03: Simple Data

What does "$<$" mean?

Consider the expression "$x < 5$".

In math class, it tells us something about x: whatever value x has, that value is less than 5.

We might combine the statement "$x < 5$" with the statements "x is even" and "x is a perfect square" to conclude "x is 4".

In Racket, "<" means something different. A constant such as `x` already has a value.

What does "$<$" mean?

Suppose we define a constant:

```
(define x 2)
```

Now we create a Racket expression as close to "$x < 5$" as possible:

```
(< x 5)
```

This is *asking* "Is it true that the value of `x` is less than 5?"

If we evaluate `(< x 5)`, we substitute in the value of the constant, so our expression becomes `(< 2 5)`. Since it is true that $2 < 5$, the statement evaluates to `true`.

On the other hand, if we define the constant:

```
(define y 10)
```

Now `(< y 5)` $\Rightarrow$ `(< 10 5)` $\Rightarrow$ `false` since it is not the case that $10 < 5$.

`true` and `false` are values, just like `0`, `100`, and `22/7` are values. `true` and `false` are **Boolean** values; the others are numeric values.

`<`, `>`, `<=`, `>=`, and `=` are functions, each of which produces a value, abbreviated `Bool` in contracts.

```
(define x 4)
```

`(< x 6)` ⇒ is `x` (4) less than 6?

`(> x 6)` ⇒ is `x` greater than 6?

`(= x 7)` ⇒ is `x` equal to 7?

`(>= 5 x)` ⇒ is 5 greater than or equal to `x`?

`(<= 5 x)` ⇒ is 5 less than or equal to `x`?

Each produces `true` or `false`. These are the only values a `Bool` may take.

A function which produces a `Bool` is called a **predicate**. For many predicates in Racket, the name ends with `?`.

We can also write our own predicates. For example:

```
;; (can-vote? age) produces true if the person is voting age.
(define (can-vote? age)
  (>= age 18))

(check-expect (can-vote? 17) false)
(check-expect (can-vote? 20) true)
```

Ex. 1

Figure out how to use each of the following predicates in DrRacket.

Be sure you understand when each produces `true` and when it produces `false`.

1. `>`
2. `even?`
3. `=`
4. `negative?`

Combining predicates

Our previous version of `can-vote?` is too simplistic. In reality, you need to be at least 18 years old and a citizen.

```
;; (can-vote-v2? age citizen?) produces true if the person is eligible to vote.
(define (can-vote-v2? age citizen?)
  (and (>= age 18) citizen?))

(check-expect (can-vote-v2? 18 true) true)
(check-expect (can-vote-v2? 18 false) false)
(check-expect (can-vote-v2? 16 true) false)
```

We combine predicates using the special forms **`and`** and **`or`**, and the function `not`. These all consume and produce `Bool` values.

Combining predicates

We combine predicates using the special forms `and` and `or`, and the function `not`. These all consume and produce `Bool` values.

- `and` has value `true` when *all* of its arguments have value `true`; `false` otherwise.
- `or` produces `true` if at least one of its arguments is `true`; `false` otherwise.
- `not` produces `true` if its argument is `false`; `false` if its argument is `true`.

Both `or` and `and` require at least two arguments, but may have more.

```
;; (between? x a y) produces true
;;   if a is between x and y
(define (between? x a y)
  (and (<= x a) (<= a y)))


;; (weak-password? p) produces true
;;   if p is definitely a weak password
(define (weak-password? p)
  (or (< (string-length p) 8)
      (string-numeric? p)
      (string-lower-case? p)
      (string-upper-case? p)))

(check-expect
  (weak-password? "fooBar") true)
```

```
;; (go-run? rain? friends? temp)
;;   determines whether one should go
;;   for a run or not.
(define (go-run? rain? friends? temp)
  (and (not rain?)
       (or (between? 13 temp 28)
           friends?)))

(check-expect
  (go-run? false true 33) true)
```

Ex. 2

Write a function that consumes an `Int`, and produces

- `"baz"` for even numbers in the interval $[10, 40]$
- `"qux"` for odd numbers in the interval $[-20, 20]$
- `"xyzzy"` for numbers less than -100 or greater than 200
- `"corge"` otherwise.

