LUND UNIVERSITY

Simplification of Expressions Using Prolog

Brück, Dag M.

1987

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Brück, D. M. (1987). *Simplification of Expressions Using Prolog*. (Technical Reports TFRT-7364). Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

CODEN:LUTFD2/(TFRT–7364)/1-014/(1987)

Simplification of Expressions using Prolog

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
July, 1987

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
July 1987

Document Number
CODEN:LUTFD2/(TFRT-7364)/1-014/(1987)

Author(s)
Dag M. Brück

Supervisor

Sponsoring organisation
The National Swedish Board of Technical Development

Title and subtitle
Simplification of Expressions using Prolog

Abstract

The report describes the development of a Prolog program for simplification of polynomial expressions. The main problem is that expressions can take many shapes, leading to a very large number of transformation rules.

The solution is to introduce a normal form for expressions, and to order the terms of a sum and the factors of a product. Ordering also changes the shape of the expression tree. These operations greatly reduce the number of transformation rules. Execution speed is considerably improved by detecting dead ends that cannot be further simplified, and by rejecting repeated results.

The evolution of the program, and the consequences of using Prolog or Lisp as implementation languages, are also discussed.

Key words
Symbolic computation, Pattern matching, Logic Programming, Prolog

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
14

Security classification

Recipient's notes

The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Telex: 33248 lubbis lund.

CODEN:LUTFD2/(TFRT–7364)/1-014/(1987)

Simplification of Expressions using Prolog

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
July, 1987

1. Introduction

This report describes the development of a program for differentiation and simplification of mathematical expressions. The work is centered on simplification; the differentiation part was taken from section 7.11 of *Programming in Prolog* [Clocksin and Mellish, 1981].

The basic idea behind the simplification algorithm is quite simple. A table of transformation rules describes possible ways of rewriting an expression. To simplify a binary expression E, we must first simplify the left-hand argument of E, then the right-hand argument, and finally see if the simplified expressions can be matched with one of the transformation rules. Normally, more than one rule will match, so multiple simplifications are generated.

Achievements

A program has been developed that can do certain simplifications of polynomials; it is implemented in Prolog. The problems of symbolic computation have been investigated (and experienced), and a number of approaches have been explored. The insights gained are readily transferable to other programming environments.

Given a polynomial expression as input, the program will try to transform it to fully expanded form. For example,

$$(a+b)(c+d) \rightarrow ac + ad + bc + bd$$

($\rightarrow$ means *is transformed to*). Similar terms or factors will be combined:

$$3x + 2x \rightarrow 5x$$
$$x^2 * x^3 \rightarrow x^5$$

The following example demonstrates what the program can do:

$$3 + x + 2x + (x+1)(x-1) \rightarrow x^2 + 3x + 2$$

which is equal to $(x+1)(x+2)$, although the computer doesn't know it.

Limitations

The existing implementation is quite limited. Firstly, the program can do useful work only on polynomials; other types of expressions (e.g., logarithms and complex numbers), will be treated as indivisible units.

Secondly, the "simplifier" has no way of measuring the simplicity or readability of an expression. Simple merits, such as the length of the expression, would be easy to implement, but would probably not be of any great use; the most desirable form of an expression depends largely on the intended use. Rather, the program will print all transformations of an expression it has been able to find.

Thirdly, factoring is not done. Therefore, neither of the following transformations will be found.

$$x^2 - 1 \rightarrow (x+1)(x-1)$$
$$x^2 + 2x + 1 \rightarrow (x+1)^2$$

Not even content factoring, i.e., extraction of the greatest common divisor of the terms, is possible. For example,

$$6x^2 + 4x \rightarrow 2x(3x + 2)$$

is not on, although the reverse is, using expansion and combination.

2. Fundamental algorithms

This section describes some of the problems with simplification of expressions, and ways of dealing with these problems.

Fundamental problems

The first version of the program was directly based on the simplification algorithm in section 7.12 of *Programming in Prolog* [Clocksin and Mellish, 1981]. Although it works on naive examples, it is easy to find special cases which require more rules, and some cases which are very difficult to express. There are three major problems:

1. Commutative operators, such as addition and multiplication, double the number of rules; $x + y$ is algebraicly but not syntacticly equal to $y + x$.
2. An expression may be represented by different expression trees. This is insignificant when evaluating the expression, but matching is done symbolicly on the tree structure. For example, $2 * 3 * a$ is represented with a different tree than $a * 2 * 3$, although the trees have the same shape. Even worse, $2 * 3 * a$ has a differently shaped tree compared with $2 * (3 * a)$.
3. Combinations of different operators, such as addition and subtraction, increases the number of rules. Many special cases, like $a + b - a = b$, must be handled.

