

- **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.

A DFA can be represented by a 5-tuple $(Q, \Sigma, \delta, q0, F)$ where:

- Q is a finite set of states.
- Σ is a finite set of symbols called the alphabet.
- δ is the transition function where $\delta: Q \times \Sigma \rightarrow Q$
- $q0$ is the initial state from where any input is processed ($q0 \in Q$).
- F is a set of final state/states of Q ($F \subseteq Q$).

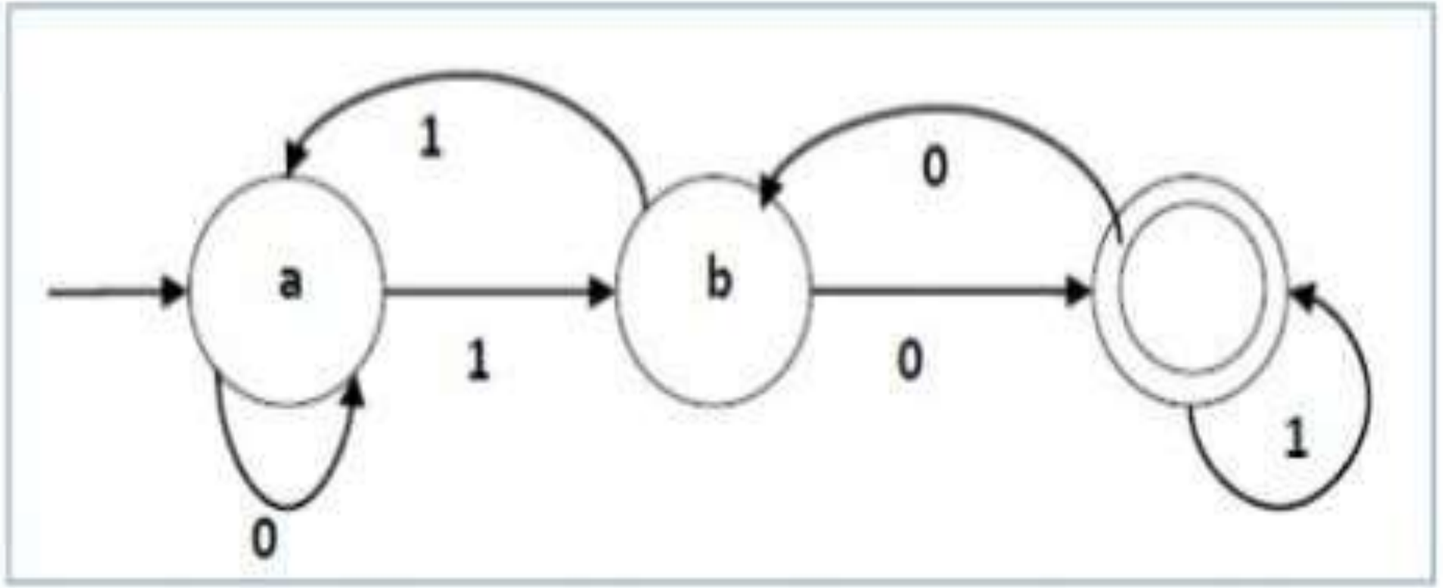
EXAMPLE DFA

Let a deterministic finite automaton be

- Q = {a, b, c},
- Σ = {0, 1},
- q0={a},
- F={c}, and
- Transition function δ as shown by the following table:

Present State	Next State for Input 0	Next State for Input 1
a	a	b
b	c	a
c	b	c

Its graphical representation would be as follows:



Non-deterministic Finite Automaton

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Automaton. As it has finite number of states, the machine is called Non-deterministic Finite Machine or Nondeterministic Finite Automaton.

An NDFA can be represented by a 5-tuple (Q, Σ, δ, q0, F) where:

- Q is a finite set of states.
- Σ is a finite set of symbols called the alphabets.
- δ is the transition function where $\delta: Q \times \Sigma \rightarrow 2^Q$

  (Here the power set of Q (2^Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)
- q0 is the initial state from where any input is processed ($q0 \in Q$).
- F is a set of final state/states of Q ($F \subseteq Q$).

