**Program 4 Design:**

**Definitions:**

**Symbolic calculator:** Is a calculator that can use variables and numbers and stores the structure of the expressions that are entered. The calculator can also perform transformations of an expression like simplification or derivatives.

**Tokens:** A token is the smallest element of a C++ program that is meaningful to the compiler. They can be classified as: keywords, identifiers, constants, strings, special symbols, and operators. The C++ parser recognizes the tokens listed above.

**Tokenizer:** The tokenizer is responsible for dividing the input stream into individual tokens, identifying the token type, and passing tokens one at a time to the parser.

**Parser:** A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. It takes input in the form of a sequence of tokens and breaks them up into parts that can be used by other components in programming.

**Abstract Syntax Tree:** is a tree representation of the abstract syntactic structure of text.

A picture containing shape

Description automatically generated

**Infix:** Infix notation is set up like: Operand Operator Operand (2+2)

**Postfix:** Postfix notation is set up like Operand Operand Operator (22+)

**Prefix:** Prefix notation is set up like Operator Operand Operand (+22)

**Specification**

**Purpose:** The purpose of this program is to implement a symbolic algebra calculator that will read in mathematical expressions and convert them into an abstract syntax tree as pictured below. The abstract syntax tree will allow for the expressions to be modified (simplified, transformed, or solved).

**Description:** The program will read expressions from cin, convert the expressions into individual tokens based on their type (operand, parenthesis, operator, etc.). Then the calculator class will read in input as tokens as long as the “.” exit character is not entered. An infix vector of tokens will be created from the tokens read in for each expression. From there, the infix expression will be checked to ensure that it is valid. If the expression is valid, it will be converted into a postfix expression that’s stored in a vector of tokens. An AST object will then be built by passing in the postfix expression of tokens. If a variable is being assigned in the expression, another AST object will be created to store the variables expression or value. The new AST will be stored in a map with a key that will be the variable’s letter. The AST will be simplified for the expression and the simplified expression will be printed in infix form as a string. The user can continue to enter in expressions until the exit character is entered, which will exit the calculator.

**Calculator Rules:**

1. The calculator will only save input into variables if assigned.
2. It can store 26 variables, with each variable being a-z and not case sensitive.
3. Assignment for a variable is (:=)
   1. Ex: y := 10
4. Variables can have new expressions assigned to them.
5. The calculator can perform the following operations:
   1. +
   2. –
   3. \*
   4. /
   5. ^
6. The calculator handles parenthesis when determining order of operations.
7. Entering a “.” into cin will cause the calculator to exit
8. If invalid syntax is used, an error message should be displayed, and the user should be able to enter in another line.
9. If a variable has not been assigned a value, the value of the variable is just the variable itself. Ex: q would have a value of q.
10. All number inputs must be positive integers.

**Assumptions:**

All numbers entered into the calculator are positive integers. 26 variables are able to be stored (a-z). The value of variables can be changed. There can be negative output. Division does not result in a float. No circular expressions are allowed.

**Special Cases:** If an expression contains the exit character “.” an exception will be thrown because this is invalid infix form for an expression. The “.” must be entered on a single line by itself.If a character from cin is none of the allowed tokens, then it is an invalid token, and an error message will be displayed. The assignment operator for the calculator (:=) will be a special case for creating a token, because it consists of two characters. Creating a number token will also be a special case because a number can be more than one digit. An assignment expression will be treated differently when creating an AST, it will store the AST in a map with the key being the variable the expression was assigned to.

**Input/Output:** The input to the calculator will be an expression like 5+5. Variables are also allowed to be entered into the calculator like x+5. Assignment of a variable will be put in like x:=10. If the syntax of the input is incorrect, an error message will be displayed, and the user can continue to enter expressions. In order to exit the calculator, a “.” must be entered. The output of the calculator will be either a simplified formula or solution to the expression. The output may also include parenthesis, depending on order of operations of the expression. The input must also be no longer than 80 characters.

**Testing:** Individual unit tests will first be performed on each individual class. After the unit tests pass with various inputs, an integrated test will be performed via inputting a variety of different expressions.

