Presidency University, Bengaluru

# CSE 217 – Compiler Design

# Introduction

# Instructors

Instructor-in-charge : **Dr. Islabudeen. M**

Instructors : **Mr. Sukruth Gowda M.A,**

**Mr. Prasad P.S,**

**Mr. Rahul Biswas,**

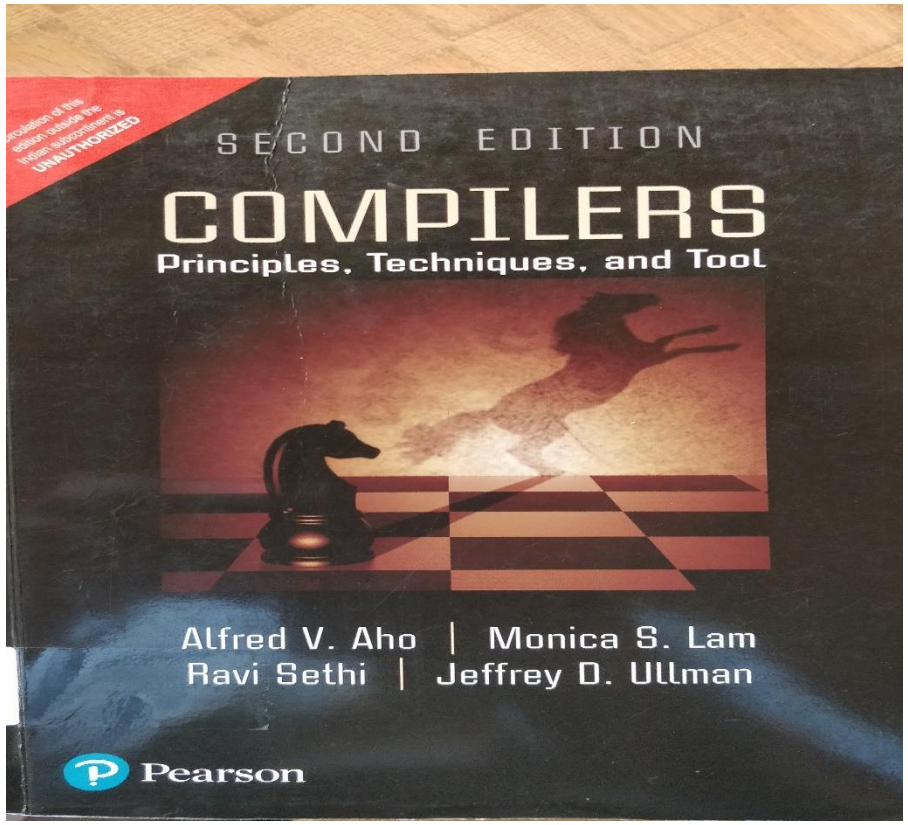
**Ms. Shivali Shukya**

**Mr. Shine V.J**

# Evaluation Components

| | Duration (minutes) | % Weightage | Marks | Date & Time |
|---|---|---|---|---|
| Test 1 | 60 min | 20 | 40 | To be announced by COE |
| Test 2 | 60 min | 20 | 40 | To be announced by COE |
| Quiz / Class Test | 60 min | 20 | 40 | To be announced by Later |
| Comprehensive exam | 180 min | 40 | 100 | To be announced by COE |

# Slides are Mostly From




**Text Books:**

1. Alfred V. Aho, Jeffrey D Ullman, “Compilers: Principles, Techniques and Tools”, Pearson second Edition, 2013.

**Reference Books:**

1. Jean Paul Tremblay, Paul G Serenson, "The Theory and Practice of Compiler Writing", BS Publications, 2005.
2. C. N. Fischer and R. J. LeBlanc, “Crafting a compiler with C”, Benjamin Cummings, 2003.
3. HenkAlblas and Albert Nymeyer, “Practice and Principles of Compiler Building with C”, PHI, 2001.
4. Kenneth C. Louden, “Compiler Construction: Principles and Practice”, Thompson Learning, 2003.
5. Dhamdhere, D. M., "Compiler Construction Principles and Practice", Macmillan India Ltd, 2008

# Course Content (Syllabus):

**Module I: INTRODUCTION AND LEXICAL ANALYSIS (13 hours) [COMPREHENSION]**

Compilers – Cousins of the Compiler - Phases of a compiler - Analysis of the source program - Grouping of phases – Compiler construction tools – Lexical Analysis – Role of the Lexical Analyzer – Input buffering – Specification of tokens – Recognizer - Introduction to LEX Programming.

**Module II: SYNTAX ANALYSIS (15 hours) [APPLICATION]**

Role of the parser - Top-down parsing - Recursive decent parser - Predictive parser - Bottom-up parsing – Shift reduce parser - LR parser – SLR parser – Canonical parser – LALR parser - YACC programming.

**Module III: SEMANTIC ANALYSIS AND INTERMEDIATE CODE GENERATION (14 hours) [APPLICATION]**

Introduction to syntax directed translation - Synthesis and inherited attributes - Type Checking - Type Conversions - Intermediate languages – Three address statements - Declarations – Assignment Statements – Boolean Expressions – Case Statements – Back patching – Looping statements - Procedure calls.

**Module IV: CODE OPTIMIZATION AND CODE GENERATION (12 hours) [COMPREHENSION]**

Basic Blocks and Flow Graphs – Principal sources of optimization – Peephole optimization - Optimization of basic Blocks - DAG representation of Basic Blocks - Issues in the design of code generator – A simple code generator.

# Course Outcomes:

**On successful completion of the course the students shall be able to:**

**CO1: Explain the various phases of compiler (COMPREHENSION)**

**CO2: Apply parsing techniques to check the syntax of given statement. (APPLICATION)**

**CO3: Produce intermediate code for the given statement. (APPLICATION)**

**CO4: Discuss how to optimize the given problem for back end of the compiler (COMPREHENSION)**

# Program Outcomes

**PO1 Engineering Knowledge:**
Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. [**High]**

**PO2 Problem Analysis:**
Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences. **[High]**

**PO3 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. (High]

**PO4 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**.[High**]

**PO5 Modern Tool usage:**
Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations. **[Moderate]**

**PO10 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. **[Low]**

# Mapping of CO with PO

| CO / PO N0. | PO1 | PO2 | PO3 | PO4 | PO5 | PO10 |
|---|---|---|---|---|---|---|
| CO1 | H | M | M | M | M | L |
| CO2 | H | H | H | H | M | L |
| CO3 | H | H | H | H | - | L |
| CO4 | H | H | M | M | - | L |

# Course Prerequisites

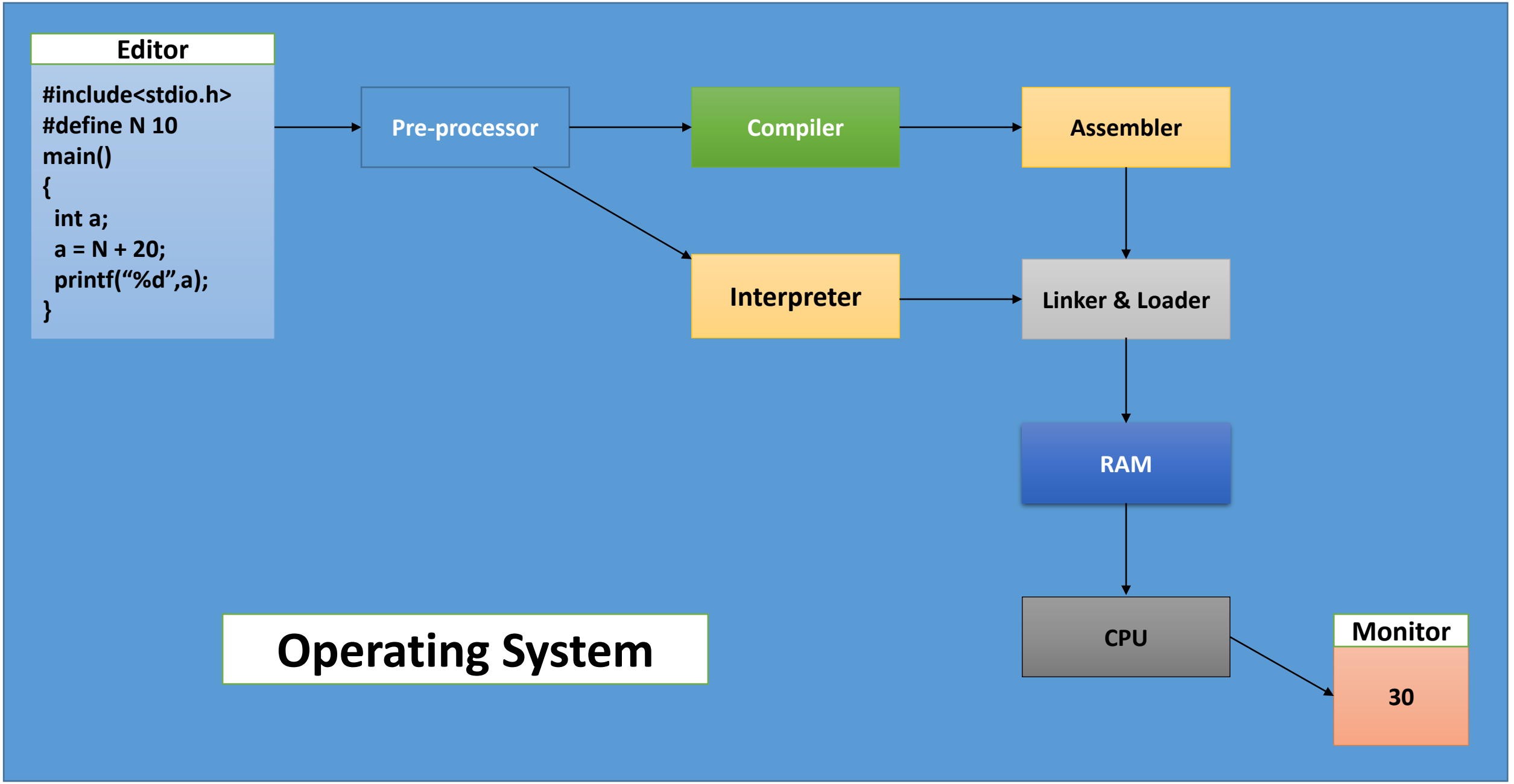
- Finite Automata, Context Free Grammar

  (CSE208 - Theory of Computation – CSE 5th Semester Course)

- System software

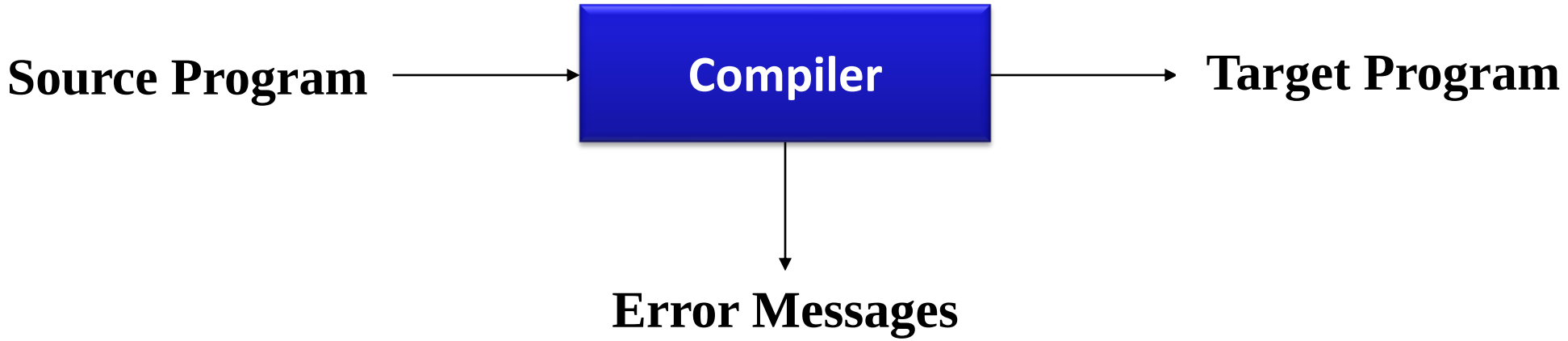
```
#include<stdio.h>
#define N 10
main()
{
  int a;
  a = N + 20;
  printf("%d",a);
}
```

| Monitor |
| --- |
| 30 |


Editor
#include<stdio.h>
#define N 10
main()
{
int a;
a = N + 20;
printf("%d",a);
}
Pre-processor
Compiler
Assembler
Interpreter
Linker & Loader
RAM
CPU
Monitor
30
Operating System

# COMPILER

- Compiler is a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (the target language).
- As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.



- Source code
  - written in a high level programming language

```
//simple example
while (sum < total)
{
  sum = sum + x*10;
}
```

- Target code
  - Assembly language which in turn is translated to machine code

```
L1: MOV   total,R0
    CMP   sum,R0
    CJ<   L2
    GOTO  L3
L2: MOV   #10,R0
    MUL   x,R0
    ADD   sum,R0
    MOV   R0,sum
    GOTO L1
L3: First Instruction following
    the while statement
```

# Why build compilers?

•Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.

