

# Autograd for Algebraic Expressions Report

2023.11.8

浙江大学

## Chapter 1: Introduction

### Background and Motivation

In the field of machine learning and deep learning, automatic differentiation has played a pivotal role in simplifying the training of complex neural networks through backpropagation. It allows us to efficiently compute gradients, a critical component for optimizing model parameters. However, the power of automatic differentiation is not limited to deep learning; it finds applications in various domains where derivatives of functions are essential.

The motivation behind this project is to extend the capabilities of automatic differentiation to algebraic expressions. Algebraic expressions are fundamental in mathematics and computer science, serving as building blocks for various calculations and computations. By developing a program that can automatically differentiate algebraic expressions, we aim to simplify the process of finding derivatives for such expressions, making it easier for researchers, engineers, and students to work with mathematical functions and optimize their computations.

### Project Overview

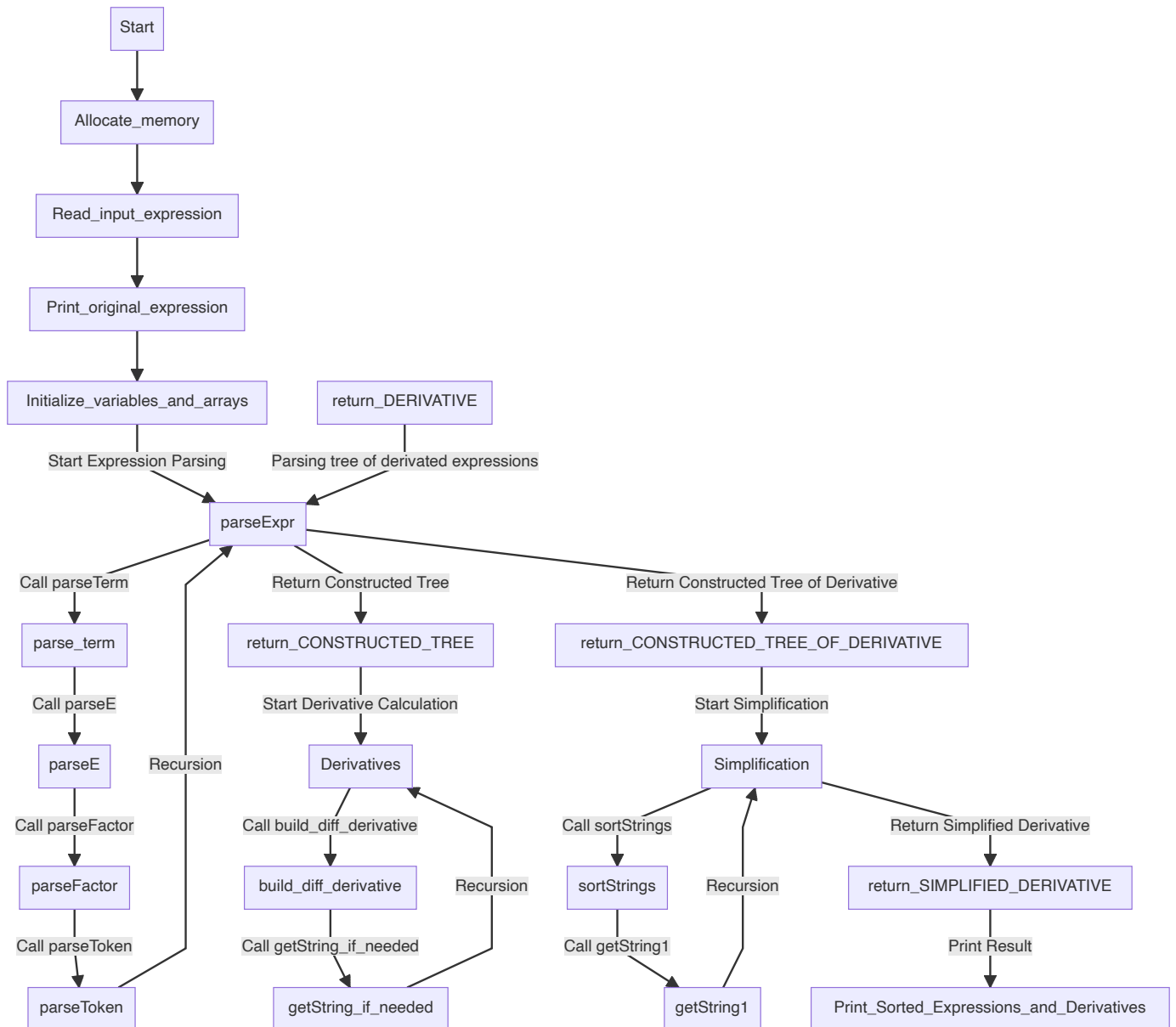
This project focuses on implementing an automatic differentiation program for algebraic expressions. The program is designed to accept algebraic expressions composed of operators, mathematical functions, and operands and provide the corresponding derivative expressions with respect to specified variables.

In this report, we will provide a comprehensive overview of the project, covering the following key aspects:

- **Algorithm Overview:** We will detail the algorithms and methods used to build expression trees, process algebraic expressions, output derivative expressions, and handle mathematical functions. Additionally, we will discuss potential simplification rules for reducing expression complexity.
- **Testing Results:** We will present various test cases to evaluate the program's accuracy and robustness. These test cases will include standard expressions, comprehensive testing scenarios, as well as small, large, and extreme cases.
- **Analysis and Comments:** We will analyze the time and space complexity of the implemented algorithm and provide insights into the code quality.
- **Source Code :** The source code of the program is provided below.