These problems would lead to a *very* large number of transformation rules, with little structure. Sensible rules can hardly be machine generated, and it is impossible to verify that all essential rules exist. It is also likely that more than one rule will yield equivalent transformations, for example, one covering a special case and one covering the general case; this may generate an uncontrollable number of so called simplifications.

The goal is to impose some structure on the problem and to reduce the number of rules. This can be done by applying three transformation steps:

- The expression is transformed to so called normal form, eliminating subtraction and division operators. This step addresses problem (3) above.
- Terms and factors of an expression are ordered to get syntacticly uniform expressions.
- Some basic simplifications must be made, for example, evaluation of expressions involving constants only (constant folding).

The last two steps will handle problems (1) and (2). The first step is needed only once, while the other two must be performed as parts of each major transformation.

Normal form

Prior to any other processing, the expression is normalized. Normal form can loosely be described as a polynomial with all factors written out, e.g., $1 * x^1$. Normal form is defined more formally using BNF:

$$
\begin{array}{rcl}
\textit{expr} & ::= & [\textit{expr} +]\ \textit{term} \\
\textit{term} & ::= & [\textit{term} *]\ \textit{factor} \\
\textit{factor} & ::= & \text{number} \\
 & | & \text{number} * \text{atom} \uparrow \text{number} \\
 & | & (\textit{expr})\ [\uparrow \text{number}]
\end{array}
$$

To get this far, a number of transformations must be made, making the expression visually much more complicated:

- Variables and symbolic constants (atoms in Prolog) are written on exponent form: $a \rightarrow 1 * a^1$. This means that no special rule is needed to handle both $x * x^n \rightarrow x^{n+1}$ and $x^m * x^n \rightarrow x^{m+n}$, and other similar cases.
- Elimination of subtraction, division and unary minus:

  $$
  \begin{aligned}
  a - b &\rightarrow a + (-1) * b \\
  a/b &\rightarrow a * b^{-1} \\
  -a &\rightarrow (-1) * a
  \end{aligned}
  $$

  These transformations reduce the number of operators, and hence greatly reduce the number of rules. They also introduce a nice symmetry: addition and multiplication are both commutative and associative, which is used for ordering expressions.
- Positive powers of complex (non-atomic) expressions are expanded:

  $$(a + b)^3 \rightarrow (a + b) * (a + b) * (a + b)$$

  The use of this transformation is not so obvious, but it is often applied in symbolic algebra [Pavelle et al., 1981]. It should be noted that negative powers are not handled gracefully. In particular, a negative power should be distributed over a product:

  $$(ab)^{-3} \rightarrow a^{-3}b^{-3}$$

In addition to these transformations, a zero term is appended to every sum, and a unity factor appended to every product. This is done during expression ordering (see below).

Ordering of expressions

The most fundamental and important task when doing symbolic calculations, is to order or sort terms of a sum and factors of a product. One reason for eliminating all operators but addition and multiplication, is to be able to rearrange an expression using the laws of commutativity and associativity.

Expressions are ordered because it is easy (and cheap with respect to computer resources) to do operations on two adjacent elements of an expression, and much more difficult when the whole expression must be considered. In particular, matching expressions using Prolog is most conveniently done by looking at two subexpressions combined with a binary operator; this may not be true when a list-based notation for expressions is used, as in Lisp.

If the expression is ordered according to some mechanism which takes into account the type of subexpressions involved, a number of fundamental operations are easy to implement. Firstly, if all numeric constants are grouped together, then numeric subexpressions can be evaluated by successively looking at pairs of constants. This operation (called constant folding) is one of the basic simplifications, see below.

