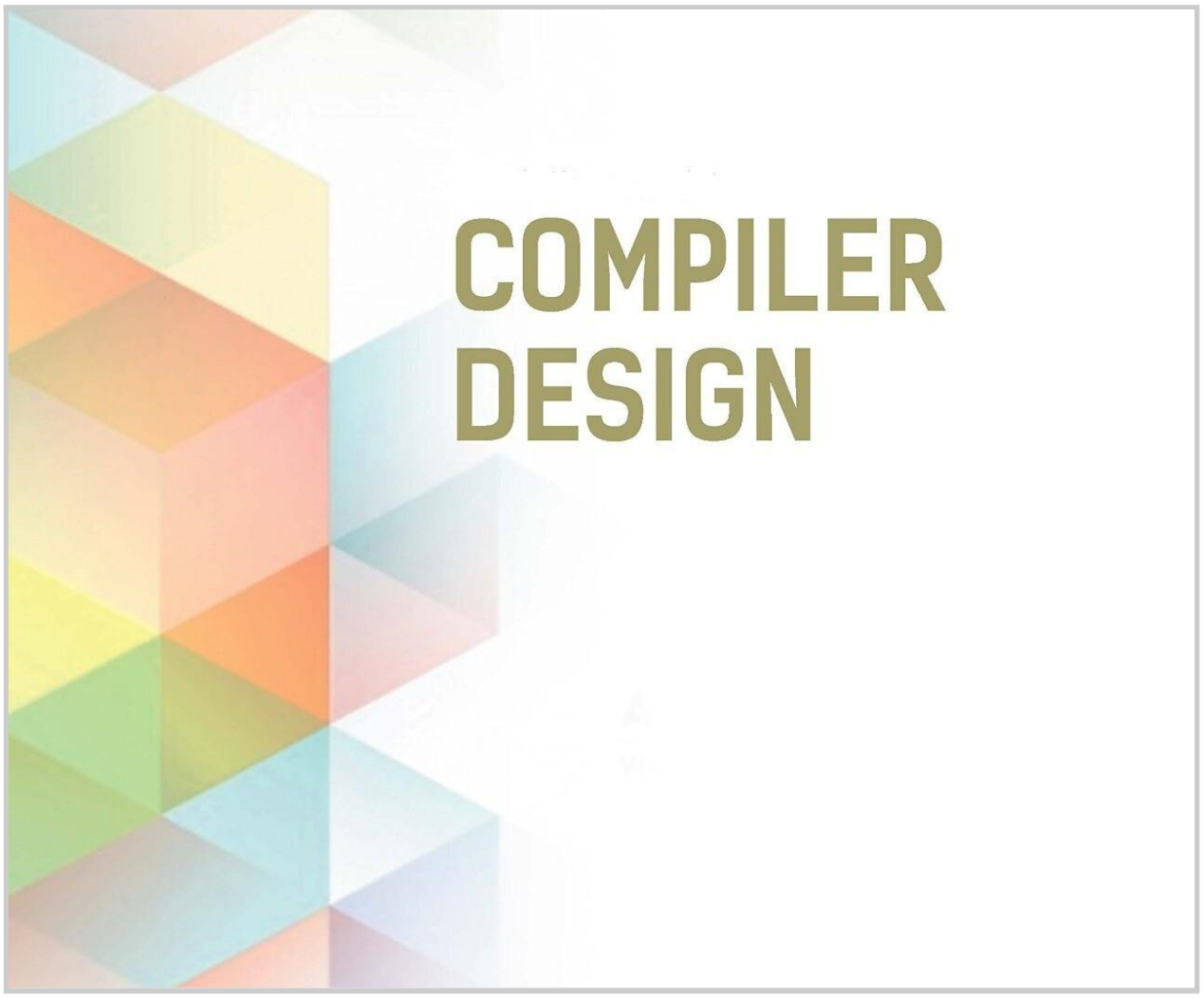

Compiler Design Lab

ASSIGNMENT - 2

15.02.2022

—

Atanu Ghosh

Roll: **001910501005**
Section: A-1
BCSE-III (2019-2023) 6th sem

The following codes have been implemented and tested with GNU Flex 2.6.4, GNU Bison 3.5.1 and GCC 9.3.0 on Linux Operating System (Distro Choice: Ubuntu 20.04 LTS).

