23CS6401 -COMPILER DESIGN

UNIT I

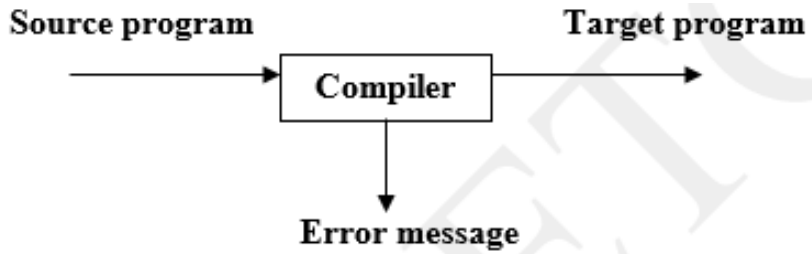
INTRODUCTION TO COMPILER DESIGN

INTRODUCTION TO COMPILER

Compilers – Analysis of the Source Program – Phases of a Compiler- Types of Compiler-Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens -Finite Automata – Regular Expressions to Automata – Minimizing DFA. – Language for Specifying Lexical Analyzers - Design of Lexical analyzer generator(LEX) - Recent trends in Compiler Design.

COMPILER

Compiler is a translator which is used to convert programs in high-level language to low-level language. It translates the entire program and also reports the errors in source program encountered during the translation.



Properties of a Compiler

- The compiler itself must be bug free.
- It must generate correct machine code.
- The generated machine code must run fast.

ANALYSIS OF THE SOURCE PROGRAM

The source program can be analysed in three phases –

1) Linear analysis: In this type of analysis the source string is read from left to right and grouped into tokens.

For example- Tokens for a language can be identifiers, constants, relational operators, keywords.

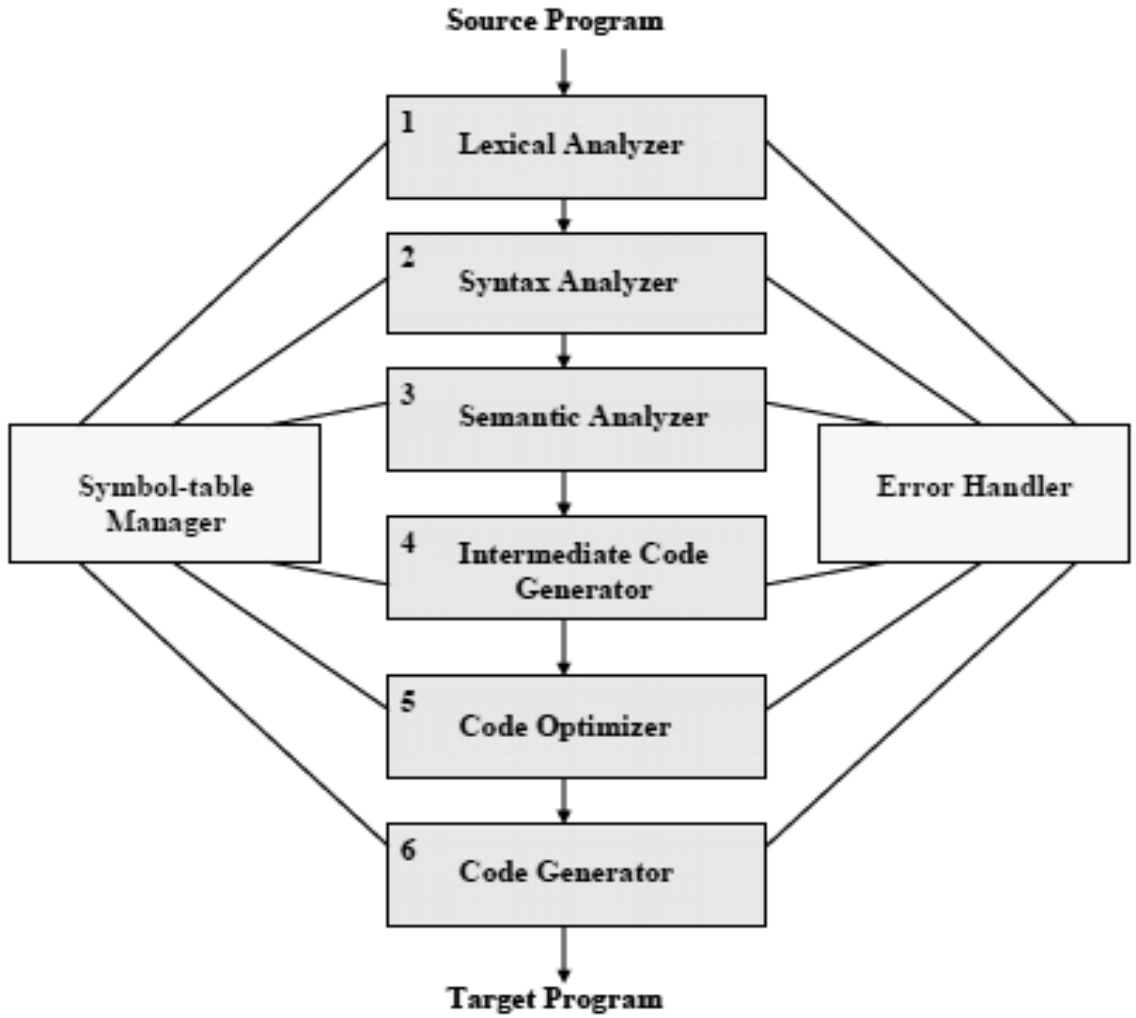
2) Hierarchical analysis: In this analysis, characters or tokens are grouped hierarchically into nested collections for checking them syntactically.

3) Semantic analysis: This kind of analysis ensures the correctness of meaning of the program.

