Project 2 :
Hard Autograd for Algebraic Expressions

Author's Name

Date: 2025-03-22

Chapter 1: Introduction

Brief Description:

The program accepts infix expressions composed of arithmetic operators (e.g., +, *, ^), literals, variables, and optionally, mathematical functions (e.g., ln, cos, pow). It parses these expressions into abstract syntax trees and applies differentiation rules recursively to compute partial derivatives. The output adheres strictly to the input format, ensuring derivative expressions are syntactically correct and equivalent to the analytical solutions.

Key challenges include handling operator precedence, nested functions, and complex differentiation rules (e.g., chain rule for composite functions). Additionally, optional simplification features reduce expression complexity by applying algebraic rules, such as constant folding or identity elimination. The implementation prioritizes readability and efficiency, adhering to C/C++ standards without relying on advanced STL containers.

This project bridges symbolic computation and Automatic Differentiation(AD), providing a foundation for applications in education, scientific computing, and optimization. It demonstrates core AD mechanics while emphasizing algorithmic clarity and robustness through rigorous testing across diverse input cases.

About Bonus:

1. Not using complex containers provided in STL

As required, simple STL containers is allowed.So here I will briefly introduce the simple STL containers(string vector map set), and it meets the requirement of the bonus.

Usage:

-----------------------std::map----------------------

1. VarMap (std::map<std::string, int>)

- Purpose: Maps variable names to unique integer IDs for efficient lookup and management.

- Usage: When a new variable is encountered, it is inserted into VarMap with a generated ID. Subsequent lookups use this map to quickly find the ID associated with a variable name.

2. Factors (std::map<ExprHash, ExprNode**>)

- Purpose: Tracks factors during polynomial simplification to identify common terms.

- Usage: During the simplification process, factors of terms are stored in this map to facilitate combining like terms and extracting common factors.

3. Common (std::map<ExprHash, CommonFactor>)

- Purpose: Identifies common factors between different terms during simplification.

- Usage: When comparing terms to find common factors, this map stores the relationships between common elements found in different parts of the expression.

4. Tg (std::map<ExprHash, ExprNode**>)

- Purpose: Temporarily holds hash values of expressions during various simplification steps.

- Usage: In multiple simplification functions, this map is used to track hashes of nodes that have been processed to avoid redundant operations and ensure each node is simplified efficiently.

std::map is chosen for these scenarios due to its efficient key-value storage and retrieval capabilities. It allows for quick lookups, insertions, and deletions, which are essential for managing variables, tracking expression components during simplification, and optimizing the overall performance of the expression processing pipeline.

-----------------------std::set----------------------

1. DupStrings (std::set<char*>)

- Purpose: Manages duplicate strings to ensure each unique string is stored only once.

- Usage: When creating string duplicates, it checks if the string already exists in the set to avoid redundancy.

2. Extracted (std::set<ExprHash>)

- Purpose: Keeps track of extracted hash values during the simplification process to prevent reprocessing.

- Usage: During various simplification steps, it stores hashes of processed nodes to ensure each node is simplified only once.

std::set is chosen for these purposes due to its properties of unique element storage and efficient lookup. It helps in managing resources efficiently and avoiding unnecessary computations during the expression tree processing.

-----------------------std::vector----------------------

1. Tokens (std::vector<Token>)

- Purpose: Stores the sequence of tokens generated from the input expression.

- Usage: After tokenization, this vector holds all the elements of the expression, such as numbers, variables, operators, and functions, which are then used to build the expression tree.

2. Nodes (std::vector<ExprNode*>)

- Purpose: Manages a pool of expression nodes to optimize memory usage and reuse nodes when possible.

- Usage: Nodes are created and stored in this vector during the construction of the expression tree, allowing for efficient node management and traversal.

3. UnusedNode (std::vector<ExprNode*>)

- Purpose: Keeps track of nodes that are no longer in use and can be reused to avoid memory reallocation.

- Usage: When nodes are released or removed from the tree, they are added to this vector to be reused later, enhancing performance.

4. Brackets (std::vector<BracketPtr>)

- Purpose: Tracks the positions of brackets during the parsing process to handle nested expressions correctly.

- Usage: As the parser encounters brackets, it records their positions in this vector to manage the scope and priority of operations within brackets.

5. TraverseSeries (std::vector<ExprNode*>)

- Purpose: Temporarily stores nodes during tree traversal operations.

- Usage: Various simplification and manipulation functions use this vector to collect nodes of specific types or properties for further processing.

6. ExprV (std::vector<ExprNode*>)

- Purpose: Used during the creation of subtrees to hold intermediate expression nodes.

- Usage: When constructing complex expressions, this vector helps manage the nodes that form parts of the larger expression tree.

7. OprV (std::vector<ExprNode*>)

- Purpose: Stores operator nodes during the parsing and tree construction process.

- Usage: Helps in managing the order and precedence of operators when building the expression tree from the token sequence.

std::vector is selected for these purposes due to its dynamic size management and efficient element access. It provides a flexible way to store and manipulate sequences of elements, making it suitable for managing tokens, nodes, and other components during the expression processing pipeline.

