Previous: Short Biographies **Next**: Conclusions

Home: Table of Contents

---

5. Computer Algebra Systems

5.1 Introduction - What is a Computer Algebra System?

A Computer Algebra system is a type of software package that is used in manipulation of mathematical formulae. The primary goal of a Computer Algebra system is to automate tedious and sometimes difficult algebraic manipulation tasks. The principal difference between a Computer Algebra system and a traditional calculator is the ability to deal with equations symbolically rather than numerically. The specific uses and capabilities of these systems vary greatly from one system to another, yet the purpose remains the same: manipulation of symbolic equations. Computer Algebra systems often include facilities for graphing equations and provide a programming language for the user to define his/her own procedures.

Computer Algebra systems have not only changed how mathematics is taught at many universities, but have provided a flexible tool for mathematicians worldwide. Examples of popular systems include Maple, Mathematica, and MathCAD. Computer Algebra systems can be used to simplify rational functions, factor polynomials, find the solutions to a system of equation, and various other manipulations. In Calculus, they can be used to find the limit of, symbolically integrate, and differentiate arbitrary equations.

Attempting to expand the equation

$$(x-100)^{1000}$$

using the binomial theorem by hand would be a daunting task, nearly impossible to do without error. However, with the aid of Maple, this equation can be expanded in less than two seconds. Differentiating the result term-by-term can then be performed in milliseconds. The usefulness of such a system is obvious: not only does it act as a time saving device, but problems which simply were not reasonable to perform by hand can be performed in seconds.

Leibniz and Newton developed calculus in terms of algorithmic processes. Computer Algebra systems can now take these methods and remove the human from the process. However, in studying Calculus and even simple algebraic operations, it would seem that computers would be extraordinarily inept at performing such tasks. After all, most of us consider there to be a great deal of problem solving involved in the mathematics taught in grade school and beyond. How is it that a computer, a mindless composition of binary digits, is able to perform such complex tasks? It would seem that the computer would be unsuitable for such tasks, but the success of popular Algebra software packages show that this is not the case. On the contrary, Computer Algebra systems often know how to perform more operations on equations than the user!

Rather than discuss the many ways that Computer Algebra systems have altered the education and use

of Calculus, we were most intrigued by how these systems actually worked. Our approach was to begin by researching the theories and issues involved in creating a Computer Algebra system. Coinciding with our research, we began writing our own Computer Algebra system in C++. The rest of this section is dedicated to a summary of our research and the specifics of the implementation we chose.

5.2 Data Structures

5.2.1 Introduction

In order for a computer program to even begin manipulating a symbolic equation, it first must store that equation somewhere in memory. At the heart of any Computer Algebra system is a data structure (or combination of data structures) responsible for describing a mathematical equation. Equations can exist in several variables, contain references to other functions, and can themselves be rational functions. There is no perfect solution to a data structure representation of an equation. One representation might be efficient for certain mathematical operations, but poor for others. Another representation might be inefficient in time and space complexity, but easy to program. Such tradeoffs need to be considered when choosing a representation; there is no absolute answer to the problem.

5.2.2 Polynomials in one variable - Coefficients

In order to begin a discussion of the issues involved in storing a symbolic equation, polynomials of single-variables will be considered. That is, equations that are only of the form $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 x^0$ where a_k is an integer or fractional coefficient. Even in storing such a simple equation, there are numerous issues involved in choosing a data structure.

The coefficient itself can not be stored as a simple data type. The amount of storage for an integer in most languages is typically 16 or 32-bits. In a 32-bit representation of an integer, only 2^{32} or approximately 4.3 billion different numbers can be represented. Though this might seem large, for many mathematical operations (such as the simplification described in the Introduction), numbers of much larger size must be possible to represent. Therefore, a numeric data type that allows for expansive growth in representation (e.g. a data type that grows dynamically with the size of the number) needs to be used. For fractional coefficients, simply storing the numerator and denominator separately in two such data types is adequate.

5.2.3 Polynomials in one variable - Terms

Having dealt with the issue of storing coefficients, the more significant problem of how to actually store each term must be dealt with. The first issue to contend with is that of finding a canonical form. That is, consider that a user types in the following single-variable equations:

$x^2 - x$ and $-x + x^2$

$(x+2)(x-2)$ and $x^2 - 4$

In both instances, a human can easily expand and re-order the equations to determine equivalence. However, for a computer, this is far from a trivial task. The Computer Algebra system must represent these equations in a canonical form, one in which only one representation exists for each equation. That is, in the above examples, the Computer Algebra system would simplify both pairs of equation into the same representation in internal memory.

