Autograd for Algebraic Expression

Project 2 Report
Fundamental of Data Structure (2023 Fall)
Zhejiang University BY

Author Name

Student ID:
Teacher: HongHua Gan

Accomplished on October 27, 2023

Made
With $\LaTeX$

Declaration

I hereby declare that all the work done in this project titled “Autograd for Algebraic Expression” is of my independent effort.

Contents

Chapter 1

Introduction

This chapter gives a brief introduction to the project, including the problem description, problem background, and the context-free grammar of the expression.

1.1 Problem Description

In this project, we are going to implement an autograd system for analysing and differentiating algebraic expressions. Given a mathematical expression in the form of a string, the task is to develop a program that can automatically analyse the structure of the expression and evaluate its derivative with respect to every variable appearing in the expression. The program should be able to handle expressions with multiple variables, support basic mathematical operations, and accurately compute its derivative.

1.2 Problem Background

Autograd, short for automatic differentiation, is a technique used in computational mathematics and machine learning to efficiently compute derivatives of functions. It plays a crucial role in optimization algorithms, neural network training [1], and various scientific computations.

By implementing such a system, it becomes possible to automate the evaluation and differentiation of algebraic expressions, saving time and effort in various mathematical and scientific applications.

1.3 Context-free Grammar of the Expression

The expression defined in the problem can be more systematically described using a context-free grammar to guide the translation from string to expression analyse tree.

A context-free grammar consists of three parts: a set of **terminal symbols** (represent in bold), a set of *non-terminal symbols* (represent in italics), and a set of production rules. Terminal symbols are the basic symbols of the language, while non-terminal symbols are symbols that can be derived into terminal symbol strings. Production rules describe how non-terminal symbols can be constructed from terminal and non-terminal symbols. One of the non-terminal symbols is designated as the start symbol, which is the symbol that can be derived into the entire language.

Based on the problem description, the terminal symbols of the expression are defined as follows:

Table 1.1: Terminal of the Expression

Symbol Name	Example	Description
digits	`19 1`	A sequence of digits.
variable	`x xyz`	A sequence of lowercase letters that is not a function name.
operator	`+ - * / ^`	An operator.
fun1	`sin cos tan exp ln`	A function with one argument.
fun2	`pow log`	A function with two arguments.
bracket	`( )`	A bracket.
comma	`,`	A comma.

The approach to defining non-terminal symbols is as follows:

- A non-terminal symbol should be defined for each level of operator precedence.
- A terminal symbol should be defined to represent factors that cannot be separated by operators, such as variables, mathematical functions, etc.

The expression mentioned in the problem description contains three different levels of operators, so I defined four non-terminal symbols. The resulting four non-terminal symbols and their production rules can be seen in Figure 1.1, and the associativity of the operators is also indicated in the production rules. The start symbol of the grammar is *expr*.

Now let's have a look at the parse tree of the expression `sin(x + 1) * 2`:

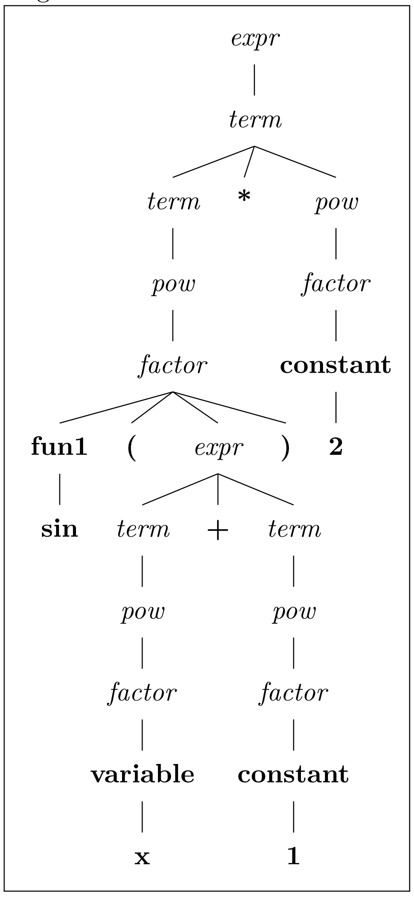
- The root of the tree is the start symbol *expr*.
- Match the production rule of *expr* with the input string, but found it could only be a *term*.
- Match the production rule of *term* with the input string, and succeeded with pattern *term* `*` *pow*.
- Recursively match the production rule of *term* and *pow* to the left and right sub-string of the above pattern.

The parsed tree is shown in Figure 1.2.

