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

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.

Specifications for *data structures*

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;
```

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;
}
```

Specifications for *Algorithms*

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;
```

```
  }
}
```

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) {
  intialize 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;
}
```

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';
}
```

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;
}
```

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;
    }
  }
}
```

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);
  }
}
```

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 + _xyˆab	expression with multi-letter variable name	_xy: (_xy ˆ ab) * (0 * ln(_xy) + ab * 1 / _xy), ab: (_xy ˆ ab) * (1 * ln(_xy) + ab * 0 / _xy), xx: (1 + (_xy ˆ ab) * (0 * ln(_xy) + ab * 0 / _xy))	*pass*
(a+b)ˆc	expression with parentheses	a: ((a + b) ˆ c) * (0 * ln((a + b)) + c * 1 / (a + b)) b: ((a + b) ˆ c) * (0 * ln((a + b)) + c * 1 / (a + b)), c: ((a + b) ˆ 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 only one variable	a: 1	*pass*
a +	invalid input expression	Invalid input	*pass*

After airthmetic simplification on our result, we can discover that our results are all correct.

4. Analysis and Comments

Analysis of the time and space complexities of the algorithms. Comments on further possible improvements.

For the runtime of Selection Sort, there are two subtasks inside the *for*-loop. Specifically,

- finding the smallest integer between `list[i]` to `list[n-1]` takes $n - i$ time units;
- interchaning `list[i]` and `list[min]` takes a constant time c.

This is independent of any particular input. Therefore, both the average and worst time complexity can be computed as:

$$T(n) = \sum_{i=0}^{n-1} (n - i + c) = \Theta(n^2).$$

There is still considerable room for improvement. For example, we could adopt the divide-and-conquer strategy to achieve an $O(n \log n)$ algorithm.

For the space requirement, since we merely need an array to store the n integers, the space complexity is $\Theta(n)$ and should be optimal.

Appendix: Source Code (in C)

Please make sure that your code is sufficiently commented. Otherwise, it will *not* be evaluated.

Sample Code:

```c
/*
 * Find the smallest element in a range of an array
 * ------------------------------------------------
 *
 *   arr: an array of integers
 *   begin: the index to start the search
 *   end: the index to stop the search (non-inclusive)
 *
 *   returns: the index of the smallest element in the range
 */
int find_min(int arr[], int begin, int end) {
  int min_idx = begin;
  int min_val = arr[min_idx];

  for (int idx = min_idx+1; idx < end; ++idx) {
    if (arr[idx] < min_val) {
      min_idx = idx;
```

```
      min_val = arr[min_idx];
    }
  }

  return min_idx;
}

/*
 * Swap the values of two integers
 * -------------------------------
 *
 *   a: pointer to the 1st integer
 *   b: pointer to the 2nd integer
 *
 *   returns: none
 */
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

/*
 * Rearrange an array of integers into increasing order in-place
 * -------------------------------------------------------------
 *
 *   arr: an unsorted array of integers
 *   n: number of elements in the array
 *
 *   returns: none
 */
void sort(int arr[], int n) {
  for (int i = 0; i < n; ++i) {
    int min_idx = find_min(arr, i, n);
    /*after the swap, arr[i] is the i-th smallest integer*/
    swap(&arr[i], &arr[min_idx]);
  }
}
```

Note: Feel free to use your own style of code or comments. Just make sure to be consistent with yourself.

Declaration

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