•Compilers provide an essential interface between applications and architectures

• Compilers efficiently bridge the gap and shield the application developers from low level machine details

# Why study compilers?

- Compilers embody a wide range of theoretical techniques and their application to practice
  - DFAs, PDAs, formal languages, formal grammars, fix points algorithms, lattice theory, etc…
- Compiler construction teaches programming and software engineering skills
- Compiler construction involves a variety of areas
  - theory, algorithms, systems, architecture
- **Is compiler construction a solved problem?**
  - No! New developments in programming languages and machine architectures (processors) present new challenges

# Desirable Properties of Compilers

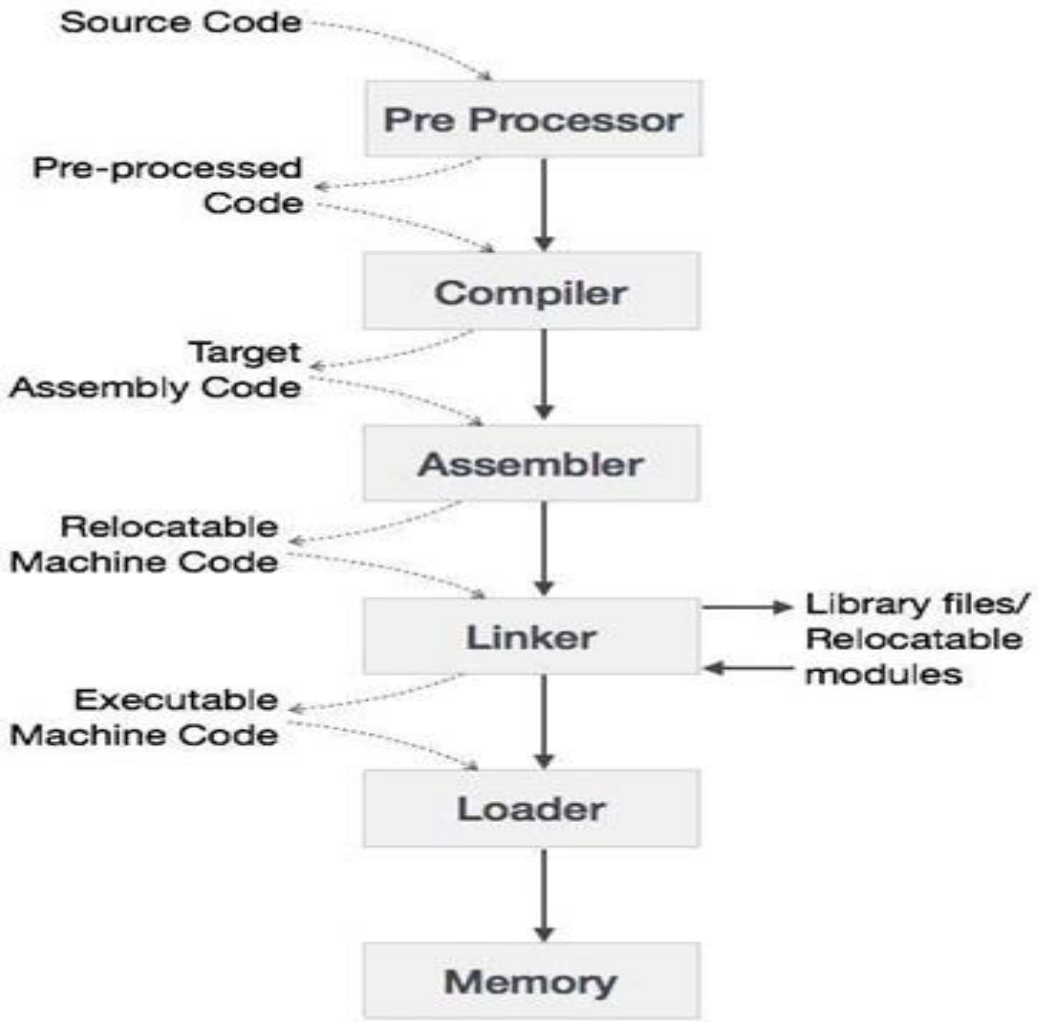
- **Compiler must generate a correct executable**
  - The input program and the output program must be equivalent, the compiler should preserve the meaning of the input program
- **Output program should run fast**
  - For optimizing compilers we expect the output program to be more efficient than the input program
- **Compiler should provide good diagnostics for programming errors**
- **Compiler should work well with debuggers**
- **Compile time should be proportional to code size**

Presidency University, Bengaluru

# COUSINS OF COMPILER

# COUSINS OF COMPILER

The input to a compiler may be processed by one or more preprocessors, and further processing of the compiler's output may be needed before running machine code is obtained.




**Cousins of Compiler are**

1. **Preprocessor**
2. **Assembler**
3. **Loader and Link-editor**

# Preprocessor

- A preprocessor is a program that processes its input data **(Source Program with Preprocessing Statements)** to produce output **(Source Program without Preprocessing Statements)** that is used as input to another program **(Compilers).**

They may perform the following functions :

1. **Macro processing**
2. **File Inclusion**
3. **Rational Preprocessors**
4. **Language extension**

**1. Macro processing:**

A Preprocessor may allow a user to define macros are shorthands for longer constructs.

**2. File Inclusion:**

Preprocessor includes header files into the program text. When the preprocessor finds an #include directive it replaces it by the entire content of the specified file.

**3. Rational Preprocessors:**

These processors change older languages with more modern flow-of-control and data- structuring facilities.

**4. Language extension :**

These processors attempt to add capabilities to the language by what amounts to built-in macros. For example, the language Equel is a database query language embedded in C.

# Assembler

Assembler creates object code by translating assembly instruction mnemonics into machine code.

**There are two types of assemblers:**

- One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them.
- 
- Two-pass assemblers create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code

# Linker and Loader

A **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

Three tasks of the linker are

1. Searches the program to find library routines used by program, e.g. printf(), math routines.
2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
3. Resolves references among files.

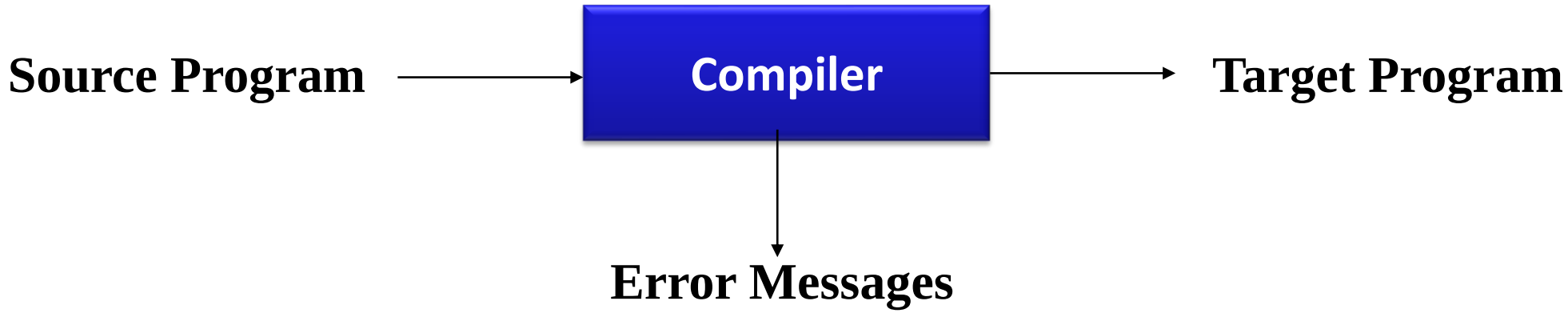
A **loader** is the part of an operating system that is responsible for loading executable programs into permanent memory for execution.

Presidency University, Bengaluru

# PHASES OF COMPILER

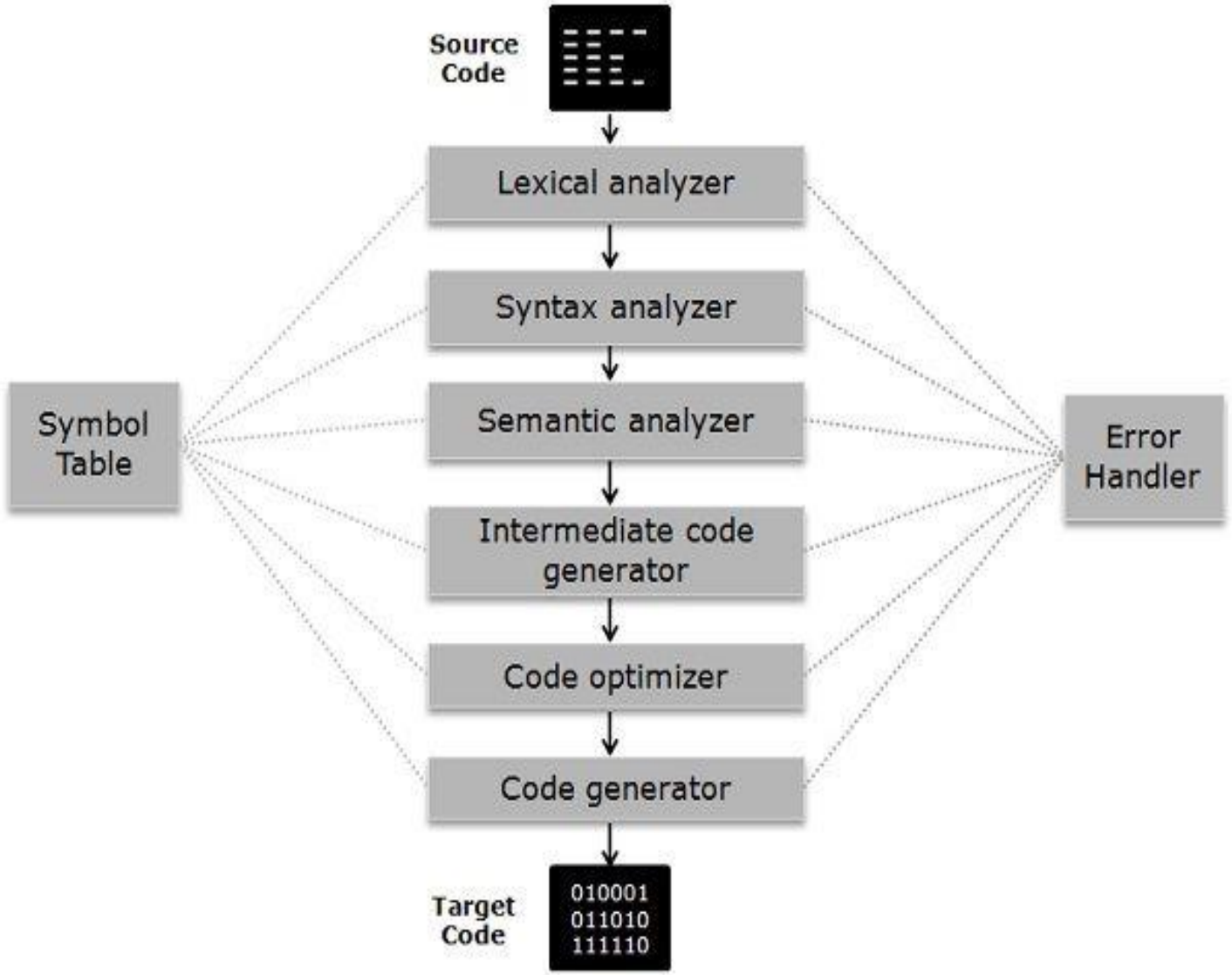
# COMPILER

- Compiler is a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (the target language).
- As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.



# Phases of a Compiler

Conceptually, a compiler operates in ***phases***, each of which transforms the source program from one representation to another.



# Lexical Analysis

- The first phase of a compiler is called lexical analysis or **scanning**. It is also called as **linear analysis**.

- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

- For each lexeme, the lexical analyzer produces as output a token of the form
  **(token-name, attribute-value)**
  that it passes on to the subsequent phase, syntax analysis.

- In the token, the first 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.

