

# SYSTEM SOFTWARE AND COMPILERS I8CS6I

## MODULE-2

Dr.Sanchari Saha  
Assistant Professor  
Dept. of CSE,  
CMRIT, Bangalore

## Text Books

1. System Software by Leland. L. Beck, D Manjula, 3rd edition, 2012

Module 1

2. Alfred V Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman ,  
Compilers-Principles, Techniques and Tools, Pearson, 2nd edition,  
2007 .

Module 2, 3, 5

3. Doug Brown, John Levine, Tony Mason, lex & yacc, O'Reilly Media,  
October 2012.

Module 4

## Course Learning Objectives:

This course (18CS61) will enable students to:

- ☐ Define System Software.
- ☐ Familiarize with source file, object file and executable file structures and libraries
- ☐ Describe the front-end and back-end phases of compiler and their importance to students

# Course Outcome and CO-PO Mapping

CO-PO and CO-PSO Mapping																			
Course Outcomes		Modules covered	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4	
CO1	Explain system software	1	3	3	2	2	-	-	-	-	-	-	-	-	-	-	-	-	
CO2	Design and develop lexical analyzers, parsers and code generators	2,3,4,5	3	3	3	3	3	2	-	-	-	-	-	-	-	-	2	-	
CO3	Utilize lex and yacc tools for implementing different concepts of system software	4	3	3	3	3	3	2	-	-	-	-	-	-	-	-	3	-	

## Module 2- syllabus

Introduction: Language Processors, The structure of a compiler, The evaluation of programming languages, The science of building compiler, Applications of compiler technology.

Lexical Analysis: The role of lexical analyzer, Input buffering, Specifications of token, recognition of tokens.

**Text book 2:**Chapter 1 1.1-1.5 Chapter 3: 3.1 – 3.4

**RBT:** L1, L2, L3

# VTU QP

Aug-2022

## Module-2

- 3 a. Explain various phases of Compiler. Show the translations for an Assignment statement.  
Position = Initial + rate \* 60.  
Clearly indicate the output of each phase. (12 Marks)  
b. What are the applications of Compiler? Explain. (08 Marks)
- OR**
- 4 a. Write a brief note on Language Processing System. (06 Marks)  
b. Explain the concept of input buffering in the Lexical analysis with its implementation. (10 Marks)  
c. Define Token , Lexeme and Pattern with example. (04 Marks)

# VTU QP

Jan-2023

## Module-2

- 3
- a. Explain the structure of a compiler with an example. (10 Marks)
  - b. List and example the applications of compiler technology. (06 Marks)
  - c. Differentiate between type checking and bound checking. (04 Marks)

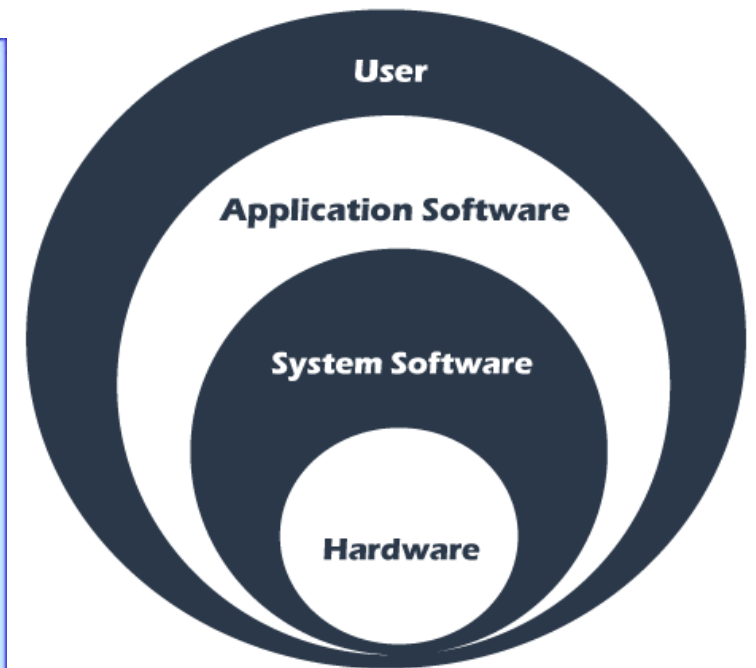
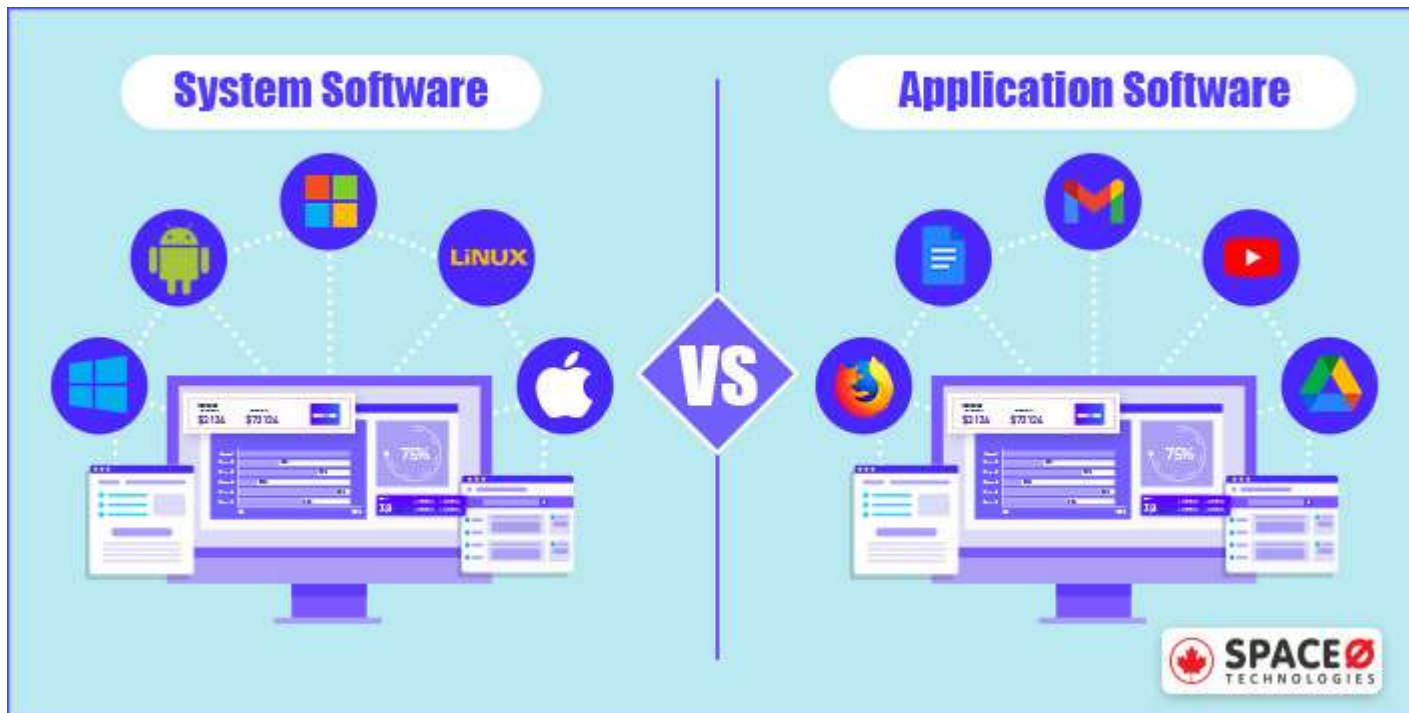
OR

- 4
- a. Explain the role of lexical analyzer. (08 Marks)
  - b. What is regular expression? Write the algebraic laws of regular expression. (06 Marks)
  - c. Explain the concept of input buffering in the Lenticels analysis. (06 Marks)



# Introduction

Software is a set of programs, procedures, and routines that instructs a computer system what to do. It is mainly of two types named as, **System Software** and **Application Software**.





# What is System Software?

*System software is a set of computer programs, which is designed to manage system resources.* It is a collection of such files and utility programs that are responsible for running and smooth functioning of your computer system with other hardware. Moreover, it is solely responsible for running the operating system (OS) and managing the computer device entirely, and without it, the system cannot run. It is not used for a specific task and is hence known as **general-purpose software**.

It acts as a platform for other software to work, such as *antivirus software, OS, compiler, disk formatting software*, etc.

System Software is usually written using Low-level language such as Assembly language. Some essential functions of System software are:

- **Disk Management**
- **Memory Management**
- **Device controlling**
- **Loading and execution of other programs.**

# What is Application Software

*Application Software is a type of software that is mainly developed to perform a specific task as per the user's request.* It acts as an interface between the end-user and system software.

Application software is not used to perform basic operations of a computer system like system software. Instead, they are installed on the the computer system to function as a working tool for the end-user.

Application software provides an interactive UI (user interface) for users to interact with it and work on it.

This software is usually developed with the help of High-level languages, such as C, C++, Java, etc. Some examples of Application software are **MS Office, Paint, Spreadsheet, Web browser, etc.**

Although application software is designed to perform a specific task, some standard functions of application software are given below:

- **Data Manipulation**
- **Writing Reports**
- **Creating Spreadsheets**
- **Managing records.**

# System Software vs Application Software

Parameter	System Software	Application Software
Definition	System Software is the type of software which is the interface between application software and system.	Application Software is the type of software which runs as per user request. It runs on the platform which is provide by system software.
Development Language	In general, System software are developed using low-level language which is more compatible with the system hardware in order to interact with.	In case of Application software, high level language is used for their development as they are developed as some specific purpose software.
Necessity	System software are essential for operating the computer hardware. Without these software, a computer even may not start or function properly.	Application software are not essential for the operation of the computer. These are installed as per the user's requirements.
Usage	System software is used for operating computer hardware.	Application software is used by user to perform specific task.

Installation	System software are installed on the computer when operating system is installed.	Application software are installed according to user's requirements.
User interaction	System software are specific to system hardware, so less or no user interaction available in case of system software.	Users can interact with an application software with the help of a User Interface (UI).
Dependency	System software can run independently. It provides platform for running application software.	An application software cannot run independently. It cannot run without the presence of system software.
Examples	Examples of system software include operating systems, compilers, assemblers, debuggers, drivers, etc.	Examples of application software include word processors, web browsers, media players, etc.

# Chapter 1: Introduction

## Language Processors

**1) Compiler:** a compiler is a program that can read a program in one language: the source language and translate it into an equivalent program in another language : the target language.

see Fig. 1.1. An important role of the compiler is to report any errors in the source program that it detects during the translation process. If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs; see Fig. 1.2.

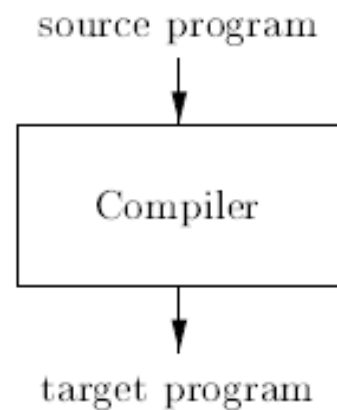


Figure 1.1: A compiler

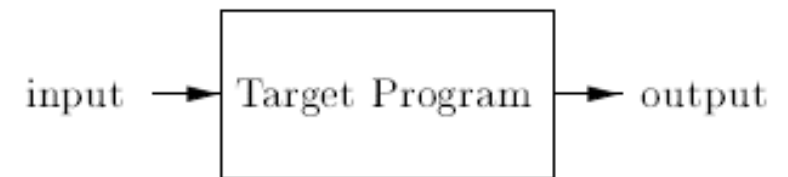


Figure 1.2: Running the target program

**2) Interpreter:** An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in Fig. 1.3.

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

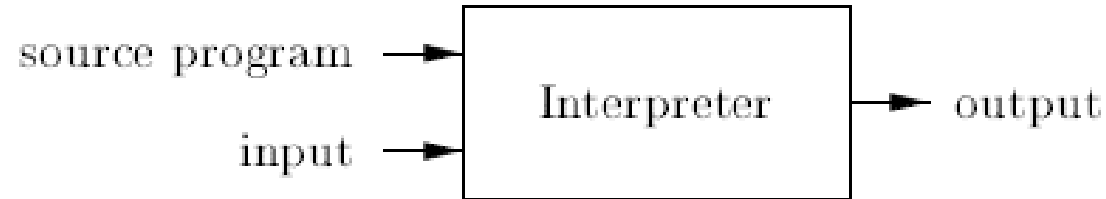


Figure 1.3: An interpreter

### Hybrid Compiler

Java language processors combine compilation and interpretation, as shown in Fig. 1.4. A Java source program may first be compiled into an intermediate form called **bytecodes**. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

In order to achieve faster processing of inputs to outputs, some Java compilers, called just-in-time compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input

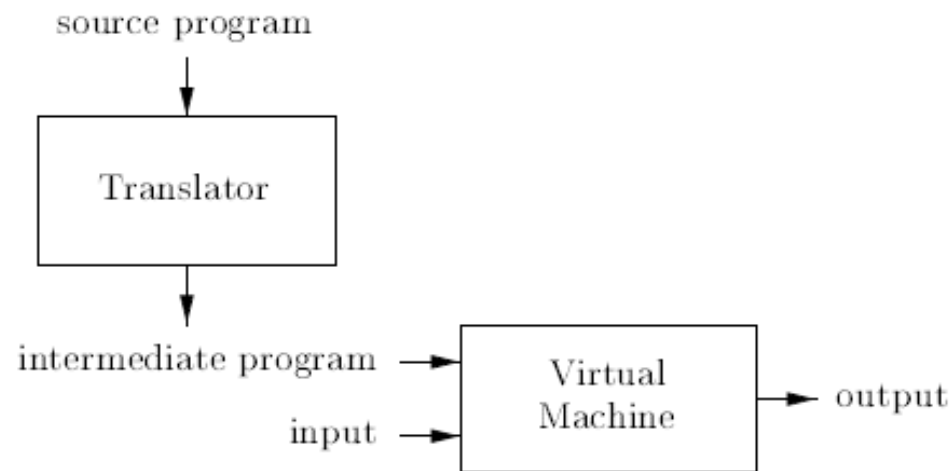



Figure 1.4: A hybrid compiler



Compiler	Interpreter
Converts the complete source program to target program before execution	Executes the statement line by line
Target program produced is much faster at mapping inputs to outputs	Mapping inputs to outputs are slower
The programs are not executed during conversion so errors diagnostics are not as good as interpreter	Error diagnostics is better as it executes the statement line by line during conversion



**3) Preprocessor:** A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a preprocessor. The preprocessor may also expand shorthand, called macros, into source language statements.