Predicates

Predicates defined in DrRacket include (read a row at a time):

`(< 3 4)` ⇒ `true`	`(< 4 3)` ⇒ `false`	similar for `<=`, `>`, `>=`, `=`
`(number? 3)` ⇒ `true`	`(number? 3.14)` ⇒ `true`	`(number? true)` ⇒ `false`
`(integer? 3)` ⇒ `true`	`(integer? 3.14)` ⇒ `false`	`(integer? true)` ⇒ `false`
`(positive? 3)` ⇒ `true`	`(positive? -3)` ⇒ `false`	`(positive? true)` ⇒ `error`
`(negative? 3)` ⇒ `false`	`(negative? -3)` ⇒ `true`	`(negative? true)` ⇒ `error`
`(even? 3)` ⇒ `false`	`(even? 4)` ⇒ `true`	`(even? true)` ⇒ `error`
`(odd? 3)` ⇒ `true`	`(odd? 4)` ⇒ `false`	`(odd? true)` ⇒ `error`
`(zero? 0)` ⇒ `true`	`(zero? 4)` ⇒ `false`	`(zero? true)` ⇒ `error`
`(exact? 3)` ⇒ `true`	`(exact? (/ 22 7))` ⇒ `true`	`(exact? true)` ⇒ `error`
`(exact? pi)` ⇒ `false`		
`(boolean? true)` ⇒ `true`	`(boolean? false)` ⇒ `true`	`(boolean? 3)` ⇒ `false`
`(false? false)` ⇒ `true`	`(false? true)` ⇒ `false`	`(false? 3)` ⇒ `false`

Racket only evaluates as many arguments of `and` and `or` as is necessary to determine the value. Examples:

```
;; Eliminate easy cases first; might not need to do
```

Rules 4-6: Substitution rules for **and**

Use the following substitution rules for tracing **and**:

```
(and false ...) => false

(and true ...) => (and ...)

(and) => true
```

Ex. 3

Perform a trace of

```
(and (= 3 3) (> 7 4) (< 7 4) (> 0 (/ 3 0)))
```

Check your work with the stepper in the commentary.

Ex. 4

Perform a trace of:

```
(define s "bravo")
(and (> 7 4) true (string=? s "bravo"))
```

Rules 7-9: Substitution rules for **or**

Use these substitution rules for tracing **or**:

```
(or true ...) ⇒ true

(or false ...) ⇒ (or ...)

(or) ⇒ false
```

Ex. 5

Perform a trace of

```
(or (< 7 4) (= 3 3) (> 7 4) (> 0 (/ 3 0)))
```

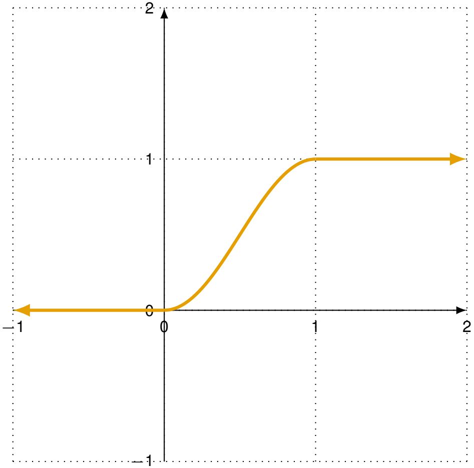
Check your work with the stepper in the commentary.

Ex. 6

Perform a trace of

```
(define s "bravo")
(or (< 7 4) false (string=? s "hooray"))
```

Sometimes expressions should take one value under some conditions, and other values under other conditions.



A sin-squared window, used in signal processing, can be described by the following piecewise function:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 1 \\ \sin^2(x\pi/2) & \text{for } 0 \leq x < 1 \end{cases}$$

We can compute the sin-squared window function $f(x)$ with a **conditional expression**:

```
(cond [(< x 0) 0]
      [(>= x 1) 1]
      [(< x 1) (sqr (sin (* x pi 0.5)))])
```

- Conditional expressions use the special form `cond`.
- Each argument is a question/answer pair.
- The **question** is a Boolean expression.
- The **answer** is a possible value of the conditional expression.
- Square brackets are used by convention, for readability.
- Properly nested square brackets and parentheses are equivalent in the teaching languages.

How do we evaluate a **cond**?