# Lexical Analysis

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 (=). Since this token needs **no attribute-value**.
3. **initial** is a lexeme that is mapped into the token **(id, 2)**, where 2 points to the symbol-table entry for initial .
4. **+** is a lexeme that is mapped into the **token (+)**.
5. **rate** is a lexeme that is mapped into the **token (id, 3)**, where 3 points to the symbol-table entry for rate .
6. ***** is a lexeme that is mapped into the **token (*).**
7. **60 is a lexeme** that is mapped into the token (number type,value). ➔ (Integer, 60)

# Lexical Analysis

**position = initial + rate * 60**

**(id,1) = (id,2) + (id,3) * (number,60)**

**For understanding easier and for our convenience, We can write this as**

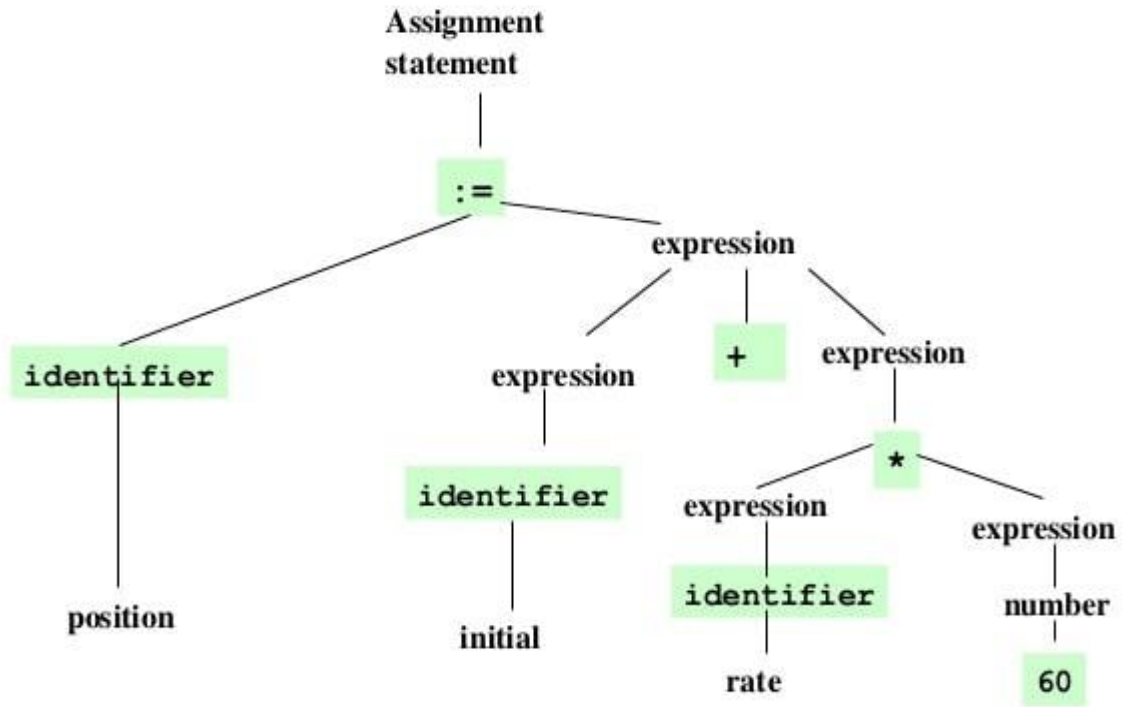
**id1 = id2 + id3 * 60**

# Syntax Analysis

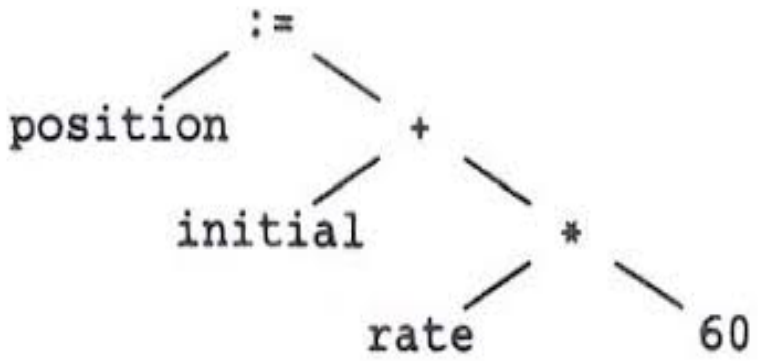
- The second phase of the compiler is **syntax analysis or parsing**.
- The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- A typical representation is a **syntax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

## Parse tree

position : = initial + rate * 60



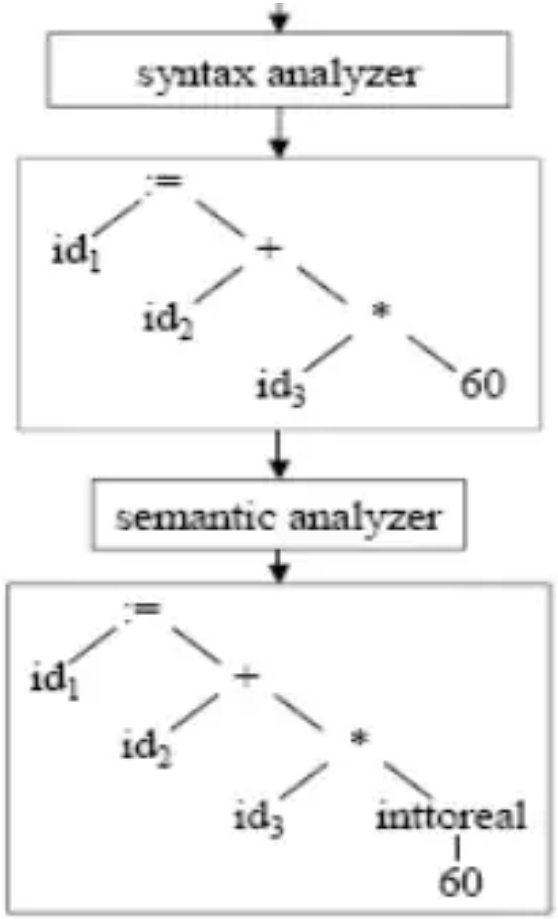

## Syntax Tree

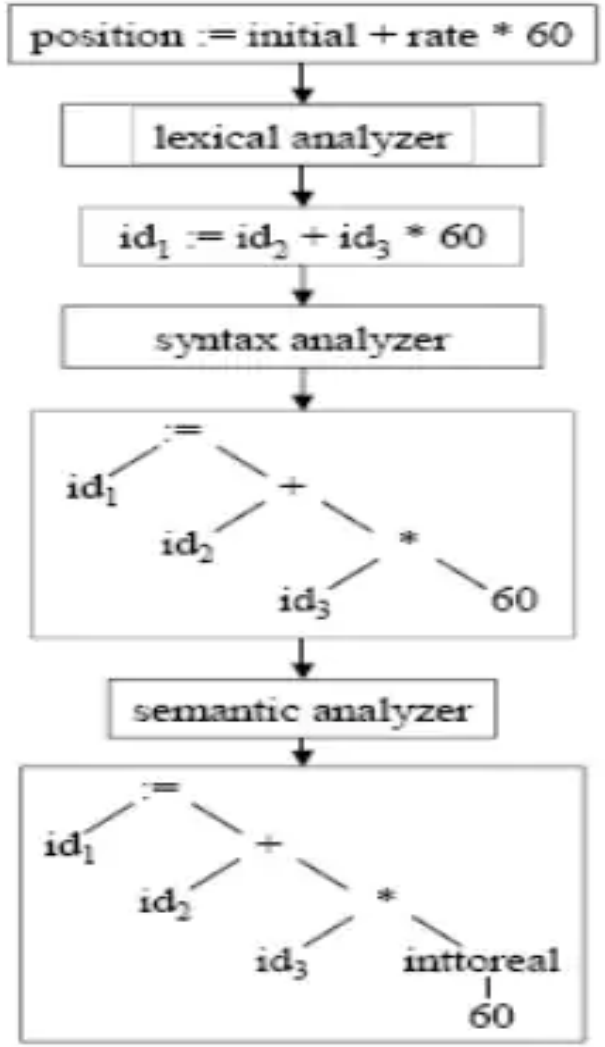


# Semantic Analysis

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

# Semantic Analysis



# Semantic Analysis

- The language specification may permit some type conversions called coercions. (Conversion from one type to another is said to be implicit if it is to be done automatically by the compiler. Implicit type conversions, also called Coercions (no information is lost in principle).

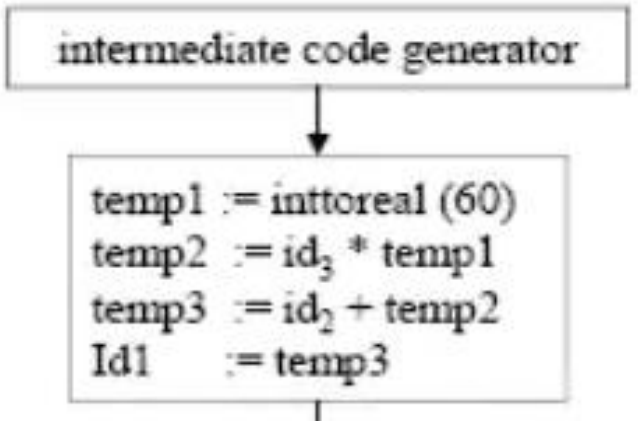
```
int b;
float a;
a = b;
```

- Conversion is said to be explicit if the programmer must write something to cause the conversion.

```
int b;
float a;
a = (int) b;
```

# Intermediate Code Generation

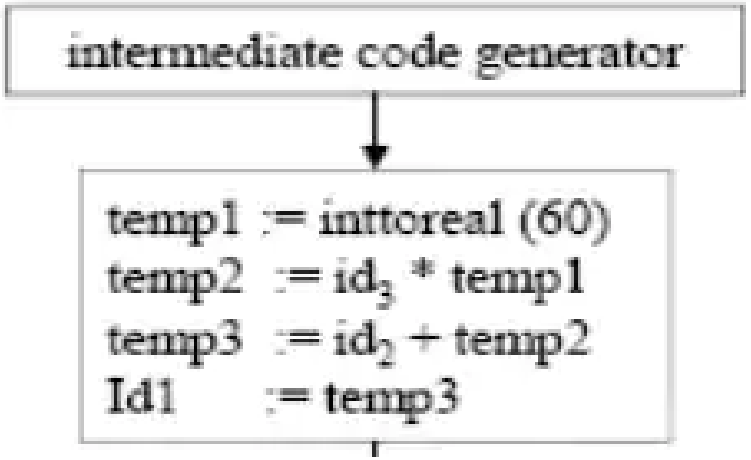
- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation.
- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a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



# Intermediate Code Generation

- One of the mostly used intermediate form called **three-address code**, which consists of a sequence of assembly-like instructions with three operands per instruction.

- Three-address instruction has several important properties:
  - First, each three-address assignment instruction has at most one operator on the right side.
  - Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction.
  - Third, some three-address instructions have fewer than three operands.

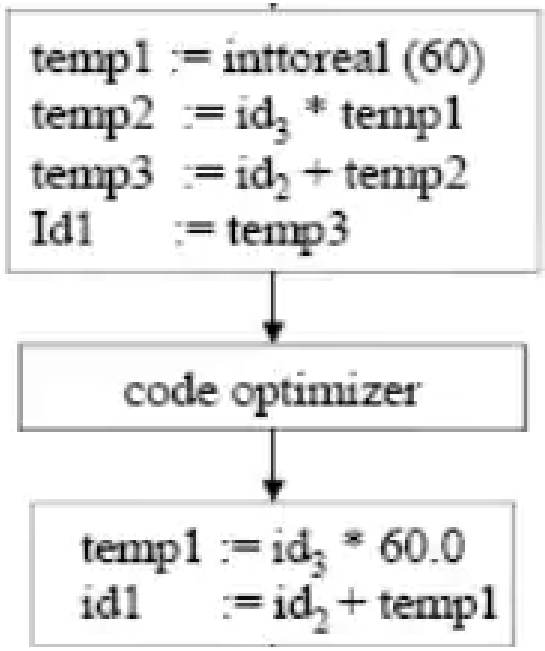


# Code Optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that **better target code will result**. Usually better means **faster**, but other objectives may be desired, such as shorter code, or target code that **consumes less power**.

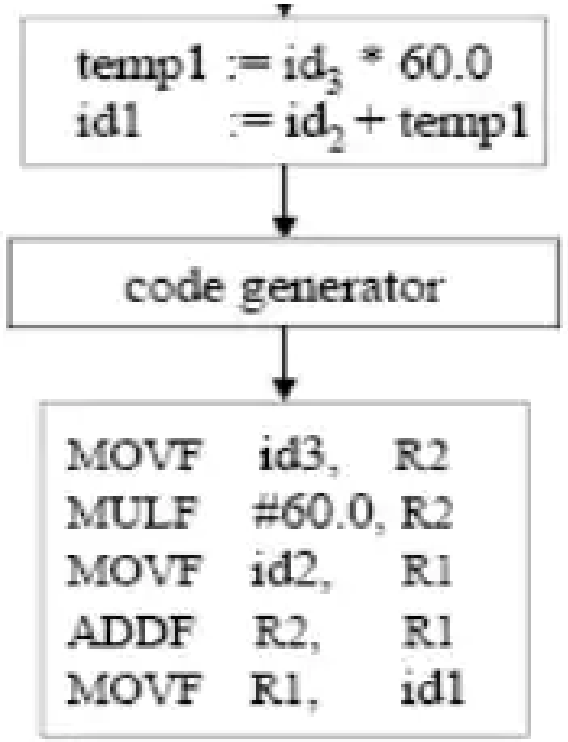
- There is a great variation in the amount of code optimization different compilers perform.