Figure 1.1: Non-terminals and their productions

$$
\begin{aligned}
\textit{factor} &\to \textbf{digits} \\
&\mid \texttt{-}\ \textit{factor} \\
&\mid +\ \textit{factor} \\
&\mid \textbf{variable} \\
&\mid \textbf{fun1}(\textit{expr}) \\
&\mid \textbf{fun2}(\textit{expr},\textit{expr}) \\
&\mid (\textit{expr}) \\
\textit{pow} &\to \textit{factor} \\
&\mid \texttt{-}\ \textit{pow} \\
&\mid +\ \textit{pow} \\
&\mid \textit{pow}\ \hat{}\ \textit{factor} \\
\textit{term} &\to \textit{pow} \\
&\mid \texttt{-}\ \textit{term} \\
&\mid +\ \textit{term} \\
&\mid \textit{term}\ \texttt{*}\ \textit{pow} \\
&\mid \textit{term}\ /\ \textit{pow} \\
\textit{expr} &\to \textit{term} \\
&\mid \textit{expr}\ +\ \textit{term} \\
&\mid \textit{expr}\ \texttt{-}\ \textit{term}
\end{aligned}
$$

Figure 1.2: Parse tree of `sin(x+1)*2`



Chapter 2

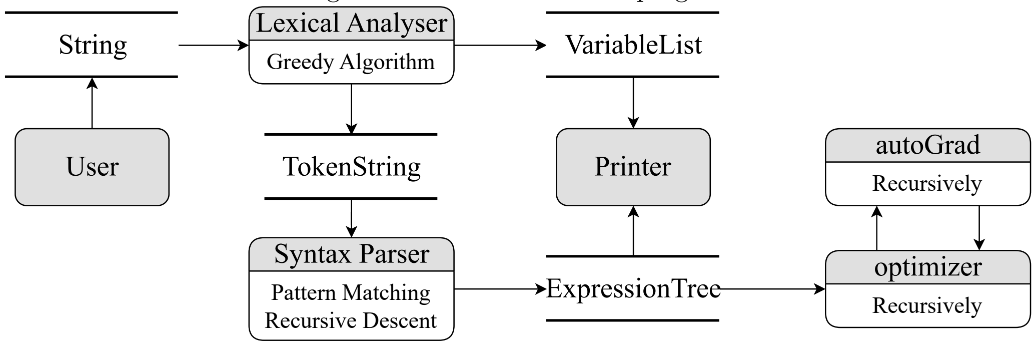
Algorithm Specfication

This chapter gives description of the main program and the algorithms used in each part.

2.1 Architecture of the Program

The program structure of the autoGrad program is similar to that of a compiler. The expression in string format goes through a lexical analyser and a syntax parser, continuously abstracting it into higher-level expressions and finally generating an expression tree. The optimizer and differentiator operate on the expression tree data structure. The collaboration between different modules of the program is illustrated in the following diagram:

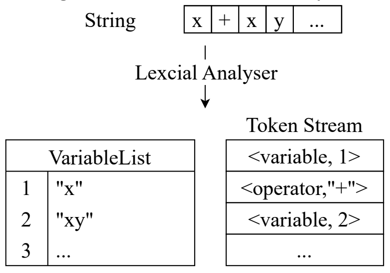
Figure 2.1: Data flow of the program



2.2 Lexcial Analyser

Tokens are terminals of the expression. Lexcial analyser uses a greedy algorithm to match the longest possible token from the input string. It converts the string into a token stream, which is then passed to the syntax parser.

Figure 2.2: Lexcial Analyser



2.2.1 `String`, `Token` and `VariableList` Data Structure

The `String` type is just a wrapper type used for convenient input and keeping track of the current position during parsing. It dynamically allocates memory based on the input size. Each time the lexer is called on a `String`, it updates the position that has been read in the string.

The `token` type is defined as a structure with two fields: `type` and `value`. `type` is an enumeration type that represents the type of the token (see 1.1), and `value` is an int type that represents the value of the token.

VariableList is a structure that stores the names of variables in the expression. It is implemented as an `char*` array for convenient add and query operations. Value of a variable token is the index of the variable in the VariableList. There is also an index array named `dictOrder` that is sorted by dictionary order. It is used to print the variable list in alphabetical order.

Table 2.1: API of `Token` and `VariableList` Type

Type	Name	Arguments	Description
String*	getString	(void)	Read one line from stdin and return a string
VariableList*	createVariableList	(void)	Create a variable list object
int	querySymbol	(VariableList *list, char *symbol)	Query the index of a variable in the list
int	addSymbol	(VariableList *list, char *symbol)	Add a variable to the list and return index
void	printVariableList	(VariableList *list)	Print variable list object
void	freeVariableList	(VariableList *list)	Free variable list object
Token	getToken	(String *s, VariableList *list)	Get a token from the string
void	printTokens	(Token *tokens, int num, VariableList *v)	Print token stream

2.2.2 Greedy Matching Algorithm

The algorithm is implemented in the function `getToken()`. It takes a string and a variable list as input, and returns the next token in the string.

