ALGEBRAIC SIMPLIFIER INCORPORATING NON-STANDARD ANALYSIS

BY EDGAR B. REEVES JR.

DECEMBER 2, 1996

COMMITTEE MEMBERS:
Dr. David E. Leasure (chair)
Dr. David R. Thomas
Dr. George D. Tintera
Contents

Abstract 01
Background and Rationale 02
Narrative 12
Environment 17
Procedure 18
End Results 30
Bibliography 33
Appendix 34
Abstract

This thesis discusses an algebraic simplifier program that can be used as either a subprogram in a more complete theorem prover, or to perform as a stand-alone program. The overall method uses natural deduction to implement Abraham Robinson's non-standard analysis for polynomial derivatives. The program also automates the symbolic calculation and algebraic simplification of polynomial, exponential, and trigonometric derivatives.
Background and Rationale

This algebraic simplifier is a program that can evaluate and simplify algebraic expressions, and can symbolically calculate many of the derivatives of elementary calculus. It calculates many elementary derivatives using the non-standard formulation of the definition of the derivative. A principal use of such an algebraic simplifiers is as a subprogram in a larger theorem prover, such as the one using non-standard analysis which was done at The University of Texas (Ballantyne, 1977).

A description of non-standard analysis follows in section A, and more on the background and rationale of this program is in section B.

Section A: Isaac Newton is generally credited with the first development of "infinitesimal calculus", although G. W. Leibniz seems to have discovered it independently not long afterward. "Infinitesimal numbers" were used in calculus by mathematicians, scientists, and engineers until the mid-nineteenth century, when Weierstrass seemingly banished infinitesimals from analysis forever. He formulated the calculus proofs using the epsilon-delta notation, which restricts them to Archimedean numbers in the standard number system. For a number \( x \) to be Archimedean means that it is not the case that \( 0 < x < 1/n \) for all positive integers \( n \). On the other hand, if the number \( x \) were one of Robinson’s non-standard infinitesimals, then the above inequality would hold. Also, infinitely large numbers can be introduced as reciprocals of
the infinitesimals.

Weierstrass' methods of proof were designed to remove certain contradictions from analysis as it was formerly done with infinitesimals. Bishop Berkeley discussed these contradictions in his paper of 1734, addressed to "an infidel mathematician", thought to be astronomer Edmund Halley. It was believed that he had persuaded one of the Bishop's friends of "the inconceivability of Christianity". Berkeley said that if a mathematician could attempt theology, then he would attempt mathematics. The resulting critique was not dealt with satisfactorily for more than one hundred years; that is, not until Weierstrass' work.

One illustration of the type of contradiction under consideration is from Newton's calculation of the instantaneous velocity of a falling body. For example, this turns out to be 32 feet per second at time \( t = 1 \) second. He obtained this by differentiating the position function \( s = 16t^2 \), which is a simplified form of \( S = S_0 + V_0t + \frac{1}{2}gt^2 \), with \( g = 32 \text{ fps}^2\), and is adequate for our purposes. One definition of the derivative is

\[
\frac{ds}{dt} = \frac{(f(s+dt) - f(s))}{dt},
\]

where the \( dt \) quantities are "infinitesimally small numbers". The result of the differentiation is 32 + 16\( dt \) feet per second at \( t = 1 \) second. The "infinitesimal quantity", 16\( dt \), is neglected in order to arrive at the 32 fps answer. However, in Newton's time, the derivative could not be calculated without using infinitesimals, and, consequently, one of them actually occurs in the first formulation of the answer above. An infinitesimal cannot
be 0, since it occurs in the denominator of a fraction in the
calculation of the derivative, yet it was neglected as though it
were 0 in the final answer. This material is referenced in any
sufficiently detailed history of mathematics.

Such contradictions were removed by expunging infinitesimals
from the system. However, since the mid-twentieth century, they
have been reinstated in Abraham Robinson’s non-standard analysis.
For example, the non-standard number system is constructed so that
if the non-standard number \( x_1 \) differs from the non-standard number
\( x_2 \) by an infinitesimal amount, we write \( x_1 \sim x_2 \), and if this is so,
then the elementary calculus formula for the derivative becomes

\[
(1) \quad \frac{dy}{dx} = \left( \frac{f(x_2) - f(x_1)}{x_2 - x_1} \right), \text{ at } x_1.
\]

This expression is easier to automate than the usual form, which is

\[
(2) \quad \frac{dy}{dx} = \lim_{h \to 0} \left( \frac{f(x+h) - f(x)}{h} \right), \text{ at } x.
\]

The foundations of non-standard analysis are found in modern
logic. The completeness theorem states that a set of sentences is
logically consistent (no contradiction can be deduced from them) if
and only if the sentences have a model; that is, if and only if
there is some "universe" in which they are all true. The
compactness theorem then proceeds as follows: Suppose that there
is a collection of sentences in the language \( L \), such that every
finite subset of the collection \( L \) is true in the standard universe.
If this is the case, then there is a non-standard universe in which the entire collection is true at the same time. This result follows from the completeness theorem, because if every finite subset of the collection $L$ is true in the standard universe, then every finite subset is logically consistent. Therefore, since any deduction can make use of only a finite number of premises, the entire collection is logically consistent. Then, by the completeness theorem, there is a (non-standard) universe which is a model for the entire collection, so that the entire collection is true in that universe. As a consequence, there is a sense in which the infinitesimals exist, though not in the sense that some of the subject matter of physics exists. Model theory does not address such ontological questions, but it does tell us that we should not get into any more trouble reasoning in the non-standard universe than we do in the standard one.

The compactness theorem states that if $L$ is a formal language as before, and if $K$ is the set of all true sentences of $L$ in the standard universe, then each of the sentences in $K$ is true, with suitable modifications, in the non-standard universe. Even though we do not know all of the elements of $K$, we can still reason and draw conclusions about it. One consideration is that since all standard real numbers are Archimedean and some of the non-standard reals are not, the Archimedean property cannot be expressed in one of the sentences in $L$, nor can Berkeley's contradiction. Of course, since $K$ is a subset of $L$, then sentences expressing either the Archimedean property or the contradiction are not in $K$. The
"pseudoobjects" in the non-standard real number system behave formally like standard objects, yet differ with respect to important properties which are not formalized by L.

Even though the standard real number system \( R \) and the non-standard system \( R^* \) are conceptually distinct, it is useful to think of the standard numbers as being embedded in the non-standard reals. Since the non-standard universe \( R^* \) is a model for L, every sentence in \( K \) (which is true for the standard real numbers) has a true interpretation in \( R^* \). In particular, we will identify the ("pseudo") object "2" in \( R^* \) with the ("real") object "2" in the standard reals, so that if we continue in this manner, \( R^* \) will contain the standard real numbers, as well as the infinite collection of infinitesimal and infinite quantities in \( R^* \). \( R^* \) is thought of as having an infinite collection of "pseudoreals" clustered around every standard real number, so that each of these infinite collections is within an infinitesimal neighborhood of a given standard number. This standard real number is called the standard part of each of the "pseudoreals" in its infinitesimal neighborhood. That is, if \( e \) is an infinitesimal number and \( r \) is a standard real number, then \( p = r + e \) is one of the infinitely many "pseudoreals" in the infinitesimal neighborhood of \( r \), and \( r \) is called the standard part of \( p \).

Note that the Newtonian falling body problem can now be formulated in the non-standard number system, so that instead of defining the instantaneous velocity as the ratio of infinitesimal increments, it is defined to be the standard part of that ratio,
and \( ds, dt, \) and the ratio of them, \( ds/dt, \) are now non-standard numbers. The careful distinguishing between the non-standard \( ds/dt \) and the standard real number \( v \) avoids the contradiction, which many of the mathematicians and scientists before Weierstrass simply ignored. We can conclude rigorously, without resorting to arguments involving limits, that \( v = 32 \) feet per second exactly, at \( t = 1 \) second.

In general, results such as proofs of theorems, which are obtained in one system, can be "imported" into the other without loss of validity. In this project, derivatives obtained in the non-standard number system are converted and returned to the standard number system.

The development of non-standard analysis is proceeding in many pure and applied areas. For example, work has been done in non-standard probability, non-standard dynamics, and a non-standard model of space and time has been used to provide new existence results for the Boltzman equation. In some areas of mathematical physics, such as those involving large finite collections of particles, the physical situation may be more accurately modeled by a hyperfinite set (A hyperfinite set is infinite, but is finite from the non-standard point of view, and therefore, inherits many of the properties of finite sets.). Also, heuristic reasoning with infinitesimals, which even the "purest" of pure mathematicians might do in a weak moment, can be made rigorous in a way that it could not before (Cutland, 1988).
Section B: Another advantage of this program is that it provides a way to obtain symbolic derivatives, rather than numerical approximation methods. This provides heuristic problem-solving advantages, and it is sometimes helpful to have a derivative function rather than a numerical answer.

