1 FLEX - LEX tool

The format of a Lex file is:

```
{ %
/*definitions*/
%}

% %
analyser specification
% %

Auxiliary functions
```

Lex Definitions. The (optional) definitions section comprises macros (see below) and global declarations of types, variables and functions to be used in the actions of the lexical analyser and the auxiliary functions (if present). All such global declaration code is written in C and surrounded by %{ and %}. Code in the definitions section is simply copied as-is to the top of the generated C file.
Lex Analyser Specifications. These have the form:

```
r1        { action1 }
r2        { action2 }
...
rn        { actionn }
```

where r1 , r2 , . . . , rn are regular expressions (possibly involving macros enclosed in braces) and action1 , action2 , . . ., actionn are sequences of C statements.

Lex translates the specification into a function ***yylex()*** which, when called, causes the following to happen:

- The current input character(s) are scanned to look for a match with the regular expres- sions.
- If there is no match, the current character is printed out, and the scanning process resumes with the next character.
- If the next m characters match r_i then
  1. the matching characters are assigned to string variable ***yytext***,
  2. the integer variable ***yyleng*** is assigned the value m,
  3. the next m characters are skipped, and
  4. $action_i$ is executed.

     If the last instruction of $action_i$ is return n; (where n is an integer expression) then the call to ***yylex()*** terminates and the value of n is returned as the function's value; otherwise ***yylex()*** resumes the scanning process.
- If end-of-file is read at any stage, then the call to yylex() terminates returning the value 0.
- If there is a match against two or more regular expressions, then the expression giving the longest lexeme is chosen; if all lexemes are of the same length then the first matching expression is chosen.

Lex Auxiliary Functions. This optional section has the form:

```
fun1
fun2
...
funn
```

where each fun_i is a complete C function.

2 Predefined variables in Lex

Variable yytext is a pointer to the matched string (NULL-terminated) and yyleng is the length of the matched string. Variable yyout is the output file and defaults to stdout. Function yywrap is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a main function. In this case we simply call yylex that is the main entry- point for lex.

Name	Function
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file

Example to prepend line numbers to each line in a file.

```
/ ****Definition Section****/

%{
int yylineno;
%}

/*****Analyser Specification Of the form <pattern> {action} ***/

%%
^(.*)\n printf("%4d\t%s", ++yylineno, yytext);
%%

/ **************Auxiliary Function ****/
yywrap()
{
return 1;
}
int main(int argc, char *argv[]) {
yyin = fopen(argv[1], "r");
yylex();
fclose(yyin);
}
```

Example 2

```
%{
int abc_count, xyz_count;
%}

%%
ab[cC]    {abc_count++; }
xyz       {xyz_count++; }
\n        { return 1; }
.         { ; }

%%

main()
{
abc_count = xyz_count = 0;
yylex();
printf( "%d occurrences of abc or abC\n", abc_count );
printf( "%d occurrences of xyz\n", xyz\_count );
}
```

- This file first declares two global variables for counting the number of occurrences of abc or abC and xyz.
- Next come the regular expressions for these lexemes and actions to increment the relevant counters.
- Finally, there is a main routine to initialise the counters and call yylex().

When executed on input:

```
akhabfabcdbcaxyzXyzabChsdk
dfhslkdxyzabcabCdkkjxyzkdf
```

the lexical analyser produces:

```
4 occurrences of abc or abC
3 occurrences of xyz
```

3 Running flex program

Lex specification file called mylang.l that describes the tokens for MyLang. Perform lexical analysis using flex by the command

```
$ flex mylang.l
```

flex creates a C-source file called lex.yy.c , which can be compiled, if you link in the flex library:

```
$ gcc -o mylang lex.yy.c -ll
```

To see the output

```
$ ./mylang
```