# CSC 280 Project: Bison and Flex

## Group: David Ajqui-Diaz, Adrian Dionicio Silva, Micheal Gerbitz

## Instruction:

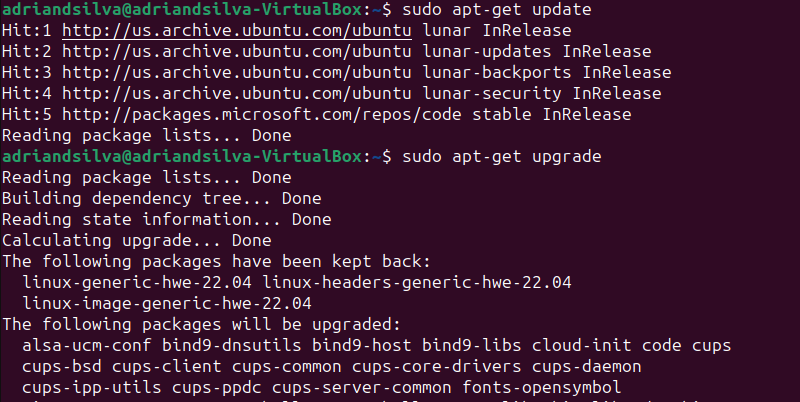
Step-by-step procedure of the tool's working functionality, a document with the procedure used and pertaining results, steps, and screenshots of the results, and a short required technical description of each step

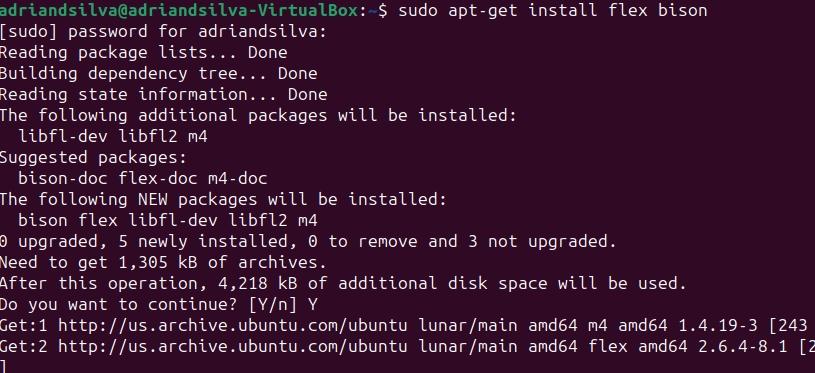
## Introduction

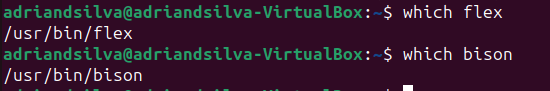
Flex is a scanner generator tool used to perform lexical analysis based on a finite state machine (FSM). The input is a set of regular expressions while the output is code used to implement the scanner according to the input rules. Bison is a LALR parser generator for syntax analysis, used for pushdown automata (PDA). The input is a set of context-free grammar (CFG) rules while the output is code to implement parser based on the input rules.

To install Flex and Bison in Ubuntu, you first need to open the terminal of Linux, with the command of CRTL + ALT + T. After you open the terminal, you need to check for updates for your Ubuntu system. The commands are the following: sudo apt-get update, sudo apt-get upgrade, and finally sudo apt-get install flex bison. To check if flex and bison are installed, you use the which command on both flex and bison.