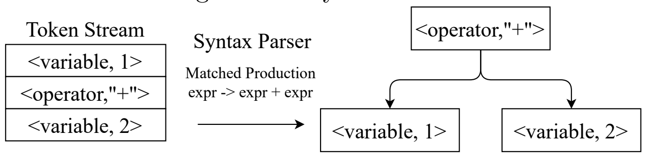
Algorithm 1: Greedy Matching Algorithm

```
Input: A string $s$ and a variable list $v$
Output: The next token in the string
$token \leftarrow$ NULL
while $s$ is not empty do
    $token \leftarrow$ NULL
    for $i \leftarrow 0$ to $n$ do
        if $s[0:i]$ is a token then
                                                            /* Greedy match token */
            $token \leftarrow s[0:i]$
        end
    end
    if $token$ is NULL then
        return NULL
    end
    else
        $s \leftarrow s[i:]$
                                                            /* Update string */
        if $token$ is a variable then
            $index \leftarrow$ querySymbol($v, token$)
                                                            /* Maintain variable list */
            if $index$ is NULL then
                $index \leftarrow$ addSymbol($v, token$)
            end
            $token \leftarrow$ createToken(VARIABLE, $index$)
        end
        return $token$
    end
end
```

2.3 Syntax Parser

Syntax parser receives a token stream from the lexical analyser and generates an expression tree. It uses a recursive descent algorithm to parse the token stream. The algorithm is implemented in the function `parse()`.

Figure 2.3: Syntax Parser



2.3.1 `Node` Data Structure

The expression tree is implemented as a binary tree. Each node of the tree is a structure with three fields: `token`, `left` and `right`. `Token` is described in the previous section. `left` and `right` are pointers to the left and right child of the node.

In the expression tree, the internal nodes are operators and functions, and the leaf nodes are variables and digits. Other types of tokens is just used for parsing and will not appear in the expression tree.

Table 2.2: API of `Node` Type

Type	Name	Arguments	Description
Node*	createNode	(Token t, Node *a, Node *b)	Create a node object
Node*	copyTree	(Node *node)	Copy a node object recursively
void	printTree	(Node *node, VariableList *v)	Print node object
void	freeNode	(Node *node)	Free node object recursively
int	isBracketPaired	(Token *t, int start, int end)	Check if brackets are paired
Node*	parse	(String *s, VariableList *list)	Wrapper function for getToken() and following functions
Node*	exprParser	(Token *t, int start, int end)	Parse the token stream into an expression tree
Node*	termParser	(Token *t, int start, int end)	Parse the token stream into a term tree
Node*	powParser	(Token *t, int start, int end)	Parse the token stream into a pow tree
Node*	factorParser	(Token *t, int start, int end)	Parse the token stream into a factor tree

2.3.2 Pattern Matching and Recursive Descent Algorithm

The algorithm is implemented in the function `exprParser()`, `termParser()`, `powParser()` and `factorParser()`. When the higher-priority parser function fails to parse the token stream or successfully matches a pattern, it will call the same-priority or lower-priority parser function to continue parsing the remaining portion of the token stream. The algorithm is described in the following block:

Algorithm 2: Recursive Descent Algorithm for non-terminal p

```
Input: A token stream $t$ and a variable list $v$
Output: An expression tree

When p is not the lowest-priority parser
if $t$ applies to the production rule of p then
    return the expression tree
end
else
    Call the lower-priority parser function
end

When p is the lowest-priority parser
if $t$ can't applies to the production rule of this parser then
    Report error
end
```

2.4 Optimizer

I implemented a simple optimizer that can simplify the expression tree by evaluating constant sub-expressions. The optimizer is implemented in the function `constantOptimizer()`. It will recursively apply the following rules to the expression tree and return a **new** optimized tree.

$$
\begin{aligned}
0+\mathrm{f}(x) = \mathrm{f}(x)+0 = \mathrm{f}(x)-0 &= \mathrm{f}(x)\\
0\cdot \mathrm{f}(x) = \mathrm{f}(x)\cdot 0 &= 0\\
0/\mathrm{f}(x) &= 0\\
1^{\mathrm{f}(x)} = \mathrm{f}(x)^0 &= 1\\
\ln(1) = \sin(0) = \tan(0) &= 0\\
\log_{\mathrm{f}(x)} \mathrm{f}(x) &= 1\\
\mathrm{pow}(\mathrm{f}(x),0) = \mathrm{pow}(1,\mathrm{f}(x)) &= 1\\
\mathrm{pow}(\mathrm{f}(x),\mathrm{f}(x)) &= \mathrm{f}(x)^{\mathrm{f}(x)}
\end{aligned}
$$

$$
\begin{aligned}
0-\mathrm{f}(x) &= -\mathrm{f}(x)\\
1\cdot \mathrm{f}(x) = \mathrm{f}(x)\cdot 1 &= 1\\
\mathrm{f}(x)/1 &= \mathrm{f}(x)\\
0^{\mathrm{f}(x)} &= 0\\
\cos(0) = \exp(0) &= 1\\
\log_{\mathrm{f}(x)} 1 &= 0\\
\mathrm{pow}(0,\mathrm{f}(x)) &= 0\\
\mathrm{pow}(\mathrm{f}(x),1) &= \mathrm{f}(x)
\end{aligned}
$$

Algorithm 3: Optimizer Algorithm

```
Input: An expression tree $t$
Output: An optimized expression tree
if $t$ has child then
    Optimize child
end
Construct a new node $n$ following the rules
return $n$
```

As for implementation, the equations above is converted to a bunch of `if` statements which is so diverse that I can't show them here. The details can be found in the function `constantOptimizer()`.