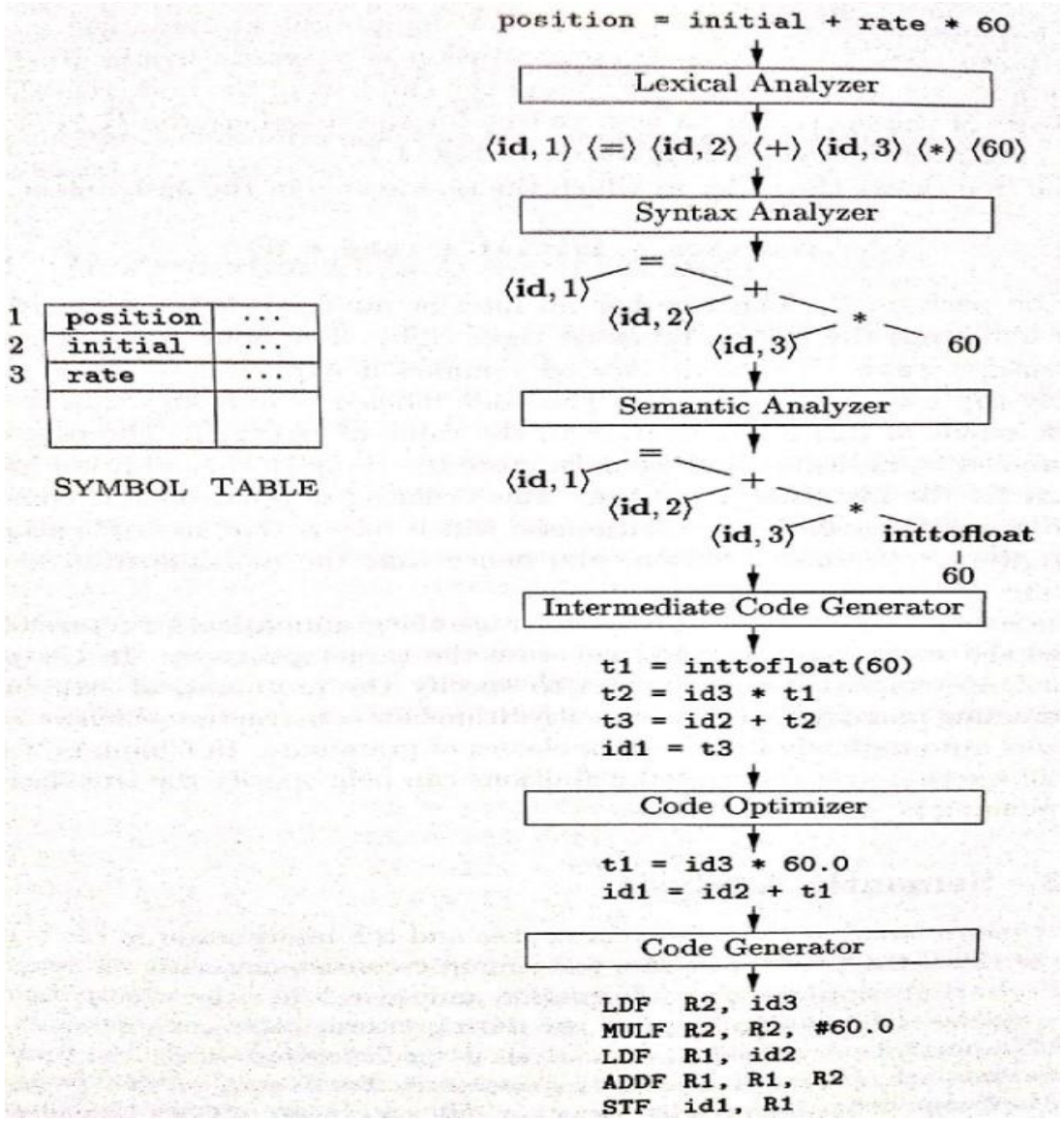
- **Optimizing compilers** spent a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.



# Code Generation

- Final phase of the compiler is generation of target code, consisting normally of **relocatable machine code or assembly code.**
- Memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.
- A crucial aspect of code generation is the judicious assignment of registers to hold variables.




position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
Semantic Analyzer
inttofloat
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
1 position ...
2 initial ...
3 rate ...
SYMBOL TABLE

# 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. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

- 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.**

# Symbol Table Management

## Symbol Table for Identifiers (Variables)

| Name of the Symbol | Location | Type | Scope | Value | Size | … |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |
| | | | | | | |

## Symbol Table for Identifiers (Functions)

| Name of the Symbol | No. of Arguments | Type of Arguments | Scope | Return type | … |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |

# Error Detection and Reporting

- Each phase can encounter Errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.

- The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.

**Some Errors in Phases of Compiler**

- ❑ Lexical analyzer: Wrongly spelled tokens
- ❑ Syntax analyzer: Missing parenthesis
- ❑ Semantic analyzer: Mismatched operands for an operator
- ❑ Intermediate code generator: When the memory is full when generating temporary variables
- ❑ Code Optimizer: When the statement is not reachable
- ❑ Code Generator: When the memory is full or proper registers are not allocated

Presidency University, Bengaluru

# Analysis of the Phases and Grouping of Phases

# Analysis of the Phases

**Compilation Process is divided into two parts:**

- **Analysis and**
- **Synthesis**

- 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**.

# Grouping of Phases into Passes

# Front End and Back End

- The **analysis part** is often called the **front end of the compiler**; the **synthesis part** is the **back end**.
- The discussion of **phases** deals with the **logical organization** of a compiler.
- In an implementation, activities from **several phases may be grouped together into a pass** that reads an input file and writes an output file.

## Front End

- For example, the **front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation** might be grouped together into one pass.
- **Frond ends** are **depends primarily** on **source language** and are largely independent of the target machine.

## Back End

- The **back end** includes those portions of the compiler that **depend on the target machine**, and generally, these portions do **not depend on the source language, just the intermediate language**.

- **Code optimization** might be an **optional pass**. Then there could be a back-end pass consisting of **code generation** for a particular target machine.

Some compiler collections have been created around carefully designed **intermediate representations** that allow the front end for a particular language to interface with the back end for a certain target machine.

With these collections, **we can produce compilers for different source languages for one target machine** by combining different front ends with the back end for that target machine.

Similarly, **we can produce compilers for different target machines**, by combining a front end with back ends for different target machines.

# Passes

Several phases of compilation are implemented in a single pass consisting of reading an input file and writing an output file.

## Reducing the number of passes

- Takes time to read and write intermediate files.
- Grouping of several phases into one pass, may force the entire program in memory, because one phase may need information in a different order than previous phase produces it.
- Intermediate code and code generation are often merged into one pass using a technique called backpatching.

For some phases, grouping into single pass presents few problems.

For example, the interface between lexical and syntax analyzers can often limited into single token.

On the other hand, it is often very hard to perform code generation until the intermediate representation has been completely generation.

To avoid these issues, **Backpatching** is used.

In some cases, it is possible to leave a blank slot for missing information, and fill in the slot when the information become available. This is called as "Backpatching".

Presidency University, Bengaluru

# Compiler Construction Tools

# Compiler Construction Tools

The compiler writer, like any **software developer, can profitably use** modern software development environments containing tools such as **language editors, debuggers, version managers, profilers, test harnesses**, and so on.

Shortly after the **first compilers were written**, systems to help with the compiler-writing process appeared. These systems have often been referred to as **Compiler-Compilers, Compiler-Generators** or **Translator-Writing Systems.**

Some general tools have been created for the **automatic design of specific compiler** components. These tools use specialized languages for specifying and implementing the component, and also many use sophisticated algorithms.

**The most successful tools produce components that can be easily integrated into the remainder of a compiler.**

# Compiler Construction Tools

The following is a list of some useful compiler-construction tools:

- **Parser Generators**
- **Scanner Generators**
- **Syntax-directed Translation Engines**
- **Automatic Code Generators**
- **Data-flow Analysis Engines**

# Compiler Construction Tools

- **Scanner Generators**

  This tool **automatically generate lexical analyzers**, normally from a specification based on **regular expressions**. The basic organization of the resulting lexical analyzers is in effect a **finite automaton**.

- **Parser Generators**

  These tools produce **syntax analyzers**, normally from input that is based on a **Context-Free Grammar (CFG)**. In early compilers, syntax analysis consumed not only a large fraction of the compilation time, but a large fraction of the intellectual effort of writing a compiler. **Many parser generator utilize powerful parsing algorithm that are too complex.**

# Compiler Construction Tools

- **Syntax-directed Translation Engines**

  These tools produce **collections of routines** for walking a parse tree and **generating intermediate code**. The basic idea is that one or more "translation" are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbor nodes in the tree.

- **Automatic Code Generators**

  These tools **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**. The rules must include sufficient details that we can handle the different possible access methods for data. For example **variable may be in register or fixed (static) location or may be in stack.**

  **The basic technique is "Template Matching" –** The intermediate code statements are replaced by "templates" that represent sequences of machine instruction.

# Compiler Construction Tools

- **Data-flow Analysis Engines**

  Much of the information needed to perform **good code optimization** involves "**Data-Flow Analysis**", 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**.

Presidency University, Bengaluru

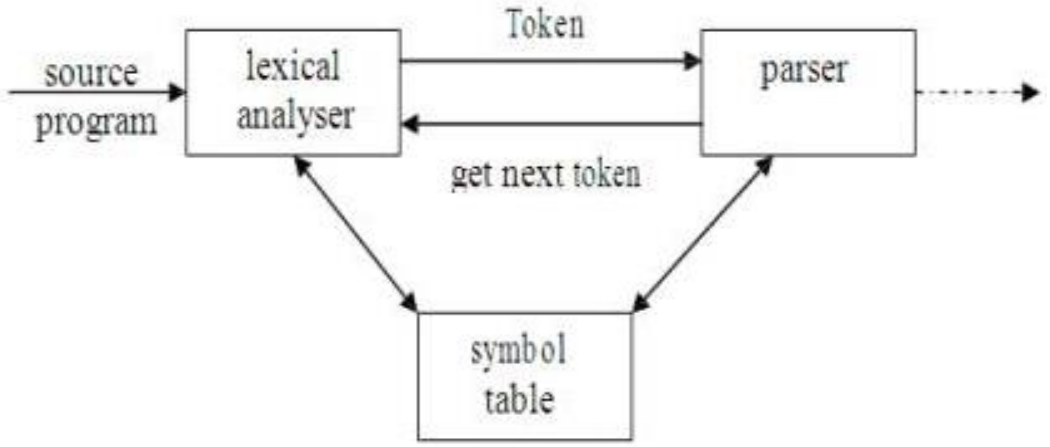
# Lexical Analysis

# -

# Role and Need of Lexical Analysis

# The Role of the Lexical Analyzer

- As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and 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.
- It is common for the lexical analyzer to interact with the symbol table as well.
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.



# The Role of the Lexical Analyzer

## Issues in Lexical Analysis

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

1. **Simplicity of design is the most important consideration.**

   The separation of lexical from syntax analysis often allows us to simplify at least one or the other of these phases. (For example, Stripping out comments and white spaces is somewhat easy job by lexical analyzer than syntax analyzer).

2. **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.

3. **Compiler portability is enhanced.**

   Input-device-specific peculiarities can be restricted to the lexical analyzer. (For example, ↑ in pascal)

# The Role of the Lexical Analyzer

**Lexical Analyzer also performs certain secondary task**

- Stripping out comments and whitespace (blank, newline, tab) in the source program.
- Correlating error messages generated by the compiler with the source program.

Sometimes, 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, where the scanner produces the sequence of tokens as output.

# The Role of the Lexical Analyzer

## Tokens, Patterns, and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

### Token

- 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 (Keyword, Identifier, Operator…)

### Pattern

- The set strings is described by a rule called a pattern associated with the token.

Or

- A pattern is a rule describing the set of lexemes that can represent a particular token in source programs.

### Lexeme

- A lexeme is a sequence of characters in the source program that matched by the pattern for a token.

# The Role of the Lexical Analyzer

| Pattern | Token | Lexeme |
|---|---|---|
| while, for, if, else, int | Keyword | while \| for \| if \| else \| int |
| Letter followed by zero or more instance of Letter or Digit | Identifier | a, mark1, length, emp123, s567 |
| Digit followed by zero of more instance of Digit | Number | 79, 22, 4, 2021, 123456 |
| Any symbols between “ and “ except “ | Literal or String | “Welcome”, “Presidency University”, “Bengaluru” |
| | | |
| | | |

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the 1 operators, either individually or in classes.
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.

# The Role of the Lexical Analyzer

## Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.
- 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 appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

**<Token Name, Attribute-Value>**

**<Identifier, 2001>  <Number, 123>  <IF>  <+>  <{>**

# The Role of the Lexical Analyzer

## Lexical Error

- It is hard for a lexical analyzer to tell source-code error.
- For instance, if the string **whiel** is encountered for the first time in a C program in the context:

  **whiel ( Mark >= 90)**

  a lexical analyzer cannot tell whether **whiel** is a misspelling of the keyword **while** or an undeclared function identifier. Since **whiel** is a **valid lexeme** for the token id, the lexical analyzer must return the token id to the parser.
- 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, We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token.

# The Role of the Lexical Analyzer

## Lexical Error

Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes, but this approach is considered too expensive in practice to be worth the effort.

