1 Mathematical procedures

1.1 Comparisons

Purpose: Computes the product of any finite collection of numbers. If there are no numbers, the “product” returned is 1
Use: (* 12 6)
=> 72
(* 12 6 2)
=> 144

Purpose: Computes the sum of any finite collection of numbers. If there are no numbers, the “sum” returned is 0
Use: (+ 125 48)
=> 173
(+ 125 48 25 88)
286

Purpose: to subtract the second (and following) number(s) from the first; negate the number if there is only one argument; cannot be applied without any arguments, i.e., (-) generates an error.
Use: (- 7 6)
=> 1
(- 5 3 2)
=> 0

/: num num num ... -> num
Purpose: to divide the first by the product of the second (and all following) number(s); only the
first number can be zero. If there is only one number, returns that number; if there are no numbers,
produces an error.
Use: (/ 144 12)
=> 12
(/ 144 12 3 2)
=> 2

expt : num num -> num
Purpose: Computes $a^b$, where $a$ is the first number and $b$ is the second.
Use: (expt 2 8)
=> 256

max : real real ... -> real
Purpose: Computes the maximum of a finite nonempty collection of numbers.
Use: (max 12 4 0 6 0 2)
=> 12

min : real real ... -> real
Purpose: Computes the minimum of a finite nonempty collection of numbers.
Use: (min 1 0 1 1 0 5 8)
=> 0

modulo : int int -> int
Purpose: Computes the remainder, on division by $|k|$, of the number $n$, where $n$ is the first argument
and $k$ is the second. The result is always a number between 0 and $k - 1$. Both $n$ and $k$ must be
integers, and $k$ must be nonzero.
Use: (modulo 12 5)
=> 2

quotient : int int -> int
Purpose: Computes the (integer) quotient, on division by $k$, of the number $n$, where $n$ is the first
argument and $k$ is the second. The result has the same sign as $n/k$, and is the result of truncating
$n/k$ to an integer. Note: truncating $-1.6$ produces $-1$, not $-2$ as you might expect. Truncation is
not the same as “rounding down.”
Use: (quotient 144 11)
=> 13
remainder: int int -> int
Purpose: Computes the remainder \( r \) on division of the first number, \( n \), by the second number, \( k \). Both must be integers, and \( k \) must be nonzero. The sign of the remainder is the same as the sign of \( n \), and the remainder. See [the online documentation](#) for a full description.
Use: `(remainder 144 11)`
=> 1

sqr : num -> num
Purpose: Computes the square of the argument.
Use: `(sqr 6)`
=> 36

sqrt : num -> num
Purpose: Computes the square root of the argument
Use: `(sqrt 9)`
=> 3

1.1 Comparisons

Almost all Racket number comparisons can take multiple arguments. We’ll use only their two-argument forms; see online [documentation](#) for details on the other forms.

<: real real -> boolean
Purpose: true if the first num is less than the second, false otherwise.
Use: `(<= 25 24)`
=> true
Purpose: true if the first number is greater than the second, false otherwise.

(> 25 5)
=> true
(> 7 7)
=> false

>= : real real -> boolean
Purpose: true if the first number is greater or equal to the second, false otherwise.

(>= 24 25)
=> false
(>= 24 24)
=> true

2 Boolean procedures and related constructs

not : boolean -> boolean
Purpose: Negates a boolean. (not true) is false and (not false) is true

2.1 Other boolean things

While not is a built-in procedure, and, or, and if are keywords that mark the beginnings of three different kinds of scheme expressions, each with its own evaluation rule.

(and expr1 expr2 ... exprn)
Purpose: Evaluates each expression in turn; until either (a) some expression evaluates to false or (b) the end of the list is reached. In case “a”, returns false; in case “b”, returns the value of exprn.

In practice, however, and always in CS17, all the expressions should evaluate to booleans, in which case the description of the behavior is simpler: evaluates each expression in turn, until some value is false or the end of the list is reached. In the first case, return false (and do not evaluate the remaining expressions!); in the second, return true. In particular, (and) evaluates to true.

(or expr1 expr2 ... exprn)
Purpose: (We give the CS17 version only) Evaluates each expression in turn; until either (a) some expression evaluates to true or (b) the end of the list is reached. In case “a”, returns true (and does not evaluate the remaining expressions!); in case “b”, returns false.