Also, this simplifier employs a natural-deduction system, which is easier to trace than, for example, a resolution theorem prover like J. A. Robinson's. The resolution theorem prover rewrites the hypotheses and the negation of the conclusion of a proposed theorem as a series of conjunctions of disjunctions, that is, in conjunctive normal form. It then tries to find a contradiction, which, if successful, proves the theorem. This rewriting of the original theorem completely obscures the original structure to human readers. Also, it is virtually impossible to trace intelligibly the deduction steps followed by the computer. This obviously would preclude its use in an interactive environment, and in any case tends to obfuscate rather than facilitate the research and solution of problems. For example, in a natural deduction system, the following might be displayed during a proof:

\[
\begin{align*}
H1: & \quad P \\
H2: & \quad (P \rightarrow Q) \\
H3: & \quad (R \& Q \rightarrow S) \\
C1: & \quad Q \\
C2: & \quad (R \rightarrow S) .
\end{align*}
\]
This would indicate that each of the \( H(n) \) is an hypothesis, and that each of the \( C(m) \) is a goal. On the other hand, a resolution system would display the same situation with a set of disjunctive clauses:

\[
\begin{align*}
1. & \quad P \\
& \quad \& 2. \quad -P \text{ or } Q \\
& \quad \& 3. \quad -R \text{ or } (-Q \text{ or } S) \\
& \quad \& 4. \quad -Q \text{ or } R \\
& \quad \& 5. \quad -Q \text{ or } -S.
\end{align*}
\]

Even though the two representations are logically equivalent, all information about goals, which is what we are trying to prove, has been lost in the second (Barr, 1981).

Some of the advantages of the Logistica programming language can be appreciated by comparing a program segment written in Scheme to a similar one in Logistica. Suppose that the following four differentiation rules were to be implemented in Scheme:

\[
\begin{align*}
dc/dx &= 0 \text{ for } c \text{ a constant or a variable different from } x, \\
\frac{dx}{dx} &= 1, \\
d/dx (u + v) &= du/dx + dv/dx, \text{ and} \\
d/dx (uv) &= u (dv/dx) + v (du/dx).
\end{align*}
\]
About a dozen lines of Scheme code would be needed, and there would have to be eleven preliminary procedures to implement selectors, constructors, and predicates to do the following:

For e an expression, a(i) a term, m(j) a factor, and v(n) a variable,

Determine whether e is a constant
Determine whether e is a variable
Determine whether v(1) = v(2)
Determine whether e is a sum
Determine whether e is a product
Obtain the addend of the sum e
Obtain the augend of the sum e
Obtain the multiplier of the product e
Obtain the multiplicand of the product e
Construct the sum of a(1) and a(2)
Construct the product of m(1) and m(2).

If the procedures to perform these functions were available, the Scheme code necessary to implement the four rules would be:

(define (deriv exp var)
  (cond ((constant? exp) 0)
       ((variable? exp)
        (if (same-variable? exp var) 1 0))
       ((sum? exp)
        (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
       ...
((product? exp)
(make-sum
(make-product (multiplier exp)
  (deriv (multiplicand exp) var))
(make-product (deriv (multiplier exp) var)
  (multiplicand exp)))))

This code would produce an unsimplified answer, and still more code would have to be written in order to simplify it. For example, the result of \( \frac{d}{dx} (xy) \) would be \( x*0 + 1*y \) instead of just \( y \), and more complex expressions, although answered correctly, would be much harder to read before simplification. The Logistica code necessary to produce the same unsimplified result would be essentially only four statements, which look very much like ordinary mathematical rules for differentiation.
This program uses a rule base which partially incorporates the standard axioms of mathematics that form the basis of the real number system. The rule base also has many rules which are not strictly axiomatic in a precise mathematical sense, but which contribute to the efficiency of the program by reducing the average number of steps necessary to simplify an expression. These might more properly be called lemmas. They are augmented by rules which constitute another set of lemmas for doing the implemented parts of calculus "algebraically". There is also a set of rules for doing derivatives using non-standard analysis definitions. These rules and the associated symbols are written in the Logistica programming language. The control structures are defined in Logistica syntax. Some of these follow from the ordering of the rule statements. The program structure appears to be somewhat "flat" in comparison to the hierarchical modular structure of most procedural languages, but the Logistica statements expand in compilation to produce approximately a one-to-ten ratio in comparison to a program written in Scheme. Therefore, in many cases each statement represents a "mini-module". This provides a significant leveraging of programmer effort, and mirrors the parallel trends in more general-purpose and business-oriented languages.

The following is a brief overview of the Logistica programming language. Logistica is a lexically-scoped functional programming language which is designed to be used in the implementation of
various kinds of deductive reasoning processes. At its simplest level, it looks like Scheme, but differs in that Logistica symbols may have more than one definition, while those in Scheme may have only one. This is helpful because many symbols have several axiomatic properties, and it is advantageous to represent each property in a usefully modular fashion by using a separate definition for each property. For example, the identity law for "&", which is (...a & T & ...b) if and only if (...a & ...b), can be coded as one axiom, and the zero law for "&", which is (...a & F & ...b) if and only if F, can be coded as another axiom. In the case in which no symbol is defined more than once, then Logistica is extremely similar to Scheme. Logistica allows multiple definitions of all of its symbols, including predicates, functions, and logical connectives, so that expressions can have multiple values. Cut operations similar to Prolog's cut operation are incorporated in the control facilities in order to deal with multiple definitions, so that if one rule is used, subsequent rules can be cut, or skipped. Since Logistica can specify arbitrary deductive systems by representing the properties of symbols as backtrackable rewrite rules which are associated with the symbols themselves, it is independent of any specific logic. Therefore, it can provide the basic tools for many kinds of deduction. Logistica also leverages conceptual problem-solving effort, in that it is a form of meta-language in which users can embed their own problem-solving language, which can function most usefully at a higher level than most procedural languages (Leasure, 1992). Problems can
be expressed in terms which facilitate their solution, rather than obscuring it. The situation is analogous to programming a computer in Cobol in order to run a payroll, versus doing the same task in machine language.

Logistica is more properly a rewrite system than a functional programming language, although it has many characteristics of functional languages. The theory of rewrite systems is well developed. Briefly stated, rewrite systems are sets of directed equations which are used in computation by replacing subexpressions of a given formula, as long as it is possible to do so. If the computations of a rewrite system always terminate in a unique normal form for equal terms, then the system serves as a decision procedure for the particular equational theory defined by the rules (Dershowitz, 1989).

The technology available in Logistica provides a dramatic simplification of the program control structure which would be extremely complex if it were written in one of the more familiar procedural languages. The control structure could be said to be data-driven, since a given simplification rule is invoked only if it can usefully rewrite the data expression at the time that the expression is checked by the rule. Logistica enhances program precision, perspicuity, and elegance.

The process of arranging or structuring the axioms is much like teaching algebra to a student who has no mathematical intuition concerning the order of application of mathematical rules. For example, here are five rules of exponents and two
related definitions from the introductory sections of many college algebra texts: For $x$ and $y$ each a real number and each of $m$ and $n$ an integer, and the usual restriction on division by 0,

1. $(x \exp m) \ast (x \exp n) = x \exp (m + n)$
2. $(x \exp m) \exp n = x \exp (m \ast n)$
3. $(x \exp m) / (x \exp n) = x \exp (m - n)$
4. $(x \ast y) \exp m = (x \exp m) \ast (y \exp m)$
5. $(x / y) \exp m = (x \exp m) / (y \exp m)$

Def. (1) $x \exp (-1) = 1 / x \exp 1$

Def. (2) $x \exp 0 = 1$, if $x$ is not 0

Simply knowing the rules without having some idea of their most expeditious application sequence can produce unnecessarily tedious solutions. For example, if def. (1) is applied before rule (4) in the simplification of $((x \exp -1) \ast (y \exp -1)) \exp -1$, many more steps will be required than if rule (4) were applied initially, even though both solutions are valid. Similarly, the Logistica program rules should be written and ordered so that efficiency is enhanced. Some specific efficiency considerations are discussed in the End Results section of this paper, with respect to the differences among versions I, II, and III of the program. Proving program optimality is a significant research problem.

This program includes symbol definitions for standard real numbers, and non-standard infinitesimal and infinite numbers. The rule base is defined so that an expression representing, for example, the derivative of a polynomial can be entered in prefix notation. If desired, the Logistica trace routine will then show
each step in the simplification procedure, including which rule was
used in it. The final result exhibits the derivative in symbolic
form, again in prefix notation. A typesetting or reformatting
module to convert the prefix notation to infix form was not
included, and is beyond the scope of this project.

The user interface is interactive. The input is typed in as
in the following example, and the result is normally returned to
the screen, with the trace of the simplification steps optional:
To get the derivative of \( x \exp 2 \) using the non-standard definition,
\((\text{real } (\text{deri } x \ (\exp x \ 2)))\) is entered. In due course, \((* \ 2 \ x)\), the
prefix form of \( 2 * x \), is returned. If an expression cannot be
completely simplified, the program returns the partly simplified
result.