The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug.

#### **4) Assembler:**

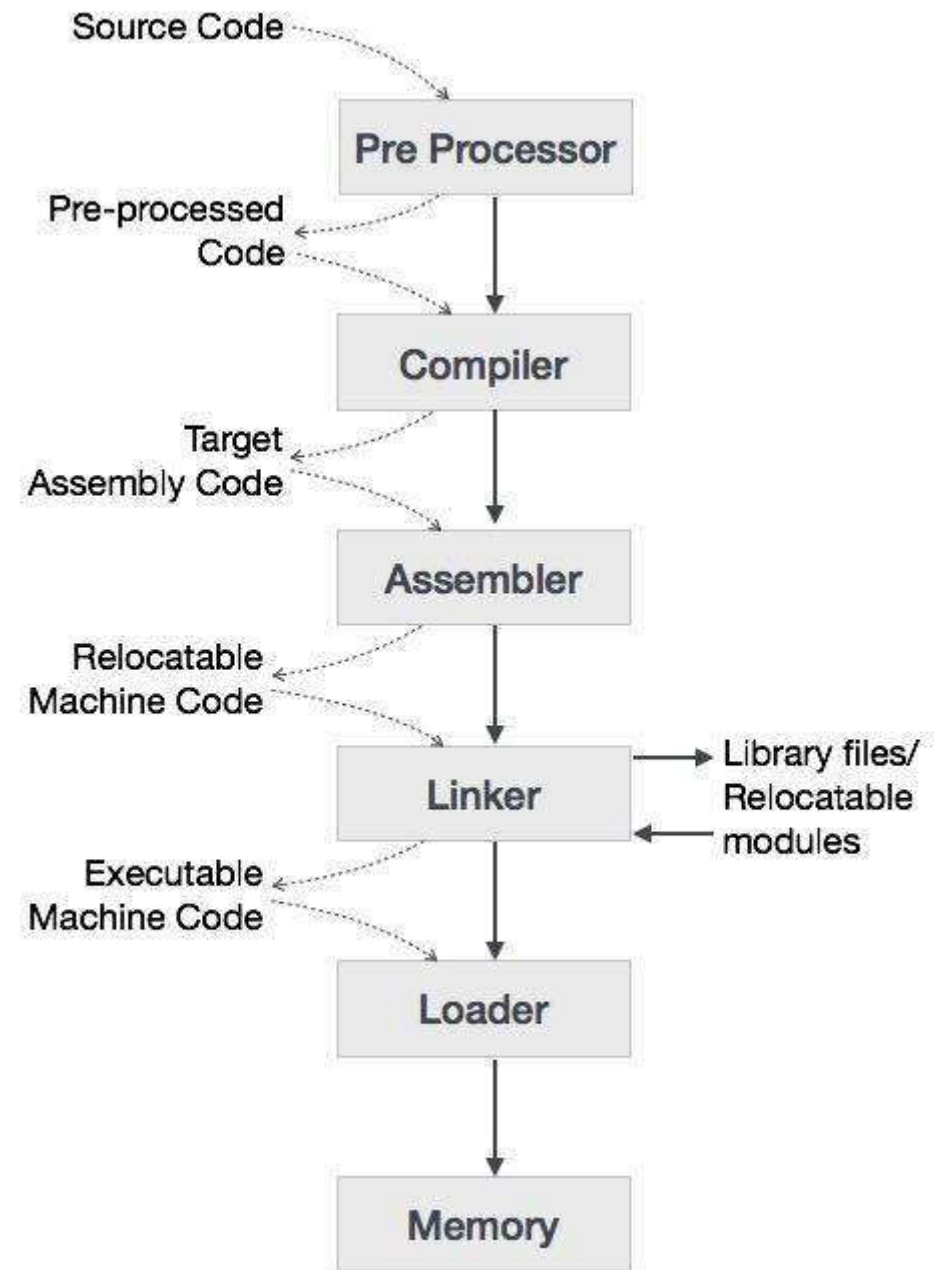
The assembly language is processed by a program called an assembler that produces relocatable machine code as its output.

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.

**5) Linker:** The linker resolves external memory addresses, where the code in one file may refer to a location in another file.

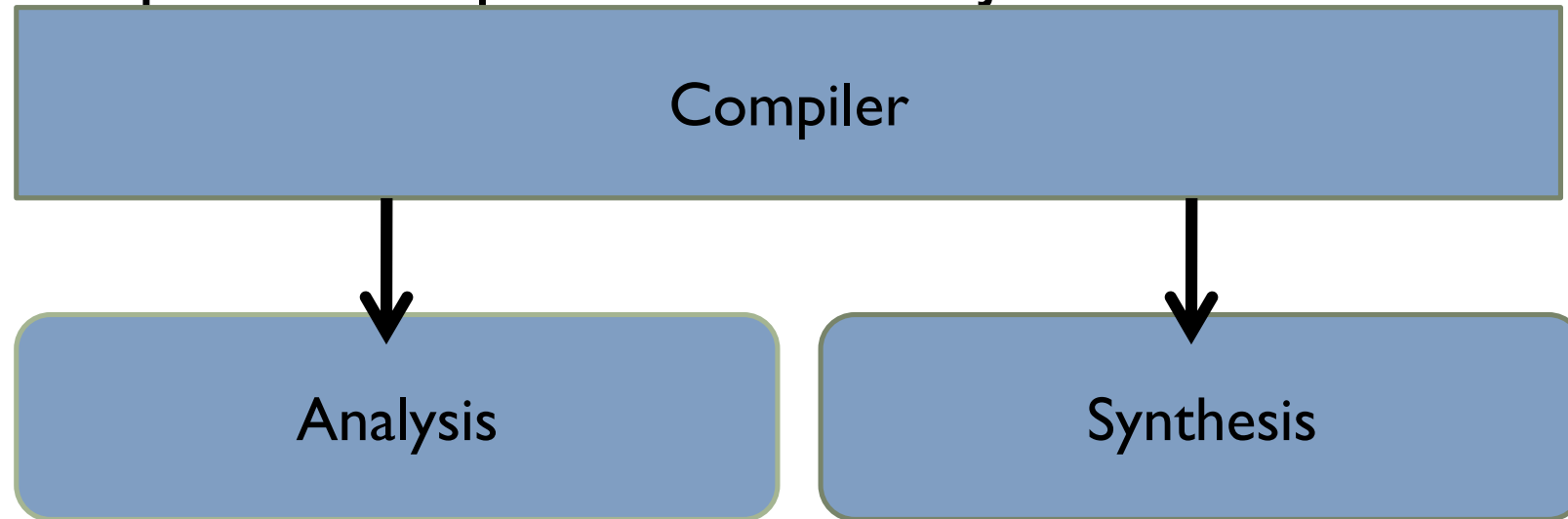
**6) Loader:** The loader puts together all of the executable object files into memory for execution.

Figure 1.5: A Language-processing system



## The Structure of a Compiler

Any compiler must perform two major tasks



**Analysis** of the source program (front end) -machine independent code)

**Synthesis** of a machine-language program-back end(machine dependent code)

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.

The **synthesis part** *constructs the desired target program from the intermediate representation and the information in the symbol table.*

The analysis part is often called the front end of the compiler; the synthesis part is the back end.

If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another.

A typical decomposition of a compiler into phases is shown in Fig. 1.6.

**Symbol Table:** The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

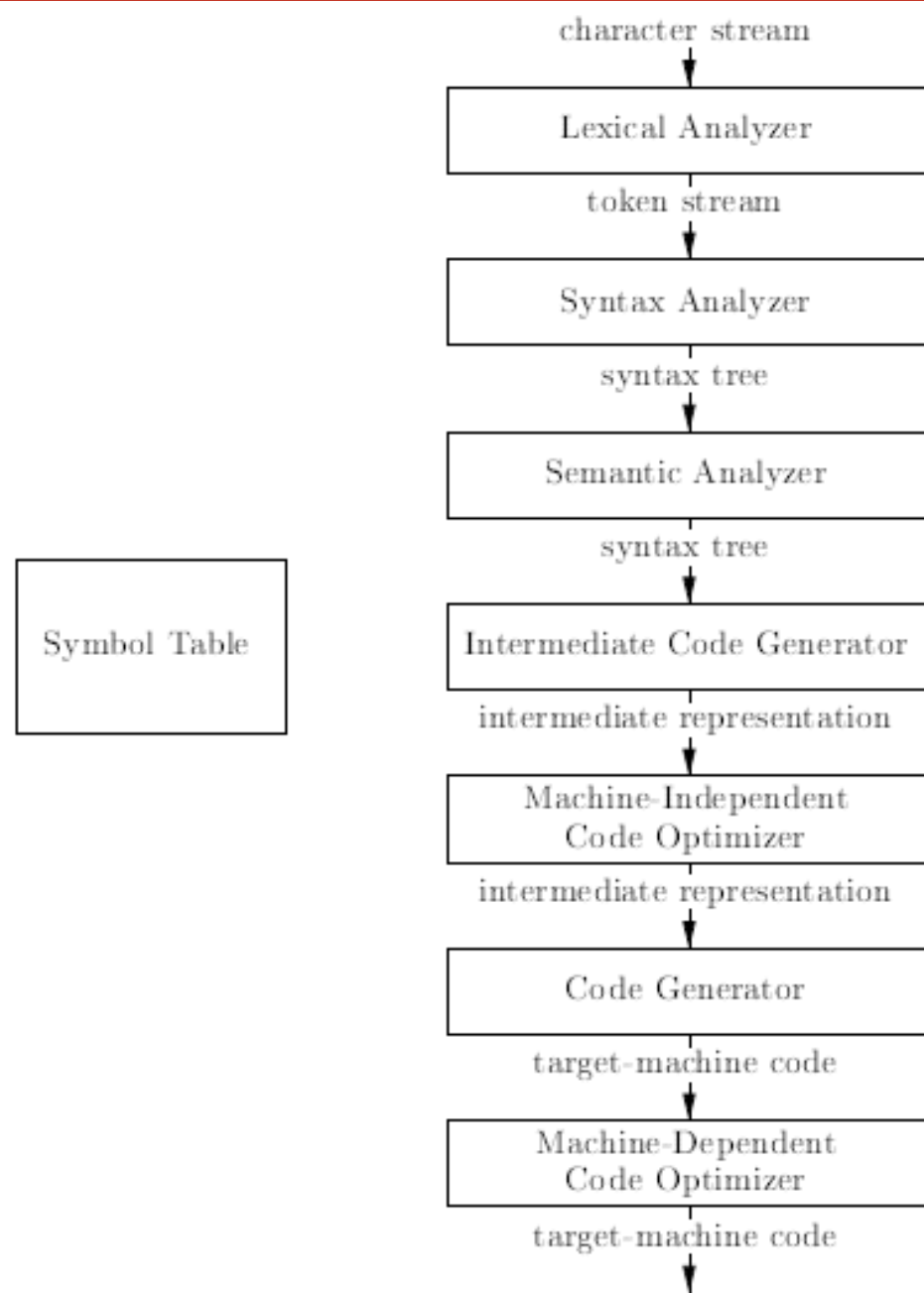
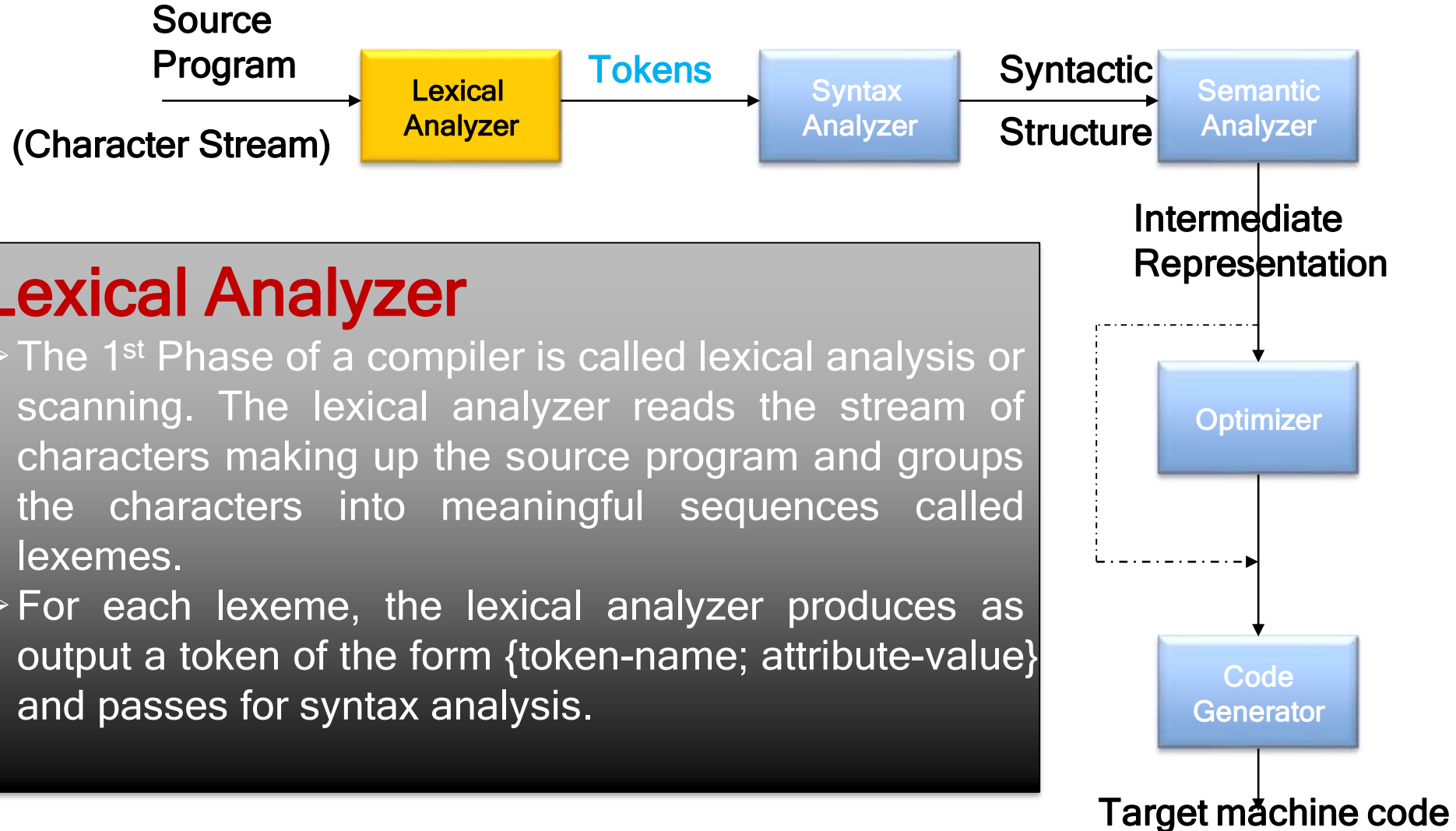


Figure 1.6: Phases of a compiler



# The Structure of a Compiler (Phase 1)



## Token and lexemes

**In the token**, the 1<sup>st</sup> component token-name is an abstract symbol that is used during syntax analysis, and 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.

