

CS4158 – PROGRAMMING LANGUAGE TECHNOLOGY 2024/25

LAB SETUP, COMMANDS AND INPUT/OUTPUT EXAMPLES

Tools:

1. GCC
2. FLEX / LEX
3. BISON / YACC

Installation:

- Windows
 - Using the chocolatey package manager:
 - <https://community.chocolatey.org/packages/mingw>
 - <https://community.chocolatey.org/packages/winflexbison3>
- Linux / macOS
 - <package-manager> install gcc flex bison
 - Replace <package-manager> with the appropriate package manager based on your system:
 - apt-get or apt for Ubuntu/Debian-based Linux.
 - brew for macOS (Homebrew).
 - dnf or yum for Fedora/CentOS/RHEL Linux.

Commands:

- Windows (Powershell)
 - gcc pgm.c ; .\a.exe // To simulate C programs
 - win_flex lex.l ; gcc lex.yy.c ; .\a.exe // To simulate LEX programs
 - win_bison -d y.y ; win_flex lex.l ; gcc y.tab.c lex.yy.c ; .\a.exe // YACC programs
- Linux / macOS
 - gcc pgm.c && ./a.out // To simulate C programs
 - flex lex.l && gcc lex.yy.c && ./a.out // To simulate LEX programs
 - bison -d y.y && flex lex.l && gcc y.tab.c lex.yy.c && ./a.out // YACC programs
- **Recommended to run commands individually:** While using && or ; to chain commands is convenient, it's generally better to run each command individually. This way, you can see the specific error for each step and debug it more effectively instead of having multiple errors hidden by the next command.
- **Flex and Bison can be replaced with lex and yacc (NOT WINDOWS):** Lex and YACC are older versions of the tools and are typically available on both Linux and macOS. However, Flex and Bison are modern, more feature-rich replacements and are recommended for complex projects or for better compatibility across platforms.
- **Not tested on macOS:** The above commands were not tested on macOS.

Additional Resources:

[YouTube Playlist](#) (Recommended)

[Documents](#)

[YouTube Playlist](#)

Instructions:

All programs should be written in C. The programs where flex and bison are used will be marked.

Write a report which explains what you have done in each task and submit it in PDF, **due on April 27, 2024**. Please submit the .l, .y, and .c files for the tasks where any or all of them are applicable. Include the screenshots of the successful output of your tasks in the final report.

EXPERIMENTS

1. Design and implement a lexical analyser for given language using C and the lexical analyser should ignore redundant spaces, tabs and new lines.

Sample Input:

```
int a,b,c;  
// comment to ignore  
a=b+c;
```

Sample Output:

```
Constants:  []  
Keyword:   ['int']  
Identifier: ['a' 'b' 'c' 'a' 'b' 'c']  
Operators: ['=' '+' ]  
Symbols:   [',' ';' ';']
```

2. Implementation of Lexical Analyzer using Lex Tool. (Use LEX)

Sample Input:

```
void main(){  
    // hello world  
    int a = 7, b = 7.35;  
}
```

Sample Output:

```
void  Reserved Keyword
```

```

main  Reserved Keyword
(    Special Operator
)    Special Operator
{    Special Operator
int  Reserved Keyword
a    Identifier
=    Arithmetic Operator
7    Constant
,    Special Operator
b    Identifier
=    Arithmetic Operator
7.35 Floating Number
;    Special Operator
}    Special Operator

```

1 Comments Ignored

3. Generate YACC specification for a few syntactic categories. (Use LEX and YACC)
 - a. Program to recognize a valid arithmetic expression that uses operator +, −, * and /.

Sample Inputs:

Enter arithmetic expression: 6+7

Enter arithmetic expression: 6\$7

Sample Outputs:

Valid Arithmetic Expression

Error. Failed to parse.

- b. Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

Sample Inputs:

Enter identifier: abc123

Enter identifier: 123

Sample Outputs:

Valid Identifier

Error. Failed to parse.

- c. Implementation of Calculator using LEX and YACC

Sample Inputs:

Enter arithmetic expression: 5*(4+1)

Enter arithmetic expression: 5\$4

Sample Outputs:

Result: 25

Error. Failed to parse.

- d. Convert the BNF rules into YACC form and write code to generate abstract syntax tree.