(if expr expr1 expr2)
Purpose: (CS17 version only). Evaluate expr; the result must be a boolean. If the boolean is true, evaluate expr1 and return the result, without evaluating expr2. If the boolean is false, evaluate expr2 and return the result, without evaluating expr1.
2.2 Predicates

We begin with a few helpful mathematical predicates.

2.3 Number Predicates

odd?: integer -> boolean
Purpose: to determine if some value is odd or not
(odd? 1)
=> true

even?: integer -> boolean
Purpose: to determine if some value is even or not
(even? 1)
=> false

zero?: number -> boolean
Purpose: to determine if some value is zero or not
(zero? 0)
=> true

positive?: number -> boolean
Purpose: to determine if some number is positive
(positive? 12)
=> true

negative?: number -> boolean
Purpose: to determine if some value is negative
(negative? -77)
=> true

2.4 Type Predicates

Each of the following predicates tells whether its single argument is of a given type (e.g., a number, a boolean, a string, etc.).

number?: any -> boolean
Purpose: to determine whether some value is a number
(number? "cs17")
=> false
(number? 17)
=> true

boolean?: any -> boolean
### Purpose: to determine whether some value is a boolean

- (boolean? true) => true
- (boolean? 17) => false

### string?: any -> boolean

Purpose: to determine whether some value is a string

- (string? "cs17") => true
- (string? 17) => false

### procedure?: any -> boolean

Purpose: to determine if a value is a procedure

- (procedure? (lambda (x) (* x 12))) => true

### struct?: any -> boolean

Purpose: to determine whether some value is a structure

- (define-struct course (course-code)) (define cs17 (make-course "CSCI0170")) (struct? cs17) => true

### empty?: any -> boolean

Purpose: to determine whether some value is the empty list

- (empty? empty) => true
- (empty? (list 1 2 3)) => false

### cons?: any -> boolean

Purpose: to determine whether some value is a constructed list

- (cons? (list 1 2 3)) => true
- (cons? empty) => false

### list?: any -> boolean

Purpose: to determine whether some value is a list, i.e., either empty or a cons.

- (list? (list 1 2 3)) => true
- (list? empty) => true

## 3 Lists

In CS17, all lists are monotype lists: they’re lists of integers, or lists of booleans, or lists of strings, or lists of cookie-structs, or lists of int-lists. But we never have a list containing two integers, two strings, and a boolean. The documentation that follows tells about such monotype lists, even though the Racket procedures can actually do far more than is documented here.

### empty: The empty list, a predefined value in Racket.

- empty

### cons: X (listof X) -> (listof X)

Purpose: Construct a list from arguments item and a-list, where a-list must be a list of X, and item must be of type X. The resulting list lst has the property that (first lst) is item and (rest lst) is a-list.
first : (listof X)) -> X
Purpose: returns the first item of a non-empty list; produces an error if called on an empty list.

rest : (listof X)) -> (listof X)
Purpose: returns the rest of a non-empty list, aside from its first element; produces an error if called on an empty list.

length : list -> number
Purpose: Computes the number of items in a list

list : any ... -> (listof any)
Purpose: to construct a list of its arguments

append : (listof any)(listof any)(listof any)... -> (listof any)
Purpose: to create a single list from several, by juxtaposition of the items

filter : ('a -> bool)(listof 'a) -> (listof 'b)
Purpose: applies the given predicate procedure to each element of the input list in order; the output list (in order) consists of those elements of the input for which the predicate returns true.

map : ('a -> 'b)(listof 'a) -> (listof 'b)
Purpose: applies the given procedure to each element of the input list in order to create the output list (in order). Also works for a function of n arguments: ('a1 ... 'an -> 'b)(listof 'a1)... (listof 'an)-> (listof 'b) by applying the procedure to the first element of all input lists to produce the first element of the output, applying it to the second element of all input lists to produce the second element of the output, and so on. The input lists must all have the same length.

reverse : (listof 'a) -> (listof 'a)
Purpose: Produce a list of the same length, n, whose ith element (counting from i = 1 as the first element) is the n + 1 – ith element of the input list; in short, produce the input list in reverse order.

4 Structs

define-struct is not a procedure, but its behavior is still worth documenting:

define-struct name (field1 field2 ...) 
Purpose: defines a struct and creates a constructor make-name; selectors, name-field1, name-field2, etc.; and a predicate name? that tells whether some item is in fact a name struct.