2.5 Differentiator

For every node (including leaf), differentiator `autoGrad` applies the following rules to the expression tree and return a **new** diffrentiated tree.

$$\begin{aligned}
\ln(\mathrm{f}(x))' &= \frac{\mathrm{f}'(x)}{\mathrm{f}(x)}\\
\sin(\mathrm{f}(x))' &= \mathrm{f}'(x)\cdot\cos(\mathrm{f}(x))\\
\cos(\mathrm{f}(x))' &= -\mathrm{f}'(x)\cdot\sin(\mathrm{f}(x))\\
\tan(\mathrm{f}(x))' &= \mathrm{f}'(x)\cdot\frac{1}{\cos^2(\mathrm{f}(x))}\\
\exp(\mathrm{f}(x))' &= \mathrm{f}'(x)\cdot\exp(\mathrm{f}(x))
\end{aligned}$$

$$\begin{aligned}
(\mathrm{f}(x)\pm\mathrm{g}(x))' &= \mathrm{f}'(x)\pm\mathrm{g}'(x)\\
(\mathrm{f}(x)\cdot\mathrm{g}(x))' &= \mathrm{f}'(x)\cdot\mathrm{g}(x)+\mathrm{f}(x)\cdot\mathrm{g}'(x)\\
(\frac{\mathrm{f}(x)}{\mathrm{g}(x)})' &= \frac{\mathrm{f}'(x)\cdot\mathrm{g}(x)-\mathrm{f}(x)\cdot\mathrm{g}'(x)}{\mathrm{g}(x)^2}\\
(\mathrm{f}(x)^{\mathrm{g}(x)})' &= \mathrm{f}(x)^{\mathrm{g}(x)}\cdot[\mathrm{g}'(x)\cdot\ln(\mathrm{f}(x))+\frac{\mathrm{g}(x)\cdot\mathrm{f}'(x)}{\mathrm{f}(x)}]\\
\log_{\mathrm{f}(x)}\mathrm{g}(x) &= \frac{\mathrm{g}'(x)}{\mathrm{g}(x)\cdot\ln(\mathrm{f}(x))}-\frac{\ln(\mathrm{g}(x))\cdot\mathrm{f}'(x)}{\mathrm{f}(x)}
\end{aligned}$$

Algorithm 4: Differentiator Algorithm

```
Input: An expression tree $t$, a variable $x$
Output: A differentiated expression tree
switch $t$ do
    case $t$ is a constant do
        return 0
    end
    case $t$ is a variable do
        if $t$ is $x$ then
            return 1
        end
        else
            return 0
        end
    end
    otherwise do
        Recursively construct a new node $n$ following the rules
        return $n$
    end
end
```

2.6 Printer

The printer is implemented in the function `printTree()`. It will recursively print the expression tree in infix notation.

The printer must check the priority of the operator and the associativity of the operator to determine whether to print brackets. It also needs to check whether the operator satisfies the commutative law to determine whether to print brackets.

Algorithm 5: Printer Algorithm

```
Input: A node $n$ and a variable list $v$
Output: Print the expression tree in infix notation
if $n$ is empty then
    return
end
switch $n$ do
    case $n$ is a variable, function or digit do
        Print
    end
    case $n$ is an operator do
        if Child of $n$ is lower-priority operator then
            Print bracket
        end
        Print left child
        Print operator
        Print right child
        if Child of $n$ is lower-priority operator then
            Print bracket
        end
        if $n$ dissatisfies the commutative law then
            if right child is same type as $n$ then
                Print bracket
            end
        end
    end
    otherwise do
        Report error: invalid node in tree
    end
end
```

Chapter 3

Testing Results

The program successfully passed all the test cases. Here I selected some test cases from my test suite to show the correctness of the program. A more complete test suite can be seen in Appendix B.

Selected test cases with their purpose, input, output and correctness are shown in Table 3.1 below.