PHASES OF A COMPILER

The different phases of compiler are as follows,

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation
5. Code generation
6. Code optimization


Source Program
1
Lexical Analyzer
2
Syntax Analyzer
3
Semantic Analyzer
Symbol-table Manager
Error Handler
4
Intermediate Code Generator
5
Code Optimizer
6
Code Generator
Target Program

1. Lexical analysis

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

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

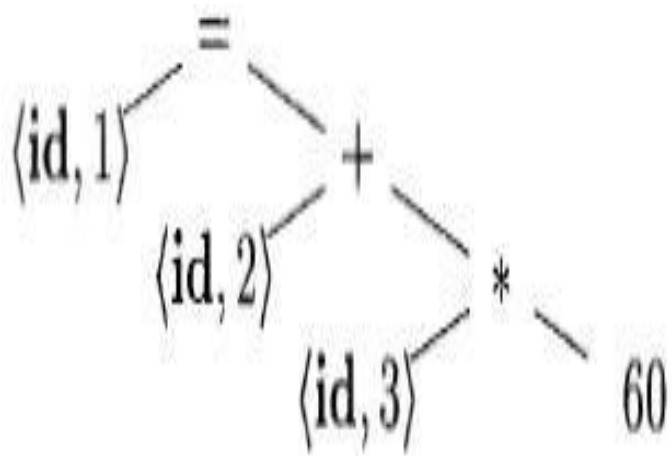
1) Identifiers 2) Keywords 3) Operators 4) Special symbols 5) Constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

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

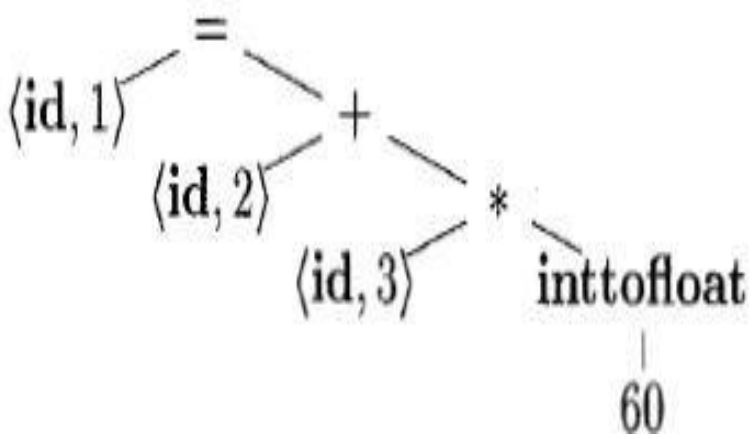
2. Syntax analysis

- The second phase of the compiler is ***syntax analysis or parsing or hierarchical analysis*.**
- Parser converts the tokens produced by lexical analyzer into a tree like representation called parse tree.
- The hierarchical tree structure generated in this phase is called syntax tree or parse tree.
- A parse tree describes the syntactic structure of the input.
    - *Input: Tokens*
    - *Output: Syntax tree*
- In a syntax tree, each interior node represents an operation and the children of the node represent the arguments of the operation


=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
60

3. 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.



4. Intermediate code generation

- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation.
- Three-address code is one of the intermediate representations, 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 for position = initial + rate * 60

t1 = int to float(60)

t2 = id3 * t1

t3 = id2 + t2

id1 = t3

5. Code optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- Optimization has to improve the efficiency of code so that the target program running time and consumption of memory can be reduced.

t1 = id3 * 60.0

id1 = id2 + t1

- 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 int to float operation can be eliminated by replacing the integer 60 by the floating-point number 60.0.

6. Code Generation

- Code generation is the final phase of compiler
- The code generator takes as input from code optimization and produces target code or object code as output.
- If the target language is machine code, then the registers or memory locations are selected for each of the variables used by the program.
- The intermediate instructions are translated into sequences of machine instructions.
- For example, using registers R1 and R2, the intermediate code might get translated into the machine code

LDF R2, id3

MULF R2, R2 , #60.0

LDF Rl, id2

ADDF Rl, Rl, R2

STF id1, Rl

TYPES OF COMPILER

Incremental Compiler

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

The basic features of incremental compiler are,

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

Cross Compiler

Basically there exists three types of languages

1. Source language i.e. the application program.
2. Target language in which machine code is written.

ROLE OF LEXICAL ANALYZER

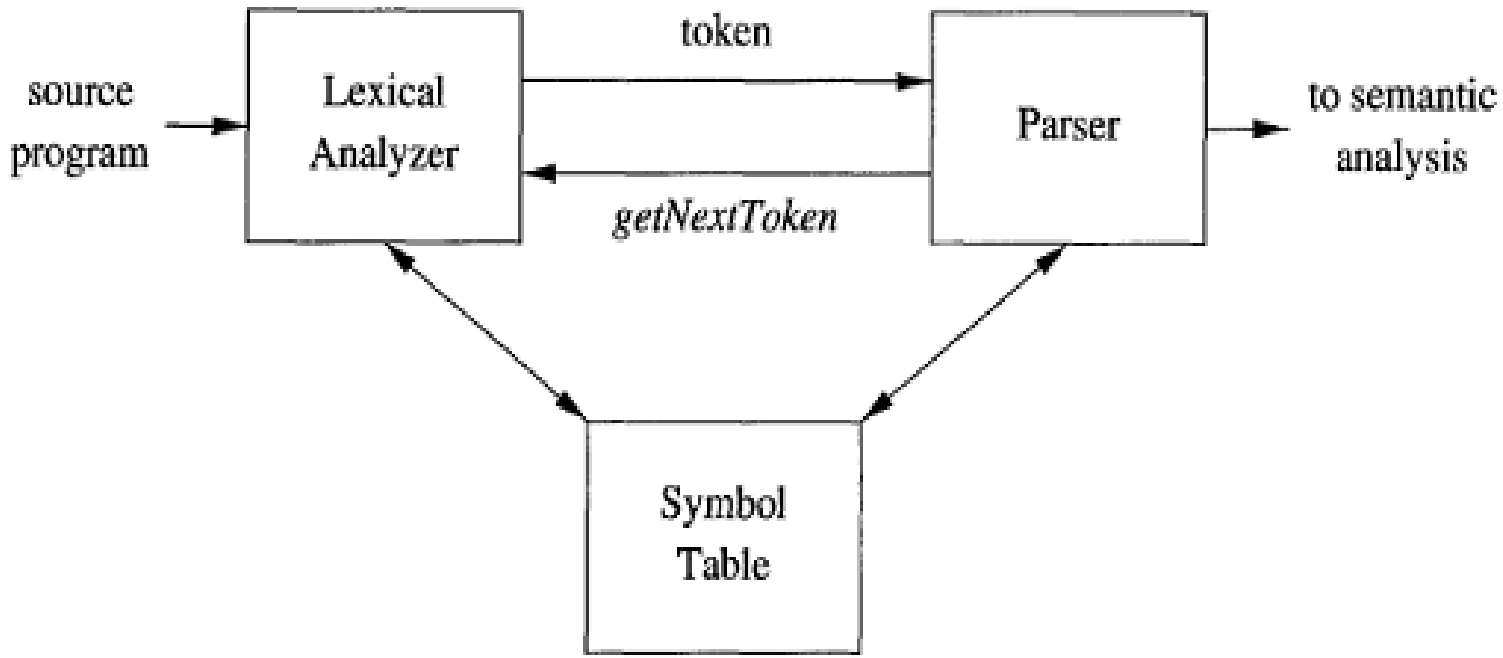
The main task of lexical analyser is to read the source program, scan the input characters, group them into lexemes and produce the token as output.