The (modified) BNF syntax of the input and output language for
finding derivatives is as follows: Using | for "or", and the *
suffix to mean 0 or more instances of the starred item,

expression ::= number | variable | (operator expression)*
operator ::= der | deri | / | * | + | - | expt.

Here, der is the symbolic derivative, deri is the non-standard
derivative, /, *, +, and - are the usual arithmetic operators, and
expt is a binary exponentiation operator, as in \( x^x = (\text{expt } x \ 2) \).
Environment

This program has been run on a DEC Alpha computer. It will run on anything that will support MIT Scheme, since the language employed is Logistica, and Logistica is run as an "upper layer" on top of MIT Scheme. Input and output is to the screen, with printed output optional.
Procedure

The major developmental thrust of this program has been the construction of the rule base by which expressions presented to the program are evaluated. The theoretical basis for the implementation of this rule base has been stated more explicitly than in the usual writeup for a procedural language program, since part of the program’s reason for being was to learn to develop ways of implementing those results. Most of the theoretical issues have been discussed in the Background and Rationale, and the Narrative sections of this paper. The program required many axioms and lemmas of the kinds discussed in those sections to be implemented, and these implementations are described below.

There are no formal modules as such in the program, so that it has a relatively "flat" structure. However, it has a conceptually modular structure which has been shown graphically in Figure 1. This hierarchical structure has been indicated by drawing modules which are further from the underlying machine hardware over the closer program layers. The diagram shows Logistica as the base layer, and it is actually implemented over Scheme. The algebraic simplifier "as such" comes next, and then the two major calculus program sections are at the same level on top of it. All of these last three sections are implemented in the program’s rule base. Modular information flows are shown by arrows. Of course, these ultimately arrive at machine hardware level, and are processed back up through whatever layers exist in the output structure, the top
Entities on the same horizontal level can interact with each other.

Figure 1
layer of which is really in Logistica. Since the bottom layer of real interest in this program is in Logistica, all of the information processed eventually arrives there, for purposes of the diagram. The integral, propositional logic, and predicate logic sections in the diagram are proposed for implementation in the future.

The rules implemented in this program are equivalence statements, in a sense. They are rewrite rules which state in effect that an input pattern in a rule which is matched in the expression being evaluated is to be rewritten in an equivalent form. There are many criteria by which the correctness and utility of an algebraic simplifier can be judged. One of the criteria often employed for a rewrite system is that it return a unique normal form if it is given equivalent input expressions. However, there is no decision procedure for our algebraic (number) system, since Godel showed that there are true propositions in the system which could not be proved. Nevertheless, if the system were restricted sufficiently, a decision procedure is available (Tarski, 1951). Hence, in a rewrite system for mathematics such as this one, equivalent input expressions could have outputs which are not identical. In this simplifier, then, the most frequently applied criterion of correctness was that a given rule be justified by being mathematically or logically sound. Almost the same thing could be said regarding criteria for judging the utility of the output. That is, if a complex expression were simplified to the usual standards of, say, algebra or calculus, it was held to have

20
been usefully simplified. There is a sense in which this criterion could be viewed as dealing with a mathematical aesthetic which is a sociological convention. For example, expressions are often said to be mathematically simplified if they are rewritten so that they are easier to read. If this seems to be painfully obvious, remember that a computer has very little mathematical intuition, and has to be told what to do explicitly. Unless otherwise indicated, the following discussion applies to Version I of the program.

The first statement in the program resets the environment to a "blank tablet" condition. The next one enables a program rule to fail on a miswritten expression, and perhaps pass the offending expression to another rule, rather than shutting down and kicking the program out of the system. The next step (in a logical, if not necessarily a syntactical sense) is the redefinition of variables and operators so that they are accessible in the user environment, and still properly accessible to the system for its normal operations. For example, if a normal operator such as "+" were defined in the user environment, as is done in this program in the axioms, then it would shadow "+" in the system environment. The system level definition of "+" is brought to the user level by the "(define s+ +)" statement prior to the "(define + _+)" statement. The latter statement defines "+" as symbolic, and is used to return the expression unsimplified in case no other rules apply.

The next section is the algebraic simplifier proper, and it begins with rules by which expressions with the "+" operator are
evaluated. In the following discussion, variables may be bound to constants or may remain symbolic variables, depending upon the expression being evaluated. The addition rule section and its justifications will be described in some detail, and the following sections will be described functionally. The next rule after the definition above is, "(defaxiom! +no-args (+) 0)" , which handles the "+" operator with no operands by redefining it as "0" , so that it makes no arithmetic changes to a symbol string. Here, the "+" is the input expression, and "0" is the output. The "+" with no arguments is the arithmetic equivalent of "0 + 0" , so the rule effectively states that "0 + 0 = 0" . The next rule, called "+x" , defines "+" with one operand as that operand. It is a Logistica statement of "x + 0 = x" . Then, the additive identity axiom of arithmetic is more explicitly incorporated in the third rule, "+ident" in which 0 is "erased" from a string of 0 or more addends in which it occurs. Its input expression is coded as "(+ @a 0 @b)" and its output as "(+ @a @b)" . The "@" operator with the variable following it stands for a sequence of terms. The justification and utility of these rules depend upon those of the mathematical axiom that they encode. The first two are the base cases of the recursive calls to other applicable simplification rules. The fourth rule, "(defaxiom! +assoc (+ @a(+ @b) @c)(+ @a @b @c))" , defines an associative grouping of addends into the default system association (right associative). This is more compatible with the language implementation, and it is justified by the (possibly iterated) associative axiom of arithmetic. The next rule,
"(defaxiom! +nums (+ @a(!app number? x)@b(!app number? y)@c)(+ @a(s+ x y)@b @c)")", combines constants which occur in a string of addends. The "+app number? x" expression applies the number (constant) predicate to x, and if it is true, as well as in the corresponding "y" expression, then x and y are added with the primitive system "s+" operator, as defined in the program definition section. This rule can be justified in any particular case by the (possibly repeated) application of the commutative and associative arithmetic axioms. The "inverse1" rule, the pattern-matching parts of which are "(+ @a x @b(-x)@c)" is rewritten as "(+ @a @b @c)", eliminates any pair of additive inverses occurring in "+x...-x)" order from the addend string. The next rule eliminates them if they occur in the reverse (commuted) order. These rules are justified primarily by the commutative and additive inverse arithmetic axioms. They are written in a generalized Logistica format, and with some input expressions, would require other arithmetic axioms for justification, such as the additive associative axiom. Note that there is no explicit commutative rule encoded, since its inclusion would cause the computer to try to produce infinitely many operand position switches. So, rules often occur in left and right pairs throughout the program. The same approach, respectively, is taken in "inverse3" and "inverse4" in more general form. The first is "(defaxiom! +inverse3 (+ @a(*(-x)y)@b(!test(* @z)(=(* x @y)(* @z)))@c)(+ @a @b @c))". That is, these rules handle sequences of terms which are additive inverses just as "inverse1" and "inverse2" did for the single symbol cases.
The next rule, "(defaxiom! +combine1 (+ @a x @b x @c) (+ @a(* 2 x)@b @c))" combines two instances of the same variable into a multiple of two times the variable. This is just simply "x + x = 2x" mathematically, and Logistica will do this when the rule is written as this one is, even if each of the x’s is buried in strings of addends. The second combination rule, "(defaxiom! +combine2 (+ @a x @b(* ( !app number? n)x)@c) (+ @a(*(+ n 1)x)@b @c))" similarly combines n of the same variables with one more instance of it. "Combine3" is the commuted form of "combine2". "Combine4" is the last rule in the addition section. It is "(defaxiom! +combine4 (+ @a (!or(*(!app number? n)x) (* @x)) @b (!test(!or(*(!app number? m)y)(* @y))(=(* @x)(* @y)))@c) (+ @a(*(+ (if (def? 'n)n 1)(if(def? 'm)m 1))@x)@b @c)". This is a way to check for the equality of a pair of sequences of factors and return a simplified result if they are found. For example, n*x +...+m*y would be returned as (n+m)x, if n and m are each integers and if x = y.

The comparatively short rule section for negatives then follows. The "-x" rule turns -x into the product of -1 and x. The "-sum" rule distributes the "-" sign into a variable plus a string. The "-binary" rule is this program’s version of the algebraic "add the opposite" rule.

