1. Introduction

We are faced with processing a mathematical expression S in **infix expression** and calculating its gradient with respect to each variable. The results should be output in **lexicographical order**, with the gradient represented by literal constants and other variables.

Automatic gradient calculation plays an important role in **Machine Learning**. Famous packages like `PyTorch` has functions such as `.backward()` to compute gradients during **backpropagation**, optimizing the *weights and biases* of the model. This project provides a chance to implement `autograd` from scratch, which is undoubtedly interesting.

2. Algorithm Specification

2.1 Overall Insight

We can handle this problem in a **Step-by-step operation**.

- First, we should **tokenize** the expression, separating **variables**, **literal values** and **operators**.
- After the tokenization, I constructed a **tokenList** to store the different elements with their types explicitly **labeled**.
- And then, in `constructExpressionTree()`, I construct a **Expression Tree** with stacks to handle the problem of precedence of different calculations.
- What's next, we need to use `collectVariables()`to **collect all the variables** and use `derive()` to **calculate the derivatives** of every variable from the root of the expression tree.
- Last, We use `calculateGrad()` to traverse all the variable and **output the derivatives** in the lexicographical order.
- On the whole, `main.c` will serve as the main program, handling the problem of user input and answer output.

2.2 Specifications for *data structures*

2.2.1 struct Node

Description

- This struct is used to store one node in the expression tree, with its type and corresponding value.

- Note that **only one** in operator, number and variable need to be stored, according to the type.

Pseudo-code

```
typedef struct Node {
  int type;
  char operator;
  int number;
  char variable[N];
  struct Node *Left, *Right;
  struct Node *parent;
} Node;
```

2.2.2 struct TokenList

Description

- This struct is used to **store tokens**, after they are extracted from the input infix expression. And then, they will be sent to **construct the expression tree.**
- **types** are the corresponding types of the token with the same index.

Pseudo-code

```
typedef struct TokenList {
  char tokens[N][N];
  char types[N];
  int cnt;
}
```

2.3 Specifications for *Algorithms*

2.3.1 tokenize()

Description:

- This function is used to **separate the expression into fractions** with types labeled, so that we can do further process to variables, literal constants and operators.

Pseudo-code

```
void tokenize(char * expression, TokenList * tokenListPtr) {
  int i = 0;
  while (i < expressionLength) {
    if isspace(expression[i]):continue;
    else if isdigit(expression[i]) loop until the end of the number;
```

```
    else if (isOperator(expression[i])): add it into the tokenlist;
    else if (isalpha(expression[i]) || expression[i] == '-') loop until the end of variable;
  }
}
```

2.3.2 createExpressionTree()

Description

- `createExpressionTree()` is used to construct a binary tree storing all the operators and operands with data from the tokenList. We maintain **two stacks**, **nodeStack** is used to store operands and one is used to store operators. And then we maintain **opStacksss** according to the precedence of operators.

Pseudo-code

```
Node * createExpressionTree(TokenList * tokenListPtr) {
  initialize tokenListLen, nodeStack, opStack;
  for (i < tokenListLen) {
    switch(tokenListPtr->type):
      process expressions with precedence considered;
      pop and push to the opStack and nodeStack;
  }
  while (opTop >= 0) {
    pop, setchildren and push;
  }
  return nodeTop;
}
```

2.3.3 getNodeExpr

Description

- This function is used to generate the corresponding string expression of a certain node. If the node is a **operand node**, then we output the string form of the operand. If the node is an **operator node**, then we output the expression string, combining the left operand, operator and the right operand. Note that the operand here came from **recursive call of getNodeExpr()**

Pseudo-code

```
char * getNodeExpr(Node * node) {
  if (node->type == TOKEN_IS_VAR) {
    return strdup(node->variable);
  } else if (node->type == TOKEN_IS_NUM) {
    return transferToString(node->number);
```

```
  } else if (node->type == TOKEN_IS_OPERATOR) {
    get left, right, op;
    return (string)("(%s %c %s)", left, right, op);
  } else return '0';
}
```

2.3.4 collectVariables

Description

- This function is used to **collect all the existing variables** in the expression. If the variable we are looking for exists, then the flag is true. If it is not found, then we add the variable from the current node into our variable list. After this, we **recursively** call collectVariables to check the variables from the left subtree and right subtree.

