COMPILER CONSTRUCTION
CS F363

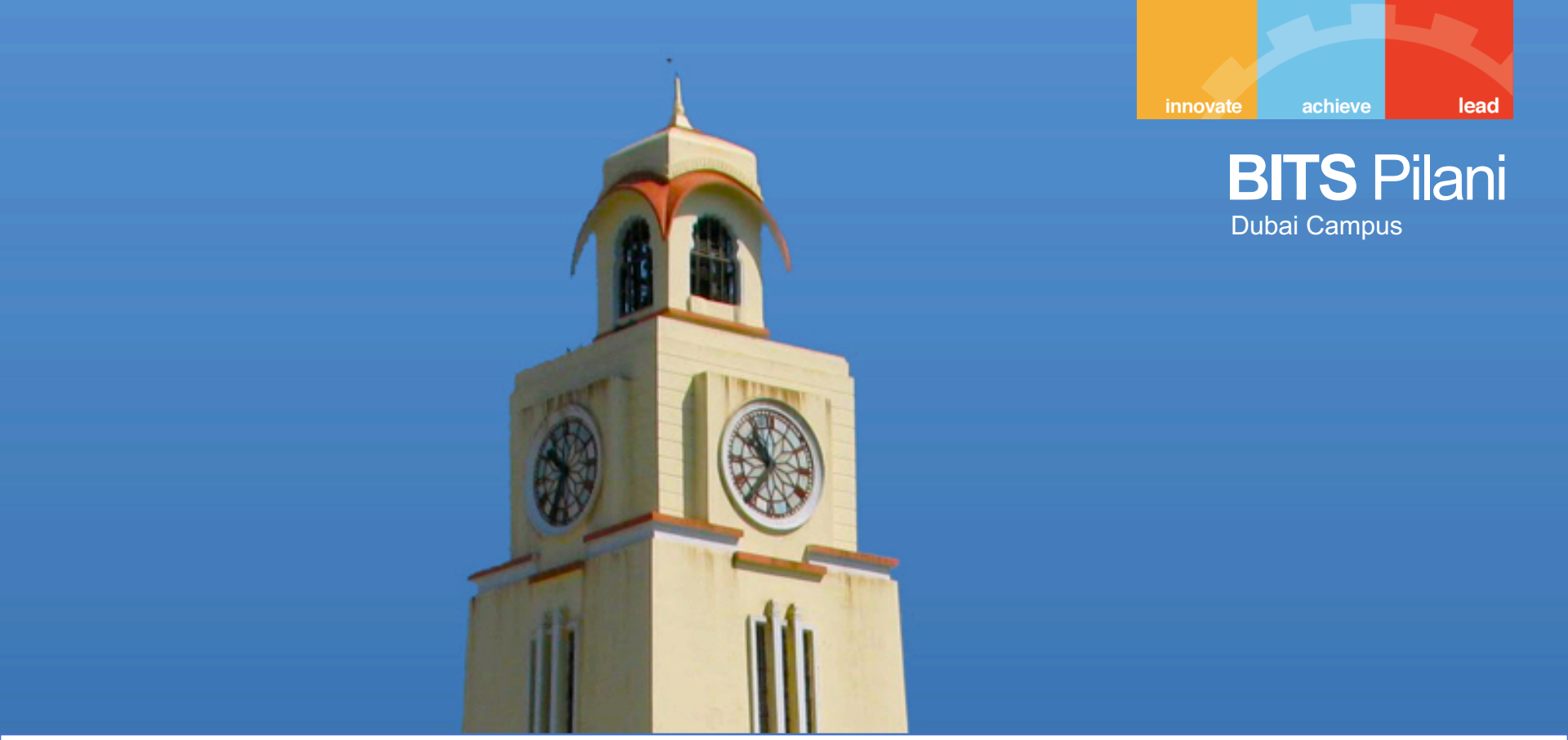
Lexical Analysis

Dr. R. Elakkiya, AP/CS

BITS Pilani
Dubai Campus

Text Book

- ✓Aho, Lam, Sethi and Ullman, “Compilers-Principles, Techniques & Tools”, Pearson/Addison-Wesley, Second Edition, 2013.



innovate achieve lead

BITS Pilani
Dubai Campus

Text Book Reading:

Chapter 3

Section 3.1 to 3.5.4

Recap

- ✓In previous lecture
  - ✓Compiler: Introduction
  - ✓Difference between Compiler, Interpreter & Translator
  - ✓Language Processing System
  - ✓Parts of Compiler
  - ✓Phases of Compiler

Overview

- ✓Lexical Analysis
  - ✓The Role of lexical analyzer - Tokens, Patterns, Lexemes, Lexical Errors
  - ✓Input Buffering - Look Ahead & Sentinels
  - ✓Specification of Tokens - Regular Expressions, Regular definitions
  - ✓Recognition of Tokens - Transition Diagrams
  - ✓The Lexical-Analyzer Generator Lex

Overview

- ✓In this Lecture….
  - ✓Lexical Analysis
    - ✓General idea
    - ✓The Role of lexical analyzer - Tokens, Patterns, Lexemes, Lexical Errors
    - ✓Input Buffering - Look Ahead & Sentinels
    - ✓Specification of Tokens - Regular Expressions, Regular definitions
    - ✓Recognition of Tokens - Transition Diagrams
    - ✓The Lexical-Analyzer Generator Lex