Table 3.1: Test Results

Begin of Table				
No.	Purpose	Input	Output	Correctness
1	Test cases that only contain constants.	`-1^2`	`[warning] No variable in expression: (-1)`	Yes
2	Test cases that only contain variables and constants.	`a*b/(a-b)*a/b^2`	`a: ((((b*(a-b)-a*b)/(a-b)^2)*a+(a*b)/(a-b))*b^2)/(b^2)^2` `b: ((((a*(a-b)-(a*b)*(-1))/(a-b)^2)*a)*b^2-(((a*b)/(a-b))*a)*(b^2*(2/b)))/(b^2)^2`	Yes
3	Test cases that contain mathematical functions.	`sin(x)+cos(x)+tan(x)+exp(x)+ln(x)+log(2, x)+pow(x, 2)`	`x: cos(x)+(-1)*sin(x)+1/cos(x)^2+exp(x)+1/x+(ln(2)/x)/ln(2)^2+x^2*(2/x)`	Yes
4	Test cases that contain variables with different names.	`vlog+a+b+wxyz`	`a: 1` `b: 1` `vlog: 1` `wxyz: 1`	Yes
5	Test cases that contain nested expressions, multiple variables and functions.	`log(a,b)^2+sin(x)^2`	`a: log(a,b)^2*((((-1)*(ln(b)/a))/ln(a)^2)*(2/log(a,b)))` `b: log(a,b)^2*(((ln(a)/b)/ln(a)^2)*(2/log(a,b)))` `x: sin(x)^2*(cos(x)*(2/sin(x)))`	Yes
6	Test cases that can be simplified.	`x+x+x`	`x: 3`	Yes

Continuation of Table 3.1				
No.	Purpose	Input	Output	Correctness
7	Large Size	`a^a^a^a^a^a^a^a^a^a^a^a`	`a: a^(a^(a^(a^(a^(a^(a^(a^(a^(a^(a^(a^a)))))))))))*(a^(a^(a^(a^(a^(a^(a^(a^(a^(a^(a^a))))))))))/a+(a^(a^(a^(a^(a^(a^(a^(a^(a^(a^(a^a))))))))))*(a^(a^(a^(a^(a^(a^(a^(a^(a^(a^a)))))))))/a+(a^(a^(a^(a^(a^(a^(a^(a^(a^(a^a)))))))))*(a^(a^(a^(a^(a^(a^(a^(a^(a^a))))))))/a+(a^(a^(a^(a^(a^(a^(a^(a^(a^a))))))))*(a^(a^(a^(a^(a^(a^(a^(a^a)))))))/a+(a^(a^(a^(a^(a^(a^(a^(a^a)))))))*(a^(a^(a^(a^(a^(a^(a^a))))))/a+(a^(a^(a^(a^(a^(a^(a^a))))))*(a^(a^(a^(a^(a^(a^a)))))/a+(a^(a^(a^(a^(a^(a^a)))))*(a^(a^(a^(a^(a^a))))/a+(a^(a^(a^(a^(a^a))))*(a^(a^(a^(a^a)))/a+(a^(a^(a^(a^a)))*(a^(a^(a^a))/a+(a^(a^(a^a))*(a^(a^a)/a+(a^(a^a)*(a^a/a+(a^a*(1+ln(a)))*ln(a)))*ln(a)))*ln(a)))*ln(a)))*ln(a)))*ln(a)))*ln(a)))*ln(a)))*ln(a)))*ln(a))`	Yes
8	Small Size	`a`	`a: 1`	Yes
9	Complex Case	`log(log(x,y),sin(z))^2`	`x: log(log(x,y),sin(z))^2*((((-1)*((((-1)*(ln(y)/x))/ln(x)^2)*(ln(sin(z))/log(x,y))))/ln(log(x,y))^2)*(2/log(log(x,y),sin(z))))` `y: log(log(x,y),sin(z))^2*((((-1)*(((ln(x)/y)/ln(x)^2)*(ln(sin(z))/log(x,y))))/ln(log(x,y))^2)*(2/log(log(x,y),sin(z))))` `z: log(log(x,y),sin(z))^2*(((cos(z)*(ln(log(x,y))/sin(z)))/ln(log(x,y))^2)*(2/log(log(x,y),sin(z))))`	Yes
End of Table				

Chapter 4

Analysis and Comments

4.1 Algorithm Analysis

In this section I will analyze the time and space complexity of each algorithm described (besides algorithm of the printer which is not important) in Chapter 2.

4.1.1 Time and Space Complexity for Greedy Matching Algorithm

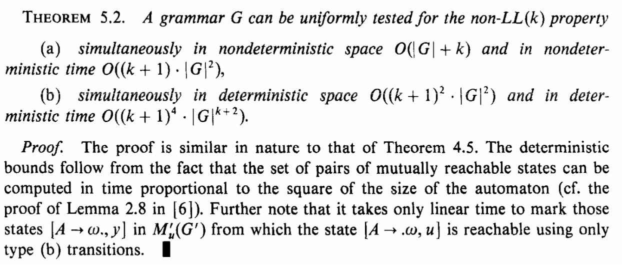
The Greedy Matching Algorithm is described in Alg 1. `strcmp()` is used to compare the tokens with the string, and the time complexity of `strcmp()` is $O(n)$, where n is the length of the string. Assume the number of tokens to be compared (include keywords and variable names that is recorded in the variable list) is m, then the time complexity of the algorithm is $O(nm)$. Usually the variable in an expression is far less than the keywords, so m can be seen as a constant. Therefore, the time complexity of the algorithm is $O(n)$. And the lexical analyzer will only scan the input string **once**, so the time complexity of the lexical analyzer is $O(n)$.

