A short note on short differentiation programs in lisp, and a comment on logarithmic differentiation

Richard Fateman
University of California, Berkeley

September 27, 2001

1 Lisp and differentiation

By tradition and design, Lisp is alleged to be a particularly convenient programming language in which to represent algebraic expressions and in which to write a differentiation program.

Over the years, most books on Lisp have included programs for differentiation. Some have extended this example to provide fodder for problems for data-directed, functional, object-oriented, rule-based or other styles of programming[1].

Since the task of algebraic differentiation can be specified with varying amounts of detail, a programmer typically has options in choosing compromises with respect to size, efficiency, robustness, generality, extensibility, and other issues. Even full-featured computer algebra systems like Maple, Macsyma and Mathematica differ in their built-in ability to compute certain derivatives or represent them operationally.

This report has yet another program, one which we wrote to have fun with using logarithmic derivatives for the product rule, and for derivatives of powers. It has a certain charm in being slightly smaller in these parts, and in providing additional challenge for a subsequent simplification step[2].

2 Logarithmic differentiation: the idea

If you have forgotten (or never learned) logarithmic differentiation, it is just a way of easing the task of differentiating a messy product of powers by writing it out in a different way.

[1]Abelson/Sussman's *Structure and Interpretation of Computer Programs* is a modern example of this.

[2]When I was first hired to teach in a mathematics department and there was a discussion of whether to include "computer algebra" in a calculus course, some instructors objected on the grounds that something then would have to be left out. I suggested logarithmic differentiation!

First, recall that $D_x \log y = (1/y) D_x y$ so that, should you choose to do so, you can compute $D_x y$ by the formula

$$D_x y = y D_x \log y.$$

This initially looks like a loser, because you now have to differentiate $\log y$ which is more complex than the problem you started with.

Here's the trick: if you are given $y = u_1^{n_1} u_2^{n_2} \cdots u_k^{n_k}$ you can use the above formula to *decrease* the bookkeeping by "simplifying" $\log y$ before doing the differentiation on the right hand side. Recall that

$$\log y = n_1 \log u_1 + n_2 \log u_2 + \cdots n_k \log u_k.$$

so that in particular we can write out $D_x y$ as

$$D_x y = y \cdot \{\frac{n_1 D_x u_1}{u_1} + \frac{n_2 D_x u_2}{u_2} + \cdots + \frac{n_k D_x u_k}{u_k}\}.$$

The calculus texts assert that this is easier, at least for complicated y, and it is especially true if you can finesse the situation by continuing to use the small symbol y for its longer written-out definition.

Logarithmic differentiation suggests a neat formula for differentiation of indexed products of indefinite size: consider the differentiation of

$$p = \prod_{i=1}^{n} a_i.$$

This is easily expressed as

$$D_x p = p \sum_{i=1}^{n} \frac{D_x a_i}{a_i}.$$

Mathematica 3.0 doesn't know this formula, although Macsyma 2.2 does.

3 Program complexity and size

In this section we describe a simple program to implement this and much of the rest of the algorithm needed. The program, given in full below, does not do much error checking, and no simplification. A differentiation program for a "real" computer algebra system would rely on substantial simplification programs, and would likely have simplification intertwined with its processing to save time and avoid unnecessarily producing large intermediate expressions. One common optimization might be checking for sub-expressions being free of any dependency on the variable of differentiation, in which case the derivative is 0.

A commercial system would most likely provide a set of built-up tools to try to make it easier to add to the program than the "rules" we have here.

Our aim here is to illustrate low complexity judging by number of symbols, functions, and construction, and yet provide a program with modest generality, able to differentiate explicit algebraic expressions written in the usual parenthesized prefix form of lisp. For example, we can differentiate `(+ a b c d)` and we do not require `+` and `*` to be binary-only operators. We allow `sin, cos, exp, log, expt` and could add other functions easily. We do an indexed product as well.

How can we measure the size of a program? We attach a modified text leaving out the comments, shortening arbitrary identifiers to single letters, removing indexed product (since this is somewhat esoteric, and we only included it for fun). We also squeezed out all non-essential white space. We still start each function on a separate line and keep lines to 80 characters. This program is 528 characters (including 38 spaces) on 14 text lines. Not counting punctuation marks, there are 128 tokens. We could, of course, make it smaller by removing more of its capabilities. Using logarithmic differentiation shortens parts of it in minor ways.

