Computer Systems Lab - II
Asignment 1

Compiler writting using Lex and Yacc

Poushali Chakraborty (2222208)
M-Tech Computer Science

Department of Computer Science
NIT Silchar
DATE SUBMITTED
[February 13, 2023]

Chapter 1

Problem 1

Some words can be more than one part of speech, e.g., “watch”, “fly”, “time”, or “bear”. How could you handle them? Try adding a new word and token type NOUN-OR-VERB, and add it as an alternative to the rules for subject, verb, and object. How well does this work?

Solution: This problem is mentioned in John Levine (1992). To solve this problem first we have to write a lexical analyzer that can recognize words and parts of speech. We have to do:

- Writing a Words and Parts of Speech recognizer Lex program including token types : NOUN, VERB, ADJECTIVE, ADVERB, PRONOUN, PREPOSITION, CONJUNCTION and NOUN-OR-VERB.
- Writing a Yacc parser that can verify a simple english statement including the given token types in the lexer.

1.1 Lex Program

We are going to write a lex program which can recognize the given words token types. To allow dynamic declaration parts of speech while the program is running, we are going to build a table of words (symbol table) so we can add new words without modifying and recompiling the lex program. Declaration lines start with the name of a part of speech followed by the words to declare. We are are going to use separate lex patterns for the names of parts of speech. We also have to add symbol table maintenance routines, in this case add-word(), which puts a new word into the symbol table, and lookup-word( ), which looks up a word which should already be entered. In the program's code, we declare a variable state that keeps track of whether we're looking up words, state LOOKUP, or declaring them, in which case state remembers what kind of words we're declaring. Whenever we see a line starting with the name of a part of speech, we set the state to declare that kind of word; each time we see a newline character we switch back to the normal lookup state. For declaring words, the first group of rules sets the state to the type corresponding to the part of speech being declared. (The caret,ˆ, at the beginning of the pattern makes the pattern match only at the beginning of an input line.)

We reset the state to LOOKUP at the beginning of each line so that after we add new words interactively we can test our table of words to determine if it is working correctly. If the state is LOOKUP when the pattern $[a-zA-Z]+$ matches, we look up the word, using lookup-word(), and if found print out its type. If we're in any other state, we define the word with add-word(). **word-list** is the link-list having each node with **word-name**, **word-type** and pointer to the **next** node.

Algorithm 1 Adding word into word-list

```
1: procedure add−word(word, type−of−word)
2:     if lookup−word(word) == null then          ▷ If word is not defined
3:         create a new−node for word−list
4:         new−node.word−name = word
5:         new−node.word−type = type−of−word
6:         insert it in the beginning of word−list
```

Algorithm 2 lookup-word : Searching word in word-list

```
1: procedure lookup−word(word)
2:     current = word−list
3:     while current! = null do
4:         if current.word−name == word then
5:             return current.word−type
6:         else
7:             current point to next node in word−list
```

Algorithm 3 Lexical Analyzer

```
1: definition section:                                   ▷ Lex copies directly to the generated C file
2: including y.tab.h                                     ▷ token codes from parser
3: LOOK−UP = 0
4: initialize state
5: end of definition section
6: rule section:
7: if verb is at beginning of the input line then
8:     state = verb
9: else if noun is at beginning of the input line then
10:    state = noun
11: else if noun-or-verb is at beginning of the input line then
12:    state = noun-or-verb
13: else if condition then
14:    Do the same for the rest of parts of speech
15: else if [a−zA−Z]+ pattern matches then
16:    if state! = LOOK− then
17:        add−word(current−word, state)                 ▷ add the current word into word-list
18:    else
19:        return look−up(current−word)                  ▷ return the type of the word from the word-list
```

1.2 Parser

To recognize common sentences now we can use yacc grammar and parser. Parser will be our high level routine. It will call lexer *yylex()* whenever it needs a token from input. The lexer then scans through the input recognizing tokens. As soon as it finds a token of interest to the parser, it returns to the parser, returning the token's code as the value of *yylex()*.

Yacc can write a C header file containing all of the token definitions. We include this file, called y.tab.h on UNIX systems in the lexer and use the preprocessor symbols in our lexer code.