Graphical Representation of an NDFA

Graphical Representation of an NDFA: (same as DFA)

An NDFA is represented by digraphs called state diagram.

- The vertices/Circles represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

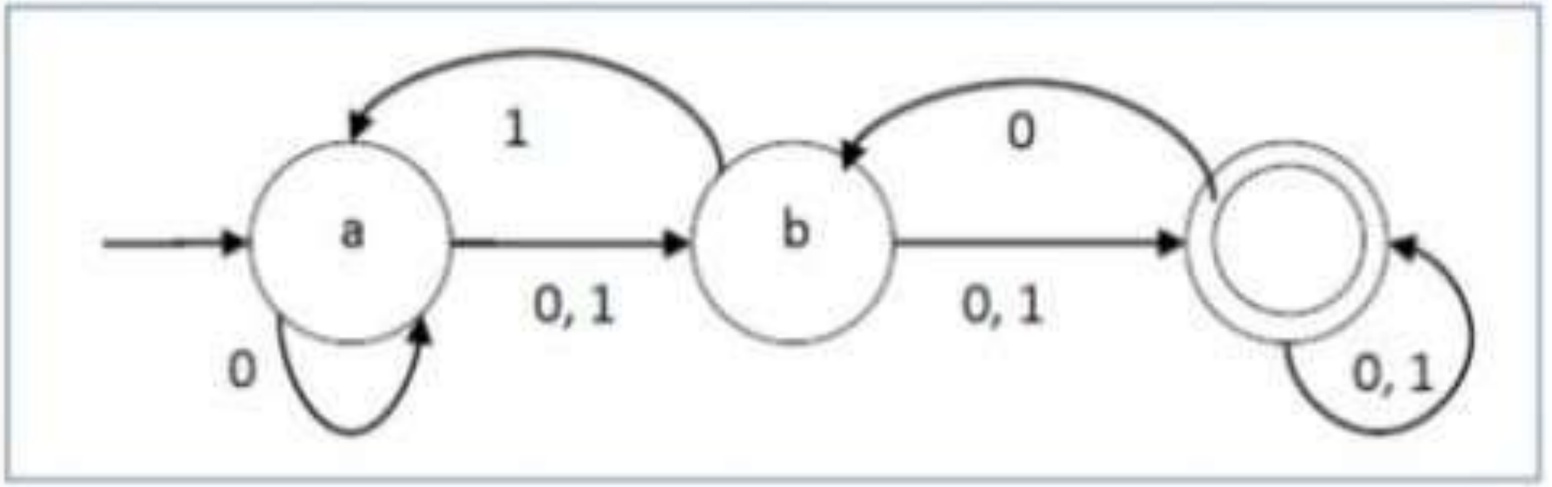
Example

Let a non-deterministic finite automaton be

- Q = {a, b, c}
- Σ = {0, 1}
- q0 = {a}
- F={c} and
- Transition function δ as shown by the following table:

Present State	Next State for Input 0	Next State for Input 1
a	a, b	b
b	c	a, c
c	b, c	c

Its graphical representation would be as follows:



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)*

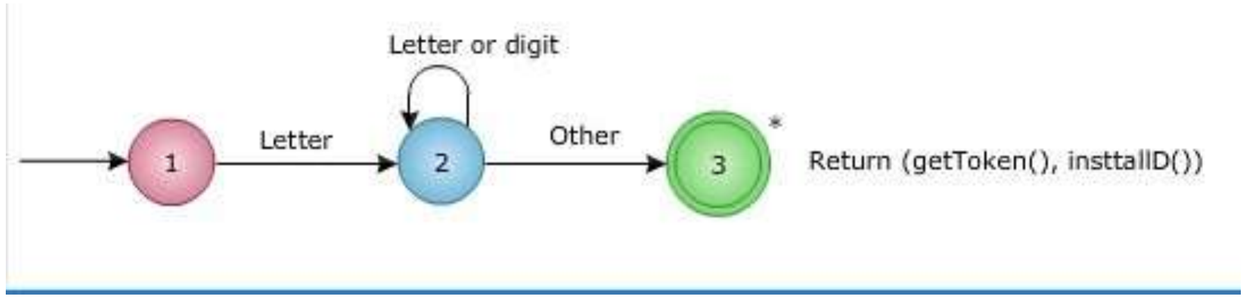
if --> if

then -->then

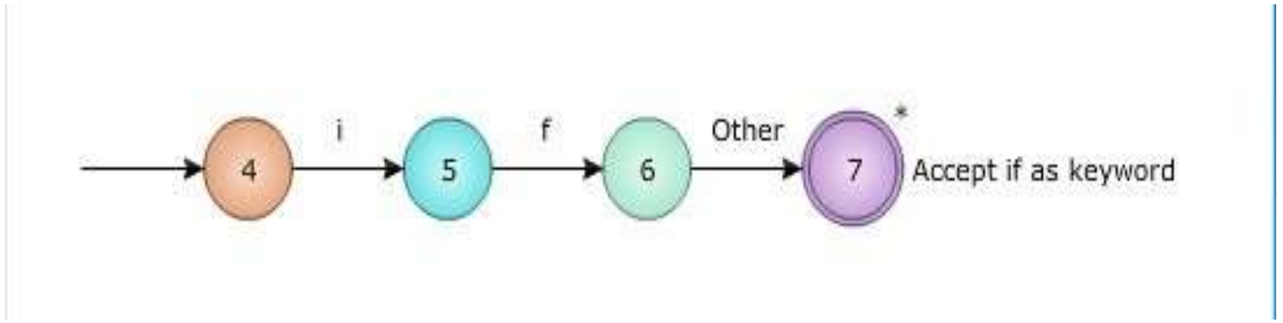
else -->else

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

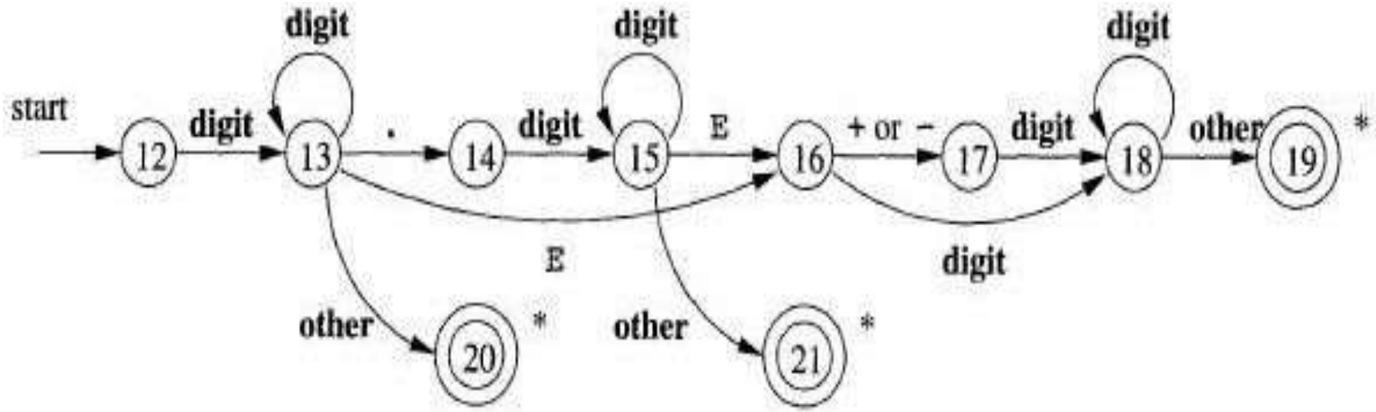
Finite Automata For Recognizing Identifiers



