Compiler Design ReferencePoint Suite

SkillSoft. (c) 2003. Copying Prohibited.

Reprinted for Esteban Arias-Mendez, Hewlett Packard
estebanarias@hp.com

Reprinted with permission as a subscription benefit of **Books24x7**,
http://www.books24x7.com/

All rights reserved. Reproduction and/or distribution in whole or in part in electronic,paper or other forms without written permission is prohibited.

Table of Contents

Point 5: Understanding Lex

Ruchi Srivastava

A parser accepts input from a lexical analyzer and translates the input into the target language, which can be a high-level language such as C or a low-level language such as assembly language. Lex is a standard UNIX tool that generates a lexical analyzer.

A lexical analyzer scans an input file for patterns and returns tokens that match the pattern. You can use tools such as Lex and Yet Another Compiler Compiler (Yacc) to create lexical analyzers and parsers.

This ReferencePoint describes how lexical analyzers work. It also explains how to use Lex to develop a lexical analyzer.

Introducing Compilers and Lexical Analyzers

A compiler is an application that converts a program written in a high-level language to a low-level language. The compiler has various components such as:

- Lexical analyzer: Scans the input from left to right and generates tokens. These tokens are variables that comprise the basic unit of a programming language. Each token is a sequence of characters associated with a regular expression. A regular expression consists of symbols that describe a pattern. The symbols are numbers, characters, and some special symbols such as * and +.
- Syntactic analyzer: Uses the tokens generated by the lexical analyzer . Tokens are used to parse the input.
- Semantic analyzer: Checks the input for semantic errors.
- Code generator: Converts the intermediate code generated by the semantic analyzer to machine language.

Figure 1-5-1 shows the various components of a compiler:

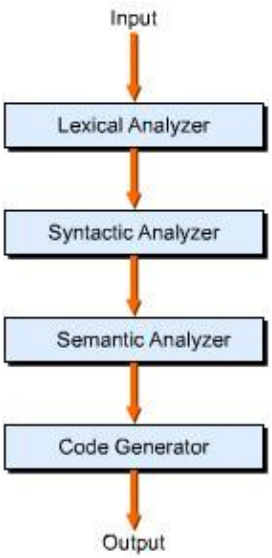

Figure 1-5-1: Components of a Compiler

Reprinted for vzzbf8, Hewlett Packard SkillSoft (c) 2003, Copying Prohibited.

Figure 1-5-2 shows the interaction between the lexical analyzer and the syntactic analyzer:

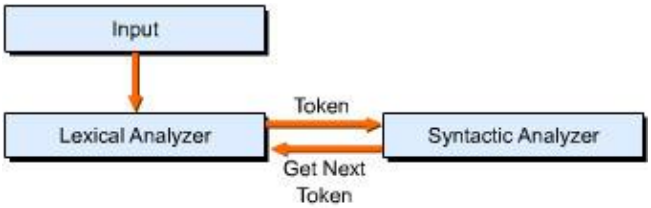

Figure 1-5-2: Interaction Between the Lexical Analyzer and the Syntactic Analyzer

Using Regular Expressions

A regular expression consists of symbols such as numbers, characters, and special symbols such as ? and *, as described in Table 1-5-1:

Table 1-5-1: Symbols in Regular Expressions

Symbol	Meaning
[]	Matches any character specified within the brackets.
*	Matches zero or more occurrences of the previous regular expression.
{}	Specifies the number of times a regular expression can occur.
|	Specifies logical OR between the regular expressions.
?	Matches zero or one occurrence of the previous regular expression.
/	Matches the previous regular expression only if followed by the next regular expression.
"..."	Takes the literal meaning of everything within the quotes.
\	Used for quoting. Quoting makes meta characters such as * and ? lose their special meaning.
( )	Used to group a series of regular expressions.
+	Matches one or more occurrence of the previous regular expression.
-	Specifies a range.
.	Matches any character apart from the new line character \n.
^	Negates a regular expression.

Regular expressions are formed using characters, numbers, and symbols. Table 1-5-2 describes some regular expressions:

Table 1-5-2: Regular Expressions

Regular Expression	Meaning
gf[pqr]*	Matches gf followed by zero or more occurrences of pqr such as gfpqrpqr and gf.
J|m|n	Matches J, m, or n.
01-?[0-9]+	Matches 01 followed by zero or one occurrence of – and followed by one or more occurrences of digits within the range 0 to 9. For example, 011 and 01-12345 are matched.
`[0-9]+G{1,5}`	Matches digits from 0 to 9 followed by 1 to 5 occurrences of G. For example, 456G and 987GGGGG.

