Computational Symbolic Differentiation & Integration

Dominic Altamura
Candidate for B.S. Degree
in Computer Science

State University of New York, College at Oswego
College Honors Program

March, 2023

Abstract

Computational symbolic differentiation and integration is a branch of computational mathematics that has been explored before, but is currently in a state of obfuscation. There are certainly many applications available that perform symbolic integration and differentiation flawlessly, but understanding *how* computers perform the necessary operations is not very clear. Many of the existing applications are not open source and those that are open source have source code that is difficult to follow and requires a large amount of previous knowledge in the world of computational algebra and calculus in order to properly understand how the algorithms work. This thesis aims to offer an open source alternative to existing software that clearly defines how the algorithms work and what the general workflow of the application is. The thesis first discusses how input from the user is parsed using an intuitive grammar and recursive descent parser. After this, the algorithms used for differentiation and integration are discussed in detail. The differentiation portion of the application is able to find the derivative of any given expression so long as it doesn't contain trigonometry, but the integration algorithm is limited to expressions that require the substitution rule to integrate. Further work on this project will require adding trigonometric differentiation and integration, a proprietary computational algebra system, and implementing the integration by parts algorithm.

Contents

1 Advice to Future Honors Students

I've read around a dozen of the honors theses we have available and they all offer the same advice. They remind you how time consuming the thesis is, how you have to schedule your progress wisely, and to make use of your thesis advisors as much as possible. I agree with all of these common talking points, but I think they're obvious to a lot of people who are getting ready to start their thesis. Perhaps reinforcement of the obvious is necessary, but there's plenty of theses available already to cover that need.

I truly only have one piece of advice that is worth saying: pick something you're almost obsessed with learning about. This is a large requirement and it is only necessary to merely be interested in the subject you're writing about, but being a little obsessed goes a long way. Over the last year and a half, I do not know if I would have been able to make the progress I did if I wasn't as invested in my project as I have been. I understand it's hard to find a topic you're incredibly passionate about, so I suggest starting by jumping into something you've always been curious about. The thesis you now read was a result of myself wondering how computers perform symbolic integration when I was a freshman. Look for things that have always piqued your interest but that you have never been able to properly explore. It's often in these topics that have already caught your interest that your multi-year thesis may lie.

All of this is to say that you ought to do something that is important to you. If you are to choose a topic that will be developed over at least a year and a half, you cannot just look at it as another writing assignment. Perhaps the topic you choose will be important to no one else, but it *has* to be of the utmost importance to you. If you don't feel this way about your thesis, it may become difficult to bring yourself to finish it. I'll be the first to admit that there were plenty of people who didn't find my thesis very important. I was and still am often reminded that symbolic integration calculators already exist and that I am wasting my time. I kept writing regardless because this thesis is important to *me*. Find a topic that makes you feel the same way.

I wish you luck on your thesis. Writing this thesis has been one of the most fulfilling things I've done at Oswego, and I have no doubt you'll choose a topic you are incredibly passionate about so that you will feel the same way once the thesis is written. I'll end with a small piece of financial advice. A few years ago, I hid twenty dollars in the pages of Dr. Daniel Schlegel's honors thesis. I would not be surprised if the money is still there, so if you go find it you will likely be twenty dollars richer.

2 Acknowledgments

This thesis would not have been anywhere near a completed state if it were not for my thesis advisors who have given me immense support throughout its development. I would like to especially thank Dr. Elizabeth Wilcox for her invaluable guidance throughout my senior year. I would also like to thank the friends who took the time out to read the thesis and give me their feedback. Along with the help of my advisors, their input has been crucial in making this thesis the best it could be. Finally, I would like to thank my grandfather, for teaching me that a job is never truly done and that there are always ways to improve. This thesis is dedicated to him.

3 Author's Reflections

Upon starting this thesis, I was very unsure of myself as a student. I have always gotten good grades and worked hard, but I always felt like I was missing something that everyone else in the Computer Science department seemed to have that I couldn't quite put my finger on. Because of this, I will admit I was a bit nervous when getting ready to start the thesis. I had known that I wanted to make a symbolic integration program since I was a freshman, but I did not know if it was something I had the skills to achieve.

Nevertheless, I went on with writing the thesis. I quickly learned just how much of the thesis involves reading. Though this thesis is largely a creative project, the initial research I needed to do was lengthy and took up the first four months or so of the thesis. No part of the thesis had a specific portion I could just develop on a whim, I first had to do research on the portion of the project I was making at the time. This experience helped me realize how important it is to review literature even when the project is a largely creative one, as the choices and advancements made prior often inspired the decisions I made when developing the application for this thesis. I also learned just how important peer review is for academic work. In addition to my thesis advisors, I've had a number of other people read this thesis. This is a complicated topic for people who are uninitiated in the background material and it has been important to me that it was possible for any undergraduate with some background in mathematics or computer science could pick up the thesis and understand what they are reading. My initial drafts tend to often be too verbose and complicated, so the people who were able to read my thesis and tell me what parts needed restructuring have brought me ever-closer to an easily-approachable thesis paper.

The peer review stages of this thesis have also reinforced my understanding that a first draft of any academic paper is rarely the last. We are often told throughout high school and college alike that one ought to never consider the first draft of anything they write to be perfect. I agree with this sentiment, but I would be lying if I said there weren't some general education classes I took where my final submission for some assignments were often closer to a minorly-edited first draft than a complete paper and I still received a good grade regardless. So, I always knew one should make multiple drafts of their writing, I just didn't know how important they were since I was often able to still receive good grades on the few papers I've submitted throughout college that I would consider a first draft. If I didn't have ten initial, regularly reviewed drafts of this thesis, there is no chance it would be in the condition it currently is. Drafts of writing projects, especially in extended projects like this thesis, are crucial in formulating ones' thoughts into an understandable, *final* piece of work. I understand that this is an obvious observation in hindsight, but it took me going through a project where drafting is absolutely necessary to truly understand its importance.

The most important thing I've learned through the completion of this thesis is that I am a capable student and programmer. I said at the start of this section that I always felt like I was missing something that all my peers seemed to have. I now realize the thing I was missing was *confidence.* Like I said, I have always done well in my classes but I have dealt with imposter syndrome throughout most of my experience in college. However, when I started reaching the end of this thesis I remembered back when I had come up with the idea of this thesis as a freshman. I wanted to dive into the topic of computational integration, but not thinking I would be capable of writing an application that seemed so complicated. Now, as a senior, I can safely say I was able to accomplish something I never knew I was capable of. Yes, it's not completely what I expected — there are a few features I had to cut for the sake of time — but I was able to largely achieve what I set out to accomplish back when I first had the idea for this thesis. As trite as it is, this thesis helped me believe in myself, and that is a lesson I will take not only through the rest of my academic career, but my life as well.

4 Introduction

There are many applications available that perform symbolic integration, but their inner-workings are not well-known. Most of these applications do not have their source code open to the public and those that do are difficult to follow and usually lack the documentation necessary to properly explain how any of the algorithms used for symbolic integration work. This thesis is the culmination of an effort to create an open-source, highly documented application that performs symbolic integration. Developing the application can be broken into three sections: development the expression parser, development of the algorithms used for differentiation, and development of the algorithms used for integration. The final application brings together multiple common computer science development techniques and design patterns that are understandable and intuitive to an average undergraduate student with some experience in programming. The final application is able to perform integration efficiently, but there are some rules that still need to be implemented and potentially require more complex algorithms and algebra systems.

5 Parsing

5.1 BNF Grammar

In order to be able to represent a mathematical expression in a form that can be passed through an algorithm for evaluation, it needs to be parsed in a way that defines all the different parts of the expression. To do this, mathematical expressions need to be contextualized with a new grammar that separates the different parts of the expression so that the application can evaluate them. To do this, rewriting an expression with a BNF grammar representation is an optimal, easy-to-follow solution.

Backus–Naur Form, or *BNF*, grammar uses rules called *productions* in order to parse a given expression (M. L. Scott, 2016, p. 49). These rules can be characterized as if-then statements where the antecedent of the statement is called a *non-terminal* and the consequent could either lead to another non-terminal or a *terminal* (M. L. Scott, 2016, p. 49). Non-terminals can be thought of as a part of grammar that must always lead to another production, meaning no parsed expression can only be made up of non-terminals. Terminals, however, are a part of grammar that can no longer be parsed and no terminal can be on the antecedent of a production. Before going on, there are some basic syntactic rules for the BNF grammars to demonstrate:

- Anything enclosed in brackets in the consequent means it is an optional part of the rule.
- Vertical lines represent the word 'or'. For instance, $x|y$ can be read as 'x or y'.
- Anything in quotes refers to its literal string representation.

Now, take this simple grammar for adding and subtracting integers, described below.

Expression → Factor [Operator Expression-Tail]

Expression-Tail → Operator Expression

Factor → Integer|Variable

Operator → '+'|'-'

Here, our terminal variables are Integer, Variable, the addition sign, and subtraction sign. For the sake of the demonstration, assume that Integer is any possible integer and Variable is any letter variable. The next section explains how combing regular expressions with the BNF grammar makes parsing numbers and variables very simple. Using this grammar, one can derive something like the following.

$$x + 3 = Expression(Factor(x)Expression - Tail(Operator('+')Expression(Factor(3)))$$

Using this, one can see what each part of the expression represents and that the expression is actually composed of other nested expressions.

