CS320
Lab 2
Lex and Yacc

Srinibas Swain (srinibas@iiitg.ac.in)

In the last lab we introduced the language CHAIN and developed a lexical analyser to identify the tokens in it. I have attached a file chain.l with the assignment for your reference. Compare your lexical analyser with the one attached and observe the minute differences and how they are playing a role in parsing described in Section 2.

1 CHAIN

The language **CHAIN** consists of expressions of the following type. An expression consists of a number of terms, with # between each pair of consecutive terms, where each term is either a string of lower-case letters or an application of the Reverse function to such a string. Examples of such expressions include

```
mala # y # Reverse(mala)
block # drive # cut # pull # hook # sweep # Reverse(sweep)
Reverse(side) # Reverse(direction) # Reverse(gear)
```

For lexical analysis, we wish to treat every lower-case alphabetical string as a lexeme for the token STRING, and the word `Reverse` as a lexeme for the token `REVERSE`. We focus mainly on the Rules section, in the middle of the file. It consists of a series of statements of the form

*p*attern { action }

where the pattern is a regular expression and the action consists of instructions, written in C, specifying what to do with text that matches the pattern. In your file, each pattern represents a set of possible lexemes which you wish to identify. These are:

- a string of lower-case letters;
  - This is taken to be an instance of the token STRING (i.e., a lexeme for that token).
- the specific string `Reverse`;
  - Such a string is taken to be an instance of the token REVERSE.
- certain specific characters: #, (,);
- white space, being any sequence of spaces and tabs;
- the newline character.

Note that all matching is case-sensitive.

Our action is, in most cases, to print a message saying what token and lexeme have been found. For white space, we take no action at all. A character that cannot be matched by any pattern yields an error message.

If you run lex on the file chain.l, then lex generates the C program lex.yy.c. This is the source code for the lexical analyser. You compile it using a C compiler such as cc.

```
$ flex chain.l
$ cc lex.yy.c
```

When you run the program, it will initially wait for you to input a line of text to analyse. Do so, pressing Return at the end of the line. Then the lexical analyser will print, to standard output, messages showing how it has analysed your input. The printing of these messages is done by the `printf` statements from the file `chain.l`. Note how it skips over white space, and only reports on the lexemes and tokens.

```
$ ./a.out
mala
# y #Reverse( mala)
Token: STRING; Lexeme: mala
Token and Lexeme: #
Token: STRING; Lexeme: y
Token and Lexeme: #
Token: REVERSE; Lexeme: Reverse
Token and Lexeme: (
Token: STRING; Lexeme: mala
Token and Lexeme: )
Token and Lexeme: <newline>
```

2 yacc

We now turn to parsing, using yacc. You can install yacc by the following command:

```
sudo apt-get install bison
```

Consider the following grammar for CHAIN.

$$S \longrightarrow E, \quad S \longrightarrow \epsilon, \quad E \longrightarrow E\#E, \quad E \longrightarrow STRING, \quad E \longrightarrow REVERSE(STRING)$$

In this grammar, the non-terminals are S and E. Treat STRING and REVERSE as just single tokens, and hence single terminal symbols in this grammar. We now generate a parser for this grammar, which will also evaluate the expressions, with # interpreted as concatenation and Reverse(...) interpreted as reversing a string. To generate this parser, you need two files, chain.l (for lex) and chain.y (for yacc):

- In the Declaration part of chain.l (attached file), observe the statement #include ”y.tab.h”;
- in the Rules section (of chain.l), in each action:
  The statements of the form
  - yylval.str = ...;
  - return TOKENNAME;
  - return *yytext;
- chain.y, the input file for yacc, is given to you with some blanks that needs to be filled by you.

  An input file for yacc is, by convention, given a name ending in .y, and has three parts, very loosely analogous to the three parts of a lex file but very different in their details and functionality:
  Declarations,
  Rules,
  Programs.

  These are separated by double-percent, %%. Comments begin with /* and end with */. Peruse the provided file chain.y, identify its main components, and pay particular attention to the following, since you will need to modify some of them.

- In the Declarations section: – lines like
  char *reverse(char *); which are declarations of functions (but they are defined later, in the Programs section);
- declarations of the tokens to be used:
  %token <str> STRING
  %token<str> REVERSE
- declarations of the nonterminal symbols to be used (which don't need to start with an upper-case letter):
  %type <str> start
  %type <str> expr
- nomination of which nonterminal is the Start symbol:
  %start start
- in the Rules section, a list of grammar rules in BNF, except that the colon ":" is used instead of →, and there must be a semicolon at the end of each rule. Rules with a common left-handside may be written in the usual compact form, by listing their right-hand-sides separated by vertical bars, and one semicolon at the very end. The terminals may be token names, in which case they must be declared in the Declarations section and also used in the lex file, or single characters enclosed in forward-quote symbols. Each rule has an action, enclosed in braces {...}. A rule for a Start symbol may print output, but most other rules will have an action of the form $$ = ... The special variable $$ represents the value to be returned for that rule, and in effect specifies how that rule is to be interpreted for evaluating the expression. The variables $1, $2, ... refer to the values of the first, second, ... symbols in the right-hand side of the rule.
- in the Programs section, various functions, written in C, that your parsers will be able to use.

After constructing the yacc file, the parser can be generated by:

```
$ yacc -d chain.y
$ flex prob1.l
$ cc lex.yy.c y.tab.c
```

The executable program, which is now a parser for CHAIN, is again named a.out by default, and will replace any other program of that name that happened to be sitting in the same directory. The expected output is:

```
$ ./a.out
mala # y #Reverse( mala)
malayalam
```