## Chapter 2: Algorithm Overview

## ketch of the main program



## PRESEDO\_CODE of key algorithms and datatype

As illustrated above, the program included few steps to calculate the derivative of the input expression. The program struct the expression tree by parsing the input expression and creating nodes for each token.

The tree is traversed to process the expression and calculate the derivative.

The program calculates the derivative of an expression by traversing the expression tree and applying the derivative rules for each operator.

The derivative of a function is the derivative of the function's body.

## TREE STRUCT and STRING STRUCT

The program uses a tree structure to store the expression. Each node of the tree contains a string that represents an operator, operand, or mathematical function. The tree is built by parsing the input expression and creating nodes for each token. The tree is traversed to process the expression and calculate the derivative.

```
1  typedef struct node *Node;
2  struct node {
3      char* data;
4      Node left;
5      Node right;
6  };
```

The program uses a string structure to store the expression and Variables. Each node of the tree contains a string that represents an operator, operand, or mathematical function. The tree is built by parsing the input expression and creating nodes for each token. The tree is traversed to process the expression and calculate the derivative.

```
1  char* parseToken(char** expr);
2  char* getString(Node node);
3  char opr[100][100]; //store the strings of the variables
```

## TREE BUILD

The program struct the expression tree by parsing the input expression and creating nodes for each token. The tree is traversed to process the expression and calculate the derivative.

here is the presedo of the expression tree building

```
1  function parseToken(expr):
2      Skip leading whitespace
3
4      If expr points to a letter:
5          Read the letter character, build a string temp
6          If temp is not in opr:
7              Add temp to opr
8          Return temp as an operand
9      If expr points to an operator:
10         Return the operator character as an operator
11     If expr points to a digit:
12         Read digit characters, build a string temp
13         Return temp as an operand
14
15 function parseExpr(expr):
16     left = parseTerm(expr)
17
```

```
18 while expr points to '+' or '-':
19     op = parseToken(expr)
20     right = parseTerm(expr)
21     node = createNode(op)
22     node.left = left
23     node.right = right
24     left = node
25
26 Return left as root
27
28 function parseTerm(expr):
29     left = parseE(expr)
30
31 while expr points to '*' or '/':
32     op = parseToken(expr)
33     right = parseE(expr)
34     node = createNode(op)
35     node.left = left
36     node.right = right
37     left = node
38
39 Return left as root
40
41 function parseE(expr):
42     left = parseFactor(expr)
43
44 while expr points to '^':
45     op = parseToken(expr)
46     right = parseFactor(expr)
47     node = createNode(op)
48     node.left = left
49     node.right = right
50     left = node
51
52 Return left as root
53
54 function parseFactor(expr):
55     token = parseToken(expr)
56
57 If token is equal to "(":
58     node = parseExpr(expr)
59     If expr points to ')':
60         Move expr to the next character
61         Return node as root
62 Else:
63     Handle error: Missing right parenthesis
64     Return NULL
```

```

65 Else:
66     Create a node with the value token
67     Return the node as a leaf node
68
69 Node root = parseExpr(&expr);

```

Here is the explanation of the presedo

```

1  parseToken: This function parses the next token (operand or operator) in an expression. It skips leading whitespace
   and returns the next token as a string.
2
3  parseExpr(+): This function parses an expression and builds the corresponding expression tree. It calls parseTerm to
   parse the left operand and parseTerm to parse the right operand. It then creates a new node for the operator and returns
   it as the root of the expression tree.
4
5  parseTerm(*): This function parses a term and builds the corresponding expression tree. It calls parseE to parse the
   left operand and parseE to parse the right operand. It then creates a new node for the operator and returns it as the root
   of the expression tree.
6
7  parseE(^): This function parses a power and builds the corresponding expression tree. It calls parseFactor to parse
   the left operand and parseFactor to parse the right operand. It then creates a new node for the operator and returns it as
   the root of the expression tree.
8
9  parseFactor(): This function parses a factor and builds the corresponding expression tree. It calls parseToken to
   parse the next token. If the token is a left parenthesis, it calls parseExpr to parse the expression inside the parenthesis.
   It then creates a new node for the expression and returns it as the root of the expression tree.

```

## DERIVATIVE CALCULATION

The program calculates the derivative of an expression by traversing the expression tree and applying the derivative rules for each operator. The derivative of a function is the derivative of the function's body. The derivative of a sum is the sum of the derivatives. The derivative of a product is  $(\text{left}' * \text{right}') + (\text{left}' * \text{right})$ . The derivative of a quotient is  $((\text{left}' * \text{right}) - (\text{left} * \text{right}')) / (\text{right}^2)$ . The derivative of a power is  $(\text{left}^{\text{right}}) * (\text{right}' * \ln(\text{left}) + \text{right} * \text{left}' / \text{left})$ . The derivative of a variable is 1.(and so on)

Here is the presedo of the derivative calculation

```

1  function derivative(root, var):
2      If root is NULL:
3          Return "0"
4      If root is the variable:
5          Return "1"
6      If root is a sum:
7          left_derivative = derivative(root.left, var)
8          right_derivative = derivative(root.right, var)
9          Return left_derivative + right_derivative
10     If root is a product:

```

```

11     left = getString(root.left)
12     right = getString(root.right)
13     left_derivative = derivative(root.left, var)
14     right_derivative = derivative(root.right, var)
15     Return (left * right_derivative) + (left_derivative * right)
16 If root is a quotient:
17     left_derivative = derivative(root.left, var)
18     right_derivative = derivative(root.right, var)
19     Return ((left_derivative * right) - (left * right_derivative)) / (right ^ 2)
20 If root is a power:
21     left = getString(root.left)
22     right = getString(root.right)
23     left_derivative = derivative(root.left, var)
24     right_derivative = derivative(root.right, var)
25     Return (left ^ right) * (right_derivative * ln(left) + right * left_derivative / left)
26 If root is a function:
27     inner_derivative = derivative(root.left, var)
28     Return inner_derivative
29 If root is a negative number:
30     left_derivative = derivative(root.left, var)
31     right_derivative = derivative(root.right, var)
32     Return left_derivative - right_derivative
33 getString(root):
34     If root is NULL:
35         Return "0"
36     If root is a variable:
37         Return the variable name
38     If root is a number:
39         Return the number
40     left = getString(root.left)
41     right = getString(root.right)
42     If root is a operator:
43         Return "(" + left + root + right + ")"

```

here is the explanation of the presedo

```

1 | derivative: This function calculates the derivative of an expression. It traverses the expression tree and applies the
  | derivative rules for each operator.
2 |
3 | getString: This function converts an expression tree to a string. It traverses the expression tree and converts each node
  | to a string. It then concatenates the strings of the left and right subtrees with the operator in the middle.

```

Also, the program uses a lot of 'If' terms to handle the special cases, such as the negative number, the power, and the function.

For example, the negative number

```

1  If root is a negative number:
2      left_derivative = derivative(root.left, var)
3      right_derivative = derivative(root.right, var)
4      if(left_derivative == "0" && right_derivative == "0"){
5          return "0";
6      }
7  .....and so on

```

## SIMPLIFICATION

The program simplifies the derivative expression by applying simplification rules. The rules are applied in a loop until no further simplification is possible. The rules are as follows:

```

1  1.  $0 + x = x + 0 = x$ 
2  2.  $0 * x = x * 0 = 0$ 
3  3.  $1 * x = x * 1 = x$ 
4  4.  $x + x = 2 * x$ 
5  5.  $x * x = x ^ 2$ 
6  6.  $x / x = 1$ 
7  7.  $x ^ 0 = 1$ 
8  8.  $x ^ 1 = x$ 

```

## SORTING

The program sorts the variables in lexicographical order. It uses the strcmp function to compare the strings and swap them if the right operand is smaller than the left operand in lexicographical order.

```

1  function sortStrings(strings, count):
2      For i = 0 to count - 2:
3          For j = i + 1 to count - 1:
4              If compare(string[i],strings[j])://strings[i] > strings[j]
5                  Swap strings[i] and strings[j]
6  function compare(string a, string b):
7      while a and b are not empty:
8          If a[i] == '(' || a[i] == ')' || a[i] == '^' || a[i] == '*' || a[i] == '/' || a[i] == '+' || a[i] == '-':
9              Remove a[i] from a
10         If b[i] == '(' || b[i] == ')' || b[i] == '^' || b[i] == '*' || b[i] == '/' || b[i] == '+' || b[i] == '-':
11             Remove b[i] from b
12         If a[i] > b[i]:
13             Return 1
14         If a[i] < b[i]:
15             Return -1
16         Increment i

```



## Chapter 3: Testing Results

### Test Cases

The table below outlines a framework for test cases:

Input Expression	Derivative that get	test reason
$x^2 + 2x + 1$	$(2*(x^{(2-1)}))$	check the ^ and * and + operants
$x - 2 + 2/x$	$(1 + ((-2)/(x^2)))$	check the - and / operants
$x^2 + 2x + 1$	$((2*(x^{(2-1)})) + 2)$	check the ^ and * operants
a	1	shortest example

### Comprehensive Testing

check long variable

```
1 | INPUT:
2 | ab2ds+ssd-dad^dsa
3 | OUTPUT:
4 | The sorted expression:
5 | d/dx(ab2ds): 1;
6 | d/dx(dad): (-((dad^(dsa-1))*dsa));
7 | d/dx(dsa): (-((dad^dsa)*Indad));
8 | d/dx(ssd): 1
```

check the () oprands

```
1 | INPUT:
2 | (a+b)*(c+d)
3 | OUTPUT:
4 | The sorted expression:
5 | d/dx(a): (c+d);
6 | d/dx(b): (c+d);
7 | d/dx(c): (a+b);
8 | d/dx(d): (a+b)
```

check the long expression

```

1 INPUT:
2 (a+b)/c^d-xy/(o-k)+c*d*f*g+b/c+2*a+f*f
3 OUTPUT:
4 The sorted expression:
5 d/dx(a): ((1/(c^d))+2)
6 d/dx(b): ((1/(c^d))+(1/c))
7 d/dx(c): ((((-(a+b))*((c^d-1)*d))/((c^d)^2))+(d*(f*g)))+((-b)/(c^2)))
8 d/dx(d): ((((-(a+b))*((c^d)*Inc))/((c^d)^2))+(c*(f*g)))
9 d/dx(f): (2*f+((c*d)*g))
10 d/dx(g): ((c*d)*f)
11 d/dx(k): (-(((1)*(-xy))/((o-k)^2)))
12 d/dx(o): (-((-xy))/((o-k)^2)))
13 d/dx(xy): (-1/(o-k))

```

The testing result and the output of the program are corroborated, which means the program is correct.

## Chapter 4: Analysis and Comments

### Time and Space Complexity Analysis

#### Time Complexity

##### TREE CONSTRUCT

The time complexity of the overall expression parsing and tree construction is  $O(n)$ , where  $n$  is the length of the input expression. This complexity arises from linear operations such as reading characters, parsing tokens, and constructing the expression tree.

##### DERIVATIVE CALCULATION

Recursive Calls:

Both functions make recursive calls on the left and right children of the current node.

The depth of recursion is proportional to the height of the expression tree, which can be up to  $O(n)$ , where  $n$  is the number of nodes in the tree.