5.2 Mathematical Expression Grammar Rules

The grammar used to parse mathematical expressions is designed such that it maintains the order of operations and is beneficial towards integrating expressions. The grammar is started off with what will be called *Expression* nodes. These Expression nodes are composed of *Term* nodes and optionally *Expression-2* or *Expression-3* nodes. The Term nodes are then composed of *Factor* nodes and optionally *Term-2* or *Term-3* nodes. A factor node can contain a constant value, a variable letter, a parenthesized Expression node, or a *Factor-Exponent* node. A parenthesized Expression node represents an Expression node wrapped in parentheses. Note that it is an Expression node that is parenthesized and not a Term or Factor node since any given operation can happen inside of parentheses. A Factor-Exponent node represents some Factor node brought to the power of another Factor node. The Factor-Exponent node is composed of a left Factor node and a right Factor node.

Term-2 and Term-3 nodes are attached to a Term node and are used to represent multiplication and division operations, respectively. Both the Term-2 and Term-3 nodes contain a Term node, and this represents the value that is being used to multiply or divide the original Term node's Factor node by. Using these grammar rules allows for us to chain multiplication or division operations together where each term being multiplied resides in its own node so that each component of an expression is connected to each other. These nodes are connected by using optional nodes to represent multiplication or division operations that are crucial in performing integration rules such as the substitution rule while each Term being multiplied can be separately explored while implementing integration rules, such as when substitution values need to be found for the substitution rule.

The Expression-2 and Expression-3 nodes work on a similar basis as the Term-2 and Term-3 nodes and have a similar structure where they represent values that are being added or subtracted from a Term node, respectively. The grammar is laid out this way specifically to aid the addition and subtraction rules of integration since it allows for us to separately integrate the Term nodes each Expression node is attached to so that they can later be added together, as will be seen when discussing how the addition and subtraction rules are performed. The full BNF grammar rules for integration for this application are as follows.

Expression → Term ([Expression-2] |[Expression-3])

Expression-2 → ‘+’ Expression

Expression-3 → ‘-’ Expression

Term → Factor ([Term-2] |[Term-3])

Term-2 → ‘*’ Term

Term-3 → ‘/’ Term

Factor → Constant|Variable|‘(’Expression‘)’ |Factor-Exponent |Natural-Log

Factor-Exponent → Factor ‘^’ Factor

Natural-Log → ‘ln[’ Expression ‘]’

Figure 1: Full BNF Grammar for every mathematical expression

As stated previously, the parsing algorithm mixes regular expression parsing with BNF grammar when parsing constants and variable letters. Regular expressions analyze a sequence of characters in search of a pattern to return a token, a string of characters that represent something (Scott, 2016, p. 45). With constants and variables, there are no nested expressions to consider like with the non-terminal parts of grammar, so regular expressions are used to convert either a letter variable or any possible constant to their respective data type. For constants, the parser needs to account for decimal numbers and negatives.

```
^(\-|(\d(\.))?)\d+(\.\d+)?
```

For letter expressions, the regular expression is fairly simple, as it only has to look for any type of letter. The parser also ensures that there is only one letter variable used throughout the expression, throwing an error otherwise. If the expression is composed entirely of constant variables, it is given the letter variable x by default for integration purposes.

```
^[A-Za-z]
```

The two things this parser does not account for are negative variables and negated parenthesized expressions. This is because these negated forms are actually just -1 multiplied by the given term, so when they appear, they are converted to this form.

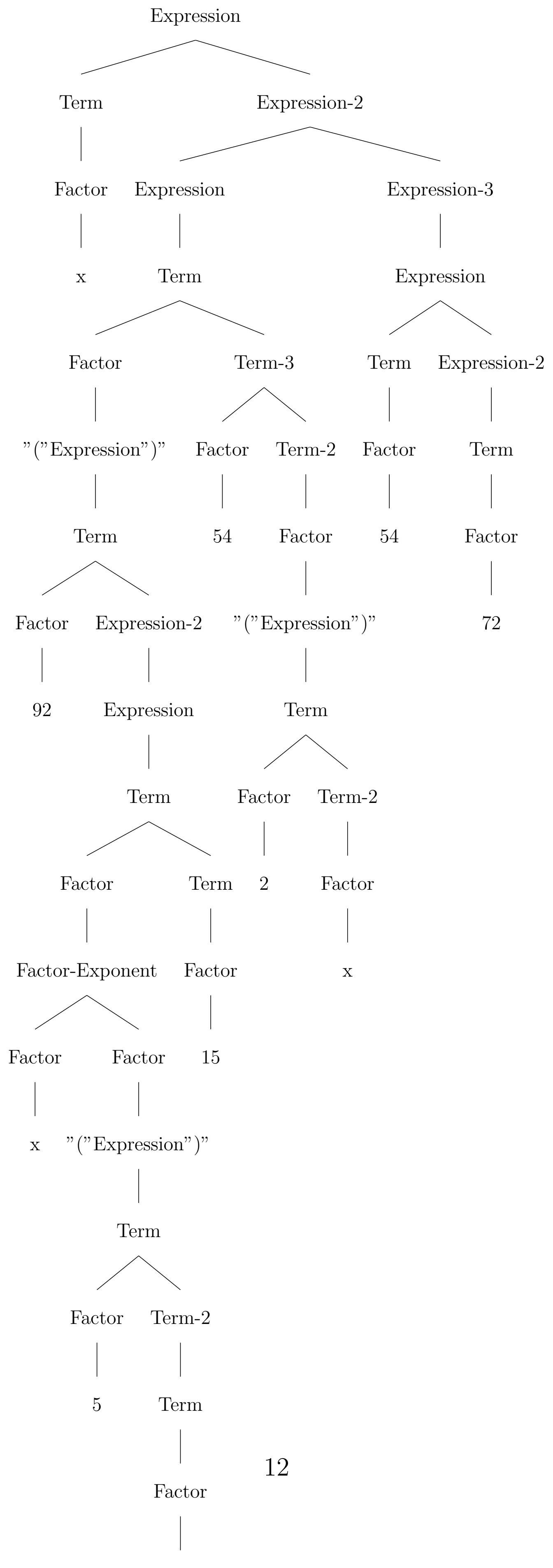
$$-x = -1 * x$$

$$-(x) = -1 * (x)$$

5.3 Recursive Descent Parsing

A recursive descent parser will take an expression and convert it into our BNF grammar. The recursive descent parser parses an expression from the left to the right by indexing through the characters of the expression. The parser begins parsing at the most outer-level non-terminal — in our case it starts as an Expression — and then recursively parses out the parts of grammar that make up that non-terminal. So, if parsing out an Expression part of grammar that included an Expression-2 node, representing that another Expression node is being added to the value within the Expression node's Term value, one would first recursively solve the Term, then the parser recursively parses the Expression-2 part of grammar. When a terminal expression is parsed—in our case when a constant or variable is parsed—the index of the current character of the expression is incremented. The parser also increments passed any operators since they are now represented with their own part of grammar.

The recursive descent parser takes on a tree structure when it is implemented because it creates a new branch every time it begins to parse a different part of grammar. All the terminal parts of grammar reside at the bottom of the tree and are referred to as the *leaves* of the tree. Using the recursive descent parser also helps preserve the order of operations, as the resulting parsed expression will be evaluated from the bottom up.


Expression
Term
Expression-2
Factor
Expression
Expression-3
x
Term
Expression
Factor
Term-3
Term
Expression-2
"("Expression")"
Factor
Term-2
Factor
Term
Term
54
Factor
54
Factor
Factor
Expression-2
"("Expression")"
72
92
Expression
Term
Term
Factor
Term-2
Factor
Term
2
Factor
Factor-Exponent
Factor
x
Factor
Factor
15
x
"("Expression")"
Term
Factor
Term-2
5
Term
Factor

5.4 The Benefits & Drawbacks of using BNF Grammar and the Recursive Descent Parser

The main drawback of this method is that there is no real way to parallelize it. Since the parser is sequential in nature by iterating through the expression one character at a time, one cannot parallelize this parser without locking, which would ultimately just make it a more complicated, sequential process. However, the one-time cost of building the parsed expression leads to large potential for concurrency when actually integrating the expression. Using the sum rule, an algorithm can be devised to create a new thread for every Expression-2 or Expression-3 part of speech that is not part of a parenthesized expression.

A huge educational benefit is gained using this type of parser. BNF grammar and recursive descent parsing is simple to trace through because they mainly involve transforming one part of grammar until its terminal form is reached, and the tree structure of the parsed expression lends itself well to how this is done recursively. The tree structure of the parsed expression also lends itself very well to explaining the concurrency of this application. The concurrent algorithm that will mainly be used in this application uses the fork-join framework that is also based on a tree structure. With this, it will be easy to draw the relationship between the concurrency of the integral-evaluator and the fork-join framework.

6 Differentiation

6.1 Addition & Subtraction Rule

This section will describe differentiation in the order in which it is performed in our tree representation of mathematical expressions. The differentiation algorithm starts at the outermost Expression node, and this is where the addition and subtraction rules of differentiation will be performed. As a reminder, the addition rule of differentiation says the derivative of a sum of functions is the sum of the derivatives of the functions (Robert Lopez and James Stewart, 2016, p. 176). Meanwhile, the subtraction rule of differentiation says the derivative of the difference of some number of functions is the difference of the derivatives of the functions (Robert Lopez and James Stewart, 2016, p. 176). The equation for these rules is shown below.

$$\frac{d}{dx}[f(x) \pm g(x)] = \frac{d}{dx}f(x) \pm \frac{d}{dx}g(x) \tag{1}$$

Let us look at an example of a parse tree of some given expression to describe how this rule of differentiation is performed by the program. Consider the expression $5 * x + x^2 - 7$. The parse tree for this expression would look like the following tree.