Algorithm 4 Parser

1: definition section:
2: defining all the tokens expected to receive from lexical analyzer,
3: NOUN, VERB, ADJECTIVE, ADVERB, PRONOUN, PREPOSITION, CONJUNCTION and NOUN-OR-VERB.
4: end of definition section
5: rule section:
6: sentence → subject verb object
7: subject → noun | pronoun | noun-or-verb
8: object → noun | noun-or-verb
9: end of rule section
10: *yyin* = lexers input file
11: calling *main()*:
12: **while** *yyin* is not ended **do**
13: *yyparse()* ▷ The routine *yyparse()* is the parser generated by yacc, so our main program repeatedly tries to parse sentences until the input runs out.

1.3 Result

When we run the program first we have to declare the words for each parts of speech. After that when we write a sentence it can identify each word with it's declared parts of speech.

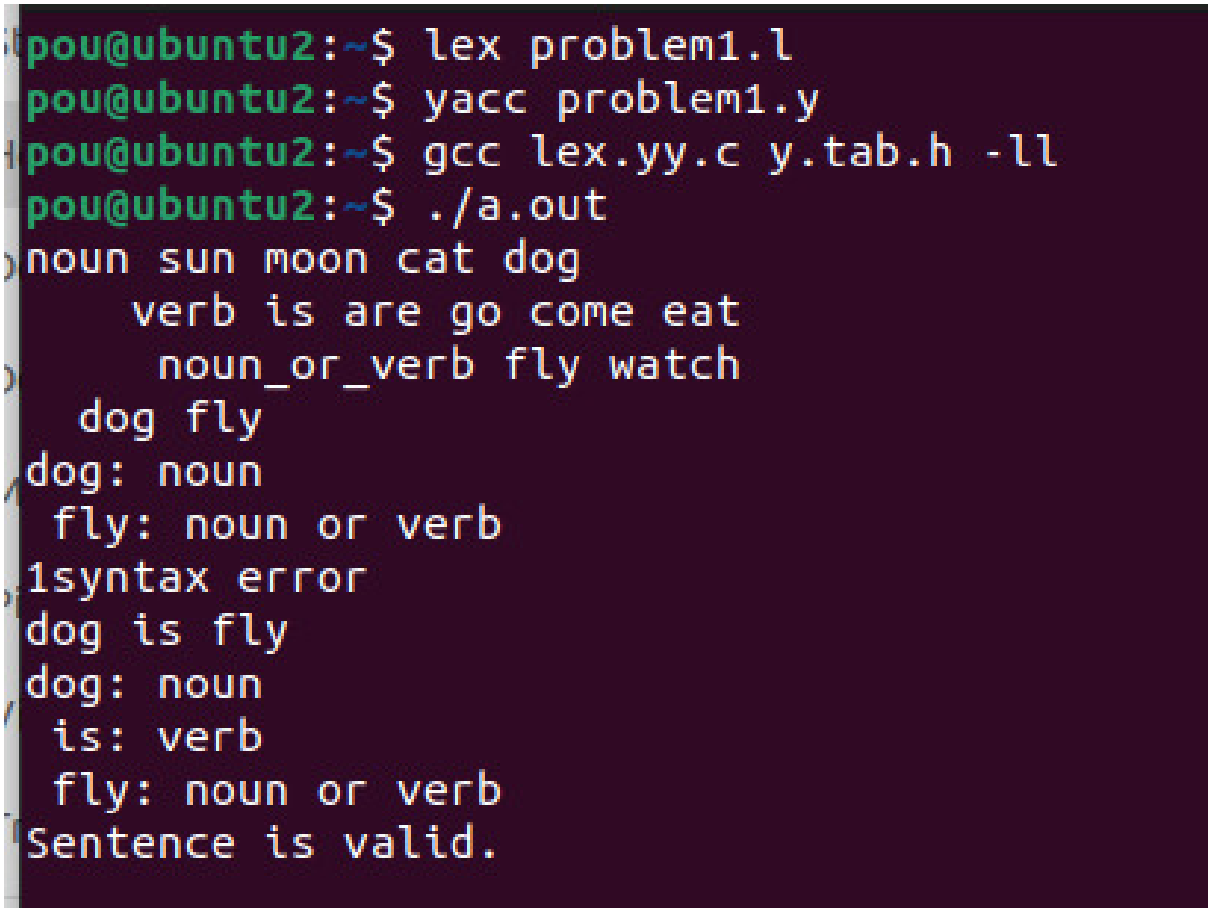

Figure 1.1: Output of problem 1

Chapter 2

Problem 2

Make the word count program smarter about what a word is, distinguishing real words which are strings of letters (perhaps with a hyphen or apostrophe) from blocks of punctuation.