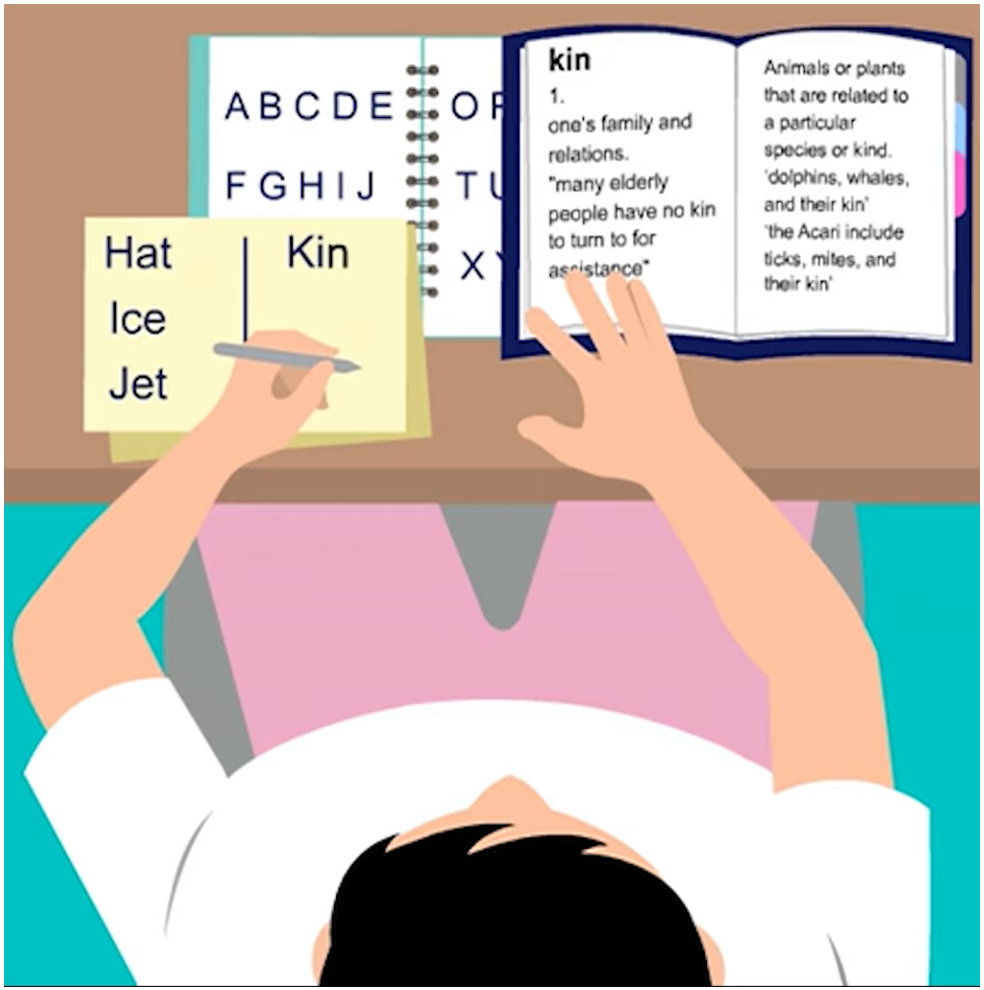
Understanding of Lexical Analyzer

Understanding of Lexical Analyzer

Understanding of Lexical Analyzer

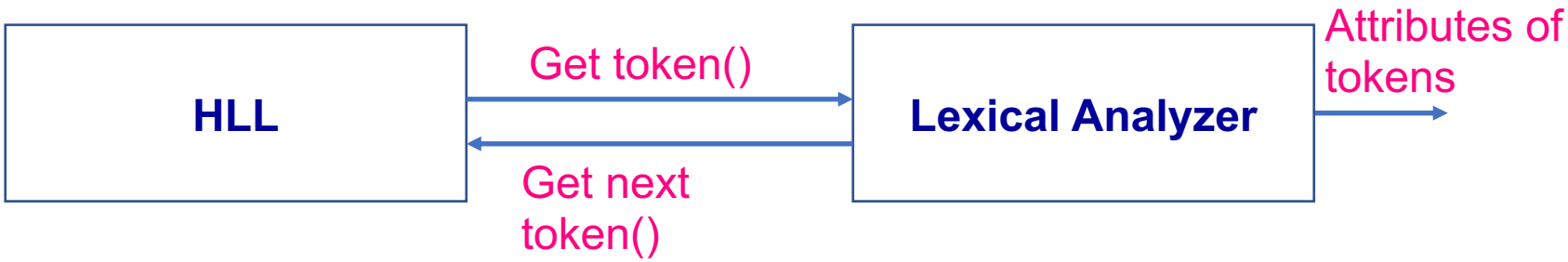
Understanding of Lexical Analyzer

Understanding of Lexical Analyzer

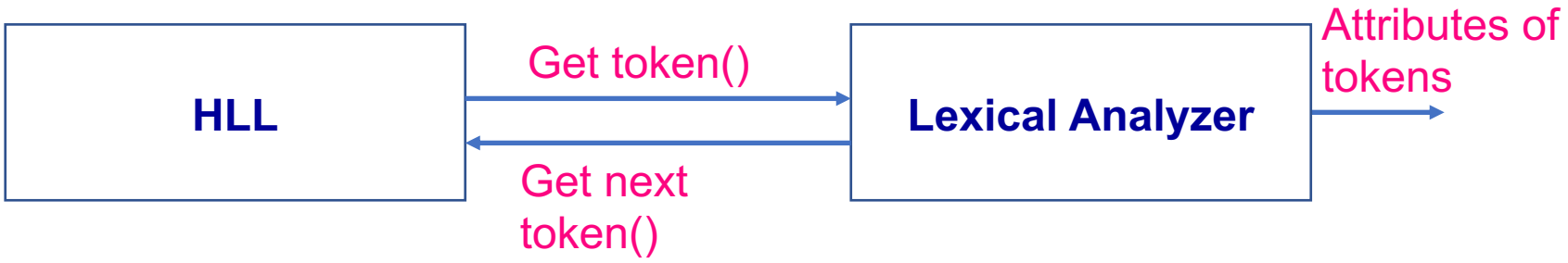


Introduction

- ✓Lexical Analyzer generate tokens
  - ✓Scanning – will get the tokens till reaches the eol
  - ✓Lexical Analysis – will identify the type of token



Introduction




```
#include <stdio.h>
void main()
{
int number;
printf("Enter an integer: ");
// reads and stores input
 scanf("%d", &number);
// displays output
printf("You entered: %d", number);
}
```

Scanning:

- Remove white spaces
- Remove new line characters (instead it will increment the line counter)
- Remove the comments

Introduction

- ✓Read input characters from the source program
- ✓Group them into “lexemes”
- ✓Output a sequence of “tokens” (for each lexeme)
- ✓Interact with the symbol table.
  - ✓Enter the lexeme into the symbol table eg. int x;
  - ✓Retrieve information

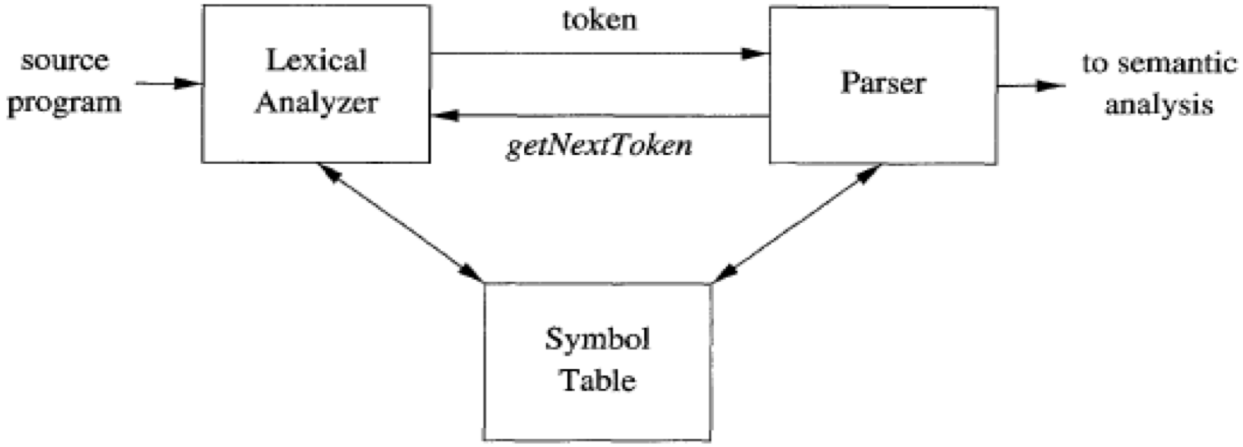


Figure 3.1: Interactions between the lexical analyzer and the parser

Role of Lexical Analyzer

- ✓Main Role is identification of lexemes
- ✓Lexeme
  - ✓ Sequence of characters that matches that pattern for a token
- ✓Pattern
  - ✓ Rule describing the format of the lexemes of a token
  - ✓ Eg For keywords it's the sequence of characters itself
- ✓Token
  - ✓ Pair consisting of "token name" and attribute (optional)
  - ✓ "token name" is a symbol representing a kind of lexical unit, and are processed by the parser

Role of Lexical Analyzer

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

Role of Lexical Analyzer

- ✓Eg C stmt: printf(“Total = %d\n”, score);
  - ✓printf is a lexeme that match the pattern for keyword printf
  - ✓score is a lexeme that match the pattern for token “id”
  - ✓“Total = %d\n” is a lexeme matching a token “literal”

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

Role of Lexical Analyzer

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

- ✓Attributes of tokens
  - ✓Additional info about the lexeme that the parser can use
  - ✓Eg int x = 10;
  - ✓ tok1:<int> tok2:<id, x> tok3:<=> tok4: <num,10> tok5:<;> or <SEMICOLON>

Role of Lexical Analyzer

- ✓ Other functions
  - ✓ Stripping of comments, whitespaces (blanks, newlines, tab)
  - ✓ Keep track of line no for error messages.
  - ✓ Macro expansion (In ‘C’ compiler, macro expansion is a separate phase or a program)

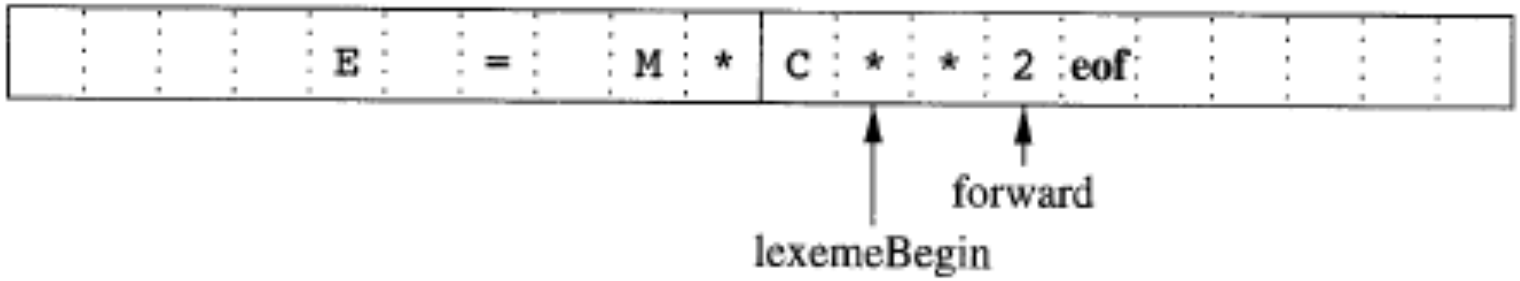
Input Buffering

- ✓HLL will be in secondary memory
- ✓While Scanning - scanner will start buffering every statement
- ✓Input buffering:
  - ✓One buffer
  - ✓Two buffer

Input Buffering

- ✓How can we speed up the task of reading the source program?
  - ✓Challenges: lookahead
    - ✓In most cases a lookahead of one character is needed
- ✓Solution
  - ✓Two-buffer scheme that handles large lookaheads safely
- ✓Two buffers are used and they are alternately reloaded
- ✓Each buffer is of the same size N, where N is usually the size of a disk block

Input Buffering



- ✓**Advantages:**
  - ✓Using one system read command we can read N characters into a buffer, rather than using one system call per character
  - ✓A special character, represented by eof marks the end of the source file
  - ✓**Note:** The eof character should be different from any possible character of the input source program

Input Buffering

- ✓Two pointers to the input are maintained:
- ✓Pointer lexemeBegin, marks the beginning of the current lexeme, whose extent we are attempting to determine
- ✓Pointer forward scans ahead until a pattern match is found
- ✓When the next lexeme is determined, the following steps are taken:
  - ✓forward is set to the character at its right end
  - ✓Record the lexeme as the attribute of the token
  - ✓lexemeBegin is set to the character immediately after the lexeme just found

Input Buffering

- ✓Sentinals
  - ✓Special characters that cannot be part of the source program
  - ✓An eof character can be used to denote the end of the buffer.
- ✓Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer
- ✓An eof character at the middle of a buffer marks the end of the input

Input Buffering

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

Figure 3.5: Lookahead code with sentinels

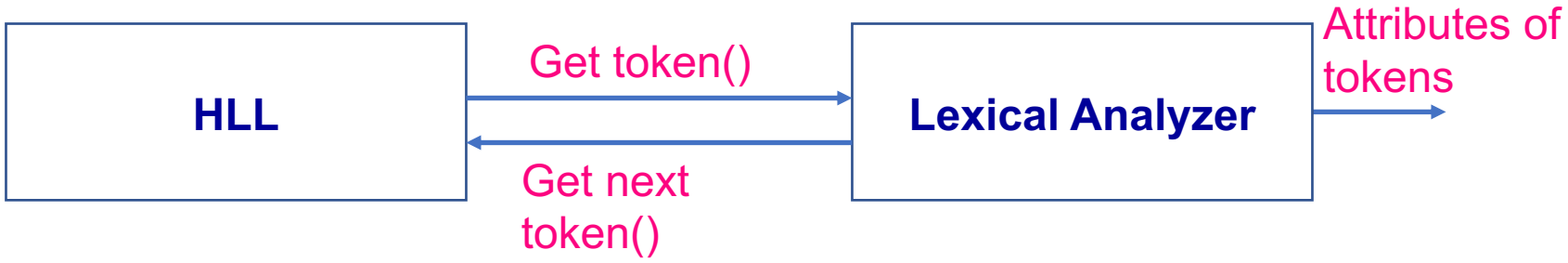
Lexical Errors

- ✓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
- ✓Recovery strategy is "panic mode" recovery
- ✓Delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left

Lexical Errors

- ✓Possible error-recovery actions are:
  - ✓Delete one character from the remaining input.
  - ✓Insert a missing character into the remaining input.
  - ✓Replace a character by another character.
  - ✓Transpose two adjacent characters
- ✓Aim: a prefix of the remaining input can be transformed into a valid lexeme

Introduction




```
#include <stdio.h>
void main()
{
int number;
printf("Enter an integer: ");
// reads and stores input
 scanf("%d", &number);
// displays output
Printf("You entered: %d", number);
}
```

Lexical Errors – Panic Mode Recovery

- Insert
- Remove
- Replace
- Transpose

Recap

- ✓In this Lecture….
  - ✓Lexical Analysis
    - ✓General idea
    - ✓The Role of lexical analyzer - Tokens, Patterns, Lexemes, Lexical Errors
    - ✓Input Buffering - Look Ahead & Sentinels
    - ✓Specification of Tokens - Regular Expressions, Regular definitions
    - ✓Recognition of Tokens - Transition Diagrams
    - ✓The Lexical-Analyzer Generator Lex

Overview

- ✓In this Lecture….
  - ✓Lexical Analysis
    - ✓General idea
    - ✓The Role of lexical analyzer - Tokens, Patterns, Lexemes, Lexical Errors
    - ✓Input Buffering - Look Ahead & Sentinels
    - ✓Specification of Tokens - Regular Expressions, Regular definitions
    - ✓Recognition of Tokens - Transition Diagrams
    - ✓The Lexical-Analyzer Generator Lex

Specification of tokens

- ✓How to specify tokens?
  - ✓Regular Expressions
- ✓What is a RE?
  - ✓It is a sequence of characters that specify a search pattern in input

Strings & Languages

- ✓Alphabet
  - ✓ Set of finites symbols, represented by ∑
  - ✓ Eg binary alphabet = { 0, 1}, ASCII = { all ascii characters }

- ✓String
  - ✓ Finite set of symbols drawn from an alphabet
  - ✓ Called as a **sentence** or **word**
  - ✓ |s| denotes the length of string s. No of symbols present.
  - ✓ ϵ denotes empty string. Length is 0.

Strings & Languages

- ✓Languages
  - ✓ Any countable set of strings over an alphabet
  - ✓ Φ denotes empty set ie. {ϵ}
  - ✓ All ‘C’ programs, set of all English sentences

Parts of Strings

- ✓Length of a string: |s|
- ✓Empty string: ϵ
- ✓Prefix of a string: Leading symbols of the string
- ✓Proper prefix of a string: Not considering the empty & Complete String

Parts of Strings

- ✓Suffix of a string: Trailing symbols of the string
- ✓Proper suffix of a string: Not considering the empty & Complete String
- ✓Substring: Remove the prefix and suffix
- ✓Proper substring

Operations on Strings &Languages

- ✓Operations on Strings
  - ✓**Concatenation** of strings x, y denoted by xy = appending y to x
    - ✓$\epsilon s = s\epsilon = s$
  - ✓**Exponentiation**
    - ✓Define $S^0 = \epsilon$, for all $i > 0$, $s^i = s^{i-1}s$
    - ✓$s^1 = s$, $(s^1 = s^0 s = s)$, $s^2 = ss$, $s^3 = sss$

- ✓Operations on Languages
  - ✓Union, Concatenation, Closure : all similar to set operations

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Operations on Strings & Languages

- ✓Let L = { A,B,..,Z,a,b,…z}, D = {0,1,…,9},
- ✓Eg of set of languages constructed from L and D

1. $L \cup D$ is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3. L^4 is the set of all 4-letter strings.
4. L^* is the set of all strings of letters, including ϵ, the empty string.
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

Regular Expression

- ✓Formal way to specify the description of patterns
  - ✓ Eg . We could say that the ‘C’ language identifier is a Letter followed by any number (0 or more) of letter or digits
  - ✓ Using Reg Expr we say : L(LUD)*

Regular Expression

- ✓ What are Reg Expr
  - ✓ Built from alphabets (symbols)
  - ✓ Operators: Union, Concatenation and Closure
  - ✓ Reg exrp: r -> L(r) : Language
  - ✓ REs are built recursively from smaller REs.

Regular Expression

- ✓How are regular expressions derived
  - ✓Basis Rules
    - ✓ϵ is a RE and L(ϵ) = {ϵ}, means a lang whose member is an empty string.
    - ✓If a is a symbol in ∑, then **a** is a RE and L(**a**) = {a}
  - ✓Induction:
    - ✓let r & s be REs and L(r) & L(s) be languages respectively
    - ✓(r)|(s) is a RE denoting the language L(r) U L(s)
    - ✓(r)(s) is a RE denoting the language L(r)L(s)
    - ✓$(r)^*$ is a RE denoting $(L(r))^*$
  - ✓Precedence of operators
    - ✓* (left associative, highest)
    - ✓Concatenation (left associative)
    - ✓| (left associative, lowest)

Example

Example 3.4: Let $\Sigma = \{a, b\}$.

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

Overview

- ✓In prevoius Lecture….
  - ✓Lexical Analysis
    - ✓General idea
    - ✓The Role of lexical analyzer - Tokens, Patterns, Lexemes, Lexical Errors
    - ✓Input Buffering - Look Ahead & Sentinels
    - ✓Specification of Tokens - Regular Expressions, Regular definitions
    - ✓Recognition of Tokens - Transition Diagrams
    - ✓The Lexical-Analyzer Generator Lex

Overview

- ✓In this Lecture….
  - ✓Lexical Analysis
    - ✓General idea
    - ✓The Role of lexical analyzer - Tokens, Patterns, Lexemes, Lexical Errors
    - ✓Input Buffering - Look Ahead & Sentinels
    - ✓Specification of Tokens - Regular Expressions, Regular definitions
    - ✓Recognition of Tokens - Transition Diagrams
    - ✓The Lexical-Analyzer Generator Lex

Regular definition

- ✓For notational convenience, give names to certain regular expressions and use those names in subsequent expressions

- ✓**Example 3.5** : **C** identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

Regular definition

- For notational convenience, give names to certain regular expressions and use those names in subsequent expressions

- **Example 3.5** : **C** identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$
\begin{aligned}
\mathit{letter\_} &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
\mathit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{id} &\rightarrow \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^*
\end{aligned}
$$

Regular definition

Example 3.6: Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

$$
\begin{aligned}
\textit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\textit{digits} &\rightarrow \textit{digit}\ \textit{digit}^{*} \\
\textit{optionalFraction} &\rightarrow \texttt{.}\ \textit{digits} \mid \epsilon \\
\textit{optionalExponent} &\rightarrow (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ \textit{digits}\ ) \mid \epsilon \\
\textit{number} &\rightarrow \textit{digits}\ \textit{optionalFraction}\ \textit{optionalExponent}
\end{aligned}
$$

Extensions to Regular Expression

- ✓One or more instances (+)
  - ✓$(r)^+$ denotes $(L(r))^+$
  - ✓Same precedence and associativity as *
  - ✓$r^* = r^+ \mid \epsilon$ or $r^+ = r\, r^*$
- ✓Zero or 1 instance (?)
  - ✓$r? = r \mid \epsilon$
- ✓Character Class
  - ✓RE $a_1|a_2|\ldots|a_n$ can be replaced by $[a_1a_2\ldots a_n]$
  - ✓If these are consecutive $[a_1\text{-}a_n]$

Regular definition

$$
\begin{aligned}
\textit{letter\_} &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
\textit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\textit{id} &\rightarrow \textit{letter\_} \; (\; \textit{letter\_} \mid \textit{digit} \;)^*
\end{aligned}
$$

Example 3.7: Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$
\begin{aligned}
\textit{letter\_} &\rightarrow [\texttt{A-Za-z\_}] \\
\textit{digit} &\rightarrow [\texttt{0-9}] \\
\textit{id} &\rightarrow \textit{letter\_} \; (\; \textit{letter} \mid \textit{digit} \;)^*
\end{aligned}
$$

Regular definition

Example 3.6: Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

$$
\begin{aligned}
\textit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\textit{digits} &\rightarrow \textit{digit}\ \textit{digit}^{*} \\
\textit{optionalFraction} &\rightarrow \texttt{.}\ \textit{digits} \mid \epsilon \\
\textit{optionalExponent} &\rightarrow (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ \textit{digits}\ ) \mid \epsilon \\
\textit{number} &\rightarrow \textit{digits}\ \textit{optionalFraction}\ \textit{optionalExponent}
\end{aligned}
$$

The regular definition of Example 3.6 can also be simplified:

$$
\begin{aligned}
\textit{digit} &\rightarrow [\texttt{0-9}] \\
\textit{digits} &\rightarrow \textit{digit}^{+} \\
\textit{number} &\rightarrow \textit{digits}\ (\texttt{.}\ \textit{digits})?\ (\ \texttt{E}\ [\texttt{+-}]?\ \textit{digits}\ )?
\end{aligned}
$$

Recognition of Tokens

- ✓Expressed patterns using regular expressions
- ✓Need to understand:
  - ✓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

Recognition of Tokens

✓Eg. if then else statement

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

Eg:if-then-else (Patterns)

✓Pattern for tokens of branching using regular definition (if then else statement)

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

Eg:if-then-else (Patterns)

✓Whitespaces with ws token

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

ws → (**blank** | **tab** | **newline**)$^+$

Eg:if-then-else (Tokens, Attribute Vaules)

- ✓Tokens and Attribute values returned by the lexical analyser, when lexemes recognised by patterns

LEXEMES	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	GE

Figure 3.12: Tokens, their patterns, and attribute values

Transition diagrams

- ✓Useful intermediate step in the construction of lexical analyzer
- ✓Convert patterns into stylized flowcharts, called “transition diagrams.”
- ✓Why?
  - ✓Reg Exprs can be translated into transition diagrams
  - ✓Transition diagrams can be translated into program code for lexical analyzer
- ✓The above steps can be done
  - ✓by hand
  - ✓by a program (called lex)

Transition diagrams

- ✓Transition Diagrams are Finite State Machines
- ✓Nodes : States
  - ✓Memory.
  - ✓Represents how much of inputs have been scanned.
- ✓Transition: Edges
  - ✓Based on the input symbol mentioned along the edges the state changes occur.
- ✓Deterministic FSA
  - ✓No more than one edge from a given state on a given symbol.

Transition diagrams

- ✓Double circle: accepting state (return of a token)
- ✓* indicates a retract is required (retract()).

Transition diagrams

- ✓Double circle: accepting state (return of a token)
- ✓* indicates a retract is required (retract()).

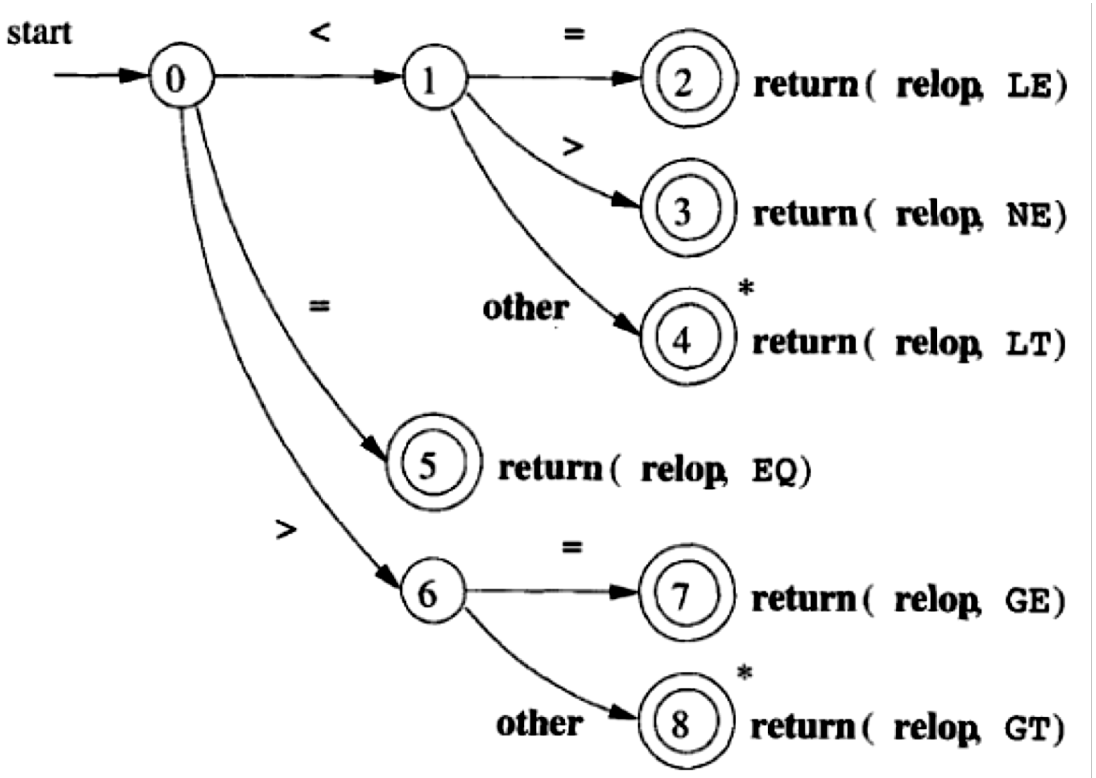


Figure 3.13: Transition diagram for **relop**

Recognition of reserved words and identifiers

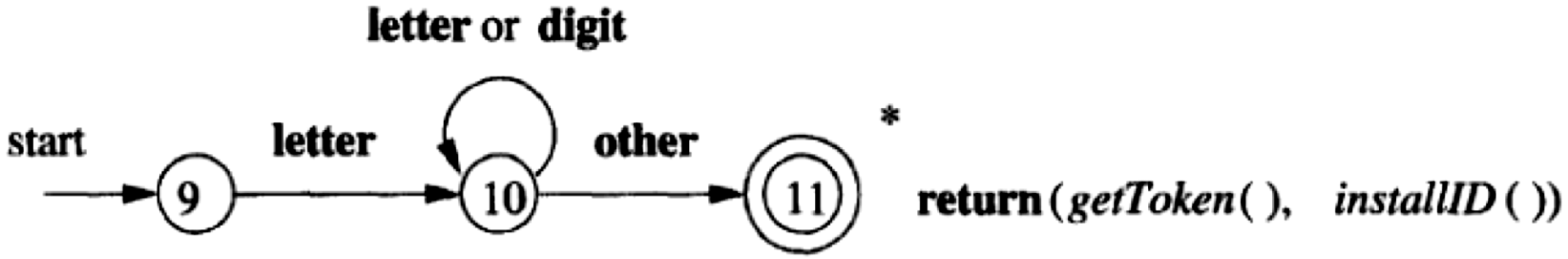


Figure 3.14: A transition diagram for id's and keywords

Recognition of Numbers

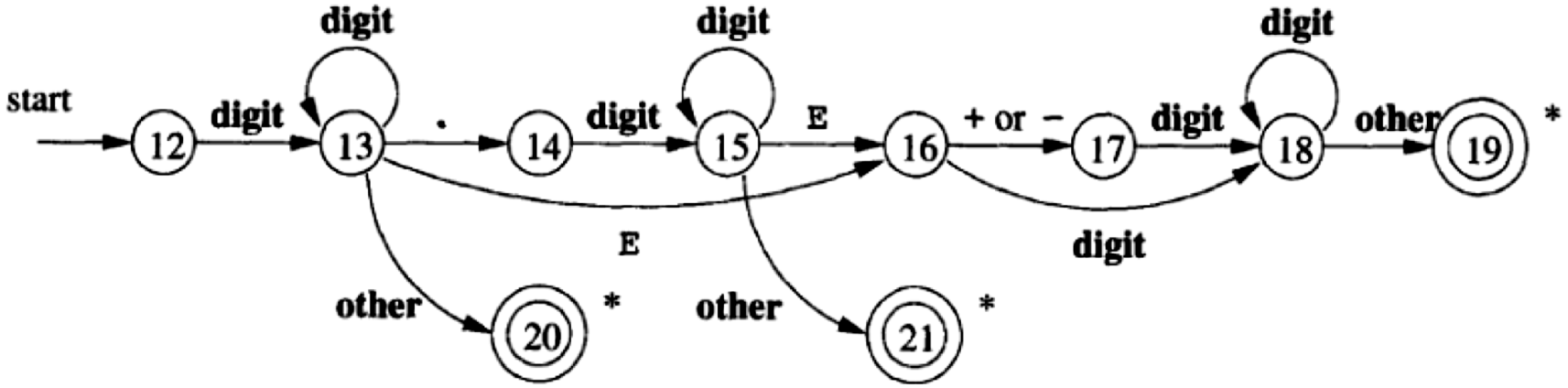


Figure 3.16: A transition diagram for unsigned numbers

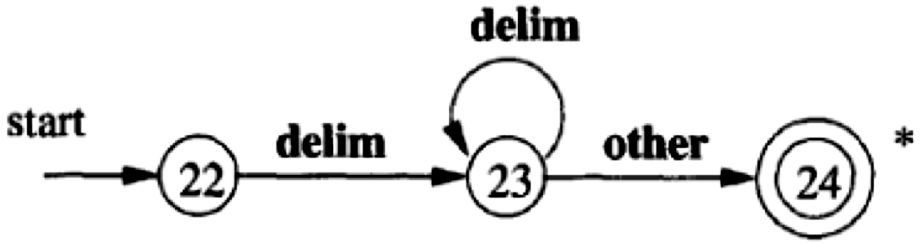


Figure 3.17: A transition diagram for whitespace

Hand Impl. of Transition Diag (relop)

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

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

Lexical Analyzer Generator: Lex

- ✓Lex (more recent Flex) - specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.

- ✓The input notation for the Lex tool is referred to as the *Lex language*

  - ✓and the tool itself is the *Lex compiler.*

- ✓Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex . yy . c

Uses of Lex

- ✓$lex lex.l
- ✓$cc lex.yy.c -ll
- ✓$a.out <prog.txt

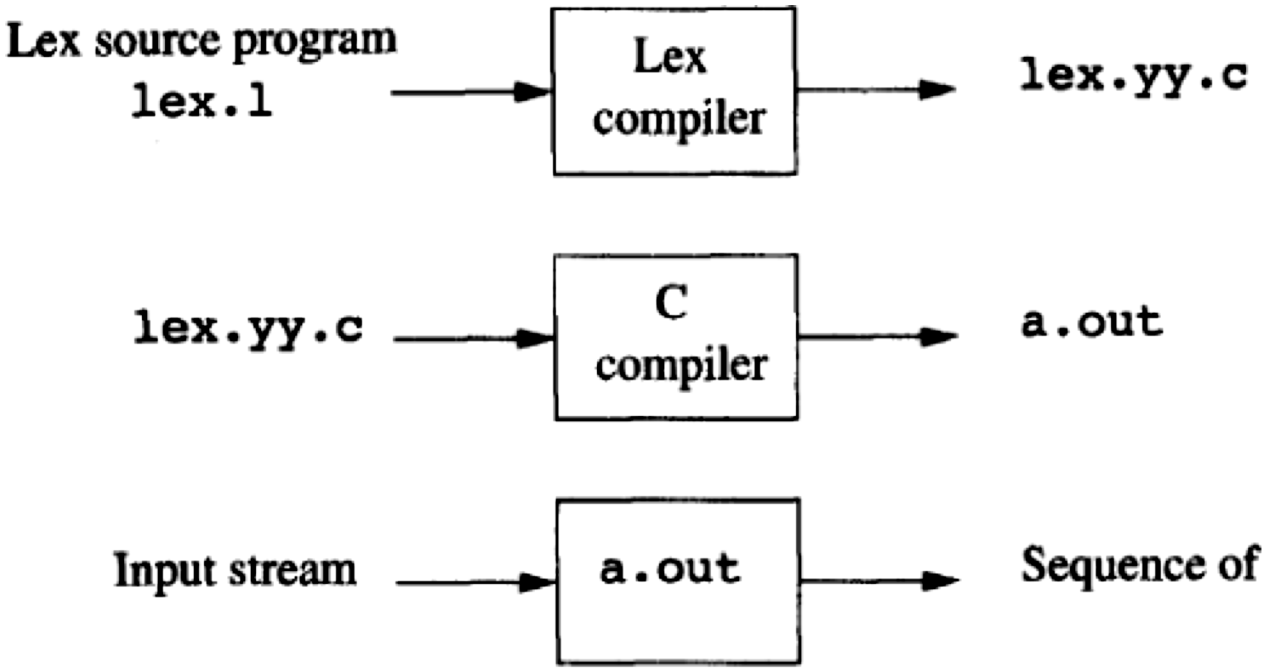


Figure 3.22: Creating a lexical analyzer with Lex

- ✓$flex lex.l
- ✓$cc lex.yy.c -lfl
- ✓$a.out <prog.txt

Structure of Lex programs

- ✓Structure

```
declarations
%%
translation rules
%%
auxiliary functions
```

- ✓Declaration:
  - ✓C code defined between

    %{ and %} - Ex: %{ #include <stdio.h>%}
  - ✓variables, *manifest constants* and regular definitions

    digit [0-9]

    letter [A-Z a-z]

Structure of Lex programs

- ✓Translation:
  - ✓Each pattern is a regular expression, which may use the regular definitions of the declaration section.

Pattern { Action }

Ex: if { printf(“Keyword ‘if’\n”);}

- ✓Auxiliary functions: additional functions – routines which are called in the action parts

Sample Lex Program

✓$lex sample.l

```
%{
        #include<stdio.h>
%}
digit    [0-9]
letter   [A-Z a-z]
ident    {letter}({letter}|{digit})*
%%
if       {printf("keyword 'if' ");}
ident    {printf("Identifier");}
%%
```

Conflict resolution in Lex programs

- ✓Conflict
  - ✓when several prefixes of the input match one or more patterns
    - ✓Always prefer a longer prefix to a shorter prefix.
    - ✓If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

Conflict resolution in Lex programs

- Conflict
  - when several prefixes of the input match one or more patterns
    - Always prefer a longer prefix to a shorter prefix.
    - If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

```
%%
a     { printf ("1A"); }
aa    { printf ("2A"); }
```

Input : aaa

Output:

```
%%
l(l|d)*     { printf ("ID"); }
if          { printf ("IF"); }
```

Input : if

Output:

Conflict resolution in Lex programs

- Conflict
  - when several prefixes of the input match one or more patterns
    - Always prefer a longer prefix to a shorter prefix.
    - If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

```
%%
a     { printf ("1A"); }
aa    { printf ("2A"); }
```

Input : aaa
Output: 2A1A

```
%%
l(l|d)*     { printf ("ID"); }
if          { printf ("IF"); }
```

Input : if
Output: ID

Lex regular expressions

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	`a`
$\backslash c$	character c literally	`\*`
`"`s`"`	string s literally	`"**"`
`.`	any character but newline	`a.*b`
`^`	beginning of a line	`^abc`
`$`	end of a line	`abc$`
$[s]$	any one of the characters in string s	`[abc]`
$[\hat{}\,s]$	any one character not in string s	`[^abc]`
$r*$	zero or more strings matching r	`a*`
$r+$	one or more strings matching r	`a+`
$r?$	zero or one r	`a?`
$r\{m,n\}$	between m and n occurrences of r	`a[1,5]`
r_1r_2	an r_1 followed by an r_2	`ab`
$r_1 \mid r_2$	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

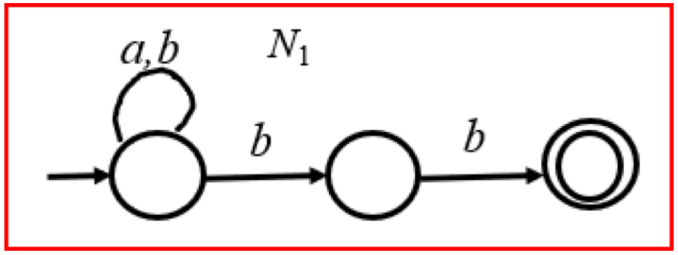
Figure 3.8: Lex regular expressions

Regular Expression & State Transition -1

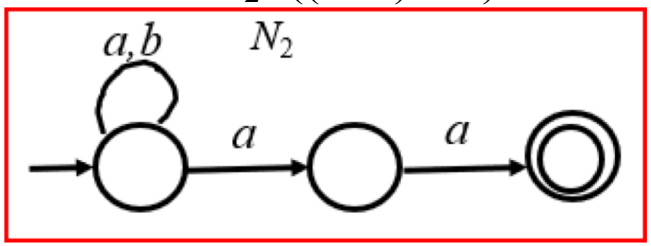
Eg. Strings starts with (a|b) which end in aa or bb

$L(M) = \{ w \in \Sigma^* \mid w = saa$ or $w = sbb$ for some string $s \in \Sigma^* \}$.

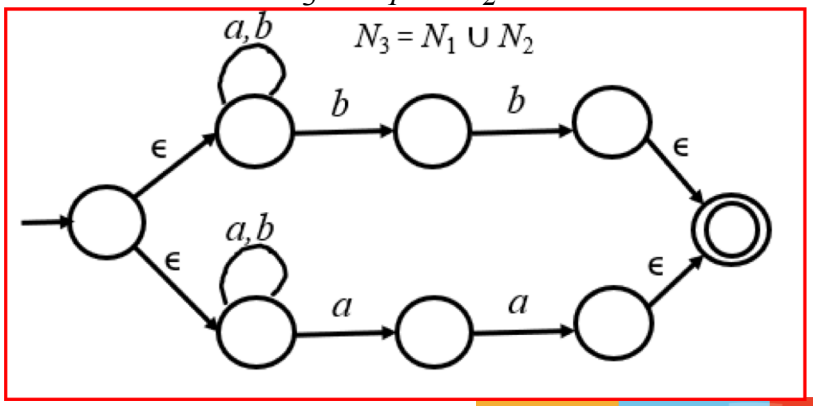
$R_1=((a+b)^*aa)$



$R_2=((a+b)^*bb)$



$R_3= R_1 + R_2$



$R_1=((a+b)^*aa)$; $R_2=((a+b)^*bb)$

$R_3=((a+b)^*aa) + ((a+b)^*bb)$

$= (a+b)^*(aa + bb)$

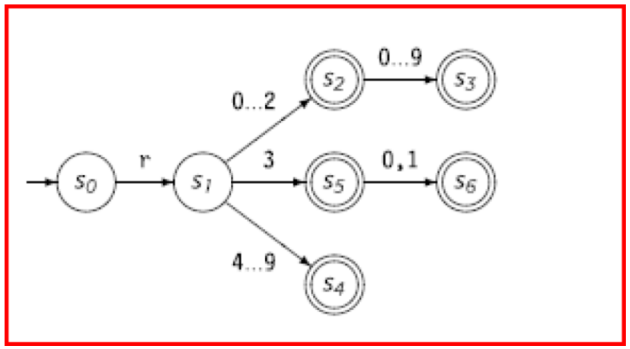
Regular Expression & State Transition -2

Regular expression for valid registers r0 to r31 (..r255)

Regular Expression & State Transition -2

Regular expression for valid registers r0 to r31 (..r255)

r0 | r00 | r1 | r01 | r2 | r02 | r3 | r03 | r4 | r04 | r5 | r05 | r6 | r06 | r7 |
r07 | r8 | r08 | r9 | r09 | r10 | r11 | r12 | r13 | r14 | r15 | r16 | r17 | r18 |
r19 | r20 | r21 | r22 | r23 | r24 | r25 | r26 | r27 | r28 | r29 | r30 | r31



$r([0..2]([0..9]|\epsilon)|[4..9]|(3(0|1|\epsilon)))$

r([0-2][0-9]?|3[01]?|[4-9])

Regular Expression & State Transition -2

Valid IPv4 address :

Correct :	Incorrect:
0.0.0.0	000.0.0.0
123.56.20.200	123.056.20.200
192.168.199.99	292.268.299.197
250.255.255.255	300.259.023.999

Lex Variables & Functions

Name	Function
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string yylval value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
ECHO	write matched string

Lex Practice -1

Write a Lex program to match everything except newline and match newline

Lex Practice -1

Write a Lex program to match everything except newline and match newline

```
%%
/* match everything except newline */
.       ECHO;
/* match newline */
\n      ECHO;
%%
```

Lex Practice -2

- ✓Write a lex program that replaces multiple spaces and tab to a single space. If there are multiple spaces at the beginning or end of a line, it should all be ignored.

- ✓eg. a.txt

123 hello

␣␣␣Hello␣␣␣world␣␣␣␣

123␣hello

hello␣world

Lex Practice -2

✓Write a lex program that replaces multiple spaces and tab to a single space. If there are multiple spaces at the beginning or end of a line, it should all be ignored.

✓eg. a.txt

123 hello

␣␣␣Hello␣␣␣world␣␣␣␣

123␣hello

hello␣world

```
%%
^[ \t]+   { }
[ \t]+    {printf("%c", ' '); }
[ \t]+$   { /* ignore space */}
```

Lex Practice -3

✓What is the output of the following lex program a.l

```
%%
cot { printf("portable bed"); }
cat { printf("thankless pet"); }
cats { printf("anti-herd"); }
%%

$lex a.l; cc lex.yy.c –ll

$echo “the cat on the cot joined the cats" | ./a.out
```

Lex Practice-3

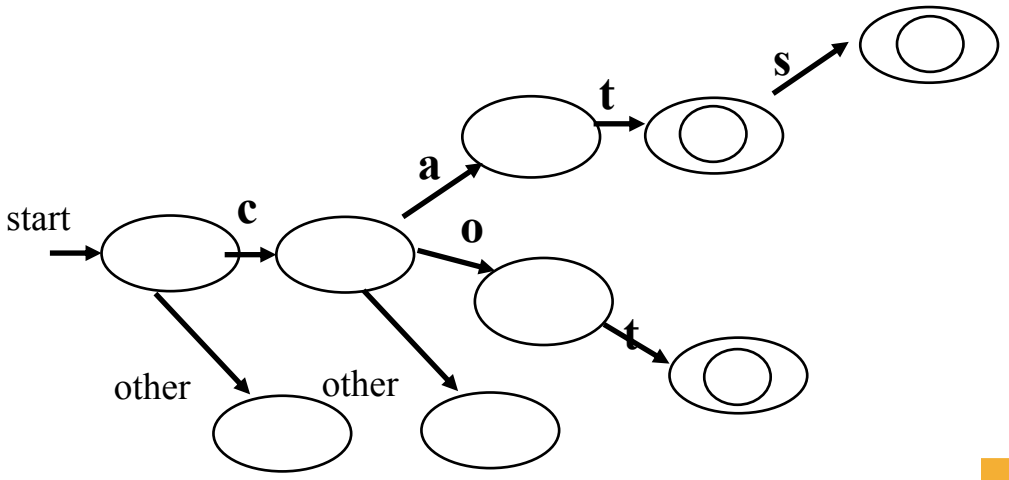
What is the output of the following lex program a.l

```
%%
cot { printf("portable bed"); }
cat { printf("thankless pet"); }
cats { printf("anti-herd"); }

$lex a.l; cc lex.yy.c –ll

$echo “the cat on the cot joined the cats" | ./a.out
```

```
the thankless pet on the
portable bed joined the anti-
herd
```



Lex Practice - 4

What is the output of the following lex program a.l

```
%%
c.t { printf(“aaa bbb"); }
cot { printf("portable bed"); }
cat { printf("thankless pet"); }
cats { printf("anti-herd"); }
%%
```

Lex Practice - 4

What is the output of the following lex program a.l

```
%%
c.t { printf(“aaa bbb"); }
cot { printf("portable bed"); }
cat { printf("thankless pet"); }
cats { printf("anti-herd"); }

$lex a.l
a.l:3: warning, rule cannot be matched
a.l:4: warning, rule cannot be matched

$cc lex.yy.c –ll
$echo “the cat on the cot joined the cats" | ./a.out

the aaa bbb on the aaa bbb joined the anti-herd
```

How Lex works

- ✓What does lex do?
  - ✓Input: Reg Exprs in a <file.l>
  - ✓Output: ‘C’ program (lex.yy.c)

- ✓Heart of Lex is Finite Automata
- ✓2 Type of FSA
  - ✓Non deterministic FSA
  - ✓Deterministic FSA (lex.yy.c)

- ✓Direct Reg Expr → DFA is difficult
- ✓So. Reg Expr → NFA → DFA

Lex program for the tokens

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
```

Lex program for the tokens

```
{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%
```

References

- ✓Material covers text book sec. 3.1-3.5.4

Lex program for the tokens

```
int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Summary

- ✓Lexical Analysis
  - ✓The Role of lexical analyzer - Tokens, Patterns, Lexemes, Lexical Errors
  - ✓Input Buffering - Look Ahead & Sentinels
  - ✓Specification of Tokens - Regular Expressions, Regular definitions
  - ✓Recognition of Tokens - Transition Diagrams
  - ✓The Lexical-Analyzer Generator Lex

Thank you