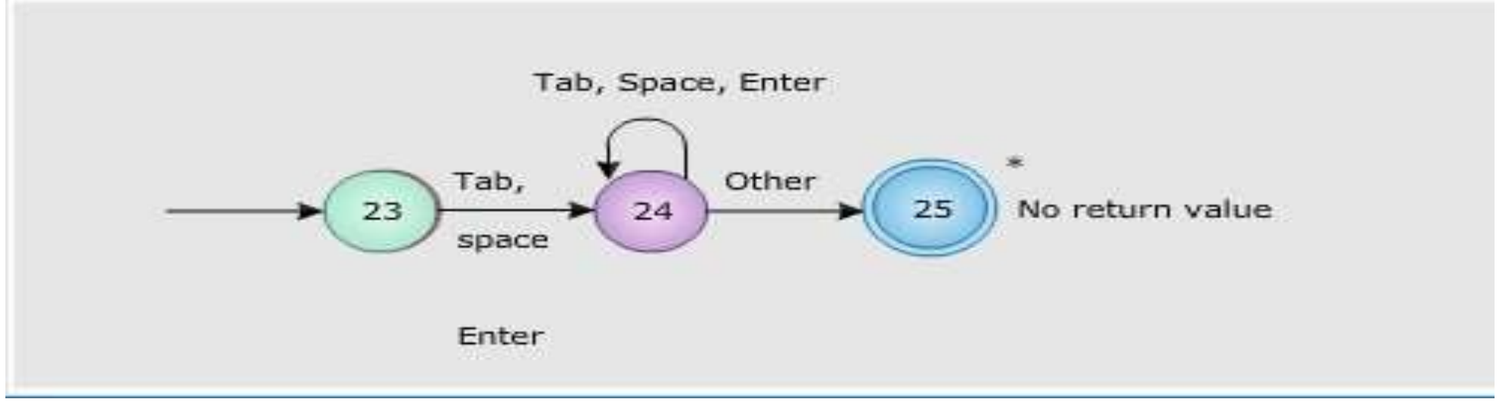
Finite Automata For Recognizing keywords



