Fundamentals of Data Structures

Laboratory Projects

Autograd for Algebraic Expressions

Author's Name

Date: 2023-11-09

Chapter 1 Introduction

With the development of Machine Learning, the application of automatic differentiation technology such as torch and tensorflow has greatly facilitated people's implementation and training of Deep Learning algorithms based on back propagation.

First, we generally introduce some of the diffrent differentiation methods being used.

Different Differentiation Methods

- Numerical gradient checking

We directly compute the partial gradient by definition.

$$\frac{\partial f(\theta)}{\partial \theta_i} = \lim_{\epsilon \to 0} \frac{f(\theta + \epsilon e_i) - f(\theta)}{\epsilon} \tag{1}$$

However, by definition, we can only calculate the numerical answer, but not for functions and expressions. Also, it will bring significant numerical error, and is less efficient to compute.

- Symbolic differentiation

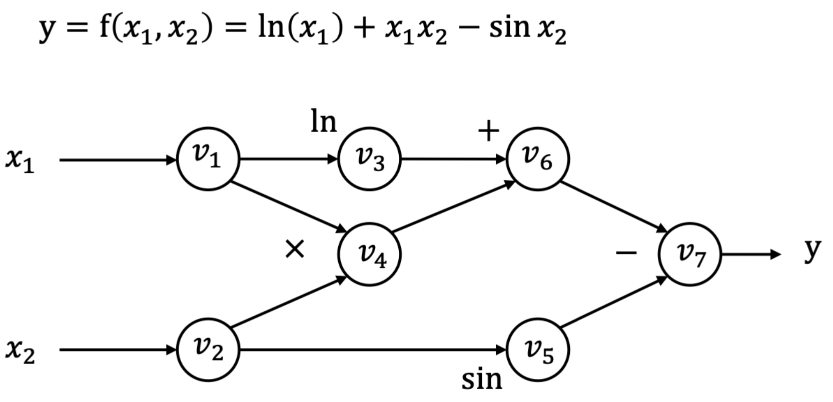
We write down the formulas, derive the gradient by sum, product and chain rules

$$\frac{\partial f(g(\theta))}{\partial \theta} = \frac{\partial f(g(\theta))}{\partial g(\theta)} \frac{\partial f(g(\theta))}{\partial \theta} \qquad \frac{\partial (f(\theta) g(\theta))}{\partial \theta} = g(\theta) \frac{\partial f(\theta)}{\partial \theta} + f(\theta) \frac{\partial g(\theta)}{\partial \theta} \tag{2}$$

However, doing this naively can result in wasted computations.

- Computational graph

We convert the function into a graph.



We define $\dot{v}_i = \frac{\partial v}{\partial x}$, and iteratively compute $\dot{v}_i$ in the forward topological order of the computational graph.

Trace back to our problem here, we hope to implement an automatic differentiation program for algebraic expressions. And based on what we have learned, I come up with the following idea to solve the problem:

1. Analyze the expression and build a expression tree
2. Apply differentiation rules of different formulas and chain rules recursively of the expression tree and build a new tree
3. Print the new expression tree out.

We should rule out the **input format** and **output format**, which is important to our further implementation.

Input Format

The expression we input must be an **infix expression**, and an expression consists of **operators** and **operands**.

Operators

Type	Examples	Notes
Bracket	()	-
Power	^	-
Multiplication & Division	* /	-
Addition & Subtraction	+ -	-
Argument separatot	,	**optional**, only used as argument separators in multivariate functions

The above operators are arranged in order of **operator precedence from top to bottom**. For example, `a+b^c*d` will be considered the same as `a + ( (b ^ c) * d )`.

Operands

Type	Examples	Notes
Literal constant	2 3 0 -5	Just consider integers consisting of pure numbers and minus signs.
Variable	ex cosval xy xx	Considering the above "mathematical functions" as reserved words, identifiers (strings of lowercase English letters) that are not reserved words are called variables.

Output Method

For each **variable** (as defined above) that appears in the expression, describe an arithmetic expression that represents the derivative of the input algebraic expression with respect to that variable, using the operators, mathematical functions, and operands defined in the input form.
The output is arranged according to the **lexicographical order** of the variables. For each line print two strings,

which are each variable and the corresponding derivative function. Separate the two strings with `:`.

- **Example:**
  - if the expression is `(c+a)*b`, then we should print in the following format:
  - `a : b`
  - `b : c + a`
  - `c : b`

Requirements

1. Process a given algebraic expression containing **operators** and **operands**, and output the algebraic expression of the derivative function of the original expression with respect to each variable. (Already explained in the **Output Method**)
2. It is only necessary to ensure that the output expression is **identical** to the correct derivative function, and no simplification or expansion of the expression is required.

   For example, if you type in the algebraic expression `x^2/y*y`, and the differentive function of `x` is `2*x`, but the answer you see is `2*x*y/y^2*y`, please be a little more patient and simplify it a little bit, because the program may not be that intelligent as you.
3. Implemented in C/C++, only the standard library can be used.

   This means libraries like `<vector><stack><string><algorithm>` can be used in `C++`. I have to admit that finishing the project without `STL` is really a headache because you have to deal with everything deep down while thinking of the abstract algorithms.
4. The code is highly readable or well-commented. Please make sure the source files can be opened correctly with utf-8 encoding.

Bonus:

- Not using complex containers provided in STL **(Not done, which is really a tough job)**
- Support for expressions containing mathematical functions **(Finished)**

  Mathematical Functions

Function	Description
ln(A) log(A,B)	logarithm. ln(A) represents the natural logarithm of A, and log(A, B) represents the logarithm of B based on A.
cos(A) sin(A) tan(A)	basic trigonometric functions.
pow(A,B) exp(A)	exponential functions. pow(A, B) represents the B power of A, and exp(A) represents the natural exponential of A.

- Simplify an algebraic expression to reduce the length of the result by **applying at least two rules**. **(Finished)**

Chapter 2 Algorithm Specification

To make the algorithms a bit more logically organized, we first draw a sketch of the the whole program

Sketch of the program

- Input the string `expression`
- `Tokenize` the expression (analyze and separate the operators and operands, store them into `vector<string>` in read-in order)
  - Iterate through the expression
  - Find the length of the current operand (operator)
  - Save the operand (operator) into the vector
- `Parse` the `expression` (build a tree according to expression `vector`)
  - Iterate through the tokens in the vector and convert the infix expression to postfix expression (which is easier to build a tree) by using stacks
  - Iterate through the postfix expression and build a tree by using stacks
  - deal with unary operators
- Iterate through the tree and get the variables, store the variables into a vector `variables`
- Remove the duplicate variables and sort them in lexicographical order
- Iterate through the variables and do three things
  - `Autograd()`
    - Recursively investigate into every node (definitely require multiple if-else statements)
      - if it is an `operand`
        - main variable - return 1
        - partial variable - regard it as a constant and return 0
      - If it is an `operator`
        - apply the rule according to the operator
  - `SimplifyTree()`
    - Recursively investigate into every node and remove redundant nodes
  - `PrintTree()`
    - Recursively investigate into every node
    - Check if a parenthesis is needed

Note that there are too many functions in the program and some of them are the **wheels** I build to simplify the structure of those more important functions, so I omit the analysis of some small functions.

Main Data Structure

- tokens: `vector<string>`
  - Under it is a list of array, each containing a string
- variables: `vector<string>`
- Tree: `class(value, Tree *left, Tree *right)`
  - recursive definition of a class

```
Class Tree
    Public String value
    Public Tree left
    Public Tree right

    Constructor Tree(value(value), left(null), right(null))
    Constructor Tree(value(value), left(left tree), right(right tree))
End Class
```

Algorithms

Tokenize()

```
function Tokenize(expression):
    tokens = an empty list of strings

    while i < length of expression:
        // call the isValid function and it will return the length of the token
        length = isValid(expression, i)

        if length is not equal to 0:
            token = substring of expression from index i with length

            // If the minus operator is a unary operator, we tag it as a '#' operator
            if first character of token is '-' and length > 1:
                set '-' to '#'

            append token to the vector
            i = i + length    // update the iterator
        else:
            // if the token is invalid
            print "Invalid token"
            i = i + 1
    return tokens
```

- We input an expression as a string, and we have to process it to sth our program can analyze - that is to break the string into individual units so we can classify and divide them to conditions we are able to cope with. So we iterate through the string, find out the length of the following string, and store it as a token.
- Thus, we have to mention the core function `isValid ()` here.

isValid ()

```
function isValid(expression, i)
    length = 0
    token = expression.substring(i)

    if token[0] is a digit
        // Iterate until we encounter a non-constant character
        for each character in token
            if character is a digit
                length = length + 1
            else
                break
    else if isUnaryOperator(expression, i)
        length = unaryExpressionParser(token)
    else if isBinaryOperator(token)
        // If the token is a binary operator
        // The length of the token is definitely 1
        length = 1
    else if token[0] is a letter
        // If the token is a variable
        if token starts with "cosval"
            length = 6
        else if token starts with "ex" or "xx" or "xy"
            length = 2
        else
            length = 1

    return length
```