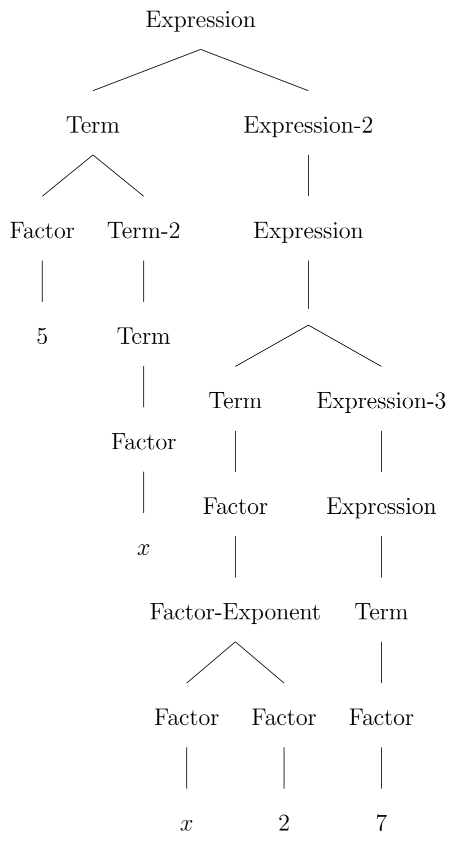
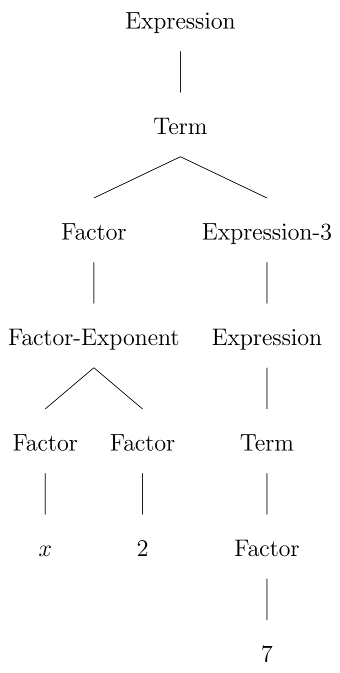


Figure 3: Parse Tree for the expression $5 * x + x^2 - 7$

Our algorithm starts at the top-layer expression node. Before differentiating the left child node, the left child Term node is first differentiated. Our algorithm will then check if the current Expression node has any right child Expression-2 or Expression-3 nodes. If the current Expression node does not have any right children, the algorithm returns the differentiation value that was returned from differentiating the node's left child Term node. If there are right children, then the algorithm will then call the differentiation function on the Expression node that resides in the current node's right child. After this differentiation value is returned, the left child node is connected with a "+" operator if the right child is an Expression-2 node and a "-" operator is the right child is an Expression-3 node. Let us walk through the application of these rules on the aforementioned tree for the expression $5 * x + x^2 - 7$.

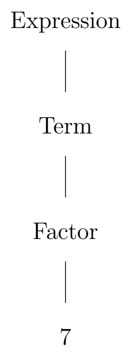
1. First, the algorithm differentiates the left child, which will return the value 5. After this, the algorithm then goes on to differentiate the Expression node residing in the Expression-2 node on the tree.

2. So, the tree structure is pictured in the following figure.



The process is now started again at the outermost Expression node pictured. First, the algorithm will differentiate the left child, which will return the value $2 * x$. After this, the differentiation algorithm is ran on the Expression node found in the Expression-3 right child node.

3. The tree structure of the expression the differentiation algorithm is being run on is shown in the following figure.



At this node, the algorithm just differentiates the child Term node and end with 0. Note that when the final derivative is simplified with the Wolfram engine, it will throw away this value.

4. This value of 0 is sent back to the tree shown in Step 2. At this tree, the algorithm then connects the left child derivative with the right child derivative to get a value of $2 * x + 0$.

5. The algorithm then takes this value of $2*x+0$ and returns it to the original expression in Step 1. The left child derivative is then connected with the right child derivative to get a value of $5 + 2 * x + 0$. This is the final derivative that is simplified with the Wolfram Engine, ending up with a resulting derivative of $5 + 2 * x$.

So, our algorithm for implementing the addition and subtraction rules of differentiation isn't that different from how people are taught to apply the rule in a Calculus 1 course. This is thanks to our representation of expressions using our grammar, where the algorithm can recognize Expression-2 and Expression-3 nodes as points where the addition and subtraction rules are implemented. However, things get a bit more complicated when running the differentiation algorithm at the T node level.

6.2 Differentiation Involving Constants

When differentiating constant values, the algorithm starts at the Term node the Constant node resides in. The example expression this section will look at will be 1, so the parse tree for this expression would look like the following figure.

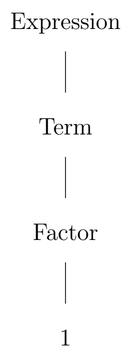


Figure 4: Parse Tree for the expression 1

6.2.1 Differentiating Lone Constants

When differentiating constants, the value is always 0. So, our differentiation algorithm checks if the Term node only contains a single constant value. If it does, integration is passed to the Factor node, which will simply return the value 0 since the current Factor node is of type constant.

Now, consider the case for when multiplying some other expression by a constant value. The expression $5 * x$ will be used as an example. The parse tree for the expression is the following figure.

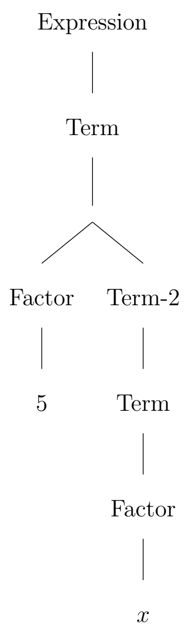


Figure 5: Parse Tree for the expression $5 * x$

The algorithm will first check the left Factor node to see if it is a constant value. If it is, then the algorithm checks the Term-2 child node to see if there is a Term node inside of it. The differentiation algorithm will be run on the Term node inside of the Term-2 node, which will just return 1. The algorithm then connects this returned value with the left child constant value by multiplying the constant value by the right child node's derivative, returning the expression $5 * 1$. This expression is then passed to the Wolfram engine to be simplified and the final derivative value is 5. Note that the algorithm only has to check for single constants on the left child node of Term nodes. This is because the Wolfram engine will simplify an expression such that any constants that are multiplied in together will be combined and then brought to the front of the expression. So, if one were to pass something like $5*x*7$ to the Wolfram engine, it would return $35*7$. When it comes to division involving constants, however, there are several cases one must consider.

6.2.2 Differentiating a Non-Constant Divided By a Constant

Consider dividing some non-constant value by a constant. This subsection will use $x/7$ as an example expression. The parse tree for this expression looks like the following figure.

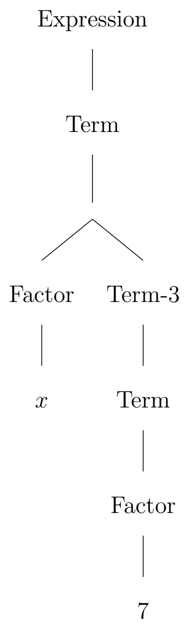


Figure 6: Parse Tree for the expression $x/7$

Our algorithm first checks the left child node and sees that it does not contain a constant value. When the left child node is not just a constant value, our algorithm checks the right child node to see if the left child is being divided by a constant. So, the algorithm checks if the right child is a Term-3 node and if it is, it checks if the Factor node residing in this Term-3 node is a constant. If this node is constant, the algorithm differentiates the left child node and divides the derivative value by the right-child constant value. In the example, the algorithm recognizes that the left child is not just a constant value, so it checks the right child node and sees that it is a Term-3 node. The algorithm then goes even further and checks if the Term-3 node contains a single constant value. In the given example, it does with the value of 7. So now the algorithm just differentiates the left child node and gets the value of 1 and divides it by the value of 7, resulting in the final derivative value of 1/7.

6.2.3 Differentiating a Constant Divided By a Non-Constant

This subsection will consider expressions where a constant is divided by some non-constant value. Consider $5/x$ as an example expression. The parse tree for this expression looks like the following figure.

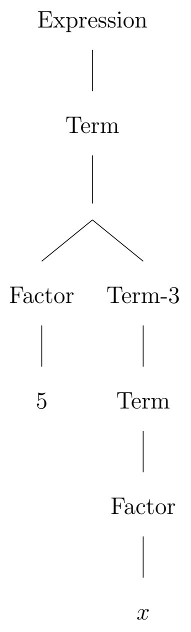
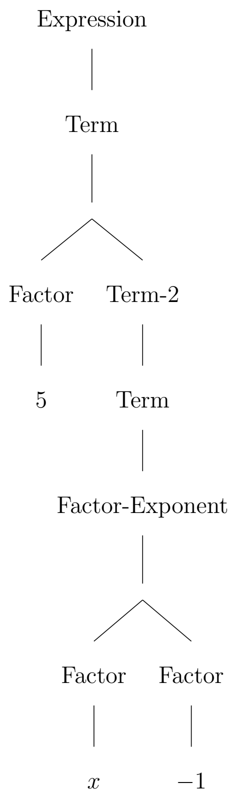


Figure 7: Parse Tree for the expression $5/x$

Our algorithm first checks the left child node and finds that it is a constant value. The algorithm then checks the right child node for if it is a Term-3 node representing division. If it is a Term-3 node, the algorithm then checks and makes sure the value inside the Term-3 node is not just a single constant. If the value is not just a single constant, the value in the the Term-3 node is modified such that it is an exponent raised to the power of -1 and the algorithm multiplies the resulting derivative by the left-child value of the original parent node. So, in our example, one will find that the left child node is a constant and the right child node represents division and does not just contain a single constant value. So, the right child node is modified such that the original expression takes the form of $5 * x^{-1}$. The tree for this modified expression looks like the following figure.



