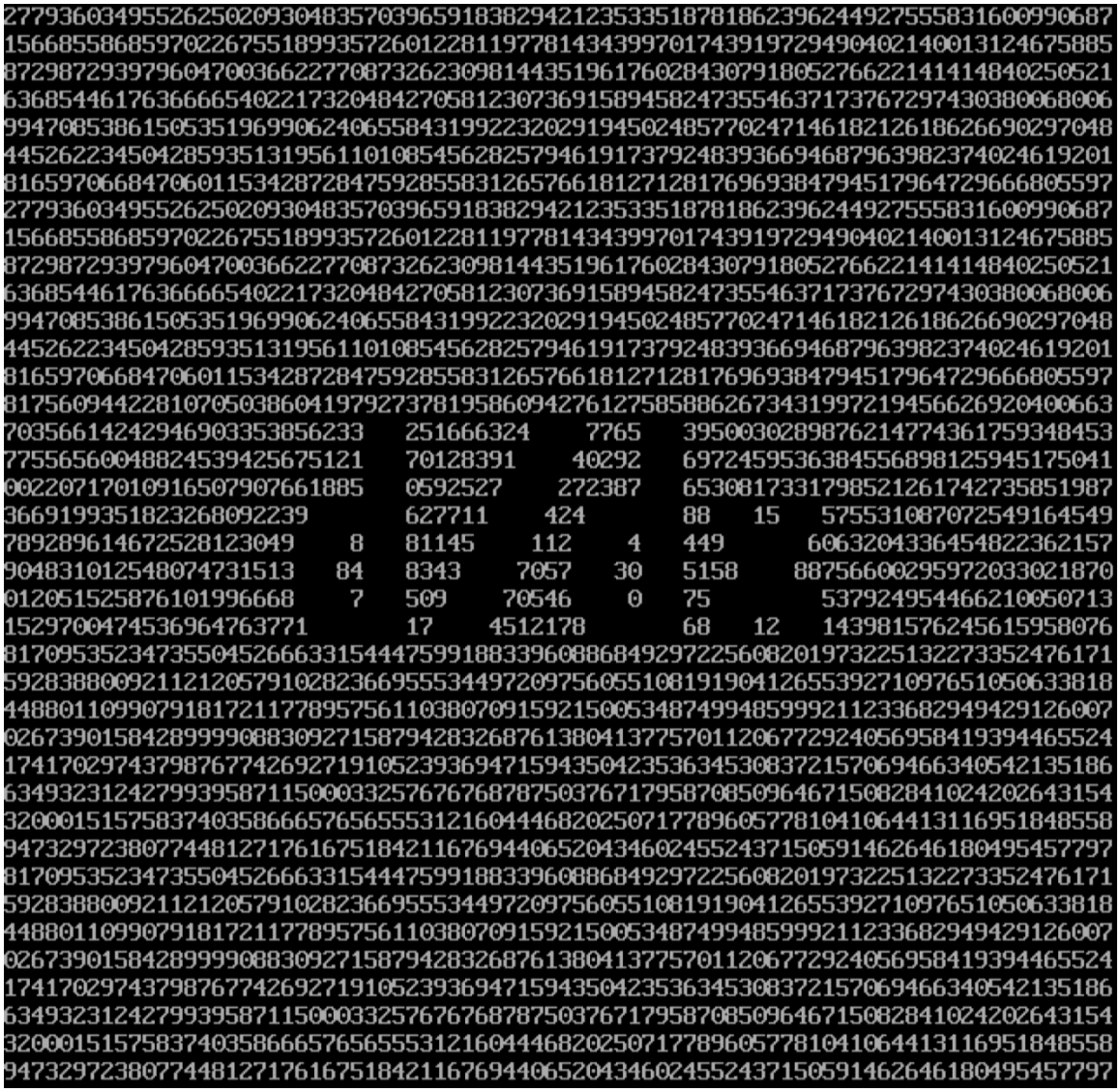
MATHEMATICAL EXPRESSION SIMPLIFYING PROGRAM

~ USING EDITABLE USER-DEFINED RULES ~



Utsav Munendra

Class: XII - A

Roll No.: _____

Subject Teacher: Ms. Pinkie Srivastava

ACKNOWLEDGEMENT

I would like to thank my computer teacher, Ms. Pinkie Srivastava for giving me the opportunity to make this project which allowed me to explore the process of solving mathematical expressions in a computational sense and also for teaching me the C++ programming language through which I was able to piece together my thoughts and ideas to form this program.

INTRODUCTION

This program is essentially a term-rewriting program which repeatedly applies all the available rules to the expression being simplified. The rules are maintained in a rule book, which can be edited by the user. The program also comes with the capabilities of allowing the user to add and delete rules.

'Rule', here and henceforth, mean a combination of two mathematical expressions, one being a match expression and the other being a replace expression. These rules try to model mathematical axioms and theorem. For example, $n(x+y) = nx + ny$ is a rule, with LHS being the match expression and RHS being the replace expression. Also, x, n and $+$ are tokens, the smallest unit of expression in this program.

The three main components of the program are the expression-syntax handler, the expression simplifier and the user interface. The main class is `Solver` and its variables and functions incorporate syntax handling and expression simplifying parts, whereas the user interface is primarily maintained through functions of `main()` and associated global functions.

The algorithm implemented for expression re-writing is based on matching and replacing parts of expressions. For each rule present, the program makes each token in the question expression a starting point for search and then tries to find a match with the rule. If the rule is found to match, then the parts of the question expression which matched are replaced by another group of tokens, as specified by the rule. Numerous functions use several algorithms to achieve their task and some of noteworthy techniques are presented as follows:

- Dijsktra's shunting yard algorithm for conversion into postfix
- Recursive function for calculating HCF
- Push and pop operations on a stack
- File manipulation: creating, deleting, writing, appending, traversing
- Window-like user interface with enter-less input functionality
- Pausing bulk writing on screen when text content start to move out of view

The program also uses various Standard Library header files, which along with their used functions are listed below:

- fstream.h : open(), cout, overloaded shift operators
- math.h : pow()
- stdio.h : rename(), remove()
- ctype.h : tolower()
- conio.h : clrscr(), getch(), cprintf(), textcolor(), textbackground(), wherex(), wherey(), gotoxy()

SPECIFICATIONS

The program is found to run successfully on the following hardware and software:

System	
Manufacturer	Dell
Model	Inspiron 3647 Desktop Computer
Processor	Intel Core i3-4125
RAM	4.00 GB
Operating System	Windows 10 Home Single Language
Type	x64-bit OS and Processor

Compiler	
IDE	Turbo C++ v3.2
Emulator	DOSBox v0.74

VARIABLES AND FILES

Constants:

Integer `t` is most used constant in the program. Almost all arrays are declared with a size of `t`. Turbo C++ does not allow objects to have a size greater than 64KB and with `t = 45`, the size of `solver` is 62837 bytes (61.36 KB), just 2.6 KB short of the maximum limit. Hence, the 45 is the maximum permissible value of `t`. Constant `s` acts as a sentinel and is only used in `applicableRules()` function.

The two enumerators are used for resolving `operatorID` and to identify the rule to be applied. Their use greatly increases the readability of program. As for rule numbers, the first 45 numbers are to identify the user-defined rules while the next 7 numbers are reserved for basic numerical operations that are fundamental, yet inexpressible as a generic user-defined rule. Numerical operators and constant differentiation had to be implemented in the program itself.

Class `Solver`:

Solver is the class which packages all the most important functions and variables together. The rest of the classes, namely, `Token`, `Stack`, and `Rule` have a composition relationship with this class.

Description of trivial functions is mentioned in the program itself. Important data members and member functions which are responsible for expression syntax handling are as follow:

- **`Token postfix[]:`** Stores the postfix expression converted by `toPostfix()`.
- **`Token ruleArray[]:`** This token array stores the expression on which a rule has to be applied. During expression simplification, this array stores the intermediate results which are processed by `toInfix()` and `fixInfix()` and then displayed to user.
- **`Stack expression:`** It is the stack on which `applyRule()` first copies the `ruleArray[]`, then applies the rule to finally copy the simplified expression back into `ruleArray[]`.
- **`char *fixExpression():`** Makes sure that the expression is in the right format for `toPostfix()` to act upon. Overall, the function removes spaces in the string expression, changes braces and square brackets to parentheses, converts prefix differentiation operator to infix differentiation operator and inserts ignored multiplication signs.
- **`void fixNumbering():`**The function renumbers the given variable and numerical placeholders of a user-defined rule expression. This allows the user to number the placeholders anything. When the expression to be fixed is for `userRules[].match`, then

the function first initializes `rn[]` and `rv[]` and then renumbers the match expression. However, if the expression to be fixed is for `userRules[].replace`, then the function straightaway renumbers the replace expression using the `rn[]` and `rv[]` values initialized during the prior run.

- **`void toPostfix():`** Converts an infix character array into a postfix token array. This is stored in the postfix variable for use by other functions. It uses Dijkstra's Shunting Yard Algorithm and also correctly manipulates the right associativity of exponent operator.

- **`void toInfix():`** Converts a postfix token array into an infix token array. This function is only called by the `solve()` function and the postfix expression from every step of evaluation is passed to this function before being displayed on screen. For most operators, if the operands consist of more than one token, then the operands are automatically parenthesized. This is not the case for addition and subtraction, hence they are dealt separately. Also, derivative is a prefix operator and unary minus requires a single operand, so these operators are also handled outside the generic algorithm.

- **`char *fixInfix():`** The `toInfix()` function outputs an infix array of tokens. This has to be converted into a string in order to be displayed on the screen or in a file. The function uses a unique approach to solve this problem. Instead of using another function which converts `operatorID` and number tokens to string, and then concatenates all of these strings, the function uses the already existing `print()` function to output the string equivalent of each token onto a temporary file, and then uses the `get()` function of `fstream.h` to input the entire expression a very few lines of code.

- **`void loadAllRules():`** Loads all the rules from `ruleArr[][]` and `replaceArr[][]` to `userRules[]`. For each rule expression in the two character arrays, it calls the appropriate functions to first fix the expression, then fix the numbering, followed by its conversion into postfix and finally copies it to a `Rule` object in the `userRules[]` array.

The rest of the data members and the member functions of this class `Solver` handle expression simplification, and are as follows:

- **`Rule userRules[]:`** The array of `Rule` object stores the user-defined rules.