Solution: This problem is mentioned in John Levine (1992). To solve this problem first we have to write a lexical analyzer that can recognize words from any strings and also can count the words. We have to provide the definition of word token which can include any alphabets, hyphen or apostrophe. We have to define a integer called wordcount, which is going to behave like a counter for the words. In rule section for each matches of a word, the counter is going to increment as defined by the action. In a pattern, lex replaces the name inside the braces with substitution, the actual regular expression in the definition section. Our example increments the number of words after the lexer has recognized a complete word. Our main function will the lexer's entry point *yylex()* and then calls *printf()* to print the results of this run. Note that our code doesn't do anything fancy; it doesn't accept commandline arguments, doesn't open any files, but uses the lex default to read from the standard input.

Algorithm 5 Lexical Analyzer

```
definition section:                                   ▷ Lex copies directly to the generated C file
initialize word−count
define pattern for token word
                    ▷ starts with a space or newline and can include any letter,a hyphen or apostrophe
end of definition section
rule section:
if word then
    word−count = word−count + 1
else
    ignore
calling main():
yylex()
print output word−count    ▷ calls the lexer's entry point yylex() and than print the results of this run.
```

2.1 Result

In input we have to write some sentences line by line. When we stop the execution it will return the number of words along with character and line count.

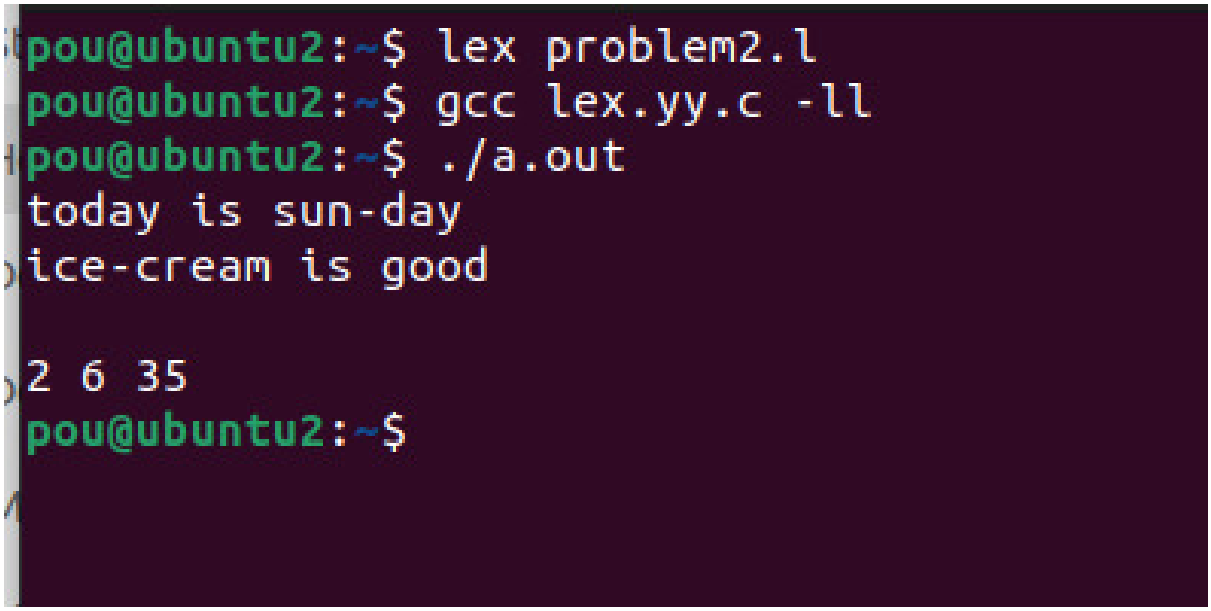

Figure 2.1: Output of problem 2

Chapter 3

Problem 3

Is lex really as fast as we say? Race it against egrep, awk, sed, or other pattern matching programs you have. Write a lex specification that looks for lines containing some string and prints the lines out. (For a fair comparison, be sure to print the whole line.) Compare the time it takes to scan a set of files to that taken by the other programs. If you have more than one version of lex, do they run at noticeably different speeds?