Secondly, the most powerful means of simplifying an expression in the current implementation, combination of factors, is also efficient when similar factors are grouped together. This is also done pair-wise, left-to-right:

$$3x + x + 2x \rightarrow 4x + 2x \rightarrow 6x$$

The definition of "relative order" used for ordering the elements is here very important. If alphabetic order is used, then

$$2x + 3y + 5x$$

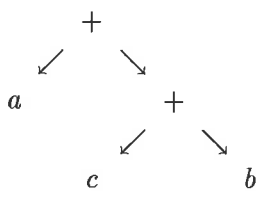
would be correct, but in this case combination is not possible. The ordering predicate used will look into the elements and, for example, disregard numeric coefficients. The last equation would come out as:

$$2x + 5x + 3y$$

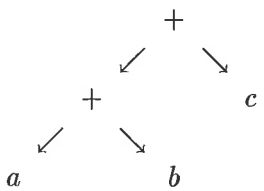
which is easily combined to give $7x+3y$. Lastly, we are used to order the terms of a polynomial when using pencil and paper. It is advantageous if a similar ordering can be used by the program, both when writing new transformation rules and when presenting the result to the user.

In the discussion above, the expression has been regarded as a linear list of terms. This is not true; an expression is represented by a binary tree. To be able to match trees of any shape, a large number of cases must be regarded. Because all (both) operators after normalization are associative, the expression can be extensively rearranged. The goal is to provide not only a uniform order of terms, but also a uniform shape of the expression tree.

The procedure for ordering expressions is as follows: Firstly, the binary expression tree is cut to pieces and all elements that can be rearranged are put in a list. Secondly, the list is ordered, disregarding numeric coefficients of terms. Lastly, a left-recursive expression tree is constructed from the list. These operations are performed recursively for all sums and products. As an example, the expression $a + (c + b)$ has the following tree.



After ordering and reconstruction, we get $a + b + c$, which has this tree.



To make matching even easier, one more trick is used. A zero term is appended to every sum, and a unity factor is appended to every product. After ordering, all numeric elements will be located in the leftmost part of each subtree. Consequently, all symbolic elements can be found by matching the right operand of each operator. The number of special cases are further reduced.

Basic simplifications

After normalization and ordering, there is usually a large number of numeric terms and factors in the expression. The routines for basic simplification will evaluate almost all numeric expressions; floating point numbers look messy, so only integer powers are evaluated. The "dummy" terms and factors inserted during ordering are therefore eliminated when not needed.

In addition, products with a zero factor are reduced to zero, and zero terms are removed. For example: $a + 0 * b \rightarrow a$. A single numeric term or factor is of course saved in every sum or product. A number of special cases involving exponentiation are detected: $x^0 \rightarrow 1$, $0^n \rightarrow 0$ and $1^n \rightarrow 1$.

Finally, to see how these three steps transform an expression, the expression $3 + x + 2x + (x + 1)(x - 1)$ is transformed to:

$$3 + 1 * x^1 + 2 * (1 * x^1) + (1 * x^1 + 1) * (1 * x^1 + -1 * 1)$$

after normalization,

$$0 + 3 + 1 * 1 * x^1 + 1 * (0 + 1 + 1 * 1 * x^1) * (0 + -1 * 1 * 1 + 1 * 1 * x^1) + 1 * 1 * 2 * x^1$$

after ordering, and

$$3 + 1 * x^1 + 1 * (1 + 1 * x^1) * (-1 + 1 * x^1) + 2 * x^1$$

after basic simplifications.

Final clean-up

Just before printing, the results of a simplification are cleaned up a little. The following simplifications are made:

$$\begin{aligned} 0 + a &\rightarrow a \\ 1 * a &\rightarrow a \\ a^1 &\rightarrow a \\ c * a &\rightarrow ca \qquad \text{when } c \text{ is numeric} \\ a * b^{-n} &\rightarrow a/b^n \\ a + (-b) &\rightarrow a - b \end{aligned}$$

The expression will become more readable, but not all transformations from the normalization step are undone.

3. The real McCoy

After the treatment of normalization, ordering and basic simplifications, we now have an expression in a form which is supposed to be "suitable" for further

processing. This section describes the core of the program; routines that try to analyze the expression and do substantial transformations of it. These are called global transformations because they may look at the full expression tree.

There is an important difference between these routines and the routines described in section 2: as many transformations as possible will be generated in every step of the process. There may even be transformation rules with contradictory goals; for example, one rule expanding the expression, and one rule factoring the expression. Both transformations will, if possible, be applied to every single expression.

The transformations are not restricted, except that normal form should be maintained. It is therefore necessary to order the resulting expression and to perform basic simplifications after each global transformation.

Operation

Global transformations are performed by two cooperating routines. One Prolog predicate (called `simp`) will generate new transformations one at a time. The other (`global`) is a driver routine which will generate all transformations, calling `simp` over and over again. This is done essentially as follows:

1. Transform the expression by calling `simp`. If no transformation was possible, the same expression is returned.
2. Clean the result of (1) by ordering and folding constants, etc.