Working with Tokens

All programming languages use tokens. For example, in C programming, tokens include the keyword and the identifier. The tokens generated by the lexical analyzer are used by the semantic analyzer to parse the input. Table 1-5-3 lists various tokens and their associated regular

SkillSoft (c) 2003, Copying Prohibited.

expressions:

Table 1-5-3: Tokens and Associated Regular Expressions

Token	Regular Expression	Meaning
Char	[A-Za-z]	Specifies any character from A to Z or a to z.
Number	[0-9]+	Specifies one or more occurrences of digits from 0 to 9.
Word	[char]+	Specifies one or more occurrences of char.
Blank	" "	Specifies a blank space.

Using the Lex Tool

The Lex tool generates a lexical analyzer by taking the lex specification as input and generating the file lex.yy.c as the lexical analyzer. This file scans the input and generates tokens according to the regular expressions defined in the Lex specification, as shown in Figure 1-5-3:

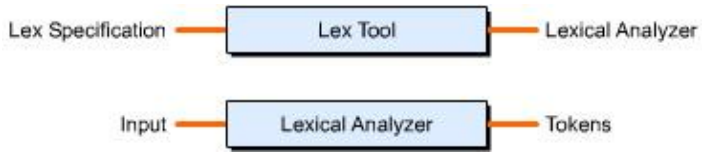

Figure 1-5-3: The Working of the Lex Tool

The Lex Specification

The lex specification is the input specified to the Lex tool. The Lex tool tokenizes the input according to the lex specification. The specification has three parts - the definition section, the rules section, and the user subroutines section.

The Definition Section

The definition section is the first part of the lex specification. This section contains header files and the declarations of the functions and variables used in the program, as shown below:

```
%{
#include<stdio.h>
#include<conio.h>
#include<string.h>
int ch;
%}
```

The above code shows the definition section, which includes header files and the variable ch that is used in the program.

The Rules Section

The rules section constitutes the main part of the lex specification. This section contains regular expressions and actions. The rules section is included between two symbols, as shown below:

```
%%
regular expression      action
%%
```

The following code shows the rules section:

```
%%
[0-9]+  printf("dig");
a|b|c     printf("alpha");
%%
```


SkillSoft (c) 2003, Copying Prohibited.

The above code shows the rules section with regular expressions and associated actions. The Lex generated scanner uses these regular expressions to tokenize the input.

Note Actions consist of the C code defined by you to perform operations such as printing the value and incrementing the value of a variable.

The User Subroutines Section

The user subroutines section is the last part of the lex specification. This section includes the main() function and all the user subroutines, as shown below:

```
main()
{
yyLex();
}
```

The above code shows the user subroutines section, which includes the main() function and the yyLex() routine called by you.

The Lex library provides several variables and functions that can be used in the Lex specification to perform various operations, as described in Table 1-5-4:

Table 1-5-4: Lex Library Functions

Library Function	Description
YyLex()	Starts the lexical analysis process. This function is automatically generated by Lex.
Ymore()	Concatenates two tokens.
Ywrap()	Enables the lexical analysis of multiple files by setting the value of the yyin variable to point to a new file to scan. This function returns zero to indicate that the lexical analysis is continued. It returns 1 to indicate that the lexical analysis is halted.
Yless(n)	Removes first n characters from the token.

The Lex variables in the Lex specification are used for various purposes, as described in Table 1-5-5:

Table 1-5-5: Lex Variables

Lex Variables	Description
Yytext	Stores the text of the token matched by the lexical analyzer. This variable is a character array. The value of yytext changes when the scanner matches a new token.
Yyin	Points to the standard input, the keyboard, by default. This variable is a pointer to the current file being scanned by the scanner.
Yyout	Points to the standard output, the computer screen, by default. This variable is a pointer to the location where the output of the scanner is written.
Yyleng	Returns the length of the matched pattern.

Developing a Lexical Analyzer

To develop a lexical analyzer using Lex, you need to:

1. Save the Lex specifications in a file with extension .l, as shown below:

```
file_name.l
```


SkillSoft (c) 2003, Copying Prohibited.