- **`char ruleArr[][]:`** This stores the infix string expressions as extracted from the `ruleBook` file and these expressions are later used to initialize the `match` expressions of user-defined rules.

- **`char replaceArr[][]:`** It is similar to `ruleArr[][]`, with the only difference being that it stores those expressions which later initialize the `replace` expressions of user-defined rules.

- **`char nameArr[][]:`** It stores the names of the user-defined rules as extracted from the rulebook file and initializes the `ruleName` data member of `Rules` objects from `userRules[]`.

- **`char rn[], rv[]:`** These variables are used by `fixNumbering()`. For example, if in a rule expression, `N`4`, `N`7` and `N`2` appear in this order, then `rn[0] = 4`, `rn[1] = 7` and `rn[2] = 2`. The indexes serve as a new numbering for the placeholders and since the indexes are consecutive and in ascending order, it makes the coding for expression simplifying shorter.

- **`char fileName[]:`** Stores the name of the file from which `loadRules()` loads the rule. The default file is set to `ruleBook.txt`, but if this file fails to open, `setFile()` prompts the user to enter another file before starting the program.

- **`void applyRule():`** It receives the position and rule number as parameters. The function then copies the tokens upto the rule application index (`at`), removes the `match` expression, pushes the `replace` expression and finally copies the remain tokens to `expression`. In the end, the entire `expression` is copied back to `ruleArray[]`, so that the result can be displayed.

- **`int applicableRules():`** Identifies the rule number for a rule which can be applied and its application index for the given token array. It acts as a sieve and tests several conditions. If any conditions evaluates to true, the next token in the array and in the rule are tested for the same conditions. If the rule is applicable, none of the conditions would be failed, but otherwise the function will start the same process again for the next rule in `userRules[]`.

- **`void solve():`**It takes the question as a string for input and fixes it, converts into postfix, finds the applicable rules and keeps applying them using the above functions until no rule can be further applied. The function also takes an `ostream` object, which can be either `cout` or an `ofstream` file for displaying the output of each expression on the screen or in a file respectively.

Files:

`RuleBook.txt` stores all the user-defined rules. Every rule must start with "Rule: ", followed by the rule name. The next two lines should contain the match and replace expressions. Any other lines are ignored.

Assignment files have questions starting with "Q: " and the answers of the assignment are appended in the file `Answers.txt`.

