# SICP Scheme Specification

> Adapted from cs61a of UC Berkeley.

## About This Specification

The version of Scheme used in this course is not perfectly true to any official specification of the language, though it is perhaps closest to R5RS, with some features and terminology from other versions, both older and newer. We deviate from the official specifications for several reasons, including ease of implementation (both for the staff in reference implementations and for students in completing the Scheme project) and ease of instruction.

This document and the linked built-in procedure reference are very long and are an attempt to formalize the variant of Scheme used in 61A. Lectures, labs, and discussion are probably better resources for first learning the language. However, the Overview and Terminology section may be useful if you wish to read a more formal description of the language.

You should not find it necessary to read the full references of special forms and built-in procedures, but specific sections may be helpful to reference when working on Scheme assignments and the project (the relevant sections will typically be linked in the assignment itself).

There are, in effect, two different reference implementations of 61A Scheme: the Python-based staff interpreter, provided in Scheme labs and homeworks, and the Dart-based web interpreter (interpreter, source code). A completed and correct implementation of the Scheme project, including all extra credit problems, should match the functionality of the staff interpreter, excluding error tracing.

This document primarily focuses on the Python-based interpreter. The web interpreter should largely match its behavior, but due to the different host language and restraints of the web platform, some inconstencies may exist, for which you can file issues. Additionally, the web interpreter contains several extra features, such as a diagrammer and JS interopability that are not documented here.

## Overview and Terminology

### Expressions and Environments

Scheme works by evaluating **expressions** in **environments**. Every expression evaluates to a **value**. Some expressions are **self-evaluating**, which means they are both an expression and a value, and that it evaluates to itself.

A **frame** is a mapping from symbols (names) to values, as well as an optional parent frame. The current environment refers to the current frame, as well as a chain of parent frames up to the **global frame** (which has no parent). When looking up a symbol in an environment, Scheme first checks the current frame and returns the corresponding value if it exists. If it doesn't, it repeats this process on each subsequent parent frame, until either the symbol is found, or there are no more parent frames to check.

## Atomic Expressions

There are several **atomic** or **primitive** expressions. Numbers, booleans, strings, and the empty list (`nil`) are all both atomic and self-evaluating. Symbols are atomic, but are not self-evaluating (they instead evaluate to a value that was previously bound to it in the environment).

## Call Expressions

The Scheme expressions that are not atomic are called **combinations**, and consist of one or more **subexpressions** between parentheses. Most forms are evaluated as **call expressions**, which has three evaluation steps:

1. Evaluate the first subexpression (the operator), which must evaluate to a **procedure** (see below).
2. Evaluate the remaining subexpressions (the operands) in order.
3. Apply the procedure from step 1 to the evaluated operands (**arguments**) from step 2.

These steps mirror those in Python and other languages.

## Special Forms

However, not all combinations are call expressions. Some are **special forms**. The interpreter maintains a set of particular symbols (sometimes called **keywords**) that signal that a combination is a special form when they are the first subexpression. Each special form has it's own procedure for which operands to evaluate and how (described below). The interpreter always checks the first subexpression of a combination first. If it matches one of the keywords, the corresponding special form is used. Otherwise, the combination is evaluated as a call expression.

## Symbolic Programming

Scheme's core data type is the **list**, built out of pairs as described below. Scheme code is actually built out of these lists. This means that the code `(+ 1 2)` is constructed as a list of the `+` symbol, the number 1, and the number 2, which is then evaluated as a call expression.

Since lists are normally evaluated as combinations, we need a special form to get the actual, unevaluated list. `quote` is a special form that takes a single operand expression and returns it, unevaluated. Therefore, `(quote (+ 1 2))` returns the actual list of the symbol `+`, the number 1, and the number 2, rather than evaluating the expression to get the number 3. This also works for symbols. `a` is looked up in the current environment to get the corresponding value, while `(quote a)` evaluates to the literal symbol `a`.

Because `quote` is so commonly used in Scheme, the language has a shorthand way of writing it: just put a single quote in front of the expression you want to leave unevaluated. `'(+ 1 2)` and `'a` are equivalent to `(quote (+ 1 2))` and `(quote a)`, respectively.