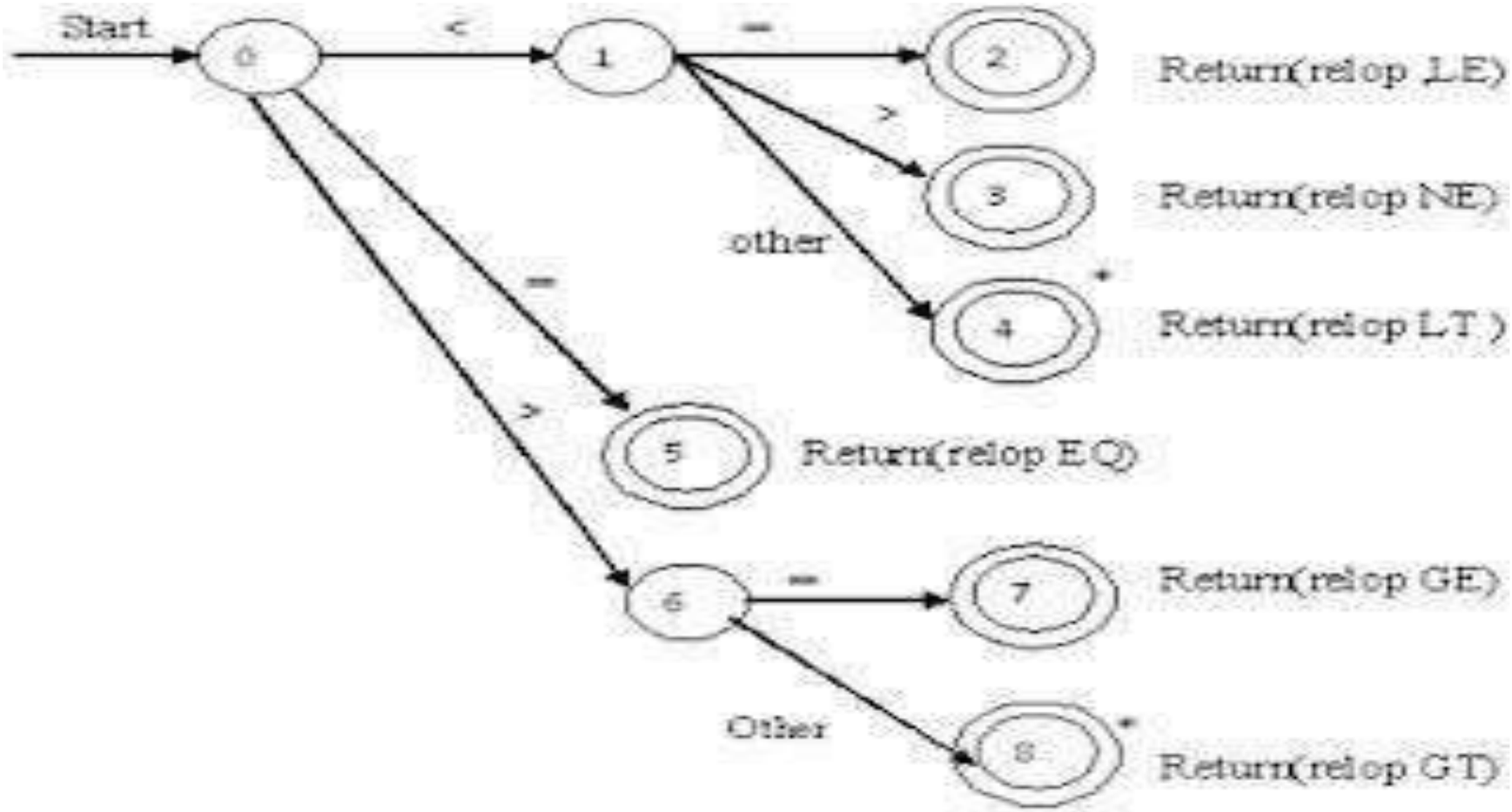
Finite Automata For Recognizing digits



Finite Automata For Recognizing white space



Finite Automata For Recognizing Relational operators



LEXICAL ANALYSIS & ITS ROLE

Lexical analysis

» The scanning/lexical analysis phase of a compiler performs the task of **reading the source program** as a file of characters and **dividing up into tokens.**

tokens

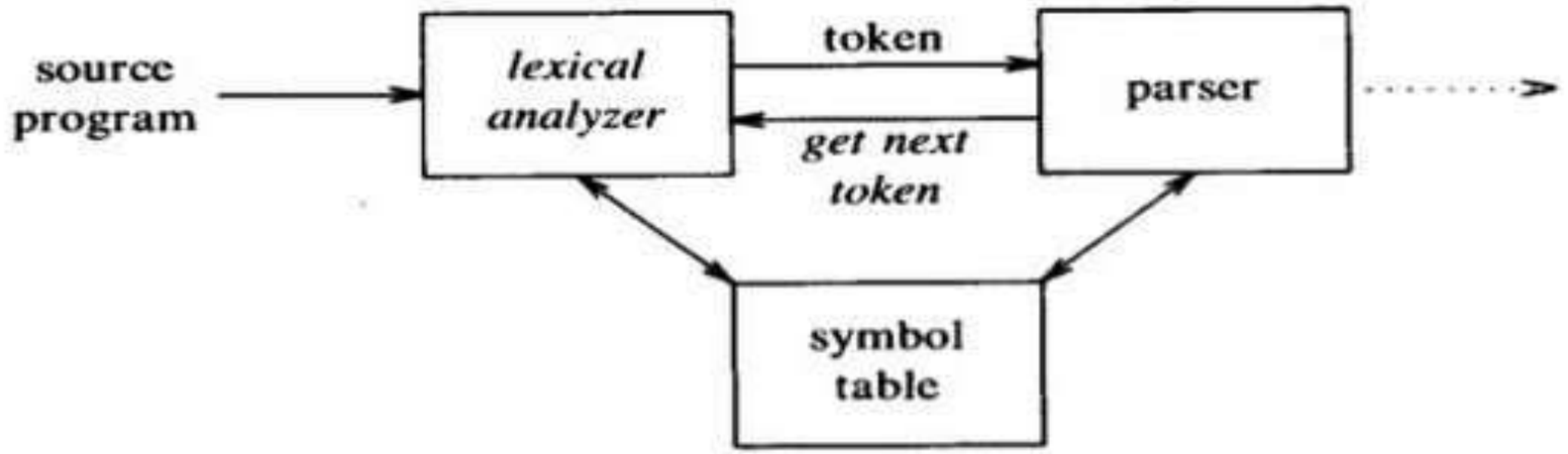
» Each token is a sequence of characters that represents a unit of information in the source program.

