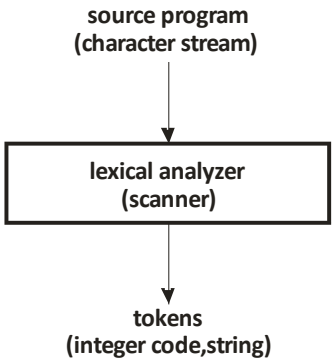

Figure 1. Lexical analyzer input and output

Input: **var** *a*,*b*,*c*:*real*;

Integer code	Integer code name	String spelling
221	**VAR**	var
200	**ID**	a
300	**COMMA**	,
200	**ID**	b
300	**COMMA**	,
200	**ID**	c
301	**COLON**	:
200	**ID**	real
302	**SEMICOLON**	;

Table 1. Lexical analyzer output for "**var** *a*,*b*,*c*:*real*;"

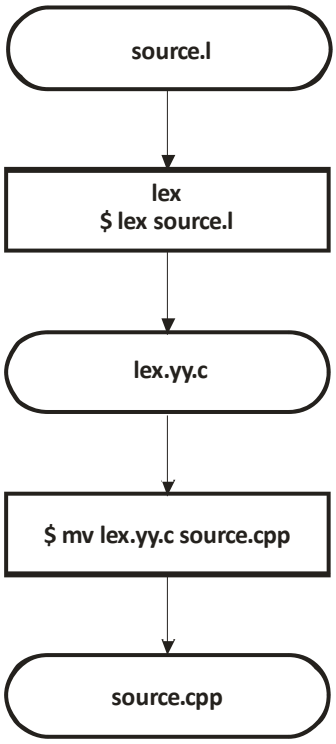

Figure 2. Invocation of lex

Notes:

1. The input file name always has the suffix **.l**
2. The output file name is always **lex.yy.c**
3. The command to invoke the lex utility
   $ lex source.l
4. Every c-program is also a c++-program. To change the output file to be a c++-program only the name needs to be changed.
   $ mv lex.yy.c source.cpp

1. ***Structure of a Lex Specification***

   … definition section
   %%
   … rules section
   %%
   … user subroutines

2. ***Definition Section***
   2.1. literal block
   %{
   … C and C++ comments, directives, and declarations
   %}
   2.2. definitions
   A definition takes the form:

   NAME expression

   The name can contain letters, digits, and underscores, and must not start with a digit.

   In the rules section, patterns may include references to substitutions with the name in braces, for example, "{NAME}". The expression corresponding to the name is substituted literally into pattern. For example.

```
DIGIT                   [0-9]
…
%%
{DIGIT}+                process_integer();
{DIGIT}+\.{DIGIT}*  |
\.{DIGIT}+              process_real();
```

Figure 1. A lex specification that containing a definition

3. ***Rules Section***
   A rule is a pattern followed by C or C++ code. For example:

   substituted literally into pattern. For example.

```
%%
[ \t\n]+ ;
%%
```

Figure 2. A lex specification that discards white space

3.1. Regular Expression Syntax

3.1.1.Metacharacters

Character	Description
.	Matches any single character except the newline character '\n'.
[]	Match any one of the characters with the brackets. A range of characters is indicated with the "-" (dash), e.g., "**[0-9]**" for any of the 10 digits. If the first character after the open bracket is a dash or a close bracket, it is not interpreted as a metacharacter If the first character is a circumflex "^" it changes the meaning to match any character except those within the brackets. (Such a character class *will* match a newline unless you explicitly exclude it.) Other metacharacters have no special meaning within square brackets except that C escape sequences starting with "\" are recognized.
*****	Matches zero or more of the preceding expression. For example, the pattern **a.*z** matches any string that starts with "a" and ends with "z", such as "az", "abz", or "alcatraz".
+	Matches one or more occurrence of the preceding regular expression. For example, **x+** matches "x", "xxx", or "xxxxx", but not an empty string, and **(ab)+** matches "ab", "abab", "ababab", and so forth.
?	Matches zero of one occurrence of the preceding regular expression. For example: **-?[0-9]+** indicates a whole number with an optional leading unary minus sign.