For example, suppose a source program contains the assignment statement:

**position = initial + rate \* 60**

**1) position is a lexeme** that would be mapped into a token <id, 1> , where id is an abstract symbol standing for identifier and 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.

**2) The assignment symbol =** is a lexeme that is mapped into the token <=>

**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  $\langle + \rangle$ .
5. **rate** is a **lexeme** that is mapped into the token  $\langle \text{id}, 3 \rangle$ , where 3 points to the symbol-table entry for rate.
6. **\*** is a **lexeme** that is mapped into the token  $\langle * \rangle$ .
7. **60** is a **lexeme** that is mapped into the token  $\langle 60 \rangle$

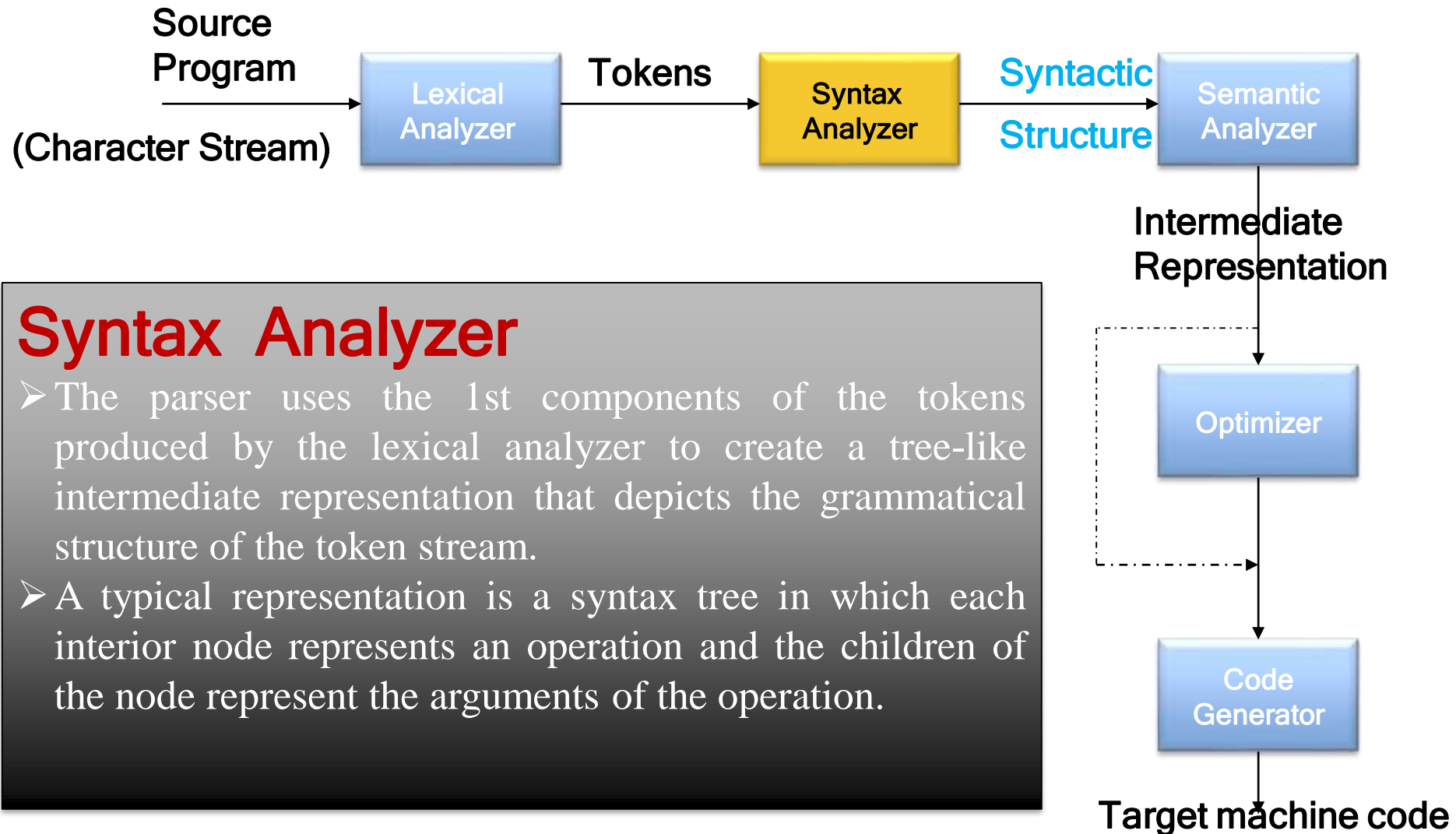
**position = initial + rate \* 60**



$\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$

SYMBOL TABLE		
1	position	...
2	initial	...
3	rate	...
4		

# The Structure of a Compiler (Phase 2)



This tree shows the order in which the operations in the assignment

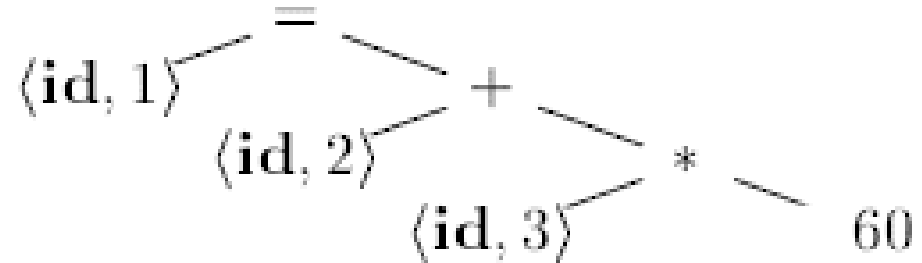
`position = initial + rate * 60`

are to be performed. The tree has an interior node labeled `*` with `<id, 3>` as its left child and the integer `60` as its right child. The node `<id, 3>` represents the identifier `rate`. The node labeled `*` makes it explicit that we must first multiply the value of `rate` by `60`. The node labeled `+` indicates that we must add the result of this multiplication to the value of `initial`. The root of the tree, labeled `=`, indicates that we must store the result of this addition into the location for the identifier `position`. This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

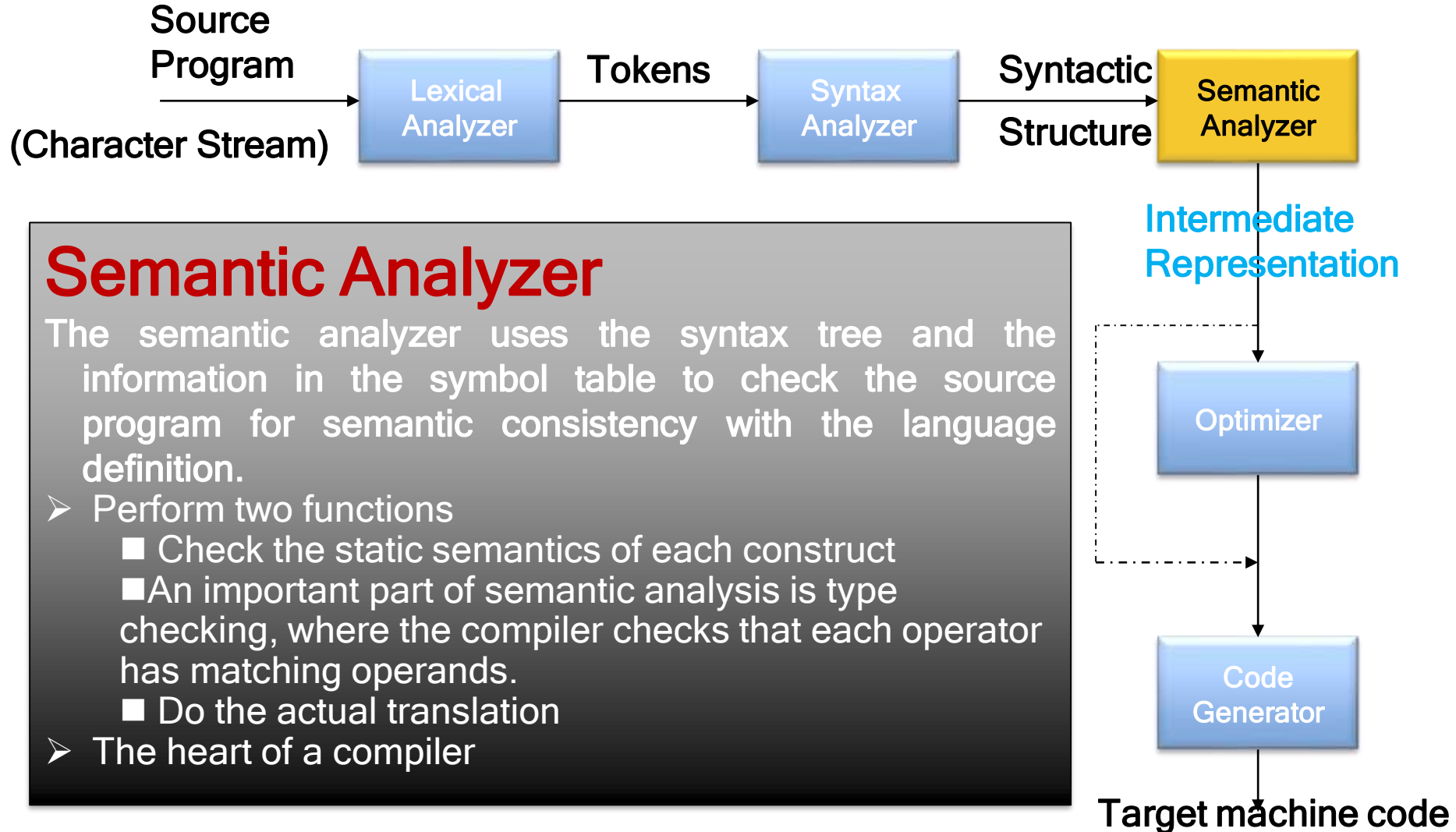
`<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>`

Syntax Analyzer  
[Parser]

Syntax Tree



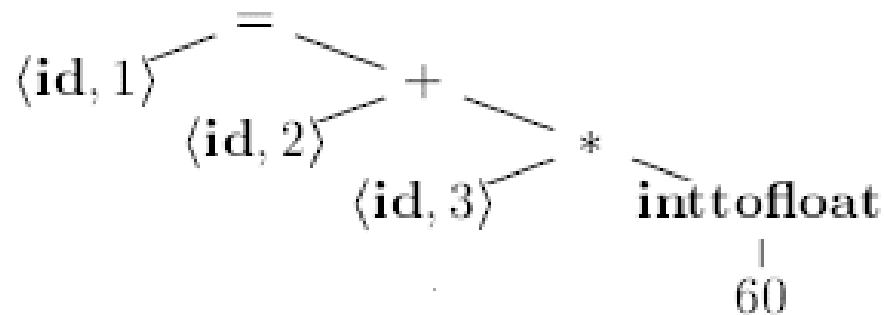
# The Structure of a Compiler (Phase 3)



## For example -three-address code-intermediate code

`position = initial + rate * 60`

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 type checker in 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. notice that the output of the semantic analyzer has an extra node for the operator **inttofloat**, which explicitly converts its integer argument into a floating-point number.





# Intermediate Code Generation:

## Characteristics of Three address instructions

We consider an intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register. The output of the intermediate code generator consists of the three-address code sequence.

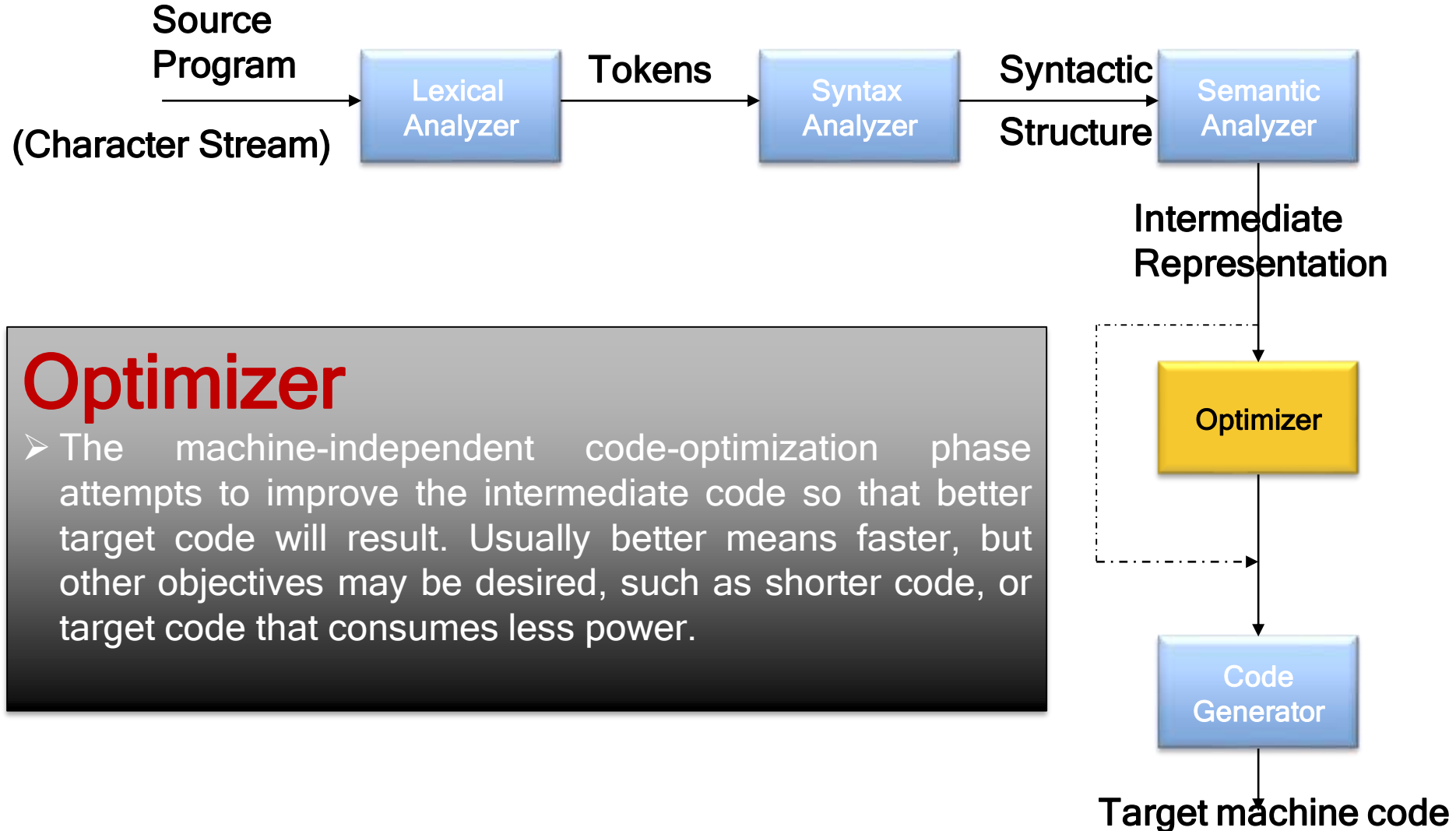
A three-address assignment instruction has **at most one operator** on the **right side**, thus, the order in which these operations are performed are fixed.