Presidency University, Bengaluru

# Input Buffering and Specification of Tokens

Presidency University, Bengaluru

# Input Buffering

# Input Buffering

- To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.
- Hence a **two-buffer scheme** is introduced to handle large lookaheads safely.
- Techniques for speeding up the process of lexical analyzer such as the use of **sentinels** to mark the buffer end have been adopted.

There are three general approaches for the implementation of a lexical analyzer:

1. By using a lexical-analyzer generator, such as **lex compiler** to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.
2. By writing the lexical analyzer in a **conventional systems-programming language**, using I/O facilities of that language to read the input.
3. By writing the lexical analyzer in **assembly language** and explicitly managing the reading of input.

# Input Buffering - Buffer Pairs

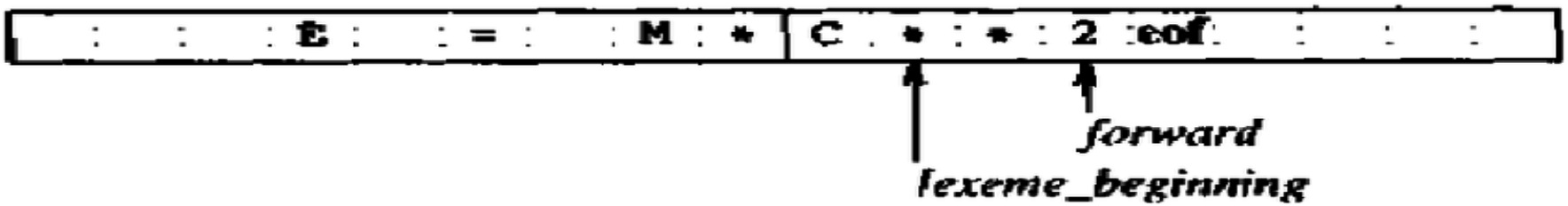
Two Techniques are used for Input Buffering:

- **Buffer Pairs**
- **Sentinels**

## Buffer Pairs

Because of large amount of time consumption in moving characters, **specialized buffering techniques** have been developed to reduce the amount of overhead required to process an input character.

Fig shows the buffer pairs which are used to hold the input data.



# Input Buffering - Buffer Pairs

- Buffer Pairs scheme Consists of **two buffers**, each consists of N-character size which are reloaded alternatively.
- N-Number of characters on one disk block, e.g., **1024 or 4096**.
- N characters are read from the input file to the buffer using one **system read command** rather than invoking a read command for each character.
- **eof** is inserted at the end if the number of characters is less than N.

Two pointers to the input are maintained:

- Pointer **lexeme_beginning**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
- 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.

**The string of characters between the two pointers is the current lexeme.**

# Input Buffering - Buffer Pairs

**Disadvantages of this scheme**

- This scheme works well most of the time, but the amount of **lookahead is limited**.
- This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.

## Code to advance forward pointer

```
if forward at end of the first half then begin
        reload second half;
        forward = forward + 1;
end
else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half;
end
else
        forward = forward + 1;
```

# Input Buffering - Sentinels

## Sentinels:

- In the previous scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.

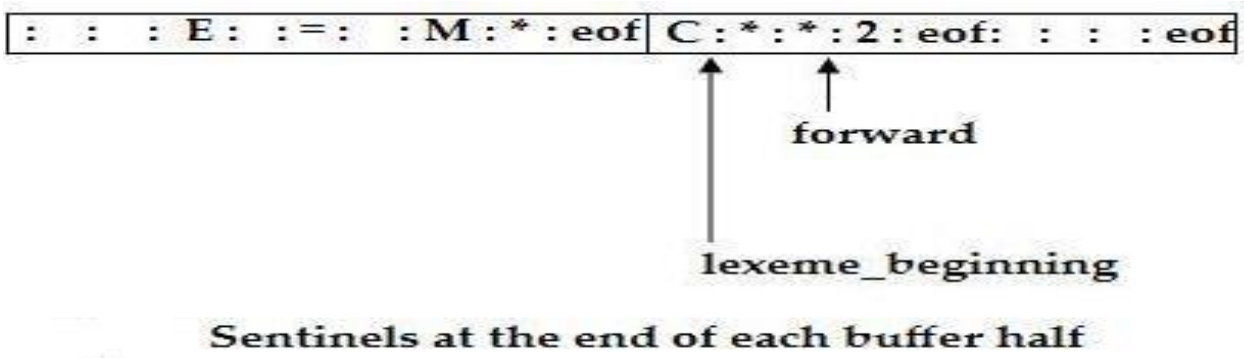
- Therefore the **ends of the buffer halves require two tests** for each advance of the forward pointer.

  ***Test 1:*** For end of buffer.
  ***Test 2:*** To determine what character is read.

- The usage of **sentinel reduces the two tests to one** by extending each buffer half to hold a sentinel character at the end.

- The **sentinel is a special character that cannot be part of the source program.** (***eof*** **character is used as sentinel**).

# Input Buffering - Sentinels




```
forward = forward + 1;
If forward = eof then begin
        if forward at end of the first half then begin
                reload second half;
                forward = forward + 1;
        end
        else if forward at end of second half then begin
                reload first half;
                move forward to beginning of first half
        end
        else    /* eof within a buffer signifying end of input */
                terminate lexical analysis;
end
```

# Input Buffering - Sentinels

**Advantages**

- **Most of the time, It performs only one test** to see whether forward pointer points to an *eof.*
- Only when it reaches the end of the buffer half or *eof,* it performs more tests.

Presidency University, Bengaluru

# Specification of Tokens

# Specification of Tokens

**Regular expressions** are an important notation for specifying lexeme patterns.

Specification of Token consists of

- **Strings and Languages**
- **Operations on Languages**
- **Regular Expressions**
- **Regular Definitions**
- **Notational Shorthands**
- **Nonregular Sets**

# Specification of Tokens - Strings and Languages

## Strings and Languages

- **An alphabet is any finite set of symbols**. Typical examples of symbols are letters, digits, and punctuation**.**
- The set {0,1} is the **binary alphabet**.
- **ASCII** is an important example of an alphabet; it is used in many software systems.
- **Unicode**, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet.

# Specification of Tokens - Strings and Languages

## Strings

- A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
  - In language theory, the terms "sentence" and "word" are often used as synonyms for "string."
  - The **length of a string s**, usually written **|s|**, is the number of occurrences of symbols in s.

    For example, banana is a string of length six. The **empty string, denoted Ɛ, is the string of length zero.**
  - If x and y are strings, then the **concatenation** of x and y, denoted xy, is the string formed by appending y to x.

## Languages

- A **language** is any set of strings over some fixed alphabet.
- Abstract languages like **Ø**, the empty set, or **{ε}**, the set containing only the empty string.

# Specification of Tokens - Strings and Languages

## Terms for parts of a string

| *TERM* | *DEFINITION* |
|---|---|
| prefix of s | A string obtained by removing zero or more trailing symbols of string s; ban is a prefix of banana. |
| suffix of s | A string formed by deleting zero or more of the leading symbols of s; nana is a suffix of banana. |
| substring of s | A string obtained by deleting a prefix and a suffix from s; nan is a substring of banana. Every prefix and every suffix of s is a substring of s, but not every substring of s is a prefix or a suffix of s. For every string s, both s and e are prefixes, suffixes, and substrings of s. |
| proper prefix, suffix, or substring of s | Any nonempty string x that is, respectively, a prefix, suffix, or substring of s such that s ≠ x. |
| subsequence of s | Any string formed by deleting zero or more not necessarily contiguous symbols from s; baaa is a subsequence of banana. |

## Operations on Languages

- There are several important operations that can be applied to Languages, For lexical analysis, the most important operations on languages are **union, concatenation, and closure.**

Definitions of operations on languages

| OPERATION | DEFINITION |
|---|---|
| *union* of $L$ and $M$ written $L \cup M$. | $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$ |
| *concatenation* of $L$ and $M$ written $LM$ | $LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$ |
| *Kleene closure* of $L$ written $L^*$ | $L^* = \bigcup_{i=0}^{\infty} L^i$ <br> $L^*$ denotes "zero or more concatenations of" $L$. |
| *positive closure* of $L$ written $L^+$ | $L^+ = \bigcup_{i=0}^{\infty} L^i$ <br> $L^+$ denotes "one or more concatenations of" $L$. |

# Specification of Tokens - Regular Expressions

## Regular Expressions

**Regular expressions** are an important notation for specifying lexeme patterns.

**Identifier - Letter (Letter | Digit)***

**Integer - Digit (Digit)***

**Float - Digit (Digit)* . Digit (Digit)***

# Specification of Tokens - Regular Expressions

The regular expressions are built up out of smaller regular expressions using set of defining rules

or

The regular expressions are built recursively out of smaller regular expressions.

Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r's subexpressions.

a) The unary operator * has highest precedence and is left associative.
b) Concatenation has second highest precedence and is left associative.
c) | has lowest precedence and is left associative.

A language that can be defined by a regular expression is called a **regular set**.

# Specification of Tokens - Regular Expressions

If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s. For instance, (a|b) = (b|a).

There are a number of algebraic laws for regular expressions.

| AXIOM | DESCRIPTION |
|---|---|
| **r\|s = s\|r** | \| is commutative |
| **r\|(s\|t) = (r\|s)\|t** | \| is associative |
| **r(st) = (rs)t** | Concatenation is associative |
| **r(s\|t) = rs\|rt**<br>**(s\|t)r = sr\|tr** | Concatenation distributes over \| |
| **εr = rε = r** | ε is the identity for concatenation |
| **r* = (r\|ε)*** | ε is guaranteed in a closure |
| **r** = r*** | * is idempotent |

# Specification of Tokens - Regular Definitions

For notational convenience, we may wish to give names to certain regular expressions and to define regular expressions using these names as if they were symbols. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$d_n \rightarrow r_n$$

Where each $d_i$ is a district name, and each $r_i$ is a regular expression over the symbols in $\Sigma \cup \{d_1, d_2, \ldots d_{i-1}\}$, i.e., the basic symbols and the previously defined names.

# Specification of Tokens - Regular Definitions

**Letter → A | B | C | ... | Z | a | b | c | ... |z|**

**Digit → 0 | 1 | 2 | ... | 9 |**

**Identifier → {Letter} ( {Letter} | {Digit} )***

# Specification of Tokens - Notational Shorthands

Certain constructs occur **so frequently in regular expressions** that it is convenient to introduce **notational shorthands** for them.

1. **One or more instances.** The unary, postfix operator **+** represents the positive closure of a regular expression and its language.

   Two useful algebraic laws, $\mathbf{r^* = r^+ \mid \varepsilon}$ and $\mathbf{r^+ = rr^* = r^*r}$ relate the Kleene closure and positive closure.

2. **Zero or one instance.** The unary postfix operator **?** means "zero or one occurrence." That is, r? is equivalent to r|ε.

3. **Character classes.** The notation **[a-z]** represents a | b | c | … | z
   The notation **[1-4]** represents 1 | 2 | 3 | 4

# Specification of Tokens - Nonregular Sets

- **Some languages cannot be described by any regular expressions.**
- A language which cannot be described by any regular expression is a **non-regular set.**
- Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression.
- This set can be specified by a context-free gramma

The set

**{ wcw | w is a strings of a's and b's}**

cannot be denoted by regular expressions for repeating strings, nor can it be described by a CFG.

**S → aSb | ab**

Presidency University, Bengaluru

# RECOGNITION OF TOKENS

# Recognition of Tokens

- In the previous section we learned how to express patterns using regular expressions.

- Now, we must study 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.

- Consider the following example:

stmt → **if** expr **then** stmt
→ **if** expr **then** stmt **else** stmt
→ **ε**

expr → term **relop** term
→ term

term → **id**
→ **number**

# Recognition of Tokens

- The grammar describes a simple form of branching statements and conditional expressions.
- 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.

| | | |
|---|---|---|
| **Letter** | → | **[A-Za-z]** |
| **Digit** | → | **[0-9]** |
| **ID** | → | **{Letter} ({Letter} \| {Digit} )*** |
| **NUMBER** | → | **{Digit}+** |
| **IF** | → | **if** |
| **ELSE** | → | **else** |
| **THEN** | → | **then** |
| **RELOP** | → | **< \| <= \| > \| >= \| == \| !=** |

- To simplify matters, we make the common assumption that keywords are also reserved words: that is, they are not identifiers, even though their lexemes match the pattern for identifiers.

# Recognition of Tokens

In addition, we assign the lexical analyzer the job of stripping out white-space, by recognizing the "token" ws defined by:

$$\mathbf{ws} \rightarrow \mathbf{(blank \mid tab \mid newline)^{+}}$$

Here, **blank, tab,** and **newline** are abstract symbols that we use to express the ASCII characters of the same names.

Token ***ws*** is different from the other tokens in that, when we recognize it, we do not return it to the parser.