## Miscellaneous

Like R5RS, SICP Scheme is entirely case-insensitive (aside from strings). This specification will use lowercase characters in symbols, but the corresponding uppercase characters may be used interchangeably.

# Types of Values

## Numbers

Numbers are built on top of Python's number types and can thus support a combination of arbitrarily-large integers and double-precision floating points.

The web interpreter attempts to replicate this when possible, though may deviate from Python-based versions due to the different host language and the need to work-around the quirks of JavaScript when running in a browser.

Any valid real number literal in the interpreter's host language should be properly read. You should not count on consistent results when floating point numbers are involved in any calculation or on any numbers with true division.

## Booleans

There are two boolean values: `#t` and `#f`. Scheme booleans may be input either as their canonical `#t` or `#f` or as the words `true` or `false`.

Any expression may be evaluated in a boolean context, but `#f` is the only value that is false. All other values are treated as true in a boolean context.

Some interpreters prior to Spring 2018 displayed the words `true` and `false` when booleans were output, but this should not longer be the case in any interpreter released/updated since then.

## Symbols

Symbols are used as identifiers in Scheme. Valid symbols consist of some combination of alphanumeric characters and/or the following special characters:

```
!$%&*/:<=>?@^_~+-.
```

All symbols should be internally stored with lowercase letters. Symbols must not form a valid integer or floating-point number.

## Strings

Unlike other implementations, 61A Scheme has no concept of individual characters. Strings are considered atomic data types in their own right. Strings can be entered into the intepreter as a sequence of characters inside double quotes, with certain characters, such as line breaks and double quotes escaped. As a general rule, if a piece of text would be valid as a JSON key, it should work as a string in 61A Scheme. Strings in 61A Scheme are immutable, in contrast to most other Scheme implementations.

These differences in how strings behave are due to the status of strings in the host languages: Python and Dart both have immutable strings with no concept of individual characters.

Because the Python-based interpreter has little use for strings, it lacks proper support for their manipulation. The web interpreter, which requires strings for JS interop (among other things), it supports a `string-append` built-in, which takes in an arbitrary number of values or any type and combines them into a string. Additional string manipulation can be done through JS interop.

## Pairs and Lists

Pairs are a built-in data structure consisting of two fields, a `car` and a `cdr` (also sometimes called first and second, or first and rest). The first value can contain any scheme datatype. However, the second value must contain nil, a pair, or a stream promise.

`nil` is a special value in Scheme which represents the empty list. It can be inputted by typing `nil` or `()` into the interpreter.

A **list** is defined as either `nil` or a pair whose `cdr` is another list. Pairs are displayed as a parenthesized, space separated, sequence of the elements in the sequence they represent. For example, `(cons (cons 1 nil) (cons 2 nil))` is displayed as `((1) 2)`. Note that this means that `cons` is asymmetric.

> There is one exception to the above rule in the case of streams. Streams are represented as the `car` of the stream, followed by a dot, followed by the promise that makes up its cdr. For example
>
> ```
> scm> (cons-stream 1 nil)
> (1 . #[promise (not forced)])
> ```

List literals can be constructed through the quote special form, so `(cons 1 (cons 'a nil))` and `'(1 a)` are equivalent.

## Procedures

Procedures represent some subroutine within a Scheme program. Procedures are first-class in Scheme, meaning that they can be bound to names and passed around just like any other Scheme value. Procedures are equivalent to functions in most other languages, and the two terms are sometimes used interchangeably.

Procedures can be called on some number of arguments, performing some number of actions and then returning some Scheme value.

A procedure call can be performed with the syntax `(<operator> <operand> ...)`, where `<operator>` is some expression that evaluates to a procedure and each `<operand>` (of which there can be any number, including 0) evaluates to one of the procedure's arguments. The term "procedure call" is used interchangeably with the term "call expression."

There are several types of procedures. Built-in procedures (or just built-ins) are built-in to the interpreter and already bound to names when it is started (though it is still possible for you to rebind these names). A list of all the built-in procedures in the Python-based interpreter is available in the Scheme built-ins document.