## Pictures of Installation:

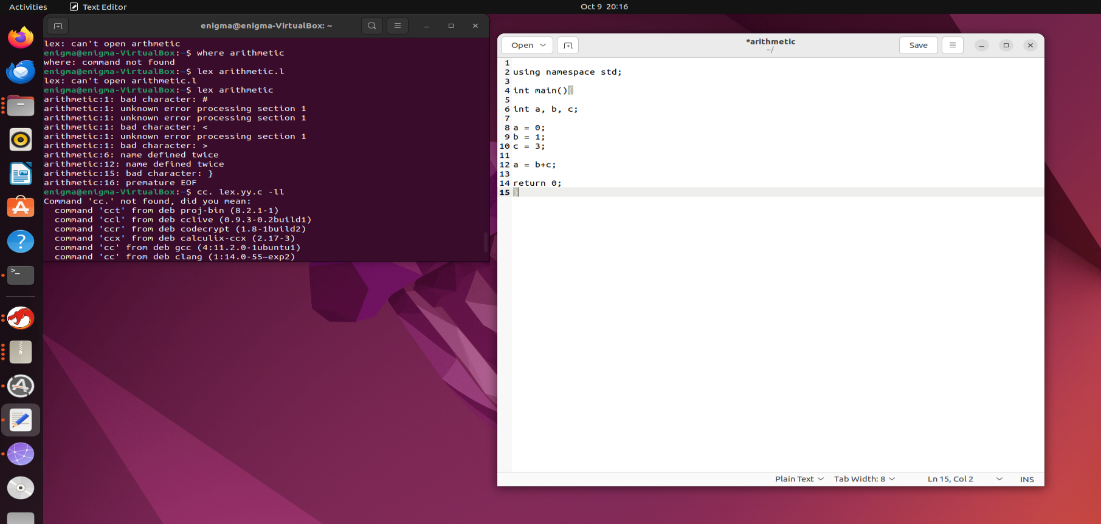
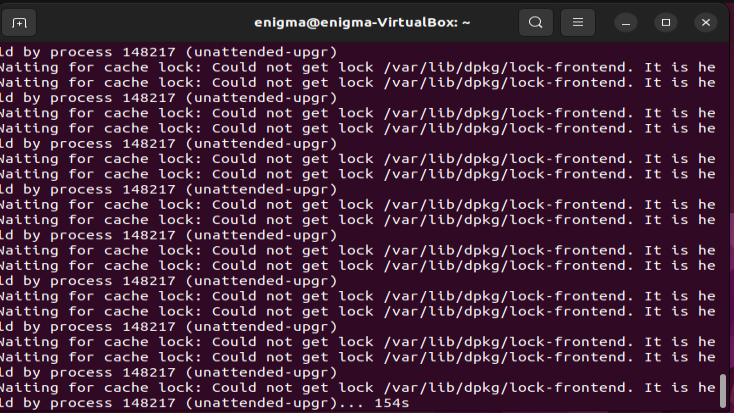


