

Algorithm Differentiation in Lisp: ADIL

Richard Fateman
Computer Science
University of California
Berkeley, CA, USA

September 4, 2009

Abstract

Algorithm differentiation is a technique used to take a program F computing a numerical function of one argument $F(x)$ and transform it into another program G that returns a pair, $\langle F'(x), F(x) \rangle$ where by $F'(x)$ we mean the derivative of F with respect to its argument, evaluated at x . That is, we have a program AD that takes as input a program, and returns another: $G := \text{AD}(F)$. This task has been specialized so that F may be expressed in a subset of any of several prominent programming languages. The task is especially simple in Lisp. We show why this is the case, how the results can be extended easily, and how, in spite of the brevity of the “ADIL” program, we can provide features which are unsupported in other AD programs. In particular, recursive functions are easily accommodated.

1 Introduction to Automatic Algorithm Differentiation (AD)

For a complete introduction to the topic of Automatic Differentiation as used in this paper we recommend a visit to the website www.autodiff.org. Here you can find a collection of links to the standard literature, including AD programs and applications. There are also links to recent conference publications and articles. There are two variation of AD techniques, forward and “reverse” differentiation. Although we have programmed both, this paper deals only with the forward version.

1.1 A brief tangent

AD is not the same as “symbolic differentiation of algebraic expressions,” a well-known task for symbolic languages such as Lisp. In order to distinguish the situation, we note that the compact representation for a symbolic differentiation program is cited as one driving application for the original Lisp language design. In a conventional setup for Lisp, $x \sin x \log x + 3$ would be written as `(+ (* x (sin x) (log x)) 3)`¹. A brief program² can differentiate this with respect to x to get

```
(+ (* (* x (sin x) (log x))
      (+ (* 1 (expt x -1)) (* (* (cos x) 1) (expt (sin x) -1))
        (* (* (expt x -1) 1) (expt (log x) -1))))
  0)
```

¹If you are uncomfortable with this notation, many parsers from more “conventional” infix notation are available. The interface may be as simple as using a normal Lisp program text, but enclosing such infix expressions in markers, for example `[x*sin(x)*log(x)+3]`.

²see Appendix 1

an answer which is correct but unsimplified and clumsy in appearance, even if it were converted to more conventional infix.

A proper “computer algebra system” or CAS would contain a (much longer) simplification program to take the symbolic result and reduce it to simpler terms, removing multiplications by 1 and additions with 0, perhaps finding common factors, etc.

1.2 An analogy in CAS

AD does not simulate the CAS feature just described. It does not offer explicit symbolic differentiation. AD technology is far more analogous to another CAS feature: arithmetic with truncated Taylor series. AD converts a conventional program treating each conventional scalar value u to a program operating on a pair such as $p = \langle u, v \rangle$ representing that scalar function u and its derivative v with respect to an implicit parameter, say t . In particular, a constant has $v = 0$, and the parameter t has derivative $v = 1$. As a series, the pair p represents $s = u + vt + \dots$. The equivalence to computation with Taylor series provides guidance as to how to compute with these pairs. (If there is any question, a CAS Taylor series program can generally provide an appropriate result.) The Taylor expansion of $\cos s$ is $\cos u - \sin u vt + \dots$ and so $\cos p = \langle \cos u, -v \times \sin u \rangle$.

The extension of this idea to higher derivatives is straightforward although tedious. Here is how it would work: we use triples, e.g. where the second derivative is called w : If $p = \langle u, v, w \rangle$, the operation of \cos yields:

$$\cos p = \cos u - \sin u vt - \frac{(\cos u v^2 + 2 \sin u w) t^2}{2} + \dots$$

The result triple consists of the coefficients of different powers of t . A program computing those coefficients naturally will need to compute $\sin u$ and $\cos u$ only once.

2 AD in Lisp

There are two fundamentally different approaches to building a forward AD system, and a third that looks plausible, at least briefly, for Lisp.

1. We can take a program P in (more-or-less) unchanged form and by *overloading* each of its operations, make P executable in another domain in which scalars are replaced by pairs.
2. We can transform the source code of P into another program in which computations are “doubled” in a particular way to compute the original function as well as the requested derivative.
3. A third technique, usually ignored, could be used in Lisp: write a new version of Lisp’s `eval` to do AD. This is not a good idea, even in Lisp, because most Lisp programs don’t use `eval`: they are compiled into assembler. The technique of overloading operators is conceptually similar to patching the `eval` program anyway, and so we discard this option.

We provide code for both approaches. Because it inherits all the Lisp facilities not specifically shadowed by AD variations, the overloading method has the advantage of covering just about everything one can do in Lisp. The code is structured in a way that is fairly easy to understand.

Overloading is unfortunately not as speedy as the second method, which tends to run at a (small) multiple of the original code’s speed. But source code transformation is ticklish, requiring attention to almost every aspect of the language. Thus our version `dcomp` does not cover as much of Lisp as overloading. The two first techniques can, fortunately, be used together. That is, a large program can be run mostly unchanged. If there is a bottleneck, some sub-part can be compiled to remove some of the run-time overhead.

Regardless of the programming technique, one part of the task is to represent the pair as a some kind of compound object, perhaps `vector`, a `defstruct`, a Common Lisp Object System (CLOS) object, or just a list. Each would be easily distinguished from a conventional scalar, should that be necessary, and each has space for a value and one (or possibly more) derivatives. For source-code transformation it may be adequate to just make near-duplicate names, e.g. if the original program has variable `v`, generate `v_diff_x` (etc.)

2.1 Overloading

Here we overload the arithmetic operators and let the rest of the language be inherited from Lisp's standard implementation. In ANSI standard Common Lisp, the accepted route to this is to establish a `package` to "shadow" the overloaded operations. We named the package `ga` for Generic Arithmetic, and can specify all the operations that are distinct as implemented for AD. All else is the same as in the standard `common-lisp` package. Here's the beginning of that package declaration. It could be extended easily. (For example, to add the hyperbolic tangent requires only one line).

```
(defpackage :ga ;generic arithmetic
  (:shadow "+" "-" "/" "*" "expt"      ;binary arith
   "=" "/=" ">" "<" "<=" ">="      ;binary comparisons
   "sin" "cos" "tan"                  ;... more trig
   "atan" "asin" "acos"               ;... more inverse trig
   "sinh" "cosh" "atanh"              ;... more hyperbolic
   "expt" "log" "exp" "sqrt"          ;... more exponential, powers
   "1-" "1+" "abs"                    ;... odds and ends
  )
  (:use :common-lisp))
```

The implementation of a consistent shadowing of Lisp arithmetic `+` and `*`, as well as comparisons is complicated by the fact that these operators can take any number of operands (including in some cases zero), in the "obvious" extension of the concept. Thus we wrote generic programs like `two-arg-*` for multiplying each possible pair of types, and then melded them together with an n -ary programs like `*`. Since the programs for each operator are very nearly the same, we wrote macro programs (program-writing programs) that helped us along. Thus to write the bulk of the "`*`" sub-programs we wrote (`defarithmetic *`). The comparison and single-argument programs are easier. We include each of them by, for example (`defcomparison /=`) for the not-equal, and (`r sin (cos x)`) to tell ADIL about `sin` and its derivative.