Lambda procedures are defined using the `lambda` or `define` special forms (see below) and create a new frame whose parent is the frame in which the lambda was defined in when called. The expressions in the lambda's body are than evaluated in this new environment. Mu procedures are similar, but the new frame's parent is the frame in which the `mu` is called, not the frame in which it was created.

SICP Scheme also has macro procedures, which must be defined with the `define-macro` special form. Macros work similarly to lambdas, except that they pass the argument expressions in the call expression into the macro instead of the evaluated arguments and they then evaluate the expression the macro returns in the calling environment afterwards. The modified process for evaluating macro call expressions is:

1. Evaluate the operator. If it is not a macro procedure, follow the normal call expression steps.
2. Apply the macro procedure from step 1 to the unevaluated operands.
3. Once the macro returns, evaluate that value in the calling environment.

Macros effectively let the user define new special forms. Macro procedures take in unevaluated operand expressions and should generally return a piece of Scheme code that the macro is equivalent to.

## Promises and Streams

Promises represent the delayed evaluation of an expression in an environment. They can be constructed by passing an expression into the `delay` special form. The evaluation of a promise can be forced by passing it into the `force` built-in. The expression of a promise will only ever be evaluated once. The first call of `force` will store the result, which will be immediately returned on subsequent calls of `force` on the same promise.

A promise must contain a pair or nil since it is used as the `cdr` of a stream. If it is found to contain something else when forced, `force` will error. If `force` errors for any reason, the promise remains unforced.

For example

```
scm> (define p (delay (begin (print "hi") (/ 1 0))))
p
scm> p
#[promise (unforced)]
scm> (force p)
hi
Error
scm> p
#[promise (unforced)]
scm> (force p)
hi
Error
```

Or, for an example with type errors:

```
scm> (define p (delay (begin (print "hi") 2)))
p
scm> p
#[promise (unforced)]
scm> (force p)
hi
Error
scm> p
#[promise (unforced)]
scm> (force p)
hi
Error
```

Promises are used to define **streams**, which are to lists what promises are to regular values. A stream is defined as a pair where the cdr is a promise that evaluates to another stream or `nil`. The `cons-stream` special form and the `cdr-stream` built-in are provided make the construction and manipulation of streams easier. `(cons-stream a b)` is equivalent to `(cons a (delay b))` while `(cdr-stream x)` is equivalent to `(force (cdr x))`.

> A note for those familiar with promises in languages like JavaScript: although Scheme promises and JS-style promises originate from the same general concept, JS promises are best described as a placeholder for a value that is computed asynchronously. The Python-based 61A Scheme interpreter has no concept of asynchrony, so its promises only represent delayed evaluation. The web interpreter continues to use promises in this way, but adds a "future" type to stand in place for JS promises.

# Special Forms

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.

The following special forms are included in all versions of 61A Scheme.

## `define`

```
(define <name> <expression>)
```

Evaluates `<expression>` and binds the value to `<name>` in the current environment. `<name>` must be a valid Scheme symbol.

```
(define (<name> [param] ...) <body> ...)
```

Constructs a new lambda procedure with `param`s as its parameters and the `body` expressions as its body and binds it to `name` in the current environment. `name` must be a valid Scheme symbol. Each `param` must be a unique valid Scheme symbol. This shortcut is equivalent to:

```
(define <name> (lambda ([param] ...) <body> ...))
```

However, some interpreters may give lambdas created using the shortcut an intrinsic name of `name` for the purpose of visualization or debugging.

In either case, the return value is the symbol `<name>`.

```
scm> (define x 2)
x
scm> (define (f x) x)
f
```

## Variadic functions

In staff implementations of the scheme language, you can define a function that takes a variable number of arguments by using the `variadic` special form. The construct `variadic` constructs a "variadic symbol" that is bound to multiple rather than a single variable. This is only allowed at the end of an arguments list

```
scm> (define (f x (variadic y)) (append y (list x)))
f
scm> (f 1 2 3)
(2 3 1)
scm> (define (f (variadic y) x) (append y (list x)))
Error
```

This is also possible in lambdas:

```
scm> (define f (lambda (x (variadic y)) (append y (list x))))
f
scm> (f 1 2 3)
(2 3 1)
scm> (define my-list (lambda ((variadic x)) x))
my-list
scm> (my-list 2 3 4)
(2 3 4)
```

You can use the special symbol `.` to construct the `variadic` special form:

```
scm> (define (f x . y) (append y (list x)))
f
scm> (f 1 2 3)
(2 3 1)
scm> '. x
(variadic x)
```

This is analogous to `,` for `unquote`.