-----------------------std::string-----------------------

1. Vars (std::vector<std::string>)

- Purpose: Stores the names of variables encountered in the expression.

- Usage: When a new variable is found during tokenization, its name is stored here for reference and management.

2. Expression (std::string)

- Purpose: Holds the input mathematical expression as a string.

- Usage: This string is processed and parsed to generate tokens and build the expression tree.

3. Token::GetText() method

- Purpose: Returns the textual representation of a token.

- Usage: Depending on the token type, it constructs and returns a string that represents the token's value.

4. HashStr functions

- Purpose: Computes a hash value for a given string.

- Usage: These functions take a string (like variable or function names) and compute a hash to be used in various hash-based operations.

5. DuplicateStr and related functions

- Purpose: Creates duplicate strings with proper memory management.

- Usage: Ensures that strings used in the expression tree are properly allocated and managed to avoid memory leaks.

6. Function names in the Funcs array

- Purpose: Stores the names of mathematical functions.

- Usage: These strings are used to identify and match functions during the parsing and evaluation of the expression tree.

std::string is essential for handling textual data in the code. It provides flexible string manipulation capabilities, allowing for efficient storage and processing of variable names, function names, and the input expression itself. This makes it possible to correctly parse and interpret the mathematical expressions provided by the user.

2. Support for expressions containing mathematical functions

I am sure that 9 functions including unary function **ln exp sin cos tan sinh cosh** and binary function **log pow** is supported.

Syntax: unary F: F(x) is ok. DO NOT FORGET the brackets. Binary:

log(x,y)= $\log_x y$ pow(x,y)=x^y

3. Simplify an algebraic expression to reduce the length of the result by applying at least two rules.

There is 7 major rules of simplification used in the program:

The details will be described in Chapter 2.

(1) Rotate Simplification

(2) 0-1 Simplification

(3) Negative Simplification

(4) Special Function Simplification

(5) Polynomial Simplification

(6) Fold Constant Simplification

(7) Final Fold Simplification

Chapter 2: Algorithm Specification

Part 1: Algorithm Overview

1. Process:

The program is an **infinite loop** evaluating the expressions. Every time the program take a line of input and calculate the partial derivative of the original expression. The output is arranged according to the **lexicographical order** of the variables.

Pseudocode:

```
while true do
    ExpressionString = GetLine()
    TokenVector = Tokenize(ExpressionString)
    ExpressionTree = Expression(TokenVector)
    if FailedToParse or DividedbyZero then continue
    Simplify(ExpressionTree)
    for Var in VariableMap do
        PartialExprTree = Partial(ExpressionTree, Var)
        if DividedbyZero then continue
        Simplify(PartialExprTree)
        print("{Var} :")
        PrintTree(PartialExprTree)
    ResetGlobalVariables()
```

2. Description:

During the parse process, if there is an error, the parse function will raise a **syntax error** and set the flag `FailedToParse` to true.

Divided by Zero is also considered an error. When it is found, the program will raise the error. When an error is raised, the program will output the error text. Another thing we should pay attention is that the flags `FailedToParse` and `DividedbyZero` are reset to false every time a new expression is created since they are global flags.

The `VariableMap` is a `std::map.` The `std::map` is ordered, so the output will automatically sorted by **lexicographical order**.

3. How to read the code:

Since this cource code is 3777 lines long, it may be hard to read all the codes. The comments of the functions are mostly in Doxygen format, you can read the source code with its help. A suitable Code Editor or IDE may help you a lot.

What's more, Use test cases to understand how different parts of the code behave. The provided test cases in the report can guide you. Enable debug output by setting the debug constants to true to get more insights into the code's execution.

4. Debugging:

To debug this algorithm, sometimes you need to **enable debug data output**. **Line 173~189** defines three debug constants. The constants controls whether the debug info will appear with `constexpr if` — this guarantees that only when you enable debug data output will the debug output commands be compiled.(this is one of the reasons why this program requires at least C++17)

Three different constants controls the debug output of different stages. Use it on your own will.

Part 2: Input

Pseudocode:

```
ExpressionString = GetLine()
```

Here we used `std::getline(std::cin, Expression)` to read a line from `std::cin` to the string `Expression`.

Here the program supports a line with blank characters and **all characters except operators/letters** are considered **SPACE.**

The operators include eight characters: `+ - * / ^ ( ) ,`

The functions are `ln exp sin cos tan sinh cosh log pow`.

Part 3: Tokenization

This part (Mainly **Line 818~1025**) Generates an array of tokens from an expression string.

Attention: for a negative number, here the function will separate the negative sign from the number in order to simplify this process. We must pay attention that this solution will create operator "-" with only 1 operands on the right, and the program would process it in **Part 4**.

Another thing we should consider that this way of processing only allows functions and variables made up of **ONLY LETTERS**. However the problem description says “identifiers (strings of lowercase English letters) that are not reserved words are called variables.” Haha.