Compiling and Executing a Yacc File

I have written a shell script named **run.sh** that compiles an yacc file to generate the **filename.tab.h** and **filename.tab.c** files, compiles a lex file to generate the **lex.yy.c** file, then compiles the C program file to generate binary executable having an extension of .exe (**res.exe)** and finally executes the program and deletes all files except the original lex (**filename.l**) and yacc (**filename.y**) files.

The shellscript contains the following :

```
#!/usr/bin/bash

bison -d filename.y

flex filename.l

gcc lex.yy.c filename.tab.c -o ./res.exe

./res.exe

rm -rf lex.yy.c filename.tab.c filename.tab.h res.exe
```

We have to set executable permission for **run.sh** by running `chmod +x run.sh` on the terminal**.** Then just type `./run.sh` and finally press **Enter** to display the input prompt.

Question-1

Learn how to use YACC (several tutorials are available on the Internet.)
Design a grammar to recognise a string of the form AA...ABB...B, i.e. any number of As followed by any number of Bs. Use LEX or YACC to recognise it. Which one is a better option? Change your grammar to recognise strings with equal numbers of As and Bs - now which one is better?

Code Snippet

Filename: a2q1.l

```
%{
      #include <stdio.h>
%}

%%
      \n                    { return 1; }
      [a]+[b]+\n            { return 0; }
      exit                  { exit(0); }
      .                     ;
%%

int yywrap() {
      return 0;
}

int main(){
      int tmp;
      while(1) {
            printf("\nEnter an expression : ");
            tmp = yylex();
if(!tmp) printf("Expression is Accepted! It is of form
a^nb^m.\n");
else printf("Expression is Rejected! It is NOT of form
a^nb^m.\n");
      }
      return 0;
}
```

I/O - Screenshot

```
a2q1
❯ flex a2q1.l && gcc lex.yy.c -o res.exe && ./res.exe

Enter an expression : aaabbb
Expression is Accepted! It is of form a^nb^m.

Enter an expression : aaaabb
Expression is Accepted! It is of form a^nb^m.

Enter an expression : aabbbb
Expression is Accepted! It is of form a^nb^m.

Enter an expression : aaa
Expression is Rejected! It is NOT of form a^nb^m.

Enter an expression : bbb
Expression is Rejected! It is NOT of form a^nb^m.

Enter an expression : abcdefg
Expression is Rejected! It is NOT of form a^nb^m.
```

Code Snippet

Filename: a2q1a.y

```
%{
#include<stdio.h>
#include<stdlib.h>
int yyerror (char *s);
int yylex();
%}

%token a b

%%
s     : as bs ;
as    : a | as a ;
bs    : b | bs b ;
%%

int yywrap() {
      return 1;
}

int yyerror(char *s) {
      printf("Expression is Rejected! It is NOT of form a^nb^m.\n");
      exit(0);
}

int main() {
      printf("\nEnter an expression : ");
      yyparse();  // reads a token value pair from yylex()
      printf("Expression is Accepted! It is of form a^nb^m.\n");
      return 0;
}
```

I/O - Screenshot

```
a2q1a
❯ ./run1.sh

Enter an expression : aaabbb
Expression is Accepted! It is of form a^nb^m.

a2q1a
❯ ./run1.sh

Enter an expression : aaaaabb
Expression is Accepted! It is of form a^nb^m.

a2q1a
❯ ./run1.sh

Enter an expression : aabbbbb
Expression is Accepted! It is of form a^nb^m.

a2q1a
❯ aaaaaaa
zsh: command not found: aaaaaaa

a2q1a
❯ ./run1.sh

Enter an expression : aaaaaa
Expression is Rejected! It is NOT of form a^nb^m.
```

- ## Conclusion:

The LEX code was much easier to write and understand that the YACC version of The same code. So, I think in this case LEX was a better option than YACC.

