Parul® University

NAAC A++

Compiler design
Chapter-1: Overview of compilation

Prof. Arpita Vaidya
Assistant Professor
Department of Computer Science and Engineering

Parul® University

NAAC A++

Content

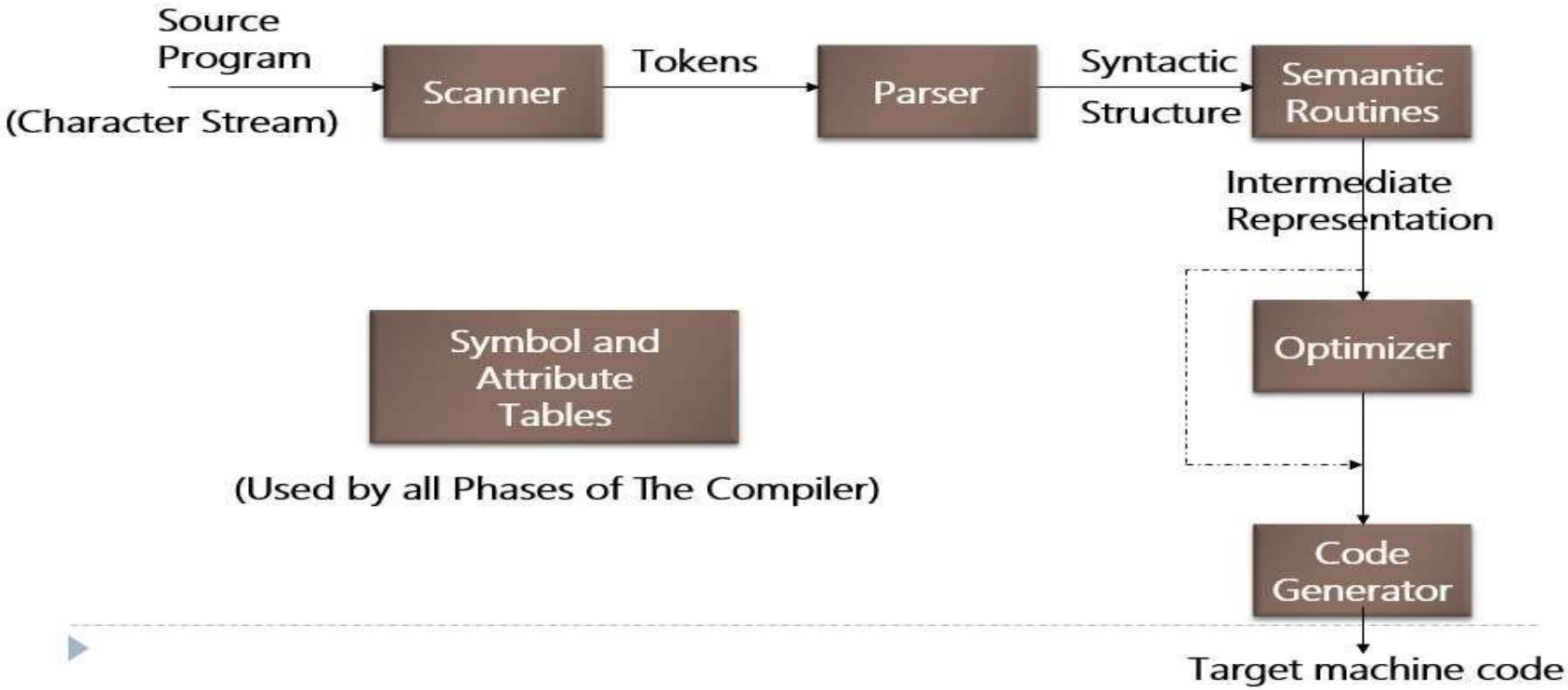
1. Compiler…….. 1
2. Structure of Compiler…… 2
3. Application of Compiler Design…… 10
4. Lexical analysis - The role of a lexical analyzer…… 11
5. Specifications of tokens……16
6. Recognition of tokens……17
7. Hand-written lexical analyser…… 20
8. Lex, Lexical of program…… 21

INDEX

Complier

- A compiler acts as a translator, transforming human-oriented programming languages into computer oriented machine languages.
- Ignore machine-dependent details for programmer

The structure of a compiler



The structure of a compiler

1 Scanner

The scanner begins the analysis of the source program by reading the input, character by character, and grouping characters into individual words and symbols (tokens)

- RE (Regular expression)
- NFA (Non-deterministic Finite Automata)
- DFA (Deterministic Finite Automata)
- LEX

The structure of a compiler

2- Parser

Given a formal syntax specification (typically as a context-free grammar [CFG]), the parse reads tokens and groups them into units as specified by the productions of the CFG being used.