The differentiation algorithm is then on the Term node residing in the Term-2 node, returning a value of $-x^{-2}$. The algorithm then multiplies this derivative by the constant found in the left child, so the final derivative is $5 * -x^{-2}$ which will be rewritten as $-5/x^2$ when passed to the Wolfram engine for simplification.

6.2.4 Differentiating a Constant Divided By a Constant

The final case for differentiation involving constants is when one constant is being divided by another constant. Consider 5/7 as an example expression. The parse tree is pictured below.

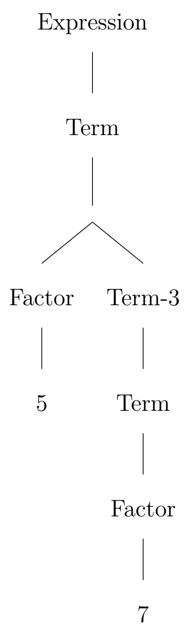


Figure 8: Parse Tree for the expression 5/7

One may already recognize this expression as merely a constant, but the program sees it as some factor divided by another factor, so it is not instantly recognized as a constant. The program can differentiate this by taking the procedure in subsection 6.2.3 and applying it to this case. We will ultimately just have $5 * 0$, and it saves us from having to write another rule specifically for an instance like this. It also scales well so that we can have expressions like $5/7 * x$, which is equivalent to $(5/7) * x$, where the tree looks like the following figure.

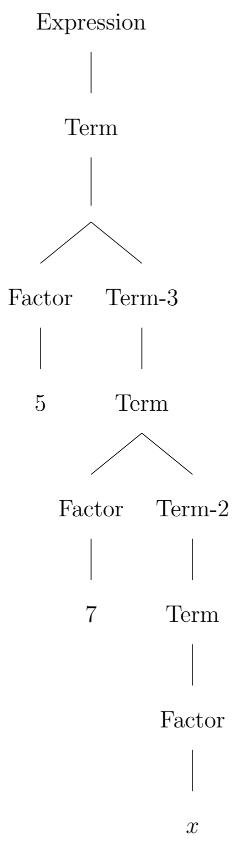


Figure 9: Parse Tree for the expression $(5/7) * x$

With a case like this, all we have to do is check the left Factor node found in the Term-3 node, use this value as a divisor of the constant found in the Factor node of the outmost Term node such that it becomes 5/7. We then get the derivative of the Term node residing in the Term-2 node and multiply that by our resulting fraction from earlier, 5/7 to get our final derivative, which in this case would just be 5/7.

6.3 Differentiating Lone Letter Variables

The next case of differentiation that starts at T nodes is when the T node contains only a single variable letter. Consider the single letter variable x as an example. The parse tree for this example expression is in the following figure.

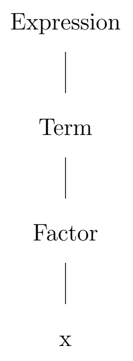


Figure 10: Parse Tree for the expression x

When differentiating lone letter variables, the algorithm treats it exactly as if someone were differentiating by hand and return a value of 1. Thus, the algorithm visits the Term node attached to the root Expression node and checks to see if there are any Term-2 or Term-3 nodes attached to the term node, as that would signify that there is not a lone constant. If there is no Term-2 or Term-3 node, the algorithm checks to see if the Factor node holds a letter variable, returning 1 in the given example.

6.4 Differentiating Parenthesized Expressions

The rules for differentiating parenthesized expressions are not much different than how we differentiate at the root Expression node. All we have to do when integrating parenthesized expressions is differentiate the expression inside the parentheses and then wrap the derivative that is returned in parentheses. As an example, consider the expression $(x-7)+x^2$. The parse tree for this expression is represented in the following figure.

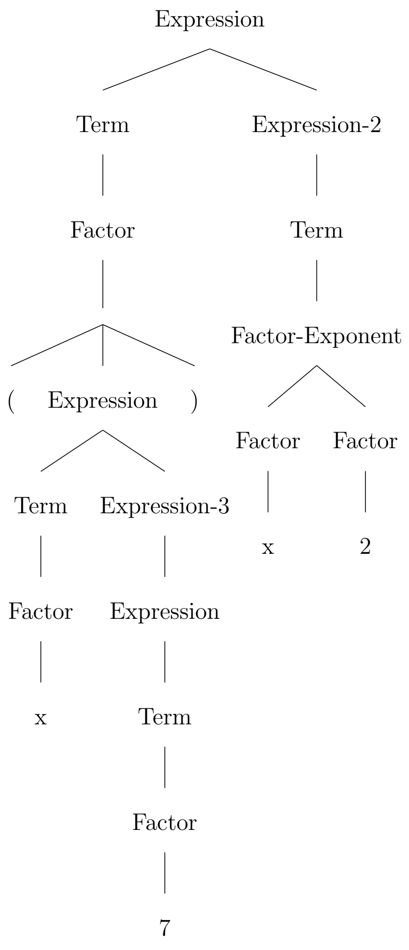


Figure 11: Parse Tree for the expression $(x-7)+x^2$

Our algorithm differentiates parenthesized expressions by first visiting the Term node and checking if the child Factor node holds a Parenthesized Expression Node. If it does, we then check if there is a child Term-2 or Term-3 node that represents multiplication or division. If no such node exists, we call the differentiation function on the parenthesized exponent that performs the process described in the preceding paragraph. So in our example function, we first visit the root Expression node's Term node whose tree representation is shown below.

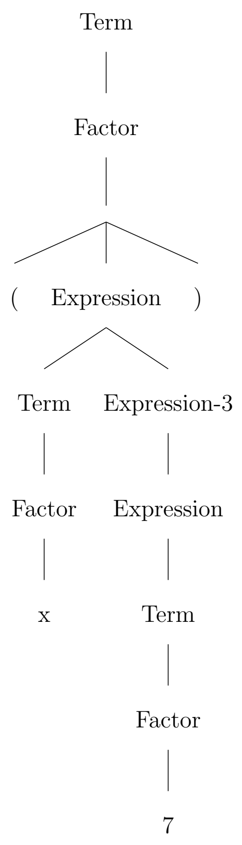


Figure 12: Parse Tree for the expression $(x - 7)$

We see that the root Term node has no Term-2 or Term-3 children and only consists of a child Factor node that contains a parenthesized expression. So, we call the differentiation function and perform the addition rule on the parenthesized expression as described in 6.1. This will return the value 1 and we will wrap it in parentheses so that the derivative at this Term node is (1). We then return the value of (1) to the Term node's parent Expression node and go on to differentiate the rest of the expression as we describe in 6.1, all to get the final derivative of (1), which will be simplified to 1 when passed to the Wolfram engine for simplification.

6.5 Differentiating Exponents

Before getting into more complex differentiation rules, we need to discuss how we differentiate exponential expressions. This subsection focuses on exponential expressions that aren't multiplied by some other non-constant expression.

6.5.1 Differentiating Variables Raised to a Constant Value

The first type of exponential expression we need to create a set of differentiation rule for is when we have a variable letter as the base and some constant value as the exponent - in other words, expressions with a power. We will use the expression x^2 as an example. The parse tree for the expression is in the following figure.

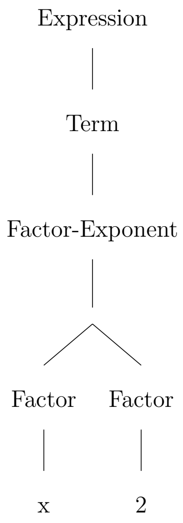


Figure 13: Parse Tree for the expression x^2

Our algorithm for differentiating exponents starts at the Term node. We check for a Factor-Exponent child node at this level and so long as there is not a Term-2 or Term-3 node containing a non-constant Factor node attached to this Term node, the differentiation procedure is passed onto the Factor-Exponent node and this is where the power rule of differentiation is performed. Recall the power rule of differentiation below.

$$\frac{d}{dx}x^n = nx^{n-1} \tag{2}$$

We have to first make sure our Factor-Exponent is of the form x^n. So, our algorithm first checks to make sure the left child in the Factor-Exponent node is a single variable and that the right child is a single constant value. If these conditions are met, then we simply return a string representation of the derivative. The derivative of this type of expression takes the following form.

$$\frac{d}{dx} < Left - Child - Variable >^{<Right-Child-Constant>}$$
$$= < Right - Child - Constant > * < Left - Child - Variable >^{<Right-Child-Constant>-1}$$

In our example expression, we would recognize the expression as a variable raised to some constant and would return the value $2 * x^1$ as the derivative of the expression.

6.5.2 Differentiating Constants Raised to a Variable Value

Our process for differentiating single constant values raised to a variable value is not too different than differentiating single variable values. The expressions include the class of exponential functions $y = b^x$ where $b > 0$ and $b \neq 1$. We will use the expression 2^x as an example. The parse tree for this expression is shown below.