Process in Pseudocode:

In main function:

```
TokenVector = Tokenize(ExpressionString)
```

Here “Type of the character” decides the type — an operator, a letter, a number, or other character(considered as **SPACE**)

The Function Tokenize:

```
function Tokenize(Expression as String)
return Result as Vector<Token>
    for Character in Expression do
        if Character is not PreviousType or Character is OPERATOR then
            if Character is not SPACE then //all other characters are SPACE
                append ParseToken(Substring of This Token) to Result
            Start a new token
    append ParseToken to Result //the Last Token
```

```
function ParseToken(SubStr as String)
return Token
    if SubStr is OPERATOR then
        return Operator from Substr
    else if SubStr is NUMBER then
        return Number from Substr
    else if Substr is SYMBOL then //made up of letters
        //judge if it is a function after receiving the whole symbol
        if SubStr is FUNCTION then
            return Function from Substr
        else
            return Variable from Substr
```

In this part the program reads the Expression string character by character, and judge the type. When the type is different from the previous type, a new token is parsed unless it just received a space. In the end, the remaining unhandled string will be considered as the last token and parsed.

When it comes to a complete token, the program judges the type and get a token with the correct type. The supported functions are registered in the `Funcs` array at **Line 421**. Helper function `GetFuncID` at **Line 999** checks if a symbol is a function by comparing the string one by one.

For the main logic and the technical details, see function `ParseToken` at **Line 856**, function `GenerateTokens` at **Line 887** and other helper function involved.

Part 4: Tree Construction

This part (Mainly **Line 1026~1298**) Generates a **binary tree** from the original infix expression in Tokens. Since there is no function holds more than 2 parameters, so the arguments are just stored as child nodes of the function node.

Process in Pseudocode:

In main function:

```
ExpressionTree = Expression(TokenVector)
```

The Function Tokenize:

Every binary tree node is composed of a Token and the Left and Right children.

```
function Expression(Tokens as Vector<Token>)
return ExprNode
    make Tokens to LinkedList<ExprNode> ranged [Begin, End)
    if Begin == End then
        return empty node

    // Handle unary minus (e.g., -x → 0 - x)
    if Begin is '-' and has no left operand then
        Insert 0 node before Begin
        Update Begin to point to new 0 node

    OprStack as Stack<ExprNode> //Operator
    ExprStack as Stack<ExprNode> //Operand
    LastIsOperand = false

    for pNode in [Begin, End) do
        skip brackets/comma nodes
        if pNode is operand (number/variable) then
            if LastIsOperand then
                // Implicit multiplication (e.g., 2x → 2*x)
                MergeTop(OprStack, priority=2)
                Push '*' to OprStack
            Push pNode to ExprStack
(see next page)
```

```
(continue prev page)
            LastIsOperand = true
        else
            // Operator or function
            MergeTop(OprStack, pNode's priority)
            Push pNode to OprStack
            LastIsOperand = false

    // Merge remaining operators
    while OprStack not empty do
        MergeTop(OprStack, priority=0)
    return top of ExprStack
```

Helper function Mergetop:

This function merges the topmost nodes in the two stacks mentioned and generate a new subtree as an operand.

```
function MergeTop(OprStack as Stack<ExprNode*>, Priority as int)
return void
    while OprStack not empty do
        Top = OprStack.top()
        if (Top is '-' or '/') and Top's priority >= Priority then
            // Right-associative operators
            Pop Top from OprStack
            Attach Top's left and right from ExprStack
            Replace ExprStack entries with merged node
        else if Top's priority > Priority then
            // Left-associative operators
            Same as above
        else break
```

In this part the function first handle the operator “-” with only the right operand generated in **Part 3**.

About skipping brackets and commas: the program first handle all ( ) and , in the Constructor of class Expr. The preprocessing algorithm merge every function and sub-expression in the bracket into **a single ExprNode** to simplify following algorithm. This algorithm is skipped in the pseudocode since it is just a simple **stack-based bracket matcher**.

Another thing we should consider is that this step may raise a ParseError unmatched brackets, arguments in incorrect numbers and missing operand for the operators. If such errors are triggered, an error text will be displayed onto your console. Read it for the cause of the error.

Note: the pseudocode describes that the program change the array of tokens to a linked list of nodes at the very beginning. Actually the real function just append a new node to the end every time it reads the next token. Both ways are **equivalent**.

For the main logic and the technical details, see function `Expr(CTOR)` at **Line 1175**, function `CreateTree` at **Line 1073**, function `CheckArgument` at **Line 1143** and other helper functions involved.

Part 5: Partial Derivative

This part (Mainly **Line 1887~2127**) Calculates the partial derivative of the expression by transforming a clone of the original expression tree.

Process in Pseudocode:

In main function:

```
PartialExprTree = Partial(ExpressionTree, Var)
```

The Function Partial:

```
function Partial(Original as ExprNode, Var as Variable)
return ExprNode
    switch Original.Type do
        case Token::Int:
            return NewNode 0
        case Token::Variable:
            if Original.Token == Var then
                return NewNode 1
            else
                return NewNode 0
        case Token::Function:
        case Token::Operator:
            return Original.Partial(Original.Left, Original.Right, Var)
```

Here Original.Partial represents a list search matching Partial Derivative functions of all operators and functions. The program uses some helper types, functions, macros and objects to simplify the specific partial functions.