As syntactic structure is recognized, the parser either calls corresponding semantic routines

directly or builds a syntax tree.

- CFG (Context-Free Grammar)
- BNF (Backus-Naur Form)
- GAA (Grammar Analysis Algorithms)
- LL, LR, SLR, LALR Parsers
- YACC

The structure of a compiler

3 - Semantic Routines

Perform two functions

- Check the static semantics of each construct
- Do the actual translation

The heart of a compiler

- Syntax Directed Translation
- Semantic Processing Techniques
- IR (Intermediate Representation)

The structure of a compiler

4 - Optimizer

- The IR code generated by the semantic routines is analyzed and transformed into functionally equivalent but improved IR code
- This phase can be very complex and slow
- Peephole optimization
- loop optimization, register allocation, code scheduling

- Register and Temporary Management
- Peephole Optimization

The structure of a compiler

5 - Code Generator

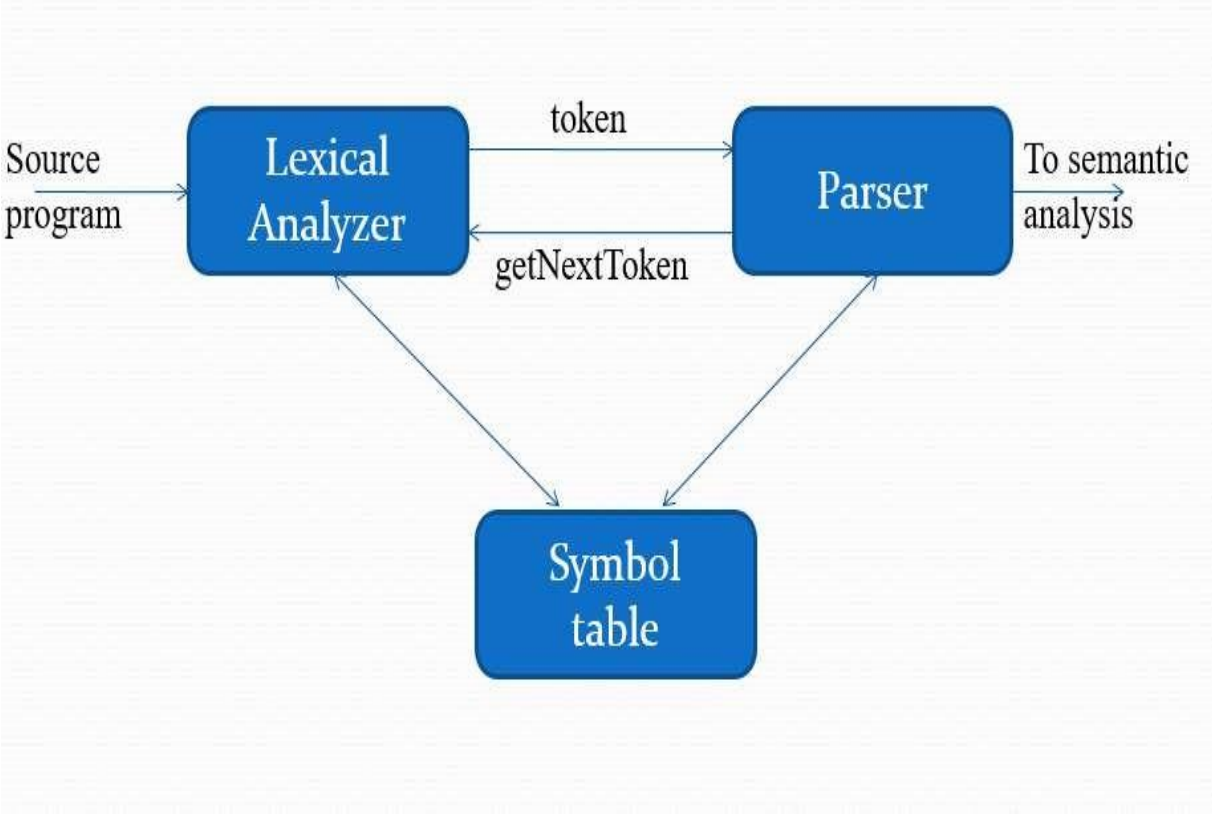
- Interpretive Code Generation
- Generating Code from Tree/Dag
- Grammar-Based Code Generator

Applications of compiler technology

Compiler technology is more broadly applicable and has been employed in rather unexpected areas.