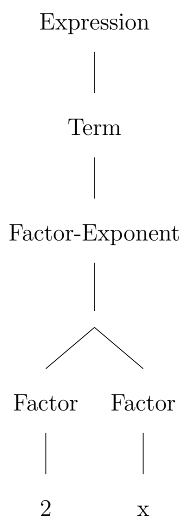


Figure 14: Parse Tree for the expression x^2

Recall the rule for differentiating constants raised to a variable value shown in the following equation.

$$\frac{d}{dx}a^x = ln(a)a^x \tag{3}$$

Our algorithm first checks the left child node of the current Factor-Exponent node to ensure that it contains a lone constant value and then checks the right child node and and ensures that it only contains a lone variable value. The application then returns a string representation of the derivative. The derivative for this expression then takes the following form.

$$\frac{d}{dx} < Left - Child - Constant >^{<Right-Child-Variable>}$$
$$= ln(< Right - Child - Variable >)* < Left - Child - Constant >^{<Right-ChildVariable>}$$

So, in our example expression, we would return the derivative $ln(2) * 2^x$.

6.6 Differentiating Using the Product Rule & Modifying the Quotient Rule

In order to generically apply the product rule where an expression has an arbitrary number of multiplicands, one must view the product rule as a recursive problem. The product rule of differentiation is represented with the following equation:

$$\frac{d}{dx}f(x)g(x) = f(x)g'(x) + f'(x)g(x) \tag{4}$$

In order to generically perform the product rule, $g(x)$ must represent all following multiplicands after the multiplier in an expression. So, for example, in the expression $(x+1)(x+2)(x+3)$, $f(x) = (x+1)$ and $g(x) = (x+2)(x+3)$. With this, the product rule algorithm will be recursively called until $g(x)$ is a single term. In the differentiation algorithm, the expression is first checked to see if there are at least two terms that are not solely composed of constants being multiplied or divided together.

The next case that needs to be considered is when one term is used to divide another where neither term is solely composed of constants. One *could* just incorporate the quotient rule of differentiation into the application, but doing so will require additional logic. For example, in expressions that contain both multiplication and division of terms, a lot of pattern matching will need to be done where the application switches between the product and quotient rule. Overall, it will require more overhead to plainly incorporate the quotient rule. Instead of doing this, the application makes modifications to a term while performing the product rule so that a quotient rule algorithm is unneeded. Take $(x+1)/(x+2)$ as an example. One could turn any term that uses division into one that solely uses multiplication by inverting the denominator and multiplying it with the numerator. Thus, the expression becomes $(x+1)*(x+2)^{-1}$. This is precisely what the application does instead of incorporating the quotient rule.

To incorporate this operation, the product rule algorithm pattern matches for denominators that are not composed solely of constant values. This is done by looking at the Term-Extension nodes and checking if they hold a Term-3 Node. If the Factor Node of that Term-3 Node is not composed solely of constants, then the Factor Node is modified such that it becomes a Factor-Exponent Node where the base is the original value of the Factor node and the exponent value is -1. After this Factor node has been modified, the Term-3 Node is changed into a Term-2 node and the product rule algorithm is performed as normal after this modification.

6.7 Differentiating Composite Functions

Given that the application does not support trigonometric functions currently, the only composite functions that are possible involve powers and natural logarithms, the latter of which is discussed in the following subsection. The possible composite functions are:

1. Constant values raised to an expression that isn't a single variable (e.g. 5^{x+1})
2. Expression values raised to a constant value (e.g. $(x+1)^2$)
3. Expression values raised to another expression (e.g. $(x+1)^{x+2}$)

The algorithm used to perform these derivatives is based on the chain rule, which is presented in the following equation.

$$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x) \tag{5}$$

One could write an algorithm that performs the chain rule shown in Equation 5, but this would require first identifying what $f(x)$ would be in the expression, what $g(x)$ would be in the expression, forming a new expression to act as $f(x)$, and then finally finding the derivatives. Instead of doing this, one could instead pattern match for the composite functions mentioned previously, which would already have to be done if one were to perform the chain rule, and then return the result that the chain rule would have given us.

Consider the first composite function involving a constant raised to some expression and the expression $c^{b(x)}$ where c is some constant and $b(x)$ is some expression that does not only consist of a single constant. Using the chain rule, $f(x) = c^x$ and $g(x) = b(x)$. The derivative of $f(x)$ would use the rule described in Section 6.5.2, resulting in $f'(x) = ln(c)c^x$. Since the composition of $b(x)$ is unknown, $g'(x) = b'(x)$. Using these derivative values, the chain rule would return $ln(c)c^{b(x)}b'(x)$. There is now a general form for this type of composite function and the rule is applies when the pattern matching algorithm recognizes the given composite form.

Next, consider the composite function where an expression is raised to a constant value and the expression $b(x)^c$ where c is some constant and $b(x)$ is some function that does not only consist of a single constant. When using the chain rule, $f(x) = x^c$ and $g(x) = b(x)$. $f'(x) = cx^{c-1}$ and $g'(x) = b'(x)$. Thus, the final derivative using the chain rule is $cb(x)^{c-1}b'(x)$. This rule is then applied when the composite form is recognized by the pattern matching algorithm.

The final composite function to consider is when expressions that are not solely constant values compose both the base and power of an exponent. Take the expression $b(x)^{c(x)}$ where $b(x)$ and $c(x)$ are both functions that are not solely composed of constants. In its current form, the expression cannot be differentiated. However, we know that expressions of the form $x^y = e^{ln(x)y}$. Thus, the expression is rewritten as $e^{ln(b(x))c(x))}$. This new form allows the chain rule where $f(x) = e^x$ and $g(x) = ln(b(x))c(x)$. The derivative of e^x is e^x itself, so the final derivative of the expression is written as $e^{ln(b(x))c(x))}(ln(b(x))c(x)))'$. Note that $e^{ln(b(x))c(x))}$ can be rewritten as $e^{ln(b(x))^{c(x)}}$. $e^{ln(b(x))} = ln(b(x))$, so the final derivative is written as $ln(b(x))^{c(x)}(ln(b(x))c(x)))'$, where $e^{ln(b(x))c(x))}$ is substituted with $ln(b(x))^{c(x)}$ using the described equivalences. This rule is then used in the differentiation algorithm when the composite form is recognized by the pattern matching algorithm.

6.8 Differentiating the Natural Logarithm

Differentiation for the natural logarithm uses the chain rule. Consider any natural logarithm to be of the form $ln(c(x))$ where $c(x)$ is the inner-composition of the natural logarithm. Note that $c(x)$ would be an EP node. It can then be said that $f(x) = ln(x)$ and $g(x) = c(x)$. The differentiation algorithm is then performed on $c(x)$. Using the chain rule, a composition is made using the grammar previously outlined that is of the form $\frac{c(x)}{c'(x)}$.

7 Integration

7.1 Addition & Subtraction Rule

The implementation of the addition and subtraction rules is what allows our integral evaluation algorithm to run in parallel using Java's fork-join framework. The addition rule states that the integral of a sum of functions is the sum of the integrals of the functions. Similarly, the subtraction rule says that the integral of the difference of some number of functions is the difference of the integrals of the functions. These rules are summarized in mathematical notation in Equation 1.

$$\int [f(x) \pm g(x)]dx = \int f(x)dx \pm \int g(x)dx \tag{6}$$

These rules tell us that the integrals of the separate functions are not related to each other and only need to be summed together after each integral is taken, so we can write our integration algorithm in parallel. As the name implies, the fork-join framework has two basic operations: forking and joining. *Forking* is the process where a job is generated to be performed in parallel and *joining* is executed after a forked task is finished. The fork-join framework operates on the idea of a work queue where any forked tasks get put onto this queue and each working thread on the given computer's CPU takes jobs out of this queue to work on Douglas Schmidt (2022). When a fork is joined, the worker thread that finished the forked task goes back and takes another task of the work queue or aids the other worker threads performing forked tasks if there are no more tasks. Using this framework is especially useful since it is more lightweight than the conventional way to make threads that normally require a lot of extra computations to maintain Douglas Schmidt (2022). With the fork-join framework, a lightweight version of threads are created to perform the tasks that don't require so much overhead memory and task management that tends to slow computation time down if the amount of operation management computations required to maintain the threads exceeds the amount of computations each thread is actually doing Douglas Schmidt (2022).

The evaluation algorithm makes use of the fork-join framework at the Expression grammar nodes if the expression grammar nodes contain an Expression-2 or Expression-3 node grammar node, the Expression node that exists within the Expression-2 or Expression-3 grammar node is taken and the task of integrating this acquired node is forked. This process is done recursively until no Expression-2 or Expression-3 nodes are found. The Term node in the parent Expression is then integrated and the integration task that was forked is joined. The integration results from the Term node and the Expression-2 or Expression-3 node are then connected with either a $+$ or $-$ depending on if the latter node is an Expression-2 or Expression-3 node.

To demonstrate how the fork-join framework is used for the addition and subtraction rules, let us use the expression $2 * x + x + x^2$ as an example. Below is the parse tree for the expression that we will reference.