**Error Handling:** If the syntax of an expression is incorrect, an exception will be thrown and caught so that the program can continue to take input from the user until the exit character is entered.

**Design**

**Token:** this class contains an enum called TokenType which will assign Tokens a type from the enum. The Token struct creates a Token with a TokenType and a string value.

**enum TokenType** {

binop, variable, number, lparen, rparen, unknown, invalid, assignop, ending, powop, eol

};  
**struct Token** {

Token(TokenType t, string v): type\_(t), value\_(v){}: constructor that takes in a TokenType t and string v and sets the type\_ and value\_

Token(): type\_(unknown), value\_(“”){}: default constructor that sets type\_ to unknown TokenType and value\_ to an empty string  
 TokType type\_: the TokenType of the Token  
 string value\_: the string value of the Token   
};

**Definition of Token Types:**

Binop: operator (+, -, \*, /)

Variable: variable (a-z)

Number: number

Lparenth: left parenthesis

Rparenth: right parenthesis

Uknown: for default constructor

Invalid: invalid token type

Assignop: assigns an expression to a variable (:=)

Ending: ends the program (.)

Powop: power operator (^)

Eol: end of line (/n)

**TokenStream:** The TokenStream class acts like a tokenizer and turns input into Tokens.

**class TokenStream** {  
public:

TokenStream(istream& inputStream): is\_(inputStream) : takes in an input stream and sets the is\_ to inputStream

TokenStream& operator>>(Token& rhs): gets one char at a time from the istream and creates tokens based on the chars from the istream

explicit operator bool() const: this allows you to perform Boolean expressions on the input stream  
  
private:  
 istream& is\_: variable that holds the istream  
};

**AST :** The AST class takes in tokens that represent an expression in postfix form and creates an AST tree. The class can also simplify the AST tree and convert the AST back into infix form.

**class AST** {  
public:

AST(vector<Token>& postfixExpr, map<string, AST> &variables): this takes in a vector of tokens that are a postfix representation of a mathematical expression and a map of the variables along with their root node that points to AST and builds an AST made out of nodes. The constructor calls the function constructTreeHelper() which assigns any variables and constructs the tree by calling constructTree(). In addition, it creates a copy of the variables map and postfixExpr so that the copy constructor can easily copy the tree. The isAssignment member variable and hasVariable are both set to false here as well.

~AST(): destructor for AST that calls clear().

void clear(): recursively deletes children nodes until base case is reached where you reach the root pointer which is deleted last.

AST(const AST& ast): copy constructor for AST that accesses the postfixExpr and map of variables and cosntructs tree that is an exact copy of ast.

AST& operator=(const AST& ast): assignment operator for AST

void constructTreeHelper(vector<Token>& postfixExpr, map<string, AST\*> &variables): this function is called in the constructor. It first determines if the postfixExpr is an assignment and if so, creates a new postfix vector without the variable and assignment operator (x:=). From there is calls constructTree(). It also calls the function assignVariable() which stores the AST expression for the variable in the varaibles map. If it is not an assignment, constructTree() is called immediately.

void constructTree(vector<Token>& postfixExpr): this function builds the AST tree for either an assignment or just a normal expression

bool isAnAssignment(vector<Token>& postfix): returns true if expression is an assignment and sets the member variable isAssignment to true

bool setHasVariable (vector<Token>& postfix): returns true if the expression has a variable and sets the member variable hasVariable to true

bool isOperator(Token t) const: returns true if token is an operator

bool isOperand(Token t) const: returns true if token is an operand

bool isPower(Token t) const: returns true if token is a ^

bool isVariable(Token t): returns true if the token has type variable

void assignVariables(string v, Node\* root, map<string, AST>& variables): assigns the variable with AST in the variables map via storing the root pointer to the AST

AST simplify(map<string, AST>& variables) const: this creates a copy of the current AST via the copy constructor. The fillVariables() method is then called, which fills any variables in the tree with the AST that the variable is assigned. Then the traverseAndSimplify() method is called which simplifies any expressions that can be simplified and returns the simplified AST, which is then returned by this function.