The stream of tokens is sent to the parser for syntax analysis. It interacts with the symbol table. When it discovers a lexeme consisting as identifier, it needs to enter that lexeme into symbol table.

In some cases, information regarding kind of identifier may be read from the symbol table by lexical analyser to assist it in determining the proper token it must pass to the parser.

The interaction between lexical analyser and the parser is implemented by the getNextToken command

Interaction of lexical analyzer with parser



Functions of Lexical analyser

- The following are the some other tasks performed by the lexical analyser
- It eliminates whitespace and comments.
- It generates symbol table which stores the information about identifiers, constants encountered in the input.
- It keeps track of line numbers.
- It reports the error encountered while generating the tokens.
- It expands the macros when the source program uses a macro-pre-processor

Processes of lexical analyser

• Sometimes, lexical analyser is divided into a cascade of two processes:

Scanning: It performs reading of input characters, removal of white spaces and comments

Tokenization: It is the more complex portion, where the scanner produces the sequence of tokens as output

Tokens, Patterns and Lexemes

Token: Token is a valid sequence of characters which are given by lexeme. In a programming language, Keywords, constant, identifiers, numbers, operators and punctuations symbols are possible tokens to be identified.**Pattern:** Pattern describes a rule that must be matched by sequence of characters (lexemes) to form a token. It can be defined by regular expressions or grammar rules

Lexeme: Lexeme is a sequence of characters that matches the pattern for a token i.e., instance of a token.

Lexical error handling approaches

- Lexical errors can be handled by the following actions:
- Deleting one character from the remaining input.
- Inserting a missing character into the remaining input.
- Replacing a character by another character.
- Transposing two adjacent characters.

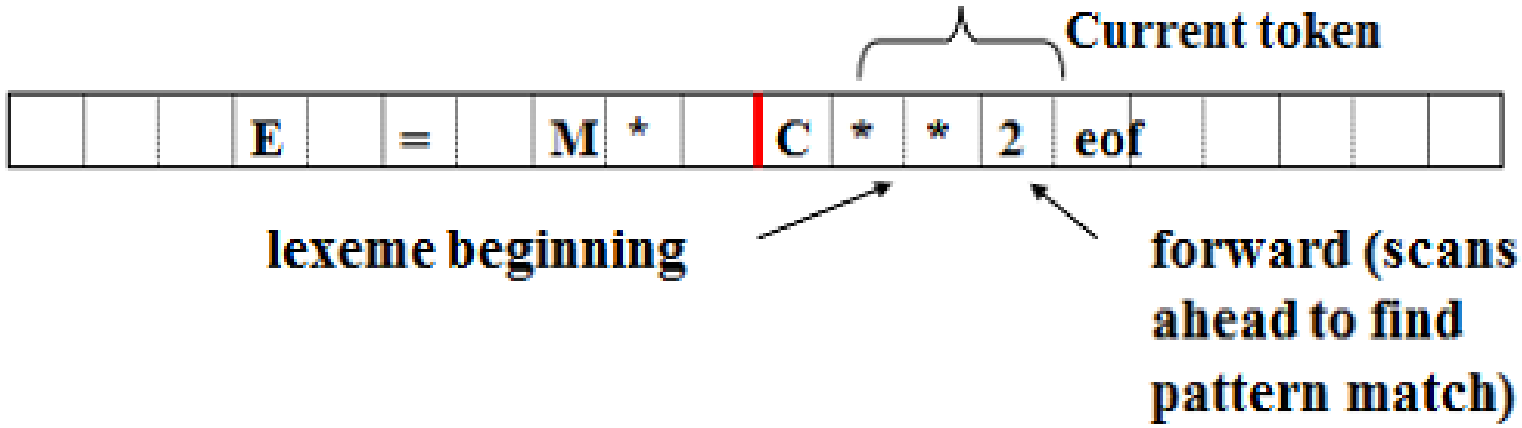
INPUT BUFFERING

Some efficiency issues concerned with the buffering of input.

- Speed of lexical analysis is a concern.
- Lexical analysis needs to look ahead several characters before a match can be announced

To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.

- A two-buffer input scheme that is useful when lookahead on the input is necessary to identify tokens.
- Techniques for speeding up the lexical analyser, such as the use of sentinels to mark the buffer end.



Buffer Pairs

SPECIFICATION OF TOKENS

1. Regular Definition of Tokens
   - Defined in regular expression

   e.g. Id → letter(letter|digit)

   letter →A|B|…|Z|a|b|…|z

   digit →0|1|2|…|9

Notes: Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for sets of strings.

2. Regular Expression & Regular language

Regular Expression

- A notation that allows us to define a pattern in a high level language.