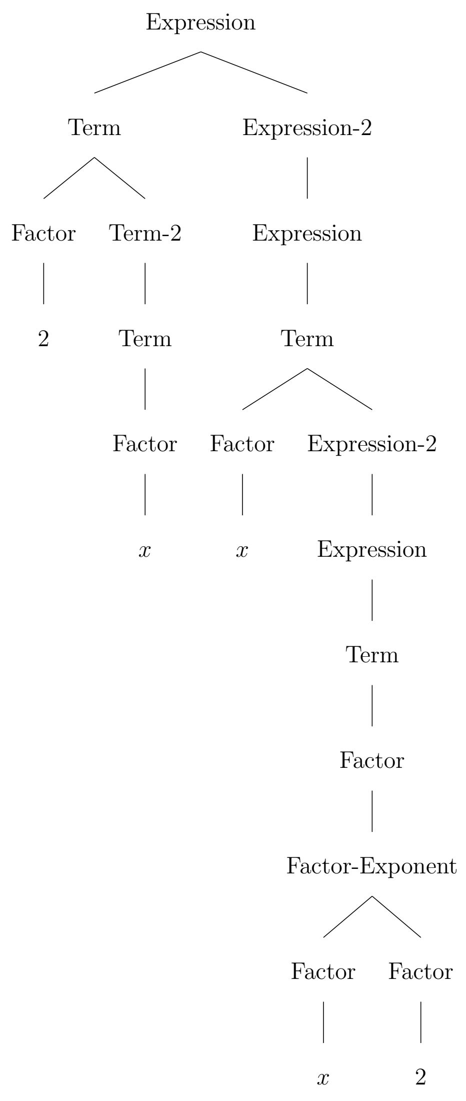


Figure 15: The parse tree representing the expression $2 * x + x + x^2$.

Our algorithm starts off at the root Expression node. At this node, we check if there exists an Expression-2 or Expression-3 right child node. If this right child node exists, we send the Term left child node to be computed. We then use the fork-join framework to visit the right child node's Expression child node by *forking* the right child node such that a separate thread evaluates the integral for it. This process then restarts at the forked child node and continues recursively until no Expression-2 or Expression-3 children are found at the visited Expression nodes.

7.2 Integrating Constants & Variables Using the Power Rule

The formula for the power rule of integration is shown below.

$$\int x^n dx = \frac{x^{x+1}}{n+1} + C \tag{7}$$

Using our grammar, this rule of integration becomes trivial to perform. When integrating constants and variables, all calculations are done using the values within the Term nodes. First, we'll discuss integrating constants since it is the simplest computation within our system. The evaluation algorithm examines a given Term node and assures there is no Term-2 or Term-3 right child node attached to it. If there is no such child node, the evaluation algorithm examines the left child Factor node. If the Factor node is of type Constant where it is just a number, the integration evaluation algorithm simply right-multiplies this constant value by the letter variable used in the originally given expression and the constant is then integrated.

When dealing with the integration of variables, things become more interesting. If the Term node again has no Term-2 or Term-3 right child nodes, yet a left child Factor node of a variable type, then the given expression is just a lone variable, for example x. If this is the case, the evaluation algorithm left multiplies the given variable by $1/2$ and turns the Factor node holding the variable into an F-Exponent node where the left child of this F-Exponent node is the given variable and the right child node is a constant value of 2. Now, if the variable is not the lone value within the Term node and is multiplied or divided by some constant — for example, say that the Term node represents the value $5 * x$ — then the constant that multiplies or divides the variable is simply multiplied by $1/2$, the variable values are still converted into an F-Expression node with an exponent value of 2.

Dealing with integrating variables raised to a power other than one — for example, x^2 — works on a similar principle in that it is all based around node type checking. First, the evaluation algorithm checks if there is a child node that is of type F-Exponent where the left child is a variable letter and the right child is a constant value. If the F-Exponent node is not being multiplied by any constant value - meaning that the given Term node has no Term-2 or Term-3 child node - then the F-Exponent node's right child is increased by one and the F-Exponent node is left-multiplied by $1/y$, where y is the value of the exponent when increased by 1 and this new expression is returned as the integral. If this F-Exponent value is multiplied by a constant value, then the exponent value is still increased by one, but the constant value is multiplied by $1/y$, where y is the value of the exponent when increased by 1 and this new expression is returned as the integral.

7.3 Integrating Compositions Using the Substitution Rule

7.3.1 The Substitution Rule Briefly Explained

It is important to understand how the substitution rule works to properly understand this section, so we shall briefly examine how u-substitution works. First, we have to look at the chain rule of differentiation.

$$\frac{d}{dx}[F(g(x))] = F'(g(x))g'(x) \tag{8}$$

1: The chain rule of differentiation

As we can see, the chain rule creates an expression that cannot be easily integrated with the integration rules we've previously described. The proven strategy for this type of problem is to substitute the inner part of the composition, $g(x)$ with some variable letter, that letter often being u (Robert Lopez and James Stewart, 2016, p. 401). Using this, we can transform the integral of the expression such that it is easily integrated with the following steps.

- When taking the integral

$$\int F'(g(x))g'(x)dx$$

we first substitute the inner-function of the composition $F'(g(x))$ with some variable value, we'll call ours u. Our integral then becomes

$$\int F'(u)g'(x)dx.$$

- Note that when integrating, we only want the expression to have one variable letter, so we will put this integral in terms of u. We can note that the differential of u is $g'(x)dx$ since u is a placeholder variable for $g(x)$ so,

$$du = g'(x)dx,$$

which can be rewritten as

$$dx = \frac{du}{g'(x)}.$$

- We will now substitute dx in for this equivalent value such that

$$\int F'(u)g'(x)\frac{du}{g'(x)}.$$

Note that the $g'(x)$ terms cancel out, so we now have

$$\int F'(u)du.$$

The integral of this is then easily calculated, assuming one knows the integral of $F'(x)$,

$$\int F'(u)du = F(u) + c.$$

Using these steps, we can write the substitution rule for integration as the following equation.

$$\int F'(g(x))g'(x)dx = \int F'(u)du = F(u) + c$$
$$\text{where } u = g'(x) \text{ and } du = g'(x)dx \tag{9}$$

We will find that the implementation of this rule is rather intuitive, though a lot of work, to perform the process algorithmically thanks to the grammar developed for the application.

7.3.2 Identifying the Substitution Choices

The way the algorithm for finding substitution values works is by first attempting to use the power rule for integration on Term nodes. When this fails, we know that a more complex integration rule is needed so the u-substitution algorithm launches and starts finding possible u-values. We *could* simply just take all possible combinations of the Term nodes and the Term-2 and Term-3 nodes that are attached to it, but this is highly inefficient, especially for complicated u-substitution problems with large expressions. Instead, we will leverage the properties of expressions that require the substitution rule. We've established in the previous subsection that the likely u-values will most likely be the inner-function of a given composition $F(g(x))$ found in Formula 2. Because of this, the algorithm searches for inner-functions of a composition found within the expression to be integrated. This is done through recursion and pattern matching techniques, of which the latter operation is made to be intuitive thanks to Scala's superior pattern matching capabilities when compared to other languages.

The first composition that is searched for are compositions that are of the form $(f(x))^n g(x)$, such as $x^2(x^3 - 7)^3$. With this, our u-value searching algorithm first examines the given Term node and any of Term-2 or Term-3 nodes that are attached to it. Inside the Term node resides a Factor node. The algorithm visits the Factor node and type checks for an F-Exponent node that resides in the Factor node. If there isn't an F-Exponent node, this part of the algorithm stops searching for this type of composition. Otherwise, the algorithm visits the base and exponent components of this F-Exponent node and searches for a parenthesized E node, since this is where the inner function of a composition would be. If this parenthesized E node doesn't contain just a simple constant or variable, then we know this is most likely the inner-function of a composition and we go forward and later on attempt to perform the rest of the u-substitution algorithm with this value. So, for example, we would take the expression $x^2(x^3-7)^3$ and find that the Term-2 node $(x^3-7)^3$ has a likely inner-composition component of (x^3-7) since the base value is a parenthesized E node.

The next type of composition that is searched for are of the form $f'(g(x))g'(x)$, such as $(4x^2-4)2x$. This is a basic composition since there is no exponent attached to it. Note that we need to differentiate how our algorithm processes possible substitution values with exponents and basic compositions since the compositions will reside in different Factor-Exp nodes and parenthesized expression nodes, respectively—so our algorithm has to recognize two different grammar patterns for substitution. Our algorithm will again visit the outermost Term node of a given expression and all the Term-2 and Term-3 nodes that may be attached to it. The algorithm then visits the Factor node that exists with these nodes to search for potential inner-functions of a composition. These inner-function will again be a parenthesized E node that is not merely composed of a simple constant or variable letter. So, for example, the algorithm would take the expression $(4x^2-4)2x$ and find that the Term node $(4x^2-4)$ is most likely the inner-function of a composition. Note that there are cases where multiple possible inner-functions are found. For example, the expression $(x^2+2x+4)^3(2x+2)$ would find that both (x^2+2x+4) and $(2x+2)$ are possible u-values since they both follow the requirements we have laid out for our algorithm.