Listing 1-5-1 shows your lex specification to the Lex tool. The Lex tool tokenizes the input according to your specification:

Listing 1-5-1: Code for the Lex Specification

```
//definition section
%{
#include<stdio.h>
#include<conio.h>
#include<string.h>
%}
//rules section
%%
//regular expression with the action
[0-9]+  printf("dig");
a|b|c printf("char");
%%
//user subroutines section
main()
{
yyLex();
}
```

The above code shows the lex specification specified as an input to the Lex tool. In the code, the definition section is the first part of the lex specification. This section contains header files such as stdio.h, conio.h, and string.h. The rules section specifies two regular expressions and their associated actions. The user subroutines section consists of only one function, yylex(), which is used to scan the input. This code generates the first.l file, as shown in Figure 1-5-4:

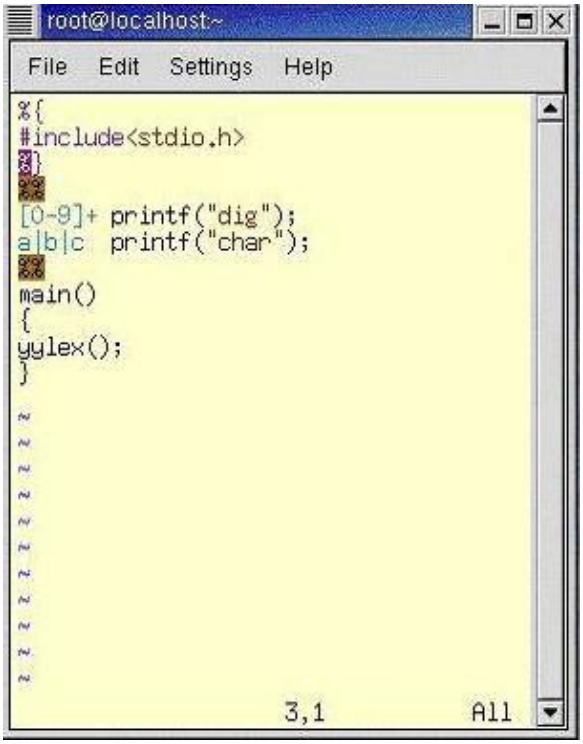

Figure 1-5-4: The Lex Specification

SkillSoft (c) 2003, Copying Prohibited.

2. Type $Lex file_name.l at the command prompt. To complete the compilation process, compile the Lex.yy.c lexical analyzer generated by the Lex tool and link it to the Lex library using –ll, as shown below:

```
$cc Lex.yy.c -ll
$./a.out
```

3. Pass the input string to be tokenized.

Figure 1-5-5 shows the use of Lex tool to tokenize your input as 123 and abab. The input, 123, generates the token dig and the input, abab, generates tokens charcharcharchar, as specified in the lex specification:

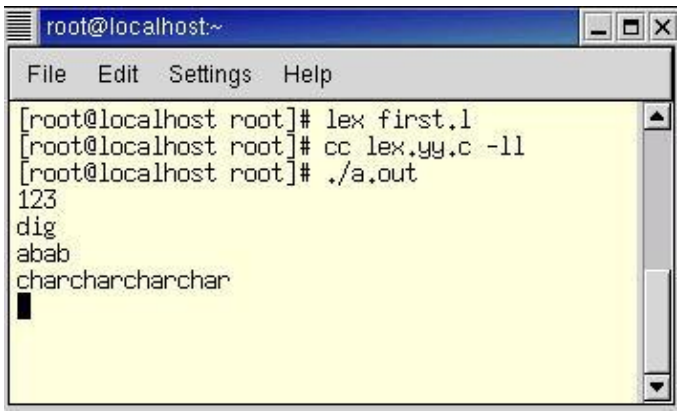

Figure 1-5-5: Using the Lex Tool to Tokenize the Input

Developing a Lex Tool

You can develop a Lex tool using programming languages such as C and C++. To illustrate the development of a Lex tool, the sample application, called MyLex, is described here. MyLex is a C++ application that contains various functions for various symbols that can be used in the regular expressions defined by you such as simp_fun, star_fun for *, dot_fun for, and plus_fun for +.

Using the void simp_fun(char *) Function

The void simp_fun(char *) function is called for all regular expressions that contain no symbols. For example, for a regular expression [rty], the void simp_fun(char *) function is called.