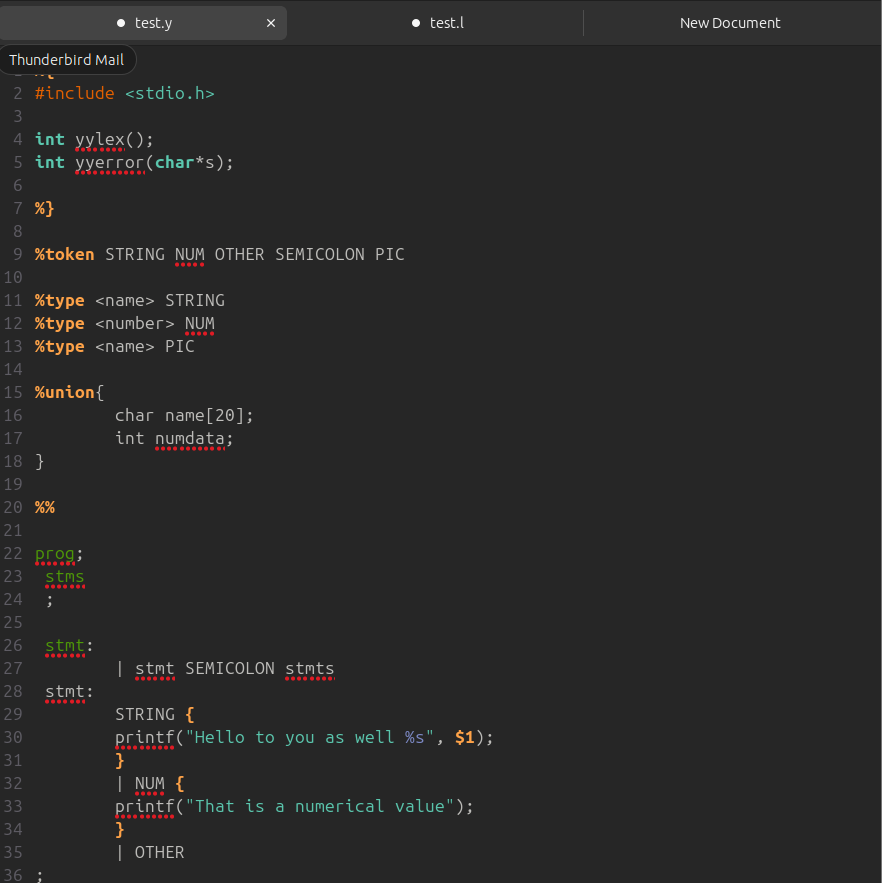


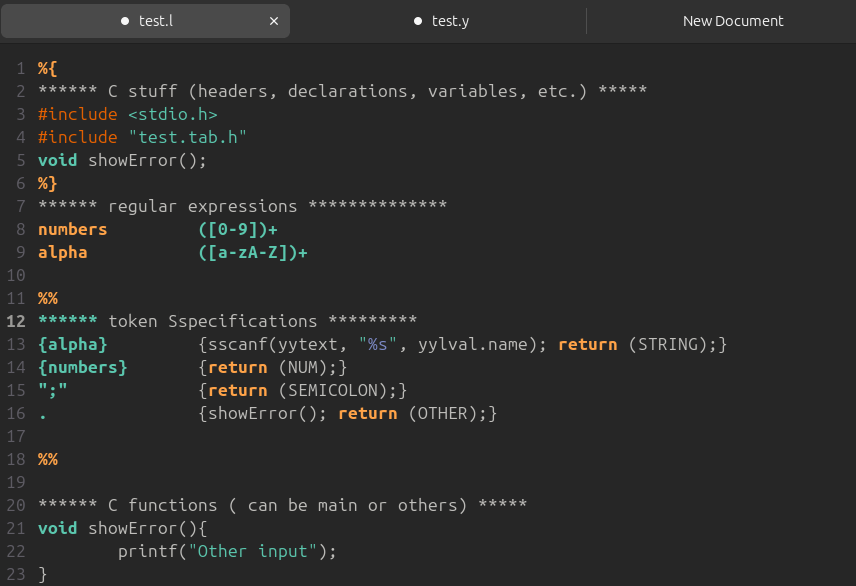


## Problems faced during compilation:

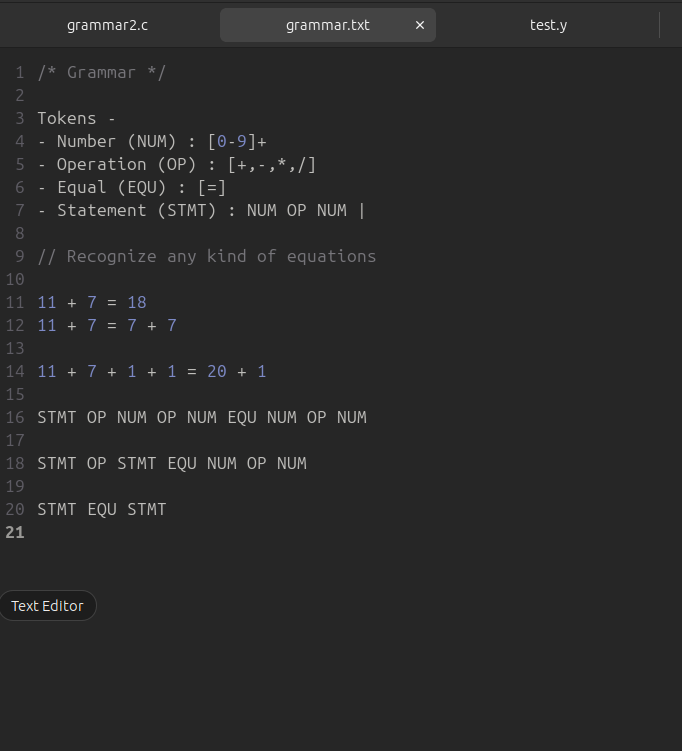
## Our first attempt to compile C++ files into the Ubuntu system.