Solution: This problem is mentioned in John Levine (1992). To solve this problem

- We have to create some text files
- Write a lex program which can search this files for some given strings in the input.
- calculate the time it takes to search
- compare it with awk and egrep for the same

Algorithm 6 Find if a string is there in the given file lists

```
1: procedure check(str)
2:     flag = 0
3:     t= time()
4:     initialize filelist[]
5:     for each file in filelist[] do
6:         open file
7:         while each line in file do
8:             if str contains in line then
9:                 print Found
10:                print line
11:                flag =1
12:        if flag == 0 then
13:            print Not Found
14:    t = time() - t
15:    print t time taken for execution
```

Algorithm 7 Lexical Analyzer

```
1: rule section:
2: if [a−zA−Z]+ matches then          ▷ rule matches with pattern including any letters
3:     check(current−word)
4: else
5:     ignore
6: calling main():
7: yylex()
8: print output word−count    ▷ calls the lexer's entry point yylex() and than print the results of this run.
```

Algorithm 8 Compare Script

```
1: echo "Enter the string to search"
2: read str
3: echo "Egrep search"
4: start = time
5: eval egrep strfile1.txtfe2il.txt
6: end = time
7: echo time to execute is end − start ns
8: echo "Awk search"
9: start = time
10: eval awk strfile1.txtfe2il.txt
11: end = time
12: echo time to execute is end − start ns
13: echo "lex search "
14: eval lexprogram file1.txtfile.txt
```

3.1 Result

We have created a script called compare in which we are going to run egrep, awk and our lexer search the given word in each file in the list and print whether it is found or not in each file, along with the line where it was found. And also going to see how much time each takes.