2.2 Using ADIL

Consider the function $f(x) = x \times \sin x \times \log x + 3$ written in Lisp as

```
(defun f(x) (+ (* x (sin x) (log x)) 3)).
```

We can evaluate f at a point x where x is 1.23 by (`f (df 1.23 1.0)`) which returns

```
<3.2399835288524628d0, 1.2227035>
```

In this system we define `df` as a constructor for a pair of numbers. This pair is display in angle-brackets. $p = \langle 1.23, 1.0 \rangle$. Note that the program `f` has not changed *at all*; applying the function to a different type of argument, and running the program within the `:ga` package is all that is needed.

A more interesting program is this one.

```
(defun s(x) (if (< (abs x) 1.0d-5) x
  (let ((z (s (* -1/3 x))))
    (-(* 4 (expt z 3))
      (* 3 z))))))
```

While it is not obvious, this computes an approximation to $\sin x$ by applying an identity recursively. That is, $\sin x$ is a polynomial $4z^3 - 3z$ in $z = \sin(-x/3)$. For small enough x , $\sin x = x$. Let us try it out by typing `(s (df 1.23 1))`. We get

```
<0.942489, 0.33423734>
```

This not only provides a value for `sin(1.23)` but also computes 0.33423734, the second part of the pair, which happens to be `cos(1.23)`. ADIL, starting with `s`, ran a program that computed `sin()` and also ran a program that consequently computed `cos()` because it ran the *derivative of the program that computes that looked like it computed sin*.

The classic recursive program in Lisp is factorial:

```
(defun fact(x) (if (= x 1) 1 (* x (fact (1- x)))))
```

which we modify to

```
(defun fact(x) (if (= x 1) (df 1 0.422784335098d0) (* x (fact (1- x)))))
```

whose peculiar base case is now the value one, with a peculiar derivative. We do this so as to make it correspond to the derivative of the Gamma function³. With this form we can provide not only the factorial but its “derivative” (at least at integer points).

In these examples we have not modified Lisp syntax at all. Anything *else* from Lisp is imported without attention or comment. Thus to compute and print ten $\sin x$ values, we can use the iterator `dotimes` as in `(dotimes (i 10) (print (s (df i 1))))`.

2.3 Newton Iteration, or, Why is AD useful?

The point is that if we are given a complicated function $F : R \rightarrow R$ arranged as an expression, and all the sub-functions “cooperate” properly, we can feed in a pair $\langle c, 1 \rangle$, representing the expression x and its derivative with respect to x , namely 1, each evaluated at $x = c$. We get out pairs $\langle f, f' \rangle = F(\langle c, 1 \rangle)$ where the latter is the pair $f = F(c)$, and $f' = D_x(F(x))|_{x=c}$. and this may be exactly what we want in an application. These are discussed in papers available via www.autodiff.org. Here we take one of the simplest but non-trivial examples, but perhaps the easiest to motivate, namely Newton iteration.

For example, to converge to a root in a Newton iteration for $f(z) = 0$ given an initial guess c_0 or $t_0 = \langle c_0, 0 \rangle$, we compute $F(t_i) = \langle f(t_i), f'(t_i) \rangle$. Then the next iteration $t_{i+1} = t_i - f(t_i)/f'(t_i)$. If this is not sufficiently accurate we consider repeating with $\langle f, f' \rangle = F(t_{i+1})$, etc.

In this Newton iteration program, which by its nature must know that it is getting a value and derivative, we have used three functions particular to ADIL, namely `df-p` a predicate which will return true if applied to a `df` object, as well as the two selection functions, `df-f` and `df-d` for extracting the value and derivative respectively from a `df` object.

The program is shorter than the explanation.

```
;; Newton iteration: (ni fun guess)
;; usage: fun is a function of one argument
;; guess is an estimate of solution of fun(x)=0
;; output: a new guess. (Not a df structure, just a number)
```

```
(defun ni (f z) ;one Newton step
  (let* ((pt (if (df-p z) z (df z 1))) ; make sure init point is a df
         (v (funcall f pt))) ;compute f, f' at pt
    (df-f (- pt (/ (df-f v)(df-d v))))))
```

³a continuous version of factorial well known among the “special functions”.

As a simple example, consider $f(x) = \sin(1 + 2x)$ or
(defun f(x)(sin(+ 1 (* 2 x))))). then

```
(setf h 2.0d0);; just a guess
(setf h (ni 'f h)) ;; one step
(setf h (ni 'f h)) ;; another step
;; h converges to 1.0707963267948966d0
```

A more useful version of Newton iteration might return the value of $f(h)$, too. Then the residual and the derivative can be taken into account in testing whether the Newton iteration has converged sufficiently. That program would look like:

```
(defun ni2 (f z)
  (let* ((pt (if (df-p z) z (df z 1)))
         (v (funcall f pt))) ;compute f, f' at pt
    (values
     (df-f (- pt (/ (df-f v)(df-d v)))) ;the next guess
     v))) ; the residual and derivative
```

;; A harder test for Newton iteration is this function,

```
(defun test(x)(+ (* 1/3 x) 1 (sin x))) ;; f(x) = x/3+1+sin x.
;; which is good if you start close enough to -0.8 or -3.2 or -5.4
;; but ni fails for most other initial guesses.
```

Now that we have a program for only one step, we can show a program that can use it to find the zero. This is a ticklish proposition because sometimes this iteration does not converge. We have to use some stopping heuristic. We could stop when two successive iterations are close in terms of relative or absolute error, or use some other measure.

A particularly simple iteration driver program uses `ni2` which returns the value of the residual, and so we test this residual, or quit after some count is exceeded.

```
(defun run-newt2(f guess &key (abstol 1.0d-8) (count 18)) ;; Solve f=0
  ;; It looks only at the residual.
  (dotimes (i count ;; do at most count times. failure prints msg
            (error "~%Newton quits after ~s iterations: ~s" count guess))
    (multiple-value-bind
      (newguess v)
      (ni2 f guess)
      (if (< (abs (df-f v)) abstol)
          (return newguess)
          (setf guess newguess)))))
```

2.4 What about speed?

The generic arithmetic (GA) system produces code that is slower than ordinary Lisp code, especially if that ordinary Lisp is “optimized”⁴

⁴Note that without type declarations and compilation, ordinary Lisp already has its own burden of generic arithmetic. Short and long (arbitrary length) integers, single or double floats, rationals, and complex numbers, as well as all plausible combinations are handled by standard ANSI Common Lisp. Indeed, code in the ADIL system using overloading also works for all these number types as much as in Common Lisp.

With appropriate instructions to the compiler Common Lisp arithmetic can be compiled to “non-generic” straight-line floating-point code, comparable to that of other high-level languages.

