CX Programming Language Specification

ALEX SHINN, JOHN COWAN, AND ARTHUR A. GLECKLER (*Editors*)

STEVEN GANZ
AARON W. HSU
BRADLEY LUCIER
EMMANUEL MEDERNACH
ALEXEY RADUL
JEFFREY T. READ
DAVID RUSH
BENJAMIN L. RUSSEL
OLIN SHIVERS
ALARIC SNELL-PYM
GERALD J. SUSSMAN

RICHARD KELSEY, WILLIAM CLINGER, AND JONATHAN REES
(Editors, Revised[5] Report on the Algorithmic Language Scheme)

MICHAEL SPERBER, R. KENT DYBVIG, MATTHEW FLATT, AND ANTON VAN STRAATEN
(Editors, Revised[6] Report on the Algorithmic Language Scheme)

Dedicated to the memory of John McCarthy and Daniel Weinreb

June 28, 2017

SUMMARY

CONTENTS

INTRODUCTION

Background

Acknowledgments

DESCRIPTION OF THE LANGUAGE

1. Overview of Scheme

1.1. Semantics

1.2. Syntax

1.3. Notation and terminology

1.3.1. Base and optional features

1.3.2. Error situations and unspecified behavior

1.3.3. Entry format

1.3.4. Evaluation examples

1.3.5. Naming conventions

2. Lexical conventions

2.1. Identifiers

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

```
...                          +
+soup+                       <=?
->string                     a34kTMNs
lambda                       list->vector
q                            V17a
|two words|                  |two\x20;words|
the-word-recursion-has-many-meanings
```

```
#!fold-case
#!no-fold-case
```

These directives can appear anywhere comments are permitted (see section 2.2) but must be followed by a delimiter. They are treated as comments, except that they affect the reading of subsequent data from the same port. The `#!fold-case` directive causes subsequent identifiers and character names to be case-folded as if by `string-foldcase` (see section **??**). It has no effect on character literals. The `#!no-fold-case` directive causes a return to the default, non-folding behavior.

2.2. Whitespace and comments

```
#|
   The FACT procedure computes the factorial
   of a non-negative integer.
|#
(define fact
  (lambda (n)
    (if (= n 0)
        #;(= n 1)
        1        ;Base case: return 1
        (* n (fact (- n 1))))))
```

2.3. Datum labels

```
(let ((x (list 'a 'b 'c)))
  (set-cdr! (cddr x) x)
  x)                         ⟹ #0=(a b c . #0#)

#1=(begin (display #\x) #1#)
                             ⟹ error
```

3. Basic concepts

3.1. Variables, syntactic keywords, and regions

3.2. Disjointness of types

```
boolean?          bytevector?
char?             eof-object?
null?             number?
pair?             port?
procedure?        string?
symbol?           vector?
```

3.3. External representations

3.4. Storage model

Rationale: In many systems it is desirable for constants (i.e. the values of literal expressions) to reside in read-only memory. Making it an error to alter constants permits this implementation strategy, while not requiring other systems to distinguish between mutable and immutable objects.

3.5. Proper tail recursion

- The last expression within the body of a lambda expression, shown as ⟨tail expression⟩ below, occurs in a tail context. The same is true of all the bodies of `case-lambda` expressions.

  (lambda ⟨formals⟩
    ⟨definition⟩* ⟨expression⟩* ⟨tail expression⟩)

  (case-lambda (⟨formals⟩ ⟨tail body⟩)*)

  (if ⟨expression⟩ ⟨tail expression⟩ ⟨tail expression⟩)
  (if ⟨expression⟩ ⟨tail expression⟩)

  (cond ⟨cond clause⟩$^+$)

```
(cond ⟨cond clause⟩* (else ⟨tail sequence⟩))

(case ⟨expression⟩
  ⟨case clause⟩+)
(case ⟨expression⟩
  ⟨case clause⟩*
  (else ⟨tail sequence⟩))

(and ⟨expression⟩* ⟨tail expression⟩)
(or ⟨expression⟩* ⟨tail expression⟩)

(when ⟨test⟩ ⟨tail sequence⟩)
(unless ⟨test⟩ ⟨tail sequence⟩)

(let (⟨binding spec⟩*) ⟨tail body⟩)
(let ⟨variable⟩ (⟨binding spec⟩*) ⟨tail body⟩)
(let* (⟨binding spec⟩*) ⟨tail body⟩)
(letrec (⟨binding spec⟩*) ⟨tail body⟩)
(letrec* (⟨binding spec⟩*) ⟨tail body⟩)
(let-values (⟨mv binding spec⟩*) ⟨tail body⟩)
(let*-values (⟨mv binding spec⟩*) ⟨tail body⟩)

(let-syntax (⟨syntax spec⟩*) ⟨tail body⟩)
(letrec-syntax (⟨syntax spec⟩*) ⟨tail body⟩)

(begin ⟨tail sequence⟩)

(do (⟨iteration spec⟩*)
    (⟨test⟩ ⟨tail sequence⟩)
  ⟨expression⟩*)
```

where

```
⟨cond clause⟩ ⟶ (⟨test⟩ ⟨tail sequence⟩)
⟨case clause⟩ ⟶ ((⟨datum⟩*) ⟨tail sequence⟩)

⟨tail body⟩ ⟶ ⟨definition⟩* ⟨tail sequence⟩
⟨tail sequence⟩ ⟶ ⟨expression⟩* ⟨tail expression⟩
```

```
(lambda ()
  (if (g)
      (let ((x (h)))
        x)
      (and (g) (f))))
```

4. Expressions

4.1. Primitive expression types

4.1.1. Variable references

4.1.2. Literal expressions

4.1.3. Procedure calls

4.1.4. Procedures

4.1.5. Conditionals

4.1.6. Assignments

`(set! ⟨variable⟩