Pseudo-code

```
void collectVariables(Node * node, char ** vars, int * count) {
  if (node->type == TOKEN_IS_VAR) {
    bool exists = false;
    for loop until we find the variable;
    if not found, add it into the list and (*count)++;
  }
  recursively collect variables from left and right;
}
```

2.3.5 derive

Description

- This function is used to **calculate the derivative** from the current node for variable var. For the simplest case for variables and numbers, we can directly get the gradient of 0 or 1. If the type is an operator, then we can calculate the derivative with the **expression and derivative** of left and right. Note that we shall apply a series of **simplification rules** for 0s and 1s. We can use `getNodeExpr()` to get the expression for a certain node and use `formatExpr()` to get the expression.

Pseudo-code

```
char * derive(Node * node, char * var) {
  if (node->type == TOKEN_IS_VAR) {
    if (strcmp(node->variable, var) == 0) {
      return "1";
    } else {
      return "0";
    }
```

```
  } else if (node->type == TOKEN_IS_NUM) {
    return "0";
  } else if (node->type == TOKEN_IS_OPERATOR) {
    get operator, leftDeriv, rightDeriv, leftExpr, rightExpr;
    initialize result;
    switch(operator) {
      calculate the derivative according to the mathematical law;
    }
  }
}
```

2.3.6 calculateGrad

Description

- Collect all the variables first and then sort the variables in lexicographical order. Then we traverse through all the variables from the root of the entire expression tree, and output the derivative of the variable.

Pseudo-code

```
void calculateGrad(Node * root) {
  initialize varCount, variables;
  collectVariables(root, variables, &varCount);
  qsort(variables, varCount, sizeof(char *), compareStrings);
  /*compareStrings() is used to implement the lexicographical sort*/
  for (int i = 0; i < varCount; i ++) {
    char * derivExpr = derive(root, variables[i]);
    printf(variables[i], derivExpr);
    free(derivExpr);
  }
}
```

2.4 Specifications for *main program*

Description

- The `main.c` here is the main program. It is used to accept input, and then print the output, and tackle with some errors. Thanks to the functions implemented in `functions.c`, the main program here can be very concise.

Pseudo-code

```
int main() {
  char * inputExpr;
  TokenList * tokenListPtr;
  Node * rootPtr;
  fgets(inputExpr);
```

```
  tokenize(inputExpr, tokenListPtr);
  rootPtr = createExpressionTree(tokenListPtr);
  if (!rootPtr) calculateGrad(rootPtr);
}
```

3. Testing Results

Here, we are testing various cases to seek potential issues in our program. These include, but are not limited to, extreme cases such as expressions without derivable terms and nested complex expressions.

Test Cases	Design Purpose	Result	Status
1+2	case without derivable term	Underivable Expression!	pass
xx+_xy^ab	expression with multi-letter variable name	_xy: (_xy ^ ab) * (0 * ln(_xy) + ab * 1 / _xy) ab: (_xy ^ ab) * (1 * ln(_xy) + ab * 0 / _xy) xx: (1 + (_xy ^ ab) * (0 * ln(_xy) + ab * 0 / _xy))	pass
(a+b)*c	expression with parentheses	a: ((a + b) ^ c) * (0 * ln((a + b)) + c * 1 / (a + b)) b: ((a + b) ^ c) * (0 * ln((a + b)) + c * 1 / (a + b)) c: ((a + b) ^ c) * (1 * ln((a + b)) + c * 0 / (a + b))	pass
a+b+c+d+e+f*g	relatively long expression	a: 1 b: 1 c: 1 d: 1 e: 1 f: (g * 1) g: (f * 1)	pass
a	expression with one variable	a: 1	pass
a+	invalid input expression	Invalid input!	pass

After **arithmetic simplification** on our result, we can discover that our results are all correct for the derivable cases. For the **underivable cases** and **invalid input cases**, we can also output suitable prompt.