How much of a speed difference is there? On a variety of benchmarks involving mostly computations of how to dispatch-to-the-right-method, using the generic arithmetic for ADIL seems to be about a factor of 10 over “normal Lisp” and perhaps a factor of 50 over “optimized” Lisp (constrained to double-floats.) On tests where most of the work is done in subroutines such as `sin` or `log`, the difference is much less: the `log` routine takes the same time whether it is called from the `ga` package or from the `user` package. On tests where most of the operations are *other than arithmetic* such as looping over indexes, the `ga` programs run at full speed because they are in fact identical.

2.5 Source code transformation

As mentioned earlier, there is another part of ADIL. We wrote a program called `dcomp` that can compile programs, in-line, in a restricted language subset of Lisp. The subset is essentially that of functional-style arithmetic programs. In this situation, benchmarking a simple example suggests that the comparison between the ordinary Lisp for computing a function f and the ADIL Lisp for computing a function f and also its derivative is about a factor of two, which is about the best you would expect if you assume that evaluating the derivative is roughly similar in cost to the function itself.

This experiment computes $3 + z * (4 + z)$ where $z = \sin x$. In Lisp this looks like
`(defun kk(x) (setf z (sin x))(+ 3.0d0 (* z (+ 4.0d0 z))))`.

A nicer version binds `z` locally, as shown below. The `dcomp` version is shown with the `defdiff` defining form. All code is run through the Lisp compiler before timing.

```
;; each timing is a run of 10000 in a compiled loop
```

```
package
|
|
user:(defun kk(x &aux z)(declare (double-float x z)(optimize (speed 3)(safety 0)))
      (setf z (sin x))(+ 3.0d0 (* z (+ 4.0d0 z)))) ;;optimized version
user: (defdiff kz(x)(progn (setf z (sin x))(+ 3.0d0 (* z (+ 4.0d0 z))))))

user: (kk 1.2d0) -->    30 ms /no derivs!
user: (kz 1.2d0) -->    70 ms /with derivs!!
```

```
ga:(defun kk(x &aux z) (setf z (sin x))(+ 3.0d0 (* z (+ 4.0d0 z))))
```

```
ga: (k (df 1.2d0)) --> 40100 ms /with derivs
ga: (k ( 1.2d0 ) --> 1523 ms /no derivs!
```

The body of `kz` can be examined by tracing the internal program `dc`:

```
(lambda (g94)
  "(progn (setf z (sin x)) (+ 3.0d0 (* z (+ 4.0d0 z)))) wrt x"
  (declare (double-float g94))
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (let ((t102 0.0d0) (f101 0.0d0)
        (t100 0.0d0) (f99 0.0d0)
        (t98 0.0d0) (f97 0.0d0)
```

```

(z_DIF_x 0.0d0) (t96 0.0d0)
(f95 0.0d0))
(declare (double-float t102 f101 t100 f99 t98 f97 z_DIF_x t96 f95))
(setf f95 (sin g94))
(setf t96 (cos g94))
(setf z f95)
(setf z_DIF_x t96)
(setf t98 0.0d0 f97 3.0d0)
(setf t100 z_DIF_x f99 z)
(setf t102 0.0d0 f101 4.0d0)
(setf t102 (+ z_DIF_x t102))
(setf f101 (+ z f101))
(setf t100 (+ (* f101 t100) (* t102 f99)))
(setf f99 (* f101 f99))
(setf t98 (+ t100 t98))
(setf f97 (+ f99 f97))
(df f97 t98))

```

What else can `dcomp` do? In addition to the usual arithmetic operations and built-in functions (`sin`, `cos`, `log`, etc.) as declared in the generic arithmetic, `dcomp` can handle `if`, `progn`, `setf`. Functions defined with `defdiff` take only scalar arguments, not `df` structures. They return only `df` structures. The derivative is always with respect to the first formal argument. Thus `(defdiff f(x y z)(cos (+ x y z)))` computes the derivative with respect to `x` only. Initially we wrote a version of `dcomp` which system allowed recursion, but full support led to too much complexity and was taken out. A function like `f` above can be used directly by the generic arithmetic system, and if suitable, will be substantially faster. The `dcomp` file is about 266 lines of code, including comments.

3 Why use Lisp?

For fans of Lisp, there is no question that one motivation is to show how easy it is to implement in Lisp. Lisp provides a natural representation for programs as data and a natural form for writing programs that write programs, which is what we do in ADIL. The code is short, and is in ANSI standard Common Lisp. Since it is not using the obvious idioms of introductory Lisp, for those with only the most cursory familiarity with Lisp, it illustrates that Lisp is more than `CAR` and `CDR`. In fact we did not use those parts of Lisp at all. For persons only slightly familiar with Lisp, but familiar with AD, the use of Lisp shows how much shorter some of the ideas of automatic differentiation can be when presented in a Lisp context.

If you care not a whit about Lisp or implementation strategies, you may prefer to refer to the recently-revised online documentation for ADIFOR 2.0 version D, some 99 pages, and read in detail how programs to be differentiated must be distinguished from ordinary FORTRAN (77) programs. Using ADIFOR requires declaring and marking program variables, adjusting numerous parameters, perhaps revising the FORTRAN in order to obey various restrictions, and following detailed guidelines on the use of these programs.

Since the Lisp programs are relatively short, simple and entirely open to view and are entirely standard ANSI Common Lisp, they should run in any ANSI-conforming implementation. The flexibility one gets is apparent by looking at the code in which Lisp macro-definitions have made the addition of new derivative information simple. For example, to insert a new rule for differentiation of $\tanh x$ one adds "`tanh`" to the `shadow` list, and execute

```
(r tan (expt (cosh x) -2)).
```

By comparison, large and monolithic systems are relatively rigid, and require some effort to port, extend or optimize: the original designers must anticipate the spectrum of possible choices, perhaps freezing some

choices, and then describe all the variants in detail. The user must then read the details, and hope there are no bugs; the user is unlikely to be able, in any case, to repair them. Some of the restrictions of the large systems seem to be arbitrary—perhaps a small point, but it did not occur to us to have to exclude recursive functions, although recursion is a feature lacking in ADIFOR.

Finally, we wish to point out that Lisp is perfectly adequate for expressing numeric computation; some sophisticated compilers are available for generating efficient code.

3.1 Comments on the usefulness of AD

Shouldn't AD (whether of Lisp or not) be widely used since it appears to be so useful? Our understanding is that getting computational scientists to adopt the relatively unfamiliar technique of AD at all, rather than using finite differences or programming a separate derivative algorithm, already presents a barrier. There are simpler (but generally far less accurate and perhaps slower) ways of approximating derivatives of “programs” with finite differences. And further complicating *our* particular “sales pitch” is the implicit assumption we make that the computation to be differentiated was originally written in Lisp, or that people are willing to see their programs translated (perhaps automatically) into Lisp. For a variety of reasons people are far more likely to wish to differentiate FORTRAN.