Listing 1-5-2 shows the code for the simp_fun function that is called when the regular expression contains no symbols:

Listing 1-5-2: Function for Regular Expression Using no Symbols

```
void simp_fun(char *a)
{
    cplus<<"\nif(";
    cplus<<"strnicmp(c,\""<<a<<"\","<<strlen(a)<<")!=0";
    cplus<<")\n\t";
    cplus<<"unmatch=1;\nelse if(strnicmp(c,\""<<a<<"\","<<strlen(a)<<")==0)\n\t";
    cplus<<"{for(int i=0,j="<<strlen(a)<<";j<strlen(c);i++,j++)\n\t";
    cplus<<"c[i]=c[j];\nc[i]=\'\\0\';\nflag=1;\nunmatch=2;}\n";
}
```

The above code shows the simp_fun function, called in the application when the regular expression specified contains no symbols such as * and +.

Using the void star_fun(char *) Function

The void star_fun(char *) function is called when you specify the (*) symbol in a regular expression. For example, for a regular expression [asd]*, this function is called after the simp_fun function.

Listing 1-5-3 shows the code for the star_fun function, which is called when you use the symbol * in the regular expression:

Listing 1-5-3: Function for Regular Expression Using the * Symbol

```
void star_fun(char *a)
{
        cplus<<"\n";
        cplus<<"while(strlen(c)!=0&&unmatch!=1)\n\t";
        cplus<<"{";
        func(a);
        cplus<<"}\n";
}
```

The above code shows the star_fun function, which is called in the application when the regular expression specified contains the symbol *.

Using the void dot_fun()Function

The void dot_fun() function is called when you specify the dot operator (.) in a regular expression. For example, for a regular expression (da), this function is called after the simp_fun function.

The following code shows the dot_fun function called when you use the dot operator in the regular expression:

```
void dot_fun()
{
        cplus<<"\nif(c[0]!=\'\\n\')";
        cplus<<"{for(int i=0;i<strlen(c)-1;i++)\n\t";
        cplus<<"c[i]=c[i+1];\nc[i]=\'\\0\';}\n";
}
```

The above code shows the dot_fun function that is called when the regular expression specified contains the dot operator.

Using the void plus_fun(char *) Function

The void plus_fun(char *) function is called when you specify the plus operator in a regular expression. For example, for the regular expression (gkj)+, this function is called after the simp_fun function.

The following code shows the plus_fun function called when the regular expression contains the plus symbol:

```
void plus_fun(char *a)
{
        cplus<<"flag=0;\nunmatch=0;\n";
        func(a);
        star_fun(a);
}
```

The above code shows the plus_fun function that is called when the regular expression contains the plus symbol.

SkillSoft (c) 2003, Copying Prohibited.

Listing 1-5-4 shows code that implements the Lex tool MyLex.cpp, which accepts the Lex specification as input and generates the lexical analyzer :

Listing 1-5-4: Implementing MyLex.cpp

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<process.h>
#include<fstream.h>
#include<ctype.h>
#include<string.h>
/* function declarations */
//function for plus operator
void plus_fun(char *);
//function for star operator
void star_fun(char *);
//function for simple regular expression without any //operator
void simp_fun(char *);
//function for dot operator
void dot_fun();
void func(char *);
/* declarations for global variables */
fstream lexi,cplus;
/* main function */
void main()
{
  //array a stores your input in the form of Lex //specification
  char  a[50];
  int l,flag=1,inidecl=0;
  int i,f1;
  clrscr();
  cout<<"enter the program"<<"\n";
  //lex.cpp is  the lexical analyzer generated by the //code
  cplus.open("lex.cpp",ios::out);
  //lex.dat is the temporary file that stores the rules  //section of the Lex specification.
  lexi.open("lex.dat",ios::out);
  /* entering the lex program and seperating into directly executable code and the one which ha
  do
  {
    //takes the input as Lex specification
    gets(a);
    //finds the length of the lex specification
    l=strlen(a);
    a[l]='\0';
    //exits when you hit &
    if (a[l-1]==38)
    {
      a[l-1]='\0';
      flag=0;
    }
    l=strlen(a);
    if (inidecl==0 &&  strcmp(a,"%{")!=0&&strcmp(a,"%}")!=0)
    {
      //input to the lexical analyzer generated by //the code.
      cplus<<a;
      cplus<<'\n';
    }
    if(strcmp(a,"%}")==0&&inidecl==0)
      inidecl=1;
    if(inidecl==2&&strcmp(a,"%%")==0)
      inidecl=0;
    if(strcmp(a,"%%")==0&&inidecl==1)
      inidecl=2;
    //The input between the %% sign, that is, in the //rules section goes to the lex.dat file
    if(inidecl==2&&strcmp(a,"%%")!=0)
    {
      lexi<<a;          //input goes to the .dat file
      lexi<<'\n';
    }
  }while(flag!=0);
  //lex.dat is closed
  lexi.close();
  //lex.dat is opened for reading the input to be //written to lex.cpp
```


SkillSoft (c) 2003, Copying Prohibited.

