Group Names:

Abdelrahman Hani

Amr Bekhiet

Menna Salah AbdelAziz

Basma Mahmoud

Flex (Automatic Lexical Generator)

Input Format:

The flex input file consists of three sections, separated by a line with just `%%' in it:

```
definitions
%%
rules
%%
user code
```

The definitions section contains declarations of simple name definitions to simplify the scanner specification, and declarations of start conditions, which are explained in a later section. Name definitions have the form:

```
name definition
```

The "name" is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non-white-space character following the name and continuing to the end of the line. The definition can subsequently be referred to using "{name}", which will expand to "(definition)". For example,

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

defines "DIGIT" to be a regular expression which matches a single digit, and "ID" to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to

```
{DIGIT}+"."{DIGIT}*
```

is identical to

```
([0-9])+"."([0-9])*
```

and matches one-or-more digits followed by a '.' followed by zero-or-more digits.

The rules section of the flex input contains a series of rules of the form:

```
pattern    action
```

<u>where the pattern(regex) must be unindented and the action must begin on the same line.</u>

Finally, the user code section is simply copied to `lex.yy.c' verbatim. *It is used for companion routines which call or are called by the scanner.* The presence of this section is optional; if it is missing, the second `%%' in the input file may be skipped, too.

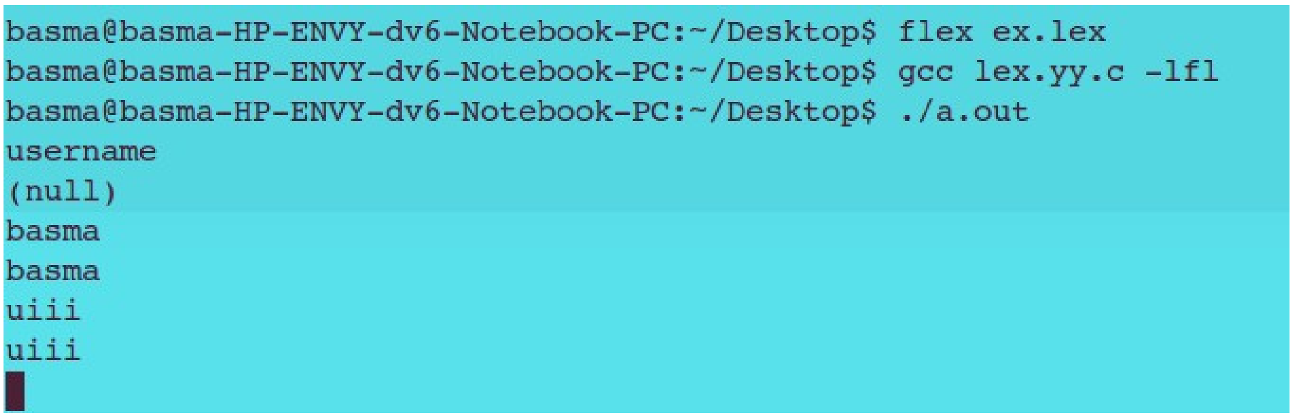
Examples:

1) First some simple examples to get the flavor of how one uses flex. The following flex input specifies a scanner which whenever it encounters the string "username" will replace it with the user's login name.

```
%%
username    printf( "%s", getlogin() );
```

By default, any text not matched by a flex scanner is copied to the output, so the net effect of this scanner is to copy its input file to its output with each occurrence of "username" expanded. In this input, there is just one rule. "username" is the pattern and the "printf" is the action. The "%%" marks the beginning of the rules.