The computational scientists may be attracted to AD only after the having written large programs (essentially treated as black boxes). AD comes into the picture in using these programs as modules in an optimization tool, or for sensitivity analysis. The optimization tools then may require separate modules to compute values for functions and derivatives. So a typical example might be a “function” F from computational fluid dynamics that is defined as a FORTRAN program (or a C program). The function might, for example, represent a solution method for a differential equation. The goal is to produce the moral equivalent of FPRIME, or a function that produces the pairs referred to earlier.

That's what the AD people have been trying to support, ADIFOR, for FORTRAN, ADOL-C for C or C++, etc.

It would be wrong to think that the AD programs for FORTRAN, C, C++ generally work on “black box” programs in those languages, automatically. They may reduce the programming effort to take a program and produce a “derivative” program considerably, say reducing the time from a year to a week. A visit to the previously mentioned <http://www.autodiff.org> website has links to some applications illustrating the level of human effort to build automation tools, document them for others to use, and then the continued effort and partnerships needed to use them. While the goal has been to differentiate nearly any program written in FORTRAN or C, it seems that some attention is required for success.

As for our own tools, we could, oddly enough, convert FORTRAN to Common Lisp using a program `f2cl` and try to differentiate that, and we could easily print out equivalent FORTRAN or C (etc.).

However, to keep this paper brief, we start and end with the Lisp; the integration of tools in Lisp is quite good, even if you are addicted to graphical interactive development environments. We ordinarily convert the AD code into assembly language without ever leaving the Lisp system.

There may be additional issues arising in ADIL if it enjoys wider use, but we are confident that the structure we have set out is consistent with solving the forward-differentiation AD task for Lisp. The major barrier may be convincing a computational scientist to write serious numerical code in Lisp. (or to continue to work on the code when the FORTRAN is translated to Lisp).

4 A brief defense of Forward Differentiation for ADIL

Consider a space of “differentiable functions evaluated at a point c .” In this space we can represent a “function f at a point c ” by a pair $\langle f(c), f'(c) \rangle$. That is, in this system every object is a pair: a value $f(c)$ and the derivative with respect to its argument $D_x f(x)$, evaluated at c . (written as $f'(c)$).

For a start, note that every number n is really a special case of its own Constant function, $C_n(x)$ such that $C_n(x) = n$ for all x . $C_3(x)$ is thus $\langle 3, 0 \rangle$. The constant π is $t_1 = \langle 3.14159265 \dots, 0.0 \rangle$, which represents a function that is always π and has zero slope. The object $t_2 = \langle c, 1 \rangle$ represents the function $f(x) = x$ evaluated at c . At this point we must be clear that all our functions are functions of the same variable, and that furthermore we will be fixing a point $x = c$ of interest. It does not make sense to operate collectively on $\langle f(y), D_y f(y) \rangle$ at $y = a$ and $\langle g(x), D_x g(x) \rangle$ at $x = b$.

We can operate on these objects. For example, \sin operating on t_2 is the pair $\langle \sin(c), \cos(c) \rangle$. In general, $\sin(\langle a, a' \rangle)$ is $\langle \sin(a), \cos(a) \times a' \rangle$.

We can compute other operations unsurprisingly, as, for example the sum of two pairs: $\langle a, a' \rangle + \langle b, b' \rangle = \langle a + b, a' + b' \rangle$. *Note, we have abused notation somewhat: The “+” on the left is adding in our pair-space, the “+” on the right is adding real numbers. Such distinctions are important when you write programs!* Similarly, the product of two pairs in this space is $\langle a, a' \rangle \times \langle b, b' \rangle = \langle a \times b, a \times b' + a' \times b \rangle$. This can be extended to many standard arithmetic operations in programming languages, at least the differentiable ones [4]. AD implementors seek to find some useful analogy for other operations which do not have obvious derivatives. Falling in this category are most data structure manipulations, calls to external routines, loops, creating arrays, etc.⁵ In ADIL, these mostly come free.

5 Is AD patented?

At least two US patents have been issued on AD. 6,223,341 and 6,397,380 also see application 20030023645 (2001). There are free on-line copies of these patents which appear to post-date the prominently-published prior art.

6 Conclusion

AD programs can be easily written and executed in Lisp. This is not surprising. What is perhaps surprising is that no one has written this paper earlier.

We have experimented with alternative approaches, both for forward and reverse differentiation, but this approach seems to be a good combination of brevity, clarity, extensibility, generality, and efficiency.

Appendix 1

First we display a seven line Lisp differentiation program (similar to many others written over the years) that is distinguished by brevity. This one was posted on a Lisp newsgroup by Pisin Bootvong, some time ago, and makes use of the Common Lisp object system (CLOS) and `destructuring-bind` nicely:

```
(defmethod d ((x symbol) var) (if (eql x var) 1 0))
(defmethod d ((x number) var) 0)
(defmethod d ((expr list) var)
  (destructuring-bind (op e1 e2) expr
    (case op
      (+ '(+ ,(d e1 var) ,(d e2 var)))
      (* '(+ (* ,(d e1 var) ,e2) (* ,e1 ,(d e2 var)))))))
```

Here's a more elaborate, but still short Lisp program with more capabilities and greater extensibility [1].

⁵It is a mistake to `declare` variables to be (say) double-floats, when in the ADIL framework they will be `df` structures. We are not aware of any other systematic problems.

```

(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)) ;;anything unknown goes here

(defmacro r(op s)'(setf(get ',op 'd) ;;define a rule to diff operator op!
                    (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)) ;; etc,
(setf(get '+ 'd) ;; rules for +, *, expt must handle n args, not just 1
      #'(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))))

```

Other programming languages, especially ones with a “functional” approach, can usually handle this task nicely, but the major issue (and one not addressed here) is simplification of the result. Trying to write a simplifier adds substantially to the programmer’s burden. The differentiation program in a computer algebra system like Macsyma is much larger, not only because it handles a larger class of functions, and trades code-size for speed, not generating such naive forms, and simplifying along the way.

Another program whose specifications seem superficially like the previous one does not build any list structure. It assume that the result of interest is the value of the derivative at a point, not its symbolic representation. Thus the `d` function takes another argument, the point `p`. It returns the derivative of the `expr` with respect to the `var` at the point `p`. We are no longer returning lists. Note that we now have to evaluate expressions involving the variable, for which we have defined the `val` method. Although such a program can be written entirely using the skeleton of the previous few programs, we illustrate a different approach using generic programming (a handle on object-oriented programs) supported in Common Lisp.

```

(defmethod d ((x symbol) var p) (if (eql x var) 1 0))
(defmethod d ((x number) var p) 0)
(defmethod d ((expr list) var p)
  (destructuring-bind (op e1 e2) expr
    (case op
      (+ (+ (d e1 var p) (d e2 var p)))
      (* (+ (* (d e1 var p) (val e2 var p)) (* (val e1 var p) (d e2 var p))))))

(defun val(expr var p)(funcall '(lambda (,var) ,expr) p))

```

This program’s `case` statement would have to be expanded for other two-argument functions, and would also need to be altered for one-argument functions like `sin` and `cos`.

