```clojure
(defn third [list]
  (second (next list)))

;; True if the form is a variable (symbol).
(defn variable? [form]
  (symbol? form))

;; True if the two forms are the same variable.
(defn same-variable? [form1 form2]
  (and (variable? form1) (variable? form2) (= form1 form2)))

;; True if the form represents a sum.
(defn sum? [form]
  (and (list? form) (= '+ (first form))))

;; Constructs a sum of a and b.
(defn make-sum [a b]
  (list '+ a b))
  ;(cond
    ;(= a 0) b
    ;(= b 0) a
    ;:else (+ a b)))

;; Selects the addend (first value) of a sum.
(defn addend [sum]
  (second sum))

;; Selects the augend (second value) of a sum.
(defn augend [sum]
  (third sum))

;; True if the form represents a difference.
(defn diff? [form]
  (and (list? form) (= '- (first form))))

;; Constructs a difference of a and b.
(defn make-diff [a b]
  (list '- a b))
  ;(cond
    ;(= a 0) (list - b)
    ;(= b 0) a
    ;:else (- a b)))

;; Selects the minuend (first value) of a difference.
(defn minuend [diff]
  (second diff))

;; Selects the minuend (second value) of a difference.
(defn subtrahend [diff]
  (third diff))

;; True if the form represents a product.
(defn prod? [form]
  (and (list? form) (= '* (first form))))

;; Constructs a product of a and b.
(defn make-prod [a b]
  (list '* a b))
  ;(cond
    ;(= a 0) 0
```

```
61      ;(= a 1) b
62      ;(= b 0) 0
63      ;(= b 1) a
64      ;:else (* a b)))
65
66  ;; Selects the multiplier (first value) of a product.
67  (defn multiplier [prod]
68    (second prod))
69
70  ;; Selects the multiplicand (second value) of a product.
71  (defn multiplicand [prod]
72    (third prod))
73
74  ;; True if the form represents a quotient.
75  (defn quot? [form]
76    (and (list? form) (= '/ (first form))))
77
78  ;; Constructs a quotient of a and b.
79  (defn make-quot [a b]
80    (list '/ a b))
81    ;(cond
82      ;(= a 0) 0
83      ;(= b 1) a
84      ;:else (/ a b)))
85
86  ;; Selets the dividend (top value) of a quotient.
87  (defn dividend [quotient]
88    (second quotient))
89
90  ;; Selets the divisor (bottom value) of a quotient.
91  (defn divisor [quotient]
92    (third quotient))
93
94  ;; True if the form represents a power.
95  (defn power? [form]
96    (and (list? form) (= '** (first form))))
97
98  ;; Constructs a power of a and b.
99  (defn make-power [a b]
100   (list '** a b))
101   ;(cond
102     ;(= b 0) 1
103     ;(= b 1) a
104     ;:else (Math/pow a b)))
105
106 ;; Selets the base value of the power.
107 (defn base-power [power]
108   (second power))
109
110 ;; Selets the exponent of the power.
111 (defn exponent-power [power]
112   (third power))
113
114 ;; True if the form represents a power.
115 (defn ln? [form]
116   (and (list? form) (= 'ln (first form))))
117
118 ;; Constructs a power of a and b.
119 (defn log-of [form]
120   (second form))
```

```clojure

;; Returns the derivative of a function expressed in Clojure notation, where variables are quoted.
;; The second parameter is the variable which the derivative is calculated with respect to.
(defn derivative [form var]
  (cond ; The derivative of a constant is 0
        (number? form) 0
        ; The derivative of a variable is 0 if the variable is not the one being derived; or 1, if it is.
        (variable? form) (if (same-variable? form var) 1 0)
        ; Sum rule
        (sum? form) (make-sum (derivative (addend form) var)
                              (derivative (augend form) var))
        ; Difference rule
        (diff? form) (make-diff (derivative (minuend form) var)
                                (derivative (subtrahend form) var))
        ; Quotient rule
        (quot? form)  (make-quot  (make-diff  (make-prod (divisor form)
                                                         (derivative (dividend form) var))
                                              (make-prod  (derivative (divisor form) var)
                                                          (dividend form)))
                                  (make-prod  (divisor form)
                                              (divisor form)))
        ; Power rule
        (power? form) (make-prod (make-prod (exponent-power form)
                                            (make-power (base-power form) (- (exponent-power form) 1)))
                                 (derivative (base-power form) var))
        ; Natural Log rule
        (ln? form)  (make-quot (derivative (log-of form) var)
                               (log-of form))
        ; Product rule
        (prod? form) (make-sum (make-prod (multiplier form)
                                          (derivative (multiplicand form) var))
                               (make-prod (derivative (multiplier form) var)
                                          (multiplicand form)))))
```