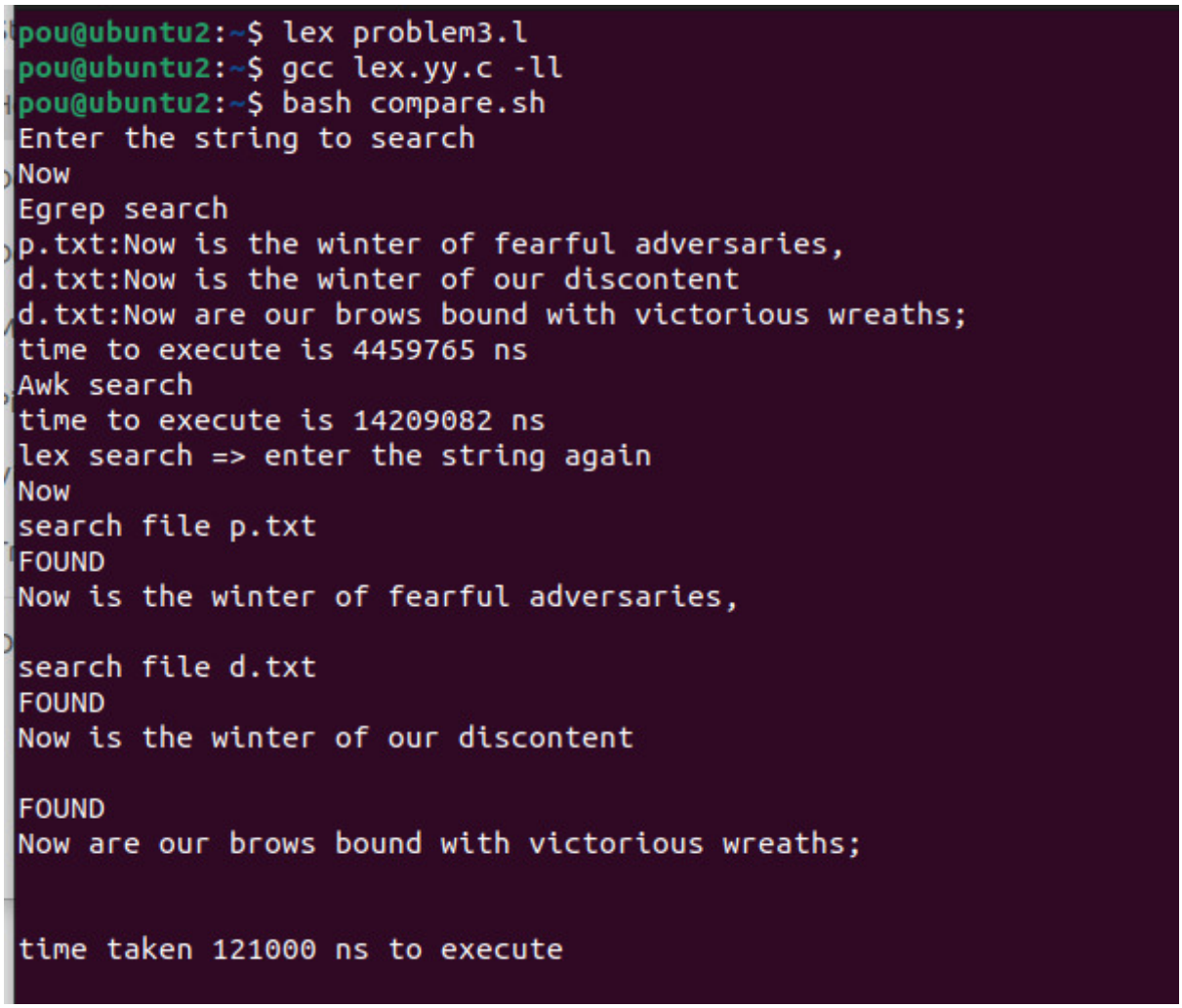

Figure 3.1: Output of problem 3

Chapter 4

Problem 4

Add a string data type, so you can assign strings to variables and use them in expressions or function calls. Add a STRING token for quoted literal strings. Change the value of an expression to a structure containing a tag for the type of value along with the value. Alternatively, extend the grammar with a stringexp non-terminal for a string expression with a string (char *) value.

Solution: This problem is mentioned in John Levine (1992). One approach to solve this problem

- Creating a program which can evaluate numerical expression like 100 * 10
- using variables instead of numerical constant values
- assigning String values to variables
- evaluating String expression including variables

Whenever the lexer returns a token to the parser, if the token has an associated value, the lexer must store the value in yylval before returning. yacc defines yylval as a union and puts the definition in y.tab.h. First we need to write a lexical analyzer for identifying the tokens : NUMBER (any number), NAME (single alphabet lower letter variale name for numbers), SNAME (single alphabet capital letter variable name for strings) , STR (string literals)

Algorithm 9 Lexical Analyzer

```
1: definition section:                                   ▷ Lex copies directly to the generated C file
2: including y.tab.h                                     ▷ token codes from parser
3: including libraries : math, string, stdio
4: end of definition section
5: rule section:
6: if token matches for number  then
7:     yyval.dval = number-of(current-word)
8:     return NUMBER
9: else if  token match for single character a-z  then
10:    yyval.valno = ascii value of character
11:    return NAME
12: else if  token match for single character A-Z  then
13:    yyval.valno = ascii value of character
14:    return SNAME
15: else if condition then
16:    Do the same for the rest of parts of speech
17: else if  [][a−zA−Z0−9]+[] pattern matches then
18:    remove extra symbols from current word
19:    yyval.strval = current word
20:    return STR
```

4.1 Parser

To recognize variables and assigning values to them, we can use yacc grammar and parser. We are going to use arrays as a structure to store the variables along with their values. For now variable names can only be of one character. Parser will be our high level routine. It will call lexer *yylex()* whenever it needs a token from input. The lexer then scans through the input recognizing tokens. As soon as it finds a token of interest to the parser, it returns to the parser, returning the token's code as the value of *yylex()*.

Yacc can write a C header file containing all of the token definitions. We include this file, called y.tab.h on UNIX systems in the lexer and use the preprocessor symbols in our lexer code.