4. Analysis and Comments

4.1 Time complexity analysis

Let's assume that we have n node in our expression tree. Before calculating the derivative, we recursively call `collectVariables()` to traverse through all the nodes in the tree, which corresponds to $O(n)$.

Next, we sort all the variables in lexicograhical order, which will cost us $O(nlogn)$ time in the worst case.

In `derive()`, we traverse through all the variables and start from the root of the expression tree. So in the worst case the time complexity would be $O(n^2)$.

The time complexity of other functions like `createExpressionTree()` or `tokenize()` are all approximately linearly related to the length of the expression, which are all smaller than $O(n^2)$.

In conclusion, the time complexity of our algorithm is $O(n^2)$.

4.2 Space complexity analysis

If the length of the expression is n, then the maximum depth of the stacks are n in the worst case.

Let's assume that there are n node in our expression tree. In `derive()` and `collectVariables()`, the maximum depth of recursion is n.

In the string malloc phase, in `formatExpr()`, the extra space required are also linarly relative to the node count of our expression tree.

In conclusion, the space complexity of our algorithm is $O(n)$.

4.3 Comments and further improvements

- Our algorithm accurately computes derivatives, but the results can be simplified to enhance readability. We could achieve this by applying additional simplification rules into the `derive()` function, such as getting rid of redundant coefficients of 1 or 0 from the output and combine like terms.
- In this program, a great amount of fixed memory is allocated for expressions and strings, potentially leading to inefficient memory usage. To optimize space utilization, we could consider using `malloc` for dynamic memory allocation.

- Our current implementation contains large functions, indicating the need for better modularization. In future projects, I could decompose these functions into more specialized ones to enhance clarity.

Appendix: Source Code (in C)

main.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <ctype.h>
/*with functions to determine whether a character is a letter or a number*/
#include <string.h>
/*a set of string processing functions*/
#include "header.h"
/*necessary header files included*/

int main()
{
    char * inputExpr = (char *)calloc(50, sizeof(char));
    /*initialize the input string*/
    TokenList * tokenListPtr = (TokenList * )calloc(1, sizeof(TokenList));
    /*create the tokenlist to store tokens that are extracted from the expression*/
    Node * rootPtr = (Node * )calloc(1, sizeof(Node));
    /*malloc memory for the input expression*/
    printf("Please input the expression: ");
    /*user input prompt*/
    fgets(inputExpr, EXPR_MAX_LEN, stdin);
    /*get the expression from the user*/
    tokenize(inputExpr, tokenListPtr);
    /*tokenize the input expression string*/
    rootPtr = createExpressionTree(tokenListPtr);
    if (rootPtr == NULL)
    /*which means that the expression tree is not successfully created*/
    {
        return 0;
    }
    else
    {
        calculateGrad(rootPtr);
        /*calculate the gradient of every variable inside*/
    }
    getchar();
    /*used to avoid the terminal from directly shutting down*/
}
```

header.h:

```c
#ifndef HEADER
#define HEADER
```

```c
/*include guard, ensuring the overall safety*/

#define EXPR_MAX_LEN 50
/*maximum length for the input expression*/
#define VAR_MAX_LEN 10
/*maximum length for the variable name*/
#define TOKEN_MAX_NUM 20
/*maximum number for tokens*/

#define TOKEN_IS_NUM 'N'
#define TOKEN_IS_VAR 'V'
#define TOKEN_IS_OPERATOR 'O'
/*define some representative values*/

typedef struct Node {
    int type;
    /*corresponding to the #define ahead*/
    char operator;
    /*operator if the type is N +, -, *, /, ^*/
    int number;
    /*literal num if the type is N*/
    char variable[VAR_MAX_LEN];
    /*name of the variable if the type is V*/
    struct Node * Left, * Right;
    /*Left and Right child tree*/
    struct Node * Parent;
    /*parent node*/
} Node;
/*the struct Node is for the construction of expression tree*/

typedef struct TokenList {
    char tokens[TOKEN_MAX_NUM][EXPR_MAX_LEN];
    /*store the tokens*/
    char types[TOKEN_MAX_NUM];
    /*store the types of the tokens, N, V, O*/
    int count;
    /*count of the tokens*/
} TokenList;
/*Implement a type of datastructure to store*/

void tokenize(char * expression, TokenList * tokenListPtr);
/*the function to parse the expression and storage the tokens into our tokenlist*/
Node * createNode(char type, char operation, int number, char * variable);
/*it is used to create a node, assigning features to it.*/
int getPrecedence(char op);
/*return the precedence of various operators*/
```