The lexical analyzer will only store the variable list, so the space complexity of the lexical analyzer is $O(m)$.

4.1.2 Time and Space Complexity for $LL(1)$ Predictive Syntax Parser

It's known that recursive descent parsers may require exponential time in some cases. Although the grammar described in the Introduction is left-recursive, the specific implementation is an $LL(1)$ grammar. The algorithm we implemented is actually a predictive parser, which is a recursive descent syntax analyzer that **does not require backtracking**. The parser scans the input from left to right, generating the leftmost derivation, and only needs to look ahead one symbol at each step to determine the syntax analysis action. The complexity proof of semantic analysis for $LL(k)$ grammar is shown in Fig 4.1 (from the paper *On the complexity of $LL(k)$ testing* [2]).

Figure 4.1: Complexity proof of semantic analysis for $LL(k)$ grammar



THEOREM 5.2. *A grammar G can be uniformly tested for the non-$LL(k)$ property*

(a) *simultaneously in nondeterministic space $O(|G| + k)$ and in nondeterministic time $O((k + 1) \cdot |G|^2)$,*

(b) *simultaneously in deterministic space $O((k + 1)^2 \cdot |G|^2)$ and in deterministic time $O((k + 1)^4 \cdot |G|^{k+2})$.*

Proof. The proof is similar in nature to that of Theorem 4.5. The deterministic bounds follow from the fact that the set of pairs of mutually reachable states can be computed in time proportional to the square of the size of the automaton (cf. the proof of Lemma 2.8 in [6]). Further note that it takes only linear time to mark those states $[A \rightarrow \omega., y]$ in $M'_u(G')$ from which the state $[A \rightarrow .\omega, u]$ is reachable using only type (b) transitions. ▮

The algorithm for constructing the canonical $LL(k)$ parser yields, as a byproduct, a test for the $LL(k)$ property. The problem of testing whether or not a context-free grammar possesses the $LL(k)$ property is studied. For each fixed integer $k \geq 0$, the problem is shown to be solvable in nondeterministic time $O(n)$, and simultaneously in deterministic space $O(n)$ and in deterministic time $O(n^{k+1})$, where n is the size of the grammar in question.

Using the above conclusion, we can conclude that the time and space complexity of our syntax analysis process is $O(n)$.

4.1.3 Time and Space Complexity for Optimizer

The optimizer is described in Alg 3. The optimizer will recursively apply the optimization rules to the expression tree. For each node, the optimizer will only execute a constant number of check and assignment operations, so the time complexity of the optimizer is $O(n)$ (n is the number of nodes in the expression tree).

As for space complexity, the optimizer will create a new expression tree, so the space complexity of the optimizer is $O(n)$.

4.1.4 Time and Space Complexity for Differentiator

The differentiator is described in Alg 4. The differentiator will recursively apply the differentiation rules to the expression tree.

The difference between the differentiator and the optimizer is that the differentiator may recursively call itself to implement chain rule. I tried to expand the differentiate results so that the differentiator will call itself at most twice for each node (to calculate $f'(x)$ and $g'(x)$). **On average,** the tree size of two child nodes is about half of the parent node, so the time complexity of the differentiator is:

$$T(n) = 4T(n/2) + O(1)$$

By master theorem, the time complexity of the differentiator is $O(n^2)$(n is the number of nodes in the expression tree).

The space complexity depends on the input expression. If the expression is deeply nested, the result expression will be more complex, so the space complexity will be higher. For example, if you use the expression below for input:

```
a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a
^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a
^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a
^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a
^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a
^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a^a
```

Then the result expression tree will have more than 40k nodes. For the worst case like this, we can infer that $S(n) = 2S(n-1) + S(1)$, so space complexity will be $O(2^n)$. On average, the space complexity of the differentiator usually is $O(2n)$ for polynomial expressions, with each node expaned to at most two nodes.

4.2 Advantages of the Program

- The program is easy to extend. It is easy to add new operators and functions to the program. The program can also be easily extended to support more complex expressions, such as matrix expressions.
- The program is easy to debug and maintain. The program is divided into several modules, and each module is responsible for a specific task. The program can be debugged by testing each module separately and can be easily maintained by modifying each module separately.
- The program is easy to use. The program is implemented in C, which is a widely used programming language. The program can be easily compiled and run on most platforms.