The multiplication section is analogous to the respective addition section in the first five rules, except for the inclusion of a rule to define the product of a string times "0" times a string as "0". The "distrib1" and "distrib2" rules are implementations of the usual algebraic distributive axiom for
multiplication over addition, with the multiplier commuted with respect to the first rule in the second. The "neg-to-left" rule pulls the negative part of a single symbol to the left of the expression as a negative one times the expression. The "num-to-left" rule acts similarly, in that it pulls constants to the left side of an expression. The "nums" rule multiplies a pair of constants which may occur in a string of factors. The "combine1" rule in this section turns two instances of the same variable in a string of factors into the square of the variable. The "combine2" rule rewrites an instance of a variable and an instance of the variable to the nth power in a string of factors as the variable to the n plus first power. "Combine3" is the commuted form of "combine2". "Combine4" rewrites a variable to the mth power and the same variable to the nth power which may occur in a string of factors as the variable to the m plus nth power.

The next section calculates factorials with the fact rule, then binary coefficients with the next two rules, and finally, the binary expansion. These rules were written in this form because they make the calculation of the non-standard derivative run in linearly increasing time when calculating the derivative of x to the n as n increases through the integers. The potentially more intuitive recursive form of implementation was more costly, and appeared to run in exponential time in one experiment.

The only division rule is the "quotient-to-product" rule which returns the quotient of x and y as the product of x and y to the negative first power.
The next section deals explicitly with exponential expressions. The first rule rewrites one to the (variable) xth power as one. The next rewrites x to the first as x. Then the following rule rewrites zero to the xth power as zero. The fourth rule returns one if given x to the zero power. The "x^sum" rule separates x to the a plus a string into the factors x to the a times x to the string. The "product^n" rule rewrites the product of x and a string, all to the nth power, as x to the nth times the string to the nth. The "x^m^n" rule redefines x to the mth all to the nth as x to the m times nth power. The "n^m" (last) axiom in this section causes the system to return the result of a constant with a constant exponent, as a constant.

The next rule is the single rule which calculates the non-standard derivative. It is simply the Logistica version of the non-standard formulation of the derivative. Its output is a symbolic expression representing the derivative which may include non-standard infinitesimal numbers, even after algebraic simplification.

The next section extracts the real or standard number component from the non-standard expression. The first rule returns the real number x as x. The next returns zero which is the real (standard) part of the non-standard expression, zero plus an infinitesimal, coded here as dx. The "rsumx" rule rewrites the real part of the sum of x and a string as the real part of x plus the real part of the string. The "rdifx" rule rewrites the real part of the difference of x and a string as the real part of x
minus the real part of the string. The "rproz" rule acts similarly upon the call to extract the real part of the product of \( x \) and a string, in that it returns the product of the real part of \( x \) times the real part of the string. The last rule in this section, "rexpx", rewrites the real part of \( x \) to the \( y \)th power as the real part of \( x \) to the real part of \( y \)th power.

The most efficient ordering of the rules depends upon the input. The rules are encountered from the bottom up, and with the cut operator, the "!" following "defaxiom", when a match is found, no more trials are made with that exact form of the input expression. Fewer failed trial matches are required if the most frequently successful rules are tried first, so optimal rule order depends upon the rule use statistics. This topic has been explored extensively by Logistica researchers.

In the program environment, when we write a statement directing an input expression to the non-standard derivative section, the keyword "deri" will be written into an expression. Similarly, the keyword "der" is used for symbolic ("algebraic") differentiation. These are good examples of how a unique problem-solving language begins to be built by using the underlying Logistica as a meta-language.

The symbolic derivative section begins with a rule for the derivative of a constant with respect to \( x \), which returns zero. Then a rule for the derivative of \( y \) with respect to \( x \) returning zero follows. The third rule returns one for the derivative of \( x \) with respect to \( x \). The fourth rule rewrites the derivative of a
negative as the negative of the derivative, and the fifth rewrites the derivative of a sum as the sum of the derivatives. The sixth rule encodes the standard calculus derivative of a product. The "log" rule evaluates the derivative of the natural log, rewriting the derivative of log a with respect to x as the derivative of a with respect to x all divided by a. The next rule evaluates the derivative of e to the u (actually coded as y), and the next three evaluate the derivatives of sin u, cos u, and tan u, respectively, with respect to x. The following rule evaluates the derivative of a to the bth power with respect to x.

The next section has several rules for equality. The first checks the equality of the addition string "(+ @a x @b)" against "(+ @c y @d)", and tests whether x = y. If x = y, it returns "(+ @a @b) = (+ @c @d)". This can recursively simplify the items which are equated, if there is a successful match. The next axiom similarly checks the equality of strings of factors, and returns the conclusion from "(* @a x @b) = (* @c x @d)" that either "(* @a @b) = (* @c @d)" or "x = 0". The next rule "=nums" checks two constants and returns "false", which works only because a previously encountered rule in the set would have checked for equality and failed if this rule were encountered. The following rule, "=factor2a" replaces "m = n * @a", where each of m and n is a constant, with "m/n = @a". The rule "=factor2b" is the commuted form of the "a" rule. The next rule, "const", replaces "x = (+ @a y @b)" if b is a constant, with "(-x y) = (+ @a @b)". The last axiom, "=t", establishes the equality of a variable x with itself,
by the reflexive property of equality.

The last executable program statement establishes "def?" in terms of the system primitive "defined". Example output follows, with partial traces of the rules used. These and other results for program versions I, II, and III are discussed in the End Results section of this paper.
End Results

There are three versions of the program code in the appendix. They are entitled Version I, Version II, and Version III, as stated in previous sections. Several example traces follow the executable code in Version I. The first is an example of the "+" input with no arguments, and shows the successive attempts to match the "+" rules in the bottom up order in which they are encountered. The first thirteen arrows pointing to the right show the input to the rule, and the final one pointing to the left shows the "0" returned from the successful match.

The next example shows a partial trace of a call on the extractors which return the real part of a non-standard expression. The expression "x + dx", with x real and dx infinitesimal, is separated into two expressions, and the real part of each, x and 0 respectively, is simplified to x.

The third example is a partial trace of only the successful matches on the call to take the non-standard derivative of x with respect to x. This example returns the expected value of 1 without needing to use the real part extractors, since all of the infinitesimals (the dx's) are algebraically simplified out of the final expression.

The two sections following the traces are screen output from untraced calls. The discussion of comparative performance times is not intended to represent a controlled scientific experiment with statistically significant results, but does roughly indicate what
a user might expect under similar circumstances. The first section is from a series of calls to calculate the non-standard derivative of \( x^2 \), or \((\text{expt } x \ 2)\), then \( \text{expt } x \) with exponents 3, 4, 5, and 6, respectively. The lines beginning after the command line, which starts with ">", are the output. The printer truncated the output if it ran to more than one line, but the next section, in which the real part of the same output is extracted, can be seen to produce the correct results. The CPU, garbage collection, and overall ("real") times are printed immediately below the results returned. It can be seen that the time cost for extracting the real part of the derivative is a small percent of the total CPU time. Also, these times can be compared with those at the end of Version III. Those for Version III are for two trials on the same input as in the first section mentioned above for Version I. Version III is the same as Version I except that it uses the recursive implementation of the factorial calculation in its evaluation of binomial coefficients, while Version I uses an iterative implementation. The results are shown in the following table:

<table>
<thead>
<tr>
<th>Version I--Exponent:</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU time (approx., sec):</td>
<td>2.9</td>
<td>5.0</td>
<td>7.4</td>
<td>10.9</td>
<td>14.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version III--Exponent:</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU time (2nd run):</td>
<td>4.5</td>
<td>7.6</td>
<td>11.5</td>
<td>16.2</td>
<td>21.9</td>
</tr>
</tbody>
</table>
Version II is the same as Version I, except that the identity and associative rules for both addition and multiplication are placed at the bottom of their respective sections, where they will be encountered first. Since they are frequently "matched", this ordering can speed up evaluation on a large class of input expressions, but this is dependent upon the input. The traces following version II show the modified order of evaluation.

This program automates the simplification of many of the standard classes of elementary algebraic expressions, and several examples have been shown. Several ways to implement algebraic simplification rules in Logistica have been explored. The program implements the calculation of derivatives for polynomials symbolically, using algebraic definitions and using the non-standard definition of the derivative as well.