The **compiler** must **generate a temporary name to hold the value computed by a three-address instruction**.

Some "three-address instructions" have **fewer than three operands**.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

# The Structure of a Compiler (Phase 4)



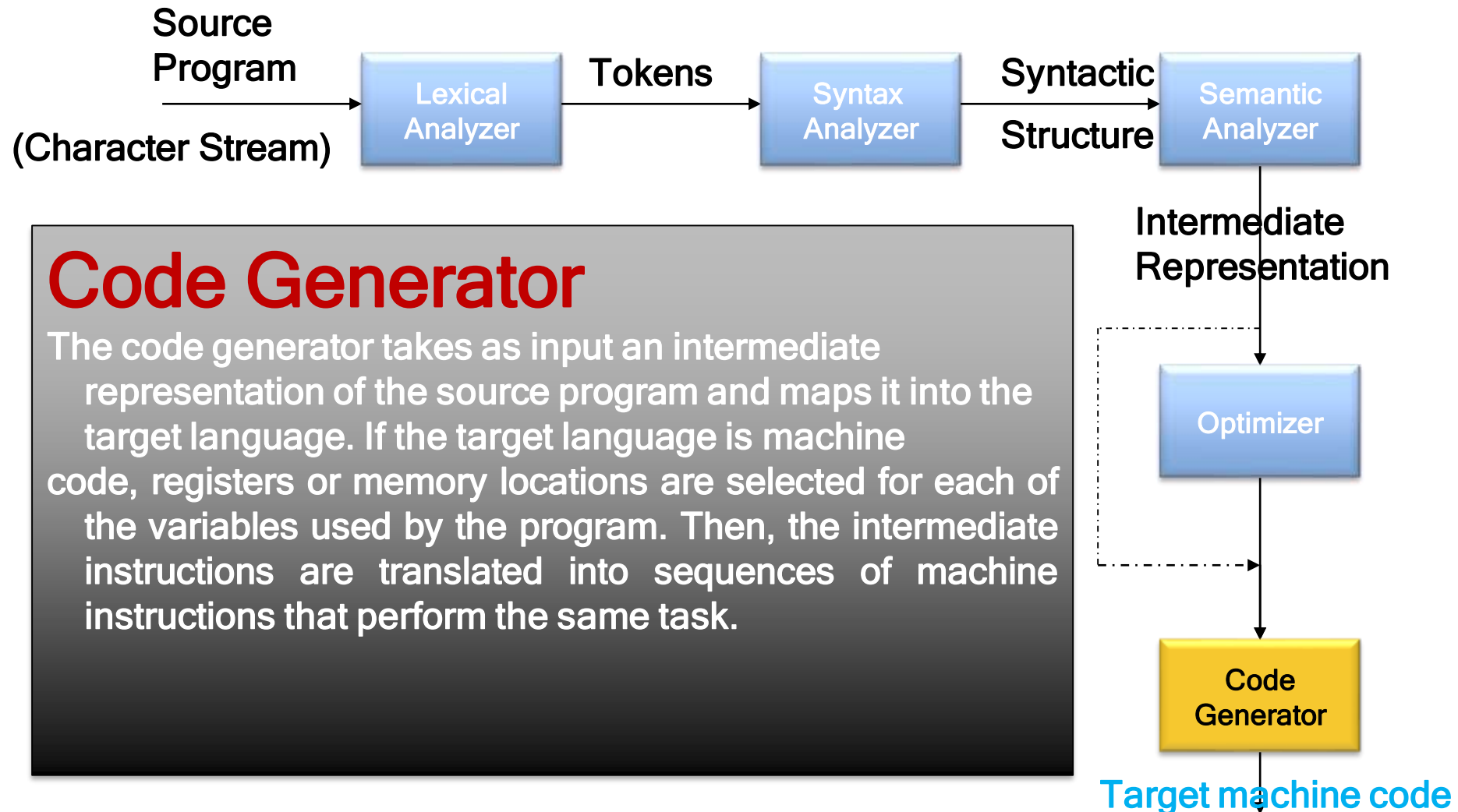
## Code optimizer

The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time,

so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0.

```
t1 = id3 * 60.0  
id1 = id2 + t1
```

# The Structure of a Compiler (Phase 5)



## Machine code-target code

The first operand of each instruction specifies a destination. The F in each instruction tells us that it deals with floating-point numbers. The code in 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 R1 and the fourth adds to it the value previously computed in register R2. Finally, the value in register R1 is stored into the address of id1.

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

# Translation of an assignment statement

SYMBOL TABLE

1	position	...
2	initial	...
3	rate	...
4		

Tokens & Lexemes

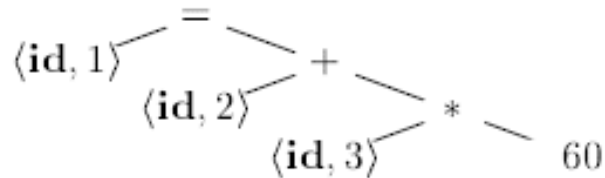
Syntax Tree

position = initial + rate \* 60

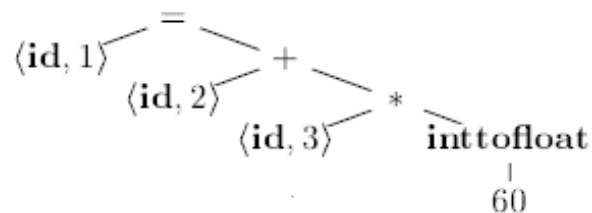
Lexical Analyzer  
[Scanner]

$\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$

Syntax Analyzer  
[Parser]



Semantic analyzer  
[Semantic Process]



Intermediate Code Generator

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Non-optimized Intermediate Code

Code Optimizer

```
t1 = id3 * 60.0
id1 = id2 + t1
```

optimized Intermediate Code

Code Generator

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

Target machine code

# Symbol-Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name

**The attributes in the symbol table** may provide information about

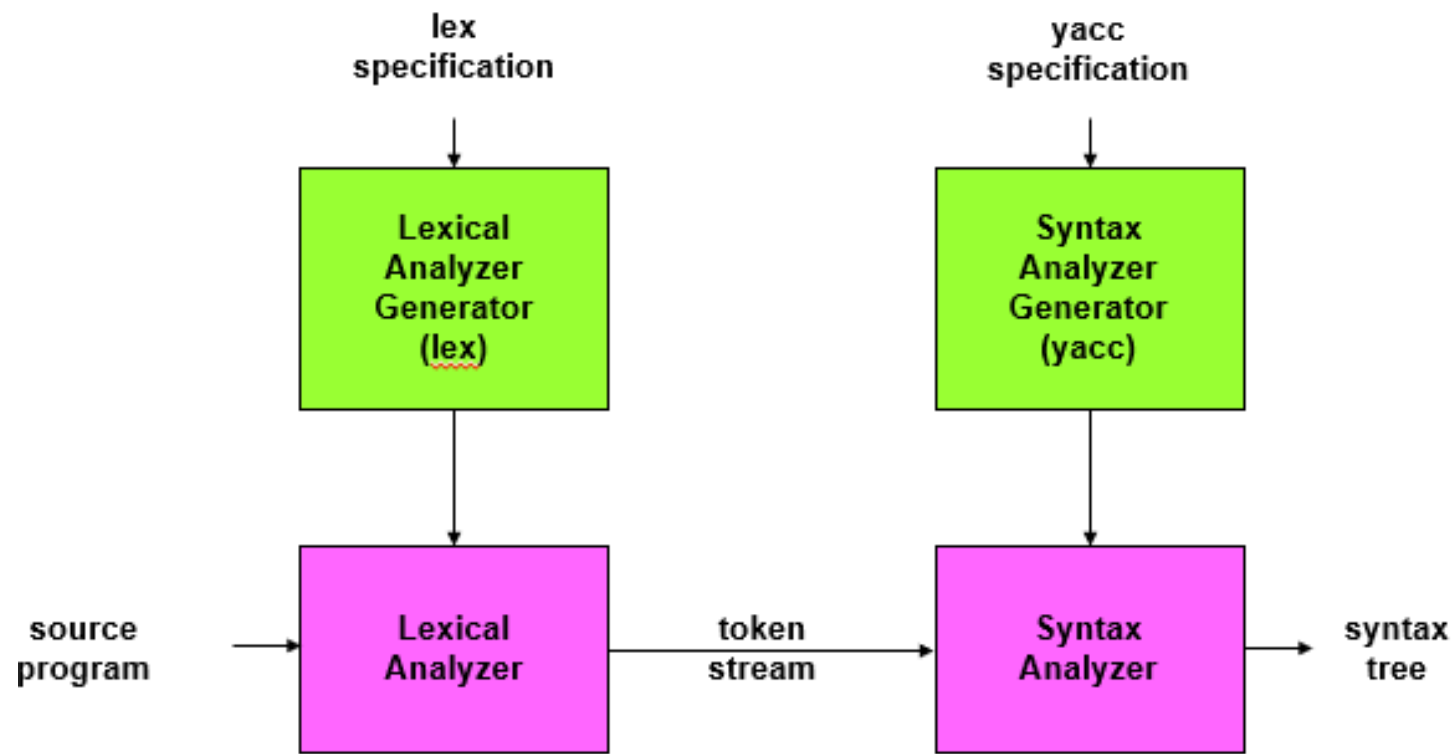
1. The **storage** allocated for a name
2. The **type** associated with a variable name
3. The **scope of a name** (where in the program its value may be used)
4. In the case of **procedure names**, it stores **the number and types of its arguments**, the **method of passing** each argument (for example, **by value or by reference**), and the **type returned**



# Compiler Construction Tools

Some commonly used compiler-construction tools include:

- 1. Scanner generators** that produce lexical analysers from a regular expression description of the tokens of a language.
- 2. Parser generators** that automatically produce syntax analysers from a grammatical description of a programming language.
- 3. Syntax-directed translation engines** that produce collections of routines for walking a parse tree and generating intermediate code.
- 4. Code-generator** generators that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
- 5. Data flow analysis engines** that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data flow analysis is a key part of code optimization.
- 6. Compiler-construction toolkits** that provide an integrated set of routines for constructing various phases of a compiler.



# The Grouping of Phases

## Compiler *front* and *back ends*:

- Front end: *analysis (machine independent)*
- Back end: *synthesis (machine dependent)*

## Compiler *passes*:

A collection of phases is done only once (*single pass*) or multiple times (*multi pass*)

Single pass: usually requires everything to be defined before being used in source program

Multi pass: compiler may have to keep entire program representation in memory

# The Evaluation of Programming Languages

## Disadvantages of low level programming

- It was slow
- Tedious
- Error prone.
- Once written, the programs were hard to understand and modify.

# The Move to Higher-level Languages

First-generation languages are the **machine languages**,

Second-generation the **assembly languages**,

Third-generation the higher-level languages like **Fortran, Cobol, Lisp, C, C++, C#, and Java**.

Fourth-generation languages are languages designed for **specific applications like NOMAD for report generation, SQL for database queries**

Fifth-generation language has been **applied to logic- and constraint-based languages like Prolog and OPS5**.

# Classification of languages

**Imperative:** for languages in which a program specifies how a computation is to be done.

Languages such as **C, C++, C#, and Java** are imperative languages

**Declarative:** for languages in which a program specifies what computation is to be done.

Prolog are often considered to be declarative languages.

**Von Neumann language:** is applied to programming languages whose computational model is based on the von Neumann computer architecture.

Fortran and C are von Neumann languages.

**Object-oriented language:** It is one that supports object-oriented programming, a programming style in which a program consists of a collection of objects that interact with one another.

Simula 67 and Smalltalk C++, C#, Java, and Ruby

**Scripting languages:** These are interpreted languages with high-level operators designed for "gluing together" computations.

Awk, JavaScript, Perl, PHP, Python, Ruby, and Tcl

# The science of building compiler

## Modeling in Compiler Design and Implementation

### Some of most fundamental models are :

**Finite-state machines and regular expressions:** These models are useful for describing the lexical units of programs (keywords, identifiers etc) and for describing the algorithms used by the compiler to recognize those units.

**Context-free grammars:** used to describe the syntactic structure of programming languages such as the nesting of parentheses or control constructs.

**Trees** are an important model for representing the structure of programs and their translation into object code

# The Science of Code Optimization

"optimization" in compiler design - compiler makes to produce code that is more efficient than the normal code

Compiler optimizations must meet the following design objectives:

- **The optimization must be correct:** The compiler should preserve the meaning of the compiled program
- **The optimization must improve the performance of many programs:** speed, minimize the size of the generated code, minimizes power consumption
- **The compilation time must be kept reasonable:** need to keep the compilation time short to support a rapid development and debugging cycle
- **The engineering effort required must be manageable:** keep the system simple so that the engineering and maintenance costs of the compiler are manageable



# Applications of compiler technology.

## 1. Implementation of High-Level Programming Languages

A high-level programming language defines a programming abstraction:

The programmer expresses an algorithm using the language, and the compiler must translate that program to the target language.

- *Java language-compared with C++/C*
- The **Java language is type-safe**; that is, an object cannot be used as an object of an unrelated type.
- All **array accesses are checked** to ensure that they lie **within the bounds of the array**.
- Java has **no pointers** and does not **allow pointer arithmetic**.
- It has a **built-in garbage-collection** facility that **automatically frees the memory of variables** that are no longer in use.