Since this is not a serious program, it is primarily an illustration of how briefly one can express mathematical ideas, appropriately encoded, as programs.

```
;;
;; Compute the derivative of expression exp wrt variable var,
;; presuming exp is a conventional lisp expression and var is a
;; symbolic atom (like x).

(defun deriv (exp var)
  (if(atom exp)(if (eq exp var) 1 0) ; dx/dx =1, other atoms 0.
        ;; data-directed dispatch to a deriv pgm or the undef-deriv pgm
          (funcall (or (get (car exp) 'deriv) #'undef-deriv)
                       exp var)))

(defun undef-deriv (exp var) `(derivative ,exp ,var)) ;unknown fn deriv

;; establish a table of derivatives with appropriate use of
;; the chain rule.

;; Here are well-known functions of a single argument


(defun make-diff-rule  ;put data-directed chain rules in lisp system
    (operator stuff)
  (setf (get operator 'deriv)
    (compile () ;convert the lambda expression into a program
            `(lambda (exp var)
                (let ((arg (cadr exp)))
                  (list '* (subst arg 'arg ',stuff) (deriv arg var))))))

(make-diff-rule 'cos '(* -1 (sin arg)))
(make-diff-rule 'sin '(cos arg))
(make-diff-rule 'exp '(exp arg))
(make-diff-rule 'log '(expt arg -1))

;; etc for other 1-argument functions

;; traditional functions of a variable number of arguments: + and *

(setf (get '+ 'deriv)
  #'(lambda (exp var)
      `(+ ,@(mapcar #'(lambda(r)(deriv r var))(cdr exp)))))

;; the traditional way of generating u*v --> u*v'+ v*u'
;; but generalized to any number of multiplicands

(setf (get '* 'deriv)
  #'(lambda (exp var)
      `(+ ,@(maplist
             #'(lambda(r)  ;u'*[expression/u]
                 `(* ,(deriv (car r) var)
                     ,@(remove (car r) (cdr exp) :count 1)))
             (cdr exp)))))

;; example
;; (deriv '(* a b x) 'x) ==>(+ (* 0 B X) (* 0 A X) (* 1 A B))

;; Here's an alternative using "logarithmic derivatives".
;; that says dy/dx = y*d/dx (log(y)).
;; If y is a product u*v, then let us use log(y)=log(u)+log(v)
;; and then dy/dx = y* {(1/u)*du/dx +(1/v)*dv/dx}
;; including generalization to any number of multiplicands.

(setf (get '* 'deriv)
  #'(lambda (exp var)
      `(* ,exp
          (+ ,@(mapcar #'(lambda(r) `(* ,(deriv r var) (expt ,r -1)))
                              (cdr exp))))))
;; same example
;;( deriv '(* a b x) ==>
;; (* (* A B X) (+ (* 0 (EXPT A -1)) (* 0 (EXPT B -1)) (* 1 (EXPT X -1))))

;; a similar "log derivative" view of y=u^v gives dy/dx = y*d/dx(v*log(u))

(setf (get 'expt 'deriv)
  #'(lambda(exp var)
      `(* ,exp ,(deriv `(* ,(caddr exp) (log ,(cadr exp))) var))))

;;this makes really unsimplified results.. e.g. derivative of x^n produces.

;;'(* (EXPT X N)
;;   (* (* N (LOG X))
;;    (+ (* 0 (EXPT N -1)) (* (* 1 (EXPT X -1)) (EXPT (LOG X) -1)))))

;;which, if you study it, is a product:
;;  n * 1/log(x) * log(x) *  x^n  * x^(-1)
;;and indeed equivalent to .. (* n (expt x (+ n -1)))
;;
;; And here is a rule that provides us with a capability lacking in
;; Mathematica 3.0

(setf (get 'product 'deriv)
  #'(lambda(exp var)
      `(* ,exp
          (sum ,(deriv (cadr exp) var) ,@(cddr exp)))))

;;(deriv '(product (a_i x) i=1 y) 'x) ==>
;;(* (PRODUCT (A_I X) I=1 Y) (SUM (DERIVATIVE (A_I X) X) I=1 Y))
```

4 A short version of the program above

```
(defun d(e v)(if(atom e)(if(eq e v)1 0)
(funcall(or(get(car e)'d)#'undef)e v)))
```

```
(defun undef(e v)`(d,e,v))
(defun r(op s)(setf(get op'd)(compile()`(lambda(e v)(let((x(cadr e)))
(list'*(subst x'x',s)(d x v)))))))
(r'cos'(* -1(sin x)))
(r'sin'(cos x))
(r'exp'(exp x))
(r'log'(expt x -1))
(setf(get'+'d)#'(lambda(e v)`(+,@(mapcar #'(lambda(r)(d r v))(cdr e)))))
(setf(get'*'d)
#'(lambda(e v)`(*,e(+,@(mapcar #'(lambda(r)`(*,(d r v)(expt,r -1)))(cdr e))))))
(setf(get'expt'd)#'(lambda(e v)`(*,e,(d`(*,(caddr e)(log,(cadr e)))v))))
```