I have learned a great deal about non-standard analysis, functional programming, logic, natural deduction and resolution theorem provers, and rewrite rule systems. I consider this to have been a very worthwhile educational experience.
Bibliography


Appendix
Version I
(define (single-sym? x) (if (symbol? (make-quote x)) #t #f))

(define s* *) (define s* _*)
(define s+ +) (define s+ _+)
(define s/ /) (define s/ _/)
(define s- -) (define s- _-)
(define s= =) (define s= _=)
(define sin sin) (define sin _sin)
(define scos cos) (define cos _cos)
(define stan tan) (define tan _tan)
(define slog log) (define log _log)
(define sexp exp) (define exp _exp)
(define sexpt expt) (define expt _expt)

(define sec _sec)

(define x _x)
(define dx _dx)
(define wx _wx)
(define y _y)
(define a _a)
(define b _b)
(define k _k)
(define m _m)
(define n _n)
(define r _r)

(defaxiom! +no-args (+) 0)
(defaxiom! +x (+ x) x)
(defaxiom! +ident (+ @a 0 @b) (+ @a @b))
(defaxiom! +assoc (+ @a (+ @b @c) (+ @a @b @c))
(defaxiom! +nums (+ @a(!app number? x)@b(!app number? y)@c)
(+ @a(+ x y)@b @c))
(defaxiom! +inverse1 (+ @a x @b(- x)@c) (+ @a @b @c))
(defaxiom! +inverse2 (+ @a(- x)@b x @c) (+ @a @b @c))
(defaxiom! +inverse3 (+ @a(*(- x)y)@b(!test(* @z)(* x @y)(* @z)))@c
(+ @a @b @c))
(defaxiom! +inverse4 (+ @a(* @z)@b(!test(*(- x)y)(* @x)(* @y)))@c
(+ @a @b @c))
(defaxiom! +combine1 (+ @a x @b x @c) (+ @a(* 2 x)@b @c))
(defaxiom! +combine2 (+ @a x @b(*(!app number? n)x)@c)
(+ @a(*(+ n 1)x)@b @c))
(defaxiom! +combine3 (+ @a(*(!app number? n)x)@b x @c)
(+ @a(*(+ n 1)x)@b @c))
(defaxiom! +combine4 (+ @a (!or(*(!app number? n)x)(* @x))@b
(!test(!or(*(!app number? m)y)
(* @y))(=(* @x)(- @y))))@c
(+ @a(+if (def? 'n)n 1)(if (def? 'm)m 1))@x@b @c))

(defaxiom! -x (- x)(* -1 x))
(defaxiom! -sum (-(+ x @a))(- x)(- (+ @a)))
(defaxiom! -binary (- x y)(+ x (- y)))
(defaxiom! *no-args (*) 1)
(defaxiom! *x (* x) x)
(defaxiom! *0 (* @a 0 @b) 0)
(defaxiom! *ident (* @a 1 @b) (* @a @b))
(defaxiom! *assoc (* @a(* @b)@c) (* @a @b @c))
(defaxiom! distrib1 (* (+ @(do (i 1 m) ((ref i x))) n)
  (+ @(do (i 1 m) (* n ((ref i x))))) )
(defaxiom! distrib2 (* n (+ @(do (i 1 m) ((ref i x))))
  @(do (i 1 m) (* n ((ref i x))))))
(defaxiom! *neg-to-left (* @a(- x)@b)(* -1 @a x @b))
(defaxiom! *num-to-left (* x @a(!app number? y)@b)(* y x @a @b))
(defaxiom! *nums (* @a(!app number? x)@b(!app number? y)@c)
  (* (s* x y)@a @b @c))
(defaxiom! *combine1 (* @a(!app single-sym? x)@b x @c)
  (* @a(b(expt x 2) @c))
(defaxiom! *combine2 (* @a(!app single-sym? x)@b(expt x n)@c)
  (* @a(b(expt x(+ n 1))@c))
(defaxiom! *combine3 (* @a(expt(!app single-sym? x) n)@b x @c)
  (* @a(b(expt x( + n 1))@c))
(defaxiom! *combine4 (* @a(expt x(!app number? n))@b
  (expt x(!app number? m))@c)
  (* @a(expt x(+ n m))@b @c))

(defaxiom! fact (fac n) (bin1 n 1))
(defaxiom! binc1 (bin1 n r) (* @(do (i (+ r 1) n) i)))
(defaxiom! bincof (binc n r) (/ (bin1 n r) (fac (- n r))))

(defaxiom! binxp (expt (+ x y) n) (+ @(do(r 0 n)
  (* (binc n r) (expt x (- n r)) (expt y r))))))

(defaxiom! quotient-to-product (/ x y)(* x(expt y -1)))

(defaxiom! 1^x (expt 1 x)1)
(defaxiom! x^1 (expt x 1)x)
(defaxiom! 0^x (expt 0 x)0)
(defaxiom! x^0 (expt x 0)1)
(defaxiom! x^sum (expt x (+ a @b))*((expt x a)(expt x (+ @b))))
(defaxiom! product^n (expt(* x @a)n)*((expt x n)(expt(* @a)n)))
(defaxiom! x^m*n (expt(expt x m)n)(expt x(* m n)))
(defaxiom! n^m (expt(!app number? n)(!app number? m))(sexpt n m))

(defaxiom! derinx (deri x (^f x)) (/ (- (^f (+ x dx)) (^f x)) dx))

(defaxiom! rx (real x) x)
(defaxiom! rdx0 (real (ltest x (equal? x dxx))) 0)
(defaxiom! rsumx (real (+ x @y)) (+ (real x) (real (+ @y))))
(defaxiom! rdiffx (real (- x @y)) (- (real x) (real (- @y))))
(defaxiom! rprox (real (* x @y)) (* (real x) (real (* @y))))
(defaxiom! rexpx (real (expt x y)) (expt (real x) (real y)))

(defaxiom! const (der x (!app number? c)) 0)
(defaxiom! var (der x (!app single-sym? y)) 0)
(defaxiom! samevar (der x x) 1)
(defaxiom! - (der x (- a))(-(der x a )))

(defaxiom! +(der x (+ a @s)))
+(+(der x a)(der x (+ @s)))
(defaxiom! *(der x (* a @s)))
+(*((der x a)@)/* a(der x (* @s))))
(defaxiom! log (der x (log a))//(der x a a))
(defaxiom! exp (der x (exp y))//(* (exp y)(der x y)))
(defaxiom! sin (der x (sin a))//(* (cos a)(der x a)))
(defaxiom! cos (der x (cos a))//(-(* (sin a)(der x a))))
(defaxiom! tan (der x (tan a))//(*expt(sec a)2)(der x a))
(defaxiom! expt (der x (expt a b))
+(* b(expt a(- b 1))(der x a))(* (expt a b)(log a)(der x b))))
(defaxiom! dist (der x (* @a(! and(+ y @b)(^t x))@c))
+(der x (+(* y @a @c)(* @a(+ @b @c)))))

(defaxiom! =sum (=(+ @a x @b)(+ @c(!test y(= x y)@d))
+(* @a @b)(+ @c @d)))
(defaxiom! =product (=(* @a x @b)(* @c x @d))
+(* @a @b)(* @c @d)))(= x 0))
(defaxiom! =nums (=(!app number? x)(!app number? y)) #f)
(defaxiom! =factor2a =(!app number? n)(*(!app number? m)@a))
+(* @a)(s/ n m))
(defaxiom! =factor2b =(*(!app number? m)@a)(*(!app number? m))
+(* @a)(s/ n m))
(defaxiom! =const = x(+ @a(!app number? y)@b))
+(- x y)(+ @a @b)))
(defaxiom! =t (= x x) #T)

(defmacro '(def ,x) '(defined? ,x (cons(car(the-environment)))'())))

#| > (+)
+combine4 0 1 .>(+)
+combine3 1 1 .>(+)
+combine2 2 1 .>(+)
+combine1 3 1 .>(+)
+inverse4 4 1 .>(+)
+inverse3 5 1 .>(+)
+inverse2 6 1 .>(+)
+inverse1 7 1 .>(+)
+nums 8 1 .>(+)
+assoc 9 1 .>(+)
+ident 10 1 .>(+)
+x 11 1 .>(+)
+no-args 12 1 .>(+)
+no-args 12 1 .<0
0
|
#|
> (real (+ x dx))
rexpx 0 1 .>(real (+ x dx))
rlenx 1 1 .>(real (+ x dx))
rdifx 2 1 .>(real (+ x dx))
repxx 3 1 .>(real (+ x dx))
rexpx 4 2 ..>(real dx)
rlenx 5 2 ..>(real dx)
rdifx 6 2 ..>(real dx)
ratsx  7  2  ..>(real dx)
rdx0  8  2  ..>(real dx)
rdx0  8  2  ..<0
rexpx 9  2  ..>(real x)
rdpx  10  2  ..>(real x)
rdix  11  2  ..>(real x)
rsumx 12  2  ..>(real x)
rdx0  13  2  ..>(real x)
rnx  14  2  ..>(real x)
x  14  2  ..<x
rsumx  3  1  ..<x
x  #
@