```c
bool isOperator(char c);
/*determine whether c is an operator*/
void setChildren(Node * parent, Node * left, Node * right);
/*set the parent of Node left and right, and set the children of the current node*/
Node * createExpressionTree(TokenList * tokenListPtr);
/*use the tokenlist to create an expression tree*/
void calculateGrad(Node * root);
/*sort the variables with lexicographical order and output their corresponding derivative*/
char* getNodeExpr(Node* node);
/*get the expression of the node*/
char* formatExpr(char* fmt, ...);
/*unified format expression function, no matter what is the length*/
void collectVariables(Node* node, char** vars, int* count);
/*collect all the variables in the expression*/
char* derive(Node* node, char* var);
/*calculate the derivative of variables*/
int compareStrings(char * a, char * b);
/*compare the lexicographical order of strings, used in qsort()*/
#endif
```

functions.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <ctype.h>
#include <stdarg.h>
#include <string.h>
#include "header.h"

/*necessary header files included*/

Node *createNode(char type, char operation, int number, char *variable) {
    Node *tempNode = (Node *)calloc(1, sizeof(Node));
    /*malloc memory for the node*/
    tempNode->type = type;
    tempNode->operator = operation;
    tempNode->number = number;
    /*assign basic information of the node*/

    strcpy(tempNode->variable, "");
    /*initialize the tempNode*/
    if (variable != NULL) {
        strcpy(tempNode->variable, variable);
        /*if the variable is not empty, then initialize the variable*/
    }
```

```
    tempNode->Left = NULL;
    tempNode->Right = NULL;
    tempNode->Parent = NULL;
    /*set to NULL first, later we would use */

    return tempNode;
}

bool isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^');
    /*implement the judgement of operators*/
}

void tokenize(char *expression, TokenList *tokenListPtr) {
    int expressionLength = strlen(expression);
    int i = 0, j = 0;
    /*there are to pointers here, i is for the traversal of the entire expression*/
    /*j is for the traversal of the storage of the token in tokenList*/
    tokenListPtr->count = 0;
    /*initialize the count of all tokens*/

    while (i < expressionLength) {
        if (isspace(expression[i])) {
        /*if it is space, then skip*/
            i++;
            continue;
        }

        j = 0;
        /*j will be initialized for every token*/

        if (isdigit(expression[i])) {
            while (i < expressionLength && isdigit(expression[i]) && j < TOKEN_MAX_NUM - 1)
                tokenListPtr->tokens[tokenListPtr->count][j++] = expression[i++];
                /*the tokens will be stored in tokenList*/
                /*note that the while loop here is for the storage of multi-bit numbers*/
            }
            tokenListPtr->types[tokenListPtr->count] = TOKEN_IS_NUM;
            /*if the type is number, then it will be stored*/
        }
        else if (isOperator(expression[i]) || expression[i] == '(' || expression[i] == ')')
            /*if the token is an operator or parentheses, it will also need to be stored*/
            tokenListPtr->tokens[tokenListPtr->count][j++] = expression[i++];
            tokenListPtr->types[tokenListPtr->count] = TOKEN_IS_OPERATOR;
            /*store the operator type*/
        }
```

```
        else if (isalpha(expression[i]) || expression[i] == '_') {
            /*store the variable type with C standard, which can start with letters of _*/
            while (i < expressionLength && (isalnum(expression[i]) || expression[i] == '_')
                /*the isalnum here is for the bits that can be numbers or letters*/
                tokenListPtr->tokens[tokenListPtr->count][j++] = expression[i++];
                /*the same as the storage in digit, we also need to consider more bits*/
            }
            tokenListPtr->types[tokenListPtr->count] = TOKEN_IS_VAR;
        }
        else {
            i++;
            /*the case of other invalid inputs, we can directly skip the characters*/
            continue;
        }

        tokenListPtr->tokens[tokenListPtr->count][j] = '\0';
        /*store the last token to be NULL string, indicating the termination of the entire s
        if (j > 0) {