Example-tokens

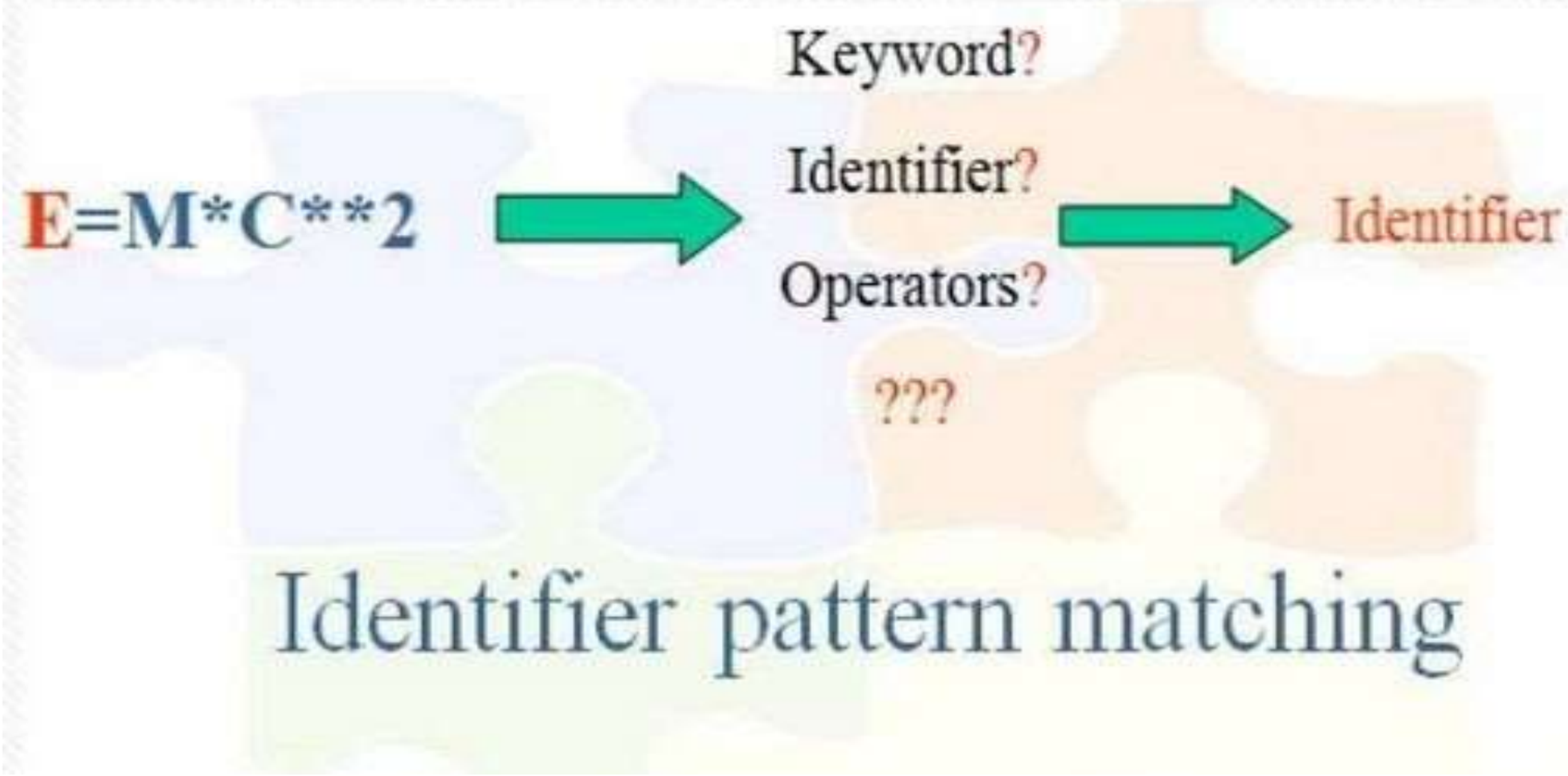
» **Keywords** which are fixed string of letters .eg: “if”, “while”.
» **Identifiers** which are user-defined strings composed of letters and numbers.
» **Special symbols** like arithmetic symbols.