Regular language

- Each regular expression r denotes a language L(r) (the set of sentences relating to the regular expression r)

Notes: Each word in a program can be expressed in a regular expression

3. The rule of regular expression over alphabet Σ

1) ε is a regular expression that denote $\{\varepsilon\}$

- ε is regular expression
- $\{\varepsilon\}$ is the related regular language

2) If *a* is a symbol in Σ, then *a* is a regular expression that denotes$\{a\}$

- a is regular expression
- {a} is the related regular language

3) Suppose α and β are regular expressions, then $\alpha|\beta$, $\alpha\beta$, α^* ,β^* is also a regular expression

Notes: Rules 1) and 2) form the basis of the definition; rule provides the inductive step.

4. Algebraic laws of regular expressions

1) $\alpha|\beta = \beta|\alpha$

2) $\alpha|(\beta|\gamma)=(\alpha|\beta)|\gamma \quad \alpha(\beta\gamma) =(\alpha\ \beta)\gamma$

3) $\alpha(\beta|\gamma\ )= \alpha\beta\ |\ \alpha\gamma \qquad (\alpha|\beta)\gamma= \alpha\gamma|\ \beta\gamma$

4) $\varepsilon\alpha = \alpha\varepsilon = \alpha$

5) $(\alpha^*)^*=\alpha^*$

6) $\alpha^*=\alpha^+|\varepsilon \qquad \alpha^+ = \alpha\ \alpha^* = \alpha^*\alpha$

7) $(\alpha|\beta)^*= (\alpha^*\ |\beta\ ^*)^*= (\alpha^*\ \beta^*)^*$

8) If $\varepsilon \notin L(\gamma)$,then

$\alpha= \beta|\ \gamma\ \alpha \qquad \alpha= \gamma^*\ \beta$

$\alpha= \beta|\ \alpha\ \gamma \qquad \alpha= \beta\ \gamma^*$

5. Notational Short-hands

a)One or more instances

(r)$^{+}$ digit^{+}

b)Zero or one instance

r? is a shorthand for r|ε (E(+|-)?digits)?

c)Character classes

[a-z] denotes a|b|c|…|z

[A-Za-z] [A-Za-z0-9]

RECOGNITION OF TOKENS

- Recognition of token explains how to take the patterns for all needed tokens. It builds a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns. Rules for conditional statement or branching statement can be given as follows

Stmt→ if expr then stmt

| If expr then stmtelse stmt

| ϵ

Expr→term relop term

| term

Term→id

|number

Tokens, their patterns and attributes values

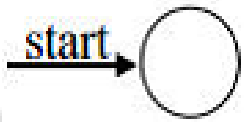
Lexeme	Token Name	Attribute Value
Any ws	_	_
If	if	_
Then	Then	-
Else	Else	-
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	FE

Transition Diagram

Transition Diagram has 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 labelled by a symbol or set of symbols. If we are in one state **s**, and the next input symbol is **a**, we look for an edge out of state **s** labeled by **a**.

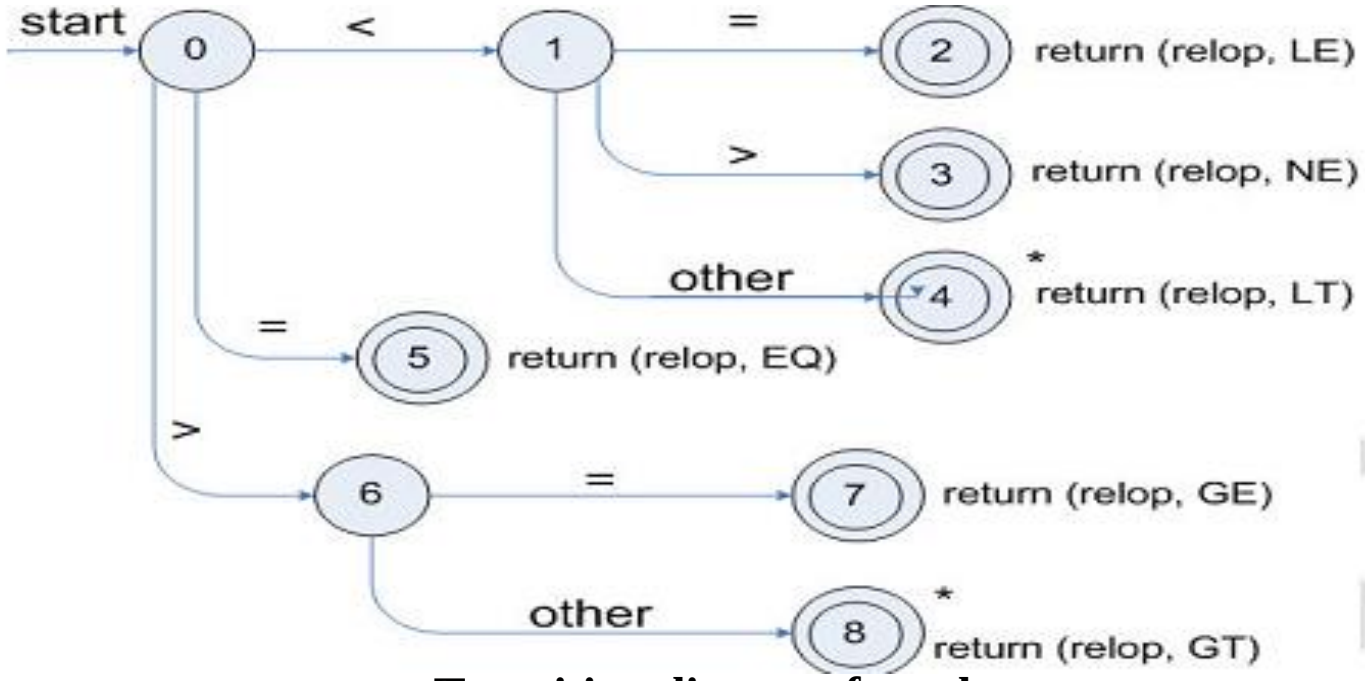
Components of Transition Diagram

1. One state is labeled the Start State; it is the initial state of the transition diagram where control resides when we begin to recognize a token. 