- We divide the unit into following conditions
  - constant digits (e.g. 1, 200)
  - unary operator (e.g. '-' in −1, `ln(a)`,`sin(b)`)
  - binary operator (e.g. `+, *, /`)
  - variable (e.g. `x, xy, a`)
- And unary operators are actually very special, so we bring a special method to process it, that is we **regard the operator and the operand it operate on as a whole**

```
function unaryExpressionParser(c)
```

```
    token = ""
    adder = 0

    if c starts with "ln"
        token = c.substring(2)
        adder = 2
    else if c starts with "log" or "pow" or "exp" or "sin" or "cos" or "tan"
        token = c.substring(3)
        adder = 3
    else if c[0] is '-'
        token = c.substring(1)
        adder = 1

    if token[0] is '('
        i = 1
        count = 1
        while count is not 0
            if token[i] is '('
                count = count + 1
            else if token[i] is ')'
                count = count - 1
            i = i + 1
        end while
        return i + adder
    else
        return isValid(token, 0) + adder
```

- And here comes two new conditions
    - cases like `-1`, `sin(a)`, which are easy, we simply have to anaylze the operator and stop at its end
    - cases like `-(a+b)`, `sin(a*(b+c))`, which are more difficult, and we have to use another `while` loop to match the parenthesis, rather than simply end at the first parenthesis

parseExpression()

- After tokenizing, we then start to parse the expression and build the expression tree

```
function parseExpression(tokens)
    treeStack = empty stack
    opStack = empty stack
    postfix = empty list

    for each token in tokens
        if token is not a binary operator
            append token to postfix
        else
            if token is "("
                push token to opStack
            else if token is ")"
                while opStack is not empty and opStack top is not "("
                    append opStack top to postfix
                    pop opStack top

                pop opStack top
            else
                // If we meet a binary operator, we pop operators with higher
precedence and push the current operator into the stack
                while opStack is not empty and Precedence(token) <= Precedence(opStack
top)
                    append opStack top to postfix
                    pop opStack top
                push token to opStack

    // Pop any remaining operators in the stack and add them to the postfix expression
    while opStack is not empty
        append opStack top to postfix
        pop opStack top

    // Construct the expression tree from the postfix expression
    for each token in postfix
        if token is a variable or a constant
                push new Tree(token) to treeStack
            else if token is an unary operator
                if token starts with "#"
                    // mention that we have convert unary '-' to '#' for easier
identification
                    temp = Tokenize(token substring from position 1)
                    tempTree = parseExpression(temp)
                    op = new Tree("neg", tempTree, nullptr)
                    push op to treeStack
                else if token starts with "ln"
                    temp = Tokenize(token substring from position 2)
                    tempTree = parseExpression(temp)
                    op = new Tree("ln", tempTree, nullptr)
                    push op to treeStack
                else if token starts with "exp" or "sin" or "cos" or "tan"
                    temp = Tokenize(token substring from position 3)
                    tempTree = parseExpression(temp)
                    op = new Tree(token substring from position 0 to 3, tempTree,
nullptr)
                    push op to treeStack
                else if token starts with "log" or "pow"
                    pos = 4
                    count = 1

                    while count is not 1 or token[pos] is not ","
                        if token[pos] is "("
                            count = count + 1
                        else if token[pos] is ")"
                            count = count - 1
```

```
                          pos = pos + 1

                      temp1 = Tokenize(token substring from position 4 to pos - 4)
                      temp2 = Tokenize(token substring from position pos + 1 to end of
token - 2)
                      tempTree1 = parseExpression(temp1)
                      tempTree2 = parseExpression(temp2)
                      op = new Tree(token substring from position 0 to 3, tempTree1,
tempTree2)
                      push op to treeStack
        else
            // Here the token must be a binary operator
            right = treeStack top
            pop treeStack top
            left = treeStack top
            pop treeStack top
            op = new Tree(token)
            op left = left
            op right = right
            push op to treeStack

    // The root of the final expression tree is at the top of the stack
    return treeStack top
```

- To make the function a bit easier to understand, we first throw the unary operators aside and things become clear and easy. Here are the two things we do:
    1. convert the infix expression into a postfix expression
    2. construct the postfix expression into an expression tree
- This has been taught in class, and we can review a little bit here

- We need a stack to temporarily store the operators and a list to store the final expression
    - If we meet a variable, directly throw it into the list
    - If we meet a operator, we have to examine it along with the stack
        - If it is a left parenthesis, then it has highest precedence and we just push it into stack (**however, when it is in the stack, it is actually of lowest precedence**). If it is a right parenthesis, we start to pop the stack **until we meet the left parenthesis**
        - If there is nothing in the stack, just throw it into the stack
        - If the stack is not empty, we should compare it with the top element
            - If it is of higher precedence, just push it into stack
            - If not, we should pop the elements in the stack until we meet a element of lower precedence (or empty), then push it into stack
    - Last, pop the remaining operators in the stack (if there is any)

- After that, we start to build the tree, we need a stack to temporarily store the nodes and here is what we are going to do
  - Start from the `postfix` list
  - If it is a variable, build a node from it and push it into the stack
  - If it is an operator, build a node from it and this time it has **two children**, and they are the **top two nodes in the stack**. After building this, we push the node into the stack
  - We repeat the process until all elements in the list has been traversed and the node remaining in the stack is the root of the expression tree we are eager for

- The above content is easy, but don't forget the **unary operators**, now we try to add the analysis of it into the above program.
  - When converting the infix expression into a postfix one, we simply regard it as a **operand** (which makes sense, because it is single). Why we do so is to guarantee that it is put into the right position, which is very important.
  - When building the tree, we can't regard it as a simple operand anymore, because it actually contains more information inside and has to be differentiated if variables are contained in it. So we do two things here
    - We find the unary operator out and build an **op node** with **one child containing the expression inside it and another child hanging up**.
    - For the expression inside it, we just need to **recursively call Tokenize() and parseExpression()**, because it is just like the string we input at the very begining!
  - BUT, there are special cases like `pow` and `log`, which contains two operand inside, so we put the two operands into the left and right child for them. Remind that parenthesis has to be carefully inspected.

Autograd()

- With the expression tree in hand, we can start differentiation!

```
function autoGrad(root, var, variables)
    if root is null
        return null

    // Initialization
    lc = left child of the tree
    rc = right child of the tree

    // Recursively calculate the derivative of the left subtree and the right subtree
    ld = autoGrad(root left, var, variables)
    rd = autoGrad(root right, var, variables)

    if root value is the main variable
        // Rule: d/dx(x) = 1
        return new Tree("1")
    else if root value is a constant or a partial variable
```

```
            // Rule: d/dx(const) = 0
            return new Tree("0")
        else if root value is "+"
            // Rule: d/dx(f + g) = d/dx(f) + d/dx(g)
            return new Tree("+", ld, rd)
        else if root value is "-"
            // Rule: d/dx(f - g) = d/dx(f) - d/dx(g)
            return new Tree("-", ld, rd)
        else if root value is "*"
            // Rule: d/dx(f * g) = f * d/dx(g) + g * d/dx(f)
            result = new Tree("+")
            result left = new Tree("*", ld, rc)
            result right = new Tree("*", lc, rd)
            return result
        else if root value is "/"
            // Rule: d/dx(f / g) = (d/dx(f) * g - f * d/dx(g)) / g ^ 2
            result = new Tree("/")
            result left = new Tree("-", new Tree("*", ld, rc), new Tree("*", lc, rd))
            result right = new Tree("^", rc, new Tree("2"))
            return result
        else if root value is "^"
            // Rule: d/dx(f ^ g) = f ^ g * (d/dx(g) * ln(f) + g * d/dx(f) / f)

            // Check if the base and the exponent are constants
            isBaseConstant = checkConstant(root left, var, variables)
            isExponentConstant = checkConstant(root right, var, variables)

            if root left value is "e"
                // If the base is e, simplify the expression
                // Rule: d/dx(e ^ g) = e ^ g * d/dx(g)
                result = new Tree("*")
                result left = new Tree("^", lc, rc)
                result right = autoGrad(rc, var, variables)
                return result
            else if isBaseConstant and isExponentConstant
                // If both the base and the exponent are constants
                // Rule: d/dx(const ^ const) = 0
                return new Tree("0")
            else if isBaseConstant
                // If the base is a constant
                // Rule: d/dx(const ^ g) = const ^ g * (d/dx(g) * ln(const))
                result = new Tree("*")
                result left = new Tree("log", new Tree("e"), lc)
                result right = new Tree("^", lc, rc)
                return result
            else if isExponentConstant
                // If the exponent is a constant
                // Rule: d/dx(f ^ const) = (const - 1) * f ^ (const - 1) * d/dx(f)
                result = new Tree("*")
                result left = new Tree("*", autoGrad(lc, var, variables), rc)
                result right = new Tree("^", lc, new Tree("-", rc, new Tree("1")))
                return result
            else
                // If both the base and the exponent are variables
                // Rule: d/dx(f ^ g) = f ^ g * (d/dx(g) * ln(f) + g * d/dx(f) / f)
                result = new Tree("*")
                result left = copyTree(root)
                resultRight = new Tree("+")
                logTree = new Tree("log", new Tree("e"), lc)
                resultRight left = new Tree("*", rc, autoGrad(logTree, var, variables))
                resultRight right = new Tree("*", logTree, autoGrad(rc, var, variables))
                result right = resultRight
                return result
        else if root value starts with "ln"
            // Rule: d/dx(ln(f)) = d/dx(f) / f
            result = new Tree("/")
            result left = autoGrad(lc, var, variables)
            result right = lc
            return result
        else if root value is "log"
            // Apply the logarithm rule and convert log to ln
            // Then recursively call the autoGrad function
            if root left value is "e"
                // Turn log to ln and use the rule of ln
                result = autoGrad(new Tree("ln", rc, null), var, variables)
                return result
            else
                // Rule: log(f, g) = ln(g) / ln(f)
                // Turn log to ln and use the rule of ln
                result = autoGrad(new Tree("/", new Tree("ln", rc, null), new Tree("ln",
    lc, null)), var, variables)
                return result
        else if root value is "pow"
            // Turn pow to ^ and use the rule of ^
            return autoGrad(new Tree("^", lc, rc), var, variables)
        else if root value is "exp"
            // Turn exp to e ^ and use the rule of ^
            return autoGrad(new Tree("^", new Tree("e"), lc), var, variables)
        else if root value is "sin"
            // Rule: d/dx(sin(f)) = cos(f) * d/dx(f)
            result = new Tree("*")
            result left = autoGrad(lc, var, variables)
            result right = new Tree("cos", lc, null)
            return result
        else if root value is "cos"
            // Rule: d/dx(cos(f)) = -sin(f) * d/dx(f)
            result = new Tree("*")
            result left = autoGrad(lc, var, variables)
            result right = new Tree("neg", new Tree("sin", lc, null), null)
            return result
        else if root value is "tan"
            // Rule: d/dx(tan(f)) = sec(f) ^ 2 * d/dx(f)
            result = new Tree("*")
            result left = autoGrad(lc, var, variables)
            result right = new Tree("^", new Tree("sec", lc, null), new Tree("2"))
            return result
        else if root value is "neg"
            // Rule: d/dx(-f) = -d/dx(f)
            return new Tree("neg", autoGrad(lc, var, variables), null
```