String Operations:

Both functions use string operations such as `strcmp`, `strcpy`, `strcat`, `sprintf`, each with a time complexity of  $O(m)$ , where  $m$  is the length of the strings involved.

Overall:

The overall time complexity for both functions is  $O(nm)$ , where  $n$  is the number of nodes in the expression tree, and  $m$  is the average length of strings involved in string operations.

## **SIMPLIFICATION AND SORTING**

strc Function:

The function iterates through the characters of both strings, performing comparisons and skipping specific characters.

The loop runs in linear time with respect to the lengths of the input strings, resulting in a time complexity of  $O(\max(\text{lena}, \text{lenb}))$ .

getString1 Function:

The function makes recursive calls on the left and right children of the current node.

The depth of recursion is proportional to the height of the expression tree, which can be up to  $O(n)$ , where  $n$  is the number of nodes in the tree.

String operations such as strcmp, strcpy, strcat, sprintf are used, each with a time complexity of  $O(m)$ , where  $m$  is the length of the strings involved.

Overall, the time complexity for the getString1 function is  $O(nm)$ , where  $n$  is the number of nodes in the expression tree, and  $m$  is the average length of strings involved in string operations.

## **TIME COMPLEXITY OF THE OVERALL PROGRAM**

The overall time complexity of the program is  $O(nm)$ , where  $n$  is the number of nodes in the expression tree, and  $m$  is the average length of strings involved in string operations.

since  $n$  and  $m$  are both related to the length of the expression, the overall time complexity of the program is  $O(n^2)$ , where  $n$  is the length of the input expression.

## **Space Complexity**

### **TREE CONSTRUCT**

The space complexity of the overall expression parsing and tree construction is  $O(n)$ , where  $n$  is the length of the input expression. This complexity arises from linear operations such as reading characters, parsing tokens, and constructing the expression tree.

Other factors, such as the opr array, used to store the variables, and the num variable, have a space complexity of less than  $n$  (also  $O(n)$ ), and thus do not affect the overall space complexity.

## **DERIVATIVE CALCULATION**

Recursive Calls:

Both functions use recursion, adding space to the call stack. The maximum depth of recursion is proportional to the height of the expression tree, which can be up to  $O(n)$ , where  $n$  is the number of nodes in the tree.

Temporary Strings:

Temporary strings are created using malloc for intermediate results during string operations. The space complexity for these temporary strings is proportional to the length of the strings involved in operations.

Overall:

The overall space complexity for both functions is  $O(n + m)$ , where  $n$  is the number of nodes in the expression tree, and  $m$  is the total length of temporary strings.

## SIMPLIFICATION AND SORTING

strc Function:

The function uses a constant amount of space for variables  $i$ ,  $j$ ,  $lena$ ,  $lenb$ ,  $numa$ ,  $numb$ , and  $c$ .

Overall, the space complexity for the strc function is  $O(1)$ .

getString1 Function:

The function makes recursive calls, adding space to the call stack. The maximum depth of recursion is proportional to the height of the expression tree, which can be up to  $O(n)$ , where  $n$  is the number of nodes in the tree.

Temporary strings are created using malloc for intermediate results during string operations.

The space complexity for these temporary strings is proportional to the length of the strings involved in operations.

Overall, the space complexity for the getString1 function is  $O(n + m)$ , where  $n$  is the number of nodes in the expression tree, and  $m$  is the total length of temporary strings.

## SPACE COMPLEXITY OF THE OVERALL PROGRAM

The overall space complexity of the program is  $O(n + m)$ , where  $n$  is the number of nodes in the expression tree, and  $m$  is the total length of temporary strings.

since  $n$  and  $m$  are both related to the length of the expression, the overall space complexity of the program is  $O(n)$ , where  $n$  is the length of the input expression.

## Chapter 5: Source Code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int n_of_time = 0;
6  // Define the node of the expression tree
7  int num = 0, nu=0;
8  char opr[100][100]; //store the strings of the variables
9
```

```

10 typedef struct node *Node;
11 struct node {
12     char* data;
13     Node left;
14     Node right;
15 } ;// Node is a pointer to the struct node
16
17 Node parseTerm(char** expr);//parse the term
18 Node parseFactor(char** expr);//parse the factor
19 Node parseE(char** expr);//parse the power
20 int strcmp(char* a, char* b);//compare two strings and skip "("
21 int my_isspace(int c) {
22     return c == ' ' || c == '\t' || c == '\n' || c == '\v' || c == '\f' || c == '\r';
23 }
24
25 int my_isalpha(int c) {
26     return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');
27 }
28 int my_isdigit(int c) {
29     return c >= '0' && c <= '9';
30 }
31 int my_isalnum(int c) {
32     return my_isalpha(c) || my_isdigit(c);
33 }
34 int my_ispunct(int c) {
35     return c == '+' || c == '-' || c == '*' || c == '/' || c == '^' || c == '(' || c == ')';
36 }
37 // Function to create a new node
38 Node createNode(char* data) {
39     n_of_time++;
40     Node newNode = (Node)malloc(sizeof(struct node));// Allocate memory for the new node
41     newNode->data = data;
42     newNode->left = newNode->right = NULL;
43     return newNode;
44 }
45
46 // Function to parse the next token (operand or operator) in an expression
47 char* parseToken(char** expr) {
48     n_of_time++;
49     // Skip the leading spaces
50     while (my_isspace(**expr)) {
51         (*expr)++;
52     }
53
54     char* start = *expr;
55
56     if (my_isalpha(**expr)) {

```

