Exercise 2.73.

Section 2.3.2 described a program that performs symbolic differentiation:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
          (make-product (multiplier exp)
                        (deriv (multiplicand exp) var))
          (make-product (deriv (multiplier exp) var)
                        (multiplicand exp))))
        <more rules can be added here>
        (else (error "unknown expression type -- DERIV" exp))))
```

We can regard this program as performing a dispatch on the type of the expression to be differentiated. In this situation the "type tag" of the datum is the algebraic operator symbol (such as `+`) and the operation being performed is `deriv`. We can transform this program into data-directed style by rewriting the basic derivative procedure as

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-varible? exp var) 1 0))
        (else ((get 'deriv (operator exp)) (operands exp)
                                           var))))

(define (operator exp) (car exp))

(define (operand exp) (cdr exp))
```

a. Explain what was done above. Why can't we assimilate the predicates `number?` and `same-variable?` into the data-directed dispatch?

b. Write the procedures for derivatives of sums and products, and the auxiliary code required to install them in the table used by the program above.

c. Choose any additional differentiation rule that you like, such as the one for exponents (exercise 2.56), and install it in this data-directed system.

d. In this simple algebraic manipulator the type of an expression is the algebraic operator that binds it together. Suppose, however, we indexed the procedures in the opposite way, so that the dispatch line `inderiv` looked like

```
((get (operator exp) 'deriv) (operands exp) var)
```

What corresponding changes to the derivative system are required?

Answer.

a. The first two clauses in this conditional play the same roles as they did formerly. All the variation appears in the third case, that is, the `else` clause. Observe that in the `else` clause the symbolic differentiation was performed by means of a general operation `deriv`, which performs a generic derivation to a particular expression respect to a designated variable. The derivatives are extracted by looking in figure 1 under the name of the operation (which is `deriv` here) and the type of the algebraic operator symbol (such as `+`).

Email address: informlarry@gmail.com

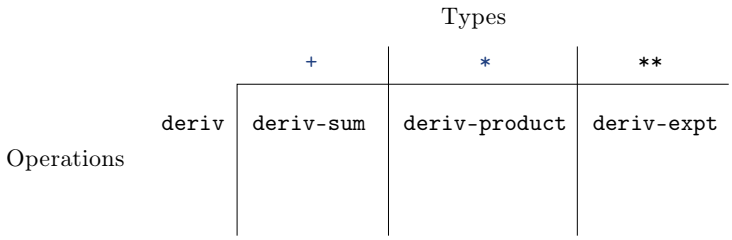

Figure 1. Table of operations for the symbolic differentiation system

Remeber that all the programs we designed in the data-directed style requires data to be represented by list structure, which serve as a foundation in performing the strategy of dispatching on type. We learnt from section 1.1.2 that if a variable or an expression is defined to be a number, evaluating it we will get the numerical value it is binded with. In the case where the expression is simply a quotation (that is, literally a string of characters) the rule of quotation indicates that the expression possesses a value which is the quoted symbols themselves. In neither cases, the expression is represented by list structure and thus can not be dispatched on type. Therefore, we can not assimilate the predicates `number?` and `same-variable?` into the data-directed dispatch.

b. Much similar to those of the complex number, the packages for derivatives of sums and products can be implemented as:[1]

```
(define (install-sum-package)
  ;; internal procedures
  (define (deriv-sum exp var)
    (make-sum (deriv (addend exp) var)
              (deriv (augend exp) var)))
  (define (addend s) (car s))
  (define (augend s) (cadr s))
  (define (make-sum a1 a2)
    (cond ((=number? a1 0) a2)
          ((=number? a2 0) a1)
          ((and (number? a1) (number? a2)) (+ a1 a2))
          (else (list '+ a1 a2))))

  ;; interface to the rest of the system
  (put 'deriv '+ deriv-sum)
  (put 'make '+
       (lambda (a1 a2) (make-sum a1 a2)))
  'done)

(define (make-sum a1 a2)
  ((get 'make '+) a1 a2))



(define (install-product-package)
  ;; internal procedures
  (define (deriv-product exp var)
    (make-sum
     (make-product (multiplier exp)
                   (deriv (multiplicand exp) var))
     (make-product (deriv (multiplier exp) var)
```

1. Note that the algebraic operator symbol (such as `+`) here serve as the "type tag" of the datum. Therefore, when a generic procedure operates on an algebraic expression of `'+` type, it strips off the tag and passes the contents which is the operands on to the internal code. So what the constructors and selectors manipulate inside the sum package is merely the operands of the algebraic expression rather than the whole of the expression. That is why the implementation of **`make-sum`**, **`addend`** and **`augend`** here vary from those we saw before. The same reason in the case of product and any other newly add representations.

```
                  (multiplicand exp))))
  (define (multiplier p) (car p))
  (define (multiplicand p) (cadr p))
  (define (make-product m1 m2)
    (cond ((or (=number? m1 0) (=number? m2 0)) 0)
          ((=number? m1 1) m2)
          ((=number? m2 1) m1)
          ((and (number? m1) (number? m2)) (* m1 m2))
          (else (list '* m1 m2))))

  ;; interface to the rest part of the system
  (put 'deriv '* deriv-product)
  (put 'make '*
       (lambda (m1 m2) (make-product m1 m2)))
  'done)

(define (make-product m1 m2)
  ((get 'make '*) m1 m2))
```

c. The following procedure installs the derivatives of exponent in this data-directed system:

```
(define (install-exponentiation-package)
  ;; internal procedures
  (define (deriv-exponentiation exp var)
    (let ((u (base exp))
          (n (exponent exp)))
      (make-product (make-product n
                                  (make-exponentiation u (- n 1)))
                    (deriv u var))))
  (define (base e) (car e))
  (define (exponent e) (cadr e))
  (define (make-exponentiation u n)
    (cond ((=number? n 0) 1)
          ((=number? n 1) u)
          ((and (number? u) (number? n)) (expt u n))
          (else (list '** u n))))

  ;; interface to the rest of the system
  (put 'deriv '** deriv-exponentiation)
  (put 'make '**
       (lambda (u n) (make-exponentiation u n))))
  'done)

(define (make-exponentiation u n)
  ((get 'make '**) u n))
```

d. The changes required by the derivative system is to swap the order of the first two arguments of all the expressions which carry out the `put` operation.