# | (deri x x)

|   derinx  0  1  ..>(deri x x)
|   -binary  14  2  ..>((- (+ x dx) x)
|   -x  17  3  ..>(- x)
|   -x  17  3  ..<(* -1 x)
|   +assoc  41  3  ..>((+ (+ x dx) (* -1 x))
|   +combine2  44  4  ..>((+ x dx (* -1 x))
|   +nums  53  5  ..>(+ -1 1)
|   +ident  64  6  ..>(+ 0)
|   +no-args  77  7  ..>(+)
|   +no-args  77  7  ..<0
|   +ident  64  6  ..<0
|   +nums  53  5  ..<0
|   *0  89  5  ..>(* 0 x)
|   *0  89  5  ..<0
|   +ident  100  5  ..>(+ 0 dx)
|   +x  112  6  ..>(+ dx)
|   +x  112  6  ..<dx
|   +ident  100  5  ..<dx
|   +combine2  44  4  ..<dx
|   +assoc  41  3  ..<dx
|   -binary  14  2  ..<dx
|   quotient-to-pro  113  2  ..>(/ dx dx)
|   *combine2  116  3  ..>(* dx (expt dx -1))
|   +nums  125  4  ..>(+ -1 1)
|   +ident  136  5  ..>(+ 0)
|   +no-args  149  6  ..>(+)
|   +no-args  149  6  ..<0
|   +ident  136  5  ..<0
|   +nums  125  4  ..<0
|   *ident  160  4  ..>(* 1)
|   *no-args  174  5  ..>(*)
|   *no-args  174  5  ..<1
|   *ident  160  4  ..<1
|   *combine2  116  3  ..<1
|   quotient-to-pro  113  2  ..<1
|   derinx  0  1  ..<1
| 1

| #

# |
> (deri x (expt x 2))
(+ (* 2 x) dx)

[ CPU: 2.867000   GC: .4330000   Real: 5.439   ]

> (deri x (expt x 3))
(+ (* 3 (expt x 2)) (* 3 dx x) (expt dx 2))

[ CPU: 5.048999   GC: .7180000   Real: 10.541   ]

> (deri x (expt x 4))
(+ (* 4 (expt x 3)) (* 6 dx (expt x 2)) (* 4 (expt dx 2) x) (expt dx 3))

[ CPU: 7.384       GC: 1.033     Real: 15.817   ]

> (deri x (expt x 5))
(+ (* 5 (expt x 4)) (* 10 dx (expt x 3)) (* 10 (expt dx 2) (expt x 2)) (* 5 (expt dx 3)))

[ CPU: 10.91699    GC: 1.566000  Real: 24.171   ]

> (deri x (expt x 6))
(+ (* 6 (expt x 5)) (* 15 dx (expt x 4)) (* 20 (expt dx 2) (expt x 3)) (* 15 (expt dx 3)))

[ CPU: 14.53399    GC: 2.183     Real: 32.779   ]

#

#

> (real (deri x (expt x 2)))

(* 2 x)

[ CPU: 2.915999    GC: .4670000  Real: 12.059    ]

> (real (deri x (expt x 3)))

(* 3 (expt x 2))

[ CPU: 5.481999    GC: .8509999  Real: 11.745    ]

> (real (deri x (expt x 4)))

(* 4 (expt x 3))

[ CPU: 8.233999    GC: 1.166000  Real: 17.794    ]

> (real (deri x (expt x 5)))

(* 5 (expt x 4))

[ CPU: 12.59999    GC: 1.766000  Real: 27.867    ]

> (real (deri x (expt x 6)))

(* 6 (expt x 5))

[ CPU: 17.18399    GC: 2.649999  Real: 39.141    ]

> |#
(define (single-sym? x) (if (symbol? (make-quote x)) #t #f))

(define * *)
(define + +)
(define / ;)
(define - -)
(define = =)
(define sin sin)
(define cos cos)
(define tan tan)
(define log log)
(define exp exp)
(define expt expt)

(define sec _sec)

(define x _x)
(define dx _dx)
(define wx _wx)
(define y _y)
(define a _a)
(define b _b)
(define k _k)
(define m _m)
(define n _n)
(define r _r)

(defaxiom! +no-args (+ 0))
(defaxiom! +x (+ x x))
(defaxiom! +nums (+ @a (!app number? x)@b (!app number? y)@c)
 (+ @a (+ x y)@b @c))
(defaxiom! +inverse1 (+ @a -x @b (- x)@c)
 (+ @a @b @c))
(defaxiom! +inverse2 (+ @a (- x)@b x @c)
 (+ @a @b @c))
(defaxiom! +inverse3 (+ @a (* -x)@b (!test (* @z) (= (* x @y) (* @z)))@c)
 (+ @a @b @c))
(defaxiom! +inverse4 (+ @a (* @z)@b (!test (* -x)@y) (= (* @x @y) (* @z)))@c)
 (+ @a @b @c))
(defaxiom! +combine1 (+ @a x @b x @c)
 (+ @a (* 2 x)@b @c))
(defaxiom! +combine2 (+ @a x @b (!app number? n)@x)
 (+ @a (+ n 1)@x)@b @c))
(defaxiom! +combine3 (+ @a (!or (!app number? n)@x) (* @x)@b
 (!test (!or (!app number? m)@y)
 (* @y))@x)@b @c))
 (+ @a (+ (if (def? 'n) n 1) (if (def? 'm) m 1) @x)@b @c))
(defaxiom! +ident (+ @a 0 @b)
 (+ @a @b))
(defaxiom! +assoc (+ @a (+ @b)@c)
 (+ @a @b @c))

(defaxiom! -x (- x) (* -1 x))
(defaxiom! -sum (- (+ x @a) (+ -x) (- (+ @a)))
(defaxiom! -binary (- x y) (+ x (- y)))
(defaxiom! *no-args (*)1)
(defaxiom! *x (* x)x)
(defaxiom! *0 (* @a 0 @b)0)
(defaxiom! distrib1 (* (+ @(do (i l m) (!ref i x))) n)
  (+ @(do (i l m) (* n (!ref i x))))))
(defaxiom! distrib2 (* n (+ @(do (i l m) (!ref i x)))
  (+ @(do (i l m) (* n (!ref i x))))))
(defaxiom! *neg-to-left (* @a(- x)@b)(* -1 @a x @b))
(defaxiom! *num-to-left (* x @a(!app number? y)@b)(* y x @a @b))
(defaxiom! *nums (* @a(!app number? x)@b(!app number? y)@c)
  (* (s x y)@a @b @c))
(defaxiom! *combine1 (* @a(!app single-sym? x)@b x @c)
  (* @a @b(expt x 2) @c))
(defaxiom! *combine2 (* @a(!app single-sym? x)@b(expt x n)@c)
  (* @a @b(expt x(+ n l))@c))
(defaxiom! *combine3 (* @a(expt(!app single-sym? x) n)@b x @c)
  (* @a @b(expt x(+ n l))@c))
(defaxiom! *combine4 (* @a(expt x(!app number? n))@b
  (expt x(!app number? m))@c)
  (* @a(expt x(+ n m))@b @c))
(defaxiom! *ident (* @a 1 @b)(* @a @b))
(defaxiom! *assoc (* @a(* @b)@c)(* @a @b @c))

(defaxiom! fact (fac n) (bin1 n 1))
(defaxiom! bincl (bin1 n r) (* @(do (i (+ r l) n) i)))
(defaxiom! bincof (bin1 n r) (/ (bin1 n r) (fac (- n r))))
(defaxiom! binxp (expt (+ x y) n) (+ @(do(r 0 n)
  (* (binc n r) (expt x (- n r)) (expt y r)))))

(defaxiom! quotient-to-product (/ x y)(* x(expt y -1)))

(defaxiom! 1^x (expt 1 x)1)
(defaxiom! x^1 (expt x 1)x)
(defaxiom! 0^x (expt 0 x)0)
(defaxiom! x^0 (expt x 0)1)
(defaxiom! x^sum (expt x (+ a @b))(*(expt x a) (expt x (+ @b))))
(defaxiom! product^n (expt(* x @a)n)(* (expt x n) (expt(* @a)n))
(defaxiom! x^m^n (expt(expt x m)n) (expt x(* m n))
(defaxiom! n^m (expt(!app number? n) (!app number? m)) (sexpt n m))

(defaxiom! derinx (deri x (^f x)) (/ (- (^ f (+ x dx)) (^ f x)) dx))

(defaxiom! rx (real x) x)
(defaxiom! rdx0 (real (ltest x (equal? x dx))) 0)
(defaxiom! rsumx (real (+ x @y)) (+ (real x) (real (+ @y))))
(defaxiom! rdiffx (real (- x @y)) (- (real x) (real (- @y))))
(defaxiom! rprox (real (* x @y)) (* (real x) (real (* @y)))
(defaxiom! repxp (real (expt x y)) (expt (real x) (real y)))

(defaxiom! const (der x (!app number? c))0)
(defaxiom! var (der x (!app single-sym? y))0)
(defaxiom! samevar (der x x) 1)
(defaxiom! - (der x (- a)) (- (der x a))
(defaxiom! + (der x (+ a @s)) (+ (der x a) (der x (+ @s)))))
(defaxiom! * (der x (* a @s)) (+ (* (der x a) @s) (* a (der x (* @s))))))
(defaxiom! log (der x (log a)) (/ (der x a) a))
(defaxiom! exp (der x (exp y)) (* (exp y) (der x y)))
(defaxiom! sin (der x (sin a)) (* (cos a) (der x a)))
(defaxiom! cos (der x (cos a)) (* (- (sin a) (der x a))))
(defaxiom! tan (der x (tan a)) (* (expt (sec a) 2) (der x a)))
(defaxiom! expt (der x (expt a b))
  (+ (* b (expt a (- b 1)) (der x a)) (* (expt a b) (log a) (der x b))))
(defaxiom! dist (der x (* @a (!and (+ y @b) (^ t x)) @c))
  (der x (+ (* y @a @c) (* @a (+ @b @c)))))