With helper functions, the specific partial functions are simpler. They include:

```
d(f+g)/dx = f’+g’

d(f-g)/dx = f’-g’

d(f*g)/dx = f'*g + f*g'
```

```
d(f/g)/dx = (f'g - fg')/g^2
d(ln(f))/dx = f'/f
d(cos(f))/dx = -f'*sin(f)
d(sin(f))/dx = cos(f) * f'
d(tan(f))/dx = f' / cos(f)^2
d(exp(f))/dx = exp(f) * f'
d(cosh(f))/dx = -f'*sinh(f)
d(sinh(f))/dx = f'*cosh(f)
```

Note: the two binary functions are transformed into unary functions before calculating partial derivative.

```
d(log_b(f))/dx = (ln(f)/ln(b))'
d(pow(x,y))/dx = (exp(y*ln(x)))'
```

Here the partial derivative function is generated recursively on the binary expression tree by naturally breaking down the function in the specific partial functions mentioned above.

For the main logic and the technical details, see function `ExprNode::Partial` at **Line 1899** , function `ExprNode::PartialOpr` at **Line 1922** , the specific partial derivative functions at **Line 1952~2120** and other helper function involved.

Part 6: Simplification

This part (Mainly **Line 2128~3725**) Simplifies the whole expression on the tree to a simpler form in order to shorten the length of the expression and discard useless parts of that. To some extent we can say that it may be a rather complex part of code.

Process in Pseudocode:

In main function:

```
Simplify(ExpressionTree)
……………
Simplify(PartialExprTree)
```

The Function Simplify:

```
function Simplify(Node as ExprNode)
    Occurred as Set<ExprHash>
    do
        Simplify_Rotate(Node)
        Simplify_01(Node)
        Simplify_Negative(Node)
        Simplify_SpecialFunction(Node)
        Simplify_Polynomial(Node)
        Simplify_FoldConstant(Node)
    while success: insert Node.Hash to Occurred

    FinalFold_GCD (pNode)
    FinalFold_Negative(pNode)
    while pNode is changed do
        Simplify_FoldConstant(Node)
        Simplify_Rotate(Node)
        Simplify_01(Node)
```

Note: Here `std::set` fails to insert a value if it already exists in the set. The function continuously simplifies the function until it can be no more optimized. Then the final expression will be finally folded and considered the simplest afterwards.

If we consider all the expressions composes a set $\mathcal{F}$, every loop of simplification can be seen as a mapping **T**: $\mathcal{F} \to \mathcal{F}$ and for all **f** $\in \mathcal{F}$ there is a **T(f)** $\in \mathcal{F}$.So if and only if **T(f)=f** the current way to simplify comes to an end as it falls into a infinite loop, and we can just stop here. But another thing should be paid attention is that this only means the process cannot advance a step, it doesn’t mean the result is simplest.

Why there is a final fold: Some of the simplification like switch x^{-1} to $\frac{1}{x}$ may interrupt the process and make it harder to keep optimizing. So the program chooses to change all “-” to “+”, “/” to “*” so the program would only consider + and * when it tries to merge some of the subtrees. In order to repack these transformed but obviously not simplest expressions into the simpler form by performing Final Fold.

Some critical tools: you may find two macros: `TraverseType` and `TraverseTypeLocal`. This mock that your binary tree has been transformed into a normal tree with multiple children each node sharing the same operator like + and * with the help of pre-traverse the subtree sharing the same operator node. By using these, you can have access to all the nodes connected by the same operator as if they share the parent node, which is important when searching for the matched node to optimize.

Below is the simplify process steps. From the formulae, the order and some necessary notes, the simplification is broken up into 8 types. In general, they can be devided into two large types of operation: shortening the expression, or transform the expression to a much simpler form to better the effect of simplification. You don’t need to worry whether the latter operaion may have a negative effect — everything will be back in the Final Fold stage.

Part 6.1 Rotate Simplification

This step is vital when you need to reduce the corner cases in the expression tree, because this step rotates the expression tree to help reanalyze the tree.

Briefing:

This step traverses the whole expression postfix. After Rotate Simplification of both children, it applies many formulae to rotate the tree and turn most uncommon cases into common cases. After rotating, the steps afterwards only needs to care about the plain cases.

At the same time, it eliminates function pow and log with operator ^ and ln. This step is also designed for reducing the corner cases to care about. Therefore, in the following steps only the most common and major cases are considered.

Note:

It must be applied before applying 0-1 Simplification ,or patterns like x*(1/y) won’t have 1 reduced. It seems that with Negative Simplification, some of the rotations are useless. However, It does not always appear along with Negative Simplification. For example, in the Final Fold stage, this step take an irreplaceable part to avoid the corner cases difficult to simplify.