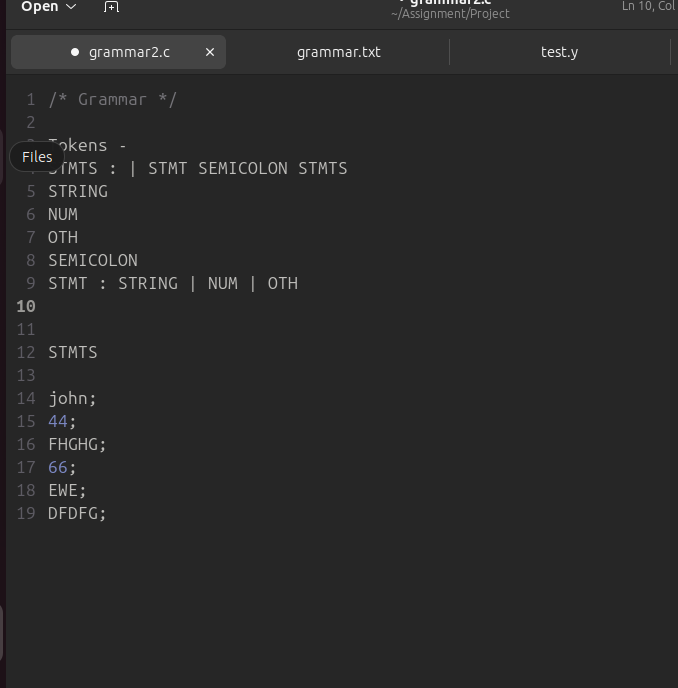


Code that will be implemented with Flex and Bison :



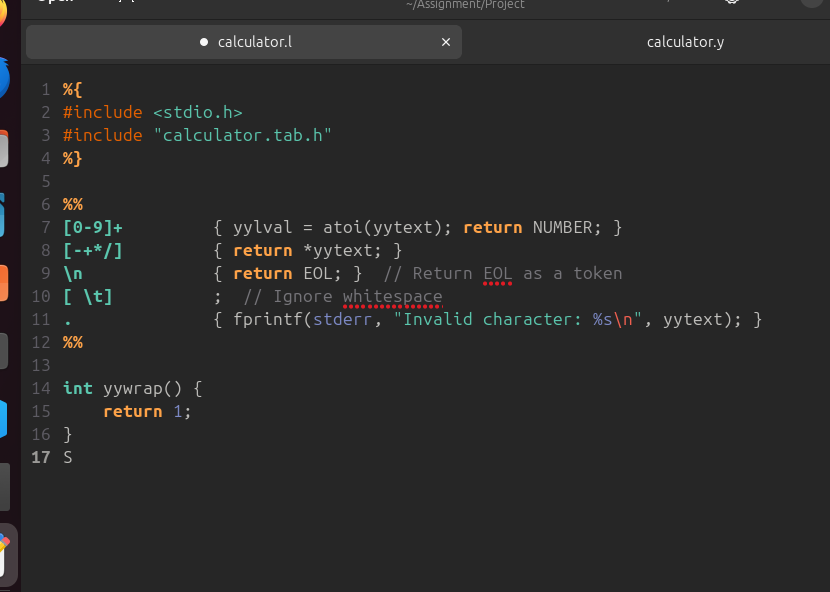
## An attempt to make a grammar:





## Code Changes as of 10/29:

## Start of the flex file of the simple calculator



## Process behind the Flex code:

calculator.l (Lexer file)

%{  
#include <stdio.h>  
#include "calculator.tab.h"  
%}

%{ ... %}: This section allows you to enter C code that will be directly copied to the generated lexer code. In this case, and will include standard input/output header and a reference to the token definitions generated by Bison in calculator.tab.h.

%%  
[0-9]+ { yylval = atoi(yytext); return NUMBER; }

{

Open brace starts the action linked with the rule

[0-9]+: Matches one or more digits.

{ yylval = atoi(yytext); return NUMBER; }:

When digits are matched, it converts the matched string (yytext) to an integer and assigns it to yylval, which is a Bison variable holding the value of the token.

Return NUMBER;

Tells the lexer to return the NUMBER token to the parser with the associated value yylval being the integer value of number

[-+\*/] {

return \*yytext; // Return the operator as a token

}

[-+\*/]: // Matches arithmetic operators (+, -, \*, /).

return \*yytext; }:

line returns the matched operator as a token, the yytext represents a character and returns the operator as a single-character token.

