Lexical Analysis

The first phase of a compiler is the lexical analyzer. Since the lexical analyzer scans through the input file, the lexical analyzer is also often referred to as a scanner. There are two main responsibilities of a lexical analyzer:

1. Chop the stream of input characters into tokens
2. Store identifiers into symbol table

How to define tokens? Your lexical analyzer needs to be able to recognize the following:

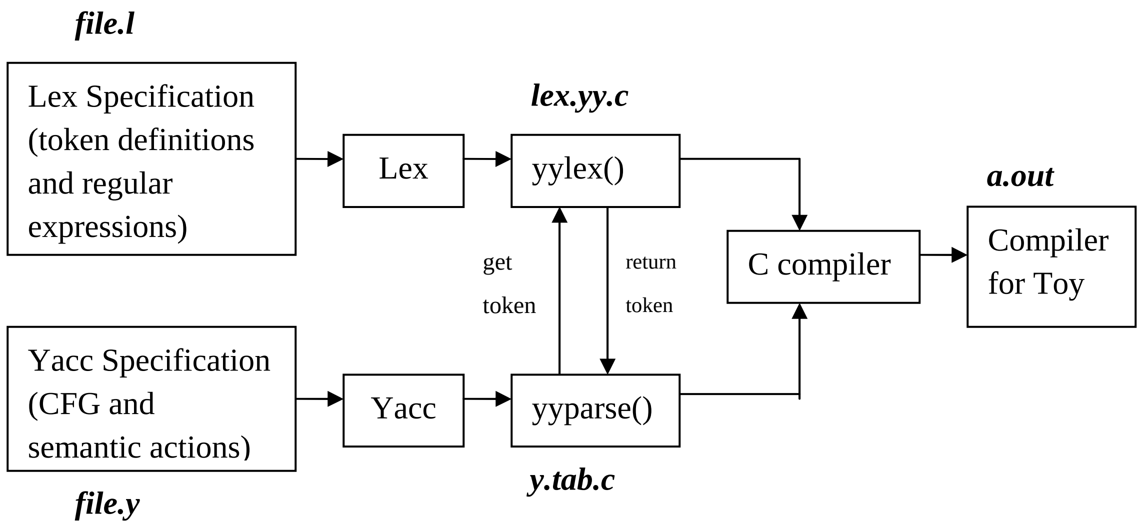
- Reserved words
- Operators and special symbols
- Identifiers
- Constants (such as integer literals, double literals, and string literals)

A lexical analyzer can be implemented as a finite state automaton with multiple paths defined from the initial state, one for each token. For example, if the first non-blank character is of value ‘a’ to ‘z’ or ‘A’ to ‘Z’, then we know this is the beginning of an identifier definition. The corresponding path will then continue processing characters until we see something that is no longer in the range of a..z, A..Z, or 0..9. In other words, the path will continue processing characters until it finds a delimiter. If the identifier is one of the reserved words, then this path will generate a token which is the corresponding reserved word. Otherwise, this path generates a generic token called ID. Please refer to the diagram and examples given in class for details of the finite state automaton.

Next, we will show how to use C and Lex to build a lexical analyzer. Students may choose other tools, such as JavaCC or JLex, for implementation.

Introduction of Lex and Yacc

Lex and Yacc are compiler construction tools designed for the Unix operating system, and target to the C programming language. Yacc stands for Yet Another Compiler Compiler. You may use Yacc without using Lex. In fact, Yacc was developed the first of the two. The following diagram illustrates the use of Lex and Yacc.


file.l
Lex Specification (token definitions and regular expressions)
Lex
lex.yy.c
yylex()
get token
return token
C compiler
a.out
Compiler for Toy
Yacc Specification (CFG and semantic actions)
Yacc
yyparse()
y.tab.c
file.y

General Format of a Lex Specification

```
definition
%%
rules
%%
user subroutines
```

The rules represent the user’s control decisions. They are a table, in which the left column contains regular expressions and the right column contains actions, i.e. C program fragments to be executed when the expressions are recognized. For example,

```
     integer   printf(“found keyword INT”);
```

Lex Regular Expressions

x	the character x
“x” or \x	an “x” even if x is an operator
x*	0,1,2,… instances of x
x+	1,2,… instances of x
x?	an optional x
x | y	an x or an y
[xy]	matches a single character x or y
[x-z]	matches x or y or z
[^x]	any character but x
x/y	an x but only if followed by y
(x)	an x
.	any character but newline
^x	an x at the beginning of a line
x$	an x at the end of a line
{}	repetitions or definition expansion

Examples:

*	represents kleene closure
“*” or *	represents text character *
xyz+	matches xyzzzz
xyz”+”	matches xyz+
ab?c	matches ac or abc
ab|cd	matches ab or cd
[-+0-9]	all digits and two signs
[^abc]	all characters except a, b, or c
ab/cd	matches ab only if followed by cd
(cd)+	matches at least one copies of cd
(“+”|”-“)?[0-9]+	signed or unsigned integers
[^A-Za-z]	any character which is not a letter
“//”.*	line comment
“{“[^\}]*”}”	Pascal comment, multiple lines allowed
ab$	equal to ab/\n
a{1,5}	matches 1 to 5 occurrences of a
{digit}	substitute a predefined definition named digit

Lex Variables

yytext   a global variable to store the matched token
          (yytext is of character array type)

yyleng   a global counter for the # of characters matched

For example,

[a-zA-Z]+ {word++; chars+=yyleng;} accumulates in chars the number of characters in the words recognized.

Token Definitions

For efficiency purpose, tokens are represented as integers internally. Since tokens are passed between Lex and Yacc, Lexer and parser have to agree what token codes are. Practically, we let Yacc define the token codes in y.tab.h and just include y.tab.h in the lexer.

The following is an example of y.tab.h where we define 9 tokens for reserved words, special symbols, number constant, and user-defined identifiers. Note that yylval is an integer used as an attribute value, which later on will be defined in Yacc.

```
intranet (54) % cat y.tab.h
#define t_INT 1000
#define t_FLOAT 1001
#define t_ADDOP 1002
#define t_MULOP 1003
#define t_COMMA 1004
#define t_SEMICOL 1005
#define t_DOT 1006
#define t_NUM 1007
#define t_ID 1008

int yylval;
```

An example of Lex Specification file

A Lex file must have filename with extension ***.l***. The following file named ***test2.l*** is an example of a Lex specification. Lines 1 through 20 are definitions. Note that everything defined in between %{ and %} will be moved to C code directly. Lines 22 through 34 are rules, and lines 36 to 50 are user subroutines.

```
1   %{
2   #define MAX_LENGTH 10
3   #define MAX 100
4   #define INT 1
5   #define FLOAT 2
6   #include <stdio.h>
7   #include <string.h>
8   #include "y.tab.h"
9   struct {
10     char name[MAX_LENGTH];
11     int type;
12  }table[MAX];
13  int t_index=0;
14  int t_flag=0;
15  %}
16  letter  [a-zA-Z]
17  digit   [0-9]
18  id      {letter}({letter}|{digit})*
19  number  {digit}+
20  ws [ \t\n]
21
22  %%
23  {ws}          ;
24  int           {t_flag=INT; printf("%s\n", yytext); return (t_INT);}
25  float         {t_flag=FLOAT; printf("%s\n", yytext); return (t_FLOAT);}
26  "+"           {printf("%s\n", yytext); return (t_ADDOP);}
27  "*"           {printf("%s\n", yytext); return (t_MULOP);}
28  ","           {printf("%s\n", yytext); return (t_COMMA);}
29  ";"           {printf("%s\n", yytext); return (t_SEMICOL);}
30  "."           {printf("%s\n", yytext); return (t_DOT);}
```

```
{number}   {printf("%s\n", yytext); return (t_NUM);}
{id}       {yytext[yyleng]='\0'; yylval=insert(yytext);
             printf("%s\n", yytext); return(t_ID);}
.          {printf("error!\n"); return (0);}

%%
int insert(char *s) {
    int i=0;
    while (i< t_index) {
        if (strcmp(s, table[i].name)==0) return i;
        i++;
    }
    strcpy(table[t_index].name, s);
    table[t_index].type = t_flag;
    t_index++;
    return t_index-1;
}
int main () {
    yylex();
}
```

Compilation and Execution

After compile your Lex specification file with command *lex*, an important C function named yylex() will be generated in file lex.yy.c. The yylex program will recognize expressions in a stream and perform the specified actions for each expression it is detected. Now compile lex.yy.c with a C compiler, link it with the Lex library, and you will get the executable a.out.

```
intranet (56) % lex test2.l
intranet (57) % cc lex.yy.c -ll
intranet (58) % a.out
hello
hello
intranet (59) % a.out
123
123
```

If we add a loop in main()

```
   while (yylex()) {}
```

The program will continue process input until EOF or control-D from keyboard.

For instance,

```
intranet (60) % cat input.dat
123
abc
  345
a1b2
 2a3b
4*3+id.;,
intranet (61) % a.out < input.dat
123
abc
345
a1b2
2
a3b
4
*
3
+
id
.
;
,
```