AST fillVariables(Node\* root, map<string, AST> &variables, AST &newAST): this function traverses the AST in postfix order and fills in AST for all variables that have an AST assigned to them from the map

AST traverseAndSimplify(Node\* root, AST &newAST) const: this function takes in the root pointer to the copy ast from simplify and simplifies every expression in the tree by calling calc

string toInfix(AST &newAST) const: this returns a string of the AST in infix form that includes parenthesis

private:

**struct Node**{

Node(Token t): creates a node with Token t

Node(Token t, Node\* leftptr, Node\* rightptr): creates a node with Token t and assigned leftptr and rightptr

Node(Token t, Node\* leftptr): creates a node with Token t and assigned the leftptr

Token token: variable that stores type of token

Node\* left: pointer to left node

Node\* right: pointer to right node

Node\* root: this points to the root of the AST

bool isAssignment: is true if the expression entered is an assignment

bool hasVariable: is true if the expression has variables

Vector<Token> postfixCopy: is a copy of the postfix expression for copy constructor

Map<string, AST> variablesCopy: is a copy of the variables map for copy constructor

Map<string, AST>:iterator it: iterator for the map, variables   
};

**Calculator :** The calculator class takes input from the user line by line and converts the input into an infix vector of Tokens. It then checks to ensure that the infix expression is valid and converts the infix expression to a postfix expression. The postfix expression is then used to create an AST which simplifies the expression and prints the infix simplified expression to the console. The user can continue to enter lines of input so long as a “.” is not entered, which will end the program.

**class Calculator** {  
public:

Calculator(): tstream(cin): constructor that initializes all member variables

~Calculator(): destructor

void calculate(): this function continuously asks the user for input until a . is entered. It stores the tokens from input into a vector and then checks to ensure that the input is a valid infix expression via the isValid() function. If it is not, it will display an error message and continue to get input from user. If it is a valid expression, the infix is converted to postfix via the convertPostFix() function. From there, an AST is created by passing in the postfix vector of tokens along with the map that holds the variables and their assignments. The AST is then simplified via the simplify() function and the solution is printed to the screen via the toInfix() function. The user can continue to enter expressions after this until the end condition is met.

void convertPostFix(const vector<Token>& infix): this function takes in a vector of infix tokens and converts the expression to postfix and which is stored in a member variable called postfix.

bool isValid(const vector<Token>& infix): this function checks to ensure that all rules for infix expressions are being followed and that it is a valid expression or assignment. This also checks to make sure the input is no longer than 80 characters.

int precedence(Token t) const: this method takes in a token and based on the token type of operand, returns the precedence in which it should be evaluated by returning a value.

private:

map<string, AST> variables: this is a map that holds the variable letter and an AST that is being assigned to that variable

vector<Token> postfix: this is a vector of tokens that is in postfix form

TokenStream tstream: this is the tokenized stream

};

**Non-trivial methods implementation:**

**TokenStream:**

**TokenStream& operator>>(Token& rhs):**

switch (char from stream, c)

{

// when to use breaks in case

case '.':

rhs = Token(ending, ".")

case '(':

rhs = Token(rparen, "(")

case ')':

rhs = Token(lparen, ")")

case '+':

rhs = Token(binop, "+")

case '-':

rhs = Token(binop, "-")

case '/':

if (next char == 'n')

{

rhs = Token(eol, "/n")

}

else

{

rhs = Token(binop, "/")

}

case '\*':

rhs = Token(binop, "\*")

case '^':

rhs = Token(powop, "^")

case ':':

if (next char == '=')

{

get() next char

rhs = Token(assignop, ":=")

}

else

{

rhs = Token(invalid, "")

}

default:

if (char is digit)

{

value += c

while (the next char is a digit)

{

Get next char

value += c

}

}

else if (char is a letter)

{

String var += c

//make sure the letter is lowercase always

Convert string to lowercase letter

rhs = Token(variable, var)

}

else

{

rhs = Token(invalid, "")