(defaxiom! =sum (= (+ @a x @b) (+ @c (!test y (= x y)) @d))
  (= (+ @a @b) (+ @c @d)))
(defaxiom! =product (= (a x @b) (* @c x @d))
  (or (= (a x @b) (* @c x @d)) (= x 0)))
(defaxiom! =nums (= (!app number? x) (!app number? y)) #f)
(defaxiom! =factor2a (= (!app number? n) (* (!app number? m) @a))
  (= (s/ n m) (* @a)))
(defaxiom! =factor2b (= (* (!app number? m) @a) (!app number? n))
  (= (* @a (s/ n m)))
(defaxiom! const (= x (+ @a (!app number? y) @b))
  (= (- x y) (+ @a @b)))
(defaxiom! =t (= x x) #t)

(defmacro `(def? ,x) `(defined? ,x (cons (car (the-environment)) )))

#| > (+)

+assoc 0 1 .(+)
+ident 1 1 .(+)
+combine4 2 1 .(+)
+combine3 3 1 .(+)
+combine2 4 1 .(+)
+combine1 5 1 .(+)
+inverse4 6 1 .(+)
+inverse3 7 1 .(+)
+inverse2 8 1 .(+)
+inverse1 9 1 .(+)
+nums 10 1 .(+)
+x 11 1 .(+)
+no-args 12 1 .(+)
+no-args 12 1 .<0
0

#| > (deri x x)

derinx 0 1 .(deri x x)
-binary 14 2 .(- (- (+ x dx) x)
-x 17 3 .(- x)
-x 17 3 .(- x)
+assoc 41 3 .(+ (+ x dx) (* -1 x))
+combine2 44 4 ......>(+ x dx (* -1 x))
+nums 53 5 ......>(+ -1 1)
+ident 64 6 ......>(+ 0)
+no-args 77 7 ..........>(+)
+no-args 77 7 ..........<0
+ident 64 6 ..........<0
+nums 53 5 ..........<0
*0 89 5 ......>(* 0 x)
*0 89 5 ......<0
+ident 100 5 ......>(+ 0 dx)
+x 112 6 ...........(+ dx)
+x 112 6 ..........<dx
+ident 100 5 ..........<dx
+combine2 44 4 ..........<dx
+assoc 41 3 ..........<dx
-binary 14 2 ..........<dx
quotient-to-pro 113 2 ..........<(/ dx dx)
*combine2 116 3 ..........>(* dx (expt dx -1))
+nums 125 4 ..........>(+ -1 1)
+ident 136 5 ..........>(+ 0)
+no-args 149 6 ...........(+)
+no-args 149 6 ..........<0
+ident 136 5 ..........<0
+nums 125 4 ..........<0
*ident 160 4 ..........>(* 1)
*no-args 174 5 ...........(*)
*no-args 174 5 ..........<1
*ident 160 4 ..........<1
*combine2 116 3 ..........<1
quotient-to-pro 113 2 ..........<1
derinx 0 1 ..........<1
1
|

# |

> (real (+ x dx))

rexpx 0 1 ..........>(real (+ x dx))
rprox 1 1 ..........>(real (+ x dx))
rdifix 2 1 ..........>(real (+ x dx))
rsumx 3 1 ..........>(real (+ x dx))
rexpx 4 2 ..........>(real dx)
rprox 5 2 ..........>(real dx)
rdifix 6 2 ..........>(real dx)
rsumx 7 2 ..........>(real dx)
rdx0 8 2 ..........>(real dx)
rdx0 8 2 ..........<0
rexpx 9 2 ..........>(real x)
rprox 10 2 ..........>(real x)
rdifix 11 2 ..........>(real x)
rsumx 12 2 ..........>(real x)
rdx0 13 2 ..........>(real x)
rx 14 2 ..........>(real x)
rx 14 2 ..........<x
rsumx 3 1 ..........<x
x
\#

\[
> (\text{real } (+ \ x \ dx))
\]

+assoc 0 1 \(\rightarrow (+ x \ dx)\)
+ident 1 1 \(\rightarrow (+ x \ dx)\)
+combine4 2 1 \(\rightarrow (+ x \ dx)\)
+combine3 3 1 \(\rightarrow (+ x \ dx)\)
+combine2 4 1 \(\rightarrow (+ x \ dx)\)
+combine1 5 1 \(\rightarrow (+ x \ dx)\)
+inverse4 6 1 \(\rightarrow (+ x \ dx)\)
+inverse3 7 1 \(\rightarrow (+ x \ dx)\)
+inverse2 8 1 \(\rightarrow (+ x \ dx)\)
+inverse1 9 1 \(\rightarrow (+ x \ dx)\)
+nums 10 1 \(\rightarrow (+ x \ dx)\)
+x 11 1 \(\rightarrow (+ x \ dx)\)
+no-args 12 1 \(\rightarrow (+ x \ dx)\)
rexpx 13 1 \(\rightarrow (\text{real } (+ x \ dx))\)
rprox 14 1 \(\rightarrow (\text{real } (+ x \ dx))\)
rdfx 15 1 \(\rightarrow (\text{real } (+ x \ dx))\)
rsumx 16 1 \(\rightarrow (\text{real } (+ x \ dx))\)
+assoc 17 2 \(\rightarrow (+ dx)\)
+ident 18 2 \(\rightarrow (+ dx)\)
+combine4 19 2 \(\rightarrow (+ dx)\)
+combine3 20 2 \(\rightarrow (+ dx)\)
+combine2 21 2 \(\rightarrow (+ dx)\)
+combine1 22 2 \(\rightarrow (+ dx)\)
+inverse4 23 2 \(\rightarrow (+ dx)\)
+inverse3 24 2 \(\rightarrow (+ dx)\)
+inverse2 25 2 \(\rightarrow (+ dx)\)
+inverse1 26 2 \(\rightarrow (+ dx)\)
+nums 27 2 \(\rightarrow (+ dx)\)
x 28 2 \(\rightarrow (+ dx)\)
+x 28 2 \(\rightarrow dx\)
rexpx 29 2 \(\rightarrow (\text{real } dx)\)
rprox 30 2 \(\rightarrow (\text{real } dx)\)
rdfx 31 2 \(\rightarrow (\text{real } dx)\)
rsumx 32 2 \(\rightarrow (\text{real } dx)\)
rdx0 33 2 \(\rightarrow (\text{real } dx)\)
rdx0 33 2 \(\rightarrow 0\)
rexpx 34 2 \(\rightarrow (\text{real } x)\)
rprox 35 2 \(\rightarrow (\text{real } x)\)
rdfx 36 2 \(\rightarrow (\text{real } x)\)
rsumx 37 2 \(\rightarrow (\text{real } x)\)
rdx0 38 2 \(\rightarrow (\text{real } x)\)
r 39 2 \(\rightarrow (\text{real } x)\)
rx 39 2 \(\rightarrow x\)
+assoc 40 2 \(\rightarrow (+ x \ 0)\)
+ident 41 2 \(\rightarrow (+ x \ 0)\)
+assoc 42 3 \(\rightarrow (+ x)\)
+ident 43 3 \(\rightarrow (+ x)\)
+combine4 44 3 \(\rightarrow (+ x)\)
+combine3 45 3 \(\rightarrow (+ x)\)
+combine2 46 3 \(\rightarrow (+ x)\)
+combine1 47 3 \(\rightarrow (+ x)\)
+inverse4 48 3 \(\rightarrow (+ x)\)
+inverse3 49 3 \(\rightarrow (+ x)\)
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line</th>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inverse2</td>
<td>50</td>
<td>3</td>
<td>...&gt;(+ x)</td>
</tr>
<tr>
<td>inverse1</td>
<td>51</td>
<td>3</td>
<td>...&gt;(+ x)</td>
</tr>
<tr>
<td>nums</td>
<td>52</td>
<td>3</td>
<td>...&gt;(+ x)</td>
</tr>
<tr>
<td>x</td>
<td>53</td>
<td>3</td>
<td>...&lt;x</td>
</tr>
<tr>
<td>ident</td>
<td>41</td>
<td>2</td>
<td>..&lt;x</td>
</tr>
<tr>
<td>rsumx</td>
<td>16</td>
<td>1</td>
<td>.&lt;x</td>
</tr>
</tbody>
</table>

x

#
Version III
(define (single-sym? x) (if (symbol? (make-quot e x)) #t #f))

(define s* *) (define * _*)
(define s+ +) (define + _+)
(define s/ /) (define / _/)
(define s- -) (define - _-)
(define s= =) (define = _=)