- Text-formatting languages,
  like nroff and troff; preprocessor packages like eqn, tbl, pic
- Silicon compiler for the creation of VLSI circuits
- Command languages of OS
- Query languages of Database systems

Lexical analysis - The role of a lexical analyzer



Lexical analysis - The role of a lexical analyzer

Lexical analyzer performs below given tasks:

- Helps to identify token into the symbol table
- Removes white spaces and comments from the source program
- Correlates error messages with the source program
- Helps you to expands the macros if it is found in the source program
- Read input characters from the source program

Lexical analysis - The role of a lexical analyzer

Tokens, Patterns and Lexemes

A token is a pair a token name and an optional token value

A pattern is a description of the form that the lexemes of a token may take

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

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but " sorrounded by "	"core dumped"

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

Lexical analysis - The role of a lexical analyzer

Attributes for tokens:

E = M * C ** 2

<id, pointer to symbol table entry for E>
<assign-op>
<id, pointer to symbol table entry for M>
<mult-op>
<id, pointer to symbol table entry for C>
<exp-op>
<number, integer value 2>

Lexical analysis - The role of a lexical analyzer

Lexical errors:

A character sequence which is not possible to scan into any valid token is a lexical error. Important facts about the lexical error

- Misspelling of identifiers, operators, keyword are considered as lexical errors
- Generally, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

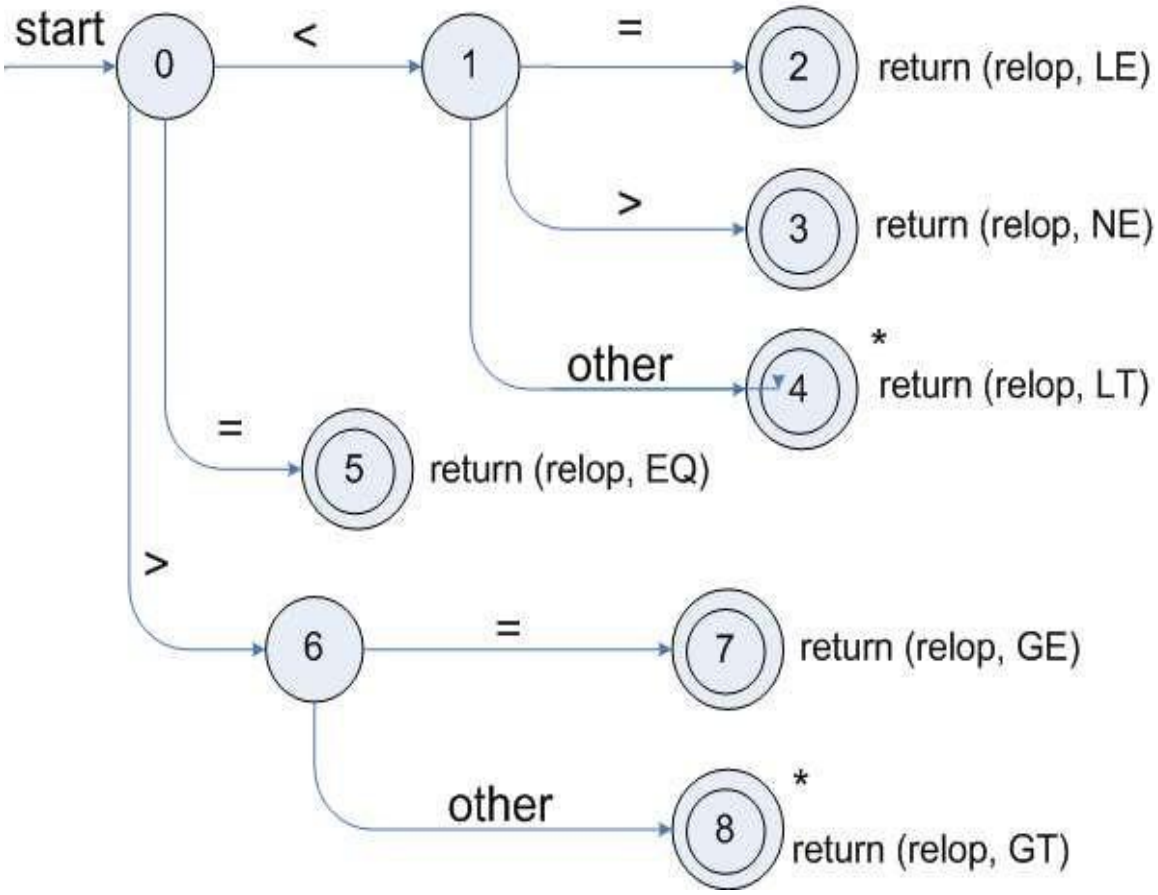
e.g: int @a1;