        /*j can serve as a flag for actual storage, if 0, then it means that no tokens are s
            tokenListPtr->count++;
            /*the count will only be increased when there exists valid token*/
        }
    }
}
/*tokenize will help us seperate the expression into different parts*/

int getPrecedence(char operator)
/*help us to get the precedence of different operators*/
{
    switch (operator) {
        case '^':
            return 3;
    /*top priority for the power operator*/
        case '*':
        case '/':
            return 2;
    /*the same priority will use the feature of fall-through of switch*/
        case '+':
        case '-':
            return 1;
        case '(':
            return 0;
        /*because the case of parentheses will be individually treated, so
        here is the minimum priority*/
        default:
            return -1;
```

```
    }
}

/*createExpressionTree: Build an expression tree from the token list*/
/*we will implement with two stacks to store numbers/variables(operands) and operators*/
Node *createExpressionTree(TokenList *tokenListPtr) {
    int len = tokenListPtr->count;  /* total number of tokens */
    /* Stack for operand nodes*/
    Node *nodeStack[TOKEN_MAX_NUM];
    int nodeTop = -1;
    /* Stack for operator nodes*/
    Node *opStack[TOKEN_MAX_NUM];
    int opTop = -1;
    /* Process each token in the token list. */
    for (int i = 0; i < len; i++) {
        /*if the token is a number, push it in the stack*/
        if (tokenListPtr->types[i] == TOKEN_IS_NUM) {
            Node *newNode = createNode(TOKEN_IS_NUM, '\0', atoi
            (tokenListPtr->tokens[i]), NULL);
            nodeStack[++nodeTop] = newNode;
        }
        /*note that every time we need to create the node*/
        else if (tokenListPtr->types[i] == TOKEN_IS_VAR) {
            Node *newNode = createNode(TOKEN_IS_VAR, '\0', 0, tokenListPtr->tokens[i]);
            nodeStack[++nodeTop] = newNode;
        }
        /*the case where the token is an operator or a parenthesis*/
        else if (tokenListPtr->types[i] == TOKEN_IS_OPERATOR) {
            char currentOp = tokenListPtr->tokens[i][0];
            /*tackle the case where left parenthesis appears*/
            if (currentOp == '(') {
                Node *newNode = createNode(TOKEN_IS_OPERATOR, currentOp, 0, NULL);
                opStack[++opTop] = newNode;
            }
            /*pop */
            else if (currentOp == ')') {
                while (opTop >= 0 && opStack[opTop]->operator != '(') {
                    if (nodeTop < 1)
                    /*error check for insufficient operands*/
                    {
                        printf("Invalid input\n");
                        return NULL;
                    }
                    /* Pop an operator node*/
                    Node *opNode = opStack[opTop--];
                    /* Pop two operand nodes from nodeStack*/
```

```
                Node *right = nodeStack[nodeTop--];
                Node *left = nodeStack[nodeTop--];
                /*set the children nodes and their parents*/
                setChildren(opNode, left, right);
                /*push the corresponding sub-tree*/
                nodeStack[++nodeTop] = opNode;
            }
            /*pop the left parenthesis if there is any of them left*/
            if (opTop >= 0)
                opTop--;
        }
        else {
            /*tackle the case of meeting greater precedence*/
            while (opTop >= 0 && getPrecedence(opStack[opTop]->operator)
            >= getPrecedence(currentOp)) {
                if (nodeTop < 1)
                /*error check for insufficient operands*/
                {
                    printf("Invalid input\n");
                    return NULL;
                }
                Node *opNode = opStack[opTop--];
                /*pop two nodes as operands*/
                Node *right = nodeStack[nodeTop--];
                Node *left = nodeStack[nodeTop--];
                /*set the children*/
                setChildren(opNode, left, right);
                nodeStack[++nodeTop] = opNode;
                /*push the node back to the stack*/
            }
            Node *newOpNode = createNode(TOKEN_IS_OPERATOR, currentOp, 0, NULL);
            opStack[++opTop] = newOpNode;
        }
    }
}
/*processing left parenthesis that are left here*/