## 2. Optimizations for Computer Architectures

### 1.Parallelism : -several levels:

at the instruction level, where multiple operations are executed simultaneously (SISD-Single instruction stream, single data stream, SIMD-Single instruction stream, multiple data stream, MIMD-multiple instruction stream, multiple data stream)

at the processor level, where different threads of the same application are run on different processors.

### 2.Memory hierarchies:

very fast storage or very large storage, but not storage that is both fast and large.

### 3 . Design of New Computer Architectures

One of the best known examples of how compilers influenced the design of computer architecture was the invention of the **RISC** (*Reduced Instruction-Set Computer*) architecture. Prior to this invention, the trend was to develop progressively complex instruction sets intended to make assembly programming easier; these architectures were known as **CISC** (*Complex Instruction-Set Computer*).

RISC	CISC
RISC (Reduced Instruction-Set Computer) architecture- Most general purpose processor architectures -PowerPC, SPARC, MIPS, Alpha, and PA-RISC	CISC (Complex Instruction-Set Computer). -x86 architecture, AMD CPU's
Fewer and Simple instructions	Large amount of different and complex instructions
Faster execution of instructions	CISC chips are relatively slow per instruction, but use little (less than RISC) instructions
Easier to design	Difficult to design



**Over the last three decades, many architectural concepts have been proposed. They include**

- Data flow machines
- Vector machines
- VLIW (Very Long Instruction Word)
- Systolic arrays
- Multiprocessors with shared memory
- Multiprocessors with distributed memory.

## 4 . Program Translations

### Binary Translation

Compiler technology can be used to **translate the binary code for one machine to that of another**, allowing a machine to run programs originally compiled for another instruction set.

### Hardware Synthesis

Not only is most software written in high-level languages; even hardware designs are mostly described in high-level hardware description languages like **Verilog and VHDL** (Very high-speed integrated circuit Hardware Description Language).

### Database Query Interpreters

Besides specifying software and hardware, languages are useful in many other applications. For example, query languages, especially SQL (Structured Query Language) are used to search databases. Database queries consist of predicates containing relational and boolean operators. They can be interpreted or compiled into commands to search a database for records satisfying that predicate.

## Compiled Simulation

Instead of writing a simulator that interprets the design, it is faster to **compile the design to produce machine code that simulates that particular design natively**. Compiled simulation can run orders of magnitude faster than an interpreter-based approach.

## 5. Software Productivity Tools

### Type Checking

Type checking is an effective and well-established technique to catch inconsistencies in programs. It can be used to catch errors, for example, where an operation is applied to the wrong type of object, or if parameters passed to a procedure do not match the signature of the procedure. Program analysis can go beyond finding type errors by analysing the flow of data through a program.

For example, if a pointer is assigned null and then immediately dereferenced, the program is clearly in error.

### Bounds Checking

It is easier to make mistakes when programming in a lower-level language than a higher-level one. For example, many security breaches in systems are caused by buffer overflows in programs written in C. Because C does not have array- bounds checks, it is up to the user to ensure that the arrays are not accessed out of bounds.



## Memory-Management Tools

Garbage collection is another excellent example of the trade-off between efficiency and a combination of ease of programming and software reliability. Automatic memory management obliterates all memory-management errors (e.g., memory leaks"), which are a major source of problems in C and C++ programs.

Various tools have been developed to help programmers find memory management errors. For example, Purify is a widely used tool that dynamically catches memory management errors as they occur.

**Exercise:** Convert the given input statement into machine code through various phases of Compiler. Show output of each phase. Assume that a,b,c are having floating point values.

**$a = a + b * c * 2$**

input-statement-

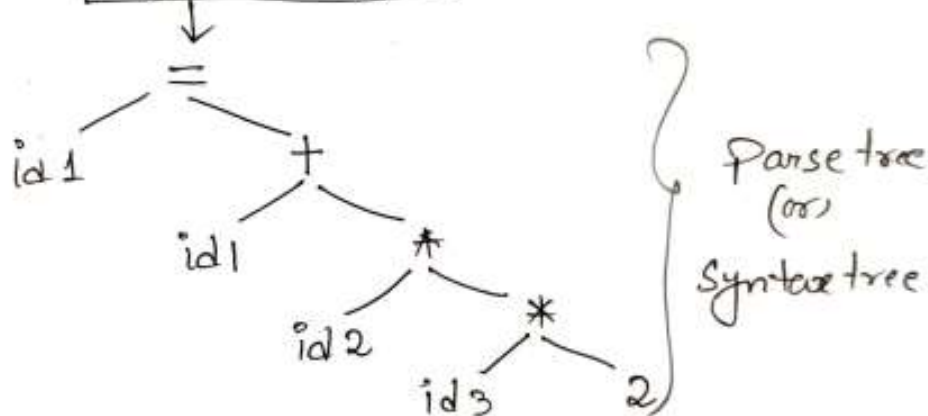
$a = a + b * c * 2$

Lexical Analysis

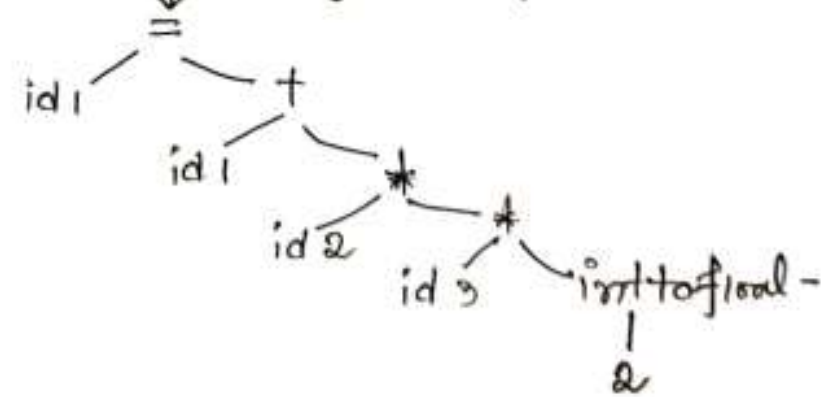
$id1 = id1 + id2 * id3 * 2$

} Tokens  
(or)  
Lexemes.

Syntax Analysis



Semantic Analysis



↓  
Intermediate  
code generation

↓  
 $t1 = \text{inttofinal} - (2)$   
 $t2 = \text{id3} * t1$   
 $t3 = \text{id2} * t2$   
 $t4 = \text{id1} + t3$   
 $\text{id1} = t4$

↓  
Code Optimization

↓  
 $t1 = \text{id3} * 2.0$   
 $t2 = \text{id2} * t1$   
 $\text{id1} = \text{id1} + t2$

→ 3-address  
Code format

→ In RHS max  
1 operator  
can be there

→ Max 3 operands  
can be there  
in each  
statement.

↓  
Code Generation

↓  
LDF R3, id3  
MULF R3, R3, #60.0  
LDF R2, id2  
MULF R2, R2, R3  
LDF R1, id1  
ADDF R1, R1, R2  
STF id1, R1

Machine code

# Chapter 2: Lexical Analysis

## The role of lexical analyzer

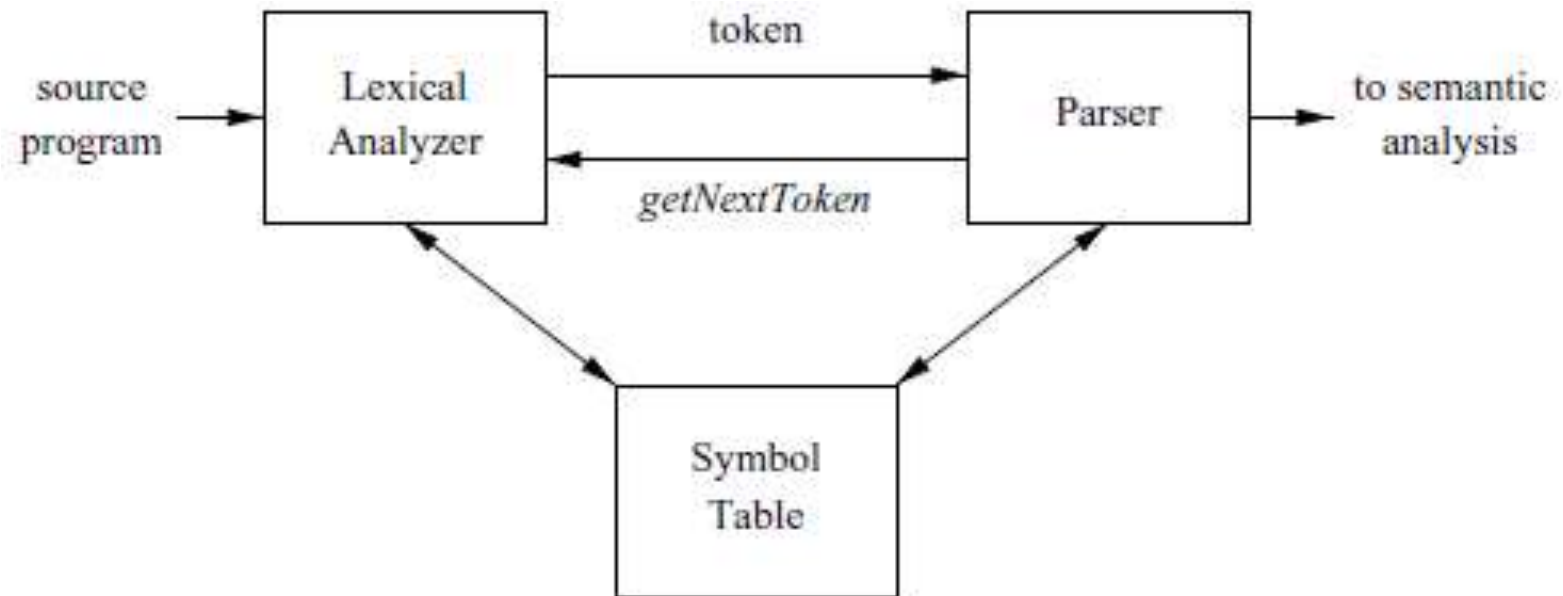


Figure 3.1: Interactions between the lexical analyzer and the parser

## Main tasks of lexical analysis

- Read the input characters of the source program
- Group input characters into lexemes
- Produce as output a sequence of tokens for each lexeme in the source program

The stream of tokens is sent to the parser for syntax analysis.

- Enter lexeme into symbol table or get information about identifier to pass to parser
- ***getNextToken()*** command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

## Other tasks of lexical analysis

- Stripping out comments and *whitespace* (blank, newline, tab, and other characters that are used to separate tokens in the input).
- Correlating error messages generated by the compiler with the source program.
  - ✓ by associate a line number with each error message.
  - ✓ by makes a copy of the source program with the error messages inserted at the appropriate positions.
- If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.
- The lexical analyzer may keep track of the number of newline characters seen



Lexical analyzers are divided into a cascade of two processes:

***Scanning*** consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

***Lexical analysis*** proper is the more complex portion, which produces the sequence of tokens from the output of scanner.



## why to separate Lexical Analysis from Parsing

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

- The separation of lexical and syntactic analysis often allows us to **simplify** at least one of these **tasks**. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.
- **Compiler efficiency is improved.** A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
- **Compiler portability is enhanced.** Input-device-specific peculiarities can be restricted to the lexical analyzer.



## Tokens, Patterns, and Lexemes

A **token** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

A **pattern** is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

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.

## Tokens, Patterns, and Lexemes (2)

- A **token** is a pair a token name and an optional token value  
ex: keyword, identifier.-if else and num1,num2
- A **pattern** is a description of the form that the lexemes of a token may take

Ex: identifier: ([a-z][A-Z]) ([a-z][A-Z][0-9])\*

- A **lexeme** is a sequence of characters in the source program that matches the pattern for a token


Figure 3.2 gives some typical tokens, their informally described patterns, and some sample lexemes. To see how these concepts are used in practice, in the C statement

ex: `printf("total = %d\n", score);`

both **printf** and **score** are lexemes matching the pattern for token **id**, and **"Total = %d\n"** is a lexeme matching **literal**.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but ", surrounded by "'s	"core dumped"

Figure 3.2: Examples of tokens



In many programming languages, the following classes cover most or all of the tokens:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token comparison mentioned in Fig. 3.2.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

