

# SICP Scheme Built-In Procedure Manual

---

Adapted from CS61A of UC Berkeley.

This document serves as a reference for the built-in procedures in the Scheme project and staff Scheme interpreter (the one in your homework and lab archive) . The [web interpreter](#) includes several additional built-in procedures.

In all of the syntax definitions below, `<x>` refers to a required element `x` that can vary, while `[x]` refers to an optional element `x`. Ellipses indicate that there can be more than one of the preceding element. It is assumed for all of these built-ins that the elements represent evaluated arguments not the literal expressions typed in.

## Core Interpreter

---

### **apply**

```
1 | (apply <procedure> <args>)
```

Calls `procedure` with the given list of `args`.

```
1 | scm> (apply + '(1 2 3))
2 | 6
```

On macros, this has the effect of calling the macro without the initial quoting or final evaluation. Thus, `apply` treats a macro as if it were a function.

### **display**

```
1 | (display <val>)
```

Prints `val`. If `val` is a Scheme string, it will be output without quotes.

A new line will not be automatically included.

### **displayln**

```
1 | (displayln <val>)
```

Like `display`, but includes a newline at the end.

### **error**

```
1 | (error <msg>)
```

Raises an `SchemeError` with `msg` as its message. If there is no `msg`, the error's message will be empty.

### **eval**

```
1 | (eval <expression>)
```

Evaluates `expression` in the current environment.

```
1 | scm> (eval '(cons 1 (cons 2 nil)))
2 | (1 2)
```

### **exit**

```
1 | (exit)
```

Exits the interpreter. In the web interpreter, this does nothing.

### **load**

```
1 | (load <filename>)
```

Loads the contents of the file with `filename` and evaluates the code within. `filename` must be a symbol. If that file is not found, `filename.scm` will be attempted.

The web interpreter's does not currently support `load`. The closest analog is `import-inline`, which takes a URL and evaluates the Scheme code in the current environment.

### **newline**

```
1 | (newline)
```

Prints a new line.

### **print**

```
1 | (print <val>...)
```

Prints the Scheme representation of each `val`, separated by spaces. Unlike `display`, this will include the outer quotes on a Scheme string, and a newline.

## Type Checking

---

### **atom?**

```
1 | (atom? <arg>)
```

Returns true if `arg` is a boolean, number, symbol, string, or nil; false otherwise.

### **boolean?**

```
1 | (boolean? <arg>)
```

Returns true if `arg` is a boolean; false otherwise.

### **integer?**

```
1 | (integer? <arg>)
```

Returns true if `arg` is a integer; false otherwise.

### **list?**

```
1 | (list? <arg>)
```

Returns true if `arg` is a well-formed list (i.e., it doesn't contain a stream); false otherwise. If the list has a cycle, this may cause an error or infinite loop.

```
1 | scm> (list? '(1 2 3))
2 | True
3 | scm> (list? (cons-stream 1 nil))
4 | False
```

### **number?**

```
1 | (number? <arg>)
```

Returns true if `arg` is a number; false otherwise.

### **null?**

```
1 | (null? <arg>)
```

Returns true if `arg` is `nil` (the empty list); false otherwise.

### **pair?**

```
1 | (pair? <arg>)
```

Returns true if `arg` is a pair; false otherwise.

### **procedure?**

```
1 | (procedure? <arg>)
```

Returns true if `arg` is a procedure; false otherwise.

### promise?

```
1 | (promise? <arg>)
```

Returns true if `arg` is a promise; false otherwise.

### string?

```
1 | (string? <arg>)
```

Returns true if `arg` is a string; false otherwise.

### symbol?

```
1 | (symbol? <arg>)
```

Returns true if `arg` is a symbol; false otherwise.

## Pair and List Manipulation

### append

```
1 | (append [lst] ...)
```

Returns the result of appending the items of all `lst`s in order into a single list. Returns `nil` if no `lst`s.

```
1 scm> (append '(1 2 3) '(4 5 6))
2 (1 2 3 4 5 6)
3 scm> (append)
4 ()
5 scm> (append '(1 2 3) '(a b c) '(foo bar baz))
6 (1 2 3 a b c foo bar baz)
7 scm> (append '(1 2 3) 4)
8 Error
```

### car

```
1 | (car <pair>)
```

Returns the `car` of `pair`. Errors if `pair` is not a pair.

### cdr

```
1 | (cdr <pair>)
```

Returns the `cdr` of `pair`. Errors if `pair` is not a pair.

## cons

```
1 | (cons <first> <rest>)
```

Returns a new pair with `first` as the `car` and `rest` as the `cdr`.

## length

```
1 | (length <arg>)
```

Returns the length of `arg`. If `arg` is not a list, this will cause an error.

## list

```
1 | (list <item> ...)
```

Returns a list with the `item`s in order as its elements.

## map

```
1 | (map <proc> <lst>)
```

Returns a list constructed by calling `proc` (a one-argument procedure) on each item in `lst`.