while (opTop >= 0) {
    if (nodeTop < 1)
    /*error check*/
    {
        printf("Invalid input\n");
        return NULL;
    }
    Node *opNode = opStack[opTop--];
    Node *right = nodeStack[nodeTop--];
    Node *left = nodeStack[nodeTop--];
```

```
            setChildren(opNode, left, right);
            /*set the corresponding children*/
            nodeStack[++nodeTop] = opNode;
        }
        if (nodeTop != 0)  /* If there is more than one node, then the input is invalid */
        {
            printf("Invalid input\n");
            return NULL;
        }
        /*return the final node, which is the root.*/
        return nodeStack[nodeTop];
    }

    void setChildren(Node *parent, Node *left, Node *right)
    /*the function to set the children of a node and the parent of the node, to
    make indexing easier.*/
    {
        parent->Left = left;
        parent->Right = right;
        /*setting the children of the parent node*/
        left->Parent = parent;
        right->Parent = parent;
        /*setting the parent node of the currentnode*/
    }

    char* getNodeExpr(Node* node)
    {
        /*used to visit the expression of the current node*/
        if (node->type == TOKEN_IS_VAR)
        {
            /*variable case*/
            return strdup(node->variable);
            /*strdup is used to return a string that is copied, but not a pointer
            to the same memory space*/
        }
        else if (node->type == TOKEN_IS_NUM)
        {
            /*number case*/
            char buf[EXPR_MAX_LEN];
            /*initialize a buffer zone*/
            sprintf(buf, "%d", node->number);
            /*we use sprintf to store the string into the buffer zone*/
            /*copy the number to the buffer zone*/
            /*note that the number here is stored in the string form*/
            return strdup(buf);
        }
```

```
    else if (node->type == TOKEN_IS_OPERATOR)
    {
        /*the case where the token is an operator*/
        char* left = getNodeExpr(node->Left);
        /*extract the number as the left operand*/
        char* right = getNodeExpr(node->Right);
        /*extract the number as the right operand*/
        char op = node->operator;
        /*extract the operator*/
        char buf[EXPR_MAX_LEN];
        /*create the buffer zone for storing*/
        sprintf(buf, "(%s %c %s)", left, op, right);
        /*push the expression into the buffer*/
        free(left);
        free(right);
        /*free the memory space that are malloced*/
        return strdup(buf);
    }
    return strdup("0");
    /*if no situation is satisfied, then return 0 directly*/
}

char* formatExpr(char* fmt, ...)
/*the ... is used to express that the parameters we accept can be different,
according to the situation we are in*/
/*we use va to tackle this problem because we might be faced with inputs with
different lengths and requirements*/
{
    va_list args;
    /*variable arguments, change depend on the situation*/
    va_start(args, fmt);
    /*fmt is the last fixed parameter*/
    int len = vsnprintf(NULL, 0, fmt, args);
    /*although we don't need the buffer, we can use this function to get the
    length of the string*/
    va_end(args);
    /*clean up the argument list*/
    char* buf = (char*)malloc(len + 1);
    /*malloc the buffer space for the expression*/
    va_start(args, fmt);
    vsnprintf(buf, len + 1, fmt, args);
    /*store the string into the buffer zone*/
    va_end(args);
    /*clean the entire variable argument list*/
    return buf;
}
```

```
void collectVariables(Node* node, char** vars, int* count)
{
/*used to determine whether the given variable exists in our expression*/
    if (!node)
    {
        return;
    }
    /*the case where the node is NULL*/
    if (node->type == TOKEN_IS_VAR)
    {
        bool exists = false;
        /*initially set to false*/
        for (int i = 0; i < *count; i++)
        /*traverse through all the variables*/
        {
            if (strcmp(vars[i], node->variable) == 0)
            /*indicates that the variable exists*/
            {
                exists = true;
                /*flag set to true*/
                break;
            }
        }
        if (!exists && *count < TOKEN_MAX_NUM)
        /*there doesn't exist the variable*/
        {