Code

```
#include <fstream.h>
#include <conio.h>
#include <math.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

const int t = 45;
const int s = -32465;

//Constants for operator ID
enum
{
    NullToken,
    Plus,
    Minus,
    Multiply,
    Divide,
    OpenP,
    CloseP,
    Exponent,
    UnaryM,
    Derivative,
    Number,
    Variable,
    RuleNumber,
    RuleVariable,
    Error
};

//Constants for Applicable Rule No.
enum
{
    NoRule = t,
    NumericAdd,
    NumericSub,
    NumericMul,
    NumericDiv,
    NumericExpo,
    NegateNum,
    DConstant
};

//Represents a unit: number, variable or operator.
class Token
{
  public:
    int operatorID;
    int isOperator;

    char variable;
    double value;
```

```
    Token();
    Token(char);
    Token(Token *);

    void setValue(int);
    void setVariableName(char);
    void initialization();
    void print(ostream &);
    void setToken(char);
    int equals(Token);
};

//Class to represent a stack of Tokens
class Stack
{
  public:
    Token array[t + 15];
    int total;

    Stack() { initialize(); }
    void push(Token c) { array[total++] = c; }
    Token pop() { return array[--total]; }
    Token see() { return array[total - 1]; }
    Token see(int index) { return array[index]; }

    void initialize()
    {
        for (int i = 0; i < t + 15; ++i)
            array[i].initialization();
        total = 0;
    }
};

//A Rule against with the given expression is matched
class Rule
{
  public:
    Token match[t];
    Token replace[t];
    char ruleName[t];
};

//Variables and functions for simplifying an expression
class Solver
{
  public:
    Token postfix[t];
    Token ruleArray[t];
    Stack expression;

    Rule userRules[t];

    char ruleArr[t][t];
    char replaceArr[t][t];
```

```
    char nameArr[t][t];

    char rn[t], rv[t];
    char fileName[t];

    //Important Functions for syntax handling
    char *fixExpression(char array[]);
    void fixNumbering(char array[], int isReplace);
    void toPostfix(char array[], int length);
    void toInfix(Token token[], Token infix[], int &l);
    char *fixInfix(Token token[], int l, char array[]);
    void loadAllRules();

    //Important functions for expression simplifying
    void applyRule(int rule, int arrayIndex, int &lenArray);
    int applicableRules(Token array[], int &len, int &at);
    void solve(ostream &file, char ques[], int fromFile);

    //Trivial functions helping in syntax handling
    void initialize();
    void initialize(char array[]);
    int getOrder(int operatorID);
    int isLeftAssociative(int c);
    int precedence(int a, int b);
    int isConstant(char c);
    int isDigit(char c);
    int length(Token array[]);
    int length(char array[]);
    void copyArray(Token a[], Token b[]);
    void copyChar(char a[], char b[]);
    void clearRow(Token array[], int l);
    void shiftRow(Token array[], int l);

    //Trivial functions helping in expression simplifying
    int getOperand(Stack &operand);
    int hcf(int, int);

} solver;

//Token class Functions
Token::Token() { initialization(); }
Token::Token(char c) { setToken(c); }

Token::Token(Token *tok)
{
    operatorID = tok->operatorID;
    isOperator = tok->operatorID;
    variable = tok->variable;
    value = tok->value;
}

void Token::setValue(int a)
{
```

```
    value = a;
    operatorID = Number;
    isOperator = 0;
}

void Token::setVariableName(char a)
{
    variable = a;
    operatorID = Variable;
    isOperator = 0;
}

void Token::initialization()
{
    isOperator = 0;
    variable = '\0';
    value = 0;
    operatorID = NullToken;
}

void Token::print(ostream &file)
{
    if (operatorID == NullToken)
        file << "NULL";
    else if (operatorID == Error)
        file << "Too Much";
    else if (isOperator)
    {
        switch (operatorID)
        {
        case Plus:
            file << '+';
            break;
        case Minus:
            file << '-';
            break;
        case Multiply:
            file << '*';
            break;
        case Divide:
            file << '/';
            break;
        case OpenP:
            file << '(';
            break;
        case CloseP:
            file << ')';
            break;
        case Exponent:
            file << '^';
            break;
        case UnaryM:
            file << '-';
            break;
```

```
        case Derivative:
            file << "d/d";
            break;
        }
    }
    else if (operatorID == Number)
        file << value;
    else if (operatorID == RuleNumber)
        file << "Num(" << value << ')';
    else if (operatorID == RuleVariable)
        file << "Var(" << value << ')';
    else
        file << variable;
}

void Token::setToken(char op)
{
    int opID;
    isOperator = 0;

    if (op == '+')
        opID = Plus;
    else if (op == '-')
        opID = Minus;
    else if (op == '*')
        opID = Multiply;
    else if (op == '/')
        opID = Divide;
    else if (op == '(')
        opID = OpenP;
    else if (op == ')')
        opID = CloseP;
    else if (op == '^')
        opID = Exponent;
    else if (op == 'U')
        opID = UnaryM;
    else if (op == 'D')
        opID = Derivative;
    else if (op == 'M')
        opID = Error;

    operatorID = opID;

    if (opID != NullToken || opID != Error)
        isOperator = 1;
}

int Token::equals(Token t)
{
    if ((isOperator && t.isOperator && operatorID == t.operatorID) ||
                                                        //Both Operators
        (operatorID == Variable && t.operatorID == Variable &&
         variable == t.variable) ||                     //Both Variables
```

```
      (operatorID == Number && t.operatorID == Number && value == t.value))
                                                          //Both Numbers
        return 1;
    return 0;
}

//Class Solver functions

//Initializes any character array for size t
void Solver::initialize(char array[])
{
    for (int i = 0; i < t; ++i)
        array[i] = '\0';
}

//Necessary initializations required when loading rules
void Solver::initialize()
{
    initialize(rn);
    initialize(rv);

    for (int i = 0; i < t; ++i)
    {
        initialize(ruleArr[i]);
        initialize(replaceArr[i]);
    }
}

//Initializes token array only upto the given length. Used only by toInfix()
void Solver::clearRow(Token array[], int l)
{
    for (int i = 0; i <= l; ++i)
        array[i].initialization();
}

//Shifts the contents of a token array. Used only by toInfix()
void Solver::shiftRow(Token array[], int l)
{
    for (int i = l; i > 0; --i)
        array[i] = array[i - 1];
}

//Return the order of precedence of a given operator
int Solver::getOrder(int operatorID)
{
    switch (operatorID)
    {
    case UnaryM:
    case Derivative:
        return 6;
    case Exponent:
        return 5;
    case Divide:
        return 2;
```

```
    case Multiply:
        return 2;
    case Plus:
        return 1;
    case Minus:
        return 1;
    case OpenP:
        return -1;
    default:
        return 0;
    }
}

//Function to return if the operator is Left Associative
int Solver::isLeftAssociative(int c)
{
    if ((c == Plus) || (c == Minus) || (c == Multiply) || (c == Divide))
        return 1;

    if (c == Exponent)
        return -1;

    return 0;
}

//Function to compare the precendence of the given operators.
//Returns 1 if a has higher precedence than b, -1 otherwise.
//Returns 0 if both the operators has the same precedence.
int Solver::precedence(int a, int b)
{
    int x = getOrder(a);
    int y = getOrder(b);

    if (x > y)
        return 1;
    else if (y > x)
        return -1;
    return 0;
}

//Returns 1 if thecharacter is a letter variable or a digit
int Solver::isConstant(char c)
{
    if (c >= '0' && c <= '9')
        return 1;
    if (c >= 'a' && c <= 'z')
        return 1;
    return 0;
}

//Returns 1 if the character is a digit
int Solver::isDigit(char c)
{
    if (c >= '0' && c <= '9')
```

```
        return 1;
    return 0;
}

//Returns the length of a token array
int Solver::length(Token array[])
{
    int l = 0;
    for (int i = 0; i < t; ++i)
    {
        if (array[i].operatorID == NullToken)
            return l;
        else
            ++l;
    }
    return t;
}

//Returns the length of a character array
int Solver::length(char array[])
{
    int l = 0;
    for (int i = 0; i < t; ++i)
    {
        if (array[i] == '\0' || array[i] == ' ' || array[i] == '\n')
            return l;
        ++l;
    }
    return t;
}

//Copies the contents of one token array to another
void Solver::copyArray(Token a[], Token b[])
{
    for (int i = 0; i < t; ++i)
        a[i] = b[i];
}

//Copies the contents of one character array to another
void Solver::copyChar(char a[], char b[])
{
    int i;
    for (i = 0; i < t && b[i] != '\0'; ++i)
        a[i] = b[i];
    for (; i < t; ++i)
        a[i] = '\0';
}

//Converts character infix expression to token postfix expression
void Solver::toPostfix(char array[], int length)
{
    int i = 0, j = 0, l = 0;

    Stack operators;
```

```
//Stores converted array of characters to array of tokens
Token infix[t];

for (i = 0; i < length; ++i)
{
    //Analysing a number
    if (isDigit(array[i]))
    {
        Token t;
        int v = 0, i2 = 0;

        do
        {
            v *= 10;
            v += (int)(array[i + i2] - 48);
            ++i2;
        } while (isDigit(array[i + i2]));

        t.setValue(v);
        infix[l++] = t;
        i += (i2 - 1);
    }

    //Analysing a variable
    else if (isConstant(array[i]) && !isDigit(array[i]))
    {
        Token t;
        t.setVariableName(array[i]);
        infix[l++] = t;
    }


    //Analyzing Rule
    else if (array[i] == 'V' || array[i] == 'N')
    {
        Token t;
        t.isOperator = 0;

        if (i != length - 1 && array[i + 1] == '`')
        {
            if (array[i] == 'V')
                t.operatorID = RuleVariable;
            else if (array[i] == 'N')
                t.operatorID = RuleNumber;

            t.value = array[i + 2] - 48;
        }
        i += 2;
        infix[l++] = t;
    }
```

```
     //Resolving Unary minus apart from Binary Subtraction
    else if (array[i] == '-' && ((i == 0) ||
             !(isConstant(array[i - 1]) || array[i - 1] == ')')))
    {
        Token t('U');
        infix[l++] = t;
    }

    //Remaining operators are directly pushed
    else
    {
        Token t(array[i]);
        infix[l++] = t;
    }
}
//Array of chars converted to array of tokens

//Converting infix to postfix
//Using Dijstra's Shunting Yard Algorithm
for (i = 0; i < l; ++i)
{

    //Non-Operators
    if (!infix[i].isOperator)
        postfix[j++] = infix[i];

    else
    {
        Token op = infix[i];

        //Open parentheses
        if (op.operatorID == OpenP)
            operators.push(op);

        //Close parentheses
        else if (op.operatorID == CloseP)
        {
            Token o = operators.pop();
            while (o.operatorID != OpenP)
            {
                postfix[j++] = o;
                o = operators.pop();
            }
        }

        //Pushing the first operator in the stack without any conditions
        //Removing an open parentheses
        else if (operators.total == 0 ||
                 operators.see().operatorID == OpenP)
            operators.push(op);

        //Pushing the operator when LP-HP situation occurs