\n { return EOL; } // Return EOL (End Of Line) as a token

\n: Matches newline character and signals the end of a line

{ return EOL; }: Returns EOL token when near the end of the current line

[ \t] { // Ignore whitespace and do nothing

}

Rule matches whitespace characters like space and tab

;: Specifies to ignore these matched characters.

. { fprintf(stderr, "Invalid character: %s\n", yytext); }

.: Matches any character that doesn't match the previous defined patterns.

{ fprintf(stderr, "Invalid character: %s\n", yytext); }: Prints an error message in response to the output stating a invalid character was found

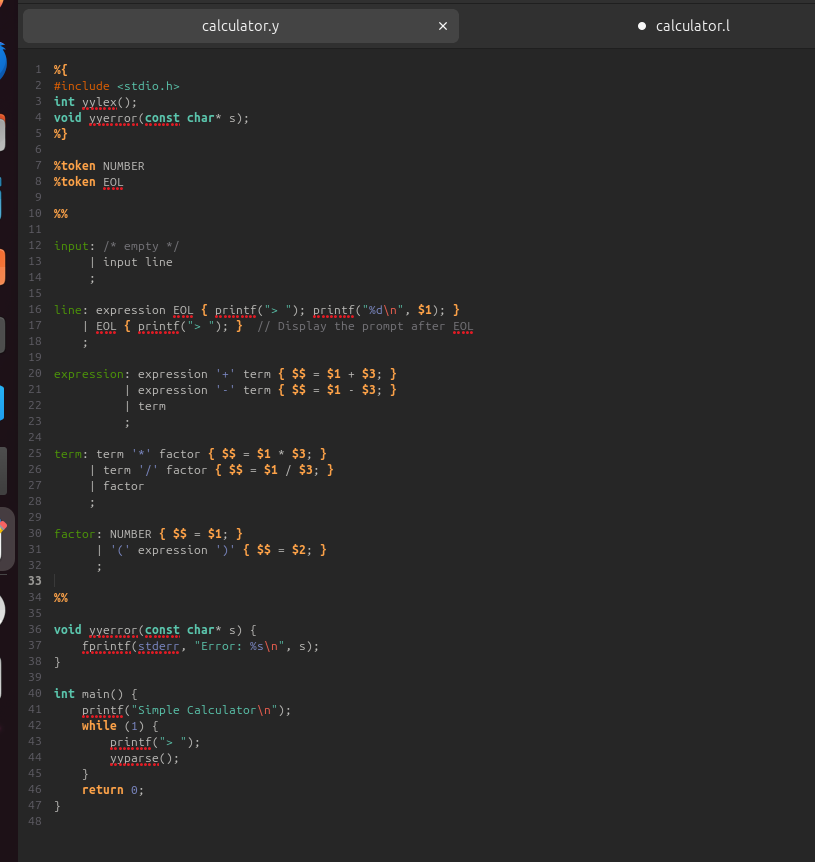
%%

%%: Delimiter that signals the end of the Flex rules and start of the C code section.

int yywrap() {  
 return 1;  
}

A user-definied function that states the end of the input, and returns 1 to indicate the lack of input streams to process

## Start of the bison file of the simple calculator



## Process behind the Bison code:

calculator.y (Parser file)

The calculator.y file defines the grammar rules using Bison syntax.

%{  
#include <stdio.h>  
int yylex();  
void yyerror(const char\* s);  
%}

%{ ... %}: Similar to Flex, You are able to insert C code that is copied into the generated parser code. It includes standard input/output header, a yylex() function, and an error handling function yyerror().

%token NUMBER  
%token EOL

%token: Declares tokens to be used in the parser. Where we define NUMBER and EOL.

%%  
input: /\* empty \*/  
 | input line  
 ;

Defines the grammar rule for input, allowing zero or more lines.

line: expression EOL { printf("> "); printf("%d\n", $1); }  
 | EOL { printf("> "); } // prints the result of the expression and display the prompt  
 ;

line: Defines a line that can be either an expression followed by EOL or just an EOL.

