Scheme Built-In Procedure Reference

> Last Updated: Fall 2021

This document serves as a reference for the built-in procedures in the Scheme project and staff Scheme interpreter (http://cs61a.org/assets/interpreter/scheme). The web interpreter (https://scheme.cs61a.org) 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

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

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

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.

Example:

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

display

```
(display <val>)
```

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

A newline will not be automatically included.

displayln

```
(displayln <val>)
```

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

error

```
(error <msg>)
```

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

eval

```
(eval <expression>)
```

Evaluates `expression` in the current environment.

Example:

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

exit

```
(exit)
```

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

load

```
(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 does not currently support `load`.

newline

```
(newline)
```

Prints a newline.

print

```
(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?

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

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

boolean?

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

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

integer?

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

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

list?

```
(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.

Example:

```
scm> (list? '(1 2 3))
#t
scm> (list? (cons-stream 1 nil))
#f
```

number?

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

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

null?

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

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

pair?

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

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

procedure?

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

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

promise?

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

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

string?

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

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

symbol?

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

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

Pair and List Manipulation

append

```
(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.

Example:

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

car

```
(car <pair>)
```

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

cdr

```
(cdr <pair>)
```

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

cons

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

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

length

```
(length <arg>)
```

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

list

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

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

map

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

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

filter

```
(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

```
(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!

> This builtin is included in the staff interpreter and the web interpreter, but it is not in scope for the course and is not included in the project.

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

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

set-cdr!

> This builtin is included in the staff interpreter and the web interpreter, but it is not in scope for the course and is not included in the project.

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

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

Arithmetic Operations

+

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

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

-

```
(- <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.

*

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

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

/

```
(/ <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.

Example:

```
scm> (/ 4)
0.25
scm> (/ 7 2)
3.5
scm> (/ 16 2 2 2)
2
```

abs

```
(abs <num>)
```

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

expt

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

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

modulo

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

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

Example:

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

quotient

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

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

Example:

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

remainder

```
(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.

Example:

```
scm> (remainder 7 3)
1
scm> (remainder -7 3)
-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 (https://docs.python.org/3/library/math.html).

- 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?

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

If `a` and `b` are both numbers, booleans, or symbols, 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.

Example:

```
scm> (eq? '(1 2 3) '(1 2 3))
#f
scm> (define x '(1 2 3))
scm> (eq? x x)
#t
```

equal?

```
(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.

Example:

```
scm> (equal? '(1 2 3) '(1 2 3))
#t
```

not

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

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

On Numbers

=

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

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

<

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

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

>

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

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

<=

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

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

>=

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

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

even?

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

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

odd?

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

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

zero?

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

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

Promises and Streams

force

> This builtin is included in the staff interpreter and the web interpreter, but it is not in scope for the course and is not included in the project.

```
(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

> This builtin is included in the staff interpreter and the web interpreter, but it is not in scope for the course and is not included in the project.

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

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

Turtle Graphics

backward

```
(backward <n>)
```

Alternatively,

```
(back <n>)
```

Alternatively,

```
(bk <n>)
```

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

begin_fill

```
(begin_fill)
```

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

bgcolor

```
(bgcolor <c>)
```

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

circle

```
(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.

clear

```
(clear)
```

Clears the drawing, leaving the turtle unchanged.

color

```
(color <c>)
```

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

end_fill

```
(end_fill)
```

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

exitonclick

```
(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 does nothing.

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

forward

```
(forward <n>)
```

Alternatively,

```
(fd <n>)
```

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

hideturtle

```
(hideturtle)
```

Alternatively,

```
(ht)
```

Makes the turtle invisible.

left

```
(left <n>)

(lt <n>)
```

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

pendown

```
(pendown)
```

Alternatively,

```
(pd)
```

Lowers the pen so that the turtle starts drawing.

penup

```
(penup)
```

Alternatively,

```
(pu)
```

Raises the pen so that the turtle does not draw.

pixel

```
(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

```
(pixelsize <size>)
```

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

rgb

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

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

right

```
(right <n>)

(rt <n>)
```

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

save-to-file

```
(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

```
(screen_width)
```

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

screen_height

```
(screen_height)
```

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

setheading

```
(setheading <h>)

(seth <h>)
```

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

setposition

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

Alternatively,

```
(setpos <x> <y>)
```

Alternatively,

```
(goto <x> <y>)
```

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

showturtle

```
(showturtle)

(st)
```

Makes the turtle visible.

speed

```
(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.

Additional Reading

- Scheme Specification (/articles/scheme-spec/) - the core specification of 61A Scheme
- R5RS Specification (http://www.schemers.org/Documents/Standards/R5RS/) - the full Scheme specificaton that 61A Scheme most closely resembles.