Character	Description
{}	A single number "{***n***}" means *n* repetitions of the preceding pattern, e.g., **[A-Z]{3}** matches any three upper case letters. If the braces contain two numbers separated by a comma, "{*n*,*m*}", they are a minimum and maximum number of repetitions of the preceding pattern. For example: **A{1,3}** matches one to three occurrences of the letter "**A**". If the second number is missing, it is taken to be infinite, so "{**1,}**" means the same as "**+**" and "**{0,}**" means the same as "*****".
****	If the following character is a lowercase letter, then it is a C escape sequence such as "**\t**" for tab. Some implementations also allow octal and hex characters in the form "**\123**" and "**\x3f**". Otherwise "****" quotes the following character, so"*****" matches an asterisk.
()	Group a series of regular expressions together. Each of the "*****", "**+**", and "**[]**" effects only the expression immediately to its left, and "**|**" normally affects everything to its left and right. Parentheses can change this, for example: **(ab|cd)?ef** matches "**abef**", "**cdef**", or just "**|**"
|	Match either the preceding regular expression or the subsequent regular expression. For example: **twelve|12** matches either "**twelve**" or "**12**"
"…"	Match everything withing the quotation marks literally. Metacharacters other than "\" lose their meaning. For example: "/*" matches the two characters
/	Matches the preceding regular expression but only if followed by the following regular expression. For example: **0/1**

matches "0" in the string "01" but does not match anything in the strings "**0**" or "**02**". Only one slash is permitted per pattern, and a pattern cannot contain both a slash and a trailing "**$**"

Character	Description
^	As the first character of a regular expression, it matches the beginning of a line; it is also used for negation within square brackets. Otherwise not special.
$	As the last character of a regular expression, it matches the end of a line – otherwise it is not special. The "**$**" has the same meaning as "**/\n**" when at the end of an expression.
<>	A name of list of names in angle brackets at the beginning of a pattern makes that pattern apply only in the given start states.

4. ***User Subroutines***

User subroutines are C and C++ functions. Function prototypes must appear before their implementations in this section.

```
%{
#include <string>
#define ID      1
#define READ    2
#define WRITE   3
#define BEGAN   4
#define END     5
int TokenMgr(int t);
%}
%%
[ \t\n]+                ;
[a-z]+                  return TokenMgr(ID);
%%
int TokenMgr(int t)
{   string rw[]={"","","read","write","begin","end"};
    for (int k=2;k<6;k++) if ((string)yytext==rw[k]) return k;
    return t;
}
```

Figure 2. A lex specification containing a user subroutine

5. ***lex and C++***

The Unix utility ***lex*** creates a C program and is designed to work with other C programs. Care must be exercised to employ ***lex*** in a C++ environment. Directives shown in figure 3 must be included to ensure the function *yylex*, the lexical analyzer produced by ***lex*** can be called from a C++ program.

```
    #ifdef __cplusplus
    extern "C"
    #endif
    int yylex (void);
```

Figure 3. C++ Preprocessor directives allowing function *yylex* to be called from a C++ program

6. ***lex and files***

Since ***lex*** creates a C program, it uses standard input/output text file definitions developed for C in include file <cstdio>. If you wish to have your scanner find tokens in an external file, you will have to redirect the standard input file from the keyboard to a FILE as defined in the include file <cstdio>. Refer to the code fragment included in figure 4.

```
#include <cstdio>
…
char ifn[255];                  //Input file name
FILE* i=fopen(ifn,"r");         //Open the file whose name is stored in string ifn.
…
yyin=i;                         // Redirect the input from the keyboard to FILE i
                                // Variable yyin is the name given to the standard
input file
                                //  by lex.
fclose(i);                      //Close FILE i.
```

Figure 4. lex and the standard input file

Invoking lex and makefiles

Typically, a programmer will want to automate the creation of a program that includes a scanner. An example ***makefile*** is given in figure 5. Note that the program consists of two source files, pas.cpp and paslex.l. File pas.cpp is compiled in the normal way. The utility lex creates file lex.yy.c from pasles.l. Then, file **lex.yy.c** is renamed to **paslex.cpp**. Next, **paslex.cpp** is translated by the C++ compiler to object file **paslex.o**. Note that *every* C program is also a C++ program. Finally, the two object files **pas.o** and **paslex.o** are bound into and executable program in file **pas**.