}

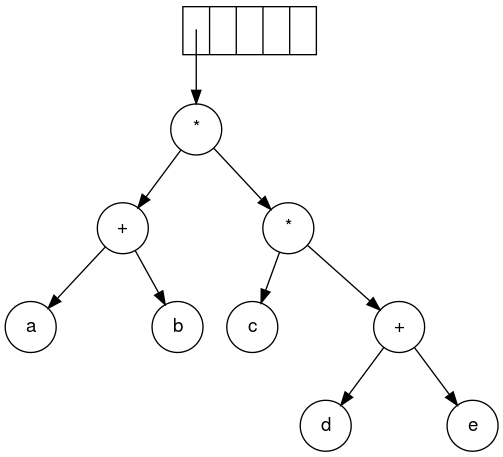
}

return this

}

**AST:**

The picture below is what the **constructTree()** method will look like in action when using a stack:



**void constructTreeHelper(vector<Token>& postfixExpr, map<string, AST>& variables):**

Call setHasVariables to set a marker for if the expression contains variables

if (the postfix is an assignment){

create new vector called assignmentPostfix and copy postfix into it

erase first two tokens which should be a variable and the assignop

constructTree(assignmentPostfix)

//add the variable along with the AST to the map of variables

assignVariable(variable to be assigned, variables from map, this)

}

else{

constructTree(postfix)

}

}

**void constructTree(vector<Token>& postfixExpr):**

create a stack of type node

for(0: postfix size){

if (token is operator or power operator){

Node\* left = stack.top();

stack.pop();

Node\* right = stack.top();

stack.pop();

Node\* node = new Node(postfix[i], left, right);

stack.push(node);

}

else if (token is operand) {

stack.push(new Node(postfix[i]));

}

}

root\_ = stack.top();

}

**void assignVariables(string v, map<string, AST>& variables):**

search through map to find variable

if variable is there, delete the AST associated with it and store new AST

variables.insert(v, AST);

**AST simplify(map<string, AST>& variables) const:**

create a copy of the AST via the copy constructor, newAST

//if the expression has variables

if (is newAST has variables){

fillVariables by calling the fillVariables function

return simplified newAST by calling traverseAndSimplify function

}

else {

return (traverseAndSimplify(root, newAST));

}

]

**AST& fillVariables(Node\* &root, map<string, AST>& variables, AST &newAST):**

if (root is nullptr){

return newAST

}

fillVariables(root->left, variables, newAST)

fillVariables(root->right, variables, newAST)

if (token is a variable)

set the root to root of AST stored in variables map

}

}

**AST& traverseAndSimplify(Node\* &root, AST &newAST):**

if (root is nullptr){

return newAST

}

traverseAndSimplify(root->left, newAST)

traverseAndSimplify(root->right, newAST)

if (token is an operator or power operator){

if (if left and right tokens are both numbers){

create a new Token and set the value and type to the number that results from the calculation which is done by calc function

delete children nodes and set the pointers to

}

}

}

**string toInfix(AST &newAST) const:**

return toInfixHelper(root of newAST);

**string toInfixHelper(AST &newAST) const:**

string infix

if (node is nullptr)

return infix

if (token is operand) {

infix+= "("

infix += left node token value

infix += right node token value

}

toInfix(left node)

infix+= node->token value

toInfix(right node)

if (token is operator) {

infix+= ")"

}

}

**string calc(string leftOperand, string binop, string rightOperand) const:**

convert string operands to ints

int solution

if (binop == "+"){

solution = left+right

}

else if (binop == "-"){

solution = left-right

}

else if (binop == "\*"){

solution = left\*right

}

else if(binop == "/"){

solution = left/right

}

else {

solution = pow(left, right)

}

return string version of int (solution)

}

**Calc:**

**void calculate():**

void Calc::calculate(){

bool done = false

//keep calculating expressions until done

while ( not done){

Token tok = Token()

vector<Token> infix

TokenStream tstream(cin)

while(tok.type\_ is not at end of line){

tstream >> tok

infix.push\_back(tok)

}

if (first element in vector is an ending token){

done = true

}

if(infix is valid expression via isValid())

convert infix to postfix via convertPostFix()

create a new AST with postfix vector and variables map

simplify ast via simplify() from AST class

print "Printing simplified expression/solution:"

print infix form of simplified tree via toInfix() from AST class

}

}

print "Exited Calculator."