(define sin sin) (define sin _sin)
(define cos cos) (define cos _cos)
(define tan tan) (define tan _tan)
(define log log) (define log _log)
(define exp exp) (define exp _exp)
(define expt expt) (define expt _expt)

(define sec _sec)

(define x _x)
(define dx _dx)
(define wx _wx)
(define y _y)
(define a _a)
(define b _b)
(define k _k)
(define m _m)
(define n _n)
(define r _r)

(defaxiom! +no-args (+)0)
(defaxiom! +x (+ x)x)
(defaxiom! +ident (+ @a 0 @b) (+ @a @b))
(defaxiom! +assoc (+ @a(+ @b @c) (+ @a @b @c))
(defaxiom! +nums (+ @a(!app number? x)@b(!app number? y)@c)
(+ @a(s+ x y)@b @c))
(defaxiom! +inverse1 (+ @a x @b(- x)@c) (+ @a @b @c))
(defaxiom! +inverse2 (+ @a (- x)@b x @c) (+ @a @b @c))
(defaxiom! +inverse3 (+ @a(*(- x)y)@b(!test (* @z) (=(* x y)(* @z)))@c)
(+ @a @b @c))
(defaxiom! +inverse4 (+ @a(* @z)@b(!test(*(- x)y) (=(* x y)(* @z)))@c)
(+ @a @b @c))
(defaxiom! +combine1 (+ @a x @b x @c) (+ @a(* 2 x)@b @c))
(defaxiom! +combine2 (+ @a x @b(*(!app number? n)x)@c)
(+ @a(*(+ n 1)x)@b @c))
(defaxiom! +combine3 (+ @a(*(!app number? n)x)@b x @c)
(+ @a(*(+ n 1)x)@b @c))
(defaxiom! +combine4 (+ @a (!or(*(!app number? n)@x)(* @x))@b
(!test(!or(*(!app number? m)@y)
(* @y)) (=(* @x)(* @y)))@c)
(+ @a(*(+if (def? 'n)n 1)(if(def? 'm)m 1))@x)@b @c))

(defaxiom! -x (- x)(* -1 x))
(defaxiom! -sum (-(+ x @a))((- x)(* (- @a))))
(defaxiom! -binary (- x y)(* x (- y)))

(defaxiom! *no-args (*1)
(defaxiom! *x (* x)x)
(defaxiom! *0 (* @a 0 @b)0)
(defaxiom! *ident (* @a 1 @b)(* @a @b))
(defaxiom! *assoc (* @a(* @b)b)(* @a @b @c))
(defaxiom! *distrib1 (* (+(@(i:do (i 1 m) (*(ref i x)))) n)
   (+(@(i:do (i 1 m) (*(n (ref i x)))))))
(defaxiom! *distrib2 (* n (+(@(i:do (i 1 m) (ref i x)))
   (+(@(i:do (i 1 m) (n (ref i x)))))))))
(defaxiom! *neg-to-left (* @a(- x)b)(* -1 @a x @b))
(defaxiom! *num-to-left (* x @a(@(app number? y)b)(* y x @a @b))
(defaxiom! *nums (* @a(@(app number? x)b(@(app number? y)c)
   (* (s* x y)a @b @c)))
(defaxiom! *combine1 (* @a(@(app single-sym? x)b x @c)
   (* @a(b(expt x 2) @c))
(defaxiom! *combine2 (* @a(@(app single-sym? x)b(expt x n) @c)
   (* @a(b(expt x(+ n 1)) @c))
(defaxiom! *combine3 (* @a(expt(@(app single-sym? x)n)b x @c)
   (* @a(b(expt x(+ n 1))) @c))
(defaxiom! *combine4 (* @a(expt @(app number? n)b
   (expt x(@(app number? m))) @c)
   (* @a(expt x(+ n m)) @b @c))
#
(defaxiom! fact (fac n) (binl 1 n 1))
(defaxiom! bincl (binl n r) (* @(i:do (i (+ r 1) n) i)))
(defaxiom! bincof (bin c n r) (/ (binl n r) (fac (- n r))))
#
(defaxiom! fac0 (fac 0 1))
(defaxiom! facx (fac x)(* x (fac (- x 1))))
(defaxiom! fac0 (fac 0 1)
(defaxiom! bincof (bin c n r) (/ (fac n) (* (fac (- n r)) (fac r))))
(defaxiom! binxp (expt (+ x y) n) (+(@(i:do(r 0 n)
   (* (bin c n r) (expt x (- n r)) (expt y r))))
(defaxiom! quotient-to-product (/ x y)(* x(expt y -1)))
(defaxiom! 1^x (expt 1 x) 1)
(defaxiom! x^1 (expt x 1)x)
(defaxiom! 0^x (expt 0 x) 0)
(defaxiom! x^0 (expt x 0) 1)
(defaxiom! x^sum (expt x (+ a @b))(*(expt x a) (expt x (+ @b))))
(defaxiom! product^n (expt(* x @a)n)(* (expt x n) (expt(* @a)n)))
(defaxiom! x^m^n (expt(expt x m) n) (expt x(* m n)))
(defaxiom! n^m (expt(@(app number? n)b(@(app number? m))(sexpt m n))
(defaxiom! derinx (deri x (^f x)) (/ (- (^f (+ x dx)) (^f x)) dx))
(defaxiom! rx (real x) x)
(defaxiom! rdx0 (real (!test x (equal? x dx)) 0)
(defaxiom! rsumx (real (+ x @y)) (+ (real x) (real (+ @y))))
(defaxiom! rdiffx (real (- x @y)) (-(real x) (real (- @y))))
(defaxiom! rprox (real (* x @y)) (* (real x) (real (* @y))))
(defaxiom! repxx (real (expt x y)) (expt (real x) (real y))))
(defaxiom! const (der x(@(app number? c))) 0)
(defaxiom! var (der x(@(app single-sym? y))) 0)
(defaxiom! samevar (der x x) 1)
(defaxiom! - (der x(- a))(-(der x a))
(defaxiom! + (der x(+ a @s))+(der x a)(der x (+ @s))))
(defaxiom! * (der x (* a @s)) (+(*(der x a)@s)(* a(der x (* @s)))))
(defaxiom! log (der x (log a)) /(der x a) a))
(defaxiom! exp (der x (exp y)) (* (exp y)(der x y)))
(defaxiom! sin (der x (sin a))(*(cos a)(der x a)))
(defaxiom! cos (der x (cos a)) -(*(sin a)(der x a)))
(defaxiom! tan (der x (tan a))(*(expt(sec a)2)(der x a)))
(defaxiom! expt (der x (expt a b))
   (+(*(b(expt a(- b 1))(der x a))(*(expt a b)(log a)(der x b)))))
(defaxiom! dist (der x (* @a(!and(+ y @b)(^t x))@c))
   (der x (+(* y @a @c)(* @a(+ @b)@c))))

(defaxiom! =sum (=(+ @a x @b)(+ @c(!test y(= x y))@d))
   (=(+ @a @b)(+ @c @d)))
(defaxiom! =product (=(* @a x @b)(* @c x @d))
   (or(=(* @a @b)(* @c @d))(= x 0)))
(defaxiom! =nums (=(!app number? x)(!app number? y)) #f)
(defaxiom! =factor2a (=(!app number? n)(*(!app number? m)@a))
   (=s/ n m)(* @a))
(defaxiom! =factor2b (=(*(!app number? m)@a)(!app number? n))
   (=(* @a)(s/ n m)))
(defaxiom! const (= x(+ @a(!app number? y)@b))
   (=(- x y)(+ @a @b)))
(defaxiom! =t (= x x) #t)

(defmacro '(def? ,x) '(defined? ,x (cons(car(the-environment))'())))

#| > (deri x (expt x 2))
(+ (* 2 x) dx)
[ CPU: 4.401000 GC: .5829999 Real: 13.831 ]
> (deri x (expt x 3))
(+ (* 3 (expt x 2)) (* 3 dx x) (expt dx 2))
[ CPU: 7.450000 GC: 1.183000 Real: 24.833 ]
> (deri x (expt x 4))
(+ (* 4 (expt x 3)) (* 6 dx (expt x 2)) (* 4 (expt dx 2) x) (expt dx 3))
[ CPU: 11.44999 GC: 1.765999 Real: 38.33 ]
> (deri x (expt x 5))
(+ (* 5 (expt x 4)) (* 10 dx (expt x 3)) (* 10 (expt dx 2) (expt x 2)) (* 5 (expt
dx 2) x) (expt dx 3))
[ CPU: 16.28299 GC: 2.449999 Real: 56.155 ]
> (deri x (expt x 6))
(+ (* 6 (expt x 5)) (* 15 dx (expt x 4)) (* 20 (expt dx 2) (expt x 3)) (* 15 (expt
dx 2) x) (expt dx 3))
[ CPU: 21.851 GC: 3.382999 Real: 75.241 ]
>