Algorithm 10 Parser

definition section:
define *vbltable[ ]* as an array to store numbers with respect to given variable name
define *strvartable[ ]* as an array to store strings with respect to given variable name
define symbol types in union *dval, vblno, strval*
defining all the tokens expected to receive from lexical analyzer,
NAME , SNAME, NUMBER, STR
define precedence of operators
define type as expression
end of definition section
rule section:
statement-list → statemet *newline* | statement-list statemet *newline*
if statement → NAME = expression **then**
    place the value of the expression in *vbltable[NAME]*
else if statement → expression **then**
    print the result of expression
else if statement → SNAME = STR **then**
    place string STR in *strvartable[SNAME]*
else if statement → SNAME **then**
    print the value of *strvartable[SNAME]*
else if statement → SNAME + SNAME **then**
    concate two strings in to first
rules for expression valuation ....
end of rule section
yyin = lexers input file
calling *main()*:
while *yyin* is not ended **do**
    yyparse() ▷ The routine *yyparse()* is the parser generated by yacc, so our main program repeatedly tries to parse sentences until the input runs out.

4.2 Result

The code is a simple parser for a calculator program. The calculator supports basic arithmetic operations such as addition, subtraction, multiplication and division, and can store and retrieve both numerical and string values in variables.

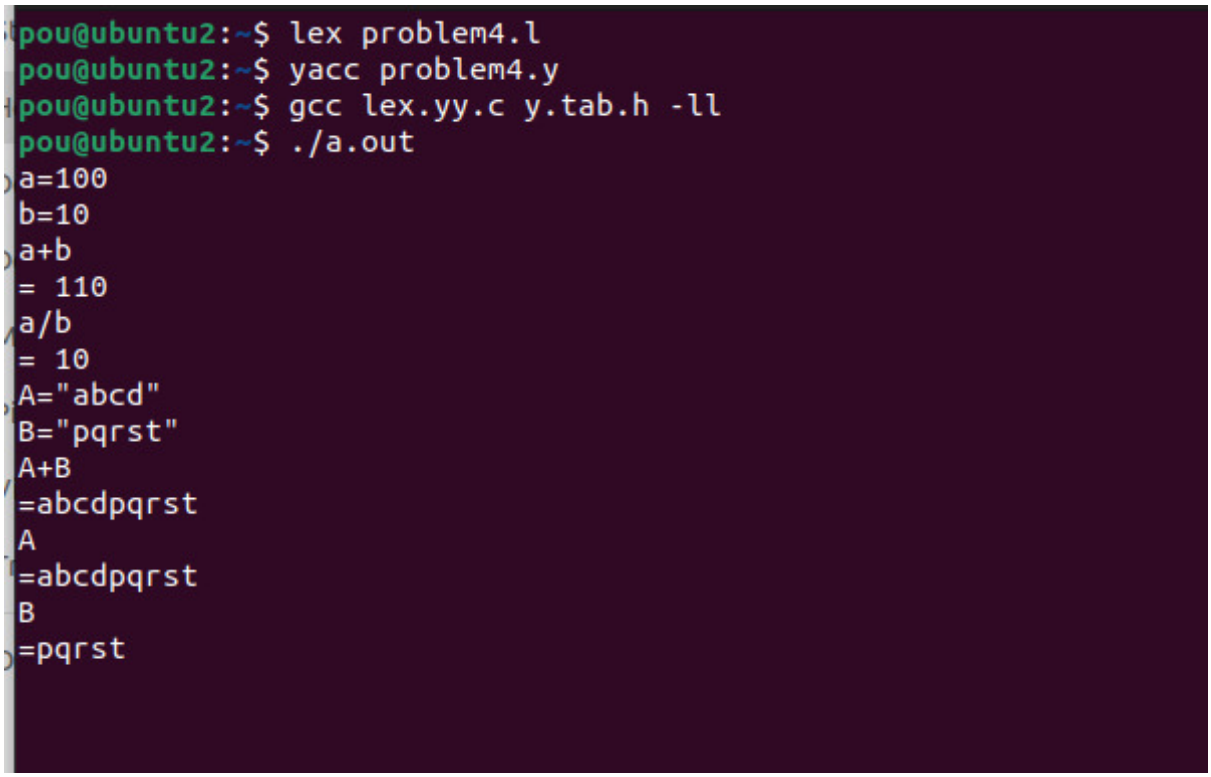

Figure 4.1: Output of problem 4

Bibliography

John Levine, Tony Mason, Doug Brown. 1992. *lex and yacc, 2nd edition.* O'Reilly Media, Inc.