```
  lexi.open("lex.dat",ios::in);
  /* now constructing the main program in the cpp file */
  cplus<<"void main()\n";
  cplus<<"{";
  cplus<<"clrscr();";
  cplus<<"\nchar b[50],c[50];";
  cplus<<"\ncout<<\"enter ur string/program\\n\";";
  //the user inputs the string in here in lex.cpp
  cplus<<"\ngets(b);\n";  cplus<<"for(int i=0;i<strlen(b);i++)\n";
  cplus<<"c[i]=b[i];\nc[i]=\'\\0\';\n";
  cplus<<"int unmatch=0,flag=0;\n";
  cplus<<"int k=0;\n";
  /* scans the input string till the whole string matches one or other rule or the remaining st
  cplus<<"while(k<=strlen(c))\n{";
  //file pointer comes to the beginning of the file //lex.dat
  lexi.seekg(0,ios::beg);
  f1=0;
  /* forming the if constructs in the final cpp file from the translation rules in the lex.dat
  //lex.dat is read till the end of file is encountered
  while (!lexi.eof())
  {
    cplus<<"\n";
    //file pointer goes to the beginning of the lex.dat
    lexi.seekg(f1);
    //20 characters from lex.dat are put into the array a
    lexi.get(a,20,' ');
    //gives the current location of the file pointer
    i=lexi.tellg();
    //moving one bit ahead in position
    f1=i+1;
    //takes the file pointer to the current location
    lexi.seekg(f1);
    if(a[0]!='\0')
    {
      // calling function func() if the string
      //from.dat file is not null
       func(a);
      //50 characters from the file lex.dat are put into //the array a.
       lexi.get(a,50,'\n');
      //gives the current location of the file pointer
       i=lexi.tellg();
      //moving to the next rule
       f1=i+4;
      //file pointer goes to the current location
       lexi.seekg(f1);
       f1-=2;
      /* if string satisfies the rule then the code written adjacent to it is executed accordin
      cplus<<"if (flag==1||(flag==0&&unmatch!=1))\n";
      cplus<<a<<'\n';
      cplus<<"if(unmatch==1)\n\tunmatch=0;\nflag=0;\n";
    }
  }
  //end while of lex.cpp
  cplus<<"k++;\n}";
  //end main of lex.cpp
  cplus<<"}";
  //closing the file lex.cpp
  cplus.close();
  //closing the file lex.dat
  lexi.close();
  getch();
}
/*The function func() checks for the unbalanced parenthesis in your input. The function transla
void func(char *f)
{
  char *a;
  a=f;
  int flag=0;
  int i,r;
  int k=0,t;
  char b[10],c[10],d[10],e[10];
  b[0]=c[0]=d[0]=e[0]='\0';
  //checking for unbalanced parenthesis
  for(i=0;i<strlen(a);i++)
  {
```

 SkillSoft (c) 2003, Copying Prohibited.