<u>The Running of Lexical Generator for the above program is the following:</u>



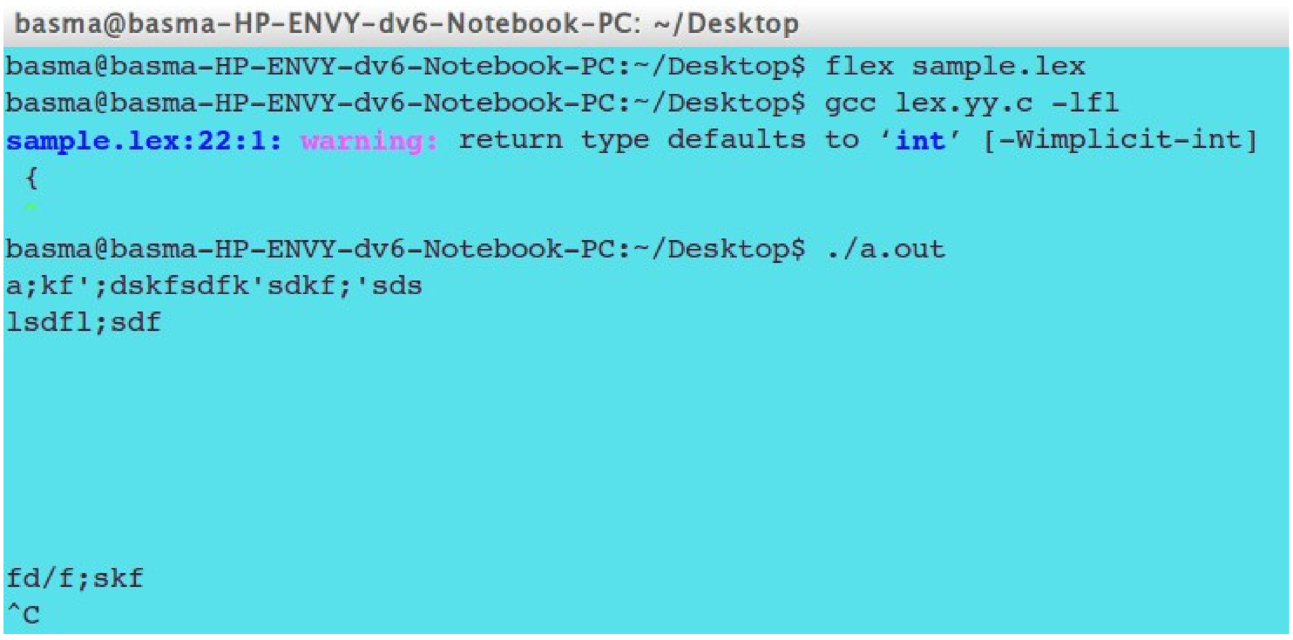
```
basma@basma-HP-ENVY-dv6-Notebook-PC:~/Desktop$ flex ex.lex
basma@basma-HP-ENVY-dv6-Notebook-PC:~/Desktop$ gcc lex.yy.c -lfl
basma@basma-HP-ENVY-dv6-Notebook-PC:~/Desktop$ ./a.out
username
(null)
basma
basma
uiii
uiii
```

2)

```
        int num_lines = 0, num_chars = 0;
%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;
%%
main()
        {
        yylex();
        printf( "# of lines = %d, # of chars = %d\n",
                num_lines, num_chars );
        }
```

This scanner counts the number of characters and the number of lines in its input *(it produces no output other than the final report on the counts).* The first line declares two globals, "num_lines" and "num_chars", which are accessible both inside `yylex()' and in the `main()' routine declared after the second "%%". There are two rules, one which matches a newline ("\n") and increments both the line count and the character count, and one which matches any character other than a newline (indicated by the "." regular expression).

The Running of Lexical Generator for the above program is the following:



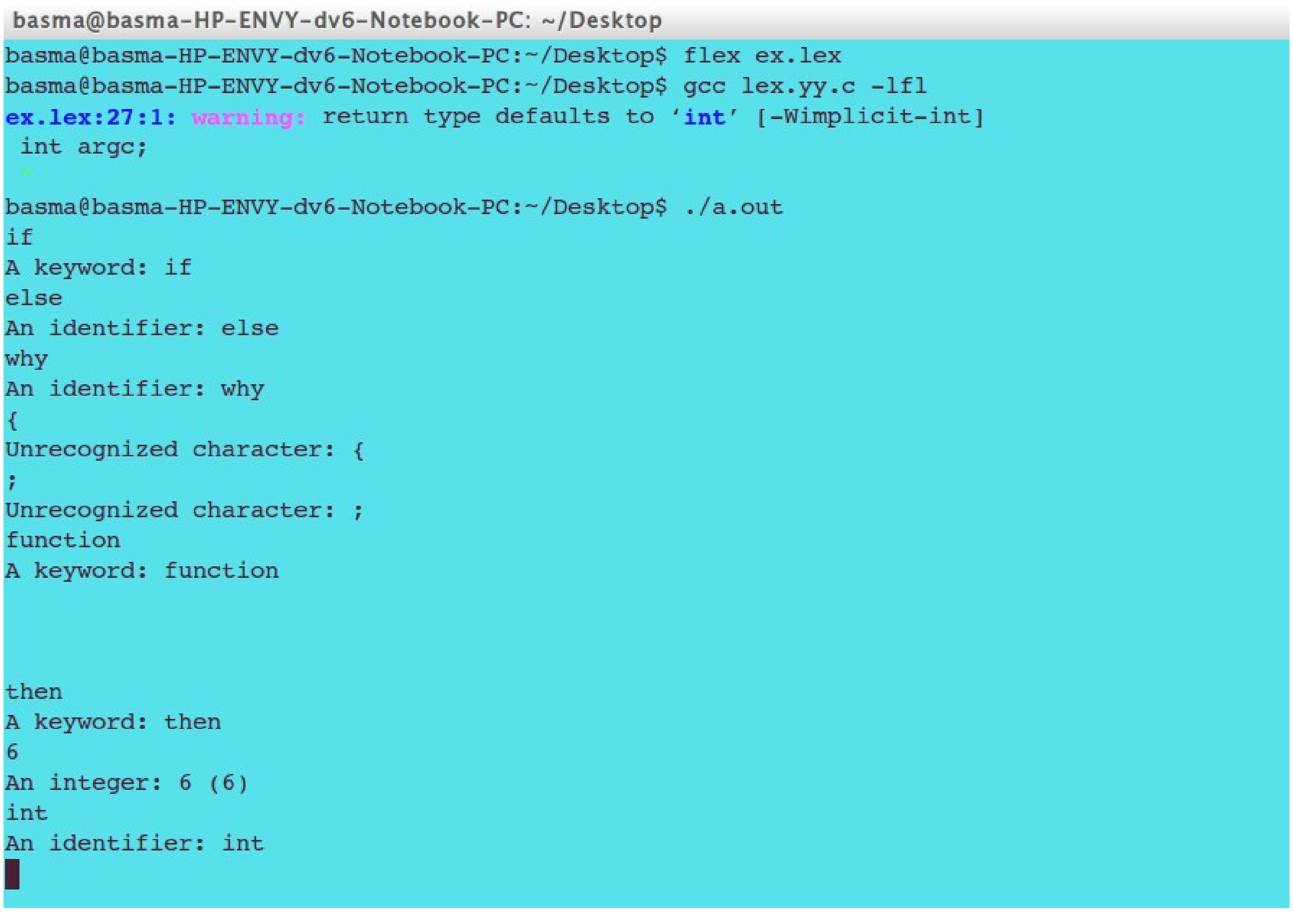
```
basma@basma-HP-ENVY-dv6-Notebook-PC: ~/Desktop
basma@basma-HP-ENVY-dv6-Notebook-PC:~/Desktop$ flex sample.lex
basma@basma-HP-ENVY-dv6-Notebook-PC:~/Desktop$ gcc lex.yy.c -lfl
sample.lex:22:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 {
 ^
basma@basma-HP-ENVY-dv6-Notebook-PC:~/Desktop$ ./a.out
a;kf';dskfsdfk'sdkf;'sds
lsdfl;sdf







fd/f;skf
^C
```

3)

```
/* scanner for a toy Pascal-like language */
%{
/* need this for the call to atof() below */
#include <math.h>
%}
DIGIT      [0-9]
ID         [a-z][a-z0-9]*
%%
{DIGIT}+ {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
 }
{DIGIT}+"."{DIGIT}*   {
            printf( "A float: %s (%g)\n", yytext,
                    atof( yytext ) );
 }
```

```
if|then|begin|end|procedure|function {
            printf( "A keyword: %s\n", yytext );
}
{ID} printf( "An identifier: %s\n", yytext );
"+"|"-"|"*"|"/"    printf( "An operator: %s\n", yytext );
"{"[∧}\n]*"}"      /* eat up one-line comments */
[ \t\n]+           /* eat up whitespace */
.           printf( "Unrecognized character: %s\n", yytext );
%%
main( argc, argv )
int argc;
char **argv;
    {
    ++argv, --argc;  /* skip over program name */
    if ( argc > 0 )
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;
    yylex();
    }
```

The Running of Lexical Generator for the above program is the following:



basma@basma-HP-ENVY-dv6-Notebook-PC: ~/Desktop

```
basma@basma-HP-ENVY-dv6-Notebook-PC:~/Desktop$ flex ex.lex
basma@basma-HP-ENVY-dv6-Notebook-PC:~/Desktop$ gcc lex.yy.c -lfl
ex.lex:27:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 int argc;
 ^
basma@basma-HP-ENVY-dv6-Notebook-PC:~/Desktop$ ./a.out
if
A keyword: if
else
An identifier: else
why
An identifier: why
{
Unrecognized character: {
;
Unrecognized character: ;
function
A keyword: function



then
A keyword: then
6
An integer: 6 (6)
int
An identifier: int
```