07-09-21 | CS6109 · COMPILER DESIGN

SACHIN RAGHUL T

WEEK 01 · OBSERVATION

2019103573

SYNTAX :

1) Format of lex input :

declaration of definition

%%

tokens rule : (or) translation rules

%%

Auxiliary procedure of subroutines.

2) Declaration :

a) string sets ;

b) standard c ; name character. class

%{ ...c declarations...

%}

3) Token rules :

a) If the expression includes a reference to a character class, enclose the class name in bracket { }.

b) regular expression operations :

*, +	→ closure, positive closure
' " or \	→ protection of special chars
|	→ or
^	→ begining of line anchor
( )	→ grouping
$	→ end of line anchor
?	→ zero or one
.	→ any char (except \n)
{ref}	→ reference to a named character class
[ ]	→ character class.
[^]	→ no character class.

4) Match rules :

* Longest match is preferred.

* If 2 matches are equal length, the first match is prefered. Remember lex partition it does not attempt to fixed nested matches once a character becomes part of a match, it is no longer considered for other matches.

5) Built in variables :

yytext → ptr to the matching lexeme (char * yytext ;)
yylex → length of matching lexeme (yytext).

6) Aux procedures :

c functions may be defined and called from the c-code of token rules or from other functions. Each of file should also have a yyerror() function to be called when lex encounter an error condition.

7.) Example header file : tokens.h

```
# define PLUS   3
# define NUM    1   // define constant used by lex
# define ID     2   // could be defined in lex rule file.
```

Example lex files :

```
    D [0-9]
    A [a-z A-Z]
    %{
    # include "token.h"
    %}
    %%

{D}+                 return (NUM);    // match int_no
{A}({A}|{D})         return (ID) :    // match identifier
"+"                  return (PLUS);   // match the plus sign
%%
```

```
void yyenor ()                         // default action is caused
{
  printf ("error \n");                  // error in yylex()
  exit (0);
}
void yywrap() { }                       // usually only needed for some linux system
```

8) Execution of lex :

To generate the yylex() function & then compile a user program.

(MS) c : flex rule file   (linux) $ lex rulefile

flex produce leyy.c   lex produces lex.yy.c

The produced .c file contains this function : int yylex()

9) User program :

```
# include <stdio.h>
# include <tokens.h>
int yylex();
 extern char * yytext ;
 main () {
    int n;
    while (n= yylex())   // call scanner until it returns for EOF
       Printf ("%d %s\n", n,yytext);
   }
                         // output the token code & lexeme string.
```

Pattern matching primitives :

^	→	begining of line
\n	→	newline
a|b	→	a orb
(ab)+	→	one or more copies of ab (grouping)
[ ]	→	character class.

Pattern matching example :

abc	→ abc
abc*	→ ab abc abcc abccc . . .
a (bc) ?	→ a abc
[abc]	→ one of a, b, c
a /b	→ one of a , b .

Lex predefined variables :

ECHO		→ write matched string
BEGIN		→ condition switch start condition
INITIAL		→ initial start condition.
FILE	*yyin	→ input file
FILE	*yyout	→ output file.

Regular expression :

```
delim      [ \t\n]
ws         {delim}+
letter     [a-z A-Z]
digit      [0-9]
^num       {{num}|{num}([Ee][^-]? {unum})?
```

Translation rules :

Translation rules are constructed as follows.

```
r.e.1    {action1}
r.e.2    {action 2}
 . - - -
r.e.n    {action n}
```

The actions are c code to be carried out when the regular expression matches the input.

For eg :

```
{ws}          { "Inothing";}
{li} [ff] .   { return (IF); }
```

compiling (f)lex :

Create your lexical source is the file lex.l and then compile it with the command

```
lex lex.l.
```

The output of flex is a C source file lex.yy.c which you must compile with the compiler.

```
gcc lex.yy.c -lfl
```

→ lex can be used as a standalone program generated and does not have to be part of a larger compiler system.

→ lex.cc.y can be set to another filename within flex as can be the input file name

→ The key function yylex() can be generated and combined with other function code instead of being connected to the standard executable a.out.

→ '-lfl' library within which scanner must be linked.

→ 'lex.yy.c' generated C++ scanner when using '-+'

→ '<FlexLexer.h>' header file defining the C++ scanner class, flex lexer - it derived class yy flexlexer.

→ 'flex.skl' skeletion scanner this file is only used when building flex not when flex execute

→ 'lex.backup' backing up information for '-b' flag.

```
%{
#include <stdio.h>
%}
% opion noyywrap
%%
[0-9]+{
    printf("saw an integer: %s\n", yytext); }
.\n{}

%%
int main(void){
    yylex;
    return0;
}
```

compiling a lex program :

```
lex count.l                    flex count.l
gcc lex.yy.c       (or)        gcc lex.yy.c -o count.exe
a.exe                          count.exe
```

compiling a yacc program :

```
yacc -dy filename.y
gcc y.tab.c
a.exe
```

```
yacc -dy filename.y                                       yacc -dy filename.y
lex filename.l                                  (or)      lex filename.l
gcc lex.yy.c y.tab.c -o filename                          gcc lex.yy.c y.tab.c
filename.exe                                              a.exe
```