# Recognition of Tokens

The following table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value.

| LEXEMES | TOKEN NAME | ATTRIBUTE VALUE |
|---|---|---|
| Any ws | - | - |
| if | if | - |
| then | then | - |
| else | else | - |
| Any Id | id | Pointer to Symbol Table Entry |
| Any Number | number | Value |
| < | relop | LT |
| <= | relop | LE |
| > | relop | GT |
| >= | relop | GE |
| == | relop | EQ |
| != | relop | NE |

# Recognition of Tokens - Transition Diagrams

- As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "**transition diagrams**“.
- 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.
- Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols.

# Recognition of Tokens - Transition Diagrams

Some important conventions about transition diagrams are:

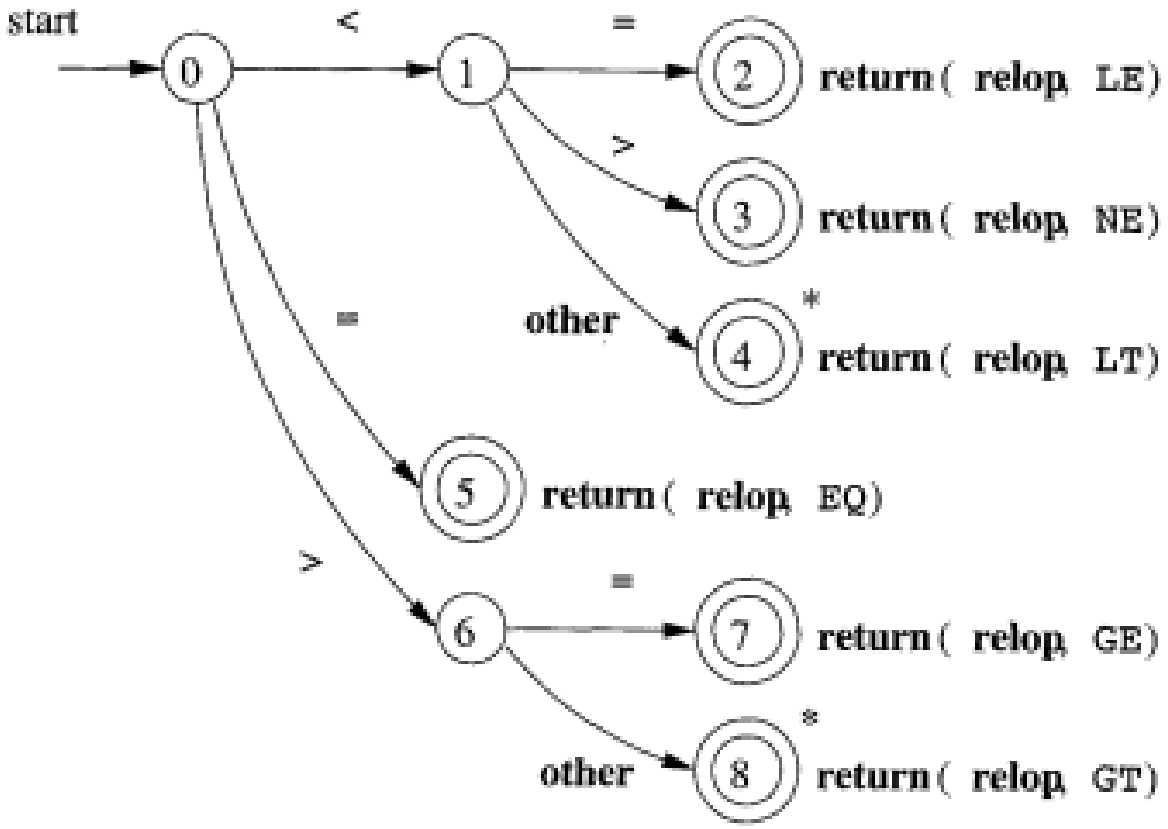
- One state is designated the **start state, or initial state**; it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

- Certain states are said to be accepting, or final. We always indicate an accepting state by a **double circle**, and if there is an action to be taken — typically returning a token and an attribute value to the parser.

- In addition, if it is necessary to retract the forward pointer one position then we shall additionally **place a *** near that accepting state.

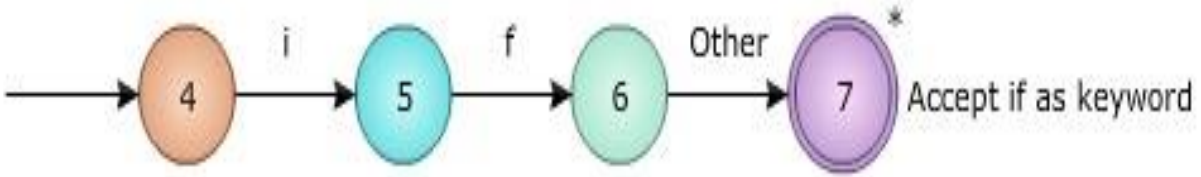
# Recognition of Tokens - Transition Diagrams

Transition diagram that recognizes the lexemes matching the token **relop**



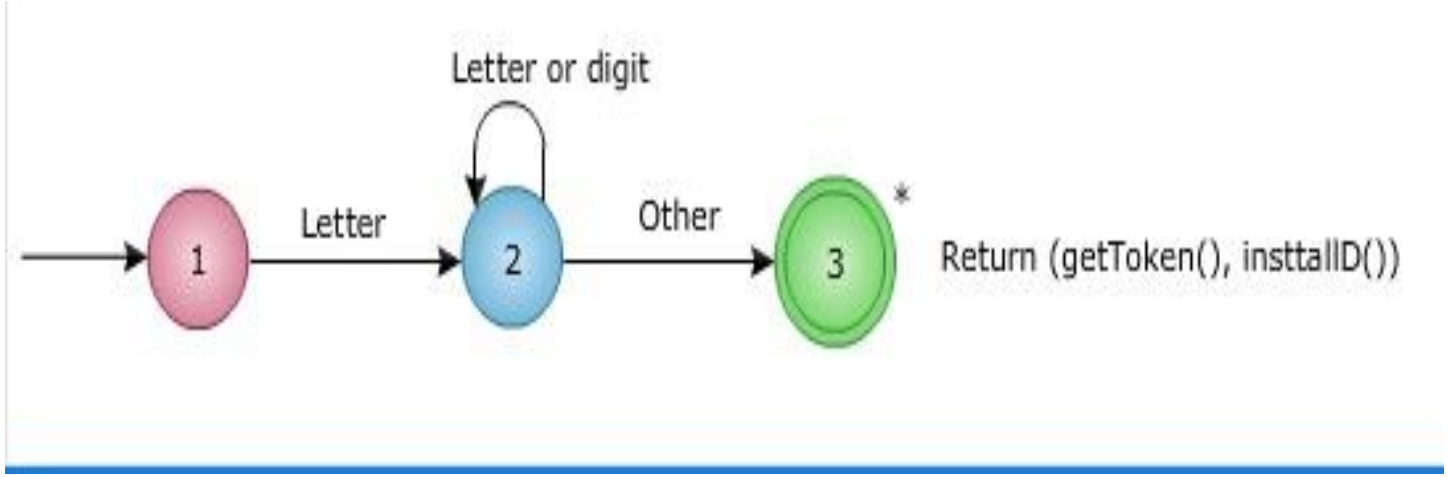
# Recognition of Tokens - Transition Diagrams

## Recognition of Reserved Words (IF)



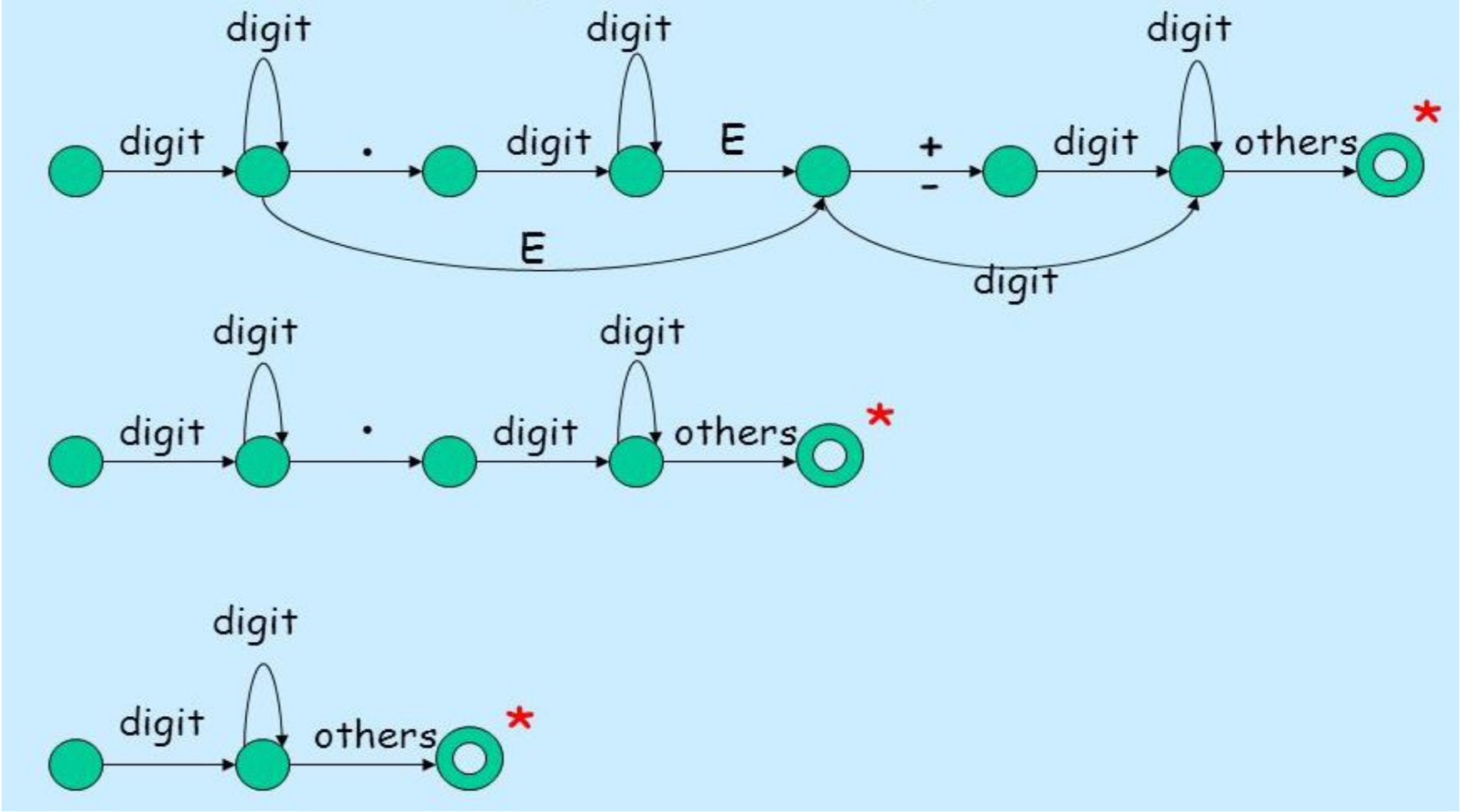
# Recognition of Tokens - Transition Diagrams

## Recognition of Identifiers



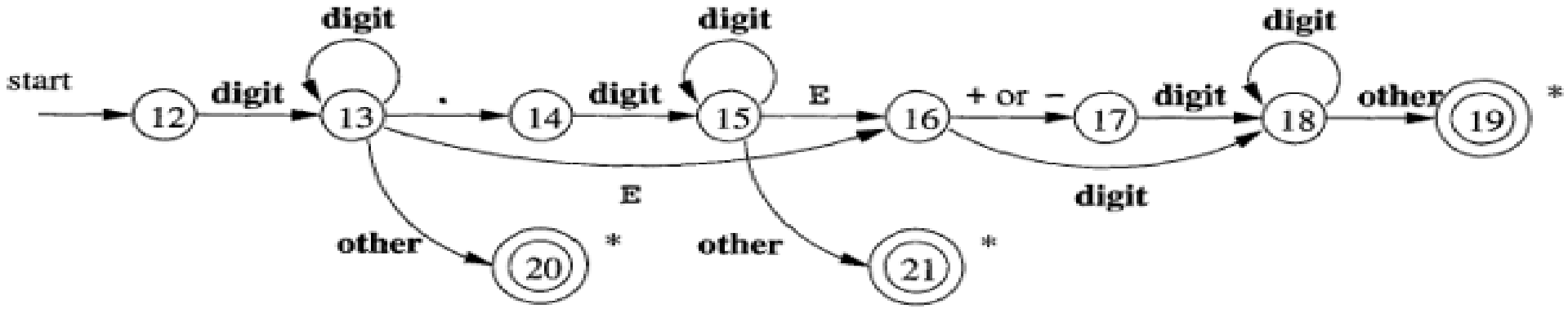
# Recognition of Tokens - Transition Diagrams