* When it's an expression EOL, it evaluates the expression and prints the result with a prompt.
* When it's just EOL, it prints the prompt without an expression.

expression: expression '+' term { $$ = $1 + $3; } // Add the result of the left and right expression  
 | expression '-' term { $$ = $1 - $3; } // Subtract right expression from left expression  
 | term // if no addition or subtraction, result is the term  
 ;

expression: Represents expressions that involve addition or subtraction operations.

* It can be an expression '+' term, which adds the values of the expressions and the term.
* It can be an expression '-' term, which subtracts the value of the term from the expression.
* It can also be just a term.

term: term '\*' factor { $$ = $1 \* $3; } // multiply left and right terms   
 | term '/' factor { $$ = $1 / $3; } // divides left term by right factor   
 | factor // if no multiplication or division, result is the factor

term: Represents mathematical terms involving multiplication or division operations.

* It can be a term '\*' factor, where it multiplies the values of the term and the factor.
* It can be a term '/' factor, where it divides the value of the term by the factor.
* It can also be just a factor.

factor: NUMBER { $$ = $1; }  
 | '(' expression ')' { $$ = $2; }  
 ;

factor: Represents a single numerical value or an expression enclosed in parentheses.

NUMBER: Represents a numeric value.

'(' expression ')': Represents an expression enclosed within parentheses. The value of the expression inside the parentheses is assigned to $$.

These rules collectively define how the arithmetic expressions are structured and are evaluated based on the precedence and associativity of operators.

void yyerror(const char\* s) {   
 fprintf(stderr, "Error: %s\n", s);  
}

yyerror(): Handles parsing errors, printing an error message to stderr with the provided error message s.

int main() {  
 printf("Simple Calculator\n");  
 while (1) {  
 printf("> ");  
 yyparse();  
 }  
 return 0;  
}

main(): The main function of the calculator program. It prints a welcome message, then continuously prompts for input (displaying > ) and invokes the parsing function yyparse() to process and evaluate the entered expressions.

## An attempt to add a parse tree

A screenshot of a computer

Description automatically generated

Here is a piece of code for a parse tree, although to our dismay it did not work. However, with this attempt, we did discover that Bison makes it possible for scanners like Flex to be able to just as well as identify possible nodes and create a parse tree w/ the help of Bison

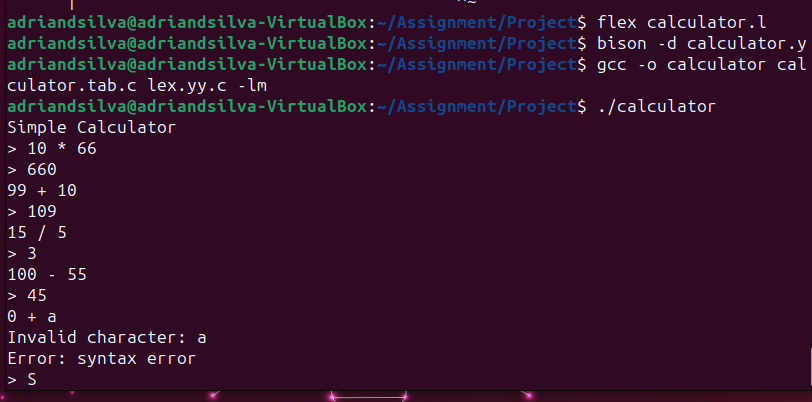
## The Simple Calculator Program in Linux:

flex calculator.l

bison -d calculator.y

gcc -o calculator calculator.tab.c lex.yy.c -lm

./calculator



## Tools Used

http://www.jonathanbeard.io/tutorials/FlexBisonC++

https://learnmoderncpp.com/2020/12/16/generating-c-programs-with-flex-and-bison-1/

<https://web.iitd.ac.in/~sumeet/flex__bison.pdf>

https://www.talkplayfun.com/bison\_flex/001\_Bison\_and\_Flex\_with\_Cygwin\_Bison\_and\_Flex.pptx   
<https://aquamentus.com/flex_bison.html>

<https://youtu.be/POjnw0xEVas?si=zAoCOGOSo8RTtYH6>

<https://www.cse.scu.edu/~m1wang/compiler/TutorialFlexBison.pdf#:~:text=Flex%20is%20a%20scanner%20generator%20tool%20for%20lexical,implement%20the%20scanner%20according%20to%20the%20input%20rules>.