```

57 char temp[100]; // Assuming the maximum length of a string is 100
58 int tempIndex = 0;
59
60 // Read the string into a temporary array
61 while(my_isalnum(**expr)) {
62     temp[tempIndex++] = **expr;
63     (*expr)++;
64 }
65 temp[tempIndex] = '\0';
66
67 // Check if the string already exists
68 int exists = 0;
69 for (int i = 0; i < num; i++) {
70     if (strcmp(opr[i], temp) == 0) {
71         exists = 1;
72         break;
73     }
74 }
75
76 // If the string does not exist, add it to the array
77 if (!exists) {
78     strcpy(opr[num++], temp);
79 }
80
81 nu = 0;
82 }
83
84 // If the next token is an operator
85 else if (my_ispunct(**expr)) {
86     (*expr)++;
87 }
88 // If the next token is a number
89 else if (my_isdigit(**expr)) {
90     while (my_isdigit(**expr)) {
91         (*expr)++;
92     }
93 }
94
95 size_t len = *expr - start;
96 char* token = (char*)malloc(len + 1);
97 strncpy(token, start, len);
98 token[len] = '\0';
99
100 return token;
101 }
102
103 // Function to parse an expression and build the corresponding expression tree
104 Node parseExpr(char** expr) {

```

```

104     n_of_time++;
105     Node left = parseTerm(expr);
106
107     while (**expr == '+' || **expr == '-') // If the next token is an operator
108     {
109         char* op = parseToken(expr); // Parse the operator
110         Node right = parseTerm(expr); // Parse the right operand
111         Node node = createNode(op); // Create a new node for the operator
112         node->left = left;
113         node->right = right;
114         left = node;
115     }
116
117     return left;
118 }
119
120 Node parseTerm(char** expr) {
121     n_of_time++;
122     Node left = parseE(expr);
123
124     while (**expr == '*' || **expr == '/') // If the next token is an operator
125     {
126         char* op = parseToken(expr);
127         Node right = parseE(expr);
128         Node node = createNode(op);
129         node->left = left;
130         node->right = right;
131         left = node;
132     }
133
134     return left;
135 }
136 Node parseE(char **expr){
137     n_of_time++;
138     Node left = parseFactor(expr);
139     while(**expr == '^') // If the next token is an operator
140     {
141         char* op = parseToken(expr);
142         Node right = parseFactor(expr);
143         Node node = createNode(op);
144         node->left = left;
145         node->right = right;
146         left = node;
147     }
148     return left;
149
150 }

```

```

151 Node parseFactor(char** expr) {
152     n_of_time++;
153     char* token = parseToken(expr);
154
155     if (strcmp(token, "(") == 0) // If the next token is a left parenthesis
156     {
157         Node node = parseExpr(expr);
158         if (**expr==')') {
159             (*expr)++; // Skip the right parenthesis
160             return node;
161         } else {
162             // Handle error: missing right parenthesis
163             return NULL;
164         }
165     }
166
167     return createNode(token);
168 }
169 char* getString(Node node)
170 {
171     //n_of_time++;
172     if (node == NULL) {
173         char *s = (char*)malloc(100);
174         strcpy(s, "0"); // return a pointer to a new string which is a duplicate of the string s
175         return s;
176     }
177     // If the operation is commutative (i.e., the order of the operands doesn't matter)
178     char* left = getString(node->left);
179     char* right = getString(node->right);
180     if(strcmp(node->data, "+") == 0 || strcmp(node->data, "*") == 0) // and the right operand is smaller than the left
operand in lexicographical order, swap them.
181     {
182         if(strc(left, right) > 0)
183         {
184             char* temp = left;
185             left = right;
186             right = temp;
187         }
188     }
189
190     if(strcmp(node->data, "+") == 0 || strcmp(node->data, "-") == 0 || strcmp(node->data, "*") == 0 || strcmp(node-
>data, "/") == 0 || strcmp(node->data, "^") == 0)
191     {
192         char* result = (char*)malloc(strlen(left) + strlen(right) + 4);
193         sprintf(result, "(%s%s%s)", left, node->data, right);
194         return result;
195     }

```



```

196     return node->data;
197 }
198
199 // Function to calculate the derivative of an expression
200 char* derivative(Node root, char* var) {
201     n_of_time++;
202     if (root == NULL) {
203         return "0";
204     }
205
206     if (strcmp(root->data, var) == 0) // If the current node is the variable
207     {
208         return "1";
209     }
210
211     if (strcmp(root->data, "+") == 0) {
212
213         char* left_derivative = derivative(root->left, var);
214         char* right_derivative = derivative(root->right, var);
215         //
216         size_t left_len = strlen(left_derivative);
217         size_t right_len = strlen(right_derivative);
218         char* result = (char*)malloc(100);
219         // simplify the expression by using the rule of the derivative of the sum
220         if (strcmp(left_derivative, "0") == 0 && strcmp(right_derivative, "0") == 0) // If both derivatives are 0
221         {
222             result = "0";
223             //printf("%s", result);
224             return result;
225         }
226         if (strcmp(left_derivative, "0") == 0) // If the left derivative is 0
227         {
228             snprintf(result, 100, "%s", right_derivative);
229             //printf("%s", result);
230             return result;
231         }
232         if (strcmp(right_derivative, "0") == 0) {
233             snprintf(result, 100, "%s", left_derivative);
234             //printf("%s", result);
235             return result;
236         }
237         snprintf(result, 100, "(%s+%s)", left_derivative, right_derivative);
238         printf("%s", result);
239         return result;
240     }
241
242     if (strcmp(root->data, "^") == 0) {

```