Appendix 2: Generic Arithmetic, AD

;; Automatic Differentiation code for Common Lisp

```

;; Richard Fateman, November, 2005
;; This is all provided in the context of a Generic Arithmetic Package.
;; Package based in part on code posted on comp.lang.functional newsgroup by
;; Ingvar Mattsson <ing...@cathouse.bofh.se> 09 Oct 2003

(defpackage :ga ;generic arithmetic
  (:shadow "+" "-" "/" "*" "expt"          ;binary arith
           "=" "/=" ">" "<" "<=" ">="    ;binary comparisons
           "sin" "cos" "tan"                ;... more trig
           "atan" "asin" "acos"             ;... more inverse trig
           "sinh" "cosh" "atanh"           ;... more hyperbolic
           "expt" "log" "exp" "sqrt"        ;... more exponential, powers
           "1-" "1+" "abs"
   )
  (:use :common-lisp))

(in-package :ga)

;; df structure for f,d: f is function value, and d derivative, default 0
(defstruct (df (:constructor df (f &optional (d 0)))) f d )

;; print df structures with < , >
(defmethod print-object ((a df) stream)(format stream "<~a, ~a>" (df-f a)(df-d a)))

;;function ARITHMETIC-IDENTITY: When fed an operator and a non-nil
;;argument, it returns a value for unary application. What does (+ a) mean?
;;A nil arg means there were NO operands. What does (+ ) mean.
;;It is used only by defarithmetic, which in turn helps
;; us to write out + * - / of arbitrary number of args.

(defmacro arithmetic-identity (op arg)
  `(case ,op
    (+ (or ,arg 0))
    (- (if ,arg (two-arg-* -1 ,arg) 0))
    (* (or ,arg 1))
    (/ (or ,arg (error "/" given no arguments)))
    (expt (or ,arg (error "expt given no arguments")))
    (otherwise nil))) ;binary comparisons?

(defun tocl(n) ; get corresponding name in cl-user package
  (find-symbol (symbol-name n) :cl-user))

(defmacro defarithmetic (op)
  (let ((two-arg
        (intern (concatenate 'string "two-arg-" (symbol-name op))
                :ga ))
        (cl-op (tocl op)))
    `(progn
      (defun ,op (&rest args)

```

```

      (cond ((null args) (arithmetic-identity ',op nil))
            ((null (cdr args))(arithmetic-identity ',op (car args)))
            (t (reduce (function ,two-arg)
                       (cdr args)
                       :initial-value (car args))))))
(defgeneric ,two-arg (arg1 arg2))
(defmethod ,two-arg ((arg1 number) (arg2 number))
  (,cl-op arg1 arg2))
(compile ',two-arg)
(compile ',op)
',op)))

(defarithmetic +) ;; defines some of + programs. See below for more
(defarithmetic -)
(defarithmetic *)
(defarithmetic /)
(defarithmetic expt)

;; defcomparison helps us generate numeric comparisons: 2 args
;; and n-arg. CL requires they be monotonic.
;; That is in Lisp, (> 3 2 1) is true.

(defun monotone (op a rest)(or (null rest)
                               (and (funcall op a (car rest))
                                    (monotone op (car rest)(cdr rest)))))

(defmacro defcomparison (op)
  (let ((two-arg (intern (concatenate 'string "two-arg-"
                                     (symbol-name op)) :ga ))
        (cl-op (tocl op)))
    `(progn
      (defun ,op (&rest args)
        (cond ((null args) (error "~s wanted at least 1 arg" ',op))
              ((null (cdr args)) t) ;; one arg e.g. (> x) is true
              (t (monotone (function ,two-arg)
                           (car args)
                           (cdr args)))))

      (defgeneric ,two-arg (arg1 arg2))
      (defmethod ,two-arg ((arg1 number) (arg2 number)) (,cl-op arg1 arg2))
      (defmethod ,two-arg ((arg1 df) (arg2 df)) (,cl-op (df-f arg1)(df-f arg2)))
      (defmethod ,two-arg ((arg1 number) (arg2 df))(,cl-op arg1(df-f arg2)))
      (defmethod ,two-arg ((arg1 df) (arg2 number))(,cl-op (df-f arg1) arg2 ))
      (compile ',two-arg)
      (compile ',op)
      ',op)))

(defcomparison >) ;;provides ALL the comparison methods
(defcomparison =)

```

```

(defcomparison /=)
(defcomparison <)
(defcomparison <=)
(defcomparison >=) ;; that's all

;; extra + methods specific to df
(defmethod ga::two-arg-+ ((a df) (b df))
  (df (cl:+ (df-f a)(df-f b))
      (cl:+ (df-d a)(df-d b))))
(defmethod ga::two-arg-+ ((b df)(a number))
  (df (cl:+ a (df-f b)) (df-d b)))
(defmethod ga::two-arg-+ ((a number)(b df))
  (df (cl:+ a (df-f b)) (df-d b)))

;;extra - methods
(defmethod ga::two-arg-- ((a df) (b df))
  (df (cl:- (df-f a)(df-f b))
      (cl:- (df-d a)(df-d b))))
(defmethod ga::two-arg-- ((b df)(a number))
  (df (cl:- (df-f b) a) (df-d b)))
(defmethod ga::two-arg-- ((a number)(b df))
  (df (cl:- a (df-f b)) (df-d (cl:- b))))

;;extra * methods
(defmethod ga::two-arg-* ((a df) (b df))
  (df (cl:* (df-f a)(df-f b))
      (cl:+ (cl:* (df-d a) (df-f b)) (cl:* (df-d b) (df-f a)))))
(defmethod ga::two-arg-*
  ((b df)(a number)) (df (cl:* a (df-f b)) (cl:* a (df-d b))))
(defmethod ga::two-arg-*
  ((a number) (b df)) (df (cl:* a (df-f b)) (cl:* a (df-d b))))

;; extra divide methods
(defmethod ga::two-arg-/ ((u df) (v df))
  (df (cl:/ (df-f u)(df-f v))
      (cl:/ (cl:+ (cl:* -1 (df-f u)(df-d v))
                  (cl:* (df-f v)(df-d u)))
            (cl:* (df-f v)(df-f v)))))
(defmethod ga::two-arg-/ ((u number) (v df))
  (df (cl:/ u (df-f v))
      (cl:/ (cl:* -1 (df-f u)(df-d v))
            (cl:* (df-f v)(df-f v)))))
(defmethod ga::two-arg-/ ((u df) (v number))
  (df (cl:/ (df-f u) v)
      (cl:/ (df-d u) v)))

;; extra expt methods
(defmethod ga::two-arg-expt ((u df) (v number))
  (df (cl:expt (df-f u) v)
      (cl:expt (df-d u) v)))

