

Symbolic data

```
(quote a) --> a
(quote this-is-a-symbol) -->
  this-is-a-symbol
'a --> a
'(a b d) --> (a b d)
'(1 2 (a (3 d) this is the life)
-->
(1 2 (a (3 d) this is the life)
```

Some cautions

```
'c --> (quote c)
'(1 'a 2) --> (1 (quote a) 2)
```

The eq? predicate

(eq? *x y*) returns #t iff *x* and *y* are both the same (small) integer, the same symbol or the same pointer to a data structure.

It always returns #f if *x* and *y* are real numbers.

Some examples

```
(eq? 'a 'a) --> #t
(eq? 'a 'b) --> #f
(eq? 1234 1234) --> #t
(eq? 1234567890 1234567890) --> #f
(define x (list 1 2))
(eq? x x) --> #t
(eq? (list 1 2) (list 1 2)) --> #f
```

An efficient membership test

```
(define (memq item list)
  (cond ((null? list) #f)
        ((eq? item (car list))
         list)
        (else (memq item
                     (cdr list)))))

(memq 'a '(b a d a b i n g)) -->
(a d a b i n g)
```

A way to count items in a list

```
(define (item-count item list)
  (define (item-count-iter sublist count)
    (let ((next-sublist (memq item sublist)))
      (if (not next-sublist)
          count
          (item-count-iter (cdr next-sublist)
                          (+ count 1)))))

(item-count-iter list 0)
(item-count 'a '(1 a (a a) b 2 a)) --> 2
```

Symbolic differentiation

- Polynomial $a + bx + cx^2$ can be represented by the list $(+ a (+ (* b x) (* c (* x x))))$
- Derivative with respect to x is $b + 2cx$, which can be represented by $(+ b (* 2 (* c x)))$
- How can the derivative be computed?

Basic calculus

$dy/dx = 0$ if y is a constant or a variable other than x

$dx/dx = 1$

$d(u+v)/dx = du/dx + dv/dx$

$d(u*v)/dx = u * dv/dx + v * du/dx$

Data abstraction to the rescue!

Some constructors, selectors and predicates:

```
(variable? x) (same-variable? x y)
(sum? x)      (product? x)
(make-sum x y) (make-product x y)
(sum-arg1 x)  (product-arg1 x)
(sum-arg2 x)  (product-arg2 x)
```

Now we can compute derivatives

```
(define deriv expr var)
  (cond ((number? expr) 0)
        ((variable? expr)
         (if (same-variable? expr var) 1 0))
        ((sum? expr)
         (make-sum
          (deriv (sum-arg1 expr) var)
          (deriv (sum-arg2 expr) var)))
```

(deriv continued)

```
((product? expr)
 (make-sum
  (make-product
   (product-arg1 expr)
   (deriv (product-arg2 expr) var))
  (make-product
   (product-arg2 expr)
   (deriv (product-arg1 expr) var))))
(else (error "Unknown type" expr)))
```

Implementation of lower layer

```
(define (variable? x) (symbol? x))

(define (same-variable? x y)
  (and (variable? x)
        (variable? y)
        (eq? x y)))
```

Sums

```
(define (make-sum x y)
  (list '+ x y))
(define (sum? x)
  (and (pair? x)
        (eq? (car x) '+)))
(define (sum-arg1 x) (cadr x))
(define (sum-arg2 x) (caddr x))
```

Products

```
(define (make-product x y)
  (list '* x y))
(define (product? x)
  (and (pair? x)
        (eq? (car x) '*)))
(define (product-arg1 x) (cadr x))
(define (product-arg2 x)
  (caddr x))
```

It works! (sort of)

```
(define expr (make-product 'x 'y))
expr --> (* x y)
(deriv expr 'x) -->
(+ (* x 0) (* y 1))
Should be y
```

Need to make simple reductions

A better make-sum

```
(define (make-sum x y)
  (cond ((and (number? x) (= x 0)) y)
        ((and (number? y) (= y 0)) x)
        ((and (number? x) (number? y))
         (+ x y))
        (else (list '+ x y))))
```

A better make-product

```
(define (make-product x y)
  (cond ((or (and (number? x) (= x 0))
            (and (number? y) (= y 0)))
        0)
        ((and (number? x) (= x 1)) y)
        ((and (number? y) (= y 1)) x)
        ((and (number? x) (number? y))
         (* x y))
        (else (list '* x y))))
```

Still room for improvement

```
(define expr (make-product 'x 'y))
expr --> (* x y)
(deriv expr 'x) --> y
but
(define expr (make-product 'x 'x))
expr --> (* x x)
(deriv expr 'x) --> (+ x x)
(* 2 x) would be better
```

Always room for improvement

More sophisticated simplifications are done in
Reduce, Macsyma, Mathematica

Theoretically, no matter how many
simplifications we build into the software,
there are always more simplifications that
can be made