Above statement has lexical error because identifier (@a1) can't start with special character

Specification of tokens

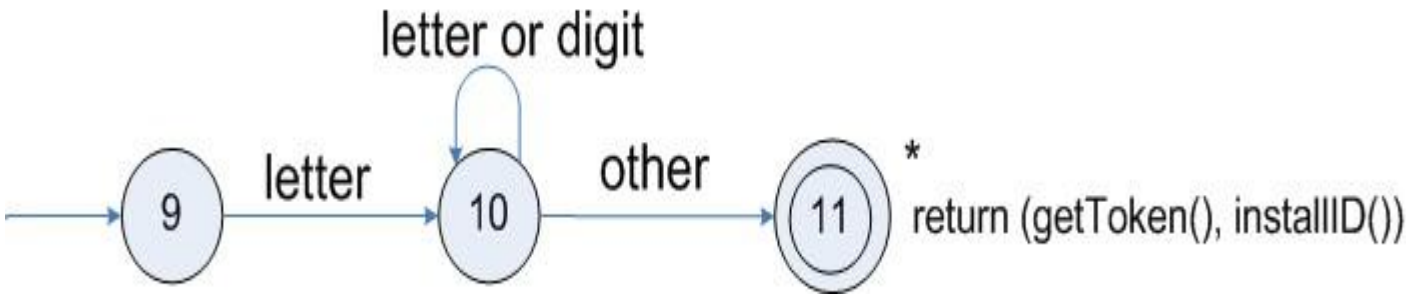
- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages Example:
  letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

Recognition of tokens- Transition diagram for relop



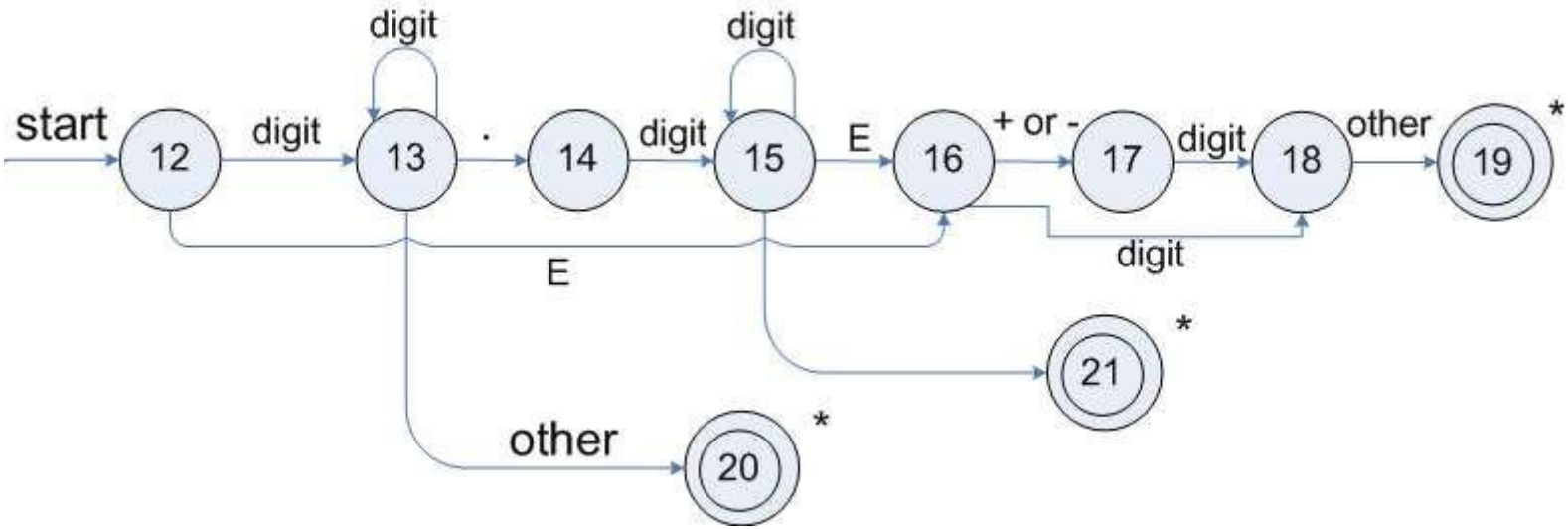
Recognition of tokens

- **Transition diagram for reserved words and identifiers :**



Recognition of tokens

Transition diagram for unsigned numbers :