## filter

```
1 | (filter <pred> <lst>)
```

Returns a list consisting of only the elements of `lst` that return true when called on `pred` (a one-argument procedure).

## reduce

```
1 | (reduce <combiner> <lst>)
```

Returns the result of sequentially combining each element in `lst` using `combiner` (a two-argument procedure). `reduce` works from left-to-right, with the existing combined value passed as the first argument and the new value as the second argument. `lst` must contain at least one item.

# Mutation

## set-car!

```
1 | (set-car! <pair> <value>)
```

Sets the `car` of `pair` to `value`. `pair` must be a pair.

## set-cdr!

```
1 | (set-cdr! <pair> <value>)
```

Sets the `cdr` of `pair` to `value`. `pair` must be a pair.

## Arithmetic Operations

+

```
1 | (+ [num] ...)
```

Returns the sum of all `num`s. Returns 0 if there are none. If any `num` is not a number, this will error.

-

```
1 | (- <num> ...)
```

If there is only one `num`, return its negation. Otherwise, return the first `num` minus the sum of the remaining `num`s. If any `num` is not a number, this will error.

\*

```
1 | (* [num] ...)
```

Returns the product of all `num`s. Returns 1 if there are none. If any `num` is not a number, this will error.

/

```
1 | (/ <dividend> [divisor] ...)
```

If there are no `divisor`s, return 1 divided by `dividend`. Otherwise, return `dividend` divided by the product of the `divisors`. This built-in does true division, not floor division. `dividend` and all `divisor`s must be numbers.

```
1 | scm> (/ 4)
2 | 0.25
3 | scm> (/ 7 2)
4 | 3.5
5 | scm> (/ 16 2 2 2)
6 | 2
```

**abs**

```
1 | (abs <num>)
```

Returns the absolute value of `num`, which must be a number.

### expt

```
1 | (expt <base> <power>)
```

Returns the `base` raised to the `power` power. Both must be numbers.

### modulo

```
1 | (modulo <a> <b>)
```

Returns `a` modulo `b`. Both must be numbers.

```
1 | scm> (modulo 7 3)
2 | 1
3 | scm> (modulo -7 3)
4 | 2
```

### quotient

```
1 | (quotient <dividend> <divisor>)
```

Returns `dividend` integer divided by `divisor`. Both must be numbers.

```
1 | scm> (quotient 7 3)
2 | 2
```

### remainder

```
1 | (remainder <dividend> <divisor>)
```

Returns the remainder that results when `dividend` is integer divided by `divisor`. Both must be numbers. Differs from `modulo` in behavior when negative numbers are involved.

```
1 | scm> (remainder 7 3)
2 | 1
3 | scm> (remainder -7 3)
4 | -1
```

## Additional Math Procedures

The Python-based interpreter adds the following additional procedures whose behavior exactly match the corresponding Python functions in the [math module](#).

- `acos`
- `acosh`
- `asin`

- `asinh`
- `atan`
- `atan2`
- `atanh`
- `ceil`
- `copysign`
- `cos`
- `cosh`
- `degrees`
- `floor`
- `log`
- `log10`
- `log1p`
- `log2`
- `radians`
- `sin`
- `sinh`
- `sqrt`
- `tan`
- `tanh`
- `trunc`

## Boolean Operations

---

### General

`eq?`

```
1 | (eq? <a> <b>)
```

If `a` and `b` are both numbers, booleans, symbols, or strings, return true if they are equivalent; false otherwise.

Otherwise, return true if `a` and `b` both refer to the same object in memory; false otherwise.



```
1 scm> (eq? '(1 2 3) '(1 2 3))
2 False
3 scm> (define x '(1 2 3))
4 scm> (eq? x x)
5 True
```

### equal?

```
1 (equal? <a> <b>)
```

Returns true if `a` and `b` are equivalent. For two pairs, they are equivalent if their `car`s are equivalent and their `cdr`s are equivalent.

```
1 scm> (equal? '(1 2 3) '(1 2 3))
2 True
```

### not

```
1 (not <arg>)
```

Returns true if `arg` is false-y or false if `arg` is truthy.

## On Numbers

### =

```
1 (= <a> <b>)
```

Returns true if `a` equals `b`. Both must be numbers.

### <

```
1 (< <a> <b>)
```

Returns true if `a` is less than `b`. Both must be numbers.

### >

```
1 (> <a> <b>)
```

Returns true if `a` is greater than `b`. Both must be numbers.

### <=

```
1 (<= <a> <b>)
```

Returns true if `a` is less than or equal to `b`. Both must be numbers.

**>=**

```
1 | (>= <a> <b>)
```

Returns true if `a` is greater than or equal to `b`. Both must be numbers.

**even?**

```
1 | (even? <num>)
```

Returns true if `num` is even. `num` must be a number.