            vars[*count] = strdup(node->variable);
            /*copy the variable that has never been seen*/
            (*count)++;
            /*count increment*/
            /*note that count is a pointer because we want to modify the value of it*/
        }
    }
    collectVariables(node->Left, vars, count);
    /*tail recursion to implement the collection in left subtree*/
    collectVariables(node->Right, vars, count);
    /*collect in the right subtree*/
}

/*calculate the derivatives*/
char* derive(Node* node, char* var)
{
    if (node->type == TOKEN_IS_VAR)
    {
        if (strcmp(node->variable, var) == 0)
```

```
    {
        return strdup("1");
        /*if the variable is equal to the current node, then the variable is one*/
        /*note that the numbers are always stored in string*/
    }
    else
    {
        /*if the variable is not equal to the current node, then the
        derivate must be independent*/
        return strdup("0");
    }
}
else if (node->type == TOKEN_IS_NUM)
{
    /*the derivative of a constant is undoubtedly 0*/
    return strdup("0");
}
else if (node->type == TOKEN_IS_OPERATOR)
{
    /*tackle the problem of derivative with operators*/
    char op = node->operator;
    /*get the operator*/
    char* leftDeriv = derive(node->Left, var);
    /*get the derivative of the left operand*/
    char* rightDeriv = derive(node->Right, var);
    /*get the derivative of the right operand*/
    char* leftExpr = getNodeExpr(node->Left);
    /*get the expression of the left operand*/
    char* rightExpr = getNodeExpr(node->Right);
    /*get the expression of the right operand*/
    /*these four expression will be used in the deriving process*/
    char* result = NULL;
    /*default output*/
    switch(op) {
        /*switch the case according to the operator*/
        case '+':
            if (strcmp(leftDeriv, "0") == 0 && strcmp(rightDeriv, "0") == 0)
            {
                result = strdup("0");
                /*if the left and right derivative are all zero, then the
                output must be also zero.*/
            }
            else if (strcmp(leftDeriv, "0") == 0)
            {
                /*simplifying the answer*/
                result = strdup(rightDeriv);
```

```
    }
    else if (strcmp(rightDeriv, "0") == 0)
    {
        /*simplifying the answer in the same way*/
        result = strdup(leftDeriv);
    }
    else
    {
        /*in this case, there's no space for further simplification*/
        result = formatExpr("(%s + %s)", leftDeriv, rightDeriv);
    }
    break;
case '-':
/*the situation of minus*/
    if (strcmp(leftDeriv, "0") == 0 && strcmp(rightDeriv, "0") == 0)
    {
        result = strdup("0");
        /*if the left and right are both 0, then output 0 directly*/
    }
    else if (strcmp(leftDeriv, "0") == 0)
    {
        /*if the left is 0, then output the negation of the right derivative*/
        result = formatExpr("(-%s)", rightDeriv);
        /*single variable format expression is also feasible
        because of our function*/
    }
    else if (strcmp(rightDeriv, "0") == 0)
    {
        /*if the right derivative is 0, then directly output the
        left derivative*/
        result = strdup(leftDeriv);
    }
    else
    {
        /*no space for simplification*/
        result = formatExpr("(%s - %s)", leftDeriv, rightDeriv);
        break;
    }
case '*':
/*here we tackle the case of multiplication*/
    if (strcmp(leftExpr, "0") == 0 && strcmp(rightExpr, "0") == 0)
    /*if all zero, then directly output 0*/
    {
        result = strdup("0");
        /*directly output 0*/
    }
```

```
else if (strcmp(leftExpr, "0") == 0)
{
    if (strcmp(rightDeriv, "0") == 0)
    {
        /*if the right derivative is zero*/
        result = strdup("0");
        /*output 0*/
    }
    else
    {
        if (strcmp(leftExpr, "1") == 0)
        {
            /*if the left expression is 1*/
            result = strdup(rightDeriv);
            /*output the right derivative*/
        }
        else if (strcmp(rightDeriv, "1") == 0)
        {
            /*if the right derivative is 1*/
            result = strdup(leftExpr);
            /*output the left expression*/
        }
        else
        {
            result = formatExpr("(%s * %s)", leftExpr, rightDeriv);
            /*output the answer of the multiplication*/