        //Pushing the operator when HP-LP situation occurs and
        //the operator is right associative
```

```
            else if ((precedence(op.operatorID,
                                 operators.see().operatorID) == 1) ||
                     (precedence(op.operatorID,
                                 operators.see().operatorID) == 0 &&
                      isLeftAssociative(op.operatorID) == -1))
                operators.push(op);

            //Emptying the stack to output expression if none above
            //condition applies
            else
            {
                while ((precedence(operators.see().operatorID,
                                   op.operatorID) == 1) ||
                       (precedence(operators.see().operatorID,
                                   op.operatorID) == 0 &&
                        isLeftAssociative(op.operatorID) == 1))
                    postfix[j++] = operators.pop();
                operators.push(op);
            }
        }
    }

    //Adds all of the remaning operators to the output expression
    while (operators.total > 0)
        postfix[j++] = operators.pop();

    //Terminates with a null token
    Token blank;
    postfix[j] = blank;
}

//Converts postfix token array to infix token array.
void Solver::toInfix(Token token[], Token infix[], int &l)
{
    Token operand[t][t];
    int j = 0, k, i, c;

    for (int a = 0; a < t; ++a)
        for (int b = 0; b < t; ++b)
            operand[a][b].initialization();


    //Converting form postfix to infix
    for (i = 0; i < l && token[i].operatorID != NullToken; ++i)
    {
        if (token[i].isOperator)
        {
            //Determining the lengths of the two operands
            int l1, l2;
            for (l1 = 0; operand[j - 1][l1].operatorID != NullToken; ++l1)
                ;
            for (l2 = 0; operand[j - 2][l2].operatorID != NullToken; ++l2)
                ;
```

```
//Plus and minus operators
if (token[i].operatorID == Plus || token[i].operatorID == Minus)
{
    Token o;
    if (token[i].operatorID == Plus)
        o.setToken('+');
    else if (token[i].operatorID == Minus)
        o.setToken('-');

    //Placing the + or - operator in front of the first operand
    operand[j - 2][l2] = o;
    ++l2;

    //Copying the second operand in front of the first operand
    for (k = 0; k < l1; ++k, ++l2)
        operand[j - 2][l2] = operand[j - 1][k];

    //Deleting the old copy of second operand
    clearRow(operand[j - 1], l1);
    --j;
}

//Unary Minus
else if (token[i].operatorID == UnaryM)
{
    Token o;
    o.setToken('U');
    Token p, c;
    p.setToken('(');
    c.setToken(')');

    shiftRow(operand[j - 1], l1);
    operand[j - 1][l1 + 1] = c;
    operand[j - 1][0] = p;
    l1 += 3;
    shiftRow(operand[j - 1], l1);
    operand[j - 1][0] = o;
    ++l1;
}

//Differentiation
else if (token[i].operatorID == Derivative)
{
    Token d;
    Token p, c;
    p.setToken('(');
    c.setToken(')');
    d.setToken('D');

    shiftRow(operand[j - 2], l2);
    operand[j - 2][0] = p;
    operand[j - 2][l2 + 1] = c;
    l2 += 2;
```

```
        if (l1 > 1)
        {
            shiftRow(operand[j - 1], l1);
            operand[j - 1][0] = p;
            operand[j - 1][l1 + 1] = c;
            l1 += 2;
        }

        shiftRow(operand[j - 1], l1);
        operand[j - 1][0] = d;
        ++l1;

        for (k = 0; k < l1; ++k, ++l2)
            shiftRow(operand[j - 2], l2);
        for (k = 0; k < l1; ++k)
            operand[j - 2][k] = operand[j - 1][k];

        --j;
        clearRow(operand[j], l1);
    }
    else
    {
        Token p, c, o;
        p.setToken('(');
        c.setToken(')');
        o = token[i];

        //Placing parentheses if either of the two operands
        //are expressions in themselves
        if (l1 > 1)
        {
            shiftRow(operand[j - 1], l1);
            operand[j - 1][l1 + 1] = c;
            operand[j - 1][0] = p;
            l1 += 2;
        }
        if (l2 > 1)
        {
            shiftRow(operand[j - 2], l2);
            operand[j - 2][l2 + 1] = c;
            operand[j - 2][0] = p;
            l2 += 2;
        }

        operand[j - 2][l2] = o;
        ++l2;

        for (k = 0; k < l1; ++k)
        {
            operand[j - 2][l2] = operand[j - 1][k];
            ++l2;
        }
        clearRow(operand[j - 1], l1);
        --j;
```

```
            }
        }

        //If token is not an operator:
        else
            operand[j++][0] = token[i];
    }

    l = 0;

    //Copying the expression from the stack to the infix array.
    for (c = 0; operand[0][c].operatorID != NullToken; ++c, ++l)
        infix[c] = operand[0][c];
    for (; c < t; ++c)
        infix[c].initialization();
}

//Converts infix token array to infix character array
char *Solver::fixInfix(Token token[], int l, char array[])
{
    int i, j, f;
    for (i = 0; i < 3 * t; ++i)
        array[i] = '\0';

    fstream file("temp1.txt", ios::in | ios::out);

    file.seekp(0);
    for (i = 0; i < l; ++i)
    {
        token[i].print(file);
        if (token[i].operatorID != Derivative)
            file << ' ';
    }
    file << 'Q';

    file.seekg(0);
    file.get(array, 3 * t, 'Q');
    file.close();
    remove("temp1.txt");
    return array;
}

//Fixes the infix character expression before it is converted to postfix.
char *Solver::fixExpression(char array[])
{

    int i, j;

    for (i = 0; array[i] != '\0' && i < t; ++i)
    {
        if (array[i] == ' ')
        {
            //Removing spaces
            for (j = i + 1; j < t; ++j)
```

```
            array[j - 1] = array[j];
        --i;
    }
}

for (i = 0; array[i] != '\0' && i < t; ++i)
{
    //Changing opening bracket
    if (array[i] == '{' || array[i] == '[')
        array[i] = '(';

    //Changing closing brackets
    else if (array[i] == '}' || array[i] == ']')
        array[i] = ')';

    //For Differentiation
    else if (array[i] == 'd' && array[i + 1] == '/' &&
             array[i + 2] == 'd')
    {
        int prev_i = i;
        i += 3;
        int a = 0, b = 0, c = 0, finished = 0;
        char var[t];
        char exp[t];

        //Extracting the variable of differentiation
        do
        {
            var[a] = array[i];
            if (var[a] == '(')
                ++finished;
            else if (var[a] == ')')
                --finished;
            ++a;
            ++i;
        } while (finished > 0);
        var[a] = '\0';

        //Extracting the operand being differentiated
        finished = 0;
        do
        {
            exp[b] = array[i];
            if (exp[b] == '(')
                ++finished;
            else if (exp[b] == ')')
                --finished;
            ++b;
            ++i;
        } while (finished > 0);
        exp[b] = '\0';

        //Making new array with fixed derivative
        char arrayFixed[t];
```

```
            int d = 0;
            for (c = 0; c < prev_i; ++c, ++d)
                arrayFixed[d] = array[c];
            for (c = 0; c < b; ++c, ++d)
                arrayFixed[d] = exp[c];
            arrayFixed[d] = 'D';
            ++d;
            for (c = 0; c < a; ++c, ++d)
                arrayFixed[d] = var[c];
            for (c = i; array[c] != '\0'; ++c, ++d)
                arrayFixed[d] = array[c];
            arrayFixed[d] = '\0';

            //Copying the fixed array to the passed argument
            for (c = 0; c <= d; ++c)
                array[c] = arrayFixed[c];

            i = prev_i;
        }

        else if ((i > 0) &&
                 ((array[i] == '(' && isConstant(array[i - 1]) == 1) ||
                  (array[i] == '(' && isDigit(array[i - 1]) == 1) ||
                  (array[i - 1] == ')' && isConstant(array[i]) == 1) ||
                  (array[i] == '(' && isDigit(array[i - 1]) == 1) ||
                  (array[i - 1] == ')' && array[i] == '(') ||
                  ((isConstant(array[i]) && isConstant(array[i - 1])) &&
                  !(isDigit(array[i]) && isDigit(array[i - 1])))))
        {
            //Inserting ignored multiplication signs
            for (int j = t - 1; j >= i; --j)
                array[j] = array[j - 1];
            array[i] = '*';
            ++i;
        }
    }
    return array;
}

//Renumbering the rules numbers and rule variables
void Solver::fixNumbering(char array[], int isReplace)
{
    int i, j;

    //If the expression is for ruleArr and not for replaceArr
    if (!isReplace)
    {
        //Initialization
        for (i = 0; i < t; ++i)
            rn[i] = rv[i] = '\0';

        for (i = 2; array[i] != '\0' && i < t; ++i)
        {
            if (isDigit(array[i]) && array[i - 1] == '`')
```

```
            {
                if (array[i - 2] == 'N')
                {
                    //Search if the digit is in rn.
                    char n = array[i];
                    int found = 0;

                    for (j = 0; rn[j] != '\0' && j < t; ++j)
                        if (rn[j] == n)
                            found = 1;
                    if (!found)
                        rn[j] = n;
                }

                else if (array[i - 2] == 'V')
                {
                    //Search if the digit is in rv.
                    char n = array[i];
                    int found = 0;

                    for (j = 0; rv[j] != '\0' && j < t; ++j)
                        if (rv[j] == n)
                            found = 1;
                    if (!found)
                        rv[j] = n;
                }
            }
        }
    }

    for (i = 2; array[i] != '\0' && i < t; ++i)
    {
        if (isDigit(array[i]) && array[i - 1] == '`')
        {
            if (array[i - 2] == 'N')
            {
                for (j = 0; rn[j] != '\0' && j < t; ++j)
                    if (rn[j] == array[i])
                        array[i] = j + 48;
            }
            else if (array[i - 2] == 'V')
            {
                for (j = 0; rv[j] != '\0' && j < t; ++j)
                    if (rv[j] == array[i])
                        array[i] = j + 48;
            }
        }
    }
}

//Loads all the rules into userRules
void Solver::loadAllRules()
{
    for (int i = 0; i < t; ++i)
```

```
    {

        fixExpression(ruleArr[i]);
        fixNumbering(ruleArr[i], 0);
        toPostfix(ruleArr[i], length(ruleArr[i]));
        copyArray(userRules[i].match, postfix);

        fixExpression(replaceArr[i]);
        fixNumbering(replaceArr[i], 1);
        toPostfix(replaceArr[i], length(replaceArr[i]));
        copyArray(userRules[i].replace, postfix);

        copyChar(userRules[i].ruleName, nameArr[i]);
    }
}

//Simplifies the given expression
void Solver::solve(ostream &file, char ques[], int fromFile = 0)
{
    initialize();
    fixExpression(ques);
    toPostfix((fixExpression(ques)), length(ques));

    int l, i;
    l = length(postfix);

    //Displaying the quesiton after fixing it
    int len = l;
    Token infix[t];
    toInfix(postfix, infix, len);
    char infixed[3 * t];
    fixInfix(infix, len, infixed);

    file << "\nQ.  " << infixed;

    int at = 0;
    int r = 1;

    for (i = 0; i < t; ++i)
        ruleArray[i] = postfix[i];

    r = applicableRules(ruleArray, l, at);

    int nextPage = 0;
    for (i = 0; i < (t * 5) && r != NoRule; ++i)
    {
        file << "\n\n   ";
        if (r < t)
            file << userRules[r].ruleName;
        else if (r == NumericAdd)