- Auto differentiation is actually not hard, it is just a traversal of a tree. We recursively visit every node of the tree and apply the specific rule according to it. Eventually we are able to construct a derivative tree.
- We use `if-else` statements to judge the case:
  - `variable`: d/dx(x) = 1
  - `constant`,`partial variable`: d/dx(const) = 0
  - `+`: d/dx(f + g) = d/dx(f) + d/dx(g)
  - `-`: d/dx(f - g) = d/dx(f) - d/dx(g)
  - `*`: d/dx(f * g) = f * d/dx(g) + g * d/dx(f)
  - `/`: d/dx(f / g) = (d/dx(f) * g - f * d/dx(g)) / g ^ 2
  - `^`: d/dx(f ^ g) = f ^ g * (d/dx(g) * ln(f) + g * d/dx(f) / f)
    - Here we first check if the base or the exponent can be regarded as a constant, to simplify the derivative function
  - `ln`: d/dx(ln(f)) = d/dx(f) / f
  - `log`: log(f, g) = ln(g) / ln(f), then we can call `Autograd()` recursively to apply the rule of `ln`
  - `pow`: we can turn `pow` to ^ and recursively call `Autograd()`
  - `exp`: we can turn `exp` to ^ and recursively call `Autograd()`
  - `sin`: d/dx(sin(f)) = cos(f) * d/dx(f)
  - `cos`: d/dx(cos(f)) = -sin(f) * d/dx(f)
  - `tan`: d/dx(tan(f)) = sec(f) ^ 2 * d/dx(f)
  - `neg`: d/dx(-f) = -d/dx(f)

SimplifyTree()

```
Function simplifyTree(root Node) returns Node
    If root is null
        Return null

    Left Node l = simplifyTree(root's left child)
    Right Node r = simplifyTree(root's right child)

    If root's value is '+'
        If l's value is "0"
            Return r
        Else If r's value is "0"
```

```
            Return l
        Else If l and r are both constants
            Return a new Node with the result of calculating l and r using '+'
        Else
            Return a new Node with the value '+', left child l, and right child r
    Else If root's value is a binary '-'
        If l's value is "0" and r's value is "0"
            Return a new Node with the value "0"
        Else If l's value is "0"
            Return a new Node with the value "neg" and left child r
        Else If r's value is "0"
            Return l
        Else If l and r are both constants
            Return a new Node with the result of calculating l and r using '-'
        Else
            Return a new Node with the value '-', left child l, and right child r
    Else If root's value is '*'
        If l's value is "0" or r's value is "0"
            Return a new Node with the value "0"
        Else If l's value is "1"
            Return r
        Else If r's value is "1"
            Return l
        Else If l and r are both constants
            Return a new Node with the result of calculating l and r using '*'
        Else If (l's value is "/" and (l's left child is isomorphic to r or l's right
child is isomorphic to r)) or
                (r's value is "/" and (r's left child is isomorphic to l or r's right
child is isomorphic to l))
            If l's left child is isomorphic to r
                Return l's right child
            Else If l's right child is isomorphic to r
                Return l's left child
            Else If r's left child is isomorphic to l
                Return r's right child
            Else If r's right child is isomorphic to l
                Return r's left child
        Else
            Return a new Node with the value '*', left child l, and right child r
    Else If root's value is '/'
        If l's value is "0"
            Return a new Node with the value "0"
        Else If r's value is "0"
            Output "Error: division by zero"
        Else If r's value is "1"
            Return l
        Else If l and r are both constants
            Return a new Node with the result of calculating l and r using '/'
        Else If l is isomorphic to r
            Return a new Node with the value "1"
        Else If (l's value is "*" and (l's left child is isomorphic to r or l's right
child is isomorphic to r)) or
```

```
                (r's value is "*" and (r's left child is isomorphic to l or r's right
child is isomorphic to l))
            If l's left child is isomorphic to r
                Return l's right child
            Else If l's right child is isomorphic to r
                Return l's left child
            Else If r's left child is isomorphic to l
                Return a new Node with the value "/", left child new Node with the
value "1", and right child r's right child
            Else If r's right child is isomorphic to l
                Return a new Node with the value "/", left child new Node with the
value "1", and right child r's right child
        Else
            Return a new Node with the value "/", left child l, and right child r
    Else If root's value is '^'
        If r's value is "0"
            Return a new Node with the value "1"
        Else If l's value is "0"
            Return a new Node with the value "0"
        Else If r's value is "1"
            Return l
        Else If l and r are both constants
            Return a new Node with the result of calculating l and r using '^'
        Else
            Return a new Node with the value '^', left child l, and right child r
    Else If root's value is "neg"
        If l is a constant
            Return a new Node with the result of calculating l and "-1" using '*'
        Else
            Return a new Node with the value "neg", left child l
    Else
        Return a new Node with the value of root, left child l, and right child r
End Function
```

- After auto differentiation, we can simplify the expression a little bit, to make it a bit easier to read for us (or there will be a lot of redundant operations which can be obviously omitted by we human-beings).
- Note that simplification can be done **when the expression tree is generated (which can simplify differentiation directly) and when the differentiation is done**.
- Here, we also use `if-else` statements to judge the cases (sorry I cannot think of any other ways)
  - `+`
    - `0 + x`, `x + 0`: x
    - `const + const`: directly return the calculated value
  - `-`
    - `0 - 0`: 0
    - `0 -`x: (neg) x
    - `x - 0`: x

    - `const - const`: directly return calculated value
- `*`
    - `0 * x`, `x * 0`: 0
    - `1 * x`, `x * 1`: x
    - `const * const`: directly return calculated value
    - `(x / y) * y`, `y * (x / y)`: x
- `/`
    - `0 / x`: 0
    - `x / 1`: x
    - `const / const`: directly return calculated value
    - `(x * y) / x`, `(y * x) / x`: y
    - `x / (x * y)`, `x / (y * x)`: 1/y
- `^`
    - `x ^ 0`: 1
    - `0 ^ x`: 0
    - `x ^ 1`: x
    - `const ^ const`: directly return calculated value
- `neg`
    - `(neg) const)`: directly return calculated value

printTree()

```
Procedure printTree(root Node, parent Node)
    If root is not null Then
        If root's value is "log" Then
            // Format: log(a, b)
            If root's left child's value is "e" Then
                // If the base is e, then we print ln
                Output "ln("
                Call printTree(root's right child, root)
                Output ")"
            Else
                Output "log("
                Call printTree(root's left child, root)
                Output ","
                Call printTree(root's right child, root)
                Output ")"
            End If
        Else If root's value is "ln" Then
            // Format: ln(a)
            Output "ln("
            Call printTree(root's left child, root)
            Output ")"
        Else If root's value is "pow" Then
            // Format: pow(a, b)
            Output "pow("
            Call printTree(root's left child, root)
            Output ","
            Call printTree(root's right child, root)
            Output ")"
        Else If root's value is "sin" Or root's value is "cos" Or
               root's value is "tan" Or root's value is "sec" Then
            // Format: sin(a), cos(a), tan(a), sec(a)
            Output root's value & "("
            Call printTree(root's left child, root)
            Output ")"
        Else If root's value is "exp" Then
            // Format: e^(a)
            Output "e^("
            Call printTree(root's left child, root)
            Output ")"
        Else If root's value is "neg" Then
            // Format: -(a)
            Output "-("
            Call printTree(root's left child, root)
            Output ")"
        Else
            // Format: (a op b)
            If checkParenthesis(parent, root) Then
                Output "("
            End If
            Call printTree(root's left child, root)
            Output root's value
            Call printTree(root's right child, root)
            If checkParenthesis(parent, root) Then
                Output ")"
            End If
        End If
    End If
End Procedure
```

- Eventually, here comes the last step - print the expression into an infix expression string
- We separate them into two case:
  - If we meet an unary operator, we simply print it out like a variable, and recursively call the function if there is still node inside the operator
  - If we meet a binary operator, we have to carefully inspect if a parenthesis is needed, and we have another function `checkParenthesis()` to do the thing

checkParenthesis()

```
Function checkParenthesis(parent Node, child Node) returns Integer
    If parent is null Then
        Return 0
    Else If child's left child is null Or child's right child is null Then
        Return 0
    Else If isBinaryOperator(parent's value) And isBinaryOperator(child's value) Then
        If (Precedence(parent's value) > Precedence(child's value)
            Or (parent's value is "^" And child's value is "^")
            Or (parent's value is "/" And child's value is "/")
            Or (parent's value is "/" And child's value is "*") Then
            Return 1
        Else
            Return 0
        End If
    Else
        Return 0
    End If
End Function
```