```

243 // Use the chain rule and the derivative formula of the power function to derive
244 char* left = getString(root->left);
245 char* right = getString(root->right);
246 if(strcmp(left, "0") == 0 ){
247     return "0";
248 }
249 if(strcmp(right, "0") == 0 ){
250     return "1";
251 }
252 char* left_derivative = derivative(root->left, var);
253 char* right_derivative = derivative(root->right, var);
254
255 size_t left_len = strlen(left_derivative);
256 size_t right_len = strlen(right_derivative);
257 char* result = (char*)malloc(100);
258 // simplify the expression by using the rule of the derivative of the power
259 if(strcmp(left_derivative, "0") == 0 && strcmp(right_derivative, "0") == 0){
260     result = "0";
261     return result;
262 }
263 if(strcmp(left_derivative, "1") == 0 && strcmp(right_derivative, "1") == 0){
264     snprintf(result, 100, "%s^%s*(%s/%s+ln%s)", left, right, right, left, right);
265     return result;
266 }
267 if(strcmp(left_derivative, "0") == 0 && strcmp(right_derivative, "1") == 0){
268     snprintf(result, 100, "(%s^%s*ln%s)", left, right, left);
269     return result;
270 }
271 if(strcmp(left_derivative, "1") == 0 && strcmp(right_derivative, "0") == 0){
272     snprintf(result, 100, "(%s^(%s-1))*%s)", left, right, right);
273     return result;
274 }
275 if(strcmp(left_derivative, "0") == 0){
276     snprintf(result, 100, "(%s^%s*%s*ln%s)", left, right, right_derivative, left);
277     return result;
278 }
279 if(strcmp(right_derivative, "0") == 0){
280     snprintf(result, 100, "(%s^(%s-1))*%s*%s)", left, right, left_derivative, right);
281     return result;
282 }
283 if(strcmp(left_derivative, "1") == 0){
284     snprintf(result, 100, "(%s^%s*(%s*ln%s+%s/%s*%s))", left, right, right_derivative, left, right, left, right);
285     return result;
286 }
287 if(strcmp(right_derivative, "1") == 0){
288     snprintf(result, 100, "%s^%s*(%s/%s+ln%s)", left, right, left, right, left);
289     return result;

```

```

290     }
291     snprintf(result, 100, "%s^%s*(%s*In%s+%/s/%s*%s)", left, right, right_derivative, left, right, left, left_derivative);
292     return result;
293 }
294
295 if (strcmp(root->data, "(") == 0) {
296     // The derivative of a function is the derivative of the function's body
297     char* inner_derivative = derivative(root->left, var); //
298     return inner_derivative;
299 }
300 if (strcmp(root->data, "-") == 0) {
301
302     char* left_derivative = derivative(root->left, var);
303     char* right_derivative = derivative(root->right, var);
304
305     // TODO: concatenate left_derivative and right_derivative
306     // return the result
307     size_t left_len = strlen(left_derivative);
308     size_t right_len = strlen(right_derivative);
309     char* result = (char*)malloc(100);
310     // simplify the expression by using the rule of the derivative of the sum
311     if (strcmp(left_derivative, "0") == 0 && strcmp(right_derivative, "0") == 0){
312         result = "0";
313         return result;
314     }
315     if (strcmp(left_derivative, "0") == 0) {
316         snprintf(result, 100, "(-%s)", right_derivative);
317         return result;
318     }
319     if (strcmp(right_derivative, "0") == 0) {
320         snprintf(result, 100, "(%s)", left_derivative);
321         return result;
322     }
323     snprintf(result, 100, "(%s-%s)", left_derivative, right_derivative);
324     return result;
325 }
326
327 if (strcmp(root->data, "*") == 0) {
328     char* left = getString(root->left);
329     char* right = getString(root->right);
330
331     char* left_derivative = derivative(root->left, var);
332     char* right_derivative = derivative(root->right, var);
333
334     // The derivative of a product is (left * right') + (left' * right)
335     // So, construct the result accordingly.
336     char* result = (char*)malloc(100);

```

```

337 // simplify the expression by using the rule of the derivative of the product
338 if (strcmp(left_derivative, "0") == 0 && strcmp(right_derivative, "0") == 0) {
339     result = "0";
340     //printf("%s", result);
341     return result;
342 }
343 if (strcmp(left_derivative, "1") == 0 && strcmp(right_derivative, "1") == 0) {
344     snprintf(result, 100, "(%s+%s)", left, right);
345     //printf("%s", result);
346     return result;
347 }
348 if (strcmp(left_derivative, "0") == 0 && strcmp(right_derivative, "1") == 0) {
349     snprintf(result, 100, "%s", left);
350     //printf("%s", result);
351     return result;
352 }
353 if (strcmp(left_derivative, "1") == 0 && strcmp(right_derivative, "0") == 0) {
354     snprintf(result, 100, "%s", right);
355     //printf("%s", result);
356     return result;
357 }
358 if (strcmp(left_derivative, "0") == 0) {
359     snprintf(result, 100, "(%s*%s)", left, right_derivative);
360     //printf("%s", result);
361     return result;
362 }
363 if (strcmp(right_derivative, "0") == 0) {
364     snprintf(result, 100, "(%s*%s)", right, left_derivative);
365     //printf("%s", result);
366     return result;
367 }
368 if (strcmp(left_derivative, "1") == 0) {
369     snprintf(result, 100, "(%s+%s*%s)", right, left, right_derivative);
370     //printf("%s", result);
371     return result;
372 }
373 if (strcmp(right_derivative, "1") == 0) {
374     snprintf(result, 100, "(%s+%s*%s)", left, right, left_derivative);
375     //printf("%s", result);
376     return result;
377 }
378
379 snprintf(result, 100, "(%s*%s+%s*%s)", left, right_derivative, left_derivative, right);
380 //printf("%s", result);
381 return result;
382 }
383

```