            file << "Adding Numbers";
        else if (r == NumericSub)
            file << "Subtracting Numbers";
        else if (r == NumericMul)
```

```
            file << "Multiplying Numbers";
        else if (r == NumericDiv)
            file << "Reducing the Fraction";
        else if (r == NumericExpo)
            file << "Applying Exponent on Numbers";
        else if (r == NegateNum)
            file << "Applying Unary Minus";
        else if (r == DConstant)
            file << "Derivative of a Constant";
        file << endl;

        applyRule(r, at, l);

        len = l;
        toInfix(ruleArray, infix, len);

        file << "=   " << fixInfix(infix, len, infixed);

        ++nextPage;
        if (!fromFile && nextPage > 5)
        {
            cout << "\n\nPress anything to continue...";
            getch();
            nextPage = 0;
        }

        r = applicableRules(ruleArray, l, at);
    }
    file << endl;
}

//Stores one operand from expression to the reference of Stack object
int Solver::getOperand(Stack &operand)
{
    int l = 0;
    for (int get = 1; get > 0 && expression.total >= 0; --get)
    {
        operand.push(expression.pop());
        if (operand.see().isOperator)
            if (operand.see().operatorID == UnaryM)
                ++get;
            else
                get += 2;
        ++l;
    }
    return l;
}

//Return the Highest Common Factor. Useful for reducing fraction to lowest
terms
int Solver::hcf(int a, int b)
{
    if (b == 0)
        return a;
```

```
    else if (b < 0)
        b = -b;
    return hcf(b, a % b);
}

//Applying the rule being passed to it to expression.
void Solver::applyRule(int rule, int arrayIndex, int &lenArray)
{
    if (rule == NoRule)
        return;

    expression.initialize();
    int eIndex = 0;
    int eLength = lenArray;

    Token one;
    Token del; //Token to store deleted values
    one.setValue(1);
    Token zero;
    zero.setValue(0);

    for (eIndex = 0; eIndex <= arrayIndex; ++eIndex)
        expression.push(ruleArray[eIndex]);

    //For addition, subtraction, multiplication, exponentiation
    if (rule == NumericAdd || rule == NumericSub || rule == NumericMul ||
        rule == NumericExpo)
    {
        Token o = expression.pop();
        Token n2 = expression.pop();
        Token n1 = expression.pop();

        double num = 0;

        if (o.operatorID == Plus)
            num = n1.value + n2.value;
        else if (o.operatorID == Minus)
            num = n1.value - n2.value;
        else if (o.operatorID == Multiply)
            num = n1.value * n2.value;
        else if (o.operatorID == Divide)
            num = n1.value / n2.value;
        else if (o.operatorID == Exponent)
            num = pow(n1.value, n2.value);

        n1.setValue(num);
        expression.push(n1);
        eLength -= 2;
    }

    //For division, in fraction form
    else if (rule == NumericDiv)
    {
        Token o = expression.pop();
```

```
    Token n2 = expression.pop();
    Token n1 = expression.pop();

    int bigger = 0, smaller = 0;
    if (n2.value > n1.value)
    {
        bigger = n2.value;
        smaller = n1.value;
    }
    else
    {
        bigger = n1.value;
        smaller = n2.value;
    }
    int divideBy = hcf(bigger, smaller);

    n1.setValue(n1.value / divideBy);
    n2.setValue(n2.value / divideBy);

    expression.push(n1);
    expression.push(n2);
    expression.push(o);
}

//Unary Minus
else if (rule == NegateNum)
{
    Token o = expression.pop();
    Token n = expression.pop();
    n.value = 0 - n.value;
    expression.push(n);
    eLength -= 1;
}

//Constant Derivative
else if (rule == DConstant)
{
    //Forgetting about derivative operator
    del = expression.pop();

    //Forgetting about the variable
    Stack removing;
    int l1 = 0;
    l1 = getOperand(removing);

    Stack operand;
    int l2 = 0;
    l2 = getOperand(operand);

    expression.push(zero);
    eLength -= l1 + l2;
}

//User-defined rules
```

```
else if (rule < t)
{
    Rule r = userRules[rule];

    Token ruleN[t];
    Stack ruleV[t];

    //Removing the match expression
    int l1 = length(r.match);
    int rIndex;
    for (rIndex = l1 - 1; rIndex >= 0; --rIndex)
    {
        if (r.match[rIndex].isOperator ||
            r.match[rIndex].operatorID == Number)
            del = expression.pop();
        else if (r.match[rIndex].operatorID == RuleNumber)
        {
            ruleN[r.match[rIndex].value].setValue(expression.pop().value);
        }
        else if (r.match[rIndex].operatorID == RuleVariable)
        {
            ruleV[r.match[rIndex].value].total = 0;
            int l = getOperand(ruleV[r.match[rIndex].value]);
            eLength -= l - 1;
        }
        --eLength;
    }

    //Replace the expression
    l1 = length(r.replace);
    for (rIndex = 0; rIndex < l1; ++rIndex)
    {
        if (r.replace[rIndex].isOperator ||
            r.replace[rIndex].operatorID == Number)
        {
            expression.push(r.replace[rIndex]);
        }
        else if (r.replace[rIndex].operatorID == RuleNumber)
        {
            Token n;
            n.setValue(ruleN[r.replace[rIndex].value].value);
            expression.push(n);
        }
        else if (r.replace[rIndex].operatorID == RuleVariable)
        {
            int b = r.replace[rIndex].value;
            int l = length(ruleV[b].array);
            int in = ruleV[b].total - 1;
            for (int i = 0; i < l; ++i)
            {
                expression.push(ruleV[b].see(in));
                --in;
            }
            eLength += l - 1;
```

```
            }
            ++eLength;
        }
    }

    //If the expression has bloated
    if (eLength >= t)
    {
        expression.initialize();
        Token tooMuch;
        tooMuch.setToken('M');
        expression.push(tooMuch);
        lenArray = 1;
    }

    //Copying the rest of ruleArray[] to expression
    else
    {

        for (eIndex = arrayIndex + 1; eIndex < lenArray; ++eIndex)
            expression.push(ruleArray[eIndex]);

        Token blank;
        expression.push(blank);
        lenArray = eLength;
    }

    for (int i = 0; i < expression.total; ++i)
    {
        ruleArray[i] = expression.array[i];
    }
}

int Solver::applicableRules(Token array[], int &len, int &at)
{
    int i, j, k;

    //Checking if arithmetic is to be done,
    //only if boolean rules are not loaded.
    if (!(tolower(fileName[0]) == 'b' || tolower(fileName[1]) == 'o' ||
          tolower(fileName[2]) == 'o' || tolower(fileName[3]) == 'l'))
        for (i = 0; i < len; ++i)
        {
            Token c = array[i];

            //UNary Minus
            if (c.operatorID == UnaryM && array[i - 1].operatorID == Number)
            {
                at = i;
                return NegateNum;
            }

            //Arithmetic
            if ((array[i - 2].operatorID == Number) &&
```

```
            (array[i - 1].operatorID == Number))
        {
            at = i;
            if (c.operatorID == Plus)
                return NumericAdd;
            if (c.operatorID == Minus)
                return NumericSub;
            if (c.operatorID == Multiply)
                return NumericMul;
            if (c.operatorID == Exponent)
                return NumericExpo;

            int bigger = 0, smaller = 0;
            if (array[i - 2].value > array[i - 1].value)
            {
                bigger = array[i - 2].value;
                smaller = array[i - 1].value;
            }
            else
            {
                bigger = array[i - 1].value;
                smaller = array[i - 2].value;
            }
            int divideBy = hcf(bigger, smaller);

            //Division, only if numerator is divisible by the denominator
            if (c.operatorID == Divide && divideBy > 1)
                return NumericDiv;
        }

        //Derivative
        if (c.operatorID == Derivative)
        {
            at = i;
            int index = i - 2;
            char var = array[i - 1].variable;
            int foundVar = 0;

            for (int j = 1; j > 0 && index >= 0; --j)
            {
                if (array[index].isOperator)
                    j += 2;
                else if (array[index].operatorID == Variable &&
                         array[index].variable == var)
                    foundVar = 1;
                --index;
            }

            if (!foundVar)
                return DConstant;
        }
    }

//Checking if user rules applies
```

```
//For each rule...
for (i = 0; i < t; ++i)
{

    //Do this... (if the rule exists)
    Rule r = userRules[i];
    if (r.match[0].operatorID != NullToken)
    {

        int l = length(r.match);

        //For each Token in the array received...
        for (j = len - 1; j >= l - 1; --j)
        {
            at = j;
            int index = j + 1;
            int apply = 1;

            int nums[t];
            Token vars[t][t];

            //Initialization
            for (k = 0; k < t; ++k)
            {
                nums[k] = s;
                for (int m = 0; m < t; ++m)
                    vars[k][m].initialization();
            }

            //For each Token in the rule...
            for (k = l - 1; k >= 0; --k)
            {
                --index;

                //Checking if both rule and expression have...
                if ((r.match[k].operatorID == Number &&
                     array[index].operatorID == Number &&       //Number
                     r.match[k].value == array[index].value) || //Operator
                    (r.match[k].isOperator && array[index].isOperator &&
                     r.match[k].operatorID == array[index].operatorID))
                    continue;

                //Number placeholder
                else if (r.match[k].operatorID == RuleNumber &&
                         array[index].operatorID == Number)
                {
                    int b = r.match[k].value;
                    if (nums[b] == s)
                    {
                        nums[b] = array[index].value;
                        continue;
                    }
                    else if (nums[b] == array[index].value)
                        continue;
```

```
                    }

                    //Variable placeholder
                    else if (r.match[k].operatorID == RuleVariable)
                    {
                        int b = r.match[k].value;
                        Stack operand;

                        //Extracting the operand in reverse
                        for (int get = 1; get > 0 && index >= 0; --get)
                        {
                            operand.push(array[index]);
                            --index;
                            if (operand.see().isOperator)
                                if (operand.see().operatorID == UnaryM)
                                    ++get;
                                else
                                    get += 2;
                        }
                        ++index;

                        if (vars[b][0].operatorID == NullToken)
                        {
                            copyArray(vars[b], operand.array);
                            continue;
                        }
                        else
                        {
                            int eq = 1, p;
                            for (p = 0; p < operand.total; ++p)