Informally, evaluate a **cond** by considering the question/answer pairs in order, top to bottom. When considering a question/answer pair, evaluate the question. If the question evaluates to true, the *whole* **cond** produces the corresponding answer.

```
(define (ssqw x)
  (cond
    [(< x 0) 0]
    [(>= x 1) 1]
    [(< x 1)
     (sqr (sin (* x pi 0.5)))]))
```

For example, consider (ssqw 4).

```
=> (cond [(< 4 0) 0]
         [(>= 4 1) 1]
         [(< 4 1) (sqr (sin (* 4 pi 0.5)))])
```

No satisfied questions

What happens if *none* of the questions evaluate to `true`?

```
(define (ssqw x)
  (cond
    [(< x 0) 0]
    [(> x 1) 1]
    [(< x 1) (sqr (sin (* x pi 0.5)))]))
```

The second test has changed from `>=` to just `>`.

An error occurs if we try to run `(ssqw 1)`

This can be helpful – if we see this error we know we've missed a case in our code.

But sometimes we want to only describe some conditions, and do something different if none of them are satisfied.

In these situations, the question in the last question/answer pair may be **else**.

```
(define (ssqw x)
  (cond
    [(< x 0) 0]
    [(>= x 1) 1]
    [else (sqr (sin (* x pi 0.5)))]))
```

There are three rules: when the first expression is false, when it is true, and when it is **else**.

```
(cond [false exp] ...) ⇒ (cond ...)

(cond [true exp] ...) ⇒ exp

(cond [else exp]) ⇒ exp
```

These suffice to simplify any **cond** expression.

Here the ellipses are serving a different role. They are not showing a pattern, but showing an **omission**. The first rule just says "whatever else appeared after the `[false exp]`, you just copy it over."

Example: tracing `cond`


```
(define n 5) (define x 6) (define y 7)

(cond [(even? n) x][(odd? n) y])
⇒ (cond [(even? 5) x] [(odd? n) y])
⇒ (cond [false x][(odd? n) y])
⇒ (cond [(odd? n) y])
⇒ (cond [(odd? 5) y])
⇒ (cond [true y])
⇒ y
⇒ 7
```

What happens if `y` is not defined?

```
(define n 5) (define x 6)

(cond [(even? n) x][(odd? n) y])
⇒ (cond [(even? 5) x] [(odd? n) y])
⇒ (cond [false x][(odd? n) y])
⇒ (cond [(odd? n) y])
⇒ (cond [(odd? 5) y])
⇒ (cond [true y])
⇒ y
⇒ y: this variable is not defined
```

DrRacket's rules differ. It scans the whole `cond` expression before it starts, notes that `y` is not defined, and shows an error. That's hard to explain with substitution rules!

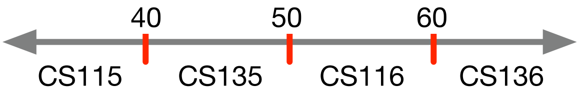
Ex. 7

Step through this program

```
(define (qux a b)
  (cond
    [(= a b) 42]
    [(> a (+ 3 b)) (* a b)]
    [(> a b) (- b a)]
    [else -42]))


(qux 5 4)
```

Verify your answer with the stepper in the commentary.

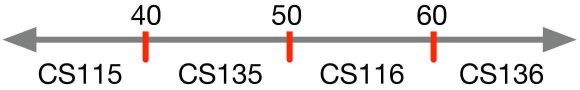



```
;; Constants for each CS course.
(define CS115 1)  (define CS116 2)
(define CS135 3)  (define CS136 4)

;; (course-after-cs135 grade) produces the
;;    recommended course, depending on the
;;    CS135 grade.
(define (course-after-cs135 grade)
  (cond
    [(< grade 40) CS115]
    [(and (>= grade 40) (< grade 50)) CS135]
    [(and (>= grade 50) (< grade 60)) CS116]
    [(>= grade 60) CS136]))
```

```
(check-expect
 (course-after-cs135 35) CS115)
(check-expect
 (course-after-cs135 40) CS135)
(check-expect
 (course-after-cs135 45) CS135)
(check-expect
 (course-after-cs135 50) CS116)
(check-expect
 (course-after-cs135 55) CS116)
(check-expect
 (course-after-cs135 60) CS136)
(check-expect
 (course-after-cs135 70) CS136)
```