```

```

      (cl:* v (cl:expt (df-f u) (cl:1- v)) (df-d u))))
(defmethod ga::two-arg-expt ((u df) (v df))
  (let* ((z (cl:expt (df-f u) (df-f v))) ;;z=u^v
        (w ;; u(x)^v(x)*(dv*log(u(x))+du*v(x)/u(x))
          ;; = z*(dv*log(u(x))+du*v(x)/u(x))
         (cl:* z (cl:+
                  (cl:* (cl:log (df-f u)) ;log(u)
                        (df-d v)) ;dv
                  (cl:/ (cl:* (df-f v)(df-d u)) ;v*du/ u
                        (df-f u))))))
    (df z w)))
(defmethod ga::two-arg-expt ((u number) (v df))
  (let* ((z (cl:expt u (df-f v))) ;;z=u^v
        (w ;; z*(dv*LOG(u(x))
         (cl:* z (cl:* (cl:log u) ;log(u)
                        (df-d v))))))
    (df z w)))

;; A rule to define rules, a new method for df, the old method for numbers
(defmacro r (op s)
  `(progn
    (defmethod ,op ((a df))
      (df (,(tocl op) (df-f a))
          (,(tocl '*) (df-d a) ,(subst '(df-f a) 'x s))))
    (defmethod ,op ((a number)) (,(tocl op) a))))

;; Add rules for every built-in numeric program.
;; Must insert the name in the shadow list too.
;; This is just a sampler.
(r sin (cos x)) ;; provides EVERYTHING ADIL needs about sin
(r cos (* -1 (sin x)))
(r asin (expt (+ 1 (* -1 (expt x 2))) -1/2))
(r acos (* -1 (expt (+ 1 (* -1 (expt x 2))) -1/2)))
(r atan (expt (+ 1 (expt x 2)) -1))
(r sinh (cosh x))
(r cosh (sinh x))
(r atanh (expt (1+ (* -1 (expt x 2))) -1))
(r log (expt x -1))
(r exp (exp x))
(r sqrt (* 1/2 (expt x -1/2)))
(r 1- 1)
(r 1+ 1)
(r abs x);; hm does this matter?

(defun re-intern(s p) ;; move expression to :ga package
  (cond ((or (null s)(numberp s)) s)
        ((symbolp s)(intern (symbol-name s) p))
        (t(cl-user::cons (re-intern (car s) p)
                          (re-intern (cdr s) p)))))

```

```

(defun cl-user::deval(r x p &optional (dx 1))
  (ga::deval (ga::re-intern r :ga)
             (ga::re-intern x :ga) p dx))

;; factorial with "right" derivative at x=1 to match gamma function

(defun fact(x) (if (= x 1) (df 1 0.422784335098d0) (* x (fact (1- x)))))

;; an integer (arbitrary! ) factorial
(defun ifact(x) (if (= x 1) x (* x (ifact (1- x)))))

;; compare fact, ifact to Stirling's approximation to factorial

(defun stir(n) (* (expt n n) (exp (- n))
                 (sqrt (* (+ (* 2 n) 1/3) 3.141592653589793d0))))

(defun ex(x)(ex1 x 1 15))

(defun ex1(x i lim) ;; generate Taylor summation for exp()
  (if (= i lim) 0 (+ 1.0d0 (* x (ex1 x (+ i 1) lim)(expt i -1)))))

'(defun ex1(x i lim) ;; generate Taylor summation exactly for exp()
  (if (= i lim) 0 (+ 1 (* x (ex1 x (+ i 1) lim)(expt i -1)))))

;; what you say depends on which package is current in your
;; top-level read-eval-print loop
;; in :user (setf one (ga::df 1 1))
;; in :user (deval '(dotimes (i 10)(print (fact (+ i one)))) 'x 'irrelevant)

;; in :ga (setf one (df 1 1))
;; in :ga (dotimes (i 10)(print (fact (+ i one))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

#|Some timing results suggest that we lose about a factor of 10 in
speed just using generic arithmetic, even if we are using (say)
floating point numbers, if we are doing mostly calls and returns, but
only trivial arithmetic. Under these conditions, if we actually use df
numbers, it is perhaps a factor of two further.

The big slowdown here is in generic over the built-in (but still generic)
Lisp arithmetic that allows floats, doubles, rationals, bignums.

If we compile with declarations for fixnum types among the lisp built-ins,
we improved by another 30 percent.

(time (dotimes (i 10)(slowmul 1000000 2 0))) ;compiled 19.38s in :ga

```

```

(time (dotimes (i 10)(slowmul 1000000 2 (df 0)))) ; 24.45s in :ga
(time (dotimes (i 10)(slowmul 1000000 2 0))) ;compiled .37s in :user
(time (dotimes (i 10)(slowmulx 10000.0d0 2.0d0 0.0d0))); .38 in :user declared doubles
(time (dotimes (i 10)(slowmulx 1000000 2 0))) ; .29 in :user declared fixnums

```

```

(defun slowmul(x y ans)(if (= x 0) ans (slowmul (1- x) y (+ y ans))))

```

```

(defun slowmulx(x y ans)
  (declare (fixnum x y ans) ;; or (double-float x y ans)
           (optimize (speed 3)(safety 0)(debug 0)))
  (if (= x 0) ans (slowmulx (1- x) y (+ y ans))))

```

```

|#

```

```

;; this is an interesting function that numerically computes (sin x)

```

```

(defun s(x) (if (< (abs x) 1.0d-5) x
                (let ((z (s (* -1/3 x))))
                  (-(* 4 (expt z 3))
                     (* 3 z))))))

```

```

#| (s (df 1.23d0 1.0)) computes both sin and cos of 1.23.

```

```

(time (dotimes (i 100) (s (df 100000.0d0 1)))) :ga 40ms
(time (dotimes (i 100) (s 100000.0d0 ))) :ga 20 ms
(time (dotimes (i 100) (s 100000.0d0 ))) :user 10ms av over more runs

```

```

;; we can compile this to run faster..