3a. If the transformations of (1) and (2) have given a new expression, the result is recorded as a fact in the Prolog database. Then the crank is turned again; new global transformations are made on the result, going back to (1).

3b. If the transformations yield a result which has already been found because of some earlier transformations, then the transformations of (1) and (2) are said to have failed.

A new transformation is generated by returning to (1) and asking `simp` for another transformation. As a last resort, `simp` will return the input expression, so all global transformations will eventually fail, having explored all possibilities.

The operation of `global` and `simp` can be described as generating the tree of all possible transformations of an expression. All cycles will be detected and cut off in (3). `Simp` returning the input expression is a cycle back to the latest node of the tree.

Dead end detection

The expressions are often quite similar between each "turn of the crank." Identical subexpressions occur every time. This turns out to be very inefficient; a large number of subexpressions, which cannot be transformed with the given rules, must be traversed and tested.

The solution is to register all so called dead ends, i.e., expressions which cannot be further simplified. In the current implementation, dead ends are registered as facts, for example,

$$\text{deadend}(x \uparrow 3).$$

but could also be registered as predicates, testing some common cases:

$$\text{deadend}(X \uparrow Y) :\!- \text{atomic}(X), \text{atomic}(Y).$$

The increase in performance is substantial; simplification of typical examples run 4–5 times faster with dead end detection.

The dead end database is valid as long as the rules remain unchanged. Dead ends found when simplifying one expression are valid when simplifying another. Regrettably, the database grows all the time, and must be manually purged now and then. A few simple predicates would greatly reduce the number of generated facts.

4. Discussion of implementations

The discussion will cover two areas: Firstly, the evolution of the current Prolog program. Secondly, the consequences of implementing such a program in Prolog or in Lisp.

Evolution of the program

It is interesting to see how the program, and in particular the transformation rules, have evolved during the development of the program.

As has been noted in section 2, it all started by only using a routine similar to the basic simplifications. Quickly, the number of special cases grew uncontrollably, and it is definitely not the case that "a more general algebra system can be constructed simply by adding more clauses," as is said in *Programming in Prolog*.

The use of ordering is emphasized in both [Pavelle et al., 1983] and [muMATH, 1983], and made generalizations possible in this program. There is no evidence that normalization, as described in section 2, is used in muMATH, but it is probably used in some routines. No single representation is the best for all applications; multiple representations may be used in a program [Pavelle et al., 1983]. Normalization and ordering reduced the number of rules to about one fifth. At the same time, a number of basic simplifications were removed in order to maintain normal form; they are now made during the final clean-up.

The rules controlling the global transformations have changed the most, becoming simpler and more general. In the beginning, there were a number of specialized rules (c_i are numbers; $k = c_1 c_2$):

$$
\begin{aligned}
(x+y)*(x-y) &\rightarrow x^2 - y^2 \\
(x+y)*(x+y) &\rightarrow x^2 + y^2 + 2xy \\
(x-y)*(x-y) &\rightarrow x^2 + y^2 - 2xy \\
(x+y)*(z+w) &\rightarrow xz + xw + yz + yw \\
c_1*(c_2+x) &\rightarrow k + c_1 x \\
(x+c_1)*(x-c_2) &\rightarrow x^2 - k
\end{aligned}
$$

Eventually, all these cases (and many more) were handled by two general rules for distributing a multiplication over an addition:

$$
\begin{aligned}
c*(x+y) &\rightarrow cx + cy \\
(x+y)*z &\rightarrow xz + yz
\end{aligned}
$$

Although the last two rules seem less powerful, the same results are found when they are repeatedly applied.

Performance *is* an important issue in this program. Much time is spent doing useless work. Testing a few special cases, such as sorting a list with one element, provided a two-fold speedup. Even more important are dead end detection and cutting off duplicate results. The "cut" in Prolog is used extensively, for two resons. Firstly, debugging is much less tedious when meaningless backtracking is avoided. Secondly, performance increases slightly.

Implementation in Prolog

Prolog is a good implementation language for this kind of work. Three advantages should be noted in particular:

- Most work is done by pattern matching. Clauses are written to handle one case each, and the Prolog system "automaticly" selects the right clause. Additional qualifications, e.g., requesting a variable to be numeric, are easy to add.
- The Prolog database is used to register facts about dead ends and results. Self-modifying programs are quite powerful in this context. Using the built-in facilities of a programming language is of course the fastest way to solve a problem.
- Prolog uses "ordinary" infix operators in forming expressions. This notation is still the best for most people. The ability to define new operators could be used in further developed systems. Prolog is one of the few programming languages where it is easy to go from infix notation to list (prefix) notation.