Simplifying conditional expressions




```
;; Constants for each CS course.
(define CS115 1)  (define CS116 2)
(define CS135 3)  (define CS136 4)

;; (course-after-cs135 grade) produces the
;;    recommended course, depending on the
;;    CS135 grade.
(define (course-after-cs135 grade)
  (cond
    [(< grade 40) CS115]
    [(< grade 50) CS135]
    [(< grade 60) CS116]
    [else CS136]))
```

```
(check-expect
 (course-after-cs135 35) CS115)
(check-expect
 (course-after-cs135 40) CS135)
(check-expect
 (course-after-cs135 45) CS135)
(check-expect
 (course-after-cs135 50) CS116)
(check-expect
 (course-after-cs135 55) CS116)
(check-expect
 (course-after-cs135 60) CS136)
(check-expect
 (course-after-cs135 70) CS136)
```

Ex. 8

Simplify the following conditional expression:

```
;; (flatten-me x) Say which interval x is in.

;; flatten-me:  Nat -> Nat
(define (flatten-me x)
  (cond [(>= x 75) 4]
        [(and (>= x 50) (< x 75)) 3]
        [(and (>= x 25) (< x 50)) 2]
        [(< x 25) 1]))
```

A museum offers free admission for people who arrive after 5 pm. Otherwise, the cost of admission is based on a person's age: age 10 and under are charged $5 and everyone else is charged $10.

A natural solution to this nests one conditional expression inside another. We use one **cond** to pick off the free admission situation. For the paid situation, we have two conditions that are distinguished in the nested **cond**.

```
;; (admission after5? age) ...
(define (admission after5? age)
  (cond [after5? 0]
        [else
         (cond [(<= age 10) 5]
               [else 10])]))
```

```
(check-expect (admission true 4) 0)
(check-expect (admission true 24) 0)
(check-expect (admission false 4) 5)
(check-expect (admission false 24) 10)
```

Often “flat” conditionals are easier to read than “nested” conditionals.

That is, instead of having a `cond` with another `cond` inside, we can rework them so they are multiple clauses of a single `cond`.

Here is an example:

```
; cond inside cond
(define (admission after5? age)
  (cond [after5? 0]
        [else
         (cond [(<= age 10) 5]
               [else 10])]))
```

↔

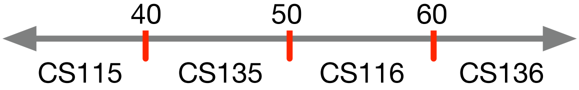
```
; simplified
(define (admission after5? age)
  (cond [after5? 0]
        [(<= age 10) 5]
        [else 10]))
```

`[else (cond ...` is considered amateurish code because it can always be easily flattened.

Testing conditional expressions

- Write at least one test for each possible answer in the conditional expression.
- That test should be simple and direct, aimed at testing that answer.
- When the problem contains **boundary conditions** (like the cut-off between passing and failing marks), they should be tested explicitly.
- DrRacket highlights unused code.

For the course-that-follows-CS135 example




```
(define (course-after-cs135 grade)
  (cond
    [(< grade 40) CS115]
    [(< grade 50) CS135]
    [(< grade 60) CS116]
    [else CS136]))
```

there are four intervals and three boundary points, so seven tests are required (for example, 35, 40, 45 50, 55, 60, 70).

Ex. 9

Write a function that consumes a `Num`, x, and produces

- `1` if $80 < x \leq 100$,
- `-1` if $0 < x \leq 80$,
- `0` otherwise.

Write tests to verify the boundaries are where they should be.

Testing **and** and **or** expressions is similar.
Consider

```
;; Is the line considered "steep"?
(define (steep? dx dy)
  (or (= dx 0)
      (>= (/ dy dx) 1)))
```

We need:

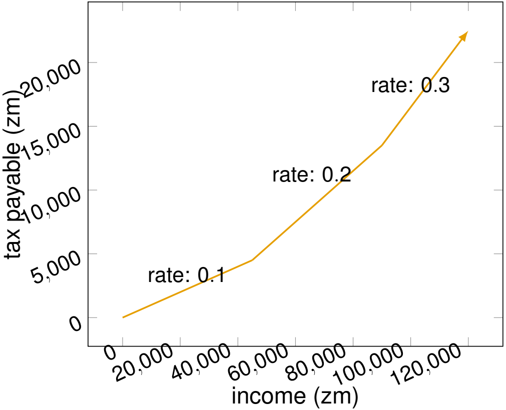
- one test case where *dx* is zero (first argument to **or** is `true`; second is not evaluated)
- one test case where *dx* is nonzero and $dy/dx \geq 1$, (first argument is `false` but second argument is `true`)
- one test case where *dx* is nonzero and $y/x < 1$. (both arguments are `false`)

In the land of Yendor, the currency is the zorkmid, zm.