```

```

(defun ss(x) (declare (double-float x)
                     (optimize (speed 3)(safety 0) (debug 0)))
             (if (< (abs x) 1.0d-5) x (let ((z (ss (* #./ -1 3.0d0) x)))
                                       (-(* 4.0d0 (expt z 3))
                                          (* 3.0d0 z))))))

```

```

(time (dotimes (i 100) (ss 100000.0d0 ))) :user 8ms av over more runs

```

```

|#

```

```

;; newton iteration (ni fun guess)
;; usage: fun is a function of one arg.
;; guess is an estimate of solution of fun(x)=0
;; output: a new guess. (Not a df structure, just a number

```

```

(defun ni (f z) ;one newton step
  (let* ((pt (if (df-p z) z (df z 1))) ; make sure init point is a df
         (v (funcall f pt))) ;compute f, f'
    (df-f (- pt (/ (df-f v)(df-d v))))))

```

```

(defun ni2 (f z) ;one newton step, give more info
  (let* ((pt (if (df-p z) z (df z 1)))
         (v (funcall f pt))) ;compute f, f' at pt
    (format t "~%v = ~s" v)
    (values
     (df-f (- pt (/ (df-f v)(df-d v)))) ;the next guess
     v)))

(defun run-newt1(f guess &optional (count 18)) ;; Solve f=0
  ;; Look only at the guesses.
  ;; though if the derivative goes to 0, we are stuck
  ;; in the newton step.
  (let ((guesses (list guess))
        (reltol #.( * 100 double-float-epsilon))
        (abstol #.( * 100 least-positive-double-float)))
    (dotimes (i 6)(push (ni f (car guesses))
                       guesses)) ;; make 6 iterations.
    (incf count -6)
    (cond ((< (abs (car guesses)) abstol)
           (car guesses))
          ((< (abs(/ (- (car guesses)(cadr guesses))(car guesses))) reltol)
           (car guesses))
          ((<= count 0)
           (format t "%newt1 failed to converge; guess =~s" (car guesses)
                 (car guesses))
           (t (run-newt1 f (car guesses) count))))))

(defun run-newt2(f guess &key (abstol 1.0d-8) (count 18)) ;; Solve f=0
  ;; Looks only at the residual.
  (dotimes (i count)
    (format t "%Newton iteration not convergent after ~s iterations: ~s" count guess))
  (multiple-value-bind
    (newguess v)
    (ni2 f guess)
    (if (< (abs (df-f v)) abstol) (return newguess)
        (setf guess newguess))))

;;try (run-newt2 'sin 3.0d0)

```

Appendix 3 Dcomp

```

;;; -*- Mode: Lisp; Syntax: Common-Lisp; -*-
;;; dcomp, use instead of /or with/ generic3.lisp for ADIL, automatic differentiation
;;; Richard Fateman 11/2005
;;; structure for f,d: f is function value, and d derivative, default 0

```

```

(defstruct (df (:constructor df (f &optional (d 0)))) f d )
(defmethod print-object ((a df) stream)(format stream "<~a, ~a>" (df-f a)(df-d a)))

(defvar *difprog* nil);; the text of the program
(defvar *bindings* nil) ;; bindings used for program

(defun dcomp (f x p)
  ;; the main program
  (let ((*difprog* nil)
        (*bindings* nil))
    (multiple-value-bind
      (val dif)
      (dcomp1 f x p)
      (emit '(values ,val ,dif))
      '(let ,*bindings*
          ,(format nil "~s wrt ~s" f x)
          ,@(nreverse *difprog*))))))

;; Assist in compilation of a function f and its derivative at a point x=p
;; dcomp1 changes *bindings* and *difprog* as it munches on the expression f.

(defun dcomp1 (f x p)
  (declare (special *v*))
  (cond((atom f)
        (cond ((eq f x) (values p 1.0d0))
              ((numberp f)(values f 0.0d0))
              (t (let ((r (make-dif-name f x))
                      (push r *bindings*)
                      (emit '(setf ,r 0.0d0)); unnecessary? already bound to 0.0d0
                      (push (cons f r) *lvkd*)
                      (values f r))))))
        (t (let* ((op (first f))
                  (program (get op 'dopcomp)));otherwise, look up the operator's d/dx
              (if program
                  (funcall program (cdr f) x p)
                  (error "~% dcomp cannot do ~s" f);; for now
                  )))))

(defun gentempb(r)
  (push (gentemp r) *bindings*) (car *bindings*))

(defun emit(item)(unless (and (eq (car item) 'setf)
                              (eq (cadr item)(caddr item)))
                (push item *difprog*)))

;; simple unary f(x) programs.
;;We define them all with the same template.

(defmacro dr(op s)

```

```

(setf(get op 'dopcomp) ;;define a rule for returning v, d for dcomp
      (dr2 op (treefloat s))))

(defun dr2 (op s) ;; a rule to make rules for dcomp
  '(lambda(l x p)
    (cond ((cdr l)
           (error "~&too many arguments to ~s: ~s" ',op l))
          (t (let ((v (gentempb 'f))
                   (d (gentempb 't)))
                (multiple-value-bind
                  (theval thedif)
                  (dcomp1 (car l) x p)
                  (emit (list 'setf v (list ',op theval)))
                  (emit (list 'setf d (*chk thedif (subst theval 'x ',s))))))
              (values v d))))))

(defun treefloat(x) ;; convert all numbers to double-floats
  (cond ((null x) nil)
        ((numberp x)(coerce x 'double-float))
        ((atom x) x)
        (t (mapcar #'treefloat x))))

;; Here's how the rule definition program works.
;; Set up rules.
(dr tan (power (cos x) -2))
(dr sin (cos x))
(dr cos (* -1 (sin x)))
(dr asin (power (+ 1 (* -1 (power x 2))) -1/2))
(dr asin (power (+ 1 (* -1 (power x 2))) -1/2))
(dr acos (* -1 (power (+ 1 (* -1 (power x 2))) -1/2)))
(dr atan (power (+ 1 (power x 2)) -1))
(dr sinh (cosh x))
(dr cosh (sinh x))
(dr log (power x -1))
(dr exp (exp x))
(dr sqrt (* 1/2 (power x -1/2)))
;; etc etc

;; Next, functions of several arguments depending on x
;; These include + - * / expt, power

(setf (get '* 'dopcomp) '*rule)
(setf (get '+ 'dopcomp) '+rule)
(setf (get '/ 'dopcomp) '/rule)
(setf (get '- 'dopcomp) '-rule)
(setf (get 'power 'dopcomp) 'power-rule)
(setf (get 'expt 'dopcomp) 'expt-rule)

```

```

(defun +rule (l x p)
  (let ((valname (gentempb 'f))
        (difname (gentempb t)))
    (multiple-value-bind (v d) (dcomp1 (car l) x p)
      (emit '(setf ,difname ,d ,valname ,v)))
    (dolist (i (cdr l))(multiple-value-bind (v d) (dcomp1 i x p)
      (emit '(setf ,difname ,(+chk d difname)))
      (emit '(setf ,valname ,(+chk v valname))))))
    (values valname difname)))

(defun -rule (l x p)
  (let ((valname (gentempb 'f))
        (difname (gentempb t)))
    (cond ((cdr l)
      (multiple-value-bind (v d) (dcomp1 (car l) x p)
        (emit '(setf ,difname ,d ,valname ,v)))
      (dolist (i (cdr l))(multiple-value-bind (v d) (dcomp1 i x p)
        (emit '(setf ,difname (- ,difname ,d)))
        (emit '(setf ,valname (- ,valname ,v ))))) )
      ;; just one arg
      (t (multiple-value-bind (v d) (dcomp1 (car l) x p)
        (emit '(setf ,difname (- ,d) ,valname (- ,v))))))
    (values valname difname)))

;; (- x y z) means (+ x (* -1 y) (* -1 z)). (- x) means (* -1 x)

(defun *rule (l x p)
  (let ((valname (gentempb 'f))
        (difname (gentempb t)))
    (multiple-value-bind (v d) (dcomp1 (car l) x p)
      (emit '(setf ,difname ,d ,valname ,v)))
    (dolist (i (cdr l))(multiple-value-bind (v d) (dcomp1 i x p)
      (emit '(setf ,difname ,(+chk (*chk v difname)
                                     *chk d valname))))
      (emit '(setf
              ,valname ,(+chk v valname))))))
    (values valname difname)))

(defun /rule (l x p)
  (let ((vname (gentempb 'f))
        (dname (gentempb t)))
    (multiple-value-bind (v d) (dcomp1 (car l) x p)
      (emit '(setf ,dname ,d ,vname ,v)))
    (multiple-value-bind (v d) (dcomp1 (cadr l) x p)
      (emit '(setf ,dname (/ (- (* ,v ,dname)(* ,vname ,d))
                              (* ,v ,v))
                          ,vname (/ ,vname ,v))))))
    (values vname dname)))

