Specification of Scheme §3—2026 edition

Gan Jaye Jyn, Kyriel Abad, Martin Henz

National University of Singapore
School of Computing

April 13, 2026

The language "Scheme §3" (scm-slang) is a teaching language for the SICP courseware in Source Academy. It is broadly aligned with R5RS Scheme, but is not a complete implementation of the standard and documents its own deviations and library behavior.
This document specifies the Chapter 3 language, which is the default and full Scheme language used in Source Academy for scm-slang.

1 Changes

Compared to Scheme §2, Scheme §3 adds the following:

- `set!` mutation
- `begin` sequencing
- `delay` and promise support
- Vector literals (`#(...)` form)

2 Syntax

A Scheme program is a sequence of *datum* forms, defined using Backus–Naur Form as follows:

program	::=	*datum...*	program
datum	::=	*atom*	
	|	*list*	
	|	*vector*	
	|	*quoted*	
atom	::=	*number*	numeric literal
	|	**#t** | **#f**	boolean literal
	|	*string*	string literal
	|	*identifier*	identifier
list	::=	**(** *form...* **)**	proper list
	|	**(** *form...* **.** *form* **)**	dotted list
vector	::=	**#(** *form...* **)**	vector literal
quoted	::=	**'** *form*	quote
	|	**`** *form*	quasiquote
	|	**,** *form*	unquote
	|	**,@** *form*	unquote-splicing
	|	**(quote** *form***)**	
	|	**(quasiquote** *form***)**	
	|	**(unquote** *form***)**	
	|	**(unquote-splicing** *form***)**	
form	::=	*special-form*	
	|	*application*	
	|	*datum*	
special-form	::=	**(define** *define-form***)**	
	|	**(lambda** *formals body* + **)**	
	|	**(if** *form form* [*form*]**)**	
	|	**(let (**binding...**)** *body* + **)**	
	|	**(cond** *clause...* [*else-clause*]**)**	
	|	**(set!** *identifier form***)**	
	|	**(begin** *form* + **)**	
	|	**(delay** *form***)**	
	|	*import*	
	|	*export*	
define-form	::=	*identifier form*	
	|	**(***identifier formals***)** *body*+	
formals	::=	**(***identifier...***)**	
	|	*identifier*	
	|	**(***identifier... . identifier***)**	
binding	::=	**(***identifier form***)**	
clause	::=	**(***form body* ∗ **)**	
else-clause	::=	**(else** *body* + **)**	
application	::=	**(***form form...***)**	
import	::=	**(import** *string* **(***identifier...***))**	
export	::=	**(export (***define-form***))**	

Any non-whitespace, non-delimiter sequence may form an identifier, and numeric lexemes follow the number formats described in Section *Numbers*.

3 Lexical Structure

Whitespace and Comments

Whitespace separates tokens. Line comments begin with `;` and continue to end of line. Block comments are delimited by `#|` and `|#` and may span multiple lines. Datum comments use `#;` and discard the next syntactic datum.

Delimiters

Parentheses `( )` and square brackets `[ ]` are interchangeable and can be freely mixed, as long as they are properly balanced.

Identifiers

Identifiers are any non-whitespace sequences that are not one of the structural punctuation characters `( ) [ ] ; |`. This means identifiers may contain characters such as `+ - * / < > = ? !` and may begin with digits. Numeric lexemes are recognized separately (see Section *Numbers*).

Booleans

Booleans are written as `#t` and `#f`.

Strings

Strings are delimited by double quotes `"..."` and may contain any character except an unescaped double quote. Newlines are permitted inside strings and are preserved.

Quoting Tokens

Shorthand quoting tokens are supported: `'` (quote), `` ` `` (quasiquote), `,` (unquote), and `,@` (unquote-splicing).

Vectors

Vector literals use the reader form `#( ... )` or `#[ ... ]`.

4 Evaluation Rules and Restrictions

Truthiness

scm-slang follows Scheme truthiness: only `#f` is false. All other values are true.

Definitions and Mutation

- `define` introduces a new binding in the current environment.
- `set!` requires that the identifier already be defined; otherwise it is an error.
- Redefinition is permitted if the identifier is already in scope; scm-slang does not block shadowing across nested scopes created by `lambda` or `let`.