Taxes are calculated as follows:

- For incomes of 45,000 zm or less, 10% is paid as tax.
- For incomes of 90,000 zm or less, taxes are calculated on the first 45,000 zm (as above) plus 20% of each additional zm.
- For incomes above 90,000 zm, taxes are calculated on the first 90,000 zm (as above) plus 30% of each additional zm.

Write `tax-payable`. It consumes the income and produces the taxes owed.



Ex. 10

Implement the function `tax-payable`. It consumes a number representing income and produces the taxes to be paid on that income.

Include appropriate `check-expect`s to test your function.

Define appropriate constants.

Ex. 11

Examine your code for `tax-payable`. Are there opportunitites to improve it with a helper function? (Unless you used one the first time, the answer is probably "yes!". Look for repeated code.)

Implement `tax-payable` using a helper function.

Racket allows one to define and use **symbols** with meaning to us (not to Racket).

A symbol is defined using a leading apostrophe or ‘quote’: `'CS115`. What follows is the same as any other identifer.

`'CS115` is a value just like `0` or `115`, but it is more limited computationally.

Symbols allow a programmer to avoid using constants to represent names of courses, colours, planets, or types of music.

Unlike numbers, symbols are self-documenting – you don’t need to define constants for them. This is the primary reason we use them.

We can use symbols instead of the constants in our previous example.

```
;; No longer needed!!!
;; (define CS115 1)  (define CS116 2)
;; (define CS135 3)  (define CS136 4)

;; (course-after-cs135 grade) produces
    ...
(define (course-after-cs135 grade)
  (cond
    [(< grade 40) 'CS115]
    [(< grade 50) 'CS135]
    [(< grade 60) 'CS116]
    [else 'CS136]))
```

```
(check-expect
 (course-after-cs135 35) 'CS115)
(check-expect
 (course-after-cs135 40) 'CS135)
(check-expect
 (course-after-cs135 45) 'CS135)
(check-expect
 (course-after-cs135 50) 'CS116)
```

Symbols can be compared using the predicate `symbol=?`.

```
(define home 'Earth)
(symbol=? home 'Mars) ⇒ false
```

`symbol=?` is the only function we'll use in CS135 that is applied *only* to symbols.

Like other types, there is a predicate: `symbol?`.

```
(define mysymbol 'blue)
(symbol=? mysymbol 'blue) ⇒ true
(symbol=? mysymbol 'red) ⇒ false
(symbol=? mysymbol 42) ⇒ error
(symbol? mysymbol) ⇒ true
(symbol? '*@) ⇒ true
(symbol? 42) ⇒ false
```

A **character** is most commonly a printed letter, digit, or punctuation symbol. *a*, *G*, ., +, and 8 are all characters.

Other characters represent less visible things like a tab or a newline in text.

More recent characters include ☺, ✉, and .

For now, we'll be interested in characters only because they are the simplest component of a **string**. We'll discuss the Racket representation of individual characters in a later module.

Strings

Strings are sequences of characters between double quotes. Examples: `"blue"` and `"These are not my shoes. My shoes are brown."`.

What are the differences between strings and symbols?

- Strings are really **compound data** (a string is a sequence of characters).
- Symbols can’t have certain characters in them (such as spaces).
- It is more efficient to compare two symbols than two strings.
- There are more built-in functions for strings than symbols.

Non-numeric types also have predicates. For example, these predicates consume strings and will be useful when we do more work with strings.