> Note: this is pretty much the same as `*args` in python, except that you can't call a function using `variadic`, you instead have to use the `#[apply]` built-in function.

## if

```
(if <predicate> <consequent> [alternative])
```

Evaluates `predicate`. If true, the `consequent` is evaluated and returned. Otherwise, the `alternative`, if it exists, is evaluated and returned (if no `alternative` is present in this case, the return value is undefined).

## cond

```
(cond <clause> ...)
```

Each `clause` may be of the following form:

```
(<test> [expression] ...)
```

The last `clause` may instead be of the form `(else [expression] ...)`, which is equivalent to `(#t [expression] ...)`.

Starts with the first `clause`. Evaluates `test`. If true, evaluate the `expression`s in order, returning the last one. If there are none, return what `test` evaluated to instead. If `test` is false, proceed to the next `clause`. If there are no more `clause`s, the return value is undefined.

## and

```
(and [test] ...)
```

Evaluate the `test`s in order, returning the first false value. If no `test` is false, return the last `test`. If no arguments are provided, return `#t`.

## or

```
(or [test] ...)
```

Evaluate the `test`s in order, returning the first true value. If no `test` is true and there are no more `test`s left, return `#f`.

## let

```
(let ([binding] ...) <body> ...)
```

Each `binding` is of the following form:

```
(<name> <expression>)
```

First, the `expression` of each `binding` is evaluated in the current frame. Next, a new frame that extends the current environment is created and each `name` is bound to its corresponding evaluated `expression` in it.

Finally the `body` expressions are evaluated in order, returning the evaluated last one.

## `begin`

```
(begin <expression> ...)
```

Evaluates each `expression` in order in the current environment, returning the evaluated last one.

## `lambda`

```
(lambda ([param] ...) <body> ...)
```

Creates a new lambda with `param`s as its parameters and the `body` expressions as its body. When the procedure this form creates is called, the call frame will extend the environment this lambda was defined in.

## `mu`

```
(mu ([param] ...) <body> ...)
```

Creates a new mu procedure with `param`s as its parameters and the `body` expressions as its body. When the procedure this form creates is called, the call frame will extend the environment the mu is called in.

## `quote`

```
(quote <expression>)
```

Returns the literal `expression` without evaluating it.

`'<expression>` is equivalent to the above form.

## `delay`

```
(delay <expression>)
```

Returns a promise of `expression` to be evaluated in the current environment.

## `cons-stream`

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

Shorthand for `(cons <first> (delay <rest>))`.

## `set!`

```
(set! <name> <expression>)
```

Evaluates `expression` and binds the result to `name` in the first frame it can be found in from the current environment. If `name` is not bound in the current environment, this causes an error.

The return value is undefined.

## `quasiquote`

```
(quasiquote <expression>)
```

Returns the literal `expression` without evaluating it, unless a subexpression of `expression` is of the form:

```
(unquote <expr2>)
```

in which case that `expr2` is evaluated and replaces the above form in the otherwise unevaluated `expression`.

`` `<expression> `` is equivalent to the above form.

## `unquote`

See above. `,<expr2>` is equivalent to the form mentioned above.

## `unquote-splicing`

```
(unquote-splicing <expr2>)
```

> Note: This special form 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.

Similar to `unquote`, except that `expr2` must evaluate to a list, which is then spliced into the structure containing it in `expression`.

`,@<expr2>` is equivalent to the above form.

## `define-macro`

```
(define-macro (<name> [param] ...) <body> ...)
```

> Note: This special form is implemented as part of an extra credit problem.

Constructs a new macro procedure with `param`s as its parameters and the `body` expressions as its body and binds it to `name` in the current environment. `name` must be a valid Scheme symbol. Each `param` must be a unique valid Scheme symbol. `(<name> [param] ...)` can be variadic.

Macro procedures should be lexically scoped, like lambda procedures.