CS320
Lab 1
INTRODUCTION: Lex

Srinibas Swain (srinibas@iiitg.ac.in)

In this part of the Assignment, you will use the lexical analyser generator *l*ex or its variant *f*lex.

Some useful references on Lex and Yacc:

- T. Niemann, *Lex & Yacc* Tutorial, http://epaperpress.com/lexandyacc/
- Doug Brown, John Levine, and Tony Mason, *lex and yacc (2nd edn.)*, O'Reilly, 2012.
- the lex manpages.

We hope you enjoy the lab and, more generally, the unit!

1 Lex

An input file to lex is, by convention, given a name ending in .l. Such a file has three parts:

- definitions,
- rules,
- C code

These are separated by double-percent, %%. Comments begin with /* and end with */. Any comments are ignored when lex is run on the file. We will use *f*lex for constructing a lexical analyzer. flex is a fast lexical analyzer generator. flex takes user's specifications and generates a combined NFA to recognize all user defined patterns, converts it to an equivalent DFA, minimizes the automaton as much as possible, and generates C code that will implement it.

1.1 Nuts and bolts of flex

flex is designed for use with C code and generates a scanner written in C. The scanner is specified using regular expressions for patterns and C code for the actions. The specification files are traditionally identified by their .l extension. You invoke flex on a .l file and it creates lex.yy.c, a source file containing a bunch of unrecognisable C code that implements a DFA encoding all your rules and including the code for the actions you specified. The file provides an extern function yylex() that will scan one token. You compile that C file normally, link with the lex library, and you have built a scanner! The scanner reads from stdin and writes to stdout by default.

flex is open source and can be installed by

```
sudo apt-get install flex
```

To run the lexical analyzer, follow the following steps:

`lex myFile.l` creates *lex.yy.c* containing C code for scanner

`gcc -o myScan lex.yy.c -ll` compiles scanner, links with lex library

`./myScan` executes scanner, will read from *stdin*

Linking with the lex library is important. It provides a simple main function that repeatedly calls the function yylex() until it reaches *EOF*.

1.1.1 Structure of a flex file

flex input files are structured as follows:

```
%{Declarations
%}
Definitions
%%
Rules
%%
User subroutines
```

1.1.2 flex global variables

The token grabbing function yylex() takes no arguments and returns an integer. Here are some of the global variables used in flex:

- *yytext* is a nullterminated string containing the text of the lexeme which was last recognized as a token. This global variable is declared and managed in the lex.yy.c file. **Do not modify its contents.** The buffer is overwritten with each subsequent token.
- *yyleng* is an integer holding the length of the lexeme stored in yytext. This global variable is declared and managed in the lex.yy.c file.
- *yylval* is the global variable used to store attributes about the token, e.g. for an integer lexeme it might store the value, for a string literal, the pointer to its characters and so on.
- *yylloc* is the global variable that is used to store the location (line and column) of the token.

2 Example

This is the example we discussed in the class. The task is to identify the follwoing English verbs (tokens).
is am are was were go.

```
%{
/* very simple*/

%}
%%

[\t ]+         /*ignore white space*/;

is|am|are|was|were|go        {printf(”%s: is a verb \n”,yytext); }

[a–zA–Z]+   {printf(”%s: is not a verb\n”,yytext);}

.|\n     {ECHO; /* normal default anyway */ }
%%

main()
{
  yylex();
}
```

3 Exercise

- Write a lexical analyzer that counts the different types of lines that contains code, that just contain comments, or are blank.
- Extend the above analyzer to count braces, keywords etc.