Role of Lexical Analyzer

- Lexical analyzer is the first phase of a compiler.
- Its main task is to read input characters and produce as output a sequence of tokens that parser uses for syntax analysis.



A Simple Lexical Analyzer



Tasks Lexical Analyzer

- **Separation of the input source code into tokens.**
- **Stripping out the unnecessary white spaces** from the source code.
- **Removing the comments** from the source text.
- **Keeping track of line numbers** while scanning the new line characters. These line numbers are used by the error handler to print the error messages.
- **Preprocessing of macros.**

Tokens , Pattern and Lexemes

- Connected with lexical analysis are three important terms with similar meaning.
- Lexeme
- Token
- Patterns
  - A ***token*** is a pair consisting of a **token name and an optional attribute value**. Token name: Keywords, operators, identifiers, constants, literal strings, punctuation symbols(such as commas,semicolons)
  - A ***lexeme*** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an **instance of that token**. E.g.**Relation** {<.<=,>,>=,==,<>}

Pattern: A set of rules describing lexems.

Lexical Errors

- **1.)** let us consider a statement **"fi(a==f)".** Here "fi" is a **misspelled keyword**. This error **is not detected in lexical analysis** as **"fi" is taken as an identifier.** This error is then detected in other phases of compilation.
- **2.)** in case the lexical analyzer is not able to continue with the process of compilation, it resorts to **panic mode of error recovery**.
- ***Deleting the successive characters*** from the remaining input until a token is detected.
- ***Deleting extraneous characters.***

- *Inserting missing characters*
- *Replacing an incorrect character by a correct character.*
- *Transposing two adjacent characters*

PATTERN, TOKENS, LEXEMS

Tokens are terminal symbols of the source language.

↳ identifiers, numbers, keywords, punctuation symbols etc.

Pattern is a rule describing all those lexemes that can represent a particular token in a source language.

ids : Start with an alphabet or _, followed by any alphanumeric char

Lexemes are matched against the pattern

↳ specific instance of a token.

`count = count + temp ;` =, +

Token: id, operator, punctuation (;)

↳ count, temp.

eg: 31 + 28 − 59

Tokens: Number : $[0\text{-}9]^+$ → 31, 28, 59.

Operator : +, −

LEXICAL ERRORS : A Lexical Analyzer may not proceed if no rule/pattern (for tokens) matches the prefix of the remaining input.

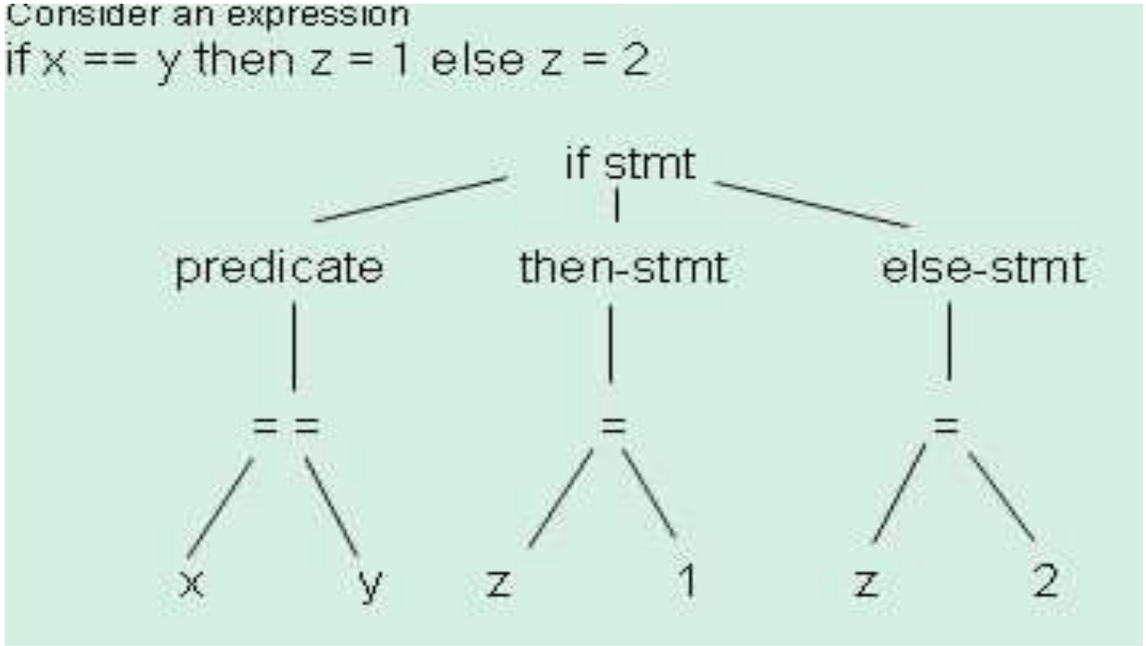
Solution :