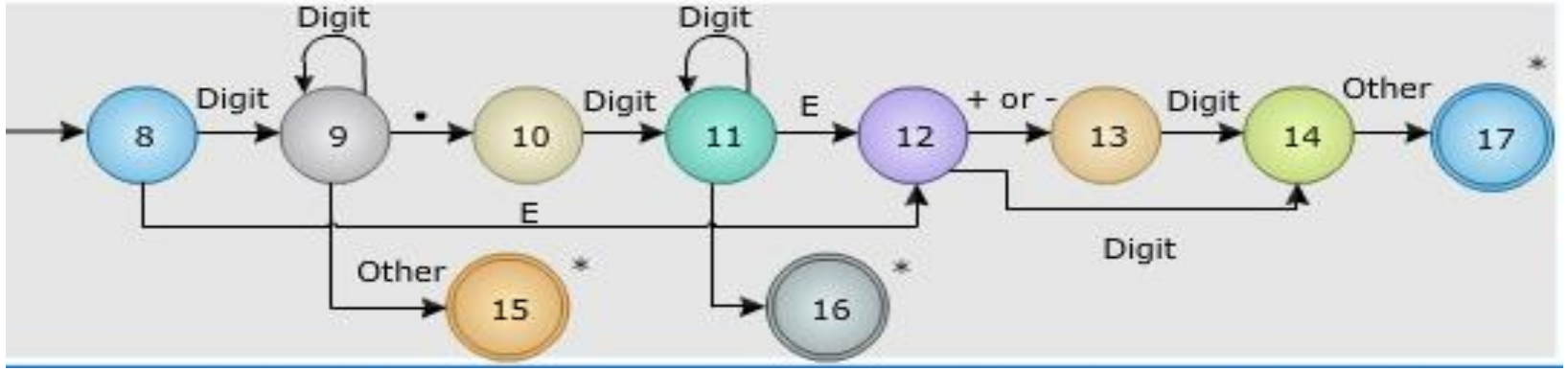
## Recognition of Number



# Recognition of Tokens - Transition Diagrams

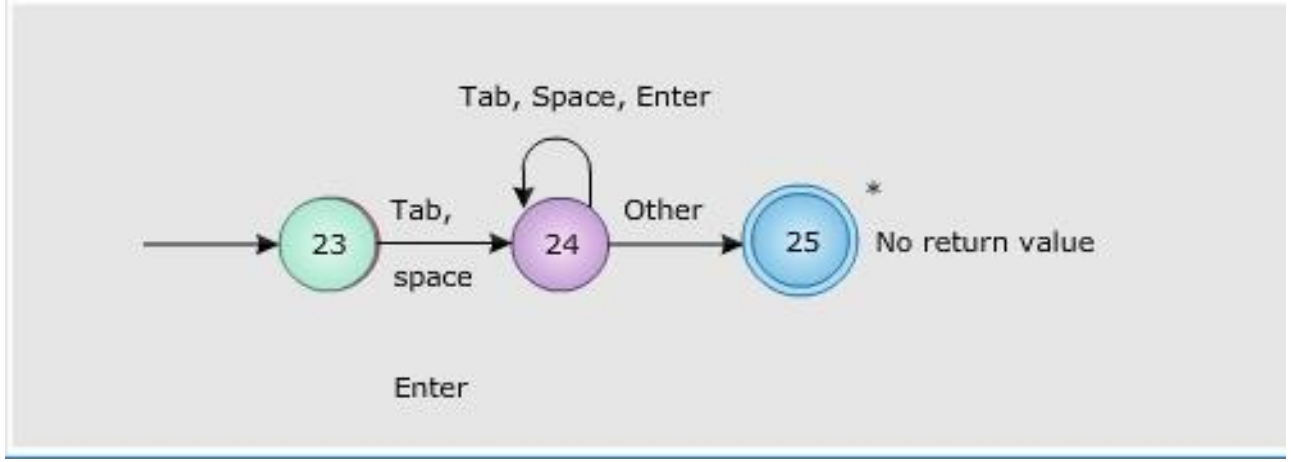
## Recognition of Number



# Recognition of Tokens - Transition Diagrams

## Recognition of White Spaces



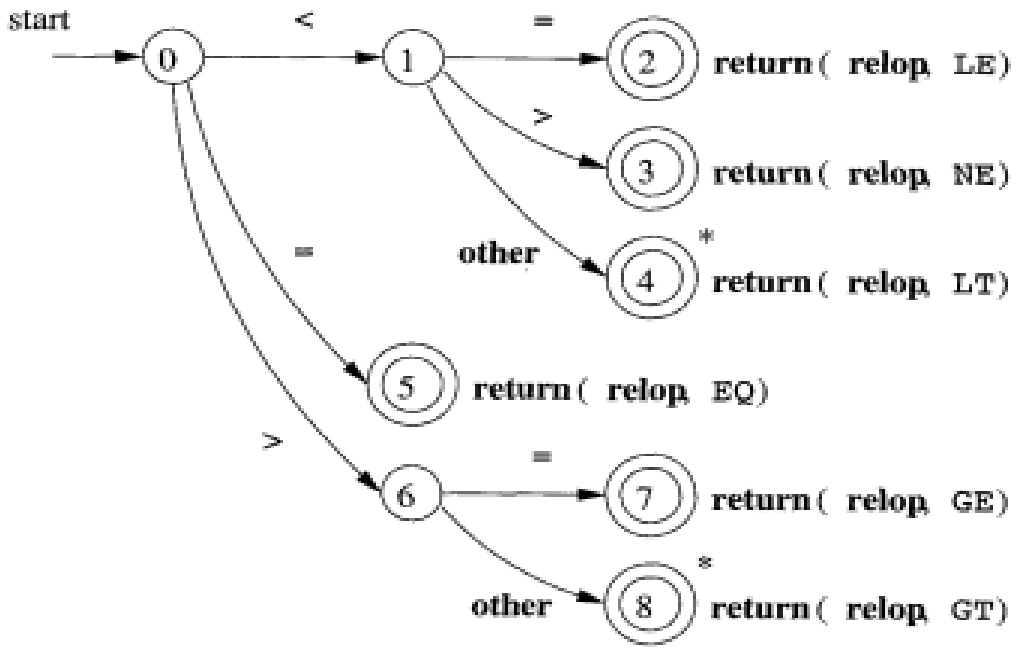
# Recognition of Tokens - Transition Diagrams

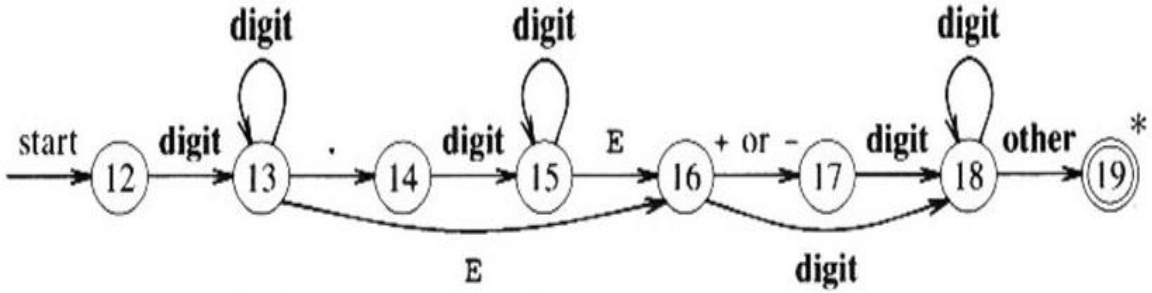
There are two ways that we can handle reserved words that look like identifiers:

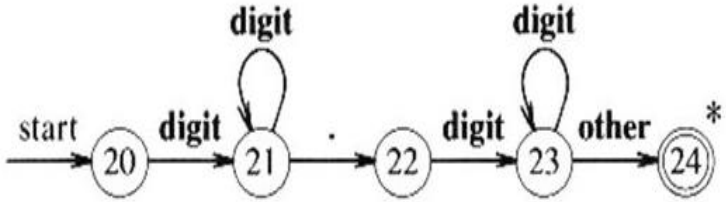
- Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. When we find an identifier, a call to installlD places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found.

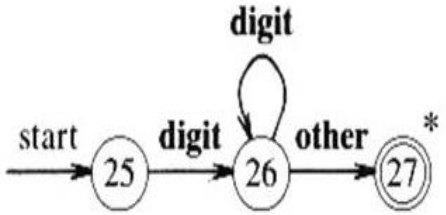
- Create separate transition diagrams for each keyword.

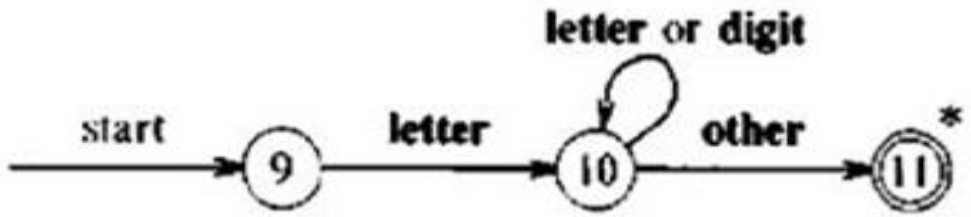
# Recognition of Tokens – Implementing a Transition Diagrams



# Recognition of Tokens – Implementing a Transition Diagrams

```
token nexttoken()
{   while(1) {
        switch (state) {
        case 0:   c = nextchar();
           /* c is lookahead character */
           if (c==blank || c==tab || c==newline) {
              state = 0;
              lexeme_beginning++;
                 /* advance beginning of lexeme */
           }
           else if (c == '<') state = 1;
           else if (c == '=') state = 5;
           else if (c == '>') state = 6;
           else state = fail();
           break;

           .../* cases 1-8 here */

        case 9:   c = nextchar();
           if (isletter(c)) state = 10;
           else state = fail();
           break;
        case 10:  c = nextchar();
           if (isletter(c)) state = 10;
           else if (isdigit(c)) state = 10;
           else state = 11;
           break;
        case 11:  retract(1); install_id();
           return ( gettoken() );

           .../* cases 12-24 here */

        case 25:  c = nextchar();
           if  (isdigit(c)) state = 26;
           else state = fail();
           break;
        case 26:  c = nextchar();
           if (isdigit(c)) state = 26;
           else state = 27;
           break;
        case 27:  retract(1); install_num();
           return ( NUM );
        }
     }
}
```

# Recognition of Tokens – Implementing a Transition Diagrams

```
int state = 0, start = 0;
int lexical_value;
    /* to "return" second component of token */

int fail()
{
    forward = token_beginning;
    switch (start) {
        case 0:    start = 9; break;
        case 9:    start = 12; break;
        case 12:   start = 20; break;
        case 20:   start = 25; break;
        case 25:   recover(); break;
        default:    /* compiler error */
    }
    return start;
}
```

# Recognition of Tokens - Finite Automata

We compile a regular expression into a recognizer by constructing a generalized transition diagram called a **finite automaton**.

Finite automat a are **recognizers**; they simply say "**yes**" **or** "**no**" about each possible input string.

Finite automat a come in two flavors:

**(a)** **Nondeterministic finite automata (NFA)** have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and **ε**, the empty string, is a possible label.

**(b)** **Deterministic finite automata (DFA)** have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

# Recognition of Tokens - Finite Automata

**Non-Deterministic Finite Automata (NFA)**

Definition: A NFA is a mathematical model that consists of 5-tuple, M=(S, ∑, $S_0$, F, δ) where,

- **S** is the set of finite states
- ∑ is the set of input symbols (The input symbol alphabet)
- **$S_0$** is the initial state
- **F** is the set of all accepting states or final states.
- **δ** is the transition function **move** that maps state-symbol pairs to set of states

**A Deterministic Finite Automaton (DFA)** is a special case of an NFA where:

1. There are no moves on input **ε**, and
2. For each state s and input symbol a, there is exactly one edge out of s.

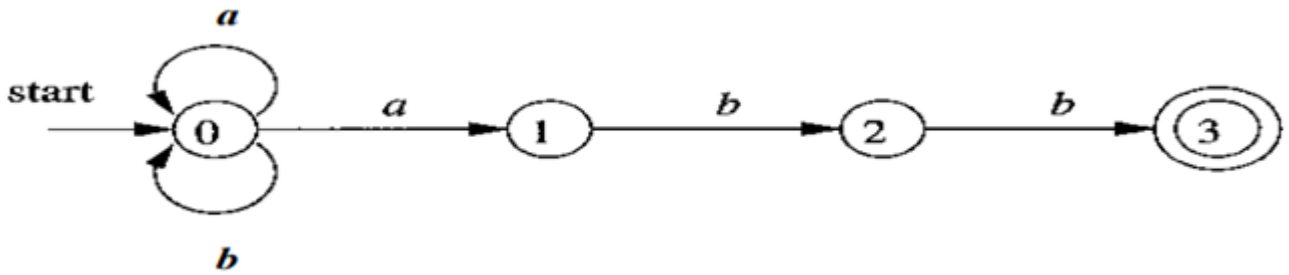
# Recognition of Tokens - Finite Automata

## Difference between NFA and DFA

| NFA | DFA |
|---|---|
| In NFA, more than one transition out of a state may be possible on the same input symbol. | For each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state. |
| There is a Time tradeoff, it is a Slow Recognizer | There is a Time tradeoff, it is a Faster Recognizer |
| NFA may have **ε transitions.** | In DFA, There is no **ε transitions.** |
| There is a Space tradeoff, it occupies less space | It occupies more space to execute (Much bigger than NFA) |

# Recognition of Tokens - Finite Automata

- NFA or DFA ban be represented by a **transition graph**, where the nodes are states and the labeled edges represent the transition function. There is an edge labeled a from state s to state t if and only if t is one of the next states for state s and input a.