For a single-variable polynomial, once fully simplified (more on this later), finding a canonical form is not difficult. In the internal representation, simply sort the terms by degree of their exponent, and each version of the equation will correspond to the same representation. The next step is to determine a way to store each term in the computer's memory. One approach would be to create an array of the size equal to the largest exponent in the equation. An array is simply a collection of items of the same type, the size of which does not change after instantiation. For example, consider a user enters the equation $x^5 + 2x + 1$. The program would create an array of 6 elements. At each element in the array, it would store the coefficient of the corresponding term (and place a 0 in all the unused terms):

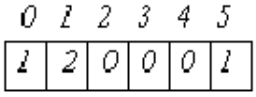

Figure 5.1

This representation is known as dense because it stores each of the terms, independent of whether the coefficient of the term is 0. An alternative representation would be to create a list or similar data structure that at each node stores the coefficient and the exponent of the term. This way, only the terms that actually are used require storage in memory:

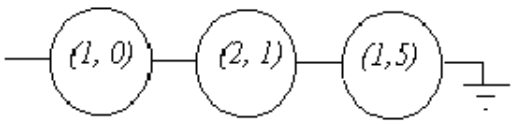

Figure 5.2

At each node in the list, the first number in the pair represents the coefficient and the second the exponent of each term. This representation is sparse: only the terms which have non-zero coefficients are stored in memory.

Typically, a sparse representation is preferable to a dense one, but there are advantages and disadvantages to both. In a dense representation, there often can be large amounts of computer memory wasted. For example, the equation $x^{2000} + x$ would require an array of 2001 elements just to store two coefficients. Comparatively, the sparse version would require only two nodes in a list. Additionally, from the programmer perspective, it is difficult to make changes to a dense representation. For example, if one were to add a new term to a polynomial that was not included in the range of the original array, the array would have to be completely recreated (in most languages, the size of an array can not be modified without completely recreating it). The sparse representation provides much more flexibility, as adding a new term is simply a matter of adding a new node to the list in the correct place (to maintain a canonical form). The choice between a sparse and dense representation is completely dependent on the task for which it will be used. In some cases, a system will shift between representations in order to optimize for specific algorithms (Davenport 59-70).

5.2.4 Polynomials in one variable - Recursive definition

A Computer Algebra system supporting only equations of the form $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 x^0$ would be quite inflexible. One enhancement, while still remaining in the realm of polynomials of one variable, is to allow equations with recursive definitions. That is, polynomials where the coefficient to a term or numerous terms can be a polynomial itself. Some examples of such equations would be $(x+1)(x+2)$, $(3x^2 + 3x)x$, and $[(3x+1)2x]x^5$. The sparse representation can be easily modified to support equations of such a form. Instead of simply storing pairs of coefficients and their exponents, the coefficients themselves can also point to lists of polynomials. In order to find the canonical form, the polynomial is simplified and all recursive polynomial coefficients are removed from the representation. In order to support rational functions, all that is needed is to store two polynomials: the numerator and denominator. Of course, simplification of a rational function into canonical form introduces a host of new issues, but these issues will be discussed later.

5.2.5 Multivariate Polynomials

The sparse representation can be extended into multivariate polynomials without too much effort in terms of representation. Rather than storing simply a coefficient and degree at each node in the list, the degree will be replaced with a list of variables and their respective degrees. The difficult issue that arises when switching to a multivariate representation is finding a canonical form. One approach that is commonly used is to first sort the terms lexicographically and then by degree. As an example, consider the equation

$$y^2 + z + xy + x^2 z + yz + yz^2$$

would be represented in the following manner after sorting

$$xy + x^2z + y^2 + yz^2 + yz + z$$

The selection of sorting is irrelevant (whether first by lexicographic order and then degree or vice versa) as long as the choice is consistent (Davenport 71-74).

5.2.6 The Syntax Tree - Our Data Structure Implementation

In the spirit of sparse representation, we chose to use a syntax tree as the internal data structure for our symbolic calculator. A syntax tree is a kind of tree, which in turn is a kind of linked data structure. Briefly, a linked data structure is an object which contains references, or links, to other like objects. A simple example is a linked list, where each element contains the data for it list entry and a link to the next list element. A tree is a linked structure that starts with a single "root" node. One or more "child" nodes are referenced from the root node, and each of these child nodes may in turn have children of their own. This linking pattern produces a branching data structure, as seen in the following diagram; hence the name "tree".

