Write up: "Scan, Parse, Evaluate"

By Ilya Duskin and Aleksandra Shanina (Partner Project)
Extra Credit: The program is also able to read and interpret "^". This project was done recursively all throught without the use of any linked list or extra objects.

Part 1: Scanner

Scanner's main purpose is to break up a file's input into tokens. For example the expression:

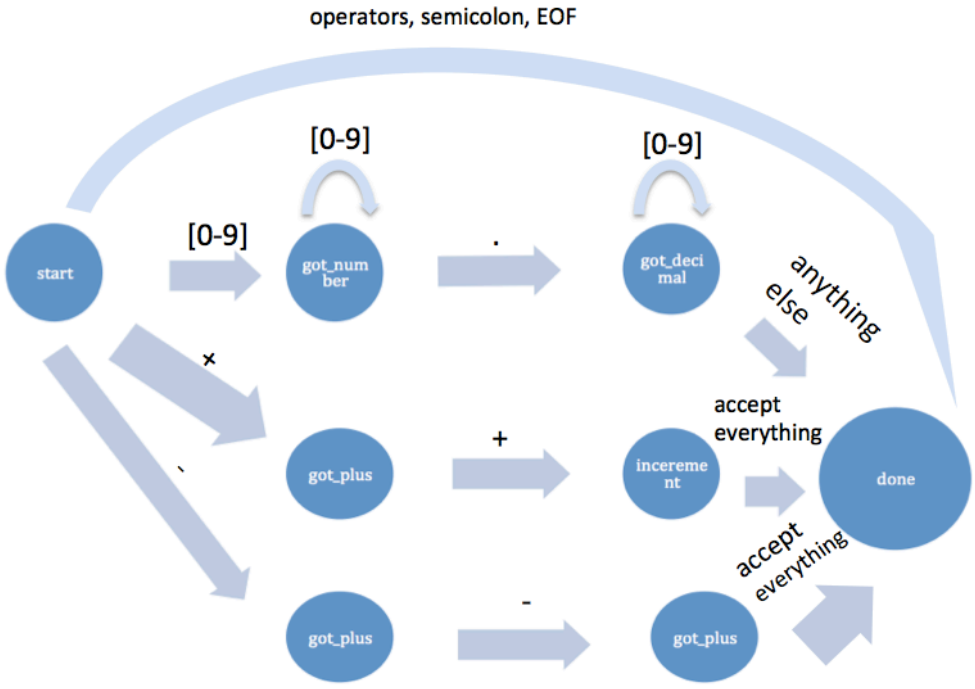
-(1 +2++) / 3.42 *2–30mod3;

Would be broken into the following tokens: "-", "(", "1", " ", "+", "2", "++", ")", " ", "/", " ", "3.42", " ", "*", "2", " ", "-", "30", "mod", and "3".

It should be noted that each token is assigned a class (e.g. "1" is a T_LITERAL, and "+" is a T_ADDOP). This is a whole list of token classes that this scanner used:

```
typedef enum {
        T_EOF,  T_LITERAL, T_ADDOP, T_MULTOP, T_SEMIC, T_LPAREN, T_RPAREN,
        T_OTHER, T_INC, T_DEC, T_POW, T_SPACE
} token_class;
```

The scanner decides what token it is looking at (class) and when a certain token ends with the use of a series of switch statements that change states. The states are the blue circles, and the cases for the states are lines emerging from it. Look at the DFA below for details.



When is token is read from a file, the location in the file where it begins, and its length are “sent” to the parser. In reality, the parser sends the address to the global variable - the token and the current location to the parser, which updates it.

Parser

Here are the grammar rules

```
/*****
                GRAMMAR RULES
                                        *****/

Program -> StatementList [EOF]
StatementList -> Statement StatementList
StatementList -> eps
Statement -> Expression ;
Statement -> ;
Expression -> Arithmetic ArithmeticTail
ArithmeticTail -> Addop Expression
ArithmeticTail -> eps
Arithmetic -> Geometric GeometricTail
GeometricTail -> Multop Arithmetic
GeometricTail -> eps
Geometric -> Negation NegationTail
NegationTail -> ( Expression ) Power
NegationTail -> literal Power
Negation -> Negation
Negation -> eps
Power -> Powerop NegationTail
Power -> eps
Addop -> +
Addop -> -
Multop -> *
Multop -> /
Powerop -> ^

Literal is any number
Eps is empty
```

My parser receives individual tokens from the scanner and starts building the parse tree using the given grammar. It automatically traverses all the way to the bottom left where it receives a token from the scanner and decides what to do with it (i.e. if it’s a literal, negative literal, expression or a negative expression). Once it parses the variable in, the parses returns the value up the tree while traversing up until it sees any of the operators after which it starts building tree branches to the right direction.

Parses recognizes a negative or a positive sign in from of a number/expression. If the sign is negative, it negates the number/expression, if positive, it just returns the number/expression by itself.

It also recognizes increments or decrements ONLY when they follow a number or an expression. It will throw a syntax error if an increment or a decrement is found in front of a number or an expression.

Parser also recognizes power operator (^) and performs the operation using double pow(double a, double b) function that doesn't always work correctly due to the nature of the pow function.

Parser has math.h library included that is used for the powers and modulus operations.

Parser outputs the results in output.txt file with a following format:
= result1
= result2
etc