On the other hand, there are problems with both the current implementation style and Prolog itself:

- List notation, as in Lisp, is more natural when representing sums and products with more than two elements. Searching and sorting is easier, which ought to be of particular importance when factoring expressions. The extensive reordering of the expression tree would be easier and faster.
- In Prolog, it is often necessary to go from infix notation to list notation of expressions, for example, to get the operands of an unspecified operator. Sometimes, lists cannot be used, for example, to evaluate a numeric expressions.
- There are very few control structures in Prolog; conditional execution must be implemented by matching clauses, and repetition by tail-recursion or backtracking. To avoid doing huge amounts of unnecessary work, extra clauses must be added when a simple if-statement would have been enough.
- Compilation is not yet common in Prolog systems. Much time is spent doing simple things in a few routines. Compiled code would be substantially faster.

The advantages and disadvantages must be carefully weighted against each other; it would have been quite feasible to make an implementation based on list notation in Prolog. It is impossible, without trying, to say which implementation would be better. Backtracking was used less than expected, so the powers of Prolog may not have been fully explored.

Implementation in Lisp

Lisp is the oldest, and still most widely used, programming language for symbolic computations. In the context of this program, the following can be noted:

- Lisp only supports list (prefix) notation of expressions. There are more powerful operations on lists in Lisp.
- Modern Lisp dialects support other data structures, such as arrays and hash tables.
- Lisp has control structures for conditional execution and iteration. Backtracking is therefore not required.
- Lisp can be compiled into efficient code.

The main problem to solve in a Lisp implementation is the pattern matching and substitution used for doing the transformations. It is available almost for free in Prolog, but must be explicitly programmed in Lisp. This is not an easy task; in particular, there is no concept of "instantiated" or "uninstantiated" variables as in Prolog. A beautiful way of creating a logic programming environment is presented in sections 4.4 and 4.5 of *Structure and Interpretation of Computer Programs* [Abelson and Sussman, 1985]. Their query language comes very close to the facilities provided by a Prolog system, but is perhaps less convenient to use.

The ability to assert new facts (or even predicates) in Prolog exists in Lisp too. The program can modify itself and define new functions. Regrettably, this ability is much less useful in Lisp because there is no built-in pattern matcher as in Prolog. It is possible to define additional code in Lisp, but not to use the Lisp system as a database (although production systems, such as YAPS [Allen, 1983], may be used). Lisp has global variables though, which is a common use of Prolog facts.

5. Summary

Simplification of expressions is not easy. Firstly, the intended use of an expression determines what is a useful simplification; only the human user knows this. Secondly, expressions can be written in many ways, leading to a huge number of cases that must be handled by the program. Thirdly, a large program with possibly hundreds of transformation rules cannot be maintained. Rules without structure are not comprehensible.

Most of these problems can be solved with a framework that serves as the basis for future developments. By transforming the expression to normal form, the number of operators and special cases are greatly reduced. Ordering makes it possible to efficiently perform transformations, and rearranging the expression reduces the number of cases that must be tested even further. Basic simplifications, such as evaluating numeric expressions, are also needed.

After that, it is not so difficult to write the rules that do the real transformations. Rules with contradictory goals (e.g., factoring and expansion) may be used. A driver routine which generates all solutions is required; this is one of the few actual uses of Prolog's backtracking mechanism. Detecting dead ends and repeated results is the most effective way of improving execution speed.

Prolog is a suitable programming language for this application. The pattern matching is extremely powerful, and quite difficult to emulate in other

languages. Using the Prolog system as a database manager is convenient. Regrettably, there are few control structures in Prolog, sometimes forcing the user to adopt an awkward programming style.

Lisp is also a good implementation language. It has better list handling and more data structures, but some important parts must be written by the user, in particular the pattern matcher. Estimating the time needed to implement these parts is difficult.

6. Acknowledgements

Professor Karl Johan Åström suggested this problem for me, and introduced me to muMATH. Per Andersson found the reference to the query language in [Abelson and Sussman, 1985]. Kjell Gustavsson reviewed the report and made a number of useful comments.

This work was supported by The National Swedish Board of Technical Development (STU).

7. References

ABELSON, H. and G. J. SUSSMAN (1985): *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, Massachusetts.