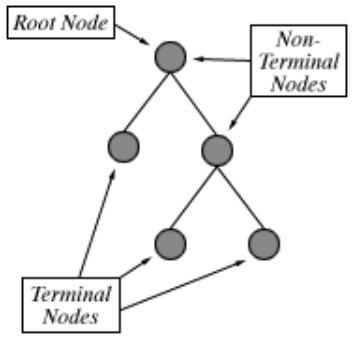

Figure 5.3

Trees are acyclic, which means that nodes cannot be linked in a loop. Each node has exactly one "parent" node, that is, one node of which it is a child. The exception is the root node, which has no parent. Nodes with no children are called terminal nodes, or "leaf" nodes. A syntax tree is a type of tree where each non-terminal node represents an operator or function, and its children represent its operands or arguments. In our program, mathematical operators such as addition and multiplication, or mathematical functions such as *sin* or *log* are represented by these non-terminal nodes. Leaf nodes represent the terminal symbols of an expression, such as numbers, constants or variables. The structure of a syntax tree represents syntactic information about the data it contains.

In our case, the syntax tree represents the syntactic steps, or order of operations involved in evaluating a symbolic expression. For example, the simple expression $a + b * c$ could be represented by the following

syntax tree:

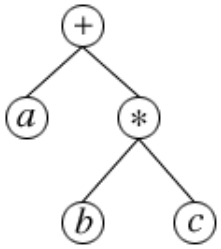

Figure 5.4

The root of this tree is the addition operation, and the children are its operands. The hierarchy of operators and arguments establishes a clear precedence of operations. The syntax tree for the expression $(a+b)*c$ is shown below:

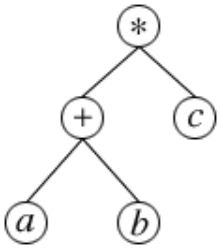

Figure 5.5

These two syntax trees are different, as are the expressions they represent. Syntax trees offer a clear and unambiguous way to store a wide variety of expressions.

5.2.7 The Syntax Tree - Advantages

The major advantage of the syntax tree is that it is flexible. It can represent a wide variety of different expressions that cannot be easily captured in the previously discussed data structures. Take, for example, an expression as simple and common as e^*. The polynomial representations are limited to just that - simple polynomials. Perhaps the sparse multivariate polynomial could be extended to think of *e* as a kind of "special" variable - namely a constant. Furthermore, and more difficult, the representation would have to be extended to allow for non-integer exponents. Even then, what happens when the exponent is more complicated than a single non-terminal symbol? What if the exponent is itself a polynomial

expression, replete with its own exponents, and so on and so forth? The problem quickly grows out of any attempts of reasonable management. This doesn't even touch on the matter of functions within expressions, as in $x^2 + \sin(x)$. Granted, the representation was designed with polynomials in mind, but we wanted something more general. All of these expressions are easily represented in abstract syntax trees:

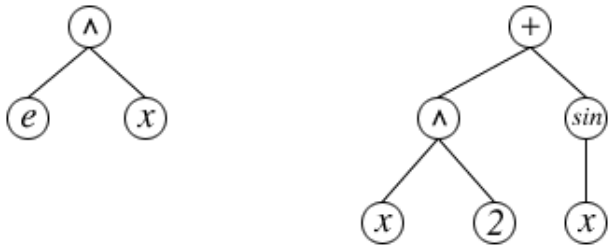

Figure 5.6

5.2.8 The Syntax Tree - Disadvantages

The generality that makes the syntax tree so appealing is also its biggest problem. While it's possible to represent numerous different expressions with a syntax tree, it's also possible to represent a single expression a number of different ways. For example, the expression $\frac{ab}{cd}$ can be represented in a number of different ways:

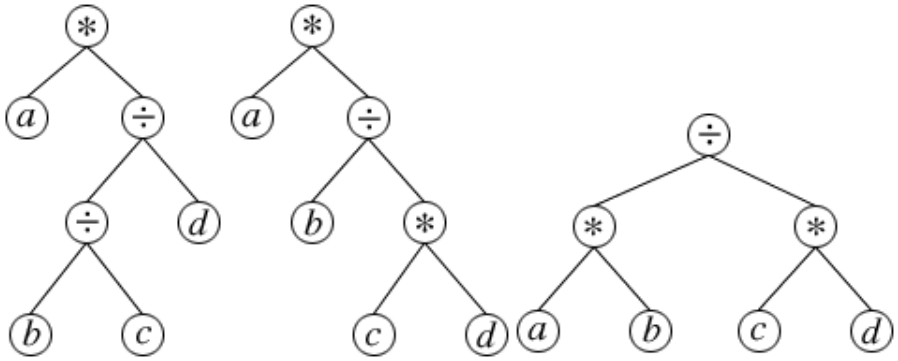

Figure 5.7

All are mathematically equivalent, but to a computer, they look nothing alike. As mentioned before, it's important to define a canonical form. Changing an expression to canonical form can be a difficult task in itself, but due to the wide variety of expressions syntax trees can represent, it's hard to define exactly what canonical form should be. Certain types of expressions tend to fit some forms better than others. A polynomial fits nicely in a form that orders terms by their degree. What happens when the exponents are