**odd?**

```
1 | (odd? <num>)
```

Returns true if `num` is odd. `num` must be a number.

**zero?**

```
1 | (zero? <num>)
```

Returns true if `num` is zero. `num` must be a number.

## Promises and Streams

---

**force**

```
1 | (force <promise>)
```

Returns the evaluated result of `promise`. If `promise` has already been forced, its expression will not be evaluated again. Instead, the result from the previous evaluation will be returned. `promise` must be a promise.

**cdr-stream**

```
1 | (cdr-stream <stream>)
```

Shorthand for `(force (cdr <stream>))`.

## Turtle Graphics

---

**backward**

```
1 | (backward <n>)
```

Moves the turtle backward `n` units in its current direction from its current position.

Aliases: `back`, `bk`

### `begin_fill`

```
1 | (begin_fill)
```

Starts a sequence of moves that outline a shape to be filled. Call `end_fill` to complete the fill.

### `bgcolor`

```
1 | (bgcolor <c>)
```

Sets the background color of the turtle window to a color `c` (same rules as when calling `color`).

### `circle`

```
1 | (circle <r> [extent])
```

Draws a circle of radius `r`, centered `r` units to the turtle's left. If `extent` exists, draw only the first `extent` degrees of the circle. If `r` is positive, draw in the counterclockwise direction. Otherwise, draw in the clockwise direction.

The web interpreter has trouble accurately drawing partial circles.

### `clear`

```
1 | (clear)
```

Clears the drawing, leaving the turtle unchanged.

### `color`

```
1 | (color <c>)
```

Sets the pen color to `c`, which is a Scheme string such as "red" or "#ffc0c0".

The web interpreter also allows `c` to be a symbol. Available named colors may vary depending on the interpreter.

### `end_fill`

```
1 | (end_fill)
```

Fill in shape drawn since last call to `begin_fill`.

### `exitonclick`

```
1 | (exitonclick)
```

In pillow-turtle mode, this exits the current program. In tk-turtle mode, it exits the current program when the window is clicked. In the web interpreter, it closes the canvas.

In the local interpreter, you can pass `--turtle-save-path PATH` to also effectively call `(save-to-file PATH)` right before exit.

### **forward**

```
1 | (forward <n>)
```

Moves the turtle forward `n` units in its current direction from its current position.

Alias: `fd`

### **hideturtle**

```
1 | (hideturtle)
```

Makes the turtle invisible.

This procedure has no effect on the web interpreter, as the turtle is always invisible.

Alias: `ht`

### **left**

```
1 | (left <n>)
```

Rotates the turtle's heading `n` degrees counterclockwise.

Alias: `lt`

### **pendown**

```
1 | (pendown)
```

Lowers the pen so that the turtle starts drawing.

Alias: `pd`

### **penup**

```
1 | (penup)
```

Raises the pen so that the turtle does not draw.

Alias: `pu`

### **pixel**

```
1 | (pixel <x> <y> <c>)
```

Draws a box filled with pixels starting at ( `x`, `y` ) in color `c` (same rules as in `color`). By default the box is one pixel, though this can be changed with `pixelsize`.

#### `pixelsize`

```
1 | (pixelsize <size>)
```

Changes the size of the box drawn by `pixel` to be `size x size`.

#### `rgb`

```
1 | (rgb <r> <g> <b>)
```

Returns a color string formed from `r`, `g`, and `b` values between 0 and 1.

#### `right`

```
1 | (right <n>)
```

Rotates the turtle's heading `n` degrees clockwise.

Alias: `rt`

#### `save-to-file`

```
1 | (save-to-file <f>)
```

Saves the current canvas to a file specified by `f`, with an added file extension.

For example, `(save-to-file "hi")`

- saves to `./hi.png` in the local interpreter using the pillow-turtle
- saves to `./hi.ps` in the local interpreter using the tk-turtle (default)
- has no effect in the web interpreter

#### `screen_width`

```
1 | (screen_width)
```

Returns the width of the turtle screen in pixels of the current size.

#### `screen_height`

```
1 | (screen_height)
```

Returns the height of the turtle screen in pixels of the current size.

#### `setheading`

```
1 | (setheading <h>)
```

Sets the turtle's heading `h` degrees clockwise from the north.

Alias: `seth`

### **setposition**

```
1 | (setposition <x> <y>)
```

Moves the turtle to position (`x`, `y`) without changing its heading.

Aliases: `setpos`, `goto`

### **showturtle**

```
1 | (showturtle)
```

Makes the turtle visible.

This procedure has no effect on the web interpreter, as the turtle is always invisible.

Alias: `st`

### **speed**

```
1 | (speed <s>)
```

Sets the turtle's animation speed to some value between 0 and 10 with 0 indicating no animation and 1-10 indicating faster and faster movement.

On the local interpreter in tk-turtle mode, this changes the animation speed. This feature has no effect on the web interpreter and on the gui-less pillow-turtle mode.