4.3 More on Data Structure

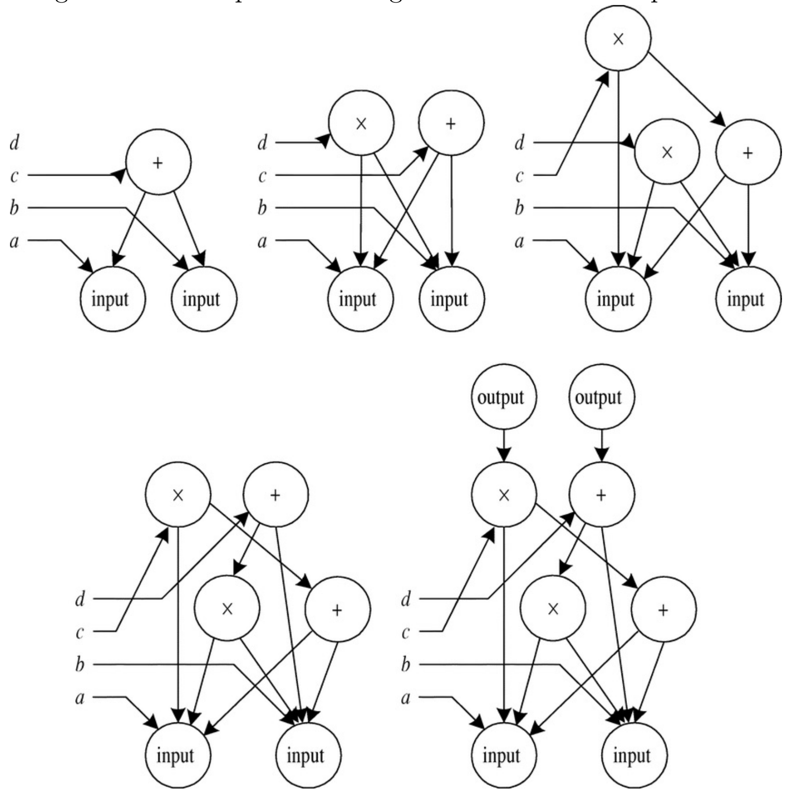
This program mainly uses tree data structure to store the expression. The tree data structure is suitable for storing expressions because it can represent the hierarchical structure of expressions. But the normal tree data structure also has some disadvantages:

- The tree data structure is not very efficient in terms of memory usage.
- It's hard to implement some operations on the tree data structure, such as collecting like terms.

A more efficient data structure for storing expressions is the DAG (directed acyclic graph) data structure. The DAG data structure can be obtained by removing the duplicate nodes in the expression tree. The DAG data structure can save memory and make some operations more efficient. Nowadays, most compilers use DAG data structure to store and optimize expressions. Fig 4.2 shows an example of building a DAG from infix expressions.

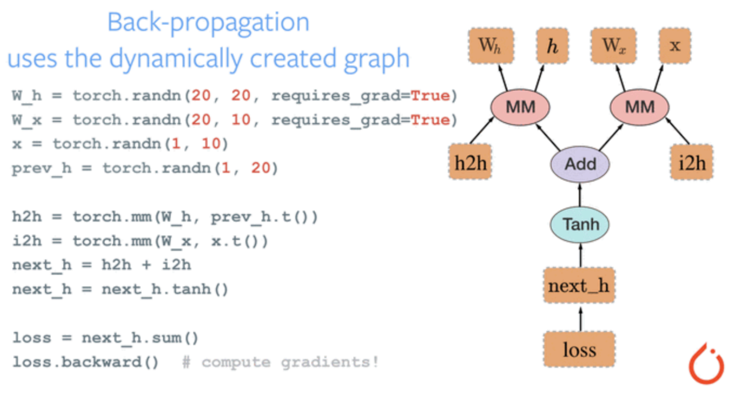
Using DAG to store expressions can also make some operations more efficient. For example, the **collecting like terms** operation can be implemented by traversing the DAG and merging the nodes with the same operator and operands. Autograd in DAG is also more efficient. The autograd algorithm in DAG can be implemented by traversing the DAG in the backward direction and merging the nodes with the same operator and operands.

Figure 4.2: Example of building a DAG from infix expressions.



Actually, the DAG data structure is used in the `autograd` module of PyTorch [1]. PyTorch is a widely used deep learning framework. The autograd module of PyTorch is used to automatically calculate the gradient of the loss function with respect to the parameters of the neural network. The autograd module uses DAG data structure to store the computation graph of the neural network. The computation graph is a DAG that represents the forward pass of the neural network. The autograd module will automatically calculate the gradient of the loss function with respect to the parameters of the neural network by traversing the computation graph in the backward direction. Fig 4.3 shows an example of building a computation graph from a neural network.

Figure 4.3: Example of building a computation graph from a neural network.



Back-propagation
uses the dynamically created graph

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)

h2h = torch.mm(W_h, prev_h.t())
i2h = torch.mm(W_x, x.t())
next_h = h2h + i2h
next_h = next_h.tanh()

loss = next_h.sum()
loss.backward()  # compute gradients!
```

Appendix A

Source Code

A.1 Source Code Statistics

The source code is in the `code` directory. I've commented the source code using Doxygen-style comments.

1. `main.c`: Driver program.
2. `expression.h, expression.c`: Implementation of the expression tree.

And here is the statistics of the source code above using `cloc` tool:

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                                2             58            471           1037
C/C++ Header                     1              9             81             67
-------------------------------------------------------------------------------
SUM:                             3             67            552           1104
-------------------------------------------------------------------------------
```

33% of the source code is well documented with Doxygen-style comments. This satisfies the requirement of the project.