$x^{(y^2+y+1)}$ $x^{(y^2+y+2)}$

more complicated? Both and could be said to have the same "degree" - that is, a polynomial in y of degree 2. Defining a general algorithm for ordering becomes complicated. Consider, also, the example of *x* as compared to *1 * x* or x^1. The trees of these expressions are as follows:

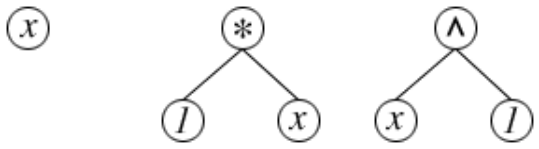

Figure 5.8

While it may be unlikely for a user to enter x^1 instead of *x*, it is not hard to imagine that such an expression may be obtained in the process of manipulating the expression symbolically. For example, $\frac{x^3}{x^2}$ might be simplified to x^1. The procedure to convert expressions to canonical form must take many factors into account.

5.3 Simplification

5.3.1 General Issues in Simplification

A very common task that any computer algebra system must perform is to simplify an expression. Simplifying expressions makes other tasks much easier, especially comparing expressions entered in different forms to see if they are equivalent. The system has to know how to add terms that should be added and how to add exponents together when the multiplicands have the same base. Computer algebra systems must always orders all the terms to arrive at a canonical form. There must be a consistent order that everything is sorted by. If exponents can be polynomials, such as x^{x^3+3+x}, then those exponents also need to be sorted. This could go on indefinitely, with each additional power being another complex polynomial. Sorting will need to take place on several levels to make it consistent. The uppermost exponents must be simplified first so that the lower level ones will sort properly. One can not simply sort from the lowest level first.

Systems must also decide whether or not to perform some simplifications. Identities for operations must be taken into account, so adding zero and multiplying by one will be dealt with properly. Either strip out all such identities or add them in to every operation, which would probably require more work and produce a more cluttered display.

There is also the decision whether or not to expand polynomials. Expanding $(x+2)^2$ may be trivial, but expanding $(6x^3+2x)^{300}$ would require an enormous amount of memory and time, not to mention far too much display space on the screen to be impractical. The factored form is much more compact. If integers are raised to a power, the limitations of the computer?s number system may hinder expansion. This again shows why it is important to choose a number representation system that allows for arbitrarily large values. It may be necessary to expand simple polynomials of a very high order into large polynomials of a very high order in order to simplify further, but a lone value should probably be left as it is for simplicity in display. Expanding might be appropriate when there are several polynomials, and terms will cancel out when expanded, but no further simplification can be done otherwise.

The systems might also want to apply trigonometric or other identities to expressions for simplification of functions. For example, $\sec^2(x^2+1)$ and $\tan^2(x^2+1)+1$ are equivalent if the trig identity $\tan^2(x)+1=\sec^2(x)$ is applied. There are many other ways to manipulate functions, especially trigonometric ones, that could hinder further simplification. Sometimes the only way to simplify further is to apply these identities, which makes knowing when to use identities difficult. The natural logarithm function also has identities, which, besides being used to simplify expressions with the natural logarithm function, can also be used to simplify some other expressions. The system must know when to apply these identities and when to leave the functions as they are.

Simplifying is not only used when an expression is first entered in by the user, but, in particular, differentiating an equation will produce an expression that will need to be simplified for it to look like what the user expects to see. It is important that the computer algebra system be able to represent everything that may happen when expressions are simplified and expanded, but it must also decide whether or not to simplify certain operations depending on the circumstances.

5.3.2 The Steps of Simplification - Our Approach

In order to break up the complex task of simplification and reduction to a canonical form, we created a number of small algorithms that performed very simple, specific operations on syntax trees. By calling these simple procedures in order, and repeatedly, we are able to simplify many equivalent representations into a single deterministic form. In the following sections, we will describe the steps taken. Some of the steps seem to move away from simplification instead of towards it - these are intermediate steps that make later simplification easier.