- If we encounter a variable, we have no need to add the parenthesis
- If we encounter a binary operator, here are the cases we have divided into:
  - If the parent node's precedence is higher, definitely a parenthesis has to be added, since the operations in the child node has to be calculated first
  - If they are of same precedence, there are very special cases
    - `a ^ (b ^ c)`, we cannot remove the parenthesis, or it will be `(a ^ b) ^ c`, which is wrong
    - `a / (b / c)`, the same
    - `a / (b * c)`, the same
    - Other parenthesis, like `a + (b - c)`, `a * (b * c)` won't bring confusion

Chapter 3 Testing Results

Check of differentiation rules

```
input:
a+b
output:
a : 1
b : 1
```

```
input:
a-b
output:
a : 1
b : -(1)
```

```
input:
a*b
output:
a : b
b : a
```

```
input:
a/b
output:
a : b/b^2
b : -(a)/b^2
```

```
input:
a^b
output:
a : b*a^(b-1)
b : ln(a)*a^b
```

```
input:
log(a,b)
output:
a : -(ln(b)*1/a)/ln(a)^2
b : (1/b*ln(a))/ln(a)^2
```

```
input:
ln(a)
output:
a : 1/a
```

```
input:
pow(a,b)
output:
a : b*a^(b-1)
b : ln(a)*a^b
```

```
input:
exp(a)
output:
e^a
```

```
input:
sin(a)
output:
cos(a)
```

```
input:
cos(a)
output:
-(sin(a))
```

```
input:
tan(a)
output:
sec(a)^2
```

Check of multiple variables and constants

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

- We can see that the operators are regarded of in natural precedence.
- Also, the printing order follows the lexicographic order of variables.

```
input:
xx*xy+cosval^2
output:
cosval : 2*cosval
xx : xy
xy : xx
```

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

- Variables can be distinguished from **reserved words** and deriviated right. Though maybe not in the simplist form but I have tried my best.

```
input:
(2+5*3-20+a)^(2^2)*log(2,3)
output:
a : 4*(-3+a)^3*log(2,3)
```

- Constants can be analyzed successfully and calculated when giving the final answer

Check of multiple parenthesis

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

- After careful simplification maunually, we can see that the answer is right.
- We can conclude that the program has the ability to process with multiple parenthesis (even if redundant) and print as less parenthesis as it can while guaranteeing validity.

Check of unary operators (BONUS)

```
input:
x*ln(y)
output:
x : ln(y)
y : x*1/y
```

- We can see that basic unary operators can be analyzed in a successful way
- Then let's see some more complicated cases

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

```
input:
1/(1+exp(-1*x))
output:
x : -(e^(-1*x)*-1)/(1+e^(-1*x))^2
```

```
input:
log(a,b)/log(c,a)
output:
a : (-(ln(b)*1/a)/ln(a)^2*log(c,a)-log(a,b)*(1/a*ln(c))/ln(c)^2)/log(c,a)^2
b : ((1/b*ln(a))/ln(a)^2*log(c,a))/log(c,a)^2
c : -(log(a,b)*-(ln(a)*1/c)/ln(c)^2)/log(c,a)^2
```

- I have to admit that this time the program is of terrible performance, because I do not do specific optimization for logorithms but the answer is correct at last after carefully simplification by ourselves.

Check of nested functions

```
input:
x^x^x
output:
x : (x^x)^x*(x*(1+ln(x))+ln(x^x))
```

```
input:
sin(log(a,b))
output:
a : -(ln(b)*1/a)/ln(a)^2*cos(log(a,b))
b : (1/b*ln(a))/ln(a)^2*cos(log(a,b))
```

- We can see that the parenthesis of nested functions can be analyzed successfully and generate an right answer.

Check of simplification (BONUS)

- Though I made terrible simplifications in some cases, but after all I do avoid a lot of redundant items.
- Since the bonus require at least two rules for simplification, here is what I have:
  - `0` will try its best to **not occur** in my final answer
  - most of redundant parenthesis are removed
  - **If there is only constants in a parenthesis**, then it will be calculated to its result constant
  - Apply the rules I have in the `simplifyTree()` function and **adjacent** variables following these rules will be simplified perfectly.
- We can see some test cases

```
input:
xx^2/xy*xy+a^a
output(before simplification):
a : (0*xy-xx^2*0)/xy^2*xy+xx^2/xy*0+a^a*(a*1/a+ln(a)*1)
xx : (1*2*xx^(2-1)*xy-xx^2*0)/xy^2*xy+xx^2/xy*0+0
xy : (0*xy-xx^2*1)/xy^2*xy+xx^2/xy*1+0
output(after simplification):
a : a^a*(1+ln(a))
xx : 2*xx
xy : 0
```

- We can easily see what I have done in comparison.

More cases

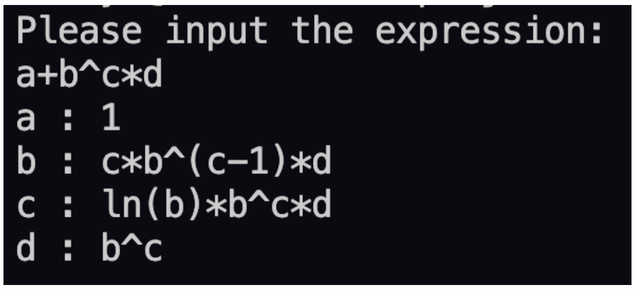
```
input:
-x+-y*-y
output:
x : -1
y : -1*-(y)+-(y)*-1
```

```
input:
pow(powln,lnlog)
output:
lnlog : ln(powln)*powln^lnlog
powln : lnlog*powln^(lnlog-1)
```

```
input:
(-5+log(x*cos(x),y))*exp(x+x+y*y)
output:
x : -(ln(y)*(cos(x)+x*-(sin(x)))/(x*cos(x)))/ln(x*cos(x))^2*e^(x+x+y*y)+
(-5+log(x*cos(x),y))*e^(x+x+y*y)*2
y : (1/y*ln(x*cos(x)))/ln(x*cos(x))^2*e^(x+x+y*y)+(-5+log(x*cos(x),y))*e^(x+x+y*y)*
(y+y)
```

Sample Testing Interface

- You are expected to see the following when testing:



```
Please input the expression:
a+b^c*d
a : 1
b : c*b^(c-1)*d
c : ln(b)*b^c*d
d : b^c
```

- You just have to type in the infix expression (blanks not allowed) after the prompt and press **ENTER**
- Output will be printed in the screen as follows

Chapter 4 Analysis and Comments

Space complexity

- We omit some data structures of only constant space complexity, and focus on those worth analyzing.
- `vector<string> tokens` `vector<string> variables`: all of $O(N)$
  - Note that `vector` uses contiguous space to store elements, it can be regarded as a sequential array, so operations like adding new elements and finding the element in the specific position is effecient, usually of constant time ($O(1)$).
- `stack<Tree *> treeStack` `stack<string> opStack` `vector<string>postfix`: all of $O(N)$
  - For a stack, operations like `push` and `pop` all require constant time ($O(1)$).
- `class Tree`: recursively defined, each node storing one token, with space complexity $O(N)$
  - Note that all we have in our program is traversal of a tree and build a new tree from it.

Time complexity

- `Tokenize()`: Iteration through the string, costing $O(N)$, and finding the length of each token cost constant of time (which is also an iteration, but is expected to stop at constant time).
- `parseExpression()`:
  - converting the infix expression to the postfix expression cost $O(N)$, which requires iterating through each token and may encounter one of the following cases - push into the container, push into the stack, pop constant number of elements from the stack and push itself in - they all require constant time.
  - Then we are going to iterate through the postfix expression and build the expression tree according to it. For each token, we will build a node according to its property - either a single node or decide its children, and push or pop it from the stack, which will require constant time. So the final time complexity is $O(N)$.
- `Autograd()` `simplifyTree()` `printTree()`: The core idea of these functions are the same, iterate through the whole tree and do constant operations.

  - For auto differentiation, we recursively call the function for further differentiation of the child node and build the tree on the current node.
  - For simplification, the difference is only that the job we do is to do simplification work.
  - Printing is the same, we will judge if the node is an operator or an operand, and sometimes print a parenthesis to guarantee precedence.
  - So the final complexity is $O(N)$.
- However, for `Autograd()` `simplifyTree()`, the analysis above are the ideal circumstances. In my implementation, the time complexity actually depends on the expression we process.
  - For example, in `Autograd()`, when we encounter operators like `log` and `pow`, we will not only do differentiation for its chilren nodes (which is actually enough), but also will generate a new sub tree `^` or `lns` and do differentiation again, this will cause the time complexity to be $O(N^2)$, instead of $O(N)$.
  - Also, in `simplifyTree()`, for operations like division and multiplication, there might be the chance that we will check if its sub trees are isomorphic, and it will bring another time complexity of $O(N)$, which cause the final time complexity to be $O(N^2)$.
  - So, it is actually of difficulty to analyze their time complexities.

Comments

The program definitely have a few drawbacks and we can do a lot of work for further optimization.