A.2 Compilation and Execution

The program is written in C and can be compiled using `gcc`. There is a `Makefile` in the `code` directory. If you use UNIX system, you can compile the program by running `make` in the `code` directory.

If you use Windows system, you can compile the program by running the following command in the `code` directory:

```
gcc -o autoGrad.exe main.c expression.c
```

If you use IDE, you can also import the source code into the IDE and compile it.

Appendix B

Raw Test Results

Here I present the more detailed test cases and its results.

B.1 Test Cases

Here I classify the test cases into categories:

- **Constant**: Test cases that only contain constants.
  - `-1^2`
  - `(-1)^2`
  - `1-1+1`
  - `2*3/3`
- **Simple**: Test cases that only contain variables and constants.
  - `x^2+2*x+1`
  - `a^2+a^2+a^2+b*3+b^2`
  - `a+(b+c)*d`
  - `a*b/(a-b)*a/b^2`
  - `-a`
- **Math Function**: Test cases that contain mathematical functions.
  - `sin(x)+cos(x)+tan(x)+exp(x)+ln(x)+log(2, x)+pow(x, 2)`
  - `pow(x, x)`
- **variable Name**: Test cases that contain variables with different names.
  - `vlog+logp`
  - `a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z`
  - `abcdefghijklmnopqrstuvwxyz`
- **Complex and Nested**: Test cases that contain nested expressions, multiple variables and functions.
  - `xx^2/xy*xy+a^a`
  - `sin(x+1)*2+cos(x+1)*2`
  - `log(a,b)^2+sin(x)^2`
  - `a+(-b)^c`
  - `log(log(a,b),log(a,b))`
- **Optimization**: Test cases that can be simplified.
  - `log(a,a)`
  - `x+x+x`
  - `x^0+0^x`

B.2 Results

B.2.1 Constant

```
> -1^2
No variable in expression: (-1)
> (-1)^2
No variable in expression: 1
> 1-1+1
No variable in expression: 1
> 2*3/3
No variable in expression: 2
```

B.2.2 Simple

```
> x^2+2*x+1
x: x^2*2/x+2
> a^2+a^2+a^2+b*3+b^2
a: a^2*2/a+a^2*2/a+a^2*2/a
b: 3+b^2*2/b
> a+(b+c)*d
a: 1
b: d
c: d
d: b+c
> a*b/(a-b)*a/b^2
a: ((b*(a-b)-a*b)/(a-b)^2*a+a*b/(a-b))*b^2/b^2^2
b: ((a*(a-b)-a*b*(-1))/(a-b)^2*a*b^2-a*b/(a-b)*a*b^2*2/b)/b^2^2
> -a
a: (-1)
```

B.2.3 Math Function

```
> sin(x)+cos(x)+tan(x)+exp(x)+ln(x)+log(2, x)+pow(x, 2)
x: cos(x)+(-1)*sin(x)+1/cos(x)^2+exp(x)+1/x+ln(2)/x/ln(2)^2+x^2*2/x
> pow(x, x)
x: x^x*(1+ln(x))
```

B.2.4 Variable Name

```
> vlog+logp
logp: 1
vlog: 1
> a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z
a: 1
b: 1
c: 1
d: 1
e: 1
f: 1
g: 1
h: 1
i: 1
j: 1
k: 1
l: 1
m: 1
n: 1
o: 1
p: 1
q: 1
r: 1
```

```
s: 1
t: 1
u: 1
v: 1
w: 1
x: 1
y: 1
z: 1
> abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz: 1
```

B.2.5 Complex and Nested

```
> xx^2/xy*xy+a^a
a: a^a*(1+ln(a))
xx: xx^2*2/xx*xy/xy^2*xy
xy: (-1)*xx^2/xy^2*xy+xx^2/xy
> sin(x+1)*2+cos(x+1)*2
x: cos(x+1)*2+(-1)*sin(x+1)*2
> log(a,b)^2+sin(x)^2
a: log(a,b)^2*(-1)*ln(b)/a/ln(a)^2*2/log(a,b)
b: log(a,b)^2*ln(a)/b/ln(a)^2*2/log(a,b)
x: sin(x)^2*cos(x)*2/sin(x)
> a+(-b)^c
a: 1
b: ((-1)*b)^c*(-1)*c/(-1)*b
c: ((-1)*b)^c*ln((-1)*b)
> log(log(a,b),log(a,b))
a: (0)
b: (0)
```

B.2.6 Optimization

```
> log(a,a)
a: (0)
> x+x+x
x: 3
> x^0+0^x
x: (0)
```

Bibliography

[1] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[2] Seppo Sippu and Eljas Soisalon-Soininen. On the complexity of ll(k) testing. *Journal of Computer and System Sciences*, 26(2):244–268, 1983.