```

384 if (strcmp(root->data, "/") == 0) {
385     char* left = getString(root->left);
386     char* right = getString(root->right);
387     if(left == NULL || right == NULL){
388         return "0";
389     }
390
391     char* left_derivative = derivative(root->left, var);
392     char* right_derivative = derivative(root->right, var);
393
394     // The derivative of a quotient is ((left' * right) - (left * right')) / (right ^ 2)
395     // So, construct the result accordingly.
396     char* result = (char*)malloc(100);
397     // simplify the expression by using the rule of the derivative of the quotient
398     if (strcmp(left_derivative, "0") == 0 && strcmp(right_derivative, "0") == 0) {
399         result = "0";
400         return result;
401     }
402     if (strcmp(left_derivative, "1") == 0 && strcmp(right_derivative, "1") == 0) {
403         snprintf(result, 100, "(%s-%s)/%s^2", right, left, right);
404         return result;
405     }
406     if (strcmp(left_derivative, "0") == 0 && strcmp(right_derivative, "1") == 0) {
407         snprintf(result, 100, "(-%s)/%s^2", left, right);
408         return result;
409     }
410     if (strcmp(left_derivative, "1") == 0 && strcmp(right_derivative, "0") == 0) {
411         snprintf(result, 100, "1/%s", right);
412         return result;
413     }
414     if (strcmp(left_derivative, "0") == 0) {
415         snprintf(result, 100, "(-%s)*(%s)/%s^2", left, right_derivative, right);
416         return result;
417     }
418     if (strcmp(right_derivative, "0") == 0) {
419         snprintf(result, 100, "(%s)/(%s)", left_derivative, right);
420         return result;
421     }
422     if (strcmp(left_derivative, "1") == 0) {
423         snprintf(result, 100, "(%s-%s*%s)/%s^2", right, left, right_derivative, right);
424         return result;
425     }
426     if (strcmp(right_derivative, "1") == 0) {
427         snprintf(result, 100, "(%s*%s-%s)/%s^2", left_derivative, right, left, right);
428         return result;
429     }
430

```

```

431
432
433     snprintf(result, 100, "(%s*%s-%s*%s)/%s^2", left_derivative, right, left, right_derivative, right);
434     return result;
435 }
436
437     return "0";
438 }
439 void preorderTraversal(Node node) // Print the expression tree in preorder
440 {
441     if (node) {
442         printf("%s ", node->data); // Print the data of the current node
443         preorderTraversal(node->left); //
444         preorderTraversal(node->right); //
445     }
446 }
447 }
448
449 void sortStrings(char strings[][100], int count)//sort the strings
450 {
451     for (int i = 0; i < count - 1; i++) {
452         for (int j = i + 1; j < count; j++) {
453             if (strcmp(strings[i], strings[j]) > 0) {
454                 char temp[100];
455                 strcpy(temp, strings[i]);
456                 strcpy(strings[i], strings[j]); //swap the strings
457                 strcpy(strings[j], temp);
458             }
459         }
460     }
461 }
462
463 void insert_0(char* str, int poi)//insert '0' in the string to handle the negative number
464 {
465     int len = strlen(str);
466     for(int i = len; i > poi; i--)
467     {
468         str[i] = str[i-1];
469     }
470     str[poi] = '0';
471 }
472 char* getString1(Node node);
473 int main() {
474     char* expr ; // The expression to be parsed
475     expr = (char*)malloc(1000); // Allocate memory for the expression
476     scanf("%s", expr); // Read the expression from the user
477     printf("Original expression: %s\n", expr);

```

```

478
479 int m = strlen(expr);
480
481 for(int i=0;i<m;i++)//handle the negative number
482 {
483     if(expr[i]=='-'&& i==0)//if the negative number is the first number
484     {
485         insert_0(expr,0);
486         m++;
487     }
488     if(expr[i]=='-'&& expr[i-1]=='(')//if the negative number is in the bracket
489     {
490         insert_0(expr,i);
491         m++;
492     }
493 }
494 // Parse the expression and build the expression tree
495
496 Node root = parseExpr(&expr);
497 //("Expression tree: ");
498 preorderTraversal(root);// Print the expression tree in preorder
499 printf("\n");
500
501 int count = num;
502 // Sort the variables in lexicographical order
503 sortStrings(opr, count);
504
505 char out[100][100];
506 // Calculate the derivative of the expression
507 for(int i = 0; i < count; i++){
508     char* tmp = derivative(root, opr[i]);
509     strcpy(out[i], tmp);
510 }
511 for(int i = 0; i < count; i++){
512     printf("d/dx(%s): %s\n", opr[i], out[i]);// Print the derivative of the expression
513 }
514 Node ROOT[100];
515 for(int i = 0; i < count; i++){
516     char *tmp = out[i];
517     int m = strlen(tmp);
518
519     for(int i=0;i<m;i++)//handle the negative number
520     {
521         if(tmp[i]=='-'&& i==0)//if the negative number is the first number
522         {
523             insert_0(tmp,0);
524             m++;

```