Sample Input

```
main()
{
    int a, b, c, x, y;
    c = a + b * c;
    x = y / c;
}
```

Sample Output

```
-----
Pos Operator Arg1 Arg2 Result
-----
0   *    b    c    t0
1   +    a   t0   t1
2   =   t1      c
3   /    y    c   t2
4   =   t2      x
-----
```

4. Write program to find Simulate First and Follow of any given grammar.

Sample Input

```
E=TR
R=+TR|#
T=FY
Y=*FY|#
F=(E)|i
```

Sample Output

Transitions Read: (9)

E=TR

R=+TR

R=#
T=FY
Y=*FY
Y=#
F=(E)
F=i

Non-Terminals Encountered: (5)

E T R F Y

Terminals Encountered: (5)

+ # * () i

First(E) = { (, i }
First(T) = { (, i }
First(R) = { +, # }
First(F) = { (, i }
First(Y) = { #, * }
Follow(E) = {), \$ }
Follow(T) = { +,), \$ }
Follow(R) = {), \$ }
Follow(F) = { +, *,), \$ }
Follow(Y) = { +,), \$ }

5. Develop the LL(1) parser for a given language.

Sample Input

S->aA

A->b|c

Sample Output

FIRST(S): { a }

FIRST(A): { b, c }

FOLLOW(S): { \$ }

FOLLOW(A): { \$ }

Predictive Parsing Table:

	a	b	c	\$

S	S->aA	_	_	_
A	_	A->b	A->c	_

Enter string for parsing: ab

Parsing sequence and actions

STACK	INPUT	ACTION

\$ S	a b \$	Shift: S->aA
\$ A a	a b \$	Reduced: a
\$ A	b \$	Shift: A->b
\$ b	b \$	Reduced: b
\$	\$	

Parsed successfully.

- Develop an operator precedence parser for a given language.

Sample Output:

```
Enter the no.of terminals :
4

Enter the terminals :
+*i$

Enter the table values :
Enter the value for + +:>
Enter the value for + *:<
Enter the value for + i:<
Enter the value for + $:>
Enter the value for * +:>
Enter the value for * *:>
Enter the value for * i:<
Enter the value for * $:>
Enter the value for i +:>
Enter the value for i *:>
Enter the value for i i:=
Enter the value for i $:>
Enter the value for $ +:<
Enter the value for $ *:<
Enter the value for $ i:<
Enter the value for $ $:A
```

```
**** OPERATOR PRECEDENCE TABLE ****
      +      *      i      $
+      >      <      <      >
*      >      >      <      >
i      >      >      =      >
$      <      <      <      A
Enter the input string:i+i*i$

STACK          INPUT STRING          ACTION

$              i+i*i$                Shift i
$<i            +i*i$                  Reduce
$              +i*i$                  Shift +
$<+            i*i$                  Shift i
$<+<i          *i$                    Reduce
$<+            *i$                    Shift *
$<+<*          i$                     Shift i
$<+<*<i        $                     Reduce
$<+<*          $                     Reduce
$<+            $                     Reduce
$              $                     String is accepted
```

- Implement Intermediate code generation for simple expressions.

Sample Output:

```
Enter the expression: x=a+b*c-d/e+f
Intermediate code:
Z=b*c
Y=d/e
X=a+Z
W=X-Y
V=W+f
x=V
```

8. Write a program to perform loop unrolling.

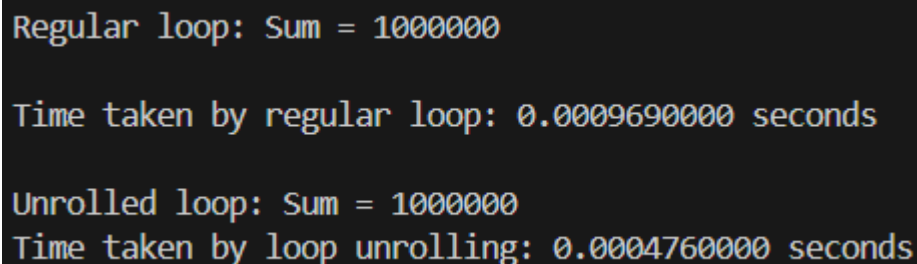
Compare the **regular loop** and the **loop unrolling technique** for summing elements in an array of integers. The unrolled version will process multiple elements per iteration, thus reducing the number of iterations.

```
#define SIZE 1000000 // Define a large array size for better performance visibility. Fill the array with 1's for simplicity
```

```
void sum_regular(int arr[], int size); // This function uses a simple loop to iterate through the array and sum the elements one by one.
```

```
void sum_unrolled(int arr[], int size); // This function uses a loop unrolling technique to sum the array. It processes 4 elements in each iteration, reducing the loop overhead.
```

Sample Output:



```
Regular loop: Sum = 1000000  
Time taken by regular loop: 0.0009690000 seconds  
Unrolled loop: Sum = 1000000  
Time taken by loop unrolling: 0.0004760000 seconds
```

9. Write a program to perform constant propagation.

Sample Input:

```
= 3 - a  
+ a b t1  
+ a c t2  
+ t1 t2 t3
```

Sample Output:

```
+ 3 b t1  
+ 3 c t2  
+ t1 t2 t3
```