        }
    }
}
else if (strcmp(rightExpr, "0") == 0)
{
    /*if the right expression is 0*/
    if (strcmp(leftDeriv, "0") == 0)
    {
        /*if the right expression and left derivative are all zero*/
        result = strdup("0");
        /*output 0*/
    }
    else
    {
        /*if the left derivative is not zero*/
        result = formatExpr("(%s * %s)", rightExpr, leftDeriv);
        /*output the answer of the multiplication*/
    }
}
else if (strcmp(leftDeriv, "0") == 0 && strcmp(rightDeriv, "0") == 0)
```

```
        {
            /*if the left and right derivative are all zero*/
            result = strdup("0");
            /*output 0*/
        }
        else if (strcmp(leftDeriv, "0") == 0)
        {
            /*if the left derivative is zero*/
            result = formatExpr("(%s * %s)", leftExpr, rightDeriv);
            /*output the answer of the multiplication*/
        }
        else if (strcmp(rightDeriv, "0") == 0)
        {
            /*if the right derivative is zero*/
            result = formatExpr("(%s * %s)", rightExpr, leftDeriv);
            /*output the answer of the multiplication*/
        }
        else
        {
            /*if there's no space for simplification*/
            result = formatExpr("(%s * %s + %s * %s)", leftExpr,
            rightDeriv, rightExpr, leftDeriv);
            /*output the answer of the multiplication*/
        }
        break;
    case '/':
        result = formatExpr("((%s * %s - %s * %s) / (%s ^ 2))",
        leftDeriv, rightExpr, rightDeriv, leftExpr, rightExpr);
        /*output the answer of the division by the division derivative
        law*/
        break;
    case '^':
        {
            char *powExpr = getNodeExpr(node);
            /*get the power of the expression*/
            char *term1 = formatExpr("%s * ln(%s)", rightDeriv,
            leftExpr);
            /*separate the expression into two parts*/
            char* term2 = formatExpr("%s * %s / %s", rightExpr,
            leftDeriv, leftExpr);
            char *sumTerms = formatExpr("(%s + %s)", term1, term2);
            /*get the sumterms*/
            result = formatExpr("%s * %s", powExpr, sumTerms);
            free(term1);
            /*free the memory space*/
            free(term2);
```

```
                    free(sumTerms);
                    free(powExpr);
                }
                break;
            default:
                result = strdup("0");
                /*default output*/
        }
        free(leftDeriv);
        free(rightDeriv);
        free(leftExpr);
        free(rightExpr);
        /*free all the memory that is malloced*/
        return result;
    }
    return strdup("0");
}

void calculateGrad(Node *root) {
    if (!root) {
        printf("Invalid input!\n");
        return;
    /*if the expression tree is not generated, then return NULL*/
    }

    char* variables[TOKEN_MAX_NUM];
    /*create the variable list*/
    int varCount = 0;
    /*count the number of variables*/
    collectVariables(root, variables, &varCount);
    /*collect variables recursively from the root pointer of the entire
    expression tree*/
    if (varCount == 0) {
        printf("Underivable Expression!\n");
        /*if there is no variable, then the expression is underivable*/
        return;
    }

    qsort(variables, varCount, sizeof(char*), compareStrings);
    /*sort the variables in the lexicographical order, with compareStrings() providing the c
    /*because the requirement is to output with the lexicographical order, I use this.*/

    for (int i = 0; i < varCount; i++) {
        char* derivExpr = derive(root, variables[i]);
        /*calculate the derivative of the expression*/
        /*traversing through every variable from the root*/
```

```
        printf("%s: %s\n", variables[i], derivExpr);
        /*output the variable and their derivatives*/
        free(derivExpr);
        /*setting free memory space*/
    }

    for (int i = 0; i < varCount; i++) {
        free(variables[i]);
        /*setting free the memory space*/
    }
}

int compareStrings(char *a, char *b) {
    return strcmp(* (char **)a, * (char **)b);
    /*compare the strings in the lexicographical order, which is going to be used in the qs
}
```

Declaration

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