```

525     }
526     if(tmp[i]=='-'&&tmp[i-1]=='(')//if the negative number is in the bracket
527     {
528         insert_0(tmp,i);
529         m++;
530     }
531 }
532 //printf("%s\n", tmp);
533
534 ROOT[i] = parseExpr(&tmp);
535
536 char* tmp1 = getString1(ROOT[i]);
537
538 strcpy(out[i], tmp1);
539 }
540 printf("The sorted expression: \n");
541 for(int i = 0; i < count; i++){
542     printf("d/dx(%s): %s\n", opr[i], out[i]);
543 }
544 // system("pause");
545 printf("%d ", n_of_time);
546 return 0;
547 }
548 int strc(char* a, char* b)//compare two strings and skip "("
549 {
550     //n_of_time++;
551     int i = 0;
552     int j = 0;
553     int lena = strlen(a);
554     int lenb = strlen(b);
555     while(i < lena && j < lenb)//compare the two strings
556     {
557         double numa = (double)a[i];
558         double numb = (double)b[j];
559         char c = a[i];
560         if(c == '+' || c == '-' || c == '*' || c == '/' || c == '^' || c == '(' || c == ')')
561         {
562             i++;
563             continue;
564         }
565         c=b[j];
566         if(c == '+' || c == '-' || c == '*' || c == '/' || c == '^' || c == '(' || c == ')')
567         {
568             j++;
569             continue;
570         }
571         if(a[i]-'a'>=0&&a[i]-'a'<=25)//change the lower case to upper case

```



```

572     {
573         numa += 'A'-'a'-0.1;
574     }
575     if(b[j]-'a'>=0&&b[j]-'a'<=25)//change the lower case to upper case
576     {
577         numb += 'A'-'a'-0.1;
578     }
579
580     if(a[i] == b[j])//a==b
581     {
582         i++;
583         j++;
584         continue;
585     }
586     else if(numa < numb)//a<b
587     {
588         return 0;
589     }
590     else if(numa > numb)//a>b
591     {
592         return 1;
593     }
594 }
595 return 0;
596 }
597 char* getString1(Node node) //simple the expression
598 {
599     //n_of_time++;
600     if (node == NULL) {
601         char *s = (char*)malloc(100);
602         strcpy(s, "0"); // return a pointer to a new string which is a duplicate of the string s
603         return s;
604     }
605     char* left = getString1(node->left);//get the left string
606     char* right = getString1(node->right);//get the right string
607
608     // If the operation is commutative (i.e., the order of the operands doesn't matter)
609     // and the right operand is smaller than the left operand in lexicographical order,
610     // swap them.
611     if (strcmp(node->data, "+") == 0 || strcmp(node->data, "*") == 0) {
612         if (strcmp(left, right) > 0) {
613             char* temp = left;
614             left = right;
615             right = temp;
616         }
617     }
618     if(strcmp(node->data, "+") == 0 )

```

```

619 {
620     if(strcmp(left, "0") == 0) // 0+x
621     {
622         return right;
623     }
624     if(strcmp(right, "0") == 0) // x+0
625     {
626         return left;
627     }
628     if(strcmp(left, right) == 0) // x+x
629     {
630         char* result = (char*)malloc(strlen(left) + 2); // +2 for the '(' and the null terminator
631         strcpy(result, "2*");
632         strcat(result, left);
633         return result;
634     }
635     char* result = (char*)malloc(strlen(left) + strlen(right) + 4);
636     sprintf(result, "(%s%s%s)", left, node->data, right); // +4 for the '(' and the ')' and the null terminator
637     return result;
638 }
639 if(strcmp(node->data, "*") == 0) //multiply
640 {
641     if(strcmp(left, "0") == 0 || strcmp(right, "0") == 0) // 0*x or x*0
642     {
643         return "0";
644     }
645     if(strcmp(left, "1") == 0) // 1*x
646     {
647         return right;
648     }
649     if(strcmp(right, "1") == 0) // x*1
650     {
651         return left;
652     }
653     char* result = (char*)malloc(strlen(left) + strlen(right) + 4);
654     sprintf(result, "(%s%s%s)", left, node->data, right);
655     return result;
656 }
657 if(strcmp(node->data, "^") == 0) //power
658 {
659     if(strcmp(left, "0") == 0) // 0^x
660     {
661         return "0";
662     }
663     if(strcmp(right, "0") == 0) // x^0
664     {
665         return "1";

```

```

666     }
667     if(strcmp(right, "1") == 0) // x^1
668     {
669         return left;
670     }
671     char* result = (char*)malloc(strlen(left) + strlen(right) + 4);
672     sprintf(result, "(%s%s%s)", left, node->data, right);
673     return result;
674 }
675 if(strcmp(node->data, "/") == 0 )
676 {
677
678     if(strcmp(left, "0") == 0) // 0/x
679     {
680         return "0";
681     }
682     if(strcmp(right, "1") == 0) // x/1
683     {
684         return left;
685     }
686     if(strcmp(left, right) == 0) // x/x
687     {
688         return "1";
689     }
690     char* result = (char*)malloc(strlen(left) + strlen(right) + 4);
691     sprintf(result, "(%s%s%s)", left, node->data, right); // +4 for the '(' and the ')' and the null terminator
692     return result;
693 }
694 // If the operation is not commutative (i.e., the order of the operands matters)
695 if(strcmp(node->data, "-") == 0 )
696 {
697     if(strcmp(left, "0") == 0) // 0-x
698     {
699         char* result = (char*)malloc(strlen(right) + 4);
700         sprintf(result, "(-%s)", right);
701         return result;
702     }
703     if(strcmp(right, "0") == 0) // x-0
704     {
705         return left;
706     }
707     if(strcmp(left, right) == 0) // x-x
708     {
709         return "0";
710     }
711     char* result = (char*)malloc(strlen(left) + strlen(right) + 4);
712     sprintf(result, "(%s%s%s)", left, node->data, right); // +4 for the '(' and the ')' and the null terminator

```

```
713 |     return result;  
714 | }  
715 | return node->data;  
716 | }
```

## Declaration

---

I hereby declare that all the work done in this project titled "Autograd for Algebraic Expressions Report" is of my independent effort.