Sequencing

`begin` evaluates its subexpressions in order and returns the value of the last expression. A `begin` form does not introduce a new scope.

Conditionals

`if` accepts two or three operands. If the alternative is omitted, the value is `undefined`. `cond` evaluates predicates in order and returns the first consequent whose predicate is true. An `else` clause, if present, must appear last.

Delayed Evaluation

`delay` produces a promise. `force` evaluates a promise at most once and memoizes the result.

Quotation

Quoted data are not evaluated. Quasiquotation supports unquote and unquote-splicing; unquote and splicing are only valid inside a quasiquoted context.

Imports and Exports

`import` and `export` follow a scm-slang specific syntax that is compatible with the Source Academy module system. Imports must appear at the top level.

5 Names

Identifiers are lexemes that are not whitespace and not structural punctuation. The following keywords are reserved and cannot be used as identifiers unless quoted: `define`, `lambda`, `if`, `let`, `cond`, `else`, `quote`, `quasiquote`, `unquote`, `unquote-splicing`, `begin`, `set!`, `delay`, `import`, `export`.

6 Numbers

scm-slang implements a numeric tower with exact and inexact numbers, including integers, rationals, reals and complex numbers. Numeric literals are recognized using the scm-slang number parser and include:

- Integers in decimal notation (e.g. `0`, `-42`)
- Rationals using `/` (e.g. `3/4`)
- Reals with decimal points and exponent notation
- Complex numbers in rectangular form (e.g. `1+2i`)
- Special values: `+inf.0`, `-inf.0`, and `+nan.0`

Predicates and constructors include `number?`, `integer?`, `rational?`, `real?`, `complex?`, `exact?`, and `inexact?`. Arithmetic is provided by `+`, `-`, `*`, `/` and comparison predicates such as `=` and `<`.

7 Strings

Strings are delimited by double quotes and evaluate to immutable string values. The standard library provides operations such as `string`, `string-length`, `string-ref`, `substring`, `string-append`, `string->number` and `number->string`.

8 Dynamic Type Checking

scm-slang is dynamically typed. Operations signal errors when given values of the wrong kind. For example, `car` and `cdr` require pairs, `vector-ref` requires a vector and a valid index, numeric operators require numbers, and `set!` requires a previously defined identifier.

9 Standard Library

The standard library provides built-in constants and procedures that are always available in the language. This specification summarizes the most commonly used operations.

Core Library Summary

- Errors: `error`
- Booleans: `boolean?`, `not`, `and`, `or`
- Numbers: `number?`, `integer?`, `rational?`, `real?`, `complex?`, `exact?`, `inexact?`, `+`, `-`, `*`, `/`, `=`, `<`, `<=`, `>`, `>=`, `abs`, `quotient`, `remainder`, `modulo`, `gcd`, `lcm`, `exp`, `log`, `sqrt`, `sin`, `cos`, `tan`, `floor`, `ceiling`, `round`
- Pairs and lists: `cons`, `car`, `cdr`, `set-car!`, `set-cdr!`, `list`, `null?`, `list?`, `append`, `map`, `fold`, `filter`, `length`
- Vectors: `vector`, `vector?`, `vector-length`, `vector-ref`, `vector-set!`, `vector->list`, `list->vector`
- Symbols: `symbol?`, `symbol=?`, `string->symbol`
- Strings: `string`, `string-length`, `string-ref`, `substring`, `string-append`, `string->number`, `number->string`
- Equality: `eq?`, `eqv?`, `equal?`
- Promises: `make-promise`, `promise?`, `force`, `promise-forced?`
- Output: `display` (returns its argument)

Deviations from R5RS Scheme

scm-slang is intended to be compatible with R5RS where possible, but there are intentional deviations:

- Continuations are not supported by the parser itself. Continuations are available only when executed in conjunction with the Source Academy Explicit Control Evaluator.
- Macros are not implemented in the Chapter 3 language.
- Character and bytevector types are not implemented.
- Parentheses `( )` and brackets `[ ]` are interchangeable.
- Named `let` is not supported.
- Import/export uses a scm-slang specific syntax compatible with Source Academy modules, e.g.
- `display` returns its argument instead of an unspecified value, matching Source's display behavior for debugging convenience.

```
(import "runes" (stack beside))
(export (define foo "bar"))
```