```
    if(a[i]=='['||a[i]=='(')
    flag++;
    if(a[i]==']'||a[i]==')')
    flag--;
  }
  if(flag!=0)
  //giving the valid output if unbalanced //parenthesis
  {
    cout<<"invalid specification"<<'\n';
    exit(0);
  }
  //if balanced parenthesis
  else
  {
    i=0;
    while(a[i]!='\0')
    {
      if(a[0]!='('&&a[0]!='[')
      /* forming the string c from the rules such that it contains the string that comes before
      while(a[i]!='('&&a[i]!='['&&a[i]!='\0')
      c[i]=a[i++];
      c[i]='\0';
      c[i+1]='\0';
      int s=0;
      /* now forming the rule for the string c */
      while(c[s]!='\0')
      {
        k=0;
        while(c[s]!='*'&&c[s]!='+'&&c[s]!='.'&&c[s]!='\0')
        e[k++]=c[s++];
        if(c[s]=='+'||c[s]=='*')
          e[--k]='\0';
        else
          e[k]='\0';
          /* call simp_fun() for those strings that don't have any operator associated with the
          simp_fun(e);
          /* call corresponding functions for the strings with operators associated with them *
          if(c[s]=='*')
            star_fun(&c[s-1]);
          if(c[s]=='+')
            plus_fun(&c[s-1]);
          if(c[s]=='.')
            dot_fun();
        }
      /* finding the first parenthesis in the rule string and storing it in the array b */
      if(a[i]=='['||a[i]=='(')
      {
        flag++;
        i++;
        for(int j=i;flag!=0&&j<strlen(a);j++)
        {
          if(a[j]=='['||a[j]=='(')
          flag++;
          if(a[j]==']'||a[j]==')')
          flag--;
        }
        k=0;
        for(t=--i;t<j;t++,k++)
        b[k]=a[t];
        b[k]='\0';
      }
      if(b[0]!='\0')
      {
        /* removing the parenthesis */
                        for(r=0;r<strlen(b)-2;r++)
                        b[r]=b[r+1];
      }
      b[r]='\0';
      i=t;
      /* storing the remaining string in array d */
      for(t=0;i<strlen(a);t++,i++)
      d[t]=a[i];
      d[t]='\0';
      /* calling the functions corresponding to the operator following the first parenthesized
      if(d[0]=='*')
```

```
        star_fun(b);
      else if(d[0]=='+')
        plus_fun(b);
      else if(d[0]=='.')
      {
        func(b);
        dot_fun();
      }
      else if(b[0]!='\0')
        func(b);
      /* if first character of d is an operator then shift d by one position to the left */
      if(d[0]=='*'||d[0]=='+'||d[0]=='.')
      {
        for(int r=0;r<strlen(d)-1;r++)
        d[r]=d[r+1];
        d[r]='\0';
      }
      /* forming translation rules for d */
      if(d[0]!='\0')
      func(d);
      a[i]='\0';
     }
    }
  }
  /* The function for the operator star(*).If a string is followed by a star ,that is,'*' then
  void star_fun(char *a)
  {
    cplus<<"\n";
    cplus<<"while(strlen(c)!=0&&unmatch!=1)\n\t";
    cplus<<"{";
    func(a);
    cplus<<"}\n";
  }
  /* The function for the operator plus(+).If a string is followed by a plus, that is,'+' then
  void plus_fun(char *a)
  {
    cplus<<"flag=0;\nunmatch=0;\n";
    func(a);
    star_fun(a);
  }
  /* The function for the operator dot(.).If a string is followed by a dot, that is,'.' then it
  void dot_fun()
  {
    cplus<<"\nif(c[0]!=\'\\n\')";
    cplus<<"{for(int i=0;i<strlen(c)-1;i++)\n\t";
    cplus<<"c[i]=c[i+1];\nc[i]=\'\\0\';}\n";
  }
  /* The function for a simple string without any operators */
  void simp_fun(char *a)
  {
    cplus<<"\n if(";
    cplus<<"strnicmp(c,\""<<a<<"\","<<strlen(a)<<")!=0";
    cplus<<")\n\t";
    cplus<<"unmatch=1;\n else if(strnicmp(c,\""<<a<<"\","<<strlen(a)<<")==0)\n\t";
    cplus<<"{for(int i=0,j="<<strlen(a)<<";j<strlen(c);i++,j++)\n\t";
    cplus<<"c[i]=c[j];\nc[i]=\'\\0\';\nflag=1;\nunmatch=2;}\n";
  }
```

The above code generates a Lex tool called MyLex. The code contains various functions for the various symbols used in the regular expressions such as:

- The simp_fun function: Called for regular expressions that do not contain symbols.
- The star_fun function: Called for regular expressions that contain the star (*) symbol.
- The dot_fun function: Called for regular expressions that contain the dot (.) symbol.

SkillSoft (c) 2003, Copying Prohibited.

- The plus_fun function: Called for regular expressions that contain the plus (+) symbol.

The above code also creates two files, lex.dat and lex.cpp. The lex.dat file is a temporary file that stores rules in the rules section of the lex specification specified as input to the MyLex tool. The code generates a lexical analyzer the lex.cpp file. The lexical analyzer uses the contents of the temporary file, lex.dat, to generate tokens according to the rules in the rules section. The lexical analyzer is generated according to the lex specification. The Lex specification consists of two parts:

- Definition section

  The following code shows the definition section of the lex specification:

```
/*Definition Section*/
//This section is included between the two symbols, //that is, %{ %}. This section conta
%{
#include<stdio.h>
#include<iostream.h>
#include<string.h>
%}
```

  The code shows the definition section of the lex specification. This section contains three header files, stdio.h, iostream.h, and string.h. The second part of the lex specification is the rules section, which is included between the symbols %% rules and actions %%. This section contains regular expressions and their associated actions. The regular expression is included between square brackets or plain brackets. The regular expression can be a simple string without symbols or with various symbols, such as star (*), plus (+), and dot(.). The action is specified by the cout statement.

- Rules section

  The following code shows the rules section of the lex specification:

```
%%
[qpr]*  cout<<"fd";
(ads).  cout<<"ew";
%%
```

  The code shows the rules section, which contains two regular expressions, [qpr]* and (ads) and their associated actions. This code generates the lexical analyzer, the lex.cpp file. The input to the code is the lex specification consisting of the definition section and the rules section. The generated lexical analyzer, lex.cpp, is compiled and given the input string to be tokenized. The lexical analyzer tokenizes the input according to the regular expressions and their associated actions defined by you in the lex specification.

Figure 1-5-6 shows the lex specification as input to the MyLex tool:

SkillSoft (c) 2003, Copying Prohibited.

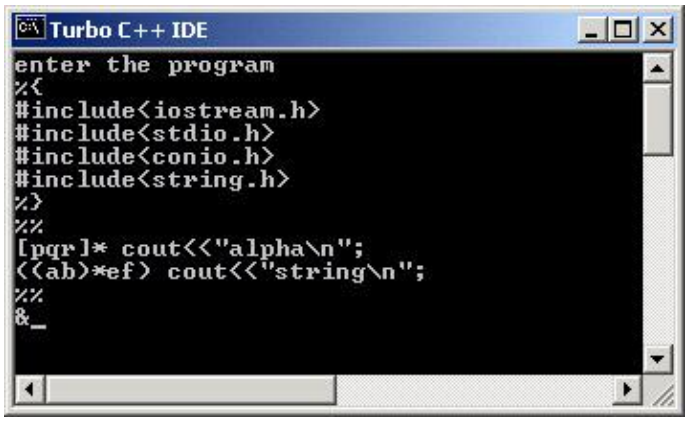

Figure 1-5-6: The Lex Specification as Input to the MyLex Tool

Listing 1-5-5 shows code for the lexical analyzer, lex.cpp generated by the MyLex tool:

Listing 1-5-5: Lexical Analyzer Generated by the MyLex Tool

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>

void main()
{
char b[50],c[50];
cout<<"enter your string/program\n";
gets(b);
for(int i=0;i<strlen(b);i++)
c[i]=b[i];
c[i]='\0';
int unmatch=0,flag=0;
int k=0;
while(k<=strlen(c))
{
while(strlen(c)!=0&&unmatch!=1)
        {
if(strnicmp(c,"pqr",3)!=0)
        unmatch=1;
else if(strnicmp(c,"pqr",3)==0)
        {
        for(int i=0,j=3;j<strlen(c);i++,j++)
         c[i]=c[j];
       c[i]='\0';
       flag=1;
       unmatch=2;
         }
    }
     if (flag==1||(flag==0&&unmatch!=1))
         cout<<"alpha\n";
    if(unmatch==1)
        unmatch=0;
     flag=0;
     while(strlen(c)!=0&&unmatch!=1)
        {
         if(strnicmp(c,"ab",2)!=0)
                  unmatch=1;
            else
            if(strnicmp(c,"ab",2)==0)
              {
           for(int i=0,j=2;j<strlen(c);i++,j++)
               c[i]=c[j];
             c[i]='\0';
             flag=1;
             unmatch=2;
                }
        }
if(strnicmp(c,"ef",2)!=0)
```

```
        unmatch=1;
else if(strnicmp(c,"ef",2)==0)
        {
          for(int i=0,j=2;j<strlen(c);i++,j++)
             c[i]=c[j];
          c[i]='\0';
           flag=1;
          unmatch=2;
          }
        if (flag==1||(flag==0&&unmatch!=1))
               cout<<"string\n";
          if(unmatch==1)
           unmatch=0;
         flag=0;
         k++;
          }
}
```

The above code shows the lexical analyzer, lex.cpp generated by the MyLex tool. The lexical analyzer generates tokens for the input according to the regular expressions and their associated actions. You define the regular expressions and their associated actions in the rules section of the lex specification given as input to the MyLex tool. The two regular expressions, [pqr]* and ((ab)*ef), are defined by you. Both these regular expressions contain the * symbol. The lex.cpp, generated by the MyLex tool is compiled and is passed the input to tokenize.