1. The readability of the program. There are a lot of `if-else` statements for judgment during the auto differentiation and the simplification process. There might be more elegant ways to write the program in a more readible way.
2. Simplification. There is definitely more space for further simplification. However, the job on the tree may be very tough, more efficient strategies like **Directed acyclic graph** may be a better choice.
3. Unary operators. My solution to deal with unary operators is actually of low-efficiency. I hang up one child node and define the specific rules for it. Also, when building the tree, I separate the conditions of unary operators from the normal ones, and the process run through tokenizing and tree-constructing. A more universal way can be dug to deal with it.
4. `Asserts`. My program has terrible ability of error detection, so if you type in an expression in a wrong way without notice, you may trigger bugs in the following process and have trouble finding them out. So the program can be better written for maintainability.

Appendix Source code

`main.cpp`

```
#include <iostream>
#include <string>
#include <vector>
```

```cpp
#include <stack>
#include <algorithm>

#include "autograd.hpp"
#include "tree.hpp"
#include "check.hpp"

using namespace std;

/********************
 * Tokenize
 * ********
 * This function will tokenize the expression
 * and return a vector of tokens
 *
 * Note: This function will not check the validity of the expression
 *       It will only check the validity of each token
 *       Unary operators will be tokenized with its operands as a whole,
 *       for example, "-(x+y)" will be tokenized as "-(x+y)", not as "-" "(" "x" "+" 
"y" ")"
 *       This is for the convenience of parsing
*/
vector<string> Tokenize(const string& expression) {
    vector<string> tokens;
    int i = 0;  //  iterator

    //  iterate through the expression
    while (i < expression.size()) {
        //  call isValid Function and it will return the length of the next token
        int length = isValid(expression, i);

        //  if the length is not 0, then we get a valid token
        if (length) {
            //  get the token
            string token = expression.substr(i, length);

            //  Note this is a very special case
            //  If the minus operator becomes a unary operator
            //  We tag it as a '#' operator for further convenience
            if (token[0] == '-' && length > 1) {
                token[0] = '#';
            }
            //  push the token into the vector
            tokens.push_back(token);
            //  update the iterator
            i += length;
        } else {
            //  if the length is 0, then we get an invalid token
            cerr << "Invalid token: " << expression.substr(i) << endl;
            i++;
        }
    }
    //  don't forget to return the vector
    return tokens;
}

/************************
 * parseExpression
 * *************
 * This function will do the following job:
 * 1. Convert the infix expression to postfix expression
 * 2. Construct the expression tree
 *
 * Note:
 * While constructing the expression tree, we will process
 * the unary operators and functions by recursively calls
 *
 * Normal operators and operands will be alright to apply the
 * method taught in the class, by using operator stack and operand stack
 *
 * However, unary operators and functions are a little bit different
 * First, we regard it as a common operand and find the right position for it
 * in the expression tree, then we construct it in detail by calling the function
 * recursively.
 *
*/
Tree *parseExpression(const vector<string>& tokens) {
    stack<Tree *> treeStack;    //  expression tree constructor
    stack<string> opStack;      //  operator stack
    vector<string> postfix;     //  postfix expression

    //  iterate through the tokens
    for (auto i : tokens) {
        if (!isBinaryOperator(i)) {
            //  if the token is not a binary operator, then
            //  it is either a variable or an unary operand as a whole
            //  then we just push it into stack
            postfix.push_back(i);
        } else {
            if (i == "(") {
                //  if the token is a left parenthesis, it has the highest precedence
                //  we just push it into operator stack
                opStack.push(i);
            } else if (i == ")") {
                //  if the token is a right parenthesis, we pop all the operators
                //  until we meet the left parenthesis
                while (!opStack.empty() && opStack.top() != "(") {
                    postfix.push_back(opStack.top());
                    opStack.pop();
                }
                //  don't forget to pop the left parenthesis
                opStack.pop();
            } else {
                //  if we meet a binary operator, we pop the operators in the stack
                //  until we meet an operator with higher precedence
                while (!opStack.empty() && Precedence(i) <= Precedence(opStack.top()))
{
                    postfix.push_back(opStack.top());
                    opStack.pop();
                }
                //  don't forget to push the operator into the stack
                opStack.push(i);
            }
        }
    }

    //  finally, we pop all the operators remaining in the stack
    while (!opStack.empty()) {
        postfix.push_back(opStack.top());
        opStack.pop();
    }

    //  construct the expression tree from the postfix expression
    for (auto i : postfix) {
        if (!isBinaryOperator(i)) {
            //  if the token is not a binary operator, then
            //  it is either a variable or an unary operand as a whole
            if (isVariable(i) || isConstant(i)) {
                //  common variable or constant, construct a children node (operand)
                treeStack.push(new Tree(i));
            } else {
                //  look into the unary operator and analyze the details
                //  we construct a father node (operator) with only the left child
(operand)
                //  and the right child will be nullptr
                if (i[0] == '#') {
                    //  if the token is a unary minus operator
                    //  we use "neg" to differentiate it from the binary minus
operator
                    vector<string> temp = Tokenize(i.substr(1));
                    //  recursively call the function to construct the expression tree
                    //  inside the unary operator
                    Tree *tempTree = parseExpression(temp);
                    Tree *op = new Tree("neg", tempTree, nullptr);
                    treeStack.push(op);
                } else if (i.substr(0, 2) == "ln") {
                    vector<string> temp = Tokenize(i.substr(2));
                    Tree *tempTree = parseExpression(temp);
                    Tree *op = new Tree("ln", tempTree, nullptr);
                    treeStack.push(op);
                } else if (i.substr(0, 3) == "exp" || i.substr(0, 3) == "sin" ||
                           i.substr(0, 3) == "cos" || i.substr(0, 3) == "tan") {
                    vector<string> temp = Tokenize(i.substr(3));
                    Tree *tempTree = parseExpression(temp);
                    Tree *op = new Tree(i.substr(0, 3), tempTree, nullptr);
                    treeStack.push(op);
                } else if (i.substr(0, 3) == "log" || i.substr(0, 3) == "pow") {
                    //  Note that log and pow are very special cases
                    //  they are two unary operators with two operands

                    //  the position of the current token
                    int pos = 4;    //  4 : the position after the first '('
                    int count = 1;  //  count the number of '(' and ')'

                    //  find the position of the comma
                    //  we have to skip the parenthesis pairs inside the operands
                    while (count != 1 || i[pos] != ',') {
                        if (i[pos] == '(') {
                            count++;
                        } else if (i[pos] == ')') {
                            count--;
                        }
                        pos++;
                    }
                    //  expression tree of two operands
                    //  same process : tokenize and expression tree construction
                    vector<string> temp1 = Tokenize(i.substr(4, pos - 4));
                    vector<string> temp2 = Tokenize(i.substr(pos + 1, i.size() - pos -
2));

                    Tree *tempTree1 = parseExpression(temp1);
                    Tree *tempTree2 = parseExpression(temp2);
                    Tree *op = new Tree(i.substr(0, 3), tempTree1, tempTree2);
                    treeStack.push(op);
                }
            }

        } else {
            //  if the token is a binary operator, then
            //  we pop two operands from the stack and construct a new node
            Tree* right = treeStack.top();
            treeStack.pop();
            Tree* left = treeStack.top();
            treeStack.pop();

            Tree* op = new Tree(i);
            op->left = left;
            op->right = right;

            treeStack.push(op);     //  push the new node into the stack
        }
    }
    //  the root of the final expression tree will be at the top of the stack
    return treeStack.top();
}

/************************
 * getVariable
 * **********
 * This function will get all the variables in the expression tree
 * and store them in a vector
*/
void getVariable(Tree* root, vector<string>& variables) {
    if (root == nullptr) {
        return;
    }

    //  if the node is a variable, then we push it into the vector
    if (isVariable(root->value)) {
        variables.push_back(root->value);
    }

    //  recursive calls
    getVariable(root->left, variables);
    getVariable(root->right, variables);
}

/************************
 * removeDuplicate
 * **************
 * This function will remove the duplicate variables in the vector
*/
void removeDuplicate(vector<string>& tokens) {
    //  sort the vector and the duplicate variables will be adjacent
    sort(tokens.begin(), tokens.end());
    //  remove the duplicate variables
    //  unique() will select all duplicate variables and move them to the end of the
vector
    //  then it will return the position of the first duplicate variable
    //  erase() will delete all the duplicate variables
    tokens.erase(unique(tokens.begin(), tokens.end()), tokens.end());
}

/*****************************
 * printTree
 * *********
 * This function will print the expression tree
*/
void printTree(Tree* root, Tree* parent) {
    //  If the node is non-empty, then we print it
    //  If not, we do nothing
    if (root) {
        if (root->value == "log") {
            //  Format : log(a, b)
            if (root->left->value == "e") {
                //  if the base is e, then we print ln
                cout << "ln(";
                printTree(root->right, root);
                cout << ")";
            } else {
```

```
            cout << "log(";
            printTree(root->left, root);
            cout << ",";
            printTree(root->right, root);
            cout << ")";
        }
    } else if (root->value == "ln") {
        //  Format : ln(a)
        cout << "ln(";
        printTree(root->left, root);
        cout << ")";
    } else if (root->value == "pow") {
        //  Format : pow(a, b)
        cout << "pow(";
        printTree(root->left, root);
        cout << ",";
        printTree(root->right, root);
        cout << ")";
    } else if (root->value == "sin" || root->value == "cos" ||
               root->value == "tan" || root->value == "sec") {
        //  Format : sin(a), cos(a), tan(a), sec(a)
        cout << root->value << "(";
        printTree(root->left, root);
        cout << ")";
    } else if (root->value == "exp") {
        //  Format : e^(a)
        cout << "e^(";
        printTree(root->left, root);
        cout << ")";
    } else if (root->value == "neg") {
        //  Format : -(a)
        cout << "-(";
        printTree(root->left, root);
        cout << ")";
    } else {
        //  When dealing with parenthesis, we have to check
        //  the precedence of the parent node and the current node
        //  If the precedence of the parent node is higher
        //  But the operation of the current node comes first
        //  A parenthesis is needed to guarantee validity

        //  Format : (a op b)
        if (checkParenthesis(parent, root)) {
            cout << "(";
        }
        printTree(root->left, root);
        cout << root->value;
        printTree(root->right, root);
        if (checkParenthesis(parent, root)) {
            cout << ")";
        }
    }
    }
}

int main() {
    while (1) {
        //  read in the expression
        string expression;
        cout << "Please input the expression:" << endl;
        cin >> expression;

        //  tokenize the expression
        vector<string> tokens = Tokenize(expression);
        // cout << "Successful tokenization!" << endl;

        //  construct the expression tree and do simplification
        Tree *root = parseExpression(tokens);
        // cout << "Successful tree construction!" << endl;
        // cout << "original expression: ";
        // printTree(simplifyTree(root), nullptr);
        // cout << endl;

        //  find all variables in the expression
        vector<string> variables;
        getVariable(root, variables);
        removeDuplicate(variables);
        // cout << "Successful variable getting!" << endl;
        // cout << "Here are the variables: ";
        // for (auto i : variables) {
        //     cout << i << " ";
        // }
        // cout << endl;

        //  iterate through all variables and do auto differentiation
        //  i : variable iterator
        for (auto i : variables) {
            Tree *result = autoGrad(simplifyTree(root), i, variables);
            cout << i << " : ";
            // printTree(result, nullptr);
            // cout << endl;
            printTree(simplifyTree(result), nullptr);
            // printTree(result, nullptr);
            cout << endl;
        }

        cout << endl;
    }
    return 0;
}
```