ALLEN, E. M. (1983): "YAPS: Yet Another Production System," TR–1146, Department of Computer Science, University of Maryland.

CLOCKSIN, W. F. and C. S. MELLISH (1981): *Programming in Prolog*, Springer-Verlag, Berlin-Heidelberg.

——— (1983): *muMATH-83 Reference Manual*, The Soft Warehouse, Honolulu, Hawaii.

PAVELLE, R., M. ROTHSTEIN and J. FINCH (1981): "Computer Algebra," *Scientific American* **245**, 6 (December), 136–152.

Appendix – Using the program

The program consists of six modules:

simp.pl	Main program, global simplifications.
normal.pl	Normalization routines.
order.pl	Ordering routines.
basic.pl	Basic simplifications.
deriv.pl	Differentiation.
tree.pl	Draws expression trees.

The main program, simp.pl, will load the other modules. The following table describes the most interesting predicates:

sim(*expr*)	Generates all "simplifications" of *expr*. Every global transformation is traced, and all results are printed.
dx(*expr*)	Calculates the derivative of *expr* with respect to x, and simplifies the result.
d(E, a, F)	Calculates derivative of E with respect to a, giving F. No simplification.
normal(E, F)	Normalizes E, giving F.
order(E, F)	Orders E, giving F.
basic(E, F)	Performs basic simplifications on E, giving F.
pr(*expr*)	Prints *expr* after final clean-up.
prr	Prints the results of the last simplification.
tree(*expr*)	Prints the expression tree for *expr* (very crude).

Example

The following example shows most of the features of the program:

```
| ?- sim(2*x*(3*x + 2)).
Starting:        2*x*(3*x+2)

0        In      2*(2+3*x^1)*x^1
0        Out     0+4*x^1+6*x^1*x^1

 1       In      0+4*x^1+6*x^1*x^1
 1       Out     0+4*x^1+6*x^2

  2      In      0+4*x^1+6*x^2
  2      Fail    0+4*x^1+6*x^2
 1       Fail    0+4*x^1+6*x^2
0        Fail    0+4*x^1+6*x^1*x^1

Result   4x+6*x^2
         4x+6x*x
         2*(2+3x)*x

no
| ?- normal(2*x*(3*x + 2), N), order(N, O), basic(O, B).
N = 2*(1*x^1)*(3*(1*x^1)+2)
O = 1*1*2*(0+2+1*1*3*x^1)*x^1
```

```
B = 2*(2+3*x^1)*x^1

yes
| ?- sim(3 + x + 2*x + (x+1)*(x-1)).
Starting:       3+x+2*x+(x+1)*(x-1)

0       In      3+1*x^1+1*(1+1*x^1)*(-1+1*x^1)+2*x^1
0       Out     3+1*(-1+1*x^1)+1*x^1+2*x^1+1*(-1+1*x^1)*x^1

 1      In      3+1*(-1+1*x^1)+1*x^1+2*x^1+1*(-1+1*x^1)*x^1
 1      Out     2+1*x^1+3*x^1+-1*x^1+1*x^1*x^1

  2     In      2+1*x^1+3*x^1+-1*x^1+1*x^1*x^1
  2     Out     2+3*x^1+1*x^2

   3    In      2+3*x^1+1*x^2
   3    Fail    2+3*x^1+1*x^2
  2     Fail    2+3*x^1+1*x^2
 1      Fail    2+1*x^1+3*x^1+-1*x^1+1*x^1*x^1
0       Fail    3+1*(-1+1*x^1)+1*x^1+2*x^1+1*(-1+1*x^1)*x^1

Result  2+3x+x^2
        2+x+3x-x+x*x
        3+(-1+x)+x+2x+(-1+x)*x
        3+x+(1+x)*(-1+x)+2x

no
| ?- dx(3 + x + 2*x + (x+1)*(x-1)).
Starting:       0+1+2*1+((1-0)*(x+1)+(1+0)*(x-1))

0       In      3+1*(-1+1*x^1)+1*(1+1*x^1)
0       Out     3+1*x^1+1*x^1

 1      In      3+1*x^1+1*x^1
 1      Out     3+2*x^1

  2     In      3+2*x^1
  2     Fail    3+2*x^1
 1      Fail    3+2*x^1
0       Fail    3+1*x^1+1*x^1

Result  3+2x
        3+x+x
        3+(-1+x)+(1+x)

no
| ?- halt.
```