```

```

(defun power-rule (l x p)
  (cond ((caddr l) (error "~&too many arguments to power: ~s" l))
        ;; now we assume that everything is a-ok
        ;; i.e. power has only two arguments, the second argument
        ;; which is the power to be raised to is independent of x
        ;; so we can use for sqrt, cube-root etc.
        (t (let ((vname (gentempb 'f))
                 (dname (gentempb t)))
              (multiple-value-bind (v d) (dcomp1 (car l) x p)
                ;; We could use +chk and *chk in the following emissions
                ;; but they are really inessential.
                (emit '(setf ,vname (power ,v ,(cadr l))))
                (emit '(setf ,dname (* ,d (power ,v (1- ,(cadr l))) ,(cadr l))))
                (values vname dname))))))

```

```

(defun expt-rule (l x p)
  "this is the general power rule where the exponent could be an
  arbitrary function of x"
  (cond ((caddr l) (error "~&too many arguments to expt : ~s" l))
        ((numberp (cadr l))(power-rule l x p))
        ;; we had  $z = (\text{expt } f \ g)$ 
        ;;  $z' = z*(g*\log f)' = z*(g*f'/f + g'*\log f)$ 
        (t (let ((vname (gentempb 'f))
                 (dname (gentempb t)))
              (multiple-value-bind (v d) (dcomp1 (cadr l) x p)
                (emit '(setf ,vname ,v ,dname ,d)))
              (multiple-value-bind (v d) (dcomp1 (car l) x p)
                (emit '(setf ,dname ,(+chk '(/ ,(*chk vname d) ,v)
                                           (*chk dname '(log ,v))))
                (emit '(setf ,vname (expt ,v ,vname)))
                (emit '(setf ,dname ,(*chk vname dname))))
                (values vname dname))))))

```

```

;; the next two programs, "optimize":
;; generated code so as to not add 0 or mult by 0 or 1

```

```

(defun +chk (a b)(cond ((and (numberp a)(= a 0)) b)
                      ((and (numberp b)(= b 0)) a)
                      (t '(+ ,a ,b))))

```

```

(defun *chk (a b)(cond ((and (numberp a)(= a 1)) b)
                      ((and (numberp b)(= b 1)) a)
                      ((and (numberp a)(= a 0)) a)
                      ((and (numberp b)(= b 0)) b)
                      (t '(* ,a ,b))))

```

```

;;; this is the main program for compiling

```

```

(defun dc (f x &optional (otherargs nil))

```

```

;; produce a program, p(v) ready to go into the compiler to
;; compute f(v), f'(v), returning result as a structure, a df
(let ((*difprog* nil)
      (*bindings* nil)
      (*v* (gentemp "g")))
  (declare (special *v*))
  (multiple-value-bind
    (val dif)
    (dcomp1 f x *v*)
    (emit '(df ,val ,dif))
    '(lambda (,*v* ,@otherargs)
        ,(format nil "~s wrt ~s" f x)
        ;; comment out these declares if you prefer v's type to be unknown
        (declare (double-float ,*v*))
        (declare (optimize (speed 3)(debug 0)(safety 0)))
        ; (assert (typep ,*v* 'double-float))
        (let ,(mapcar #'(lambda (r)(list r 0d0)) *bindings*)
          (declare (double-float ,@*bindings*))
          ,@(nreverse *difprog*))))))

;; A plausible way to use dc is as follows:

(defmacro defdiff (name arglist body) ;; put the pieces together
  (progn
    (setf (get name 'defdiff) name)
    (let ((r (dc body (car arglist) (cdr arglist)))) ;; returns a df.
      '(defun ,name , (cadr r) ,@(caddr r)))))

;; usage (defdiff f (x) (* x (sin x)))
;; Then you can call (f 3.0d0)

;; a few more pieces to allow inside defdiff: if, progn, setf.
;; we could try for a few more. Oh, > < = etc. work "automagically."

(defun if-rule (l x p)
  (let ((valname (gentempb 'f))
        (difname (gentempb t)))
    (emit '(multiple-value-setq
            (,valname ,difname)
            ;; assume only variables, not derivatives in condition
            (if ,(subst p x (car l))
                (funcall (function ,(dc (cadr l) x)) ,p)
                (funcall (function ,(dc (caddr l) x)) ,p))))
          (values valname difname)))

(defun progn-rule(l x p)
  (mapc #'(lambda (k)(dcomp1 k x p)) (butlast l))
  (dcomp1 (car (last l)) x p))

```

```

(defun setf-rule(l x p)
  (if (not(symbolp (car l)))(error "dcomp cannot do setf ~s" (car l)))
  (multiple-value-bind (v d)
    (dcomp1 (cadr l) x p)
    (emit '(setf ,(car l) ,v))
    (let ((dname (make-dif-name (car l) x)))
      (push dname *bindings*)
      (emit '(setf ,dname ,d)))
    (values v d)))

(defun make-dif-name(s x) ;s is a symbol. make a new one like s_dif_x
  (intern (concatenate 'string (symbol-name s) "_DIF_" (symbol-name x))))

(setf (get 'if 'dopcomp)      'if-rule)
(setf (get 'progn 'dopcomp)   'progn-rule)
(setf (get 'setf 'dopcomp)    'setf-rule)

```

References

- [1] "A Short Note on Short Differentiation Programs in Lisp, and a Comment on Logarithmic Differentiation," *SIGSAM Bulletin Volume 32, Number 3*, Sept., 1998, pp. 2-7. www.cs.berkeley.edu/~fateman/papers/deriv.pdf.
- [2] R. Fateman, "Backward Automatic Differentiation in Lisp," (in progress)
- [3] A. Griewank, "On Automatic Differentiation," in M. Iri & K. Tanabe (Eds.) *MATHEMATICAL PROGRAMMING*, Kluwer Academic Publishers, 1989, pp. 83-107.
- [4] A. Griewank and G. Corliss (eds), *Automatic Differentiation of Algorithms*, SIAM, 1991. esp. paper by L. Rall