It is important to note the recursive steps our algorithm takes, as well. The correct substitution value for an integration problem using the substitution rule is not always the one that is in the outmost function. For example, the substitution value for the expression $4xe^{x^2+e^{x^2}}$ is the x^2 that serves an an exponent for e^{x^2}. To incorporate this in our algorithm, we recursively run the substitution value finding algorithm on any possible substitution values that are found. It will attempt to find any further substitution values and attempt the rest of the substitution rule algorithm with these further values and if this leads to an integration failure, the parent substitution value is then used to perform the substitution rule. So, for example, with $4xe^{x^2+e^{x^2}}$, the algorithm finds that $x^2 + e^{x^2}$ is a potential substitution value. We then run the substitution value finding algorithm on this potential substitution value and find that x^2 is another potential substitution value. The program will then attempt to perform the substitution rule with x^2 and if it finds that integration is unsuccessful with this value, though we know it would not be, the u-substitution is then performed with $x^2 + e^{x^2}$ as the substitution value. This is especially useful for cases that will require multiple substitution steps, since these cases will have a substitution value that's found in *another* substitution value. Our substitution rule algorithm will first attempt to find the integral of a given expression using the inner-most u-value. The algorithm will perform the substitution rule with this inner-most u-value and then attempt to recursively integrate the expression with the substituted value. This recursive process allows for us to perform the substitution rule for expressions that recursively use the substitution rule, like $4xe^{x^2+e^{x^2}}$. The recursion does not stop until the algorithm successfully integrates the expression with the given substitution value or it fails integrating the expression with the substitution value. If the algorithm fails with the current substitution value, the substitution rule algorithm is performed again with the current substitution value's parent substitution value. So, for example, with the expression $4xe^{x^2+e^{x^2}}$, we will try the substitution rule with x^2 but if for some reason integration with this substitution value fails, we will then perform the substitution rule algorithm with $x^2 + e^{x^2}$.

7.3.3 Bringing It All Together

Now that all the possible substitution values are found, the final algorithm that performs the substitution rule needs to be developed. First, we start out with a hashmap that stores the possible substitution values we found with the algorithm described in the previous section such that the keys are integers and the values are lists of substitution values. The keys of the hashmap are integers representing the likelihood that the respective substitution values will lead to an integration result. The more nested the substitution value is, the higher its likelihood value is. So, for example, the expression $4 * x * e^{x+x^{x^2}}$ would create the following hashmap:

$$0 \longrightarrow x^2 + e^{x^2}$$
$$1 \longrightarrow x^2$$

We go through the possible substitution values until the integral is solved or substitution has been attempted with all possible substitution values and no integration value has been found. If the latter case happens, the integration value is set to null to indicate the substitution rule could not be applied to the given expression.

When starting the substitution rule, the algorithm starts with the substitution values found at the highest likelihood level. The steps for performing the substitution rule with this value are as follows:

1. **Preparation**

   This step substitutes the current substitution value into the given expression with either the letter u or v. The letter that is chosen alternates in case the expression requires multiple substitution steps. This also makes u and v protected variables, so the user cannot use them in their expression, as they are reserved for this part of the program. This done using a regular expression that is equivalent to the string representation of the current substitution value and replacing all instances of the substitution value in the expression with our substituted letter variable.

   After this, we then take the expression that now includes the substituted letter variable and multiply it by 1 divided by the derivative of the substitution value, which we find using our differentiation algorithm. This expression is then simplified using the Wolfram engine. To make sure the simplification did not result in an expression that still contains the substitution value, we substitute any instances of the substitution value with our letter variable again. If all the variables in the resulting expression are only u or only v, depending on what the substituted letter is, then this step has been successful and we move onto the integration step. If the expression has multiple different letter variables, then we move onto the next likely substitution value if there is one, otherwise null is returned for the integration value to indicate the substitution rule could not be applied.

2. **Integration**

   This step runs the integration algorithm on the expression that results from the preparation step. If we run the integration algorithm on the expression and it returns a null value, then we know that the current substitution value has failed. If there exists a next likely substitution value in the hashmap, we start the substitution algorithm over again with this value and return null otherwise. If the integration algorithm does not return a null value, we replace every instance of u or only v depending on what the substituted letter is with the substitution value and send the result to the Presentation step.

3. **Presentation**

   In this step, the final integration value is simplified with the Wolfram engine and returned.

As an example in the substitution rule algorithm's process of integrating, consider the expression $4 * x * e^{x^2+e^{x^2}}$ that a hashmap was generated for in the previous figure.

1. First, the program attempts to substitute the value x^2 with the letter variable u. The result is $4 * x * e^{u+e^u}$

2. Next, the program multiplies this resulting value by $\frac{1}{2*x}$ and simplify the result using the Wolfram engine. The resulting expression is $2 * e^{u+u^2}$

3. Next, the program runs the integration algorithm on $2 * e^{u+u^2}$. Our integration algorithm will find that the substitution rule is needed again. The hashmap for this implementation of the substitution rule is as follows.

$$0 \longrightarrow u + e^u$$
$$1 \longrightarrow e^u$$

4. The program now attempts to substitute the value e^u with the letter v, the result is $2 * e^{u+v}$.

5. The program then multiplies $2 * e^{u+v}$ by $\frac{1}{e^u}$ and simplifies the result with the Wolfram engine. The result is $2 * e^v$.

6. Now the program runs the integration algorithm on $2 * e^v$, this will return $2 * e^v$.

7. The program now replaces instances of v with $2 * e^u$, resulting in $2 * e^{e^u}$. The program will attempt to simplify the expression with the Wolfram engine, but it will result in the same expression.

8. $2 * e^{e^u}$ is then returned to the previous integration function call and all instances of u are replaced with x^2, resulting in $2 * e^{e^{x^2}}$

9. The Wolfram engine will attempt to simplify this expression but it will not be able to, leaving the final integration value of $2 * e^{e^{x^2}} + C$.

7.4 Integrating the Natural Logarithm

Integrating a natural logarithm function that is not composed solely of constant values requires integration by parts. One could hard-code the integration value of a the function $ln(x)$, but incorporating the integration rule with the natural logarithm makes future work on the application easier when integration by parts functionality will have to be added for more complicated integrals. The integration by parts rule can be represented with the following equation:

$$\int f(x)g'(x)dx = f(x)g(x) - \int g(x)f'(x)dx \tag{10}$$

When integrating a natural logarithm, $f(x)$ is equal to the natural log itself and $g'(x)$ is 1. From here, the integration algorithm runs the differentiation algorithm on the natural logarithm and creates a composition where the derivative of the natural log is multiplied by x. This composition is then simplified with the Wolfram engine and then integrated using our integration algorithm. If the integration result returned is null, then integration has failed and the integral of the natural log is set to null to reflect the failure. If the integration value does not contain null, then the integration result is used to return the integration value shown in Equation 6.

It is also important to account for natural logarithms that require the substitution rule to perform. For example, $ln(x+1)$ and $ln(x^2)$ both have compositions that could use the substitution rule but using the substitution rule on $ln(x^2)$ would not yield a non-null result. Because of this, the the main integration algorithm will first attempt the substitution rule algorithm in these instances, but if the substitution rule algorithm yields a null result and the given composition contains a natural logarithm that is not multiplied by any non-constant values, the integration by parts algorithm will be performed in an attempt to find the integral of the given composition.

8 Further Work & Conclusion

Though this application is able to differentiate and integrate complex expressions, there is still a lot of functionality to be added that had to be cut for this thesis. Trigonometric differentiation and integration needs to be added, but there are some caveats to consider. One will first have to create new grammar rules for the parser to implement all the trigonometric functions, implement the rules into the parser, and only then start developing the differentiation and integration algorithms. Differentiating trigonometric algorithms ought to be a simple addition to the application, but trigonometric integration requires specific simplification rules much of the time that the Wolfram Engine is currently not capable of performing reliably. Along with this, the substitution algorithm can currently handle some very complex expressions, but it falters when one would first have to perform a specific algebraic simplification on the expression before performing the substitution algorithm, which, again, the Wolfram Engine is unable to do reliably. It is also worth mentioning that the map used for the substitution rule is really emulating the behavior of a descending priority queue and, in hindsight, a priority queue should replace the map used for this rule on a future iteration of the application.

The other feature that is yet to be implemented is the integration-by-parts integration rule. A possible workflow for this rule may be to first attempt the substitution rule and upon failure of the rule, perform integration by parts with the terms of a given expression. This process, however, also currently runs into the potential problem of needing algebraic simplifications that the Wolfram Engine does not reliably perform.

All of this is to say that moving forward, one would need to either develop their own computer algebra system that considers the simplifications that the Wolfram Engine does not, or build upon the Wolfram Engine that is currently in use to handle the simplifications that the Wolfram Engine misses, or find another computer algebra system that is more well-suited to the application being developed. The second of these options is the one that is the most feasible since the other two options require starting from scratch with the given computer algebra system.

It is also possible to parallelize some of the given integration and differentiation algorithms, but it may not be totally necessary to do so and benchmarks would need to be run to diagnose which parts of the application are the slowest and if their functionality is even possible to run in parallel.

This problem proves to be a multi-faceted one. What one might initially think is primarily a calculus problem, turns into a parsing, computational calculus, and computational algebra problem. In order to progress, it is necessary to go back to the computational algebra system used for this application in order to support algebraic simplifications that will be needed to further improve upon the integration algorithms. Regardless of the work to be done, though, the current state of the application is one that is unlike a lot of the available software performing the same tasks because everything is explained and the code is written with understand-ability to the layman in mind. The code currently written is also easy to build upon so that the previously listed functionality can be implemented with relative ease. This application proves to be a solid foundation to build upon and create a fully functional, understandable, and open source symbolic differentiation and integration application

Bibliography

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley. Date Accessed: 10/12/22.

Bronstein, M. (2004). *Symbolic Integration I: Transcendental Functions (Algorithms and Computation in Mathematics)*. Springer-Verlag, Berlin, Heidelberg. Date Accessed: 09/15/22.

Douglas Schmidt (2022). Overview of the Java Fork-Join Framework. `https://www.youtube.com/watch?v=1i79YE2N0w8`. Date Accessed: 08/22/22.

Krommer, A. R. and Ueberhuber, C. W. (1998). *Computational Integration*. Society for Industrial and Applied Mathematics. Date Accessed: 05/4/22.