2. Positions in a transition diagram are drawn as circles and are called states. 
3. The states are connected by arrows called edges. Labels on edges are indicating the input characters. 
4. The Accepting states in which the tokens has been found. 
5. Retract one character use * to indicate states on which this input retraction.

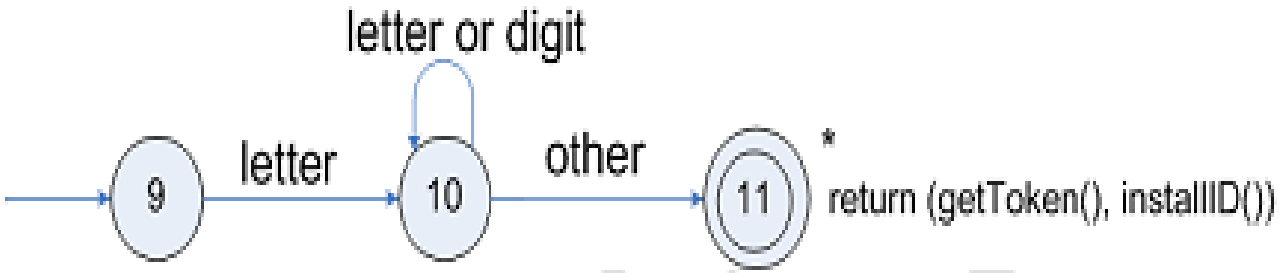
Examples:

1. Transition diagram for relop



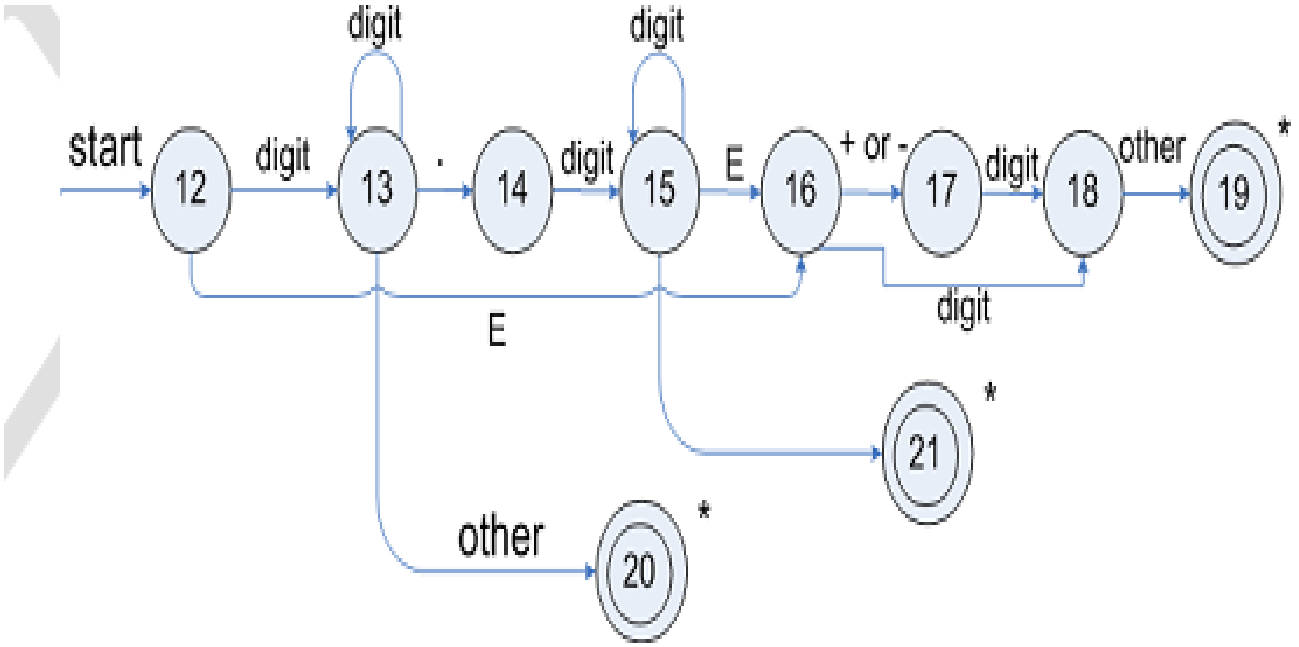
Transition diagram for relop

2. Transition diagram for reserved words and identifiers



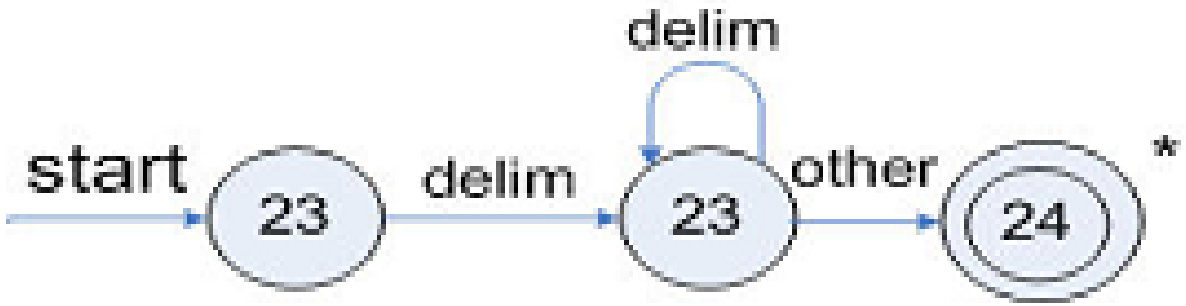
Transition diagram for id's and keywords

3. Transition diagram for unsigned numbers



Transition diagram for unsigned numbers

4. Transition diagram for whitespace



Transition diagram for whitespace

FINITE AUTOMATA

Finite automata are recognizer; they simply say “yes” or “no” about each possible input string.

There are two types of finite automata:

- a) Nondeterministic finite automata (NFA)
- b) Deterministic finite automata (DFA)

The mathematical model of finite automata consists of:

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

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), $Q \times \Sigma \rightarrow Q$

Construction of Finite Automata

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

States: States of FA are represented by circles. State names are written inside circles.