5.3.3 Transforming Negatives

In this step, all negative operators (unary negative and subtraction) are transformed to terminal constants with negative values. For example, *x* becomes *1 * x* and *a - b* becomes *a + (-1 * b)*. The trees of these expressions are as follows:

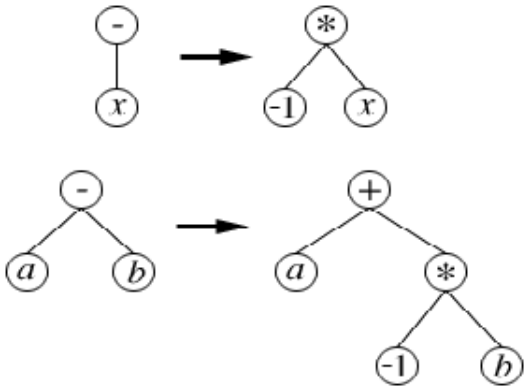

Figure 5.9

This step, while extremely simple, has a number of advantages in terms of defining a canonical form and simplifying later operations. For the canonical benefit, the above pairs of expressions, and others like them can be determined to be equivalent, obviously. The real benefit of this operation is that it significantly reduces the syntactic complexity of the expression trees. Two elements, negation and subtraction, are removed from the set of operators that have to be dealt with in later stages.

Subtraction is replaced by addition, a commutative operator, which allows greater flexibility in ordering. A subtraction node must have exactly two children, and their order cannot be reversed. An addition node, on the other hand, can have any number of children, and they can appear in any order.

5.3.4 Leveling Operators

When the expressions *a * b * c* and *a + b + c* are parsed by our calculator, the following syntax trees result:

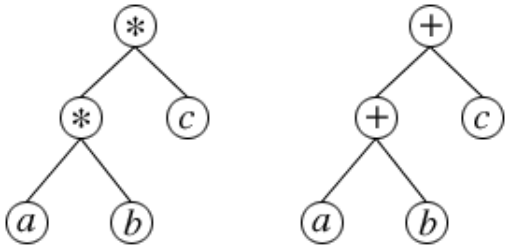

Figure 5.10

This is due to the fact that our parser assumes that all operators, with the exception of negation, are binary - that is, they have two operands. Since the parser is designed to read expressions written in infix notation, this is a valid assumption. But there’s no reason that commutative operators such as addition

and multiplication have to be binary in another notation. For example, *a + b + c* could be written as *(+ a b c)* in prefix notation. The same is true of our syntax tree. For example, the expressions *a + b + c + d* and *(a + b) + (c + d)* would be parsed as follows:

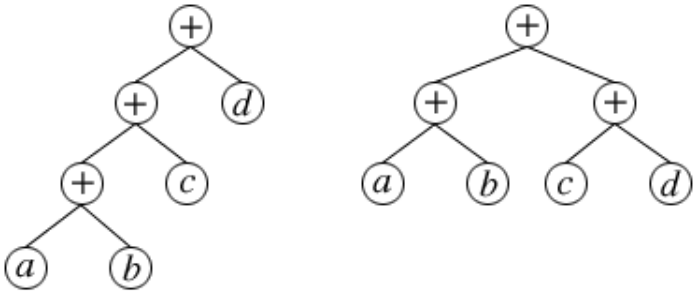

Figure 5.11

However, after the simplification step of leveling operators, both expressions are represented as:

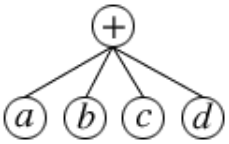

Figure 5.12

This operation trims unnecessary complexity from the syntax trees and resolves problems of associativity in canonical form.

5.3.5 Simplifying Rational Expressions

Recall the various syntax trees for the expression $\frac{ab}{cd}$ illustrated in section 5.2.8. This simplification step will transform a syntax tree so that a division node cannot be the immediate child of either a division node or a multiplication node. The end result is that any expression formed of multiplicative operators (multiply and divide) will be transformed so that there is a single division node at the top of the tree, with only multiplication operators below it. This simplification takes three specific cases into account in order to form a general procedure for other cases. The first case is the event when a division node (D_1) has another division node (D_2) as its numerator. In order to simplify this situation, the numerator of D_1 must become the numerator of D_2, and the denominator of D_1 must become the product of the denominators of D_1 and D_2. This transformation can be seen in the following diagram:

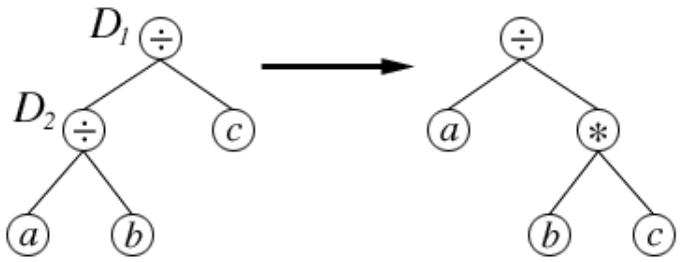

Figure 5.13

The second case is very similar to the first. In this case, the second division node (D_2) occurs in the denominator of the first (D_1). The numerator of D_1 becomes the product of the numerator of D_1 and the denominator of D_2. The denominator of D_1 becomes the numerator of D_2, as seen in the following illustration:

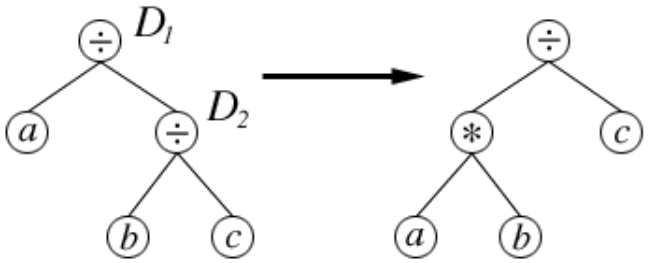

Figure 5.14

The final case is when a child of a multiplication node (M) is a division node (D). It doesn't matter how many children the multiplication node has, or how many of those children are division nodes. Only the first division node is considered in this simplification. This situation is a little more complicated than the previous two since the operation of the top node must be changed and its children moved, rather than just reshuffling some node links. To simplify this case, M is replaced by a division node whose numerator is the product of the numerator of D and the children of M (with the exception of D itself), and whose denominator is the denominator of D. This transformation is shown below:

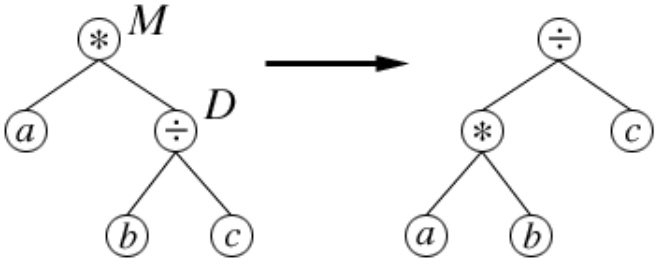

Figure 5.15

From repeated application of these three cases, more complicated expressions can be reduced:

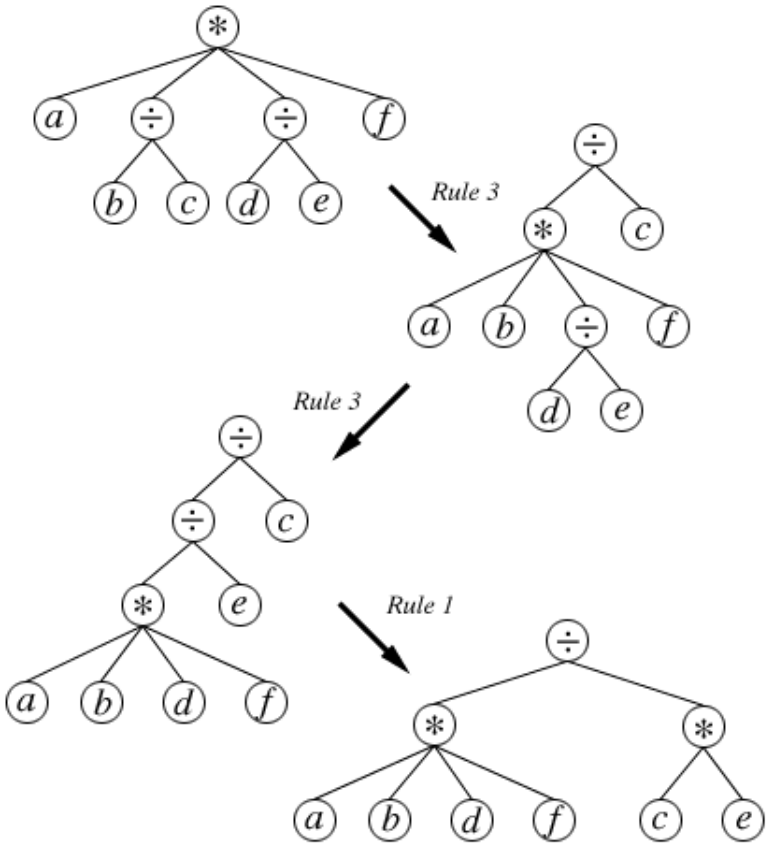

Figure 5.16

5.3.6 Collecting Like Terms

The first step involved in collecting like terms is to explicitly represent any coefficients or exponents a term may possess. For example, *x + 2x* becomes *1x + 2x* and $x * x^2$ becomes $x^1 * x^2$. The main reason for doing this transformation is to make all the children of an addition or multiplication node share a common form - that is, all the children are either multiplication nodes or power nodes, respectively. In order to collect like terms below a multiplication node, one compares the base of each child power node (P_i) with the bases of the remaining children (P_{i+n}). In the event that two bases are equal, the exponent of P_i becomes the sum of the exponents of P_i and P_{i+n}, and P_{i+n} is removed:

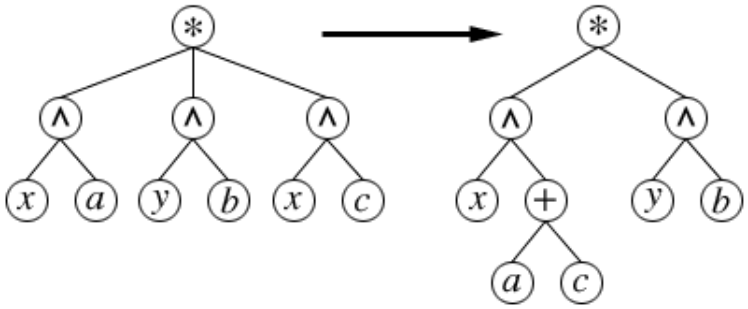

Figure 5.17

A similar operation would take place for collecting like terms under an addition node, but we have not actually implemented it in our calculator.

5.3.7 Folding Constants

Once terms have been collected together, unnecessary constants can be collected or removed. A constant, in this sense, is a real number in the expression, such as the three in $3 * e^{kx}$. The *k* is a mathematical constant, but for purposes of symbolic manipulation, it is treated as a variable. When multiple constants occur below an addition or multiplication node, they can be combined (added or multiplied as the case may be) into a single constant. When both children of a power node are constants, it can optionally be replaced with a single number, although it is not always wise to do so. The number 10^{9000} is usually expressed as such because a 901 digit number is unwieldy. In our calculator, we fold a constant power term if the result is less than 1000, an arbitrary choice. Furthermore, a power term with a base of zero can be folded to zero, unless the exponent is also zero. In that case, our calculator simply leaves 0^0 alone since it has no provisions for indeterminate forms. Power nodes with a base of one can be reduced to one, and power nodes with exponents of one or zero can be reduced to the base alone or one, respectively, with the previously mentioned exception of 0^0. If a multiplication node contains a one, that child can be eliminated; if it contains a zero, the whole multiplication node can be replaced with zero. Also, if a multiplication or addition node is left with a single child in the course of these reductions, the node can be eliminated and replaced with its sole child.

5.3.8 Canonical Order

All of these simplifications are fine and wonderful, but what's the use if they can't even determine that *a + b* and *b + a* are equivalent? That's why it's important to define a canonical ordering of terms, as discussed in section 5.2.2 and 5.3.1. In order to arrange our syntax trees in canonical order, all the children of a commutative node are sorted with a simple ordering function. The children are sorted first by their node type. In our calculator, there is a different node type for each operator, and one for each of the following: variables, functions, and constants. After node type, the children are sorted

lexicographically. This ordering scheme doesn't always order expressions the way one would expect to see it written, but it works well with syntax trees and is consistent - which is the important part.

5.3.9 Full Simplification

Sometimes a single iteration of the simplification steps is not enough to reduce an equation as much as it should be. To compensate for this, we keep iterating through these simplifications until the syntax tree ceases to change.

5.4 Advanced Operations

5.4.1 Introduction

After canonically representing an equation in memory, the Computer Algebra system can demonstrate its true power. The advanced operations that a system is capable of performing are what separate one system from another. Advanced operations include factorization, differentiation, integration, and finding the limit of a function.

5.4.2 Differentiation

Mathematical operations that are defined in terms of algorithmic processes are rather painlessly integrated into Computer Algebra systems. Assuming that an appropriate representation is chosen for describing an equation, any algorithmic manipulation can be fairly easily translated into a Computer Algebra system. Differentiation is one such operation that is defined algorithmically in a very general way and is therefore particularly well suited to a computational definition.

