Karina Banda
CSC 173
C Week 3/4 Project

For this project the goal was to build a calculator by implemented a scanner, parser, and evaluator for arithmetic equations. Within the scanner the input is read character by character and travels through my DFA until it has finished accepting the current token and making sure it is of my language. After getting a token, the token transverses through my recursive parser, making sure it is legal within my grammar and figuring out the structure of the equation. It does this until it reaches an end node within the implicit parse tree. Once it reaches an end node then the information of the token gets sent back up to be evaluated within the appropriate places within the tree.

Scanner:

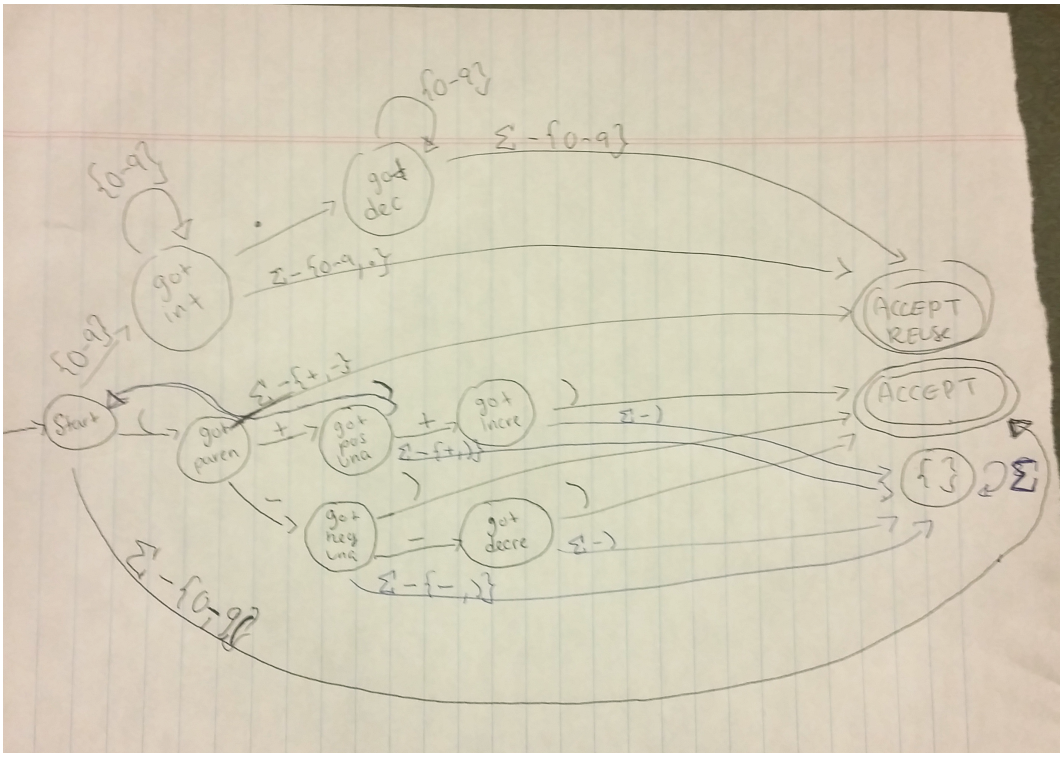
Alphabet:

For my DFA language, the alphabet is the following:

Σ={num, +, -, *,/,%,(+),(-),(++),(--),(,),;, .}

Num being all numbers 0-9.

DFA:



Within the scanner, whitespaces and new lines are not of importance so they are skipped and do not go into the parser. In my grammar, unary operators must be enclosed within parenthesis. The unary operator (+) does not change the answer so it is also skipped as well and the scanner goes onto the next token without transcending the parser.

In the DFA in the scanner, most characters of my alphabet can be immediately accepted which are: +, -, /, *, %, and ;. There is only a need to look ahead for when there is a number and when there is a (. It is essential to keep going through the DFA when a number is found because it might be more than a one digit number so I have to keep building the number within the DFA before the token can be returned. From Start, when a number is found, it enters "got_num" state. In the got_num state, it may keep entering this same state if more digits follow or go into "got_dec" state if a period is found to be within the number. If anything else besides a number or a period is found then the method ACCEPT_REUSE is called because we have to accept the token but we also have to put the location of the next token back one so that the scanner doesn't eat up a character in the process. When we see a (, we have to stay within the DFA until it has evaluated which type of unary operator it is or if it does not enclose one at all.