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;
    }
}
```

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 created

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

## Attributes for Tokens

When more than one lexeme can match a pattern, the lexical analyzer must provide additional information to the subsequent compiler phases about the particular lexeme that matched. During the parsing stage, the compiler will only be concerned with tokens. Any integer constant, for example, is treated like any other. But during later processing, it will certainly be important just which constant was written. To deal with that, a token that can have many associated lexemes has an attribute, which can be the lexeme if you like. ***During semantic processing, the compiler examines the token attributes.***

For example, the pattern for token- **number** matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the token - **id**, where we need to associate with the token a great deal of information. Normally, information about an identifier : e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) , is kept in the **symbol table**. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.



## Lexical errors

**A character sequence which is not possible to scan into any valid token is a lexical error.** It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For instance, if the string `fi` is encountered for the first time in a C program in the context:

***fi ( a == f(x)) ...***

a lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an undeclared function identifier. Since ***fi*** is a valid lexeme for the token ***id***, the lexical analyzer must return the token `id` to the parser and let some other phase of the compiler probably the parser in this case handle an error due to transposition of the letters.

However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. The simplest recovery strategy is : **panic mode recovery**.

In **panic mode recovery**, we delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.



## Error Recovery- But time consuming

- **Panic mode recovery** : successive characters are ignored until find delimiter such as semi-colon
- **Statement mode recovery** : it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead.

*For example,*

- inserting a missing semicolon, replacing comma with a semicolon, Delete one character from the remaining input, Transpose two adjacent characters, Replace a character by another character etc
- **Error productions recovery** - adding error production

## Input buffering

To ensure that a right lexeme is found, often one or more characters have to be looked up beyond the next lexeme. there are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for **id**. In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=.

- Thus, we shall introduce a **two-buffer scheme** that handles large lookaheads safely.
- We then consider an improvement involving **sentinels** that saves time checking for the ends of buffers.

## why Buffer pairs?

- **Time** taken to process **a character by character** from left to right-time consuming
- Buffering techniques introduced – **2 buffer of N size** normally – each 4096 bytes
- Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file

**Ex:  $E = M * C ** 2$**

E	:	=	:	M	:	*	:	eof	C	:	*	:	*	:	2	:	eof	:	eof
---	---	---	---	---	---	---	---	-----	---	---	---	---	---	---	---	---	-----	---	-----

## Buffer Pairs

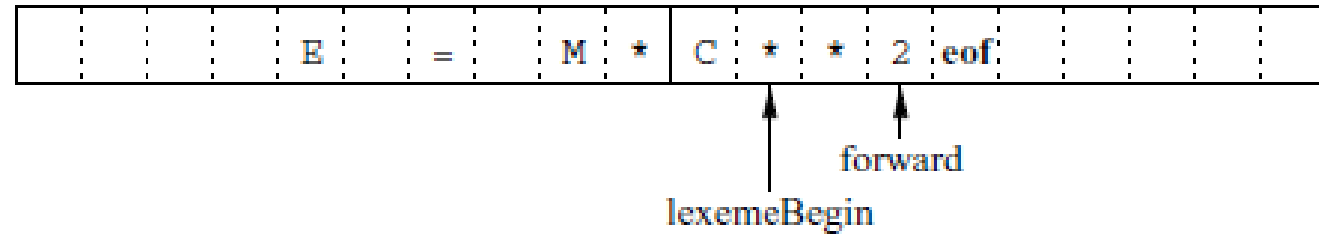


Figure 3.3: Using a pair of input buffers

### Two pointers to the input are maintained:

1. Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer **forward** scans ahead until a pattern match is found;

Once the next lexeme is determined, **forward** is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, **lexemeBegin** is set to the character immediately after the lexeme just found.

In the **buffer pair scheme** each time we advance forward i.e, for each character read, we make **two tests**:

- For the end of the buffer, and
- To determine what character is read.

### Using sentinel:

combine the **buffer-end test with the test for the current character** if we extend each buffer to hold a sentinel(**eof**) character at the end.

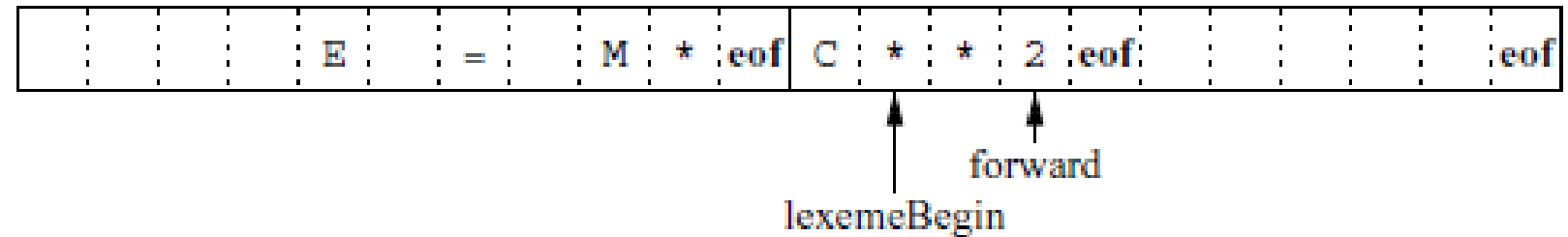
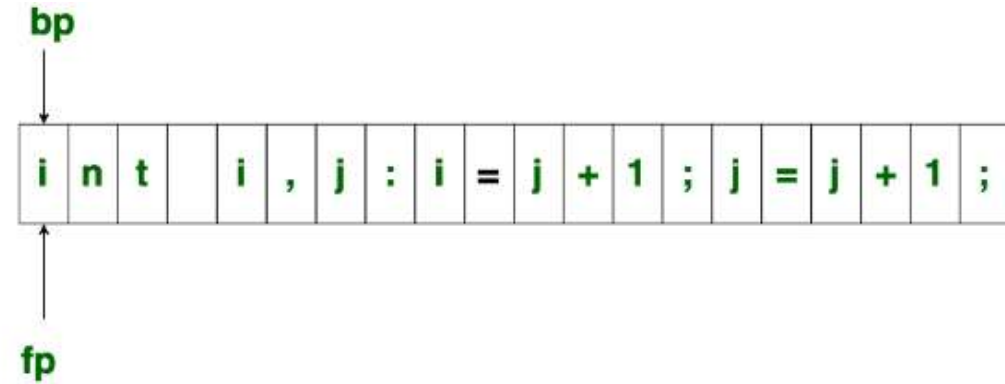


Figure 3.4: Sentinels at the end of each buffer

Figure 3.4 shows the same arrangement as Fig. 3.3, but with the sentinels added. Note that **eof** retains its use as a marker for the end of the entire input. Any eof that appears other than at the end of a buffer means that the input is at an end.

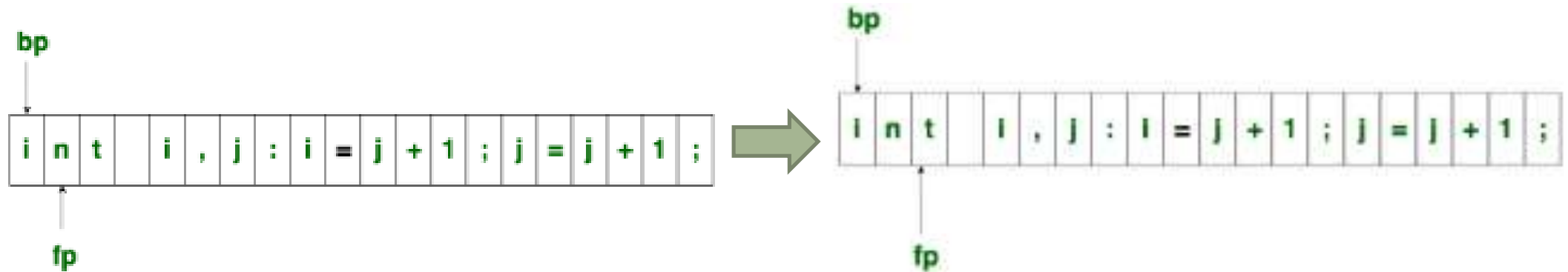
## Example of Input Buffering

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers lexemeBegin pointer (bp) and forward pointer (fp) to keep track of the pointer of the input scanned.

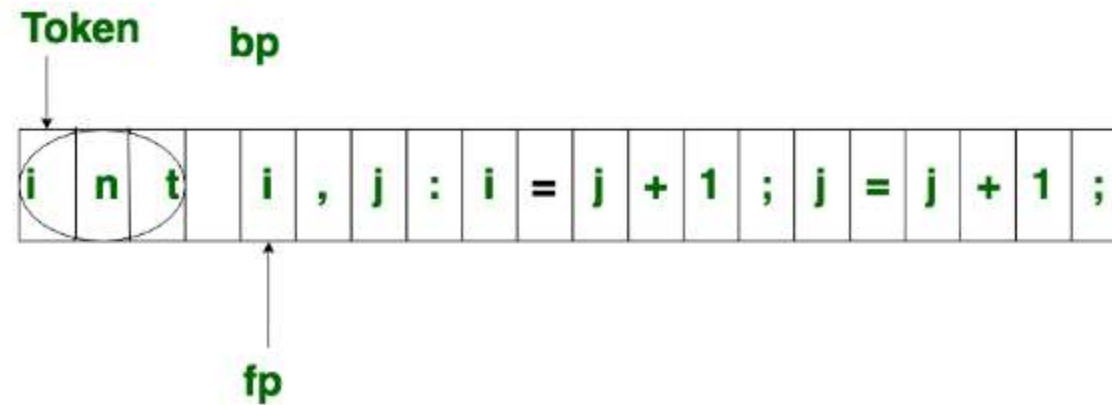


Initial Configuration

Initially both the pointers point to the first character of the input string as shown below

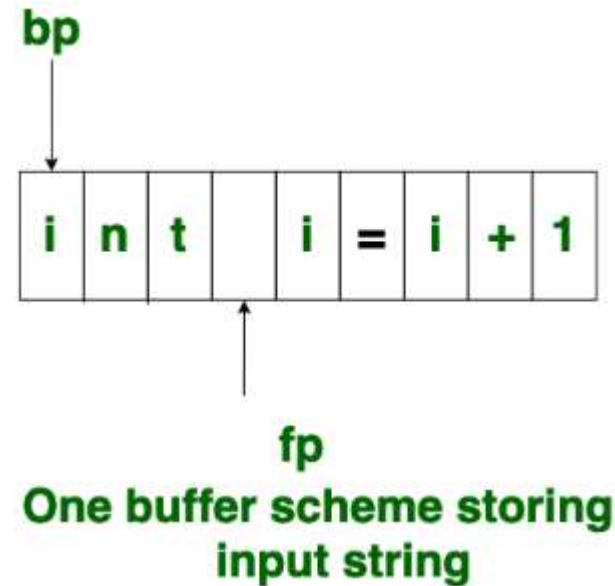


The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme “int” is identified. The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token.



**Two methods used in this context: One Buffer Scheme, and Two Buffer Scheme.**

**One Buffer Scheme:** In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.

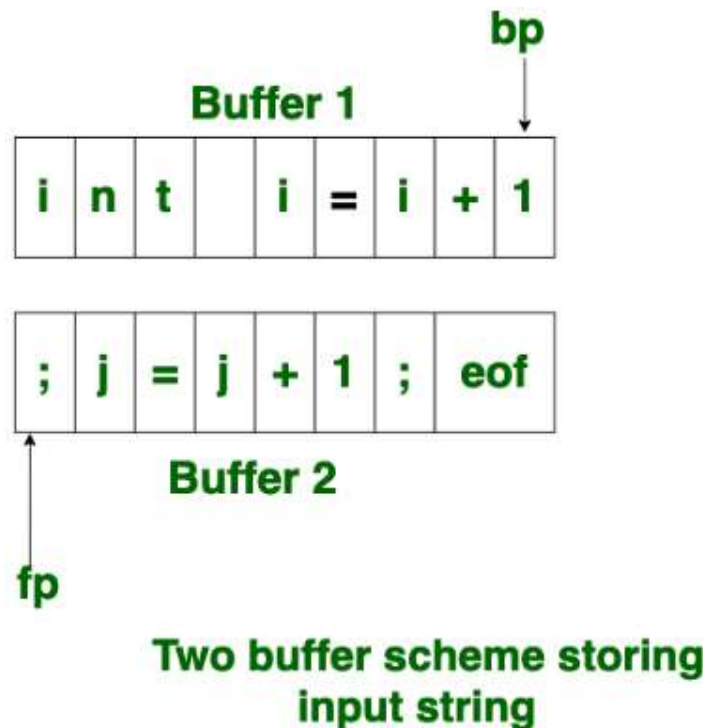


**Two Buffer Scheme:** To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. the first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled. the only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely. Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right in search of end of lexeme. as soon as blank character is recognized, the string between bp and fp is identified as corresponding token.



To identify the boundary of first buffer end of buffer character should be placed at the end first buffer. Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. when fp encounters first **eof**, then one can recognize end of first buffer and hence filling up second buffer is started. in the same way when second **eof** is obtained then it indicates of second buffer. Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified.

This eof character introduced at the end is called **Sentinel** which is used to identify the end of buffer.



**Figure 3.5 summarizes the algorithm for advancing forward.**