Figure 1-5-7 shows the tokenization of input to the generated lexical analyzer, lex.cpp:

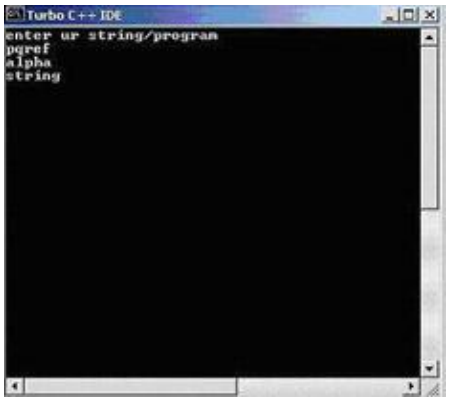

Figure 1-5-7: Tokenization of Input

Figure 1-5-8 shows the lex specification containing two regular expressions, one with the dot operator and one without an operator:

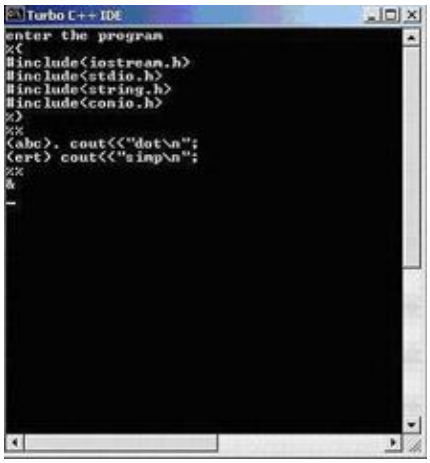

Figure 1-5-8: Lex Specification

SkillSoft (c) 2003, Copying Prohibited.

Listing 1-5-6 shows the lexical analyzer, lex.cpp, that is generated according to your lex specification to the MyLex tool:

Listing 1-5-6: The Lexical Analyzer lex.cpp

```
#include<iostream.h>
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{
  clrscr();
  char b[50],c[50];
  cout<<"enter your string/program\n";
  gets(b);
  for(int i=0;i<strlen(b);i++)
  c[i]=b[i];
  c[i]='\0';
  int unmatch=0,flag=0;
  int k=0;
  while(k<=strlen(c))
  {
    if(strnicmp(c,"abc",3)!=0)
      unmatch=1;
    else
      if(strnicmp(c,"abc",3)==0)
      {
        for(int i=0,j=3;j<strlen(c);i++,j++)
        c[i]=c[j];
        c[i]='\0';
        flag=1;
        unmatch=2;
      }
    if(c[0]!='\n'){for(int i=0;i<strlen(c)- 1;i++)
    c[i]=c[i+1];
    c[i]='\0';
  }
  if (flag==1||(flag==0&&unmatch!=1))
    cout<<"dot\n";
    if(unmatch==1)
      unmatch=0;
      flag=0;
      if(strnicmp(c,"ert",3)!=0)
        unmatch=1;
      else
        if(strnicmp(c,"ert",3)==0)
        {
          for(int i=0,j=3;j<strlen(c);i++,j++)
          c[i]=c[j];
          c[i]='\0';
          flag=1;
          unmatch=2;
        }
    if (flag==1||(flag==0&&unmatch!=1))
    cout<<"simp\n";
    if(unmatch==1)
      unmatch=0;
    flag=0;
    k++;
    }
}
```

The above code shows the lexical analyzer, lex.cpp generated by the MyLex tool. The lexical analyzer generates tokens for the input according to the regular expressions and the associated actions as defined by you in the rules section of the lex specifications given to the MyLex tool. You define two regular expressions, abc and ert. One contains the dot operator. The other does not contain any symbols. The lex.cpp, the lexical analyzer generated by the MyLex tool, is compiled. When input is specified to lex.cpp, it generates tokens.

SkillSoft (c) 2003, Copying Prohibited.

Figure 1-5-9 shows the tokenization of input according to the regular expressions and the associated actions defined:

Figure 1-5-9: Tokenization of Input

Related Topics

For related information on this topic, you can refer to:

- Introducing Compiler Design
- Principles of Compiler Design
- Introducing Optimization in Compiler Design
- Error Detection and Recovery
- Understanding Yacc

SkillSoft (c) 2003, Copying Prohibited.