check.cpp

```
#include "check.hpp"
```

```
/*****************************
 * isomorphicTree
 * **************
 * Check if two trees are isomorphic
 * return 1 if they are, 0 if not
 * The function is almost the same as what
 * we have done in PTA's HW
*/
int isomorphicTree(Tree* T1, Tree* T2) {
    if (T1 == nullptr || T2 == nullptr) {
        if ((T1 == nullptr && T2 != nullptr) || (T1 != nullptr && T2 == nullptr))
{return 0;}
            else {return 1;}
    } else {
        if (T1->value == T2->value) {
            int l = isomorphicTree(T1->left, T2->left) && isomorphicTree(T1->right,
T2->right);
            int r = isomorphicTree(T1->left, T2->right) && isomorphicTree(T1->right,
T2->left);
            //  We have to note that the order of the children of
            //  log and pow function cannot be swapped
            //  So the second check (r) cannot happen in these two cases
            if (T1->value == "log" || T1->value == "pow") {
                r = 0;
            }
            //  If the two trees are isomorphic, then either l or r should be 1
            if (l || r) {return 1;}
                else {return 0;}
        } else {
            return 0;
        }
    }
}

/******************************
 * isValid
 * **********
 * Check if the token is valid analyze and return
 * the length of the token
 * Here are the possible cases:
 * 1. A constant
 * 2. A variable
 * 3. A unary operator
 * 4. A binary operator
*/
int isValid(const string& expression, const int i) {
    int length = 0;

    //  token: the string started from the current position (i)
    string token = expression.substr(i);
```

```cpp
    if (token[0] >= '0' && token[0] <= '9') {
        //  If the token is a constant
        //  Iterate until we encounter a non-constant character
        for (auto i : token) {
            if (i >= '0' && i <= '9') {
                length++;
            } else {
                break;
            }
        }
    } else if (isUnaryOperator(expression, i)) {
        //  If the token is a unary operator
        //  Function unaryExpressionParser will return the length of the expression
        length = unaryExpressionParser(token);
    } else if (isBinaryOperator(token)) {
        //  If the token is a binary operator
        //  The length of the token is definitely 1
        length = 1;
    } else if (isVariable(token)) {
        length = isVariable(token);
    }
    return length;
}

/***************************
 * isVariable
 * **********
 * Check if the token is a variable
 * return the length of it if it is, 0 if not
*/
int isVariable(const string& c) {
    if (c[0] >= 'a' && c[0] <= 'z') {
        if (c.substr(0, 3) == "sin" || c.substr(0, 3) == "cos" || c.substr(0, 3) ==
"tan" ||
            c.substr(0, 3) == "log" || c.substr(0, 3) == "pow" || c.substr(0, 3) ==
"exp" ||
            c.substr(0, 3) == "neg") {
            if (c[3] >= 'a' && c[3] <= 'z') {
                int i;
                for (i = 3; i < c.length(); i++) {
                    if (c[i] >= 'a' && c[i] <= 'z') {
                        continue;
                    } else {
                        break;
                    }
                }
                return i;
            } else {
                return 0;
            }
        } else if (c.substr(0, 2) == "ln") {
            if (c[2] >= 'a' && c[2] <= 'z') {
                int i;
                for (i = 2; i < c.length(); i++) {
                    if (c[i] >= 'a' && c[i] <= 'z') {
                        continue;
                    } else {
                        break;
                    }
                }
                return i;
            } else {
                return 0;
            }
        } else {
            int i;
            for (i = 0; i < c.length(); i++) {
                if (c[i] >= 'a' && c[i] <= 'z') {
                    continue;
                } else {
                    break;
                }
            }
            return i;
        }
    } else {
        return 0;
    }
}

/*************************
 * isUnaryOperator
 * ***************
 * Check if the token is a unary operator
 * return 1 if it is, 0 if not
 * Here are the possible cases:
 * ln, log, sin, cos, tan, pow, exp, negative(which is really a headache)
*/
int isUnaryOperator(const string& expression, const int i) {
    //  token: the string started from the current position (i)
    string c = expression.substr(i);
    if (c.substr(0, 2) == "ln" || c.substr(0,3) == "log" ||
        c.substr(0, 3) == "sin" || c.substr(0, 3) == "cos" || c.substr(0, 3) == "tan"
||
        c.substr(0, 3) == "pow" || c.substr(0, 3) == "exp") {
        return 1;
    } else if (c[0] == '-') {
        /******************************
         * Unary operator minus is really a headache
         * Here we separate it into three cases to judge:
         * 1. The first token of the expression
         * 2. The token after a binary operator but not a right parenthesis
         * 3. The token after a left parenthesis
        */
        if (i == 0) {
            return 1;
        } else if (isBinaryOperator(expression.substr(i - 1, 1)) && expression[i - 1]
!= ')') {
            return 1;
        } else if (expression[i - 1] == '(') {
            return 1;
        } else {
            return 0;
        }
    } else {
        return 0;
    }
}

/***********************
 * isBinaryOperator
 * **************
 * Check if the token is a binary operator
 * return 1 if it is, 0 if not
 * Here are the possible cases:
 * +, -, *, /, (, ), ^
*/
int isBinaryOperator(const string& c) {
    switch (c[0]) {
        case '+': case '-': case '*': case '/':
        case '(': case ')': case '^':
            return 1;
        default:
            return 0;
    }
}

/***********************
 * Precedence
 * **************
 * Return the precedence of the operator
 * Here are the possible cases:
 * +, -: 1
 * *, /: 2
 * ^: 3
 * (, ): 0
*/
int Precedence(const string& op) {
    switch (op[0])
    {
    case '+': case '-':
        return 1;
    case '*': case '/':
        return 2;
    case '^':
        return 3;
```

```cpp
    default:
        return 0;
    }
}

/*******************
 * isConstant
 * **********
 * Check if the token is a constant
 * return 1 if it is, 0 if not
 * Here are the possible cases:
*/
int isConstant(const string& c) {
    //  the token is a constant if
    //  1. it is a positive number starting with a digit character
    //  2. it is a number with a preceding minus operator
    return (c[0] >= '0' && c[0] <= '9') || (c[0] == '-' && c[1] >= '0' && c[1] <=
'9');
}

/**********************
 * isPartialVariable
 * *************
 * Check if the token is a partial variable
 * return 1 if it is, 0 if not
 * Here are the possible cases to check:
 * 1. Whether the token is the same as the main variable
 * 2. Whether the token is in the vector of variables
*/
int isPartialVariable(const string& partial, const string& main, const vector<string>&
variables) {
    if (partial == main)
        return 0;

    //  Iterate through the vector of variables
    for (auto i : variables) {
        if (i == partial)
            return 1;
    }

    return 0;
}

/*********************
 * checkConstant
 * ************
 * The function is written for the case of pow function
 * Check if the base(exponent) of the pow function is a constant (relatively)
*/
bool checkConstant(Tree *root, const string& var, const vector<string>& variables) {
    if (root == nullptr) {
        return true;
    }

    //  If it can be seen as a constant when operating partial differentiation
    //  Possible cases:
    //  1. A constant
    //  2. A partial variable
    //  3. A binary operator (look into it recursively)
    if (isConstant(root->value) || isPartialVariable(root->value, var, variables) ||
isBinaryOperator(root->value)) {
        return checkConstant(root->left, var, variables) && checkConstant(root->right,
var, variables);
    } else {
        return false;
    }

}

/************************
 * unaryExpressionParser
 * ***************
 * When we call this function, it means we have encountered a unary operator,
 * What remains to be done is to find the end of it.
 * Here, to simplify the problem, we see the unary operator and its operand as a whole
 * And we will further analyze the operand when constructing the expression tree
 *
 * Things to be done:
 * Find the total length of the unary operator and its operand
 * then return the length
*/
int unaryExpressionParser(const string& c) {
    string token;
    //  the length of the unary operator
    //  log, pow, exp, sin, cos, tan: 3
    //  ln: 2
    //  -: 1
    int adder;
    if (c.substr(0, 2) == "ln") {
        token = c.substr(2);
        adder = 2;
    } else if (c.substr(0, 3) == "log" || c.substr(0, 3) == "pow" || c.substr(0, 3) ==
"exp" ||
               c.substr(0, 3) == "sin" || c.substr(0, 3) == "cos" || c.substr(0, 3) ==
"tan") {
        token = c.substr(3);
        adder = 3;
    } else if (c[0] == '-') {
        token = c.substr(1);
        adder = 1;
    }

    //  We check if there is a parenthesis after the unary operator
    //  (which will make the problem more complicated)
    //  We have to find the right parenthesis that matches the left parenthesis
    //  which represents the end of the operand
    if (token[0] == '(') {
        int i = 1;
        int count = 1;
        while (count != 0) {
            if (token[i] == '(') {
                count++;
            } else if (token[i] == ')') {
                count--;
            }
            i++;
        }
        return i + adder;
    } else {
        //  If there is no parenthesis,
        //  which means there is only a single operand after it
        //  we just need to find the end of the operand
        return isValid(token, 0) + adder;
    }
}

/************************
 * checkParenthesis
 * ***************
 * The function works when we want to print the expression
 * We know a expression tree natually states the precedence of the operators
 * But when we turn it into an inorder expression, we have to add parenthesis if
needed
 *
 * Things to be done:
 * Check the precedence of the operator
 * return 1 if the child node needs parenthesis, 0 if not
*/
int checkParenthesis(const Tree* parent, const Tree* child) {
    if (parent == nullptr) {
        return 0;
    } else if (child->left == nullptr || child->right == nullptr) {
        return 0;
    } else if (isBinaryOperator(parent->value) && isBinaryOperator(child->value)) {
        //   If the parent node and the child node are both binary operators
        //   We have to check the precedence of the two operators
        //  If the precedence of the parent node is higher than the child node,
        //  but we have to calculate the operations in the children nodes first,
        //  a parenthesis will be added to guarantee the correctness of the expression
        if ((Precedence(parent->value) > Precedence(child->value))
         || (parent->value == "^" && child->value == "^")
         || (parent->value == "/" && child->value == "/")
         || (parent->value == "/" && child->value == "*")) {
            //  we have to be very careful if the parent operator and the child
operator
```

```
            //  are of the same precedence, which means we have to check the
associativity
            //  of the operators.
            return 1;
        } else {
            return 0;
        }
    } else {
        return 0;
    }
}

/***************************
 * calculateConstRes
 * ***************
 * The function is written when we simplifying the expression tree
 * When we encounter a constant expression, we can calculate the result
 * However, we have to be careful about the type of the result
 * If it is an integer, we don't need to return the double type,
 * which we bring further convenience to the program
*/
string calculateConstRes(const string a, const string b, const string op) {
    switch (op[0]) {
        case '+':
            if ((int)(stoi(a) + stoi(b)) == (double)(stod(a) + stod(b)))
            {return to_string((int)(stoi(a) + stoi(b)));}
                else {return to_string(stod(a) + stod(b));}
        case '-':
            if ((int)(stoi(a) - stoi(b)) == (double)(stod(a) - stod(b)))
            {return to_string((int)(stoi(a) - stoi(b)));}
                else {return to_string(stod(a) - stod(b));}
        case '*':
            if ((int)(stoi(a) * stoi(b)) == (double)(stod(a) * stod(b)))
            {return to_string((int)(stoi(a) * stoi(b)));}
                else {return to_string(stod(a) * stod(b));}
        case '/':
            if ((int)(stoi(a) / stoi(b)) == (double)(stod(a) / stod(b)))
            {return to_string((int)(stoi(a) / stoi(b)));}
                else {return to_string(stod(a) / stod(b));}
        case '^':
            if ((int)(pow(stoi(a), stoi(b))) == (double)(pow(stod(a), stod(b))))
            {return to_string((int)(pow(stoi(a), stoi(b))));}
                else {return to_string(pow(stod(a), stod(b)));}
        default:
            return "";
    }
}
```