Start state: The state from where the automata start is known as the start state. Start state has an arrow pointed towards it.

Intermediate states: All intermediate states have at least two arrows; one pointing to and another pointing out from them.

Final state: If the input string is successfully parsed, the automaton is expected to be in this state. Final state is represented by double circles.
Transition: The transition from one state to another state happens when a desired symbol in the input is found.

NONDETERMINISTIC FINITE AUTOMATA (NFA)

A nondeterministic finite automaton (NFA) consists of:

1. A finite set of states S.

2. A set of input symbols Σ, the input alphabet. We assume that e, which stands for the empty string, is never a member of Σ.

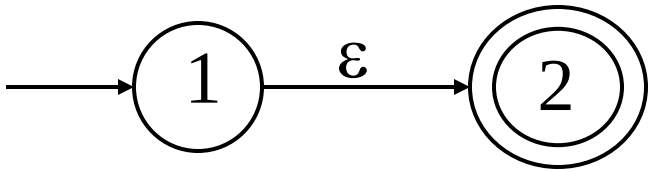
3. A transition function that gives, for each state, and for each symbol in Σ U {e} a set of next states.

4. A state so from S that is distinguished as the start state (or initial state).

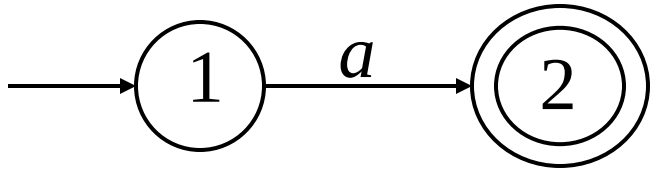
5. A set of states F, a subset of S that is distinguished as the accepting states (or final states).

Rules

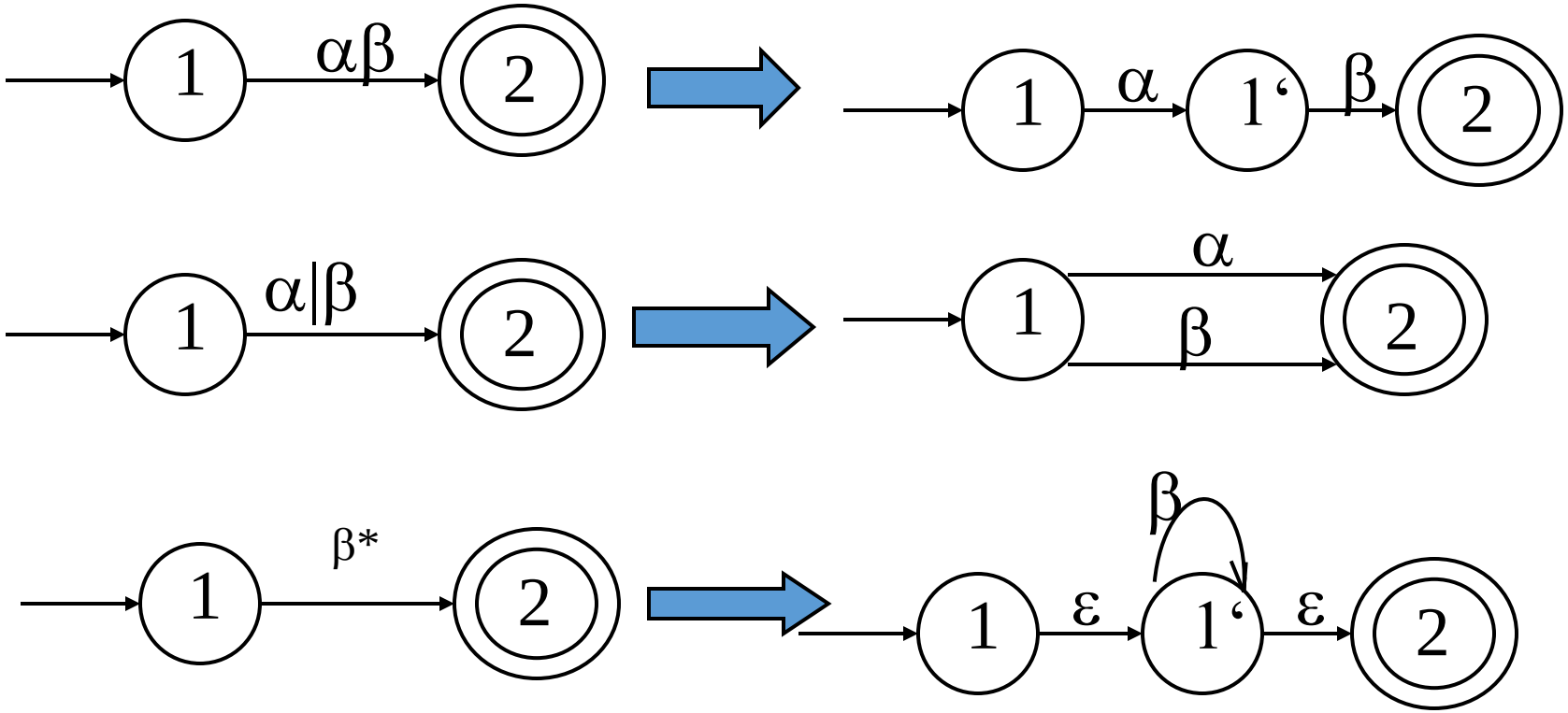
1. For ε,



2. For *a* in Σ,



3. Rules for complex regular expressions



DETERMINISTIC FINITE AUTOMATA (DFA)

- The finite automata which satisfy the following conditions can be called as Deterministic finite automata
- It should have one start state
- It should have one or more accepting or final states
- It should not allow € transitions
- It should allow only one transitions on a particular i/p symbol

Formal Definition of DFA

M= (Q, Σ, δ, q0, F)

1) The finite set of states which can be denoted by Q.

2) The finite set of input symbols Σ.

3) The start state q_0 such that $q_0 \epsilon$ Q.

4) A set of final states F such that F ϵ Q.

5) The mapping function or transition function denoted by δ.