```
Switch (*forward++)
{ case eof:
  if (forward is at end of first buffer) { reload second buffer;
  forward = beginning of second buffer;
  }
  else if {forward is at end of second buffer) { reload first buffer;\
  forward = beginning of first buffer;
  }
  else /* eof within a buffer marks the end of input */ terminate lexical analysis;
  break;
  cases for the other characters;
}
```

**Figure 3.5: Lookahead code with sentinels**

## Specifications of token

**There are 3 specifications of tokens:**

- 1)Strings
- 2) Language
- 3)Regular expression

**Regular expressions are a notation to represent lexeme patterns for a token.**

- They are used to represent the language for lexical analyzer.
- They assist in finding the type of token that accounts for a particular lexeme.

## Strings and Languages:

**Alphabets** are finite, non-empty set of input symbols.

$$\Sigma = \{0, 1\} \text{ – binary alphabets}$$

**String** represents the collection of alphabets.

**Language (L)** is the collection of strings .

$$w = \{0, 1, 00, 01, 10, 11, 001, 010, \dots\}$$

w indicates the set of possible strings for the given binary alphabet  $\Sigma$

Length of string is defined as the number of input symbols in a given string. It is found by  $||$  operator.

$$\text{Let } \omega = 0101$$

$$| \omega | = 4$$

Empty string denotes zero occurrence of input symbol. It is represented by  $\epsilon$ . Concatenation of two strings  $p$  and  $q$  is denoted by  $pq$ .

Let  $p = 010$

And  $q = 001$

$pq = 010001$

$qp = 001010$

i.e.,  $pq \neq qp$

Empty string is identity under concatenation.

Let  $x$  be a string.

$Ex = XE = X$

**Prefix** A prefix of any string  $s$ , is obtained by removing zero or more symbols from the end of  $s$ .

(eg.)  $s = \text{balloon}$

Possible prefixes are:  $\epsilon$ , balloon, ball

**Suffix** A suffix of any string  $s$ , is obtained by removing zero or more symbols from the beginning of  $s$ .

(eg.)  $s = \text{balloon}$

Possible prefixes are:  $\epsilon$ , balloon, loon

**Proper prefix:** Proper prefix  $p$  of a strings, can be given by  $s \neq p$  and  $p \neq \epsilon$   
We can obtain proper prefix by removing  $\epsilon$  and the string itself.

**Proper suffix:** Proper suffix  $x$  of a string  $s$ , can be given by  $s \neq x$  and  $x \neq \epsilon$   
We can obtain proper suffix by removing  $\epsilon$  and the string itself.

**Substring:** Substring is part of a string obtained by removing any prefix and any suffix from  $s$ .

## Precedence of operators

- Unary operator (\*) is having highest precedence.
- Concatenation operator (-) is second highest and is left associative.

letter\_ (letter\_ | digit )\*

- Union operator ( | or U) has least precedence and is left associative.

## Operations on Languages

Important operations on a language are:

- Union
- Concatenation and
- Closure

### Union

Union of two languages  $L$  and  $M$  produces the set of strings which may be either in language  $L$  or in language  $M$  or in both. It can be denoted as,  
 $L \cup M = \{p \mid p \text{ is in } L \text{ or } p \text{ is in } M\}$

### Concatenation

Concatenation of two languages  $L$  and  $M$ , produces a set of strings which are formed by merging the strings in  $L$  with strings in  $M$  (strings in  $L$  must be followed by strings in  $M$ ). It can be represented as,  
 $LM = \{pq \mid p \text{ is in } L \text{ and } q \text{ is in } M\}$

## Kleene closure ( $L^*$ )

Kleene closure refers to zero or more occurrences of input symbols in a string, i.e., it includes empty string  $\epsilon$  (set of strings with 0 or more occurrences of input symbols).

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

## Positive closure ( $L^+$ )

Positive closure indicates one or more occurrences of input symbols in a string, i.e., it excludes empty string  $\epsilon$  (set of strings with 1 or more occurrences of input symbols).

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

$L^3$  - set of strings each with length 3.

(eg.) Let  $\Sigma = \{a, b\}$

$L^* = \{\epsilon, a, b, aa, ab, ba, bb, aab, aba, aaba, \dots\}$

$L^+ = \{a, b, aa, ab, ba, bb, aab, aaba, \dots\}$

$L^3 = \{aaa, aba, abb, bba, bab, bbb, \dots\}$



## Regular Expressions

**Regular expressions** is used for describing all the languages that can be built from operators (union concatenation and closure) applied to the symbols of some alphabet.

if *letter*- is established to stand for any letter or the underscore, and *digit*- is established to stand for any digit,

Then we could describe the language of **C identifiers** by:

**letter\_ ( letter\_ | digit )\***

The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of," and letter\_ placed with the remainder of the expression signifies concatenation.

## Languages for regular expressions

Regular expression	Language
$r$	$L(r)$
$a$	$L(a)$
$r \mid s$	$L(r) \mid L(s)$
$rs$	$L(r) L(s)$
$r^*$	$(L(r))^*$

**Example** : Let  $\Sigma = \{a, b\}$ .

- The regular expression  $a \mid b$  denotes the language  $\{a, b\}$ .
- $(a \mid b)(a \mid b)$  denotes  $\{aa, ab, ba, bb\}$ , the language of all strings of length two over the alphabet  $\Sigma$ . Another regular expression for the same language is  $aa \mid ab \mid ba \mid bb$ .
- $a^*$  denotes the language consisting of all strings of zero or more a's, that is,  
 $\{\epsilon, a, aa, aaa, \dots\}$
- $(a \mid b)^*$  denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's:  $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$ . Another regular expression for the same language is  $(a^*b^*)^*$ .
- $a|a^*b$  denotes the language  $\{a, b, ab, aab, aaab, \dots\}$ , that is, the string a and all strings consisting of zero or more a's and ending in b

A language that can be defined by a regular expression is called a **regular set**. If two regular expressions  $r$  and  $s$  denote the same regular set, we say they are **equivalent** and write  $r = s$ . Example  $(a|b) = (b|a)$ .

## Algebraic laws of regular expressions

Law	Description
$r \mid s = s \mid r$	$\mid$ is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s \mid t) = rs \mid rt; (s \mid t)r = sr \mid tr$	Concatenation is distributive
$\varepsilon r = r\varepsilon = r$	$\varepsilon$ is identity for concatenation
$r^* = (r \mid \varepsilon)^*$	$\varepsilon$ is guaranteed in closure
$r^{**} = r^*$	$*$ is idempotent

## Representing valid tokens of a language in regular expression

If  $x$  is a regular expression, then:

- $x^*$  means zero or more occurrence of  $x$ .  
i.e., it can generate  $\{ e, x, xx, xxx, xxxx, \dots \}$
- $x^+$  means one or more occurrence of  $x$ .  
i.e., it can generate  $\{ x, xx, xxx, xxxx \dots \}$  or  $x.x^*$
- $x^?$  means at most one occurrence of  $x$   
i.e., it can generate either  $\{x\}$  or  $\{e\}$ .

$[a-z]$  is all lower-case alphabets of English language.

$[A-Z]$  is all upper-case alphabets of English language.

$[0-9]$  is all natural digits used in mathematics.

## Representing occurrence of symbols using regular expressions

letter = [a – z] or [A – Z]

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]

sign = [ + | - ]

Representing language tokens using regular expressions

Decimal = (sign)?(digit)+

Identifier = letter\_ ( letter\_ | digit )\* //in C Language

Identifier = letter( letter | digit )\* //in Pascal Language

## Recognition of Tokens.

In this topic we will see how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns

<i>stmt</i>	→	if <i>expr</i> then <i>stmt</i>
		if <i>expr</i> then <i>stmt</i> else <i>stmt</i>
		ε
<i>expr</i>	→	<i>term</i> relop <i>term</i>
		<i>term</i>
<i>term</i>	→	id
		number

Figure 3.10: A grammar for branching statements

The grammar fragment of Fig. 3.10 describes a simple form of branching statements and conditional expressions. This syntax is similar to that of the language Pascal, in that then appears explicitly after conditions

For relop, we use the comparison operators of languages like Pascal or SQL, where = is \equals" and <> is \not equals,"

The terminals of the grammar, which are **if**, **then**, **else**, **relop**, **id**, and **number**, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions, as in Fig. 3.11.

**ws -> ( blank | tab | newline )+**

Here, blank, tab, and newline are abstract symbols that we use to express the ASCII characters of the same names. Token **ws** (**word space**) is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.

<i>digit</i>	→	[0-9]
<i>digits</i>	→	<i>digit</i> <sup>+</sup>
<i>number</i>	→	<i>digits</i> ( . <i>digits</i> ) ? ( E [ + - ] ? <i>digits</i> ) ?
<i>letter</i>	→	[A-Za-z]
<i>id</i>	→	<i>letter</i> ( <i>letter</i>   <i>digit</i> ) *
<i>if</i>	→	if
<i>then</i>	→	then
<i>else</i>	→	else
<i>relop</i>	→	<   >   <=   >=   =   <>

Figure 3.11: Patterns for tokens



Our goal for the lexical analyzer is summarized in Fig. 3.12. The table shows for each lexeme or family of lexemes, which token name and what attribute value is returned to the parser.

Note that for the six relational operators, symbolic constants LT, LE, and so on are used as the attribute value, in order to indicate which instance of the token **relop** we have found.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any <i>ws</i>	--	--
<b>if</b>	<b>if</b>	--
<b>then</b>	<b>then</b>	--
<b>else</b>	<b>else</b>	--
Any <i>id</i>	<b>id</b>	Pointer to table entry
Any <i>number</i>	<b>number</b>	Pointer to table entry
<	<b>relop</b>	LT
<=	<b>relop</b>	LE
=	<b>relop</b>	EQ
<>	<b>relop</b>	NE
>	<b>relop</b>	GT
>=	<b>relop</b>	GE

Figure 3.12: Tokens, their patterns, and attribute values

**Transition diagrams**, a variant of finite state automata, are used to implement regular definitions and to recognize tokens.

- **Transition diagrams** have a collection of **nodes or circles**, called **states**. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of a state as summarizing all we need to know about what characters we have seen between the lexemeBegin pointer and the forward pointer
- **Edges** are directed from one state of the transition diagram to another. Each edge is labelled by a symbol or set of symbols. If we are in some state **S**, and the next input symbol is **a**, we look for an edge out of state **S** labelled by **a** (and perhaps by other symbols, as well). If we find such an edge, we advance the forward pointer and enter the state of the transition diagram to which that edge leads. We shall assume that all our transition diagrams are deterministic, meaning that there is never more than one edge out of a given state with a given symbol among its labels.
- ❑ Certain states are said to be **accepting**, or **final**. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the **lexemeBegin** and **forward** pointers . In Transition Diagram, we indicate an **accepting state** by a **double circle**.
- ❑ If it is necessary to retract ( take back) the forward pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a **\*** near that accepting state

# Regular Definition

- Giving names to regular expressions is referred to a **Regular definition**.
- If  $\Sigma$  is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$d1 \rightarrow r1$

$d2 \rightarrow r2$

.....

$dn \rightarrow rn$

where,

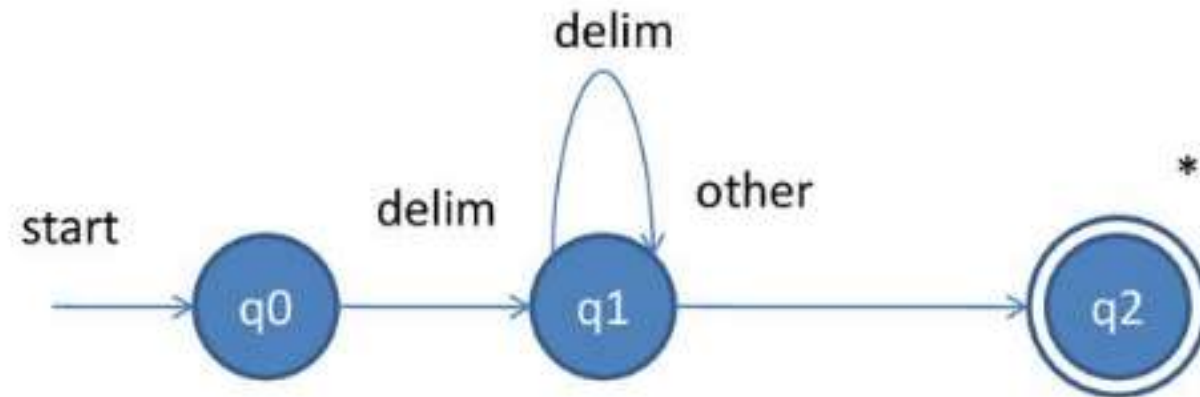
Each  $di$  is a new symbol, not in  $\Sigma$ .

Each  $ri$  is a regular expression.

## Example

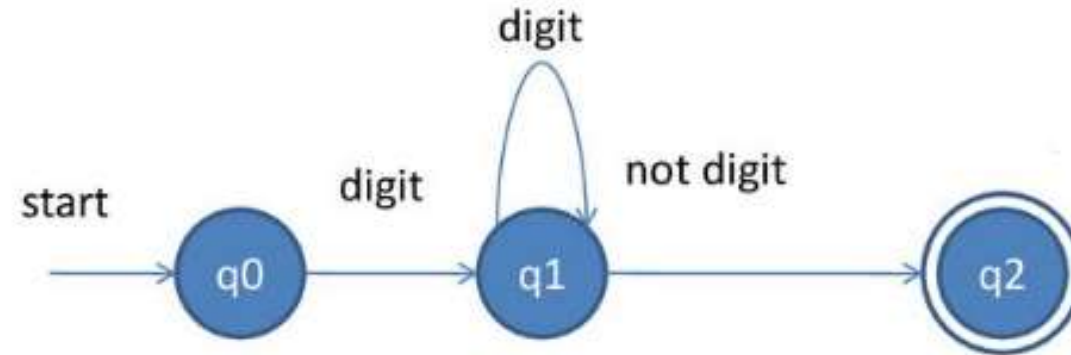
- Strings of letters, digits and underscores are C Identifiers. Regular definition for the language of C identifiers:
- $\text{letter} \rightarrow A|B|...|Z|a|b|...|z|_$
- $\text{digit} \rightarrow 0|1|...|9$
- $\text{id} \rightarrow \text{letter}(\text{letter}|\text{digit})^*$

## Transition diagram for whitespace



delim -> blank | new line | tab

## Transition diagram for constant/digits

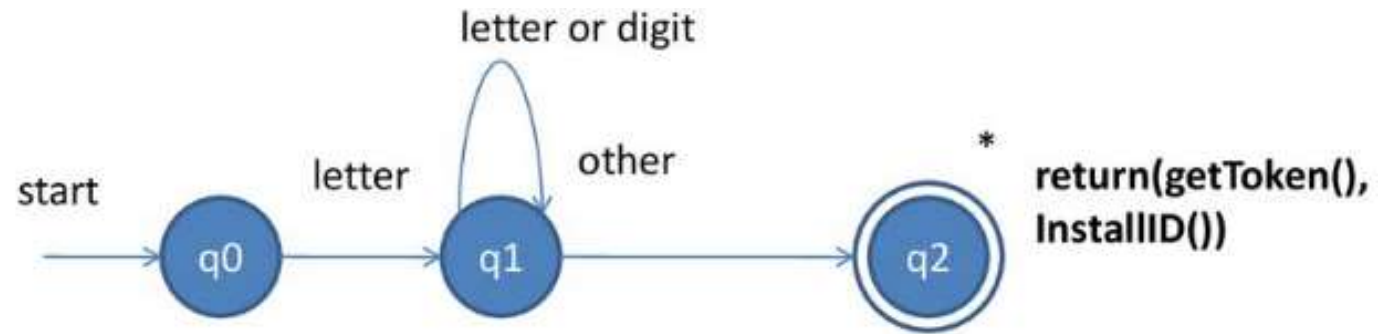


Pattern for constant:  
 $[0-9] ([0-9])^*$

Regular Definition:  
 $Id \rightarrow digit(digit)^*$

## Transition diagram for identifier and Keywords (*approach 1*)

Recognizing keywords and identifiers presents a problem. Usually, keywords like **if** or **then** are reserved, so they are not identifiers even though they look like identifiers. Thus, although we typically use a transition diagram like this to search for identifier lexemes, this diagram will also recognize the keywords if, then, and else.



Pattern for identifier:

`([a-z]|[A-Z]) ([a-z]|[A-Z]|[0-9])*`

Regular Definition:

`Id -> letter(letter | digit)*`

In this Figure, state q0 is the start state and state q2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state q1. We remain in state q1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state q2.

3-state machine can be easily programmed in following manner:

```
start:  goto state0

state0: read c
        if c = letter goto state1
        goto state0

state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2

state2: accept string
```

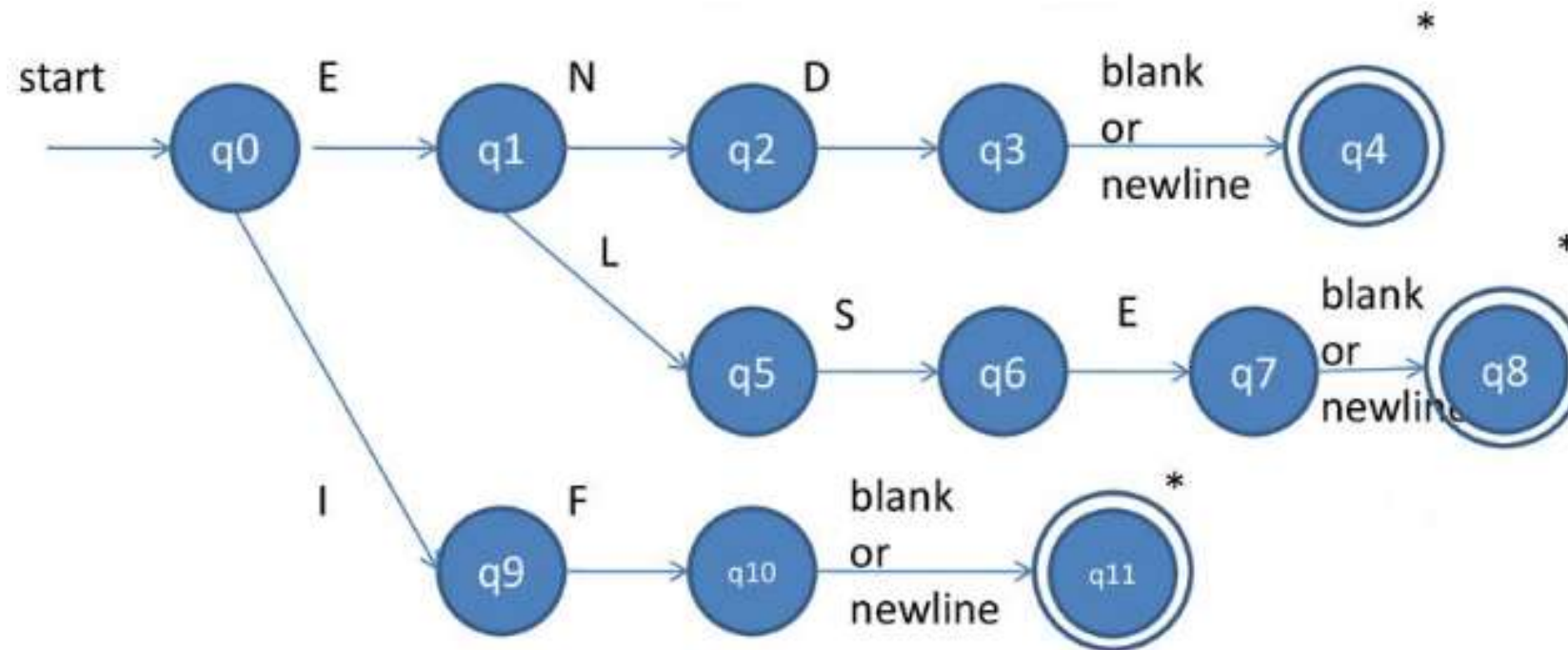


## There are two ways that we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in this Figure. When we find an identifier, a call to **installID()** places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id. The function **getToken()** examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents | either id or one of the keyword tokens that was initially installed in the table.
2. Create separate transition diagrams for each keyword; an example for the keywords-end, else, if is shown in approach-2 of recognizing a keyword



## Transition diagram for Keyword (approach 2)



Pattern for keywords:  
e( nd | lse ) | if

## Transition diagram for relop

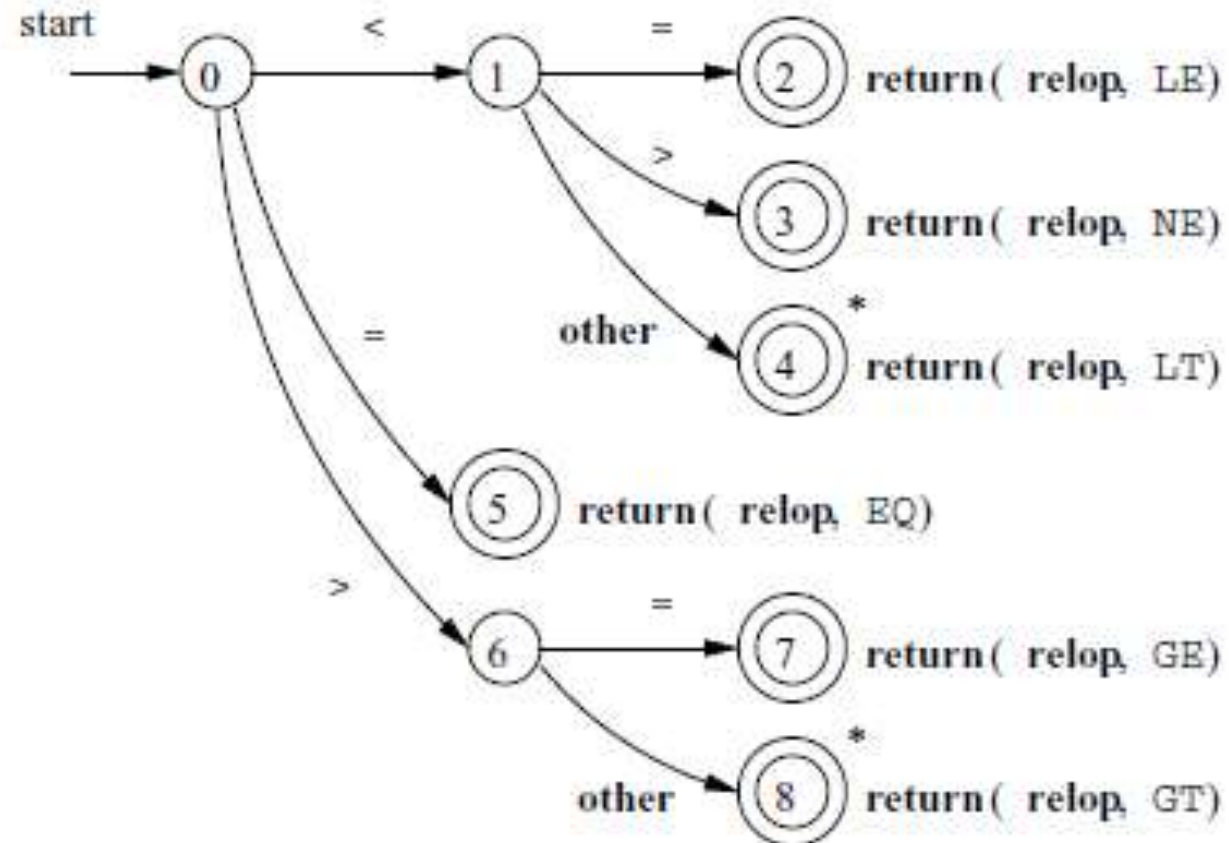


Figure 3.13: Transition diagram for `relop`

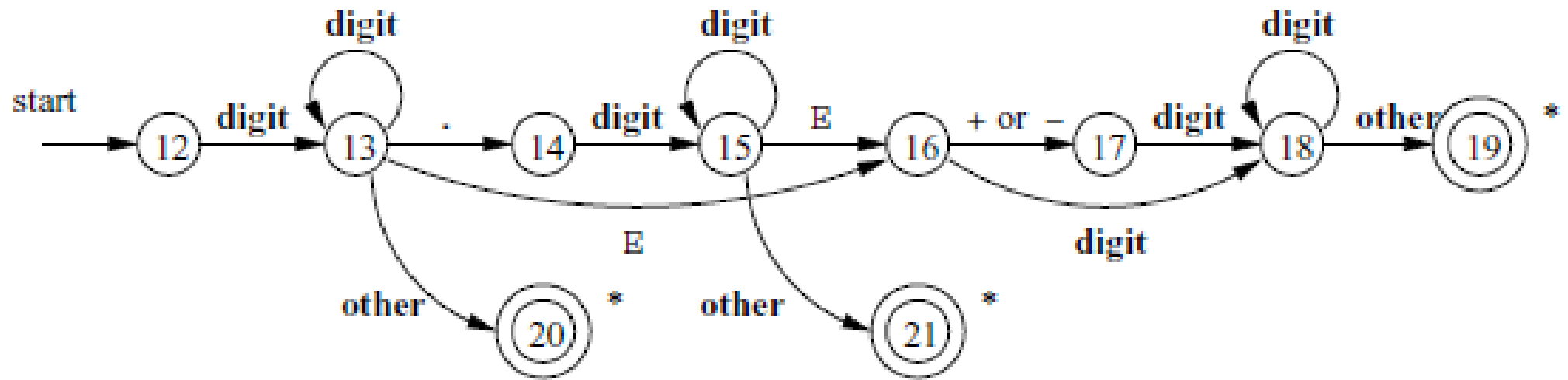
## Program Sketch of implementation of relop transition diagram

1. **getRelop()** first creates a new object **retToken** and initializes its first component to **RELOP**, the symbolic code for token **relop**.
2. A function **nextchar()** obtains the next **character from the input** and assigns it to **local variable c**
3. if the next input character is **=**, **we go to state 5**.
4. If the **next input character** is not one that can begin a **comparison operator**, then a function **fail()** is called. **fail ()** should **reset the forward pointer to lexemeBegin**, in order to **allow another transition diagram to be applied**.

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Figure 3.18: Sketch of implementation of **relop** transition diagram|

## Transition diagram for an unsigned number



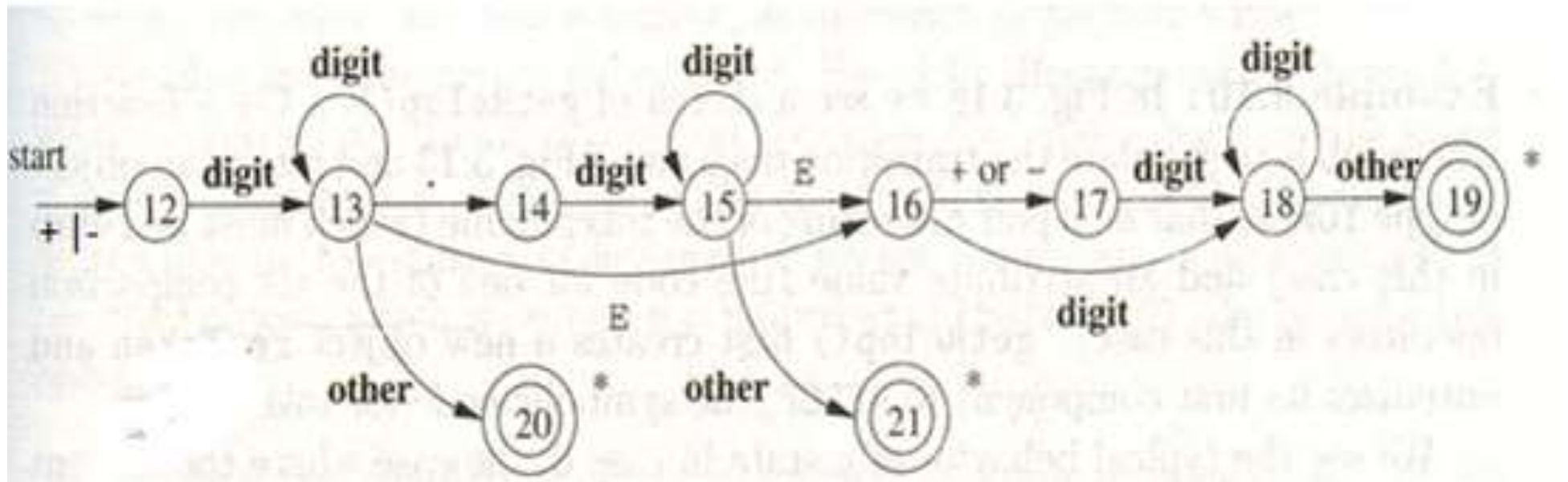
return(number, pointer to table)

Pattern for Unsigned Number:


$[0-9] ([0-9])^* (. [0-9] ([0-9])^*) ? (E [+|-]? [0-9] ([0-9])^*) ?$

Regular Definition for Unsigned number->  $\text{digits} (.\text{digits})? (E[+|-]? \text{digits})?$

## Transition diagram for a signed number



Regular Definition for signed number->  $[+-] ? \text{digits} (. \text{digits})? (E[+-]? \text{digits})?$



Beginning in state 12, if we see a digit, we go to state 13. In that state, we can read any number of additional digits. However, if we see anything but a digit, dot, or E, we have seen a number in the form of an integer; 123 is an example. That case is handled by entering state 20, where we return token number and a pointer to a table of constants where the found lexeme is entered.

If we instead see a dot in state 13, then we have an "optional fraction." State 14 is entered, and we look for one or more additional digits; state 15 is used for that purpose. If we see an E, then we have an "optional exponent," whose recognition is the job of states 16 through 19. Should we, in state 15, instead see anything but E or a digit, then we have come to the end of the fraction, there is no exponent, and we return the lexeme found, via state 21.



# Question Bank-Module-2

- 1) What is a language processor? What are the various types of language processors? *(Ans: slide: 12-16)*  
*(or) What is a compiler?*  
*(or) What is an interpreter? What are the differences between a compiler and interpreter*  
*(or) What is an assembler?*  
*(or) What is hybrid compiler?*  
*(or) What are the cousins of the compiler? or Explain language processing system*
- 2) With a block diagram explain different phases of compiler. *(Ans: slide: 21- 33)*  
*(or) Explain the block diagram of the compiler construction method*
- 3) Show the output of each phase of the compiler for the assignment statement:  
sum = initial + value \* 10 *(Ans: slide: 34)*  
*(or) Explain the various phases of a compiler with a neat diagram. Show the translations for an assignment statement  $I = P * T / 100 - C$ , clearly indicate the output of each phase*
- 4) What is the difference between a phase and pass? What is a multi-pass compiler?  
Explain the need for multiple passes in compiler? *(Ans: slide: 38)*
- 5) What are the various applications of compiler technology? *(Ans: slide: 44-51)*
- 6) What is lexical analysis? What is the role of lexical analyzer? *(Ans: slide: 54-56)*

7) Why analysis portion of the compiler is separated into lexical analysis and syntax analysis phase? *(Ans: slide: 58)*

8) What is Token, Lexeme and Pattern? Explain with an example. *(Ans: slide: 59-61)*

9) Identify lexemes and tokens in the following statement: `a= b * d;`

*(or) Identify lexemes and tokens in the following statement: `printf("Simple Interest = %f\n", si);`*

10) What are the various Error recovery mechanisms in Lexical analysis? List and explain them briefly. *(Ans: slide: 66-67)*

11) What is the need for returning attributes for tokens along with token name? Give the token names and attribute values for the following statement: `= M * C **2`, where `**` in FORTRAN language is used as exponent operator. *(Ans: slide: 65, Text Book Example)*

*(or) Give the token names and attribute values for the following statement: `si= p * t * r / 100`*

12) Explain input buffering techniques in lexical analysis and justify why is it important? Explain the use of sentinels in recognizing the tokens. *(Ans: slide: 68-76)*

13) Write the regular definition for relational operator. Also draw the transition diagram.

14) Write the regular definition for identifier/keywords. Also draw the transition diagram .

15) Write the regular definition for signed and unsigned number. Also draw the corresponding transition diagram.

16) Write the regular definition for word space. Also draw the transition diagram

17) Construct transition diagram for recognizing relation operator and sketch the program segment to implement it showing the first state and one final state

18) What is regular expression? Write algebraic laws of regular expression *(Ans: slide: 86)*

*(Ans:  
slide:  
89-103)*





# Thank You