Differentiation essentially consists of four basic rules (Davenport 165):

$$(a \pm b)' = a' \pm b'$$
$$(ab)' = a'b + ab'$$
$$(\frac{a}{b})' = \frac{a'b - ab'}{b^2}$$
$$f(g(x))' = f'(g(x))g'(x)$$

The algorithm must only know two additional pieces of information. First, the algorithm must be

$$(x^{\frac{p}{q}})' = (\frac{p}{q} x^{1-\frac{p}{q}})$$

informed that , which enables the computation of the derivative of any function that does not contain references to other functions. Second, to be a completely flexible at differentiation, the algorithm must be aware of the derivatives of functions (e.g. sin, cos, ln, etc.). The differentiation of functions can easily be accomplished by storing a table of derivatives.

The ability of a computer to perform differentiation is thus demystified. Because we, as human problem solvers, compute derivatives in a very algorithmic way, it is easy for a computer to emulate such behavior. Artificial Intelligence is the attempt at algorithmically modeling a human's ability to think. However, when there is no obvious algorithm that exists, modeling such behavior becomes extremely difficult (and, at this point, all attempts are nothing more than an approximation). The same is true in Computer Algebra systems: some mathematical computations are not clearly performed algorithmically.

5.4.3 Integration

Integration is an example of an operation which, at first, appears to have no algorithmic definition. The only general rule that appears to be useable is that $\int f + g = \int f + \int g$. However, even this rule turns out to be unusable in certain cases. Consider attempting to find $\int x^x + (\log x)x^x$; breaking it up at its addition will not yield a solution. The two respective parts have no integral, yet the integral of the combination yields x^x (Davenport 167).

Integration appears to be a compendium of different techniques such as integration by parts, integration by substitution, and simply consulting a table of known integrals. Which integration problems require which technique can not be generally defined. The first attempts at computer integration, then, took an obvious brute force approach. That is, try all possible known techniques until an answer is found.

Ultimately though, a full theory of integration in terms of an algorithmic process that computers can perform was developed. This theory is beyond the scope of this paper, but a summary of the theory is developed in Davenport, Siret, and Tournier pp. 167-186.

5.4.3 Differentiation - Our Implementation

The only advanced operation we chose to implement is differentiation in one variable. In order to differentiate a syntax tree, one must take a top-down approach. The differentiation procedure starts with the root node and tries to differentiate it based on what type of node it is. For example, if the node is an addition node, the derivative of the node is an addition node whose children are the derivatives of each of the original node's children. Differentiation is an inherently recursive procedure, and syntax trees are well suited to recursive evaluation. Below is a list of how each type of node is differentiated:

- *Addition Node*: As mentioned above, the derivative of an addition node is an addition node whose children are the derivatives of each of the original node's children.

- *Multiplication Node*: If a multiplication node has *n* children, then by the product rule, the derivative of a multiplication node is an addition node with *n* children. Each child of the addition

node is a multiplication node, also with *n* children. In the i^{th} multiplication node, the i^{th} child is the derivative of the i^{th} child of the original node, and the other children are the same as the other children in the original node. For example, the derivative of *x * y * z * w* is $x' * y * z * w + x * y' * z * w + x * y * z' * w + x * y * z * w'$.

- *Division Node*: The derivative of a division node is simply expressed by the quotient rule. The only difference is that the subtraction is replaced by addition and the second term is multiplied by -1 (in keeping with the idea of eliminating subtraction operations).

- *Power Node*: Though the power rule is one of the first methods of differentiation we learned, it wasn't very practical for our calculator. Instead we used the more general form $\frac{d(b(x)^{e(x)})}{dx} = b(x)^{e(x)} \frac{d(e(x) * \ln(b(x)))}{dx}$. It makes a mess of simple things like constant bases or powers, but if the resulting tree is simplified, everything is cleaned up.

- *Function Node*: Our program will only try to differentiate functions of one variable, although it will symbolically manipulate functions of an arbitrary number of variables. In fact, the only functions it knows how to differentiate at the moment are *ln*, because it occurs so much in the differentiation of power nodes, and *sin* and *cos*. If the program doesn't know how to differentiate a function, it will simply encase the function and its arguments in a *Deriv(a,b)* function, where *a* is the unknown function, and *b* is the variable with respect to which the function is differentiated.

- *Variable Node:* If the variable is same as the independent variable for which we are differentiating, then the derivative node is the constant 1. Otherwise the variable is considered a symbolic constant and the derivative is the constant 0.

- *Constant Node*: The derivative is always the constant 0.

Previous: Short Biographies **Next**: Conclusions

Home: Table of Contents