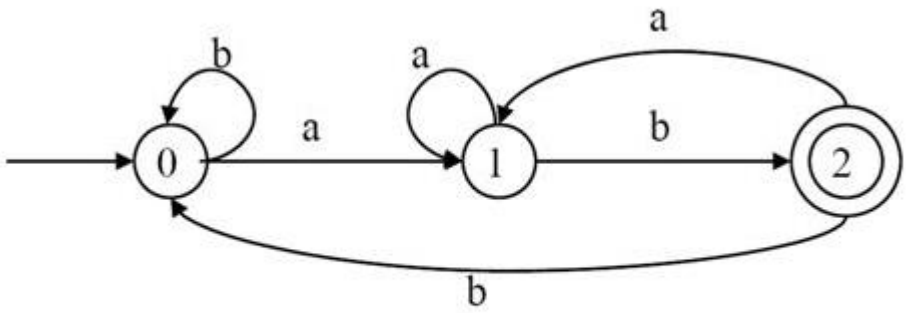

- We can also represent an NFA or DFA by a **transition table**, whose rows correspond to states, and whose columns correspond to the input symbols.

| State | Input Symbol | |
|---|---|---|
| | *a* | *b* |
| 0 | {0, 1} | {0} |
| 1 | – | {2} |
| 2 | – | {3} |

# Recognition of Tokens – Regular Expression to DFA

**Simulating a DFA**

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```




DFA for (a|b) * a b

## Converting a Regular Expression Directly to a DFA

Algorithm: Construction of a DFA from a regular expression r.

INPUT: A regular expression r.

OUTPUT: A DFA D that recognizes L(r).

METHOD :

1. Construct a **syntax tree T from the augmented regular expression (r)#** .

2. Compute **nullable, firstpos, lastpos** and **followpos** for T.

3. **Construct Dstates**, the set of states of DFA D, and Dtran, the transition function for D**.** The states of D are sets of positions in T. Initially, each state is "unmarked," and a state becomes "marked" just before we consider its out-transitions. The start state of D is firstpos(no), where node no is the root of T. The accepting states are those containing the position for the endmarker symbol #.

# Recognition of Tokens – Regular Expression to DFA

## Rules for computing nullable, firstpos and lastpos

| Node $n$ | $nullable(n)$ | $firstpos(n)$ | $lastpos(n)$ |
|---|---|---|---|
| A leaf labeled by $\varepsilon$ | **true** | $\varnothing$ | $\varnothing$ |
| A leaf with position $i$ | **false** | $\{i\}$ | $\{i\}$ |
| $\mathbf{n = c_1 \mid c_2}$ | $nullable(c_1)$ **or** $nullable(c_2)$ | $firstpos(c_1) \cup firstpos(c_2)$ | $lastpos(c_1) \cup lastpos(c_2)$ |
| $\mathbf{n = c_1\ c_2}$ | $nullable(c_1)$ **and** $nullable(c_2)$ | **if** ( $nullable(c_1)$ ) $firstpos(c_1) \cup firstpos(c_2)$ **else** $firstpos(c_1)$ | **if** ( $nullable(c_2)$ ) $lastpos(c_1) \cup lastpos(c_2)$ **else** $lastpos(c_2)$ |
| $\mathbf{n = c_1^*}$ | $true$ | $firstpos(c_1)$ | $lastpos(c_1)$ |

**Rules for Computing followpos:**

There are only two ways that a position of a regular expression can be made to follow another

1. If n is a **cat-node** with left child C1 and right child C2, then for every position ***i*** in lastpos(C1), all positions in firstpos(C2) are in followpos(i).

2. If n is a **star-node**, and ***i*** is a position in lastpos(n), then all positions in firstpos(n) are in followpos(i).

# Recognition of Tokens – Regular Expression to DFA

**Construct DFA for the regular expression (a|b)*abb**

Step1:

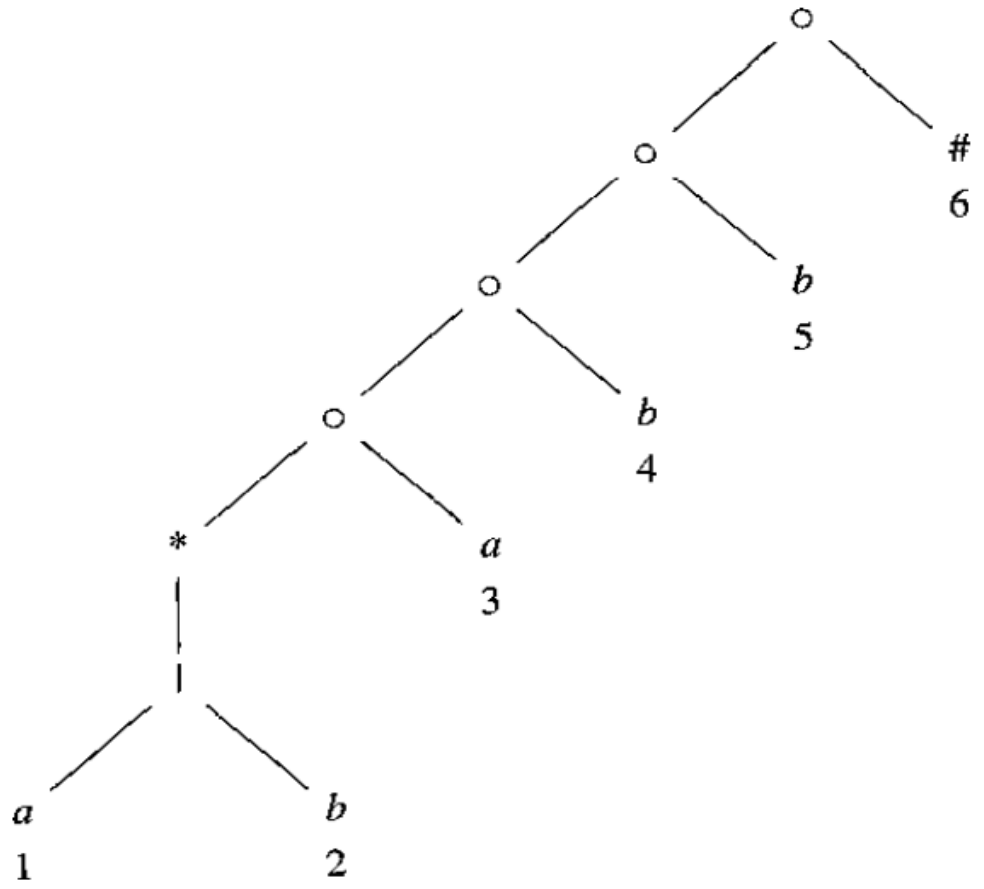
Convert the regular expression r into augmented regular expression (r)#.

**(a|b)abb#**

Step2:

Construct a syntax tree T from the augmented regular expression (r)#

**(a|b) a b b #**

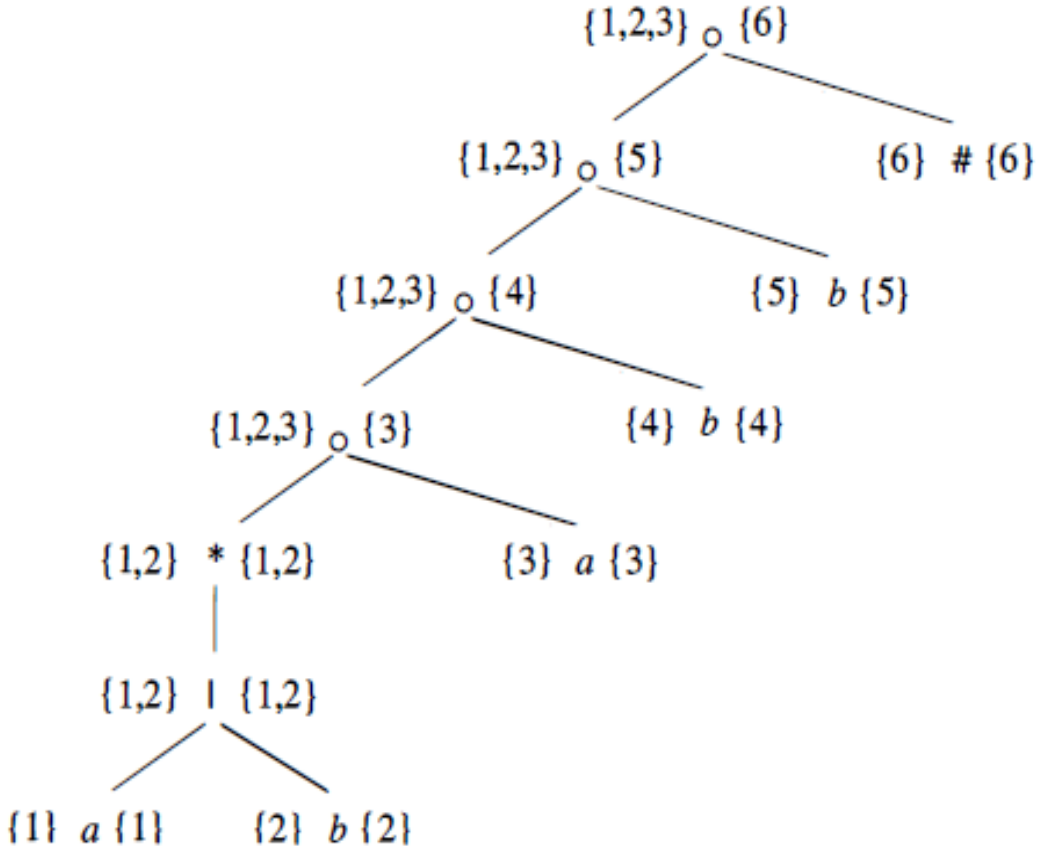


**Construct DFA for the regular expression (a|b)*abb**

Step3:

Compute nullable, firstpos and lastpos

| NODE $n$ | *followpos(n)* |
|---|---|
| 1 | {1,2,3} |
| 2 | {1,2,3} |
| 3 | {4} |
| 4 | {5} |
| 5 | {6} |
| 6 | 0 |



**Construct DFA for the regular expression (a|b)*abb**

Step4:

Construct Dstates

The value of firstpos for the root of the tree is {1,2,3} , so this set is the start state of D.

A = {1,2,3}

| | | |
|---|---|---|
| Dtran[A, a] = followpos(l) U followpos(3) | = {1,2,3,4} | = B |
| Dtran[A, b] = followpos(2) | = {1,2,3} | = A |
| Dtran[B, a] = followpos(l) U followpos(3) | = {1,2,3,4} | = B |
| Dtran[B, b] = followpos(2) U followpos(4) | = {1,2,3,5} | = C |
| Dtran[C, a] = followpos(l) U followpos(3) | = {1,2,3,4} | = B |
| Dtran[C, b] = followpos(2) U followpos(5) | = {1,2,3,6} | = D |
| Dtran[D, a] = followpos(l) U followpos(3) | = {1,2,3,4} | = B |
| Dtran[D, b] = followpos(1) | = {1,2,3} | = A |

# Recognition of Tokens – Regular Expression to DFA

**Construct DFA for the regular expression (a|b)*abb**

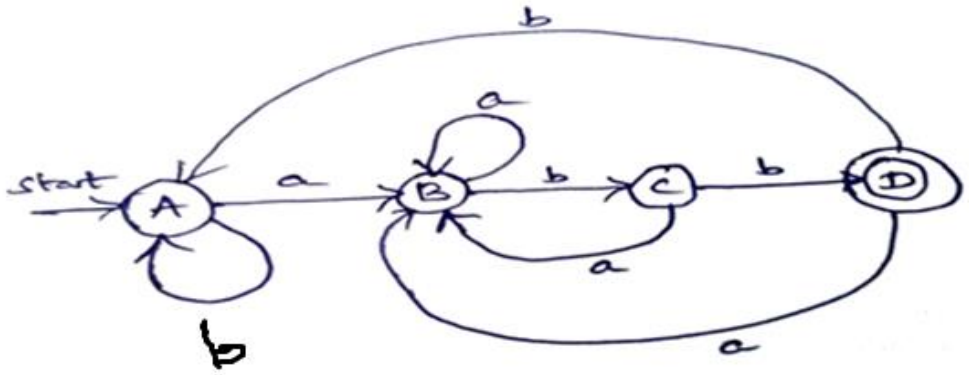
Step5:

Construct DFA Table

| States | Input string | |
|---|---|---|
| | a | b |
| A | B | A |
| B | B | C |
| C | B | D |
| D | B | A |

Step6:

Draw DFA Diagram



# Recognition of Tokens – Regular Expression to DFA

**Construct DFA for the following regular expression**

1. a*(a|b)abb(a|b)*
2. ab(a|b)(a|b)*b*a
3. (a|b|c)c*abb*
4. (0|1)101(1|0)*
5. 1*0*10100*