}

**void convertPostFix(const vector<Token>& infix, vector<Token>& postfix):**

stack<token> s

for (0: infix size)

{

switch (token)

{

if(token is number or variable)

add token to postfix

// If the scanned character is an

// ‘(‘, push it to the stack.

else if(token is left parenthesis)

push onto stack

// If the scanned character is an ‘)’,

// pop and to output string from the stack

// until an ‘(‘ is encountered.

else if(token is right parenthesis) {

while(top of stack token is not left parenthesis)

{

Add top of stack to postfix

Pop stack

}

Pop stack

}

//If an operator is scanned

else {

while(stack is not empty and precedence of t is less than or equal to precedence of top of stack token) {

add top of stack to postfix

pop stack

}

Push token onto stack

}

}

// Pop all the remaining elements from the stack

while(stack is not open) {

add top of stack to postfix

pop stack

}

}

**int precedence(Token t) const:**

if (token type is powop){

return 3

}

else if (token value is “/” or “\*”){

return 2

}

else if (token value is "+" or "-"){

return 1

}

else {

return -1

}

}

**bool isValid(const vector<Token>& infix):**

if (infix size is less than 80){

print error message "Error, input must be no longer than 80 characters."

return false

}

//if contains an invalid token type or ending token type return false

for (0 to infix size){

if (infix type is invalid){

print "Invalid character entered."

print "Please enter a valid expression."

return false

}

if (token is of type ending and not the first element and infix is larger than 1 element){

print "To terminate, enter a . on a single line."

return false

}

}

//ensure correct first token type

if (first token is not a number, right parenthasis or variable)

return false

}

//ensure correct end tokentype

if (end token is not a left parenthasis, number, or variable){

return false

}

//ensure rest of expression follows infix rules

for (0 to size of infix){

if (i+1 is not greater than the length){

if(infix[i].type\_ == rparen && infix[i+1].type\_ != powop ||

infix[i+1].type\_ != binop){

return false

}

if(infix[i].type\_ == number && infix[i+1].type\_ != binop ||

infix[i+1].type\_ != powop || infix[i+1].type\_ != lparen){

return false

}

if(infix[i].type\_ == lparen && infix[i+1].type\_ != number ||

infix[i+1].type\_ != variable){

return false

}

if(infix[i].type\_ == powop && infix[i+1].type\_ != number ||

infix[i+1].type\_ != variable){

return false

}

if(infix[i].type\_ == variable && infix[i+1].type\_ != assignop ||

infix[i+1].type\_ != binop || infix[i+1].type\_ != lparen

|| infix[i+1].type\_ != powop){

return false

}

if(infix[i].type\_ == binop && infix[i+1].type\_ != number ||

infix[i+1].type\_ != variable || infix[i+1].type\_ != lparen

|| infix[i+1].type\_ != rparen){

return false

}

//this needs to ensure that 10 + x:= does not work

if(infix[i].type\_ == assignop){

if (i != 2 || infix[i-1].type\_ != variable){

//must be second token always for assignment operator

return false

}

else if (infix[i+1].type\_ != number

|| infix[i+1].type\_ != variable || infix[i+1].type\_ != lparen ) {

return false

}

}

return true

}

**Implementation Plan**

**Unit Test:**

1. Test the TokenStream class and ensure that the input stream gets converted into the correct tokens.
   1. Input: istream
   2. Output: tokens with correct token type and value
2. Test the AST class
   1. Input: postfix mathematical expression, postfix mathematical assignment
   2. Print tree before simplifying
   3. Print tree after simplifying
3. Test the Calc class
   1. Read lines of input and test to ensure the ending token ends the program
   2. Test isValid()
      1. Input
         1. A valid infix expression
         2. An invalid infix expression
      2. Output
         1. For the valid it should return true
         2. For the invalid it should display an error message and return false
   3. Test convertPostfix()
      1. Input
         1. A valid infix expression
      2. Output
         1. Print the member variable postfix which should be altered by this function

**Integration Tests:**

1. Create a calculator
2. Calculate()
   1. Enter many different correct and incorrect infix expressions
   2. Ensure the output is correct, which will be a simplified expression in infix form

https://canvas.uw.edu/courses/1494706/pages/program-4-description-a-symbolic-algebra-calculator?module\_item\_id=13974841