Formulae:

```
(a-b)+(c-d)=(a+c)-(b+d)
x+(y-z)=(x+y)-z
(x-y)+z=(x+z)-y
(a-b)-(c-d)=(a+d)-(c+b)
x-(y-z)=(x+z)-y
(x-y)-z=x-(y+z)
```

```
(a/b)*(c/d)=(a*c)/(b*d)
x*(y/z)=(x*y)/z
(x/y)*z=(x*z)/y
(a/b)/(c/d)=(a*d)/(c*b)
x/(y/z)=(x*z)/y
(x/y)/z=x/(y*z)
(x^a)^b=x^(a*b)
pow(x,y)=x^y
log(x,y)=ln(y)/ln(x)
```

For the main logic and the technical details, see function `Simplify_Rotate` at **Line 2577** and other helper function involved.

Part 6.2 0-1 Simplification

This step is vital for reducing the subtrees obviously OK to discard composed of 0 and 1.

Briefing:

0 and 1 are always generating when the program calculates the partial derivative since they are the very end of linked calculation. Meanwhile, 0 and 1 also let the operators and functions turn to some special values. This step is designed for reducing 0 and 1 appearing all the time.

Formulae:

```
ln(1)=0
exp(0)=1
cos(0)=1
cosh(0)=1
sin(0)=0
tan(0)=0
sinh(0)=0
x^0=1
0^x=0
1^x=1
```

```
x^1=x
0-x=-1*x
```

For the main logic and the technical details, see function `Simplify_01` at **Line 2447** and other helper function involved.

Part 6.3 Negative Simplification

This step is designed for eliminate the operartor – and / interfering the simplification process.

Briefing:

Mostly this step converts minus an expression to add its opposite and divide an expression to multiply its reciprocal. Although most of the time this step lengthens the expression, this process is necessary is important when you try to use the Polynomial Simplification. The extra parts generated in this step will be shortened in the Negative Final Fold step.

Formulae:

```
C/x^y=C*x^((-1)*y)
C/exp(x)=C*exp(-1*x)
C/x=C*x^(-1)
(a+b)*C=C*a+C*b
C*(a+b)=C*a+C*b
(-1)*(-1)=1
a-b=a+(-1)*b
a-(-1)*b=a+b
sin(-1*x)=-1*sin(x)
cos(-1*x)=cos(x)
tan(-1*x)=-1*tan(x)
sinh(-1*x)=-1*sinh(x)
cosh(-1*x)=cosh(x)
```

For the main logic and the technical details, see function `Simplify_Neg` at **Line 2940** , function `Simplify_FinalNeg` at **Line 3084** and other helper function involved.

Part 6.4 Special Function Simplification

This section focuses on simplifying expressions containing special mathematical functions like **logarithms, exponentials, and trigonometric functions**.

Briefing:

These simplifications leverage known mathematical identities to reduce complexity and improve the analytical quality of the results. The goal is to transform these functions into more basic operations or other functions with simpler forms while preserving mathematical equivalence. The simplification rules applied here are based on standard mathematical identities and properties of these special functions. These transformations not only reduce the complexity of expressions but also help in making the analytical results more interpretable.

Note:

The x in the formulae below is judged by pattern matching in hash. This expands the generality of this algorithm.

Formulae:

```
exp(ln(x))=x

ln(exp(x))=x

ln(a^b)=b*ln(a)

sin^2+cos^2=1

cos^2+sin^2=1

1-sin^2=cos^2

1-cos^2=sin^2

cosh^2-sinh^2=1

sinh^2+1=cosh^2

1+sinh^2=cosh^2

cosh^2-1=sinh^2

sin/cos=tan

cos/sin=1/tan
```

For the main logic and the technical details, see function `Simplify_SpecialFuncs` at **Line 2750** and other helper function involved.

Part 6.5 Polynomial Simplification

In the 8 types of simplification, Polynomial Simplification accounts for most of the time and lines of codes. However this is the most vital as we have to say that it make a great difference to the final output becauses it truly **calculates the expression** rather than simply matching the form and transforming the tree.

Briefing:

This step is then divided into 4 smaller stages. This step switches round and round between monomial folding and polynomial folding when it found different operators. When it comes to a polynomial associated with +, this algorithm merge the subtree with the distributive law and some other methods.

Stage 1: this algorithm recursively handles the monomials in the processing polynomial by a postfix traversal. This step turns the monomials partly simpler for the following steps.

Stage 2: this stage separates the coefficient from the monomials. The rest of the monomials are then simplified again by merging the powers of a subrtree like x*x $\rightarrow$ x^2. After this, Rotate, 0-1 and Negative Simplifications are reapplied.

Stage 3: this stage combines like terms by simply checking the rest part of each algorithm.

Stage 4: with the distributive law, the algorithm search for common subexpressions and then merge them. After that, the coefficients are restored.

Note:

The x in the formulae below is judged by pattern matching in hash. This expands the generality of this algorithm.

Formulae:

```
(y*z)^x=y^x*z^x

x^a*x^b=x^(a+b)

C*x+C'*x=(C+C')*x

x*y+x*z=x*(y+z)
```

For the main logic and the technical details, see function `Simplify_Polynomial` at **Line 3247** , function `Simplify_Monomial_I` at **Line 3185** , function `Simplify_Monomial_II` at **Line 3213** , function `GetCommonFactor` at **Line 2342** , function `Simplify_Monomial_III` at **Line 3229** and other helper function involved.