M. L. Scott (2016). *Programming Language Pragmatics*. Morgan Kaufmann, Elsevier. Date Accessed: 06/13/22.

Robert Lopez and James Stewart (2016). *Single Variable Calculus*. Morgan Kaufmann, Elsevier. Date Accessed: 08/26/22.

9 Appendices

9.1 Pseudo-code

This application is thousands of lines long and it would be inefficient to simply paste the code I've written into this paper. What follows is pseudocode for the more important algorithms in the application that handle complex tasks like the substitution rule and choosing what rule to use for differentiation. For a more in-depth look at the application, the source code is available at `https://github.com/daltamur/SIDC` and is being regularly updated. Note that this pseudocode is not meant to be a complete replacement for documenting the entire codebase, but rather as an aid for understanding the flow of the general functionality.

```
#This is Performed on E nodes
#This same algorithm is performed on EP nodes, the final integration/
                                                     differentiation value is just
                                                     parenthesized
def AddSubRule():
    if integrating:
        l.integrate
        integrationVal = l.integrationVal
    else:
        l.differentiate
        differentiationVal = l.differentiationVal

    if r.isNotNone:
        if integrating:
            r.integrate
            if r.isE2:
                integrationVal = integrationVal + '+' + r.integrationVal
            else:
                integrationVal = integrationVal + '-' + r.integrationVal
        else:
            r.differentiate
            if r.isE2:
                differentiationVal = differentiationVal + '-' + r.
                                                         differentiationVal
            else:
                differentiationVal = differentiationVal + '-' + r.
                                                         differentiationVal
```

```
#This is performed within T nodes
def IntegrateTerm():
    if(this.isComposedOfConstants):
        integrationVal = "(" + this.getValue + ")" + getVarLetter()
    else if checkIfPossibleSubstitutionRule():
        SubstitutionRule()
    else:
        ElementaryIntegration()
```

```
#This is performed within T nodes
def RunElementaryIntegration():
    if r.isNotNone:
        if l.isConst:
            r.l.IntegrateTerm
            integrationVal = l + r.getOperator + r.l.integrationVal
        else if l.isVar:
            if r.l.l.isConst:
                if r.operation == '*':
                    exponentRule()
                else if r.operation == '/':
                    l.IntegrateFactor()
                    integrationVal = l.integrationVal + "/" + r.l.getValue
                                                         ()
        else if l.isExponent:
            if l.isNotAllConstants:
                if r.l.isComposedOfConstants && r.operation == '/':
                    l.IntegrateFactor()
                    integrationVal = "(" + l.getIntegrationVal() + ")/(" +
                                                          r.getValue() + "
                                                         )"

        else if l.isNaturalLog:
            if !l.innerFunction.isAllConstants:
                if(r.l.isComposedOfConstants && r.operation = '/'):
                    l.IntegrateFactor()
                    integrationVal = "(" + l.getIntegrationVal() + ")/(" +
                                                          r.l.getValue() +
                                                          ")"

        else if l.isEP:
            if l.isCompostedOfConstants:
                r.l.IntegrateTerm()
                integrationVal = l.getValue() + r.operation + r.l.
                                                     integrationVal
```

```
    #This is performed within T nodes
def differentiate():
    #checks if there are multiple multiplication/division parts of the T
                                        node involving non-constants
    if(!this.checkMoreProducts(1)):
        if l.isConst:
            leadingConstantNonProductRule()
        else if l.isFExp:
            nonProductRuleFExp()
        else if l.isVar:
            nonProductRuleVar()
        else if l.isEP:
            if r.isNotNone:
               if r.operation == '*':
                    l.differentiateFactor()
                    r.l.differentiate()
                    differentiationVal = l.getValue() + "*" + r.l.
                                                         getDifferentiationVal
                                                          + "+" + l.
                                                         getDifferentiationVal
                                                          + "*" +l.
```

```
                                                              getValue()
                else:
                     if r.l.l.isConst:
                         l.differentiateFactor()
                         differentiationVal = l.getDifferentiationVal + "*1
                                                                      /" + r.l.l.
                                                                      getValue()
                     else:

             else:
                 l.differentiateFactor()
                 differentiationVal = l.getDifferentiationVal()
                 generalProductRule()
    else:
        l.differentiate()
        differentiationVal = l.getDifferentiationVal
```

```
#Assume there is some mapping that exists such that importance -> list of
                                    potential Substituion Values
#Call this mapping 'substitutionMap'
def getPossibleUValues(currentNode, currentImportance):
    if currentNode.l.isEP:
        getNestedUVals(currentNode.l, currentImportance)
        checkTExtension(currentNode.l, currentImportance)
    elif currentNode.l.isNaturalLog:
        #If we get to this point we know that the natural log itself is
                                            not just some constant value,
                                             so it becomes a
        #potential substitution value
        substitutionMap.put(currentImportance: substitutionMap.get(
                                            currentImportance).append(
                                            currentNode.l))
        substitutionMap.put(currentImportance+1: substitutionMap.get(
                                            currentImportance).append(
                                            currentNode.l.innerFunction))
        getNestedUVals(currentNode.l.innerFunction, currentImportance)
        checkTExtension(currentNode.r, currentImportance)
    elif currentNode.l.isFExp:
        #there's a lot of logic that goes into finding the substitution
                                            values of exponents,
        #we will simplify it here to just check if the base and/or
                                            exponent are not merely
                                            composed of constants
        #check the github for a more in-depth version of the logic
        if currentNode.l.l.isNotConstant and currentNode.l.l.
                                            isNotSingleVariable:
            substitutionMap.put(currentImportance: substitutionMap.get(
                                                currentImportance).append
                                                (currentNode.l.l))

        if currentNode.l.r.isNotConstant and currentNode.l.r.
                                            isNotSingleVariable:
            substitutionMap.put(currentImportance: substitutionMap.get(
                                                currentImportance).append
                                            (currentNode.l.r))

        getNestedUVals(currentNode.l.l, currentImportance)
        getNestedUVals(currentNode.l.r, currentImportance)
        checkTExtension(currentNode.r, currentImportance)
    else:
        checkTExtension(currentNode.r, currentImportance)
```

```
def checkTExtension(currentNode, currentImportance):
    if currentNode is not None:
        getPossibleUValues(currentNode.l, currentImportance)
```

```
#Note that if we get here, a natural log node will never be met
def getNestedUVals(currentNode, currentImportance):
    if currentNode.isEP:
        substitutionMap.put(currentImportance: substitutionMap.get(
                                     currentImportance).append(
                                     currentNode))
        getPossibleUValues(currentNode.l, currentImportance+1)
        checkTExtension(currentNode.r, currentImportance+1)
    elif currentNode.isFExp:
        if  currentNode.l.isNotConstant and currentNode.l.
                                     isNotSingleVariable:
            substitutionMap.put(currentImportance: substitutionMap.get(
                                         currentImportance).append
                                         (currentNode.l))
            getPossibleUValues(currentNode.l, currentImportance + 1)
            checkTExtension(currentNode.l.r, currentImportance + 1)
        if  currentNode.r.isNotConstant and currentNode.r.
                                     isNotSingleVariable:
            substitutionMap.put(currentImportance: substitutionMap.get(
                                         currentImportance).append
                                         (currentNode.r))
            getPossibleUValues(currentNode.r, currentImportance + 1)
            checkTExtension(currentNode.r.r, currentImportance + 1)
```

```
def integrationSubRule():
    #This refers to the T node that is running this method
    getPossibleUValues(this, 0)
    for subKey in substitutionMap.keys().sort(reverse=True):
        for subVal in substitutionMap.get(subKey):
            localSubValIsU = True
            substitutionExpression = None
            # The substituted letter variable alternates between u and v,
                                         this is to allow
                                         recursive integration
            # that may require the substitution rule. Assume, for the sake
                                          of this pseudocode,
                                         there is a global
            # boolean value named ‘subIsU‘, which is true when the
                                         substituted expression
                                         gets substituted with ‘u‘
            # and false when substituted with ‘v‘.
            if subIsU:
                # Sub with u finds all instances of the subVal in the
                                                     current expression
                                                     with ‘u‘
                # works similarly to something like the replace function
                                                     in a string
                substitutionExpression = subWithU(this, subVal)
                #set the subIsU to false for the next time the
                                                     substitution rule is
                                                     run
                subIsU = False
            else:
                #same thing as the previous block, just with the ‘v‘
                                                     letter variable now.
                substitutionExpression = subWithV(this, subVal)
                #set the subIsU to false for the next time the
                                                     substitution rule is
                                                     run
                subIsU = True

            subVal.differentiate()
            # here we divide the substution expression by the derivative
                                                 of subVal as you would
            # normally when doing the substitution rule. In the actual
                                                 program,
            # the result of this is simplified by the computational
                                                 algebra system we are
                                                 using
            # assume for the sake of this pseudocode that the result of
                                                 the division is
                                                 simplified
            simplifiedSubVal = T(EP(E(substitutionExpression, None)), TE(T
                                                 (subVal.
                                                 getDerivativeValue(),None
                                                 ),'/'))
            simplifiedSubVal.integrate()
            #If it is None, it means no integral could be found with this
                                                 substitution value
            if(simplifiedSubVal.getIntegral() is not None):
                this.integralValue = simplifiedSubVal.getIntegral()
                return
            #reset what our substitution letter is
            subIsU = !subIsU
    # note that if the outer loop is completed without interruption, it
                                             means no integral was found using
                                              the substitution rule
```