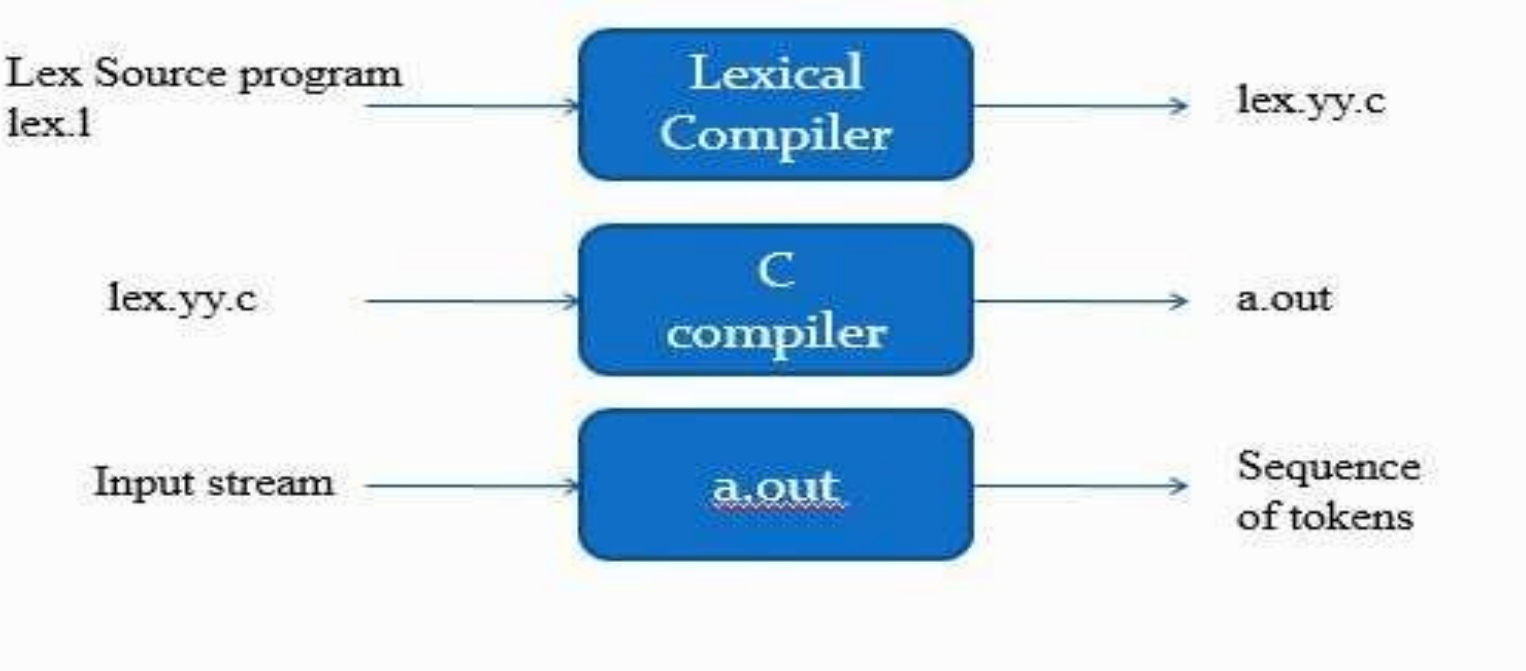


Hand-written lexical analyzers

Transition diagram for unsigned numbers :

