Exercise 2.73.

Section 2.3.2 described a program that performs symbolic differentiation:

```scheme
(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

```scheme
(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 in`deriv` looked like

```scheme
((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.

We see in the `else` clause that the symbolic differentiation was performed by means of a general "operation" procedure named `deriv`, which performs a generic derivation operation to the operands in a particular expression respect to a designated variable. The derivation operation is extracted by looking in table 1 under the name of the operation (which is `deriv` here) and the type of the algebraic operator symbol (such as `+`).

		Types		
		+	*	**
Operations	deriv	deriv-sum	deriv-product	deriv-expt

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 variable (that is, literally a string of characters) the rule of quotation indicates that the expression possesses a value which is the symbols themselves. In neither cases, the expression is represented by list structure. 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 operands var)
    (make-sum (deriv (addend operands) var)
              (deriv (augend operands) 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
  (define (tag x) (attach-tag '+ x))
  (put 'deriv '+ deriv)
  (put 'addend '+ addend)
  (put 'augend '+ augend)
  (put 'make-sum '+
       (lambda (a1 a2) (tag (make-sum a1 a2))))
  'done)

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

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.

```scheme
  (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)
          (else (list m1 m2))))

  ;; interface to the rest part of the system
  (define (tag x) (attach-tag '* x))
  (put 'deriv '* deriv)
  (put 'multiplier '* multiplier)
  (put 'multiplicand '* multiplicand)
  (put 'make-product '*
       (lambda (m1 m2) (tag (make-product m1 m2))))
  'done)
```

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

```scheme
(define (install-exponent-package)
  ;; internal procedures
  (define (deriv operands var)
    (let ((u (base operands))
          (n (exponent operands)))
      (make-product (make-product n
                                  (make-exponent u (- n 1)))
                    (deriv u var))))
  (define (base e) (car e))
  (define (exponent e) (cadr e))
  (define (make-exponent 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
  (define (tag x) (attach-tag '** x))
  (put 'deriv '** deriv)
  (put 'base '** base)
  (put 'exponent '** exponent)
  (put 'make-exponent '**
       (tag (lambda (u n) (make-exponent u n))))
  'done)
```

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