REGULAR EXPRESSIONS TO AUTOMATA

CONVERSION OF AN NFA TO A DFA: The Subset Construction Algorithm

- The algorithm for constructing a DFA from a given NFA such that it recognizes the same language is called subset construction. The reason is that each state of the DFA machine corresponds to a set of states of the NFA.
- The DFA keeps in a particular state all possible states to which the NFA makes a transition on the given input symbol. In other words, after processing a sequence of input symbols the DFA is in a state that actually corresponds to a set of states from the NFA reachable from the starting symbol on the same inputs.

Three operations that can be applied on NFA states

OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}$ ϵ-*closure*(s).
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some state s in T.

CONSTRUCTION OF NFA FROM REGULAR EXPRESSION

- The McNaughton-Yamada-Thompson algorithm is used to convert a regular expression to an NFA.

Thompson algorithm

Input: A regular expression r over alphabet Σ

Output: An NFA N accepting (r)

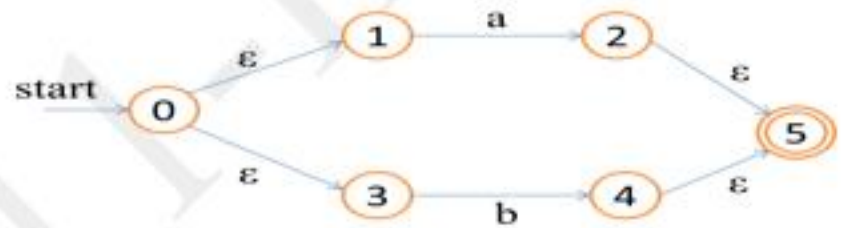
Method:

- Begin by parsing r into its constituent sub-expressions
- The rules for constructing an NFA consist of basis rules for handling sub expressions with no operators
- Induction rules for constructing larger NFA‟s from the NFA‟s for the immediate Sub expressions of a given expression with operators

1. R = ε

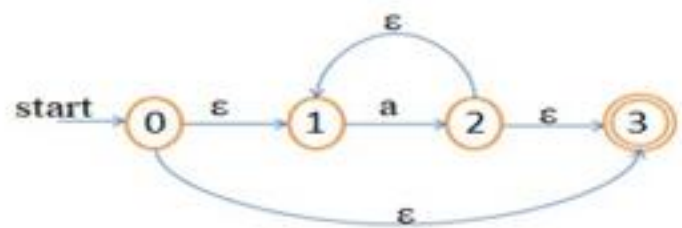
2. R = a

3. R= a | b

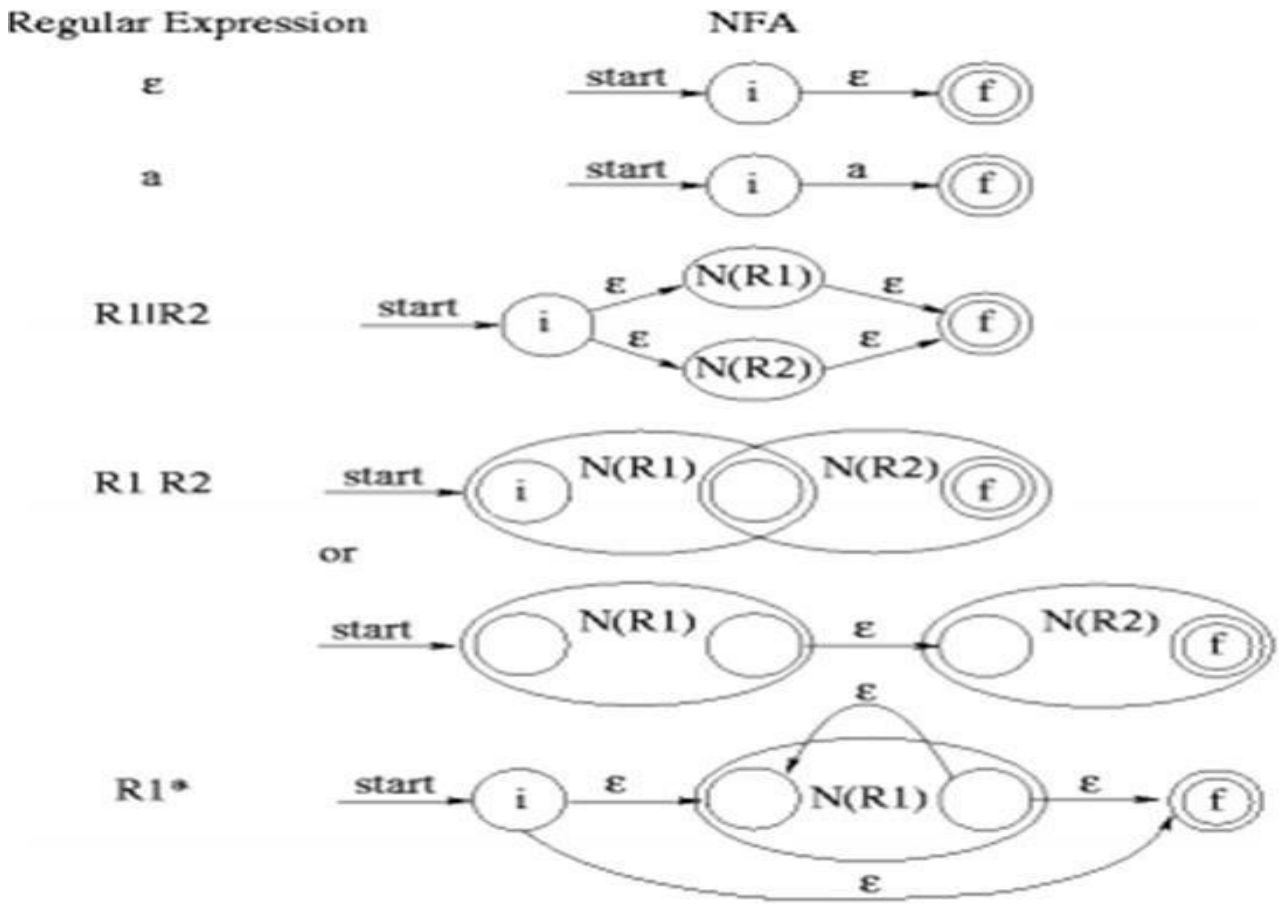


4. R=ab

5. R= a*



Thompson's construction method



MINIMIZING DFA

- Lexical analyser can be implemented by DFA. The number of states of constructed DFA must be minimized as for each state entry is made in a table which describes the lexical analyser
- Minimization of DFA focuses on reducing the number of states in the given finite automata
- The states which are equivalent are grouped thereby enabling the reduction of states
- Two states are said to be equivalent if on taking the same input symbol they transit to same group of states (final or non-final). Otherwise, they are distinguishable states