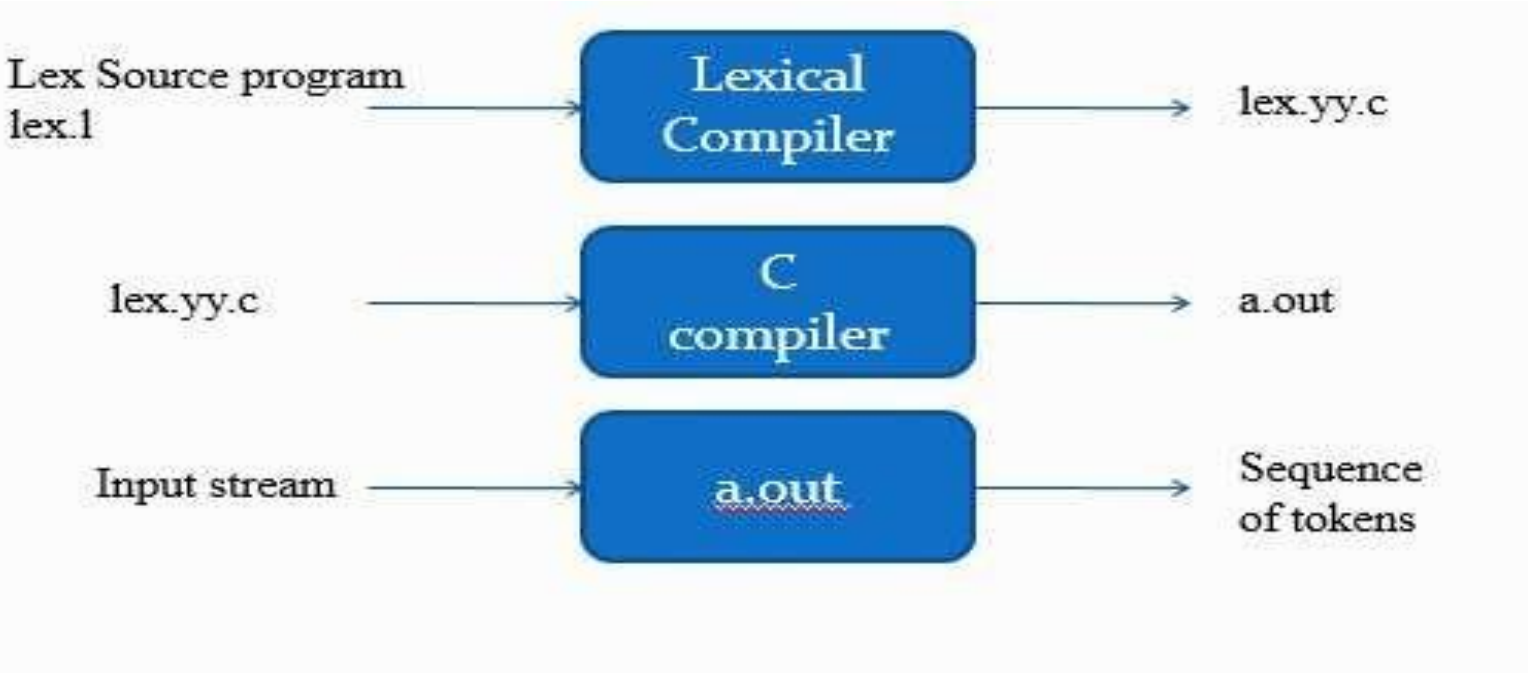
```
TOKEN getRelop()
{
        TOKEN retToken = new (RELOP)
        while (1) {    /* repeat character processing until a
                                        return or failure occurs    */
        switch(state) {
                case 0: c= nextchar();
                        if (c == '<') state =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

Lexical Analyzer Generator - Lex



Lexical Analyzer Generator - Lex

Transition diagram for unsigned numbers :



Structure of Lex programs

A lex program consists of three parts: the definition section, the rules section, and the user subroutines.

...definition section ...

%%

... rules section...

%%

... user subroutines ...

PPT Content Resources Reference Sample:

1. **Book Reference**

   Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson Education.

2. **Journal Article**

   Muchnick, S. S., & Hecht, M. S. (2018). Advances in compiler optimization: Modern approaches for language processing. *Journal of Computer Science and Engineering*, 12(4), 233–245

3. **Website Reference**

   GeeksforGeeks. (2024). *Structure of a compiler*. Retrieved November 7, 2025, from https://www.geeksforgeeks.org/structure-of-compiler/ .

   1. **Conference Presentation**

      Sharma, R., & Patel, D. (2022). *Applications of compiler technology in modern software development*. Paper presented at the International Conference on Advanced Computing and Communication Systems (ICACCS), Chennai, India.

4. **Report**

   IEEE Computer Society. (2021). *Trends in programming language translation and compiler design*.

5. **Sources**

   TutorialsPoint. (2024). *Lexical Analysis in Compiler Design.*

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/