Code Snippet

Filename: a2q1b.y

```
%{
      #include<stdio.h>
      #include<stdlib.h>
      int yyerror (char *s);
      int yylex();
%}

%token a b

%%
      s : a s b | a b ;
%%


int yywrap() {
      return 1;
}

int yyerror(char *s) {
      printf("Expression is Rejected! It is NOT of form a^nb^n.\n");
      exit(0);
}

int main() {
      printf("\nEnter an expression : ");
      yyparse();  // reads a token value pair from yylex()
      printf("Expression is Accepted! It is of form a^nb^n.\n");
      return 0;
}
```

I/O - Screenshot

```
a2q1b
❯ ./run2.sh

Enter an expression : aaabbb
Expression is Accepted! It is of form a^nb^n.

a2q1b
❯ ./run2.sh

Enter an expression : aaaaaabb
Expression is Rejected! It is NOT of form a^nb^n.

a2q1b
❯ ./run2.sh

Enter an expression : aabbbbbbbbbb
Expression is Rejected! It is NOT of form a^nb^n.

a2q1b
❯ ./run2.sh

Enter an expression : aaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb
Expression is Rejected! It is NOT of form a^nb^n.

a2q1b
❯ ./run2.sh

Enter an expression : aaaabbbbbbbbbccccccccddddddddddeeee
Expression is Rejected! It is NOT of form a^nb^n.
```

● Conclusion:

To check strings of type ‘A^nB^m ’ where n is not equal to m then both of YACC and LEX is almost the same as per convenience is concerned.

But in case ‘A^nB^n ’ then YACC is far easier than LEX. In LEX we have to use different states by using the concept of DFA, along with a counter, since only DFA cannot determine the given string. In YACC however, we got to use CFG, which solves the issue.

Question-2

Write the lex file and the yacc grammar for an expression calculator. You need to deal with i) binary operators '+', '*', '-'; ii) unary operator '-'; iii) boolean operators '&', '|' iv) Expressions will contain both integers and floating point numbers (up to 2 decimal places). Consider left associativity and operator precedence by order of specification in yacc.

Code Snippet

Filename: a2q1b.y

```
%{
#include<stdio.h>
#include<stdlib.h>
int yylex(void);
int yyerror(char *);
%}

/* BISON Declarations */
%token  id
%left '+' '-'    /* Arithmatic Operators */
%left '*' '/'    /* Arithmatic Operators */
%left '(' ')'    /* Braces */
%left '|' '&'    /* Boolean Operators */
%left '!'        /* Negation */
%left neg        /* Unary minus */


/* Grammar Follows */
%%
E1   : E { printf("Value of the above expression is = %d\n", $1);}

E    : E '+' E { $$ = $1 + $3;}
     | E '-' E { $$ = $1 - $3;}
     | E '*' E { $$ = $1 * $3;}
     | E '/' E { $$ = $1 / $3;}
     | E '|' E { $$ = $1 | $3;}
     | E '&' E { $$ = $1 & $3;}
     | '!' E   { $$ = !$2;}
     | '-' E %prec neg{ $$ = -1 * $2;}
     | '(' E ')' { $$ = $2;}
     | id { $$ = $1;}
     ;
%%
```

```
int main(){
      printf("\nEnter an expression : ");
      yyparse();
      return 0;
}

int yywrap(){
      return 0;
}

int yyerror( char* s){
      printf("Invalid expression! Try Again.\n");
      exit(0);
      return 0;
}
```

I/O - Screenshot

```
a2q2
❯ ./run.sh

Enter an expression : 7+8-5+2*10-280/28
Value of the above expression is = 20

a2q2
❯ ./run.sh

Enter an expression : -100
Value of the above expression is = -100

a2q2
❯ ./run.sh

Enter an expression : 1 & 1
Value of the above expression is = 1

a2q2
❯ ./run.sh

Enter an expression : 1 & 0
Value of the above expression is = 0

a2q2
❯ ./run.sh

Enter an expression : 0 & 0
Value of the above expression is = 0

a2q2
❯ ./run.sh

Enter an expression : 1 | 1
Value of the above expression is = 1

a2q2
❯ ./run.sh

Enter an expression : 0 | 1
Value of the above expression is = 1
```