Algorithm:

1. Start with initial partition Π with two groups, F and Q-F, the accepting and non-accepting states of D

2. Construct a new partition Π new as follows:

Let Πnew=Π

For each group G of Π

{partition G into subgroups such that two states p and q are in same group iff for all input symbols a, states p and q have transitions to states in same group of Π.

Replace G in Πnew by the set of subgroups formed.}

3. If Πnew=Π stop partitioning, otherwise repeat step 2

4. Choose representative states from each group and construct finiteautomata.

Language for Specifying Lexical Analyzers

- A lexical analyzer (scanner) identifies tokens from source code
- To simplify scanner development, **specification languages** are used
- These languages describe **patterns and actions** clearly

Need for Lexical Specification Language

- Manual coding of lexical analyzer is complex
- Specification language reduces errors
- Automatically generates efficient lexical analyzers
- Easy to modify and maintain

Basic Structure of Lexical Specification

- A lexical specification has **three parts**:
- **Declarations**
- **Translation Rules**
- **Auxiliary Procedures**

Declarations Section

- Defines **regular expressions** and **named patterns**
- Declares tokens and variables
- Example: digit, letter, keyword definitions

Translation Rules

- Core part of the specification
- Format:
  Pattern → Action
- When a pattern is matched, corresponding action is executed
- Example:
  - identifier → return ID
  - number → return NUM

Auxiliary Procedures

- Contains supporting code
- Includes helper functions
- Used for error handling and symbol table operations

Regular Expressions in Lexical Specification

- Used to describe token patterns
- Examples:
  - letter = [A–Z a–z]
  - digit = [0–9]
  - identifier = letter(letter | digit)*

Example Lexical Specification

- Keywords: if, else, while
- Identifiers: letter followed by letters/digits
- Numbers: sequence of digits
- Whitespace ignored

Lex Tool (LEX)

- LEX is a popular lexical analyzer generator
- Takes lexical specification as input
- Produces a C program as output
- Works with YACC parser

Advantages

- Fast development of lexical analyzers
- Clear and readable specifications
- Automatic generation of efficient code
- Widely used in compiler construction

Applications

- Compiler and interpreter design
- Syntax highlighting
- Text processing tools
- Pattern matching applications

Design of Lexical Analyzer Generator (LEX)

- LEX is a lexical analyzer generator
- It automatically generates scanner programs
- Used to recognize tokens in source code

What is LEX?

- LEX is a tool for generating lexical analyzers
- Input: Lex specification
- Output: C program (lex.yy.c)
- Works with YACC

Need for LEX

- Manual lexical analyzer is complex
- LEX reduces coding effort
- Easy to modify and maintain
- Efficient and reliable

Structure of LEX Program

- LEX program has three sections:
- 1. Declarations
- 2. Rules
- 3. Auxiliary Functions

Declarations Section

- Defines regular expressions
- Declares tokens and variables
- Included between %{ %}

Rules Section

- Pattern → Action format
- Patterns are regular expressions
- Actions are C code statements

Auxiliary Functions

- Contains helper C functions
- Main function
- Error handling code

LEX Processing Steps

- LEX specification → lex compiler
- Generates lex.yy.c
- Compiled using C compiler
- Produces lexical analyzer

Advantages of LEX

- Automatic code generation
- Faster development
- Clear specification
- Widely used in compilers

Applications

- Compiler construction
- Text processing tools
- Syntax highlighting
- Pattern matching

Recent trends in Compiler Design.

- Compiler design has evolved with modern computing needs
- Traditional compilers focused only on correctness
- Modern compilers focus on **performance, security, and scalability**

Just-In-Time (JIT) Compilation

- Compilation happens at **runtime**
- Improves execution speed
- Uses actual program behavior for optimization
- Used in **Java JVM, .NET, JavaScript engines**

Machine Learning in Compiler Optimization

- Machine Learning helps choose best optimizations
- Replaces fixed heuristic rules
- Improves **register allocation, loop optimization, inlining**
- Makes compilers more intelligent

Domain-Specific Languages (DSLs)

- Languages designed for specific applications
- Easier coding for complex problems
- Allows domain-aware optimizations
- Used in scientific and data-intensive applications

Parallelism and Multi-Core Optimization

- Modern processors have multiple cores
- Compilers automatically generate parallel code
- Supports **threading and vectorization**
- Improves program performance

LLVM and Modular Compiler Frameworks

- LLVM is a popular compiler infrastructure
- Uses reusable optimization components
- Supports multiple programming languages
- Targets different hardware platforms

Security-Aware Compilation

- Compilers help detect security vulnerabilities
- Prevents buffer overflow and memory attacks
- Uses sanitizers and control-flow integrity
- Improves software security

Quantum Compiler Design

- New compilers for quantum computers
- Convert high-level code to quantum instructions
- Optimize qubit usage and gate operations
- Important for future computing

Heterogeneous Computing Support

- Programs run on **CPU, GPU, FPGA**
- Compiler automatically selects best hardware
- Improves speed and energy efficiency
- Used in AI and high-performance computing

Advanced Intermediate Representations

- New IRs provide multiple abstraction levels
- Example: **MLIR (Multi-Level IR)**
- Helps better analysis and optimization
- Supports modern compiler architectures

Future Directions

- AI-assisted compiler design
- Compilers for IoT and edge devices
- Energy-efficient code generation
- Secure and privacy-aware compilation