Parser:

Grammar:

S->ALL ;
ALL->EXPR ALL | ε
EXPR->TERM TMORE
TMORE->OP2 TERM TMORE | ε
TERM->UNA UNIT UMORE | ε
UMORE->OP1 UNIT UMORE | ε
UNIT->(UNA EXPR UNA)| num
UNA->(++) | (--) | (-) |ε
OP2->+|-
OP1-> * | / | %

NOTE: num is a terminal, it is any number value, every unary must be enclosed in parenthesis in my grammar

FIRST(S) = { (++) , (--) , (-), (, num , + , - , ;}	Follow(S) =NONE
FIRST(ALL) = { (++) , (--) , (-), (, num , + , - }	Follow(ALL) = { (++) , (--) , (-), (, num , + , - , ;}
FIRST(EXPR) = { (++) , (--) , (-), (, num , + , -}	Follow(EXPR) = { (++) , (--) , (-), (, num , + , - , ;}
FIRST(TMORE) = { + , - }	Follow(TMORE) = { (++) , (--) , (-), (, num , + , - , ;}
FIRST(TERM) = { (++) , (--) , (-), (, num }	Follow(TERM) = { (++) , (--) , (-), (, num , + , - , ;}
FIRST(UMORE) = { * , / , % }	Follow(UMORE) = { (++) , (--) , (-), (, num , + , - , ;}
FIRST(UNIT) = { (, num }	Follow(UNIT) = { * , / , %, (++) , (--) , (-),) , + , - , ;}
FIRST(UNA) = { (++) , (--) , (-) }	Follow(UNA) = { (++) , (--) , (-), (, num , + , - , ; }
FIRST(OP2) = { + , - }	Follow(OP2) = { (++) , (--) , (-), (, num , + , - , ; }
FIRST(OP1) = { * , / , % }	Follow(OP1) ={ (, num }

PREDICT(S-> ALL ;) = { (++) , (--) , (-), (, num , + , - , ;}	EPS(S) = false
PREDICT(ALL ->EXPR ALL) = { (++) , (--) , (-), (, num , + , - , ;}	EPS(ALL) =true
PREDICT(ALL->ε) = {;}	EPS(EXPR) =true
PREDICT(EXPR -> TERM TMORE) = { (++) , (--) , (-), (, num , + , - , ;}	EPS(TMORE) = true
PREDICT(TMORE -> ε) ={ (++) , (--) , (-), (, num , + , - , ;}	EPS(TERM) = true
PREDICT(TMORE ->OP2 TERM TMORE) = { + , - }	EPS(UMORE) = true
PREDICT(TERM ->ε) = { (++) , (--) , (-), (, num , + , - , ;}	EPS(UNIT) = false
PREDICT(TERM -> UNA UNIT UMORE) = { (++) , (--) , (-), (, num}	EPS(UNA) = true
PREDICT(UMORE ->ε) = { (++) , (--) , (-), (, num , + , - , ;}	EPS(OP2) = false
PREDICT(UMORE -> OP1 UNIT UMORE) = { * , / , % }	EPS(OP1) = false
PREDICT(UNIT -> num) = {num};	
PREDICT(UNIT -> (UNA EXPR UNA)) = {(};	
PREDICT(UNA -> ε) = { (++) , (--) , (-), (, num , + , - , ; }	
PREDICT(UNA -> (-)) ={(-)}	
PREDICT(UNA -> (--)) ={(--)}	
PREDICT(UNA -> (++)) ={(++)}	
PREDICT(OP2 ->-) ={-}	
PREDICT(OP2 ->+) ={+}	
PREDICT(OP1 -> %) ={%}	
PREDICT(OP1 ->/) ={/}	
PREDICT(OP1 ->*) = {*}	

Although my grammar is not an LL(1) grammar some of the productions that have the same left side are joint, it still works. As I was carefully building my parser, it was easy to make the choice of which production to take. For example, the productions of TERM are jointed but here I have shown what choices I made on the picture to the right. Each production has a point to it. TERM has the point of getting more numbers to trail behind the equation. If there are operators that show up instead then that means it is not a number we need so we need to exit

```
float term(){
    float temp=0;

    switch(tok.tc){
        //Term->E
        case T_PLUS:
        case T_MINUS:
        case T_SEMI:
            break;
        //valid chars, TERM->UNIT UMORE
        case T_LPAREN:
        case T_NUM:
        case T_DEC:
            temp=unit();
            temp=umore(temp);
            break;
```

(TERM -> E). If it is a number or "(", it will continue to transcend the parse tree to evaluate it. I also have "(" there because that means that what ever comes after it will be evaluated into one single value due to order of operations.

Evaluator

Based on how I built my tree, I was able to evaluate the implicit parse tree by having the productions send information to each other. This also enabled my calculator to follow order of operations because in the instance of the above TERM method, for the production TERM ->UNA UNIT UMORE, UMORE is the production responsible for having a *, /, or % which has higher precedence over + or -. Here the value that is received from UNIT is sent into UMORE where it is evaluated within UMORE if there is a +, -, / with the leftmost element before it checks to see if there are more *, /, % to the right. .

```
//UMORE->OP1 UNIT UMORE
case T_MULT:
    op1();
    tempmore=unit();
    //multiply the right side of the * with what was on the left and brought over from within term (for order of operations)
    new=x*tempmore;
    //printf("ANSWER %f",new);
    //printf("\nToken:%u",tok.tc);
    new=umore(new);
    break;
```