We can tell if two strings are the same:

```
(string=? "pearls" "gems") ⇒ false
(string=? "pearls" "pearls") ⇒ true
```

We can also tell if a pair of strings are in alphabetic order. If one string comes before another, it is “less than” it. If it comes after, it is “greater than”. Some examples:

```
(string<? "pearls" "swine") ⇒ true  ; "pearls" before "swine".
(string<? "pearls" "pasta") ⇒ false ; the "e" should come after the "a".
(string>? "kneel" "zod") ⇒ false    ; "kneel" before "zod".
(string<=? "pearls" "pearls") ⇒ true
(string<? "Pearls" "pearls") ⇒ true ; "P" before "p"
```

Here are more functions which operate on strings:

```
(string-append "alpha" "bet") ⇒ "alphabet"
(string-length "perpetual") ⇒ 9
(string-upcase "Hello") ⇒ "HELLO"
(string-downcase "Hello") ⇒ "hello"
(substring "substring" 3 6) ⇒ "str"
```

Ex. 12

Use `string-append` and `substring` to complete the function `chop-word`:

```
;; (chop-word s) selects some pieces of s.
;; Examples:
(check-expect (chop-word "In a hole in the ground there lived a hobbit.")
              ;;          ^    ^    ^    ^    ^    ^    ^    ^    ^
              ;;  index:  0    5   10   15   20   25   30   35   40
              "a hobbit lived in the ground")
(check-expect (chop-word "In a town by the forest there lived a rabbit.")
              ;;          ^    ^    ^    ^    ^    ^    ^    ^    ^
              ;;  index:  0    5   10   15   20   25   30   35   40
              "a rabbit lived by the forest")
(check-expect (chop-word "ab c defg hi jkl mnopqr stuvw xyzAB C DEFGHIJ")
              "C DEFGHI xyzAB hi jkl mnopqr")
```

Ex. 13

Use the constants `the-str` and `len-str`, along with the string functions `string-append`, `string-length`, and `number->string` to complete the function `describe-string`:

```
(define the-str "The string '")
(define len-str "' has length ")




;; (describe-string s) says a few words about s.
;; Examples:
(check-expect (describe-string "foo") "The string 'foo' has length 3")
(check-expect (describe-string "") "The string '' has length 0")
```

Symbols vs. strings

Consider the use of symbols when a small, fixed number of labels are needed (e.g. planets) that only need to be compared for equality.

Use strings when the set of values is more indeterminate (e.g. names of students), or when more computation is needed (e.g. comparison in alphabetical order).

Each built-in type has a predicate that consumes an `Any`, and produces `true` if the value is of that type, and `false` otherwise.

For example:

```
(symbol? 4) ⇒ false
```

Recap: Substitution rules (so far)

1. `(f v1...vn)` ⇒ `v` when `f` is built-in...
2. `(f v1...vn)` ⇒ `exp'` when `(define (f x1...xn) exp)` occurs to the left...
3. `id` ⇒ `val` when `(define id val)` occurs to the left.
4. `(and false ...)` ⇒ `false`
5. `(and true ...)` ⇒ `(and ...)`
6. `(and)` ⇒ `true`
7. `(or true ...)` ⇒ `true`
8. `(or false ...)` ⇒ `(or ...)`
9. `(or)` ⇒ `false`
10. `(cond [false exp] ...)` ⇒ `(cond ...)`
11. `(cond [true exp] ...)` ⇒ `exp`
12. `(cond [else exp])` ⇒ `exp`

We will add to this semantic model as we introduce new Racket features.

Doing a step-by-step reduction with these rules is called **tracing** a program. It is an important skill in any programming language. We will test this skill on assignments and exams.

Goals of this module

- You should understand Boolean data, and be able to perform and combine comparisons to test complex conditions on numbers.
- You should understand the syntax and use of a conditional expression.
- You should be aware of other types of data (symbols and strings), which will be used in future lectures.
- You should understand how to write tests with `check-expect` and use them in your assignment submissions.
- You should look for opportunities to use helper functions to structure your programs, and gradually learn when and where they are appropriate.
- You should be able to trace a program using the twelve substitution rules we've defined so far.

Summary: built-in functions

The following functions and special forms have been introduced in this module:

```
< <= = > >= and boolean? check-error cond else even? integer? negative? not
    number->string number? odd? or positive? string-append string-downcase
    string-length string-lower-case? string-numeric? string-upcase
    string-upper-case? string<=? string<? string=? string>=? string>? string?
    substring symbol=? symbol? zero?
```

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

```
* + - / < <= = > >= abs and boolean? ceiling check-error check-expect check-within cond
    cos define e else even? exp expt floor integer? log max min modulo negative? not
    number->string number? odd? or pi positive? quotient remainder round sgn sin sqr sqrt
    string-append string-downcase string-length string-lower-case? string-numeric?
    string-upcase string-upper-case? string<=? string<? string=? string>=? string>?
    string? substring symbol=? symbol? tan zero?
```