Part 6.6 Fold Constant Simplification

Constants cannot be just forgotten somewhere or simply pass them directly to the final result. The real simplest part needs a simple form of coefficient and constants.

Briefing:

This step focuses on identifying and evaluating constant expressions within the tree to replace them with their computed values, thereby reducing the complexity of the expression and improving computational efficiency.

Note: although in this step the operator ^ is directly replaced with standard library function pow, this doesn't lose precision because the program will fall into this part only if both operands are integers. It is always guaranteed that no irrational numbers will be created as a single node as we seek for a analytical result.

Formulae: No specific formula, depending on the current constant status.

For the main logic and the technical details, see function `Simplify_FoldConst` at **Line 3124** , function `RotateCoefficient` at **Line 3084** , function `ExtractCoefficient` at **Line 2174** and other helper function involved.

Part 6.7 GCD Final Fold

To some extent, this step is the inverse step of Polynomial Simplification.

Briefing:

This is a short step. Same as Polynomial Simplification, this step switches round and round between monomial folding and polynomial folding when it found different operators. When it comes to a polynomial associated with +, this function tries to extract the greatest common divisor (GCD) of coefficients. This means this function may reread the constants generated before.

Note:

Because the simple criteria of searching for polynomial tend to have no effect after Negative Final Fold as – and / go back, this step must be applied before applying Negative Final Fold.

Formulae:

```
y^x*z^x=(y*z)^x
Ax+By=M((A/M)x+(B/M)x) M=gcd(A,B)
```

For the main logic and the technical details, see function `FinalFold_GCDPoly` at **Line 3643** , function `FinalFold_GCDMono` at **Line 3571** and other helper function involved.

Part 6.8 Negative Final Fold

This step is to some extent the negative version of Negative Simplification. When the previous steps introduced negative powers and adding negative expressions, this step replace them with reintroduced operator “-” and “/”.

Briefing:

This step focuses on final adjustments to create negative signs and divisions that may have been eliminated during earlier simplification steps. It transforms expressions to ensure that negative signs are applied to the fewest possible terms and divisions are represented as multiplications by reciprocals wherever beneficial for further simplification or readability. This is the last step as it improves readability while interferes simplification.

Formulae:

```
x^((-1)*y)=1/x^y
(-1)*a+(-1)*b=(-1)*(a+b)
a+(-1)*b=a-b
(-1)*a+b=b-a
 a-(-1)*b=a+b
(-1)*a-b=(-1)*(a+b)
(-1)*a-(-1)*b=b-a
sin(-1*x)=-1*sin(x)
cos(-1*x)=cos(x)
tan(-1*x)=-1*tan(x)
pow(x,-1*y)=1/x^y
exp(-1*x)=1/exp(x)
sinh(-1*x)=-1*sinh(x)
cosh(-1*x)=cosh(x)
```

For the main logic and the technical details, see function `FinalFold_Neg` at **Line 3384** , function `Simplify_FinalNeg` at **Line 3084** and other helper function involved.

Part 7: Output

This part (Mainly **Line 1815~1871**) Print a binary expression tree to plain text close to natural writing within a infix traversal.

By using `std::map` to sort the variables automatically, here the output will naturally by **lexicographical order**.

Note: Here the algorithm reduces the sign * between a integer and another type of token. If you dislike this, please remove **Line 1845.**

Process in Pseudocode:

In main function:

```
print(“{Var} :”)
PrintTree(PartialExprTree)
```

The Function PrintTree:

Here default : Parent is nullptr, IsLeft is false(details see **Line 508**)

```
function PrintTree(Node as ExprNode, Parent as ExprNode, IsLeft as bool)
     if Node is null then
        return
    switch Node.Type do
        case Token::Int:
            if Node.Value < 0 or (Parent is Operator and IsLeft) then
                print '(' + Node.Value + ')'
            else
                print Node.Value
        case Token::Variable:
            print Node.Name
        case Token::Function:
             print Node.FuncName + '(' PrintTree(Node.Left, Node, true)
            if Node.FuncParamCount == 2 then
                print ',' PrintTree(Node.Right, Node, false)
                print ')'
        case Token::Operator:
            if NeedsBracket(Node, Parent, IsLeft) then
                print '(' PrintTree(Node.Left, Node, true)
(see next page)
(continue prev page)
          if Node.Operator is '' and
             (Node.Left is Int and Node.Right is not Int) then
          // skip printing ''
          else
             print Node.Operator
          PrintTree(Node.Right, Node, false)
          if NeedsBracket(Node, Parent, IsLeft) then
             print ')'
```

Helper function NeedsBracket:

Here I used a rather complex method to minimize the use of brackets to take closest to the natural expression writing.

```
function NeedsBracket(Node as ExprNode, Parent as ExprNode, IsLeft as bool)
return bool
   if Parent is null then
      return false
   if Parent.Operator is '^' and IsLeft then
      // right-associated like x^y^z → x^(y^z)
      return true
   if Parent.Precedence > Node.Precedence then
      return true
   if Parent.Precedence == Node.Precedence then
      if (Parent.Operator is '-' or '/' and not IsLeft) then
         // right operand priority like a-b-c & (a-b)-c
         return true
      return false
```

Here the tree is processed recursively and outputted by infix order. The complicated codes are just for reducing the unnecessary brackets. Both the priority of the parent operator and the relative position of the subtree is considered.

