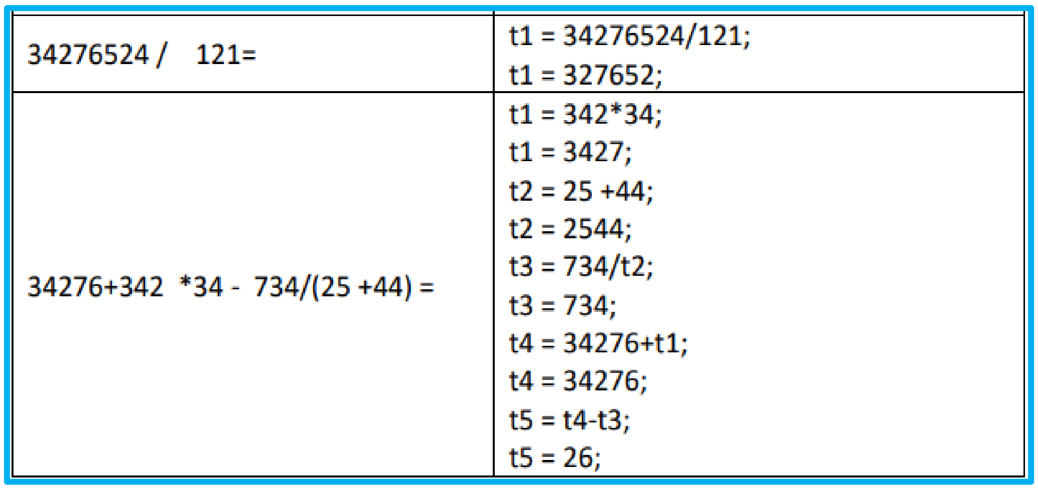
Arithmetic Expression Compiler (using Flex & Bison)

Mohammad Sadra Sarparandeh 40030506

It was tasked to convert arithmetic expression to Three Address Code (TAC) so that

- b+a: digits of number b that are not in number a are added to the end of a.
- b-a: digits of number b that are in number a are removed from a.
- b*a: the digit resulting from the sum of the digits (or the sum of the digits) of the number b is added to the end of a if it does not exist in a.
- b/a: the digit obtained from the sum of the digits (or the sum of the digits) of the number b, if it exists in a, is removed from a.



34276524 / 121=	t1 = 34276524/121;
	t1 = 327652;

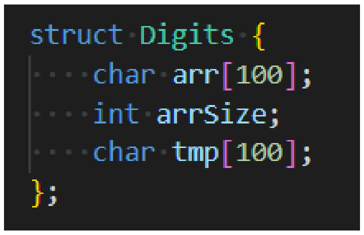
34276+342 *34 - 734/(25 +44) =	t1 = 342*34;
	t1 = 3427;
	t2 = 25 +44;
	t2 = 2544;
	t3 = 734/t2;
	t3 = 734;
	t4 = 34276+t1;
	t4 = 34276;
	t5 = t4-t3;
	t5 = 26;

We created Lexical phase with Flex and Syntax Analysis and TAC with bison.

In Lexical phase we accepted just digits from [0-9] as number & operators as it is

and several white spaces will be ignored.

If we reach number on input we convert yytext as char* to struct Digits. This helps us to have multiple necessary data (such as temporary variable and desired data) on Bison file.

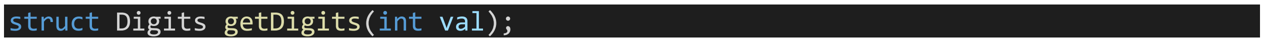


```c
struct Digits {
    char arr[100];
    int arrSize;
    char tmp[100];
};
```

char* arr stores number as digits character array (ex 12 : arr[0] = '1' , arr[1] = '2')

arrSize is size of char* arr and length of number

char* tmp stores result as temporary variable such as t1 , t2,…

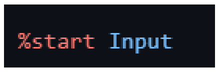


```c
struct Digits getDigits(int val);
```

converts the input number to struct Digits.

After Lexical phase we pass valid data to bison file for Syntax analysis and generatinh three address code.

%token helps us transfer tokens from Lexical to syntax so our struct digits is in NUMBER token and if we hit '\n' this return END token. Arithmetic priority is also important so we add %left '+' '-' , %left '*' '/' for this purpose.



```
%start Input
```

starts our semantic rules from here like this

Input -> Input Line

Line -> END | E END

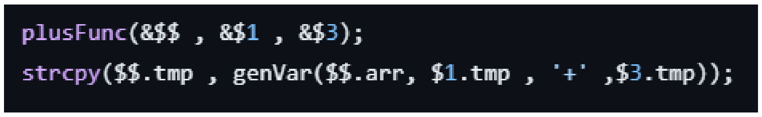
E -> T | E '+' T | E '-' T

T -> F | T '*' F | T '/' F

F -> '(' E ')' | NUMBER

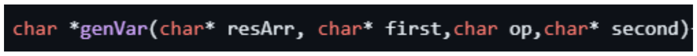
NUMBER as we said is terminal token that is actually our number returned from lexical phase.

Semantic action takes place after the desired rule is met for example sumFunc that is for '+' functionality & and storing numerical result in struct array will take place in E '+' T.



```
plusFunc(&$$ , &$1 , &$3);
strcpy($$.tmp , genVar($$.arr, $1.tmp , '+' ,$3.tmp));
```

$$ is current value and will be initialized in function with '+' , $1 is E and will be initialized after parse tree is finished , $3 is third token. For generating temporary variables we created



```
char *genVar(char* resArr, char* first,char op,char* second)
```

function and it add to global variable varInd for next temp variable and also print arithmetic calculation to terminal . temp variable then is copied to current data tmp attribute.

If we hit the end of input END token will be returned and semantic action return 0; takes place.

References:

https://web.mit.edu/gnu/doc/html/bison_6.html

https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/

http://www-h.eng.cam.ac.uk/help/tpl/languages/flexbison/

https://github.com/tyrro/simple-calculator-using-Flex-and-Bison

https://codereview.stackexchange.com/questions/280086/mathematical-expression-evaluator-c-using-flex-and-yacc

https://www.geeksforgeeks.org/yacc-program-to-evaluate-a-given-arithmetic-expression/