                                if (!(operand.array[p].equals(vars[b][p])))
                                    eq = 0;

                            if (vars[b][p].operatorID != NullToken)
                                eq = 0;

                            if (eq)
                                continue;
                        }
                    }

                    apply = 0;
                    break;
                }

                if (apply)
                    return i;
            }
        }
    }

    return NoRule;
}
```

```
//---------------------------USER-INTERFACE-----------------------------//

//User Instructions
void addRules(Solver &solver);
void deleteRules(Solver &solver);
void solveFile(Solver &solver);
void solveQ(Solver &solver);
void help();

//Associated Functions
void printLine(ostream &file);
void setFile(Solver &solver);
void loadRules(Solver &solver);
void wait(int);

//Prints a horizontal line onto the screen or file
void printLine(ostream &file)
{
    file << '\n';
    char c = ((file == cout) ? ((char)196) : '-');
    for (int i = 0; i < 80; ++i)
        file << c;
}

//Sets the rulebook
void setFile(Solver &solver)
{
    char nameFile[14] = "ruleBook.txt\0";
    for (int i = 0; i < 13; ++i)
        solver.fileName[i] = nameFile[i];

    ifstream file("ruleBook.txt");

    if (!file)
    {
        printLine(cout);

        do
        {
            cout << "\n\tError in loading " << solver.fileName << endl;
            cout << "\tRe-enter File Name: ";
            cin.getline(solver.fileName, t);

            int l;
            for (l = 0; solver.fileName[l] != '\0'; ++l)
                ;

            if (solver.fileName[l - 4] != '.')
            {
                solver.fileName[l] = '.';
                solver.fileName[l + 1] = 't';
                solver.fileName[l + 2] = 'x';
                solver.fileName[l + 3] = 't';
                solver.fileName[l + 4] = '\0';
```

```
                l += 4;
            }

            file.open(solver.fileName);
        } while (!file);

        printLine(cout);
    }

    file.close();
}

//Adds a rule to the rulebook
void addRules(Solver &solver)
{
    printLine(cout);

    char lhs[t];
    char rhs[t];
    char n[t];

    cout << "\n\tRule Name: ";
    cin.getline(n, t);

    while (n[0] == '\0' || n[0] == '\n')
        cin.getline(n, t);

    cout << "\tLHS: ";
    cin.getline(lhs, t);
    cout << "\tRHS: ";
    cin.getline(rhs, t);

    ofstream file(solver.fileName, ios::app);

    file << "\nRule: " << n << '\n'
         << lhs << '\n'
         << rhs << '\n';

    cout << "\n\t'" << n << "' added.\n";

    printLine(cout);
    loadRules(solver);
    file.close();
}

//Loads all rules from the rulebook to the userRules[]
void loadRules(Solver &solver)
{
    ifstream file(solver.fileName, ios::nocreate);

    solver.initialize();
    if (!file)
    {
        printLine(cout);
```

```
        cout << "\n\t" << solver.fileName << " does not exixts. "
             << "Reload ruleBook.\n\n";
        printLine(cout);
    }
    else
    {
        char line[t];
        int i = 0;

        int feedLHS = 0, feedRHS = 0;

        while (!file.eof())
        {
            solver.initialize(line);
            file.getline(line, t);

            if ((tolower(line[0]) == 'r') &&
                (tolower(line[1]) == 'u') &&
                (tolower(line[2]) == 'l') &&
                (tolower(line[3]) == 'e'))
            {
                solver.initialize(solver.nameArr[i]);
                for (int j = 6; line[j] != '\0'; ++j)
                    solver.nameArr[i][j - 6] = line[j];
                feedLHS = 1;
            }
            else if (feedLHS)
            {
                solver.copyChar(solver.ruleArr[i], line);
                feedRHS = 1;
                feedLHS = 0;
            }
            else if (feedRHS)
            {
                solver.copyChar(solver.replaceArr[i], line);
                feedRHS = 0;
                ++i;
            }
        }
    }

    solver.loadAllRules();
    file.close();
}

//Deletes a rule from the rulebook and refreshes userRules[] by
//reloading rulebook.
void deleteRule(Solver &solver)
{
    printLine(cout);
    loadRules(solver);
    cout << "\n\tLoaded rules: \n\n";
    int del = 0, i, nextPage = 0;
```

```
for (i = 0;
     i < t && solver.userRules[i].match[0].operatorID != NullToken;
     ++i, ++nextPage)
{
    cout << "\t" << i + 1 << ".\t"
         << solver.userRules[i].ruleName << endl;
    if (nextPage > 18)
    {
        nextPage = 0;
        cout << "\n\t More rules ahead. Press anything to "
             << "continue...\n\n";
        getch();
    }
}

cout << "\n\n\tEnter Rule No. to forget: ";
cin >> del;

ifstream file(solver.fileName);
ofstream temp("temp.txt");

i = 0;
while (!file.eof())
{
    char line[t];
    for (int a = 0; a < t; ++a, line[a] = '\0')
        ;

    file.getline(line, t, '\n');
    if ((tolower(line[0]) == 'r') &&
        (tolower(line[1]) == 'u') &&
        (tolower(line[2]) == 'l') &&
        (tolower(line[3]) == 'e'))
        ++i;

    if (i != del)
    {
        temp << line << endl;
    }
    else
    {
        solver.userRules[i - 1].match[0].operatorID = NullToken;
        solver.userRules[i - 1].replace[0].operatorID = NullToken;
        solver.userRules[i - 1].ruleName[0] = '\0';
    }
}

file.close();
temp.close();

cout << "\tRule " << del << " forgotten.\n";
printLine(cout);
```

```
    remove(solver.fileName);
    rename("temp.txt", solver.fileName);
    loadRules(solver);
}

//Solves a question on-screen
void solveQ(Solver &solver)
{
    printLine(cout);
    cout << "\n   Simplify: ";
    char ques[t];
    cin.getline(ques, t);
    solver.solve(cout, ques, 0);
    printLine(cout);
}

//Displays help
void help()
{
    printLine(cout);

    cout << " MathSolver is a C++ term re-writing program coded by"
         << " Utsav Munendra which\n can simplify mathematical"
         << " expressions based on the rules taught to it.\n"
         << " The following instructions can be given to this program:\n"
         << "\n   LEARN:  To teach a new rule to the program."
         << "\n   FORGET: To make the program forget about a rule."
         << "\n   SHOW:   To simplify an expression in the program."
         << "\n   SOLVE:  To simplify expressions from a file."
         << "\n   EXIT:   To end the program."
         << "\n\n When entering a expression, the character set is "
         << "a-z for variables and 0-9 for\n numbers. When entering "
         << "rules, enter numbers like N`1 and variables like V`1, \n "
         << "where the the N and V are seperated by ` and are followed "
         << "by a number. In a \n rule expression, when two numbers or "
         << "variables have the same number, they will\n be checked for "
         << "equality to verify the applicability of the rule.\n\n Allowed "
         << "operators are: + - * / () ^ d/dx \n NOTE 1: If the fileName"
         << " for rulebook startes with 'bool', then bool mode is\n        "
         << " activated and arithmetic operations are ignored.\n"
         << " NOTE 2: In bool mode, + is OR, * is AND and - is NOT.";

    printLine(cout);
}

//Solves an assignment file
void solveFile(Solver &solver)
{
    printLine(cout);

    cout << "\n\tAssignment File: ";
    char assignment[t];
    cin.getline(assignment, t);
```

```
    int l;
    for (l = 0; assignment[l] != '\0'; ++l)
        ;

    if (assignment[l - 4] != '.')
    {
        assignment[l] = '.';
        assignment[l + 1] = 't';
        assignment[l + 2] = 'x';
        assignment[l + 3] = 't';
        assignment[l + 4] = '\0';
        l += 4;
    }

    ifstream file(assignment, ios::nocreate);

    if (!file)
    {
        cout << "\n\tCannot find " << assignment << ". Retry\n\n";
        printLine(cout);
        return;
    }

    ofstream answers("Answers.txt", ios::app);

    while (!file.eof())
    {
        char question[t + 3];
        file.getline(question, t + 3);

        if (question[0] == 'Q' && question[1] == '.')
        {
            char q[t];
            for (int i = 0; i < t; ++i)
                q[i] = question[i + 3];
            printLine(answers);
            solver.solve(answers, q, 1);
            printLine(answers);
        }
    }

    cout << "\n\t" << assignment << " solved in Answers.txt" << endl;
    file.close();
    answers.close();

    printLine(cout);
}

//Inputs any character and displays it on screen.
//Useful as instructions can be determined by the first
//two characters only. So, the next characters are dumped here.
void wait(int i)
{
    char c;
```

```
    for (int j = 0; j < i; ++j)
    {
        c = getch();
        cout << c;
    }
}

enum menu
{
    HOME,
    HELP,
    SHOW,
    SOLVE,
    LEARN,
    FORGET
};

void printHeader(menu m = HOME, int newScreen = 0, int starting = 0, int
helpMenu = 0)
{
    if (newScreen)
        clrscr();

    int x = wherex();
    int y = wherey();

    gotoxy(1, 1);

    if (helpMenu)
        gotoxy(1, 2);

    textbackground(7);
    textcolor(RED);
    cprintf("  ");
    cprintf("                              ");
    textcolor(0);
    textbackground(14);

    if (m == HOME)
        cprintf("  The MathSolver Program  ");
    else if (m == HELP)
        cprintf("     About the Program    ");
    else if (m == SHOW)
        cprintf("  Simplifying in Program  ");
    else if (m == SOLVE)
        cprintf("    Solving Assignment    ");
    else if (m == LEARN)
        cprintf("     Adding new Rule      ");
    else if (m == FORGET)
        cprintf("      Deleting a rule     ");
    textbackground(7);
    cprintf("                              ");
    textbackground(BLACK);
    textcolor(7);
```

```
    if (starting)
    {
        textcolor(LIGHTGRAY);
        textbackground(BLACK);
        cprintf("            ");
        cprintf("Instruction Set: learn, forget, show, solve, help, exit\n");
    }
    else
        gotoxy(x, y);
}

void main()
{
    printHeader(HOME, 1, 1);
    setFile(solver);
    loadRules(solver);

    char ins[2];

    //Instructions
    while (1)
    {
        cout << "\n > ";
        ins[0] = getch();
        printHeader(HOME, 1);
        cout << "\n\n\n > " << ins[0];
        ins[1] = getch();
        cout << ins[1];

        if (tolower(ins[0]) == 'l' && tolower(ins[1]) == 'e')
        {
            printHeader(LEARN);
            wait(3);
            addRules(solver);
        }
        else if (tolower(ins[0]) == 'f' && tolower(ins[1]) == 'o')
        {
            printHeader(FORGET);
            wait(4);
            deleteRule(solver);
        }
        else if (tolower(ins[0]) == 's' && tolower(ins[1]) == 'h')
        {
            printHeader(SHOW);
            wait(2);
            solveQ(solver);
        }
        else if (tolower(ins[0]) == 's' && tolower(ins[1]) == 'o')
        {
            printHeader(SOLVE);
            wait(3);
            solveFile(solver);
        }
```

```
        else if (tolower(ins[0]) == 'h' && tolower(ins[1]) == 'e')
        {
            printHeader(HELP);
            wait(2);
            help();
            printHeader(HELP, 0, 0, 1);
        }
        else if (tolower(ins[0]) == 'e' && tolower(ins[1]) == 'x')
        {
            wait(2);
            return;
        }

        else
            cout << "\nUnknown instruction. Retry.\n";

        ins[0] = ins[1] = '\0';
    }
}
```