autograd.cpp

```
#include "autograd.hpp"
```

```cpp
/********************
 * simplifyTree
 * ************
 * This function will try its best to remove redundant nodes in the tree.
 * Here are the rules we apply:
 * 1. replace (x + 0), (0 + x), (x - 0), (x * 1), (1 * x), (x / 1), (x ^ 1) by x
 * 2. replace (x - x), (x * 0), (0 * x), (0 / x), (0 ^ x) by 0
 * 3. replace (x / x), (x ^ 0) by 1
 *
 * We have to denote that simplification is actually a redundant thing,
 * we have to list special cases for possible simplification.
 * So there is a lot of if-else statements to maintain the logic as possible as we can.
*/
Tree* simplifyTree(const Tree* root) {
    if (root == nullptr)
        return nullptr;

    //  recursively simplify the left and right subtree
    Tree *l = simplifyTree(root->left);
    Tree *r = simplifyTree(root->right);

    if (root->value[0] == '+') {
        if (l->value == "0") {
            //  (0 + x), we just have to return x
            return r;
        } else if (r->value == "0") {
            //  (x + 0), we just have to return x
            return l;
        } else if (isConstant(l->value) && isConstant(r->value)) {
            //  (const + const), we can calculate the result
            return new Tree(calculateConstRes(l->value, r->value, "+"));
        } else {
            //  (x + y), we just have to return the original tree
            return new Tree("+", l, r);
        }
    } else if (root->value[0] == '-' && root->value.length() == 1) {
        //  Here we have to emphasize again on "-", which is really a headache
        //  In the expression tree, we allow constant like -1 to occur on the single node
        //  So we have to carefully investigate whether we encounter a binary operator or a unary operator
        if (l->value == "0" && r->value == "0") {
            //  (0 - 0), we just have to return 0
            return new Tree("0");
        } else if (l->value == "0") {
            //  (0 - x), we just have to return -x
            Tree *result = new Tree("neg", r, nullptr);
            return result;
        } else if (r->value == "0") {
            //  (x - 0), we just have to return x
            return l;
        } else if (isConstant(l->value) && isConstant(r->value)) {
            //  (const - const), we can calculate the result
            return new Tree(calculateConstRes(l->value, r->value, "-"));
        } else {
            //  (x - y), we just have to return the original tree
            return new Tree("-", l, r);
        }
    } else if (root->value[0] == '*') {
        if (l->value == "0" || r->value == "0") {
            //  (0 * x), (x * 0), we just have to return 0
            return new Tree("0");
        } else if (l->value == "1") {
            //  (1 * x), we just have to return x
            return r;
        } else if (r->value == "1") {
            //  (x * 1), we just have to return x
            return l;
        } else if (isConstant(l->value) && isConstant(r->value)) {
            //  (const * const), we can calculate the result
            return new Tree(calculateConstRes(l->value, r->value, "*"));
        } else if ((l->value == "/" && isomorphicTree(l->right, r))
                || (r->value == "/" && isomorphicTree(r->right, l))) {
            //  If the parent operator is '*', things are easier
            //  (x / y) * y, x * (y / x), we just have to return x(y)
            if (isomorphicTree(l->right, r))
                return l->left;
            else if (isomorphicTree(r->right, l))
                return r->left;
        } else {
            //  (x * y), we just have to return the original tree
            return new Tree("*", l, r);
        }
    } else if (root->value[0] == '/') {
        if (l->value == "0") {
            //  (0 / x), we just have to return 0
            return new Tree("0");
        } else if (r->value == "0") {
            //  (x / 0), we just have to return error
            cout << "Error: division by zero" << endl;
        } else if (r->value == "1") {
            //  (x / 1), we just have to return x
            return l;
        } else if (isConstant(l->value) && isConstant(r->value)) {
            //  (const / const), we can calculate the result
            return new Tree(calculateConstRes(l->value, r->value, "/"));
        } else if (isomorphicTree(l, r)) {
            //  (x / x), we just have to return 1
            return new Tree("1");
        } else if ((l->value == "*" && (isomorphicTree(l->left, r) ||
isomorphicTree(l->right, r)))
                || (r->value == "*" && (isomorphicTree(r->left, l) ||
isomorphicTree(r->right, l)))) {
```

```
            //  (x * y) / x, (x * y) / y, we just have to return y(x)
            //  However, if we meet x / (x * y), y / (x * y), we have to return 1 /
y(x)
            if (isomorphicTree(l->left, r))
                return l->right;
            else if (isomorphicTree(l->right, r))
                return l->left;
            else if (isomorphicTree(r->left, l))
                return new Tree("/", new Tree("1"), r->right);
            else if (isomorphicTree(r->right, l))
                return new Tree("/", new Tree("1"), r->left);
        } else {
            //  (x / y), we just have to return the original tree
            return new Tree("/", l, r);
        }
    } else if (root->value[0] == '^') {
        if (r->value == "0") {
            //  (x ^ 0), we just have to return 1
            return new Tree("1");
        } else if (l->value == "0") {
            //  (0 ^ x), we just have to return 0
            return new Tree("0");
        } else if (r->value == "1") {
            //  (x ^ 1), we just have to return x
            return l;
        } else if (isConstant(l->value) && isConstant(r->value)) {
            //  (const ^ const), we can calculate the result
            return new Tree(calculateConstRes(l->value, r->value, "^"));
        } else {
            //  (x ^ y), we just have to return the original tree
            return new Tree("^", l, r);
        }
    } else if (root->value == "neg") {
        if (isConstant(l->value)) {
            //  (-const), we can calculate the result
            return new Tree(calculateConstRes(l->value, "-1", "*"));
        } else {
            //  (-x), we just have to return the original tree
            return new Tree("neg", l, nullptr);
        }
    }
    else {
        //  (x), we just have to return the original tree
        return new Tree(root->value, l, r);
    }

    cout << "Error: simplifyTree, returning nullptr" << endl;
    return nullptr;
}

/*******************************
 * autoGrad
 * **************
 * This function will automatically calculate the derivative of the expression.
 *
 * Input:
 * root: the root of the expression tree
 * var: the variable we want to calculate the derivative
 * variables: the vector of variables in the expression
 *
 * Output:
 * the root of the derivative expression tree
*/
Tree* autoGrad(Tree* root, const string& var, const vector<string>& variables) {
    if (root == nullptr)
        return nullptr;

    //  Recursively calculate the derivative of the left subtree and the right subtree
    Tree *ld = autoGrad(root->left, var, variables);
    Tree *rd = autoGrad(root->right, var, variables);

    if (root->value == var) {
        //  If the current node is the variable itself
        //  Rule : d/dx(x) = 1
        return new Tree("1");
    } else if (isConstant(root->value) || isPartialVariable(root->value, var,
variables) || root->value == "e") {
        //  If the current node is a constant or a partial variable
        //  Rule : d/dx(const) = 0
        return new Tree("0");
    } else if (root->value[0] == '+') {
        //  Rule : d/dx(f + g) = d/dx(f) + d/dx(g)
        return new Tree("+", ld, rd);
    } else if (root->value[0] == '-') {
        //  Rule : d/dx(f - g) = d/dx(f) - d/dx(g)
        return new Tree("-", ld, rd);
    } else if (root->value[0] == '*') {
        //  Rule : d/dx(f * g) = f * d/dx(g) + g * d/dx(f)
        Tree *result = new Tree("+");
        result->left = new Tree("*", ld, root->right);
        result->right = new Tree("*", root->left, rd);
        return result;
    } else if (root->value[0] == '/') {
        //  Rule : d/dx(f / g) = (d/dx(f) * g - f * d/dx(g)) / g ^ 2
        Tree *result = new Tree("/");
        result->left = new Tree("-", new Tree("*", ld, root->right), new Tree("*",
root->left, rd));
        result->right = new Tree("^", root->right, new Tree("2"));
        return result;
    } else if (root->value[0] == '^') {
        //  Rule : d/dx(f ^ g) = f ^ g * (d/dx(g) * ln(f) + g * d/dx(f) / f)
        //  However, for possible simplification, we first check if
        //  the base and the exponent are constants or not
        //  which will help us to simplify the expression
        bool isBaseConstant = checkConstant(root->left, var, variables);
        bool isExponentConstant = checkConstant(root->right, var, variables);

        if (root->left->value[0] == 'e') {
            //  If the base is e, we can simplify the expression
            //  Rule : d/dx(e ^ g) = e ^ g * d/dx(g)
            Tree *result = new Tree("*");
            result->left = new Tree("^", root->left, root->right);
            result->right = autoGrad(root->right, var, variables);
            return result;
        } else
        if (isBaseConstant && isExponentConstant) {
            //  If both the base and the exponent are constants
            //  Rule : d/dx(const ^ const) = 0
            return new Tree("0");
        } else if (isBaseConstant) {
            //  If the base is a constant
            //  Rule : d/dx(const ^ g) = const ^ g * (d/dx(g) * ln(const))
            Tree *result = new Tree("*");
            result->left = new Tree("log", new Tree("e"), root->left);
            result->right = new Tree("^", root->left, root->right);
            return result;
        } else if (isExponentConstant) {
            //  If the exponent is a constant
            //  Rule : d/dx(f ^ const) =  (const - 1) * f ^ (const - 1) * d/dx(f)
            Tree *result = new Tree("*");
            result->left = new Tree("*", autoGrad(root->left, var, variables), root-
>right);
            result->right = new Tree("^", root->left, new Tree("-", root->right, new
Tree("1")));
            return result;
        } else {
            //  If both the base and the exponent are variables
            //  Rule : d/dx(f ^ g) = f ^ g * (d/dx(g) * ln(f) + g * d/dx(f) / f)
            Tree *result = new Tree("*");
            result->left = root;

            Tree *resultRight = new Tree("+");
            Tree *logTree = new Tree("log", new Tree("e"), root->left);
            resultRight->left = new Tree("*", root->right, autoGrad(logTree, var,
variables));
            resultRight->right = new Tree("*", logTree, autoGrad(root->right, var,
variables));

            result->right = resultRight;
            return result;
        }
    } else if (root->value.substr(0, 2) == "ln") {
        //  Rule : d/dx(ln(f)) = d/dx(f) / f
        Tree *result = new Tree("/");
        result->left = autoGrad(root->left, var, variables);
        result->right = root->left;
        return result;
    } else if (root->value.substr(0, 3) == "log") {
        //  We can apply the logorithm rule and convert log to ln
        //  Then we can simply recursively call the autoGrad function
        if (root->left->value == "e") {
            //  Turn log to ln and use the rule of ln
            Tree *result = autoGrad(new Tree("ln", root->right, nullptr), var,
variables);
            return result;
        } else {
            //  Rule : log(f, g) = ln(g) / ln(f)
            //  Turn log to ln and use the rule of ln
            Tree *result = autoGrad(new Tree("/", new Tree("ln", root->right,
nullptr), new Tree("ln", root->left, nullptr))
                                    , var, variables);
            return result;
        }
    } else if (root->value.substr(0, 3) == "pow") {
        //  Turn pow to ^ and use the rule of ^
        return autoGrad(new Tree("^", root->left, root->right), var, variables);
    } else if (root->value.substr(0, 3) == "exp") {
        //  Turn exp to e ^ and use the rule of ^
        return autoGrad(new Tree("^", new Tree("e"), root->left), var, variables);
    } else if (root->value.substr(0, 3) == "sin") {
        //  Rule : d/dx(sin(f)) = cos(f) * d/dx(f)
        Tree *result = new Tree("*");
        result->left = autoGrad(root->left, var, variables);
        result->right = new Tree("cos", root->left, nullptr);
        return result;
    } else if (root->value.substr(0, 3) == "cos") {
        // Rule : d/dx(cos(f)) = -sin(f) * d/dx(f)
        Tree *result = new Tree("*");
        result->left = autoGrad(root->left, var, variables);
        result->right = new Tree("neg", new Tree("sin", root->left, nullptr),
nullptr);
        return result;
    } else if (root->value.substr(0, 3) == "tan") {
        //  Rule : d/dx(tan(f)) = sec(f) ^ 2 * d/dx(f)
        Tree *result = new Tree("*");
        result->left = autoGrad(root->left, var, variables);
        result->right = new Tree("^", new Tree("sec", root->left, nullptr), new
Tree("2"));
        return result;
    } else if (root->value.substr(0, 3) == "neg") {
        //  Rule : d/dx(-f) = -d/dx(f)
        return new Tree("neg", autoGrad(root->left, var, variables), nullptr);
    }

    //  If we cannot find the rule to be applied, we just return the original tree
    cout << "Error: autoGrad, returning itself; No rule to be applied! Value: " <<
root->value << endl;
    return root;
```

```
}
```