We need IsLeft parameter because if a-b occurs on the left of a “-”, no brackets are needed, but when it occurs on the right, brackets are critical to get rid of ambiguity.

Chapter 3:   Testing Results

Note:

1. (**usually it won’t happen**)if you **simply copy** the input, on **some of the computers**

the answer might be INCORRECT due to the PDF format. If such situation occurs on your computer, please type the test cases MANUALLY.

2. The constant coefficient will be directly linked to a variable/bracket/function. e.g. **10*x will be shown as 10x**. If you dislike this, please remove **Line 1845.**

```
1845      if(!(V.ID=='*'&&Ty0()==Token::Int && Ty1()!=Token::Int))putchar(V.ID);
```

Test Case 1:

Input:

```
a+b^c*d
```

Output:

```
a: 1
b: c*b^(c-1)*d
c: ln(b)*b^c*d
d: b^c
```

Current Status: Correct

Test Case 2:

Input:

```
a*10*b+2^a/a
```

Output:

```
a: 10b+2^a*(ln(2)*a-1)/a^2
b: 10a
```

Current Status: Correct

Test Case 3:

Input:

```
xx^2/xy*xy+a^a
```

Output:

```
a: (ln(a)+1)*a^a
xx: 2xx
xy: 0
```

Current Status: Correct

Test Case 4:

Input:

```
x*ln(y)
```

Output:

```
x: ln(y)
y: x/y
```

Current Status: Correct

Test Case 5:

Input:

```
x*ln(x*y)+y*cos(x)+y*sin(2*x)
```

Output:

```
x: ln(x*y)+1+y*(2cos(2x)-sin(x))
y: x/y+sin(2x)+cos(x)
```

Current Status: Correct

Test Case 6:

Input:

```
log(a,b)/log(c,a)
```

Output:

```
a: -2ln(b)*ln(c)/(a*ln(a)^3)
b: ln(c)/(b*ln(a)^2)
c: ln(b)/(c*ln(a)^2)
```

Current Status: Correct

Test Case 7:

Input:

```
a^b^c
```

Output:

```
a: b^c*a^(b^c-1)
b: c*b^(c-1)*ln(a)*a^b^c
c: ln(b)*b^c*ln(a)*a^b^c
```

Current Status: Correct

Test Case 8:

Input:

```
x/5+y/10
```

Output:

```
x: 1/5
y: 1/10
```

Current Status: Correct

Test Case 9:

Input:

```
x^2*exp(x)
```

Output:

```
x: exp(x)*(x^2+2x)
```

Current Status: Correct

Test Case 10:

Input:

```
pow(x,-1*y)
```

Output:

```
x: -y/x^(1+y)
y: -ln(x)/x^y
```

Current Status: Correct

Test Case 11:

Input:

```
2x+2y+2z+2a+2b+2x
```

Output:

```
a: 2
b: 2
x: 4
y: 2
z: 2
```

Current Status: Correct

Test Case 12:

Input:

```
(y+1)/(y-1)
```

Output:

```
y: (-2)/(y-1)^2
```

Current Status: Correct

Test Case 13:

Input:

```
(x/y)/(x*(y/(x/(z/(y/x)/x)))/z)
```

Output:

```
x: 1/y
y: -x/y^2
z: 0
```

Current Status: Correct

Test Case 14:

Input:

```
114514*x*ln(exp(log(y, cos( sin (tan(0))))))
```

Output:

```
x: 0
y: 0
```

Current Status: Correct

Test Case 15:

Input:

```
x^x^x
```

Output:

```
x: ((ln(x)+1)*x^x*ln(x)+x^(x-1))*x^x^x
```

Current Status: Correct

Test Case 16:

Input:

```
(x+y)^3*x
```

Output:

```
x: 3(x+y)^2*x+(x+y)^3
y: 3(x+y)^2*x
```

Current Status: Correct

Test Case 17:

Input:

```
longparamI/ln(longparamII)
```

Output:

```
longparamI: 1/ln(longparamII)
longparamII: -longparamI/(longparamII*ln(longparamII)^2)
```

Current Status: Correct

Test Case 18:

Input:

```
tan(x/y)
```

Output:

```
x: 1/(y*cos(x/y)^2)
y: -x/(y*cos(x/y))^2
```

Current Status: Correct

Test Case 19:

Input:

```
(5x)^(1/2)
```

Output:

```
x: x^((-1)/2)*5^(1/2)/2
```

Current Status: Correct

Test Case 20:

Input:

```
(y+1)/(y-1)*(x+1)/(x-1)
```

Output:

```
x: -2(y+1)/((y-1)*(x-1)^2)
y: -2(x+1)/((x-1)*(y-1)^2)
```

Current Status: Correct

Test Case 21:

Input:

```
(exp(-y)*cos(x)^2)-(exp(-y)*sin(x)^2)
```

Output:

```
x: -4sin(x)*cos(x)/exp(y)
y: -(cos(x)^2-sin(x)^2)/exp(y)
```

Current Status: Correct

Test Case 22:

Input:

```
x^y^z
```

Output:

```
x: y^z*x^(y^z-1)
y: z*y^(z-1)*ln(x)*x^y^z
z: ln(y)*y^z*ln(x)*x^y^z
```

Current Status: Correct

Chapter 4: Analysis and Comments

Time Complexity:

To Simplify a tree with **M** nodes, since the Polynomial Simplification has to compare the trees one-by-one, so the Simplification time complexity is **$O(M^2)$**.

Generally, the time complexity is **$O(M^2 *V*U)$** for each input, where:

- **M** is the total number of nodes in the expression tree.
- **V** is the number of unique variables in the input expression.
- **U** is the round of simplification, depending on the form of the expression.

Further Analysis:

Operations except '+' outputs **$O(K^2)$** nodes when we input **K** nodes. We can consider that **K** is **$O(N)$** since the number of nodes is always increasing when **N** rises(here **N** is the length of the original expression).

Therefore, We can roughly say that **M** is **$O(N^2)$**in the common cases. For the worst cases, **$K \approx N$**，the maximum time complexity is **$O(N^2)$** as well.

V is weakly relevant to **N**. so in the average case, we can only say that the time complexity is **$O(N^4 * V * U)$**. However, in the worst case, the maximum time complexity will be **$O(N^5 * U)$**.

Space Complexity: As tested(see Time Complexity), the intermediate nodes is **$O(N^2)$** where **N** is the length of the original expression. Other space cost is no more than **$O(N^2)$**, so we can conclude that **$O(N^2)$** is the total space complexity.

Strengths:

1. **Modular Expression Tree Architecture**
The implementation employs a hierarchical expression tree structure, enabling recursive differentiation through well-defined node types (operators, functions, variables). This modular design simplifies the application of differentiation rules (e.g., chain rule for functions) and ensures clarity in handling nested expressions.

2. **Memory Efficiency via Node Reuse**
The UnusedNode pool and Nodes vector minimize dynamic memory allocation overhead by reusing deallocated nodes. This approach reduces fragmentation and improves performance for repeated parsing/differentiation tasks.

3. **Basic Algebraic Simplification**
The code implements foundational simplification rules (e.g., x^0 → 1, 0*x → 0) through the Simplify module, reducing redundant terms early in the process. This lightweight optimization enhances readability of derivative expressions without heavy computational cost.

4. **Operator Precedence Handling**
The tokenization and tree construction phases rigorously respect operator precedence (e.g., ^before*before+), ensuring accurate parsing of complex infix expressions like a+b^c*d.

Weaknesses:

1. **Recursion Depth Limitations**

The recursive tree traversal for differentiation and simplification may cause stack overflow for deeply nested expressions (e.g.,f(g(h(...))) with too many layers), limiting scalability.

2. **Error Handling Gaps**
Input validation is minimal—malformed expressions (e.g., unmatched brackets, invalid function arguments) often lead to ambiguous Error Report without clear hints and error indications.

3. **Inefficient Hash-Based Comparisons**
The reliance on ExprHash for node equivalence checks, while fast, risks hash collisions in large expressions, potentially causing incorrect simplifications.

Potential Optimizations

4. **Iterative Tree Traversal**
Replace recursive traversal with an iterative approach using stacks to mitigate recursion depth limits and improve performance for deep expressions.

5. **Pattern-Based Simplification Engine**
Implement a rule engine with pattern matching (e.g., rewrite sin^2(x) + cos^2(x) → 1) to enable deeper algebraic simplifications, leveraging tools like term rewriting systems.

6. **Symbolic Differentiation Cache**
Cache derivatives of common subexpressions (e.g., d/dx (ln(x))) to avoid redundant calculations, particularly beneficial for expressions with repeated terms.

7. **Parallel Subtree Processing**
For multi-variable derivatives, process independent subtrees (e.g., terms in additive expressions) concurrently using thread pools to leverage modern multi-core architectures.

8. **Enhanced Error Diagnostics**
Integrate detailed error reporting with context-aware messages (e.g., "Mismatched bracket at position 12") by tracking token positions during parsing.

Appendix: Source Code

See code folder for source code. The code is too long, so it won't be displayed here.

Usage: See readme.txt

File: AutoGrad.cpp

Note: the source code **MUST** be compiled with **C++17** or higher versions. Dev-C++ may fail to compile this program if internal GCC version is too low to support C++17. Check your C++ version when a compile error occurred.

Declaration

I hereby declare that all the work done in this project titled "Project 2" is of my independent effort.

Please keep in mind that these are the “maximum” requirements. Other requirements will be specified according to each project assignment.