OUTPUT

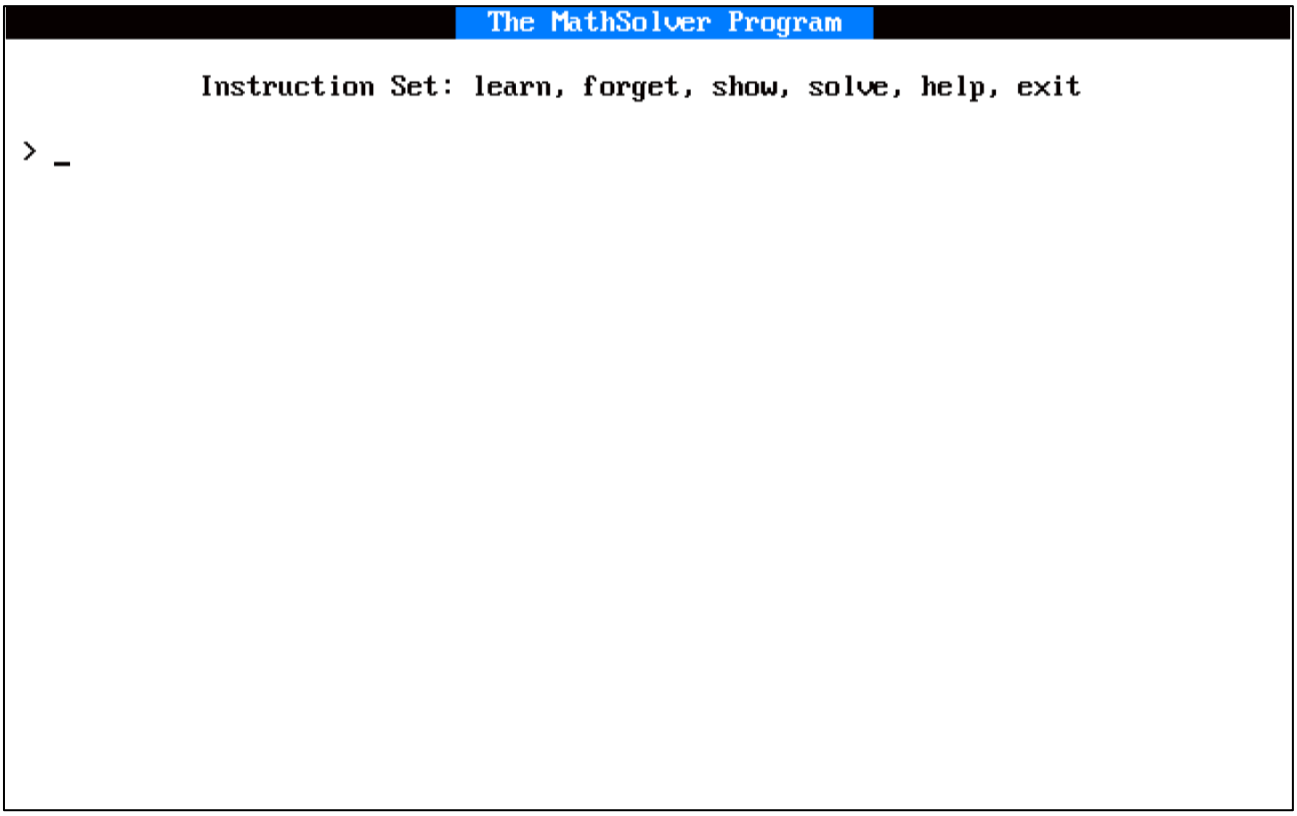


Figure 1: User Interface

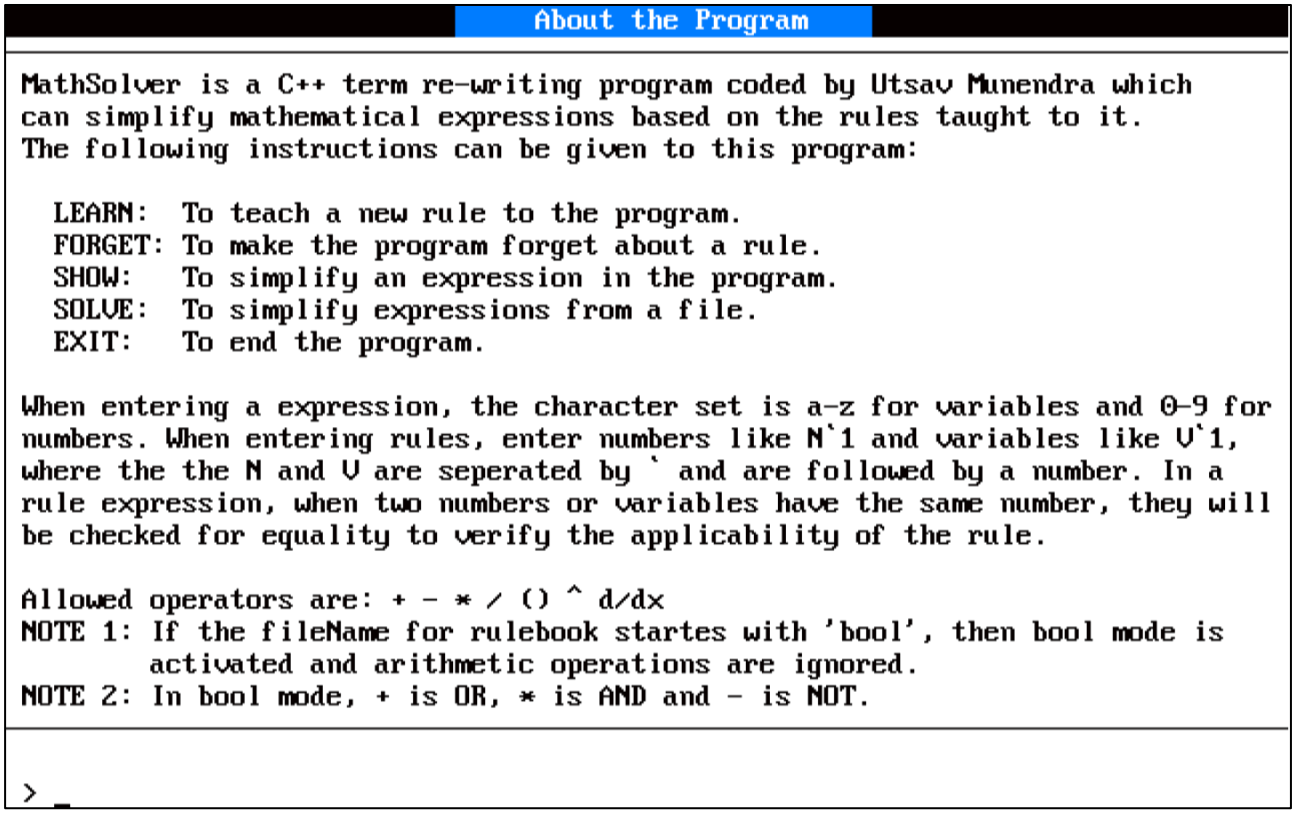


Figure 2: Help Screen

Figure 3:
Pre-loaded Rules

As displayed by the program during deletion.

```
1.      Zero Addition
2.      Zero Addition
3.      Zero Multiplication
4.      Zero Multiplication
5.      Multiplication by One
6.      Multiplication by One
7.      Zero Subtraction
8.      Division by One
9.      Division of Zero
10.     Division by itself
11.     Subtraction of Same Terms
12.     Adding to fractions
13.     Adding to fractions
14.     Subtracting to fractions
15.     Subtracting to fractions
16.     Multiplying to fractions
17.     Multiplying to fractions
18.     Dividing with fraction
19.     Division of fractions
20.     Adding Fractions

 More rules ahead. Press anything to continue...
```

Figure 4:
More Pre-Loaded Rules

```
 More rules ahead. Press anything to continue...

21.     Subtracting Fractions
22.     Multiplying Fractions
23.     Dividing Fractions
24.     Distributive Law over Addition
25.     Distributive Law over Subtraction
26.     Zero Exponent
27.     One Exponent
28.     Common Base of Exponents
29.     Common Base of Exponents
30.     Exponent of Exponents
31.     Common Exponents
32.     Grouping Like Terms
33.     Grouping Like Terms
34.     Multiplicative Associativity
35.     Since d(x)/dx = 1
36.     Derivative of a Sum
37.     Derivative of a Difference
38.     Leibnitz Product Rule
39.     Derivative Division Rule

 More rules ahead. Press anything to continue...
```

Figure 5:
Deleting a rule

Rule 41 has been deleted from the RuleBook. This will now be inserted again.

```
28.     Common Base of Exponents
29.     Common Base of Exponents
30.     Exponent of Exponents
31.     Common Exponents
32.     Grouping Like Terms
33.     Grouping Like Terms
34.     Multiplicative Associativity
35.     Since d(x)/dx = 1
36.     Derivative of a Sum
37.     Derivative of a Difference
38.     Leibnitz Product Rule
39.     Derivative Division Rule

 More rules ahead. Press anything to continue...

40.     Derivative of Exponents
41.     Multiplicative Associativity


Enter Rule No. to forget: 41
Rule 41 forgotten.

>
```

Figure 6:
Adding a rule

Rule for Multiplicative Associativity has been added.

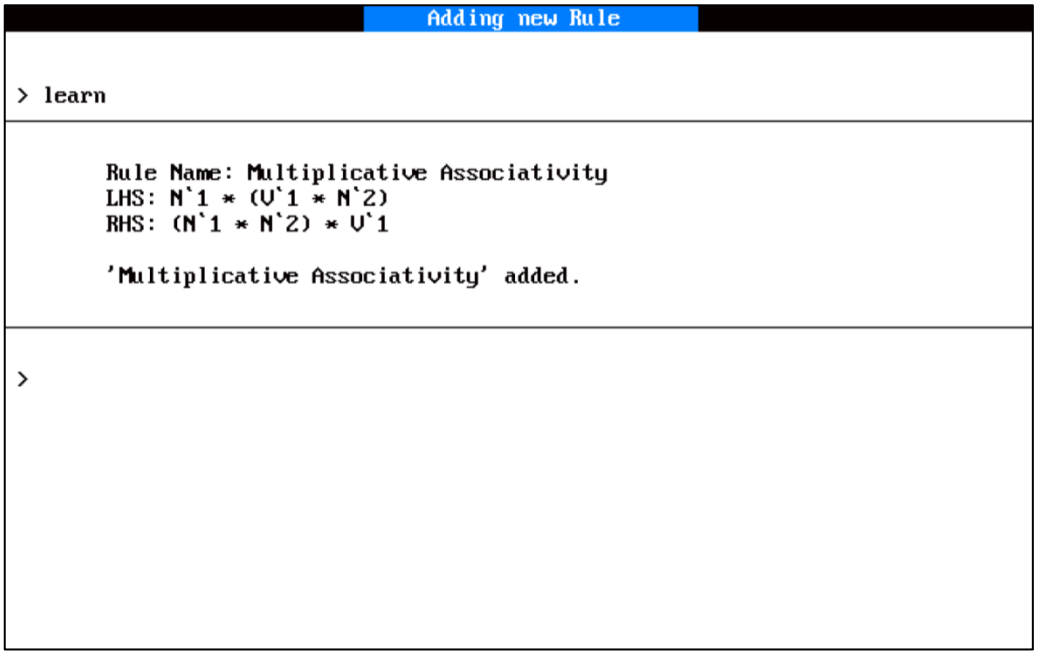


Figure 7:
Error in loading Rule Book

Program asks for the file which stores all the rules. Text file extension is automatically added.

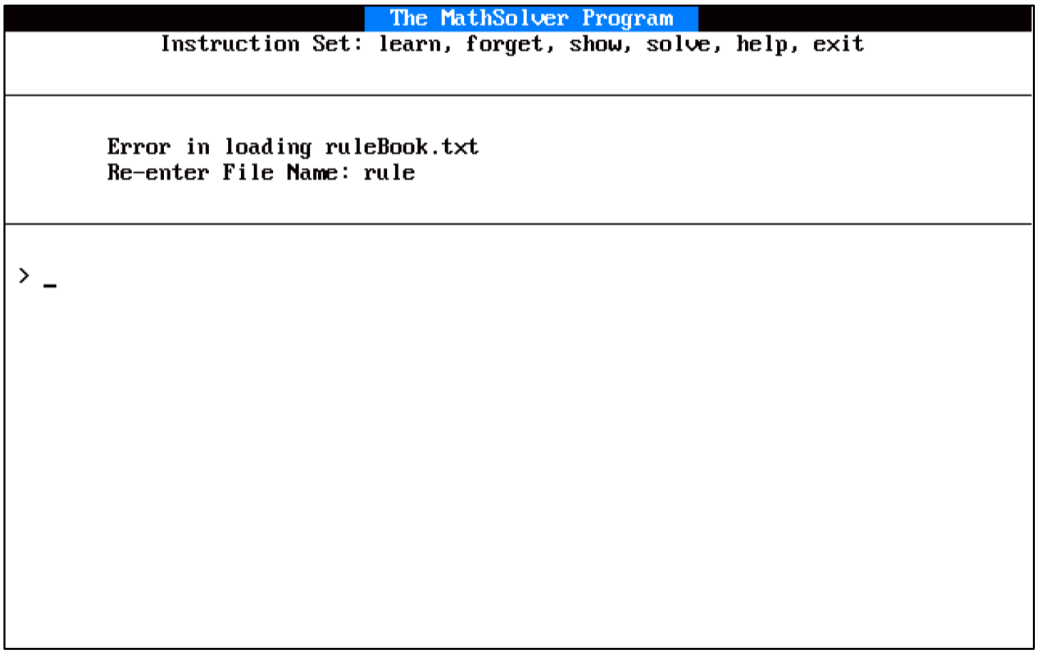


Figure 8:
Problems with commutative and associative operators

Adding these properties would result in an endless loop and without these, some simple expressions cannot be simplified.