tree.hpp

```
#ifndef _TREE_H_
#define _TREE_H_

#include <string>

class Tree {
    public:
        std::string value;
        Tree *left;
        Tree *right;

        //  Default constructor I
        //  Initialize the tree without any children
        Tree(std::string value) {
            this->value = value;
            this->left = nullptr;
            this->right = nullptr;
        }

        //  Default constructor II
        //  Initialize the tree with children
        Tree(std::string value, Tree *left, Tree *right) {
            this->value = value;
            this->left = left;
            this->right = right;
        }
};

#endif
```

Appendix Basic Explanation of C++

auto

- In C++, the `auto` keyword is used for type inference, allowing the compiler to automatically determine the data type of a variable based on its initialization. This feature is especially useful when working with iterators in loops, like a `for` loop.
- When using `auto` in a `for` loop, it can automatically deduce the data type of the loop variable based on the range or container we are iterating over.

```
for (auto element : container)
```

- This will iterate all the elements in the container and we can do anything we want to each element in the container.

class

- Similar to `struct` in C, we have `class` in C++.
- In our program, I only use one feature, that is the constructor.
- When we want to create an object (Here is each node of the tree), we can use `Tree *node = new Tree(...)`. Here, the constructor works.
- For example, if we write

```
 Tree(std::string value, Tree *left, Tree *right) {
     this->value = value;
     this->left = left;
     this->right = right;
}
```

- and we construct a node

```
Tree *myNode = new Tree("1", left, right);
```

- The constructor will assign `"1"` to its value, `left` and `right` to its left and right child.

vector

- `vector` is a container in **STL**, useful for us to construct data structures, since we no longer have to care about its concrete implementation in detail.
- We construct a vector object using `vector<type> ObjectName;`
- We add an element into it using `ObjectName.push_back(Element);`
- You can easily find other operations if you want.
- Other STLs used in this program like `stack` and `string` are the same, the program is readible even if you know nothing about it.

Declaration

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