- deleting an extraneous character(s)
- inserting a missing character.
- transposing 2 adjacent character.
- replacement of a particular char. by another char.
- deletion of successive chars. from input.

LECTURE CONTENTS WITH A BLEND OF NPTEL CONTENTS

Parsing

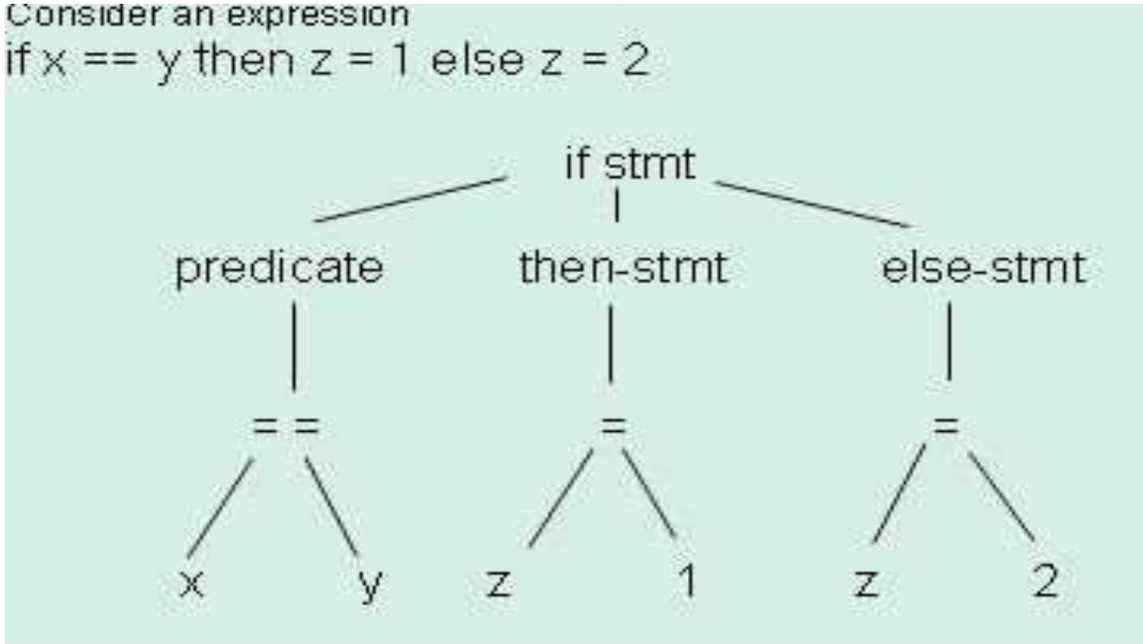
Just like a natural language, a programming language also has a set of grammatical rules and hence can be broken down into a parse tree by the parser. It is on this parse tree that the further steps of semantic analysis are carried out. This is also used during generation of the intermediate language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



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.



REFERENCES/BIBLOGRAPHY

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/

JECRC Foundation

JAIPUR ENGINEERING COLLEGE
AND RESEARCH CENTRE

Thank you!