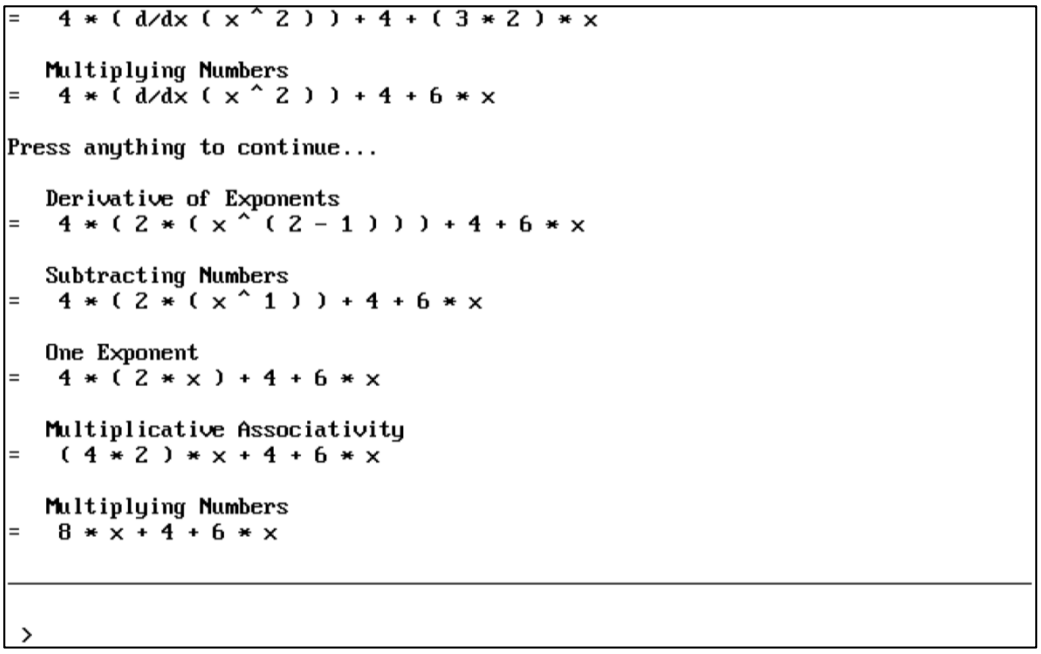


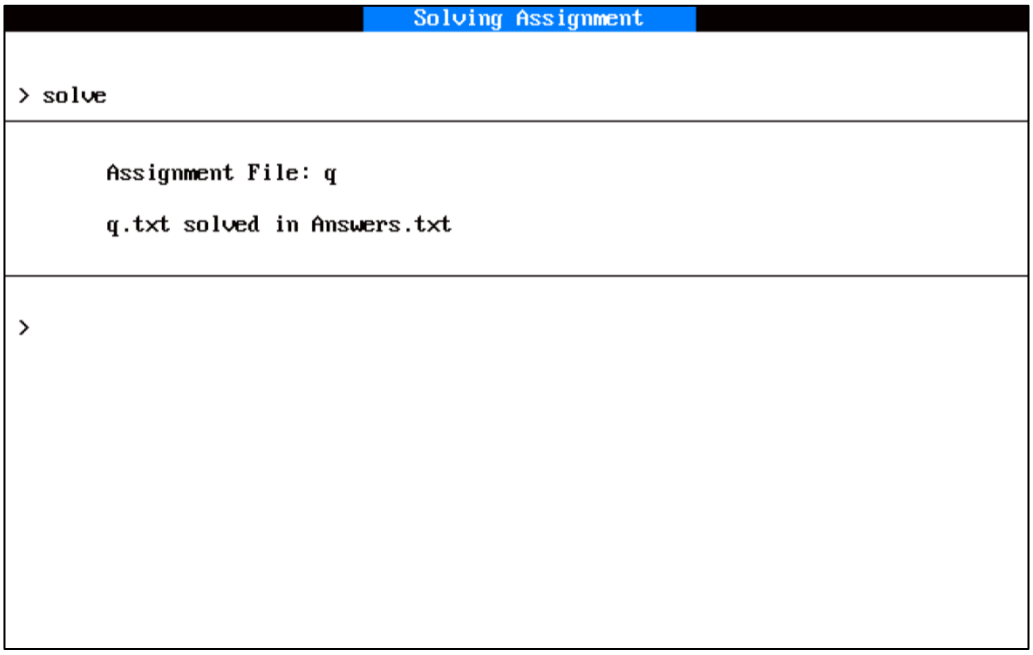
Figure 9: Simplifying an expression on screen

$$Solving\ \frac{d}{dx}\left(\frac{d}{dx}x^3\right) \qquad Ans: 6x$$

```
                    Simplifying in Program

 > show
 ──────────────────────────────────────────────────────────────────────────────

    Simplify: d/dx(d/dx(x^3))

Q.  d/dx ( d/dx ( x ^ 3 ) )

    Derivative of Exponents
=    d/dx ( 3 * ( x ^ ( 3 - 1 ) ) )

    Subtracting Numbers
=    d/dx ( 3 * ( x ^ 2 ) )

    Leibnitz Product Rule
=    ( d/dx ( 3 ) ) * ( x ^ 2 ) + 3 * ( d/dx ( x ^ 2 ) )

    Derivative of a Constant
=    0 * ( x ^ 2 ) + 3 * ( d/dx ( x ^ 2 ) )

    Zero Multiplication
=    0 + 3 * ( d/dx ( x ^ 2 ) )

    Zero Addition
=    3 * ( d/dx ( x ^ 2 ) )

    Derivative of Exponents
=    3 * ( 2 * ( x ^ ( 2 - 1 ) ) )

    Subtracting Numbers
=    3 * ( 2 * ( x ^ 1 ) )

    One Exponent
=    3 * ( 2 * x )

    Multiplicative Associativity
=    ( 3 * 2 ) * x

    Multiplying Numbers
=    6 * x

 ──────────────────────────────────────────────────────────────────────────────

 >
```

Figure 10:
Solving a file

Text file extension is automatically added.



File 1: **Q.txt:** *Contains the questions which are simplified by the above solve command.*

```
Questions from a file.
Only ones starting with Q. are considered.

Arithmetic Question
Q. 2+3-4*5/3*-4+5

Exponent Question
Q. (2*2)^x * 4^y * 8^(x+y)

Differentiation Question
Q. d/dx(x^2^2 + x^1^(2/5) + 45^0)
```

File 2: **Answers.txt**

```
--------------------------------------------------------------------------------
Q.  2 + 3 - ( ( 4 * 5 ) / 3 ) * ( - ( 4 ) ) + 5

   Adding Numbers
=   5 - ( ( 4 * 5 ) / 3 ) * ( - ( 4 ) ) + 5

   Multiplying Numbers
=   5 - ( 20 / 3 ) * ( - ( 4 ) ) + 5

   Applying Unary Minus
=   5 - ( 20 / 3 ) * -4 + 5

   Multiplying to fraction
=   5 - ( -4 * 20 ) / 3 + 5

   Multiplying Numbers
=   5 - -80 / 3 + 5
```

```
   Subtracting to fractions
=   ( 5 * 3 - -80 ) / 3 + 5

   Multiplying Numbers
=   ( 15 - -80 ) / 3 + 5

   Subtracting Numbers
=   95 / 3 + 5

   Adding to fractions
=   ( 5 * 3 + 95 ) / 3

   Multiplying Numbers
=   ( 15 + 95 ) / 3

   Adding Numbers
=   110 / 3
```

```
Q.  ( ( ( 2 * 2 ) ^ x ) * ( 4 ^ y ) ) * ( 8 ^ ( x + y ) )

   Multiplying Numbers
=   ( ( 4 ^ x ) * ( 4 ^ y ) ) * ( 8 ^ ( x + y ) )

   Common Base of Exponents
=   ( 4 ^ ( x + y ) ) * ( 8 ^ ( x + y ) )

   Common Exponents
=   ( 4 * 8 ) ^ ( x + y )

   Multiplying Numbers
=   32 ^ ( x + y )
```

```
Q.  d/dx ( x ^ ( 2 ^ 2 ) + x ^ ( 1 ^ ( 2 / 5 ) ) + 45 ^ 0 )

   Applying Exponent on Numbers
=   d/dx ( x ^ 4 + x ^ ( 1 ^ ( 2 / 5 ) ) + 45 ^ 0 )

   Applying Exponent on Numbers
=   d/dx ( x ^ 4 + x ^ ( 1 ^ ( 2 / 5 ) ) + 1 )

   Raised to power of 1
=   d/dx ( x ^ 4 + x ^ 1 + 1 )

   One Exponent
=   d/dx ( x ^ 4 + x + 1 )

   Derivative of a Sum
=   d/dx ( x ^ 4 + x ) + d/dx ( 1 )
```

```
   Derivative of a Constant
=   d/dx ( x ^ 4 + x ) + 0

   Zero Addition
=   d/dx ( x ^ 4 + x )

   Derivative of a Sum
=   d/dx ( x ^ 4 ) + d/dx ( x )

   Since d(x)/dx = 1
=   d/dx ( x ^ 4 ) + 1

   Derivative of Exponents
=   4 * ( x ^ ( 4 - 1 ) ) + 1

   Subtracting Numbers
=   4 * ( x ^ 3 ) + 1

--------------------------------------------------------------------------------
```

File 3: **Q.txt:** *Contains some Boolean algebra questions which are simplified by the rules in BoolRule.txt*

```
Boolean Algebra Questions.

Q. 0 + 1 + 0 + 0

Q. 0 * 1 * 0 * 0

Q. -(1 + 0 * 1) + (1 + 0 * -1)

Q. -((y+xy) + ((-x) + (-xy)))
```

File 4: **Answers.txt**

```
--------------------------------------------------------------------------------
Q.  0 + 1 + 0 + 0

   Property of Zero for OR
=   1 + 0 + 0

   Property of Zero for OR
=   1 + 0

   Property of Zero for OR
=   1

--------------------------------------------------------------------------------
```

Q. ((0 * 1) * 0) * 0

   Property of Zero for AND
=   (0 * 0) * 0

   Property of Zero for AND
=   0 * 0

   Property of Zero for AND
=   0

Q. - (1 + 0 * 1) + 1 + 0 * (- (1))

   Basic Postulate of NOT
=   - (1 + 0 * 1) + 1 + 0 * 0

   Property of Zero for AND
=   - (1 + 0 * 1) + 1 + 0

   Property of Zero for OR
=   - (1 + 0 * 1) + 1

   Property of Zero for AND
=   - (1 + 0) + 1

   Property of Zero for OR
=   - (1) + 1

   Basic Postulate of NOT
=   0 + 1

   Property of Zero for OR
=   1

Q. - (y + x * y + - (x) + (- (x)) * y)

   Absorption Laws
=   - (y + x * y + - (x))

   Absorption Laws
=   - (y + - (x))

   DeMorgan's Theorem
=   (- (y)) * (- (- (x)))

   Involution
=   (- (y)) * x

BIBLIOGRAPHY

Wolf, Carol E. *"**Infix to Postfix Conversion Algorithm**"*
From www.csis.pace.edu/~wolf/CS122/infix-postfix.htm

Arora, Sumita (2016). *"**Computer Science with C++**"*