Lexical Analysis

Outline of the Lecture

- What is lexical analysis?
- Why should LA be separated from syntax analysis?
- Tokens, patterns, and lexemes
- Difficulties in lexical analysis
- Recognition of tokens - finite automata and transition diagrams
- Specification of tokens - regular expressions and regular definitions
- LEX - A Lexical Analyzer Generator

What is Lexical Analysis?

- The input is a high level language program, such as a ’C’ program in the form of a sequence of characters
- The output is a sequence of *tokens* that is sent to the parser for syntax analysis
- Strips off blanks, tabs, newlines, and comments from the source program
- Keeps track of line numbers and associates error messages from various parts of a compiler with line numbers

Separation of Lexical Analysis from Syntax Analysis

- Simplification of design - software engineering reason
- I/O issues are limited LA alone
- More compact and faster parser
  - Comments, blanks, etc., need not be handled by the parser
  - A parser is more complicated than a lexical analyzer and shrinking the grammar makes the parser faster
    - No rules for numbers, names, comments, etc., are needed in the parser
- LA based on finite automata are more efficient to implement than pushdown automata used for parsing (due to stack)

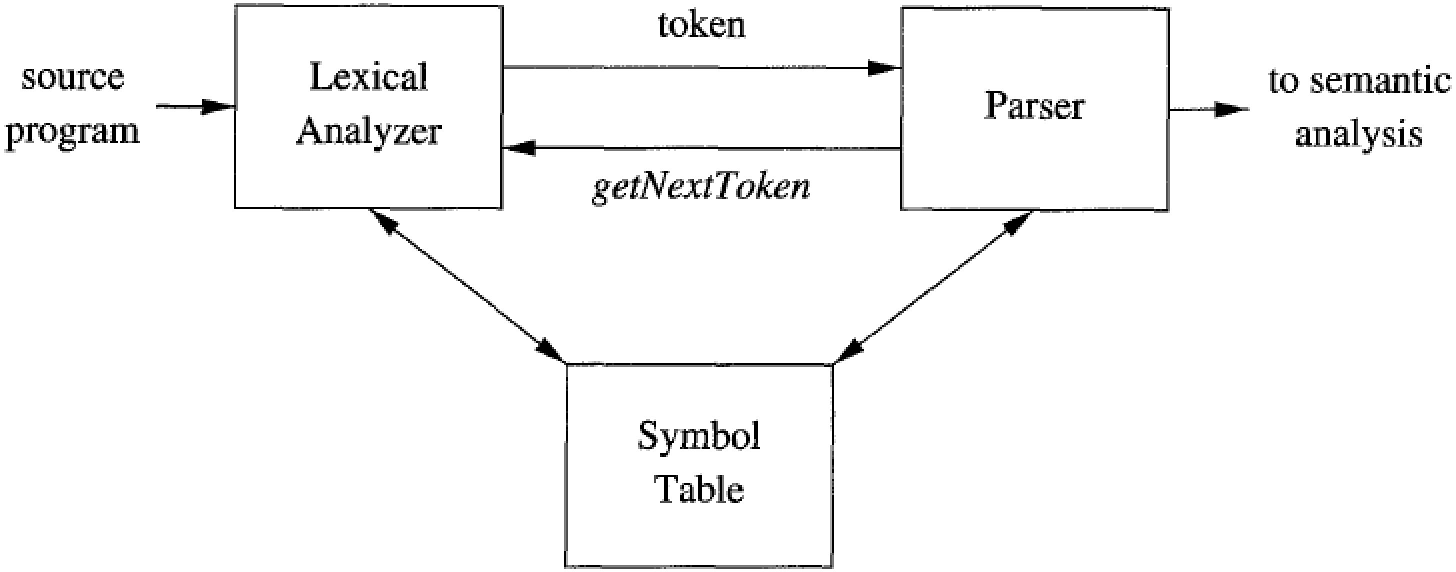

Figure 3.1: Interactions between the lexical analyzer and the parser

- When parser needs the next token it asks the LA to fetch the next token.

3.1.2 Tokens, Patterns, and Lexemes

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

- A *token* is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.

- 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 and is identified by the lexical analyzer as an instance of that 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 letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Figure 3.2: Examples of tokens

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

both `printf` and `score` are lexemes matching the pattern for token **id**,

`"Total = %d\n"` is a lexeme matching **literal**.

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.

2. Tokens for the operators, either individually or in classes such as the token `comparison` mentioned in Fig. 3.2.

3. One token representing all identifiers.

4. One or more tokens representing constants, such as numbers and literal strings.

5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

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

Figure 3.2: Examples of tokens

What symbol table holds

- Info. about a token
- for identifier: its lexeme, its type, and the location (line number in source file) where it is first found. {err. message might require line no.}

Difficulties in Lexical Analysis

- Certain languages do not have any reserved words, *e.g.*, **while, do, if, else**, etc., are reserved in 'C', but not in PL/1
- In FORTRAN, some keywords are context-dependent
  - In the statement, *DO 10 I = 10.86*, **DO10I** is an identifier, and **DO** is not a keyword
  - But in the statement, *DO 10 I = 10, 86*, **DO** is a keyword
  - Such features require substantial *look ahead* for resolution
- Blanks are not significant in FORTRAN and can appear in the midst of identifiers, but not so in 'C'
- LA cannot catch any significant errors except for simple errors such as, illegal symbols, etc.

Difficulties in LA

PL/I keywords are not reserved

IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN

- C++ template→ template <class T>
- C++ stream syntax → cin >> var;

- Nested template →
  template <template <class T>>
- Only way out is
  template <template <class T> > /*insert a blank*/

Example 3.2: The token names and associated attribute values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs.

<**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

For the code fragment below,
choose the correct number of tokens in
each class that appear in the code fragment

```
x = 0;\n\twhile (x < 10) {\n\tx++;\n}
```

- ○ W = 9; K = 1; I = 3; N = 2; O = 9
- ○ W = 11; K = 4; I = 0; N = 2; O = 9
- ○ W = 9; K = 4; I = 0; N = 3; O = 9
- ○ W = 11; K = 1; I = 3; N = 3; O = 9

W: Whitespace
K: Keyword
I: Identifier
N: Number
O: Other Tokens:
{ } ( ) < ++ ; =

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

a lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an undeclared function identifier.

- what if no pattern (regular expression) matches with the next sequence of characters?
- Can LA correct errors?
  - yes. Some errors can be corrected. In LaTeX you might have heard “inserted missing $”

- Panic mode recovery: skip till valid lexeme is found.

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.

3.2 Input Buffering

- an implementation issue.
- read the book **Reading Assignment**

Specification and Recognition of Tokens

- Regular definitions, a mechansm based on *regular expressions* are very popular for specification of tokens
  - Has been implemented in the lexical analyzer generator tool, LEX
  - We study regular expressions first, and then, token specification using LEX
- Transition diagrams, a variant of finite state automata, are used to implement regular definitions and to recognize tokens
  - Transition diagrams are usually used to model LA before translating them to programs by hand
  - LEX automatically generates optimized FSA from regular definitions

Languages

- **Symbol**: An abstract entity, not defined
  - Examples: letters and digits
- **String**: A finite sequence of juxtaposed symbols
  - **abcb, caba** are strings over the symbols *a,b,* and *c*
  - $|w|$ is the length of the string *w*, and is the #symbols in it
  - ϵ is the empty string and is of length 0
- **Alphabet**: A *finite* set of symbols
- **Language**: A set of strings of symbols from some alphabet
  - Φ and $\{\epsilon\}$ are languages
  - The set of palindromes over {0,1} is an infinite language
  - The set of strings, {01, 10, 111} over {0,1} is a finite language
- If Σ is an alphabet, Σ^* is the set of all strings over Σ

Terms for Parts of Strings

The following string-related terms are commonly used:

1. A *prefix* of string s is any string obtained by removing zero or more symbols from the end of s. For example, **ban**, **banana**, and ϵ are prefixes of **banana**.

2. A *suffix* of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, **nana**, **banana**, and ϵ are suffixes of **banana**.

3. A *substring* of s is obtained by deleting any prefix and any suffix from s. For instance, **banana**, **nan**, and ϵ are substrings of **banana**.

4. The *proper* prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not ϵ or not equal to s itself.

5. A *subsequence* of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, **baan** is a subsequence of **banana**.

Language Representations

- Each subset of Σ^* is a language
- This set of languages over Σ^* is uncountably infinite
- Each language must have by a finite representation
  - A finite representation can be encoded by a finite string
  - Thus, each string of Σ^* can be thought of as representing some language over the alphabet Σ
  - Σ^* is countably infinite
  - Hence, there are more languages than language representations
- **Regular expressions** (type-3 or regular languages), **context-free grammars** (type-2 or context-free languages), **context-sensitive grammars** (type-1 or context-sensitive languages), and **type-0 grammars** are finite representations of respective languages
- RL $<<$ CFL $<<$ CSL $<<$ type-0 languages

Examples of Languages

Let $\Sigma = \{a, b, c\}$

- $L_1 = \{a^m b^n | m, n \geq 0\}$ is regular
- $L_2 = \{a^n b^n | n \geq 0\}$ is context-free but not regular
- $L_3 = \{a^n b^n c^n | n \geq 0\}$ is context-sensitive but neither regular nor context-free
- Showing a language that is type-0, but none of CSL, CFL, or RL is very intricate and is omitted

Automata

- Automata are machines that accept languages
  - Finite State Automata accept RLs (corresponding to REs)
  - Pushdown Automata accept CFLs (corresponding to CFGs)
  - Linear Bounded Automata accept CSLs (corresponding to CSGs)
  - Turing Machines accept type-0 languages (corresponding to type-0 grammars)
- Applications of Automata
  - Switching circuit design
  - Lexical analyzer in a compiler
  - String processing (*grep*, *awk*), etc.
  - State charts used in object-oriented design
  - Modelling control applications, e.g., elevator operation
  - Parsers of all types
  - Compilers

Finite State Automaton

- An FSA is an **acceptor** or **recognizer** of regular languages
- An FSA is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, where
  - Q is a finite set of states
  - Σ is the input alphabet
  - δ is the transition function, $\delta : Q \times \Sigma \to Q$
    That is, $\delta(q, a)$ is a state for each state q and input symbol a
  - q_0 is the start state
  - F is the set of *final* or *accepting* states
- In one move from some state q, an FSA reads an input symbol, changes the state based on δ, and gets ready to read the next input symbol
- An FSA **accepts** its input string, if starting from q_0, it consumes the entire input string, and reaches a final state
- If the last state reached is not a final state, then the input string is rejected

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

Figure 3.6: Definitions of operations on languages

3.3.3 Regular Expressions

BASIS: There are two rules that form the basis:

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.

2. If a is a symbol in Σ, then $\mathbf{a}$ is a regular expression, and $L(\mathbf{a}) = \{a\}$, that is, the language with one string, of length one, with a in its one position. Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.[1]

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting languages $L(r)$ and $L(s)$, respectively.

1. $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.

2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.

3. $(r)^*$ is a regular expression denoting $\left(L(r)\right)^*$.

4. (r) is a regular expression denoting $L(r)$. This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

Certain pairs of parentheses can be removed with the following precedence rules.

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.

for example, we may replace the regular expression $(\mathbf{a})|((\mathbf{b})^*(\mathbf{c}))$ by $\mathbf{a}|\mathbf{b}^*\mathbf{c}$.

Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.

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

1. The regular expression $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.

2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ 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 $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.

3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \ldots\}$.

4. $(\mathbf{a}|\mathbf{b})^*$ 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 $(\mathbf{a}^*\mathbf{b}^*)^*$.

5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ 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.

LAW	DESCRIPTION
$r \mid s = s \mid r$	$\mid$ is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s \mid t) = rs \mid rt;\ (s \mid t)r = sr \mid tr$	Concatenation distributes over $\mid$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r \mid \epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Figure 3.7: Algebraic laws for regular expressions

- a|b is same as b|a

3.3.4 Regular Definitions

$$\begin{array}{ccc} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \cdots & \\ d_n & \rightarrow & r_n \end{array}$$

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

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.

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

□

3.3.5 Extensions of Regular Expressions

1. *One or more instances.* The unary, postfix operator $^+$ represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then $(r)^+$ denotes the language $\big(L(r)\big)^+$. The operator $^+$ has the same precedence and associativity as the operator $*$. Two useful algebraic laws, $r^* = r^+|\epsilon$ and $r^+ = rr^* = r^*r$ relate the Kleene closure and positive closure.

$$
\begin{array}{lcl}
digit & \to & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
digits & \to & digit^+
\end{array}
$$

2. *Zero or one instance.* The unary postfix operator ? means "zero or one occurrence." That is, $r?$ is equivalent to $r|\epsilon$, or put another way, $L(r?) = L(r) \cup \{\epsilon\}$. The ? operator has the same precedence and associativity as $*$ and $+$.

$$\begin{array}{rcl} digits & \rightarrow & digit^{+} \\ number & \rightarrow & digits\ (.\ digits)?\ (\ \mathtt{E}\ (+\mid -)\ ?\ digits\ )? \end{array}$$

3. *Character classes.* A regular expression $a_1|a_2|\cdots|a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2\cdots a_n]$. More importantly, when $a_1, a_2, \ldots, a_n$ form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a_1-a_n, that is, just the first and last separated by a hyphen. Thus, **[abc]** is shorthand for **a|b|c**, and **[a-z]** is shorthand for **a|b|** $\cdots$ **|z**.

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

. in lex means something else. But, here it matches with character .

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

! Exercise 3.3.2: Describe the languages denoted by the following regular expressions:

a) $\mathbf{a}(\mathbf{a}|\mathbf{b})^*\mathbf{a}$.

b) $((\epsilon|\mathbf{a})\mathbf{b}^*)^*$.

c) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$.

d) $\mathbf{a}^*\mathbf{b}\mathbf{a}^*\mathbf{b}\mathbf{a}^*\mathbf{b}\mathbf{a}^*$.

- Give example strings in each.
- Can you give NFA/DFA for each?

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**)+

White space (ws) can be seen as a token in itself. As a rule, it is discarded without giving it to the syntax analysis.
Here, actually **blank, tab, newline** needs to be given regular definitions.

Reserved words/keywords are preloaded into the symbol table (One way to overcome the problem of saying ***if*** as an **identifier)**

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

Lexemes are searched in the symbol table to identify a token's presence or absence.

Exercise : Write regular definitions for the following languages:

a) All strings of lowercase letters that contain the five vowels in order.

b) All strings of lowercase letters in which the letters are in ascending lexicographic order.

c) Comments, consisting of a string surrounded by `/*` and `*/`, without an intervening `*/`, unless it is inside double-quotes (`"`).

==> d) All strings of digits with no repeated digits. *Hint*: Try this problem first with a few digits, such as $\{0, 1, 2\}$.

==> e) All strings of digits with at most one repeated digit.

==> f) All strings of a's and b's with an even number of a's and an odd number of b's.

g) The set of Chess moves, in the informal notation, such as `p-k4` or `kbp`$\times$`qn`.

==> h) All strings of a's and b's that do not contain the substring abb.

==> i) All strings of a's and b's that do not contain the subsequence abb.

- **Assignment 1**: Do problems *a,b,c,g,h* →25 marks
- Deadline 22nd Jan (submit in class)

- **LEX regular expressions are different from standard regular expressions.**
- **They have some extra power by giving context sensitive features**

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]
[^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]
$r_1 r_2$	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

3.4.1 Transition Diagrams

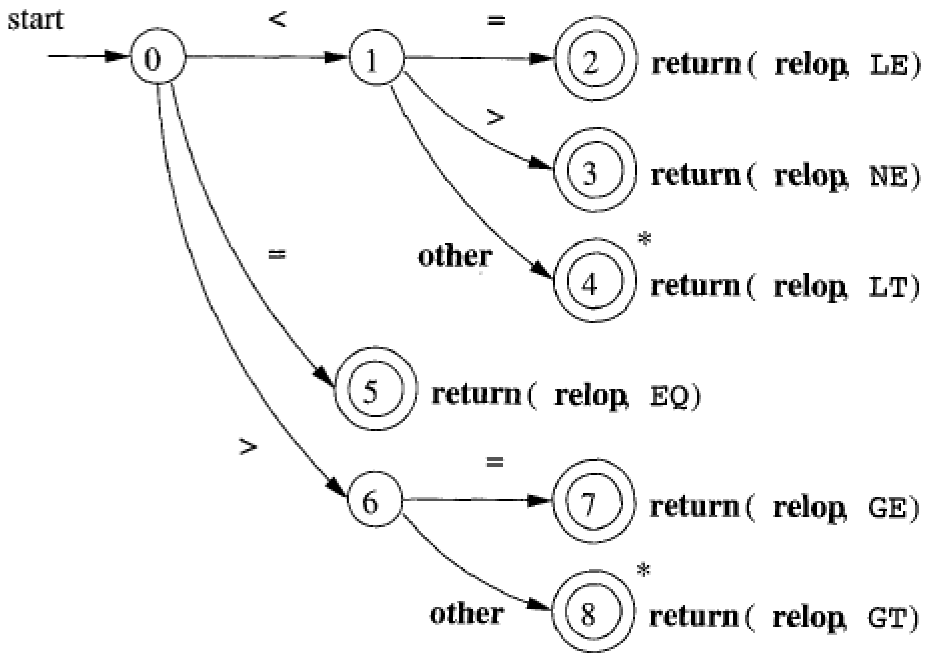

Figure 3.13: Transition diagram for **relop**

* on accepting state means retract the forward pointer by one position

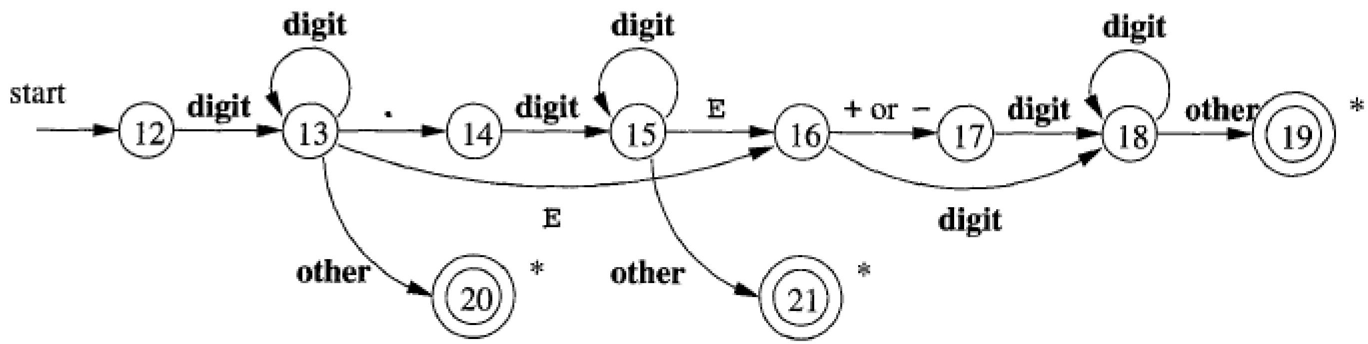

Figure 3.16: A transition diagram for unsigned numbers

Keyword problem

- Keyword/reserved word matches **identifier** token
- Two ways to overcome this
  - Preload the symbol table with all keywords (Token-name is the keyword name itself)
    - Symbol Table is searched with a lexeme, and if found the corresponding token is returned.
  - Give keywords higher priority in recognition than identifiers (a rule is embedded in to LA)

Prefix problem

- Prefix of == matches with =
- == itself matches
- Which one to choose?

- Choose the longer one.
  - Always this works.

3.5.2 Structure of Lex Programs

A Lex program has the following form:

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

- The declarations section includes
  - Declaration of variables
  - Declaration of *manifest constants* (identifiers declared to stand for a constant)
  - Regular definitions

3.5 The Lexical-Analyzer Generator Lex

- Lex /Flex

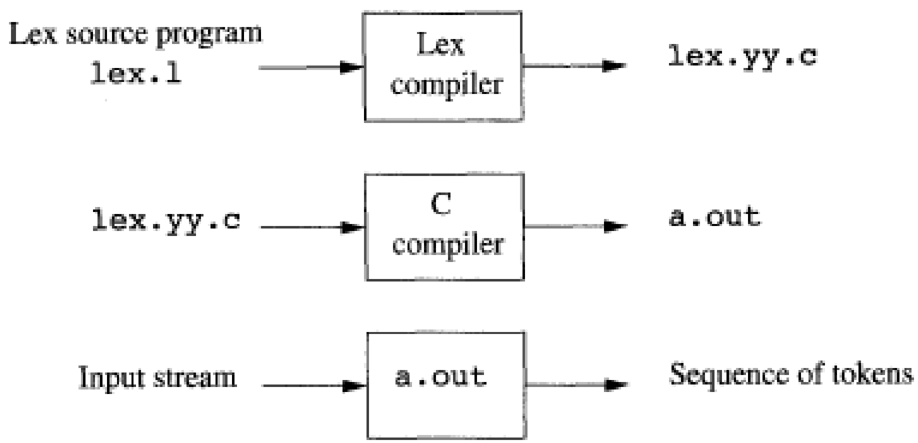

Figure 3.22: Creating a lexical analyzer with Lex

$gcc lex.yy.c
To produce a.out, you need to have main function in auxiliary section.

- The C function, $yylex(\,)$ returns an integer which is a code for one of the possible token names.
- A global variable $yylval$ is used to place the attribute value.
- $yylval$ is shared between LA and parser

Translation rules

- Each will be of the form

  $$\text{Pattern} \quad \{ \text{ Action } \}$$

- Each pattern is a regular expression (which may use regular definitions of the declaration section).
- *Actions* are fragments of C code.
- If action uses a function call. That function can be written in $auxiliary\ functions$ section.
  - Can be written in a separate C file also.

- Lex works along with parser.
- When parser calls $yylex(\,)$, the LA begins reading its remaining input, till it finds a lexeme.
  - Longest lexeme is preferred in conflict
  - First occurring definition is used in conflict
- It does the associated action which returns token number to the parser.
  - For white spaces, action is nothing; then it goes to the next lexeme which will return something.
- Attribute value is kept in the global variable $yylval$

Line count and character count

```
%{
int charcount=0, linecount=0;
%}

%%
.   {charcount++;}
\n {linecount++; charcount++;}

 %%

main() {
   while(yylex());
   printf("lines %d", linecount);
   printf("characters %d", charcount);
 }
```

I do not know whether there are any mistakes in this. If there are mistakes, correct it. See Lex tutorials/manuals.

```
%{
    /* 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}+)?

%%

{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);}

%%

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 */
}
```

Any thing between %{ and %} is directly copied to the **lex.yy.c** . C declaration can go here. Not regular definitions. #define can be used to give int codes for operators.

To use a regular definition, use { } around it.

In the auxiliary-function section, we see two such functions, `installID()` and `installNum()`. Like the portion of the declaration section that appears between `%{...%}`, everything in the auxiliary section is copied directly to file `lex.yy.c`, but may be used in the actions.

Figure 3.23: Lex program for the tokens of Fig. 3.12

The action taken when *id* is matched is threefold:

1. Function `installID()` is called to place the lexeme found in the symbol table.
2. This function returns a pointer to the symbol table, which is placed in global variable `yylval`, where it can be used by the parser or a later component of the compiler. Note that `installID()` has available to it two variables that are set automatically by the lexical analyzer that `Lex` generates:
   (a) `yytext` is a pointer to the beginning of the lexeme, analogous to `lexemeBegin` in Fig. 3.3.
   (b) `yyleng` is the length of the lexeme found.
3. The token name `ID` is returned to the parser.

The action taken when a lexeme matching the pattern *number* is similar, using the auxiliary function `installNum()`.

Whatever text is seen the same is outputted.

```
 ...
text [a-zA-Z]
%%
 ...
 ...
{text}+       {ECHO;}
  ...
 %%
  ...
```

Some actions like ECHO needs to be understood. ECHO outputs the lexeme.

3.5.3 Conflict Resolution in Lex

1. Always prefer a longer prefix to a shorter prefix.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

Programming assignment -- Individual

- In a given input C program, eliminate unnecessary white spaces, comments.
- Do proper indentation using appropriate number of tabs. So that the outputted C program is more readable.
- Give the output C program.

- Deadline 27th Jan (Saturday) 5 pm. Submit your Lex code through email attachment. Subject line “**CD Assignment 2**”
- Submit to viswanath.p@iiits.in

Reading Assignments from ToC

- Conversion of NFA to DFA
- Minimizing a DFA
- Converting a regular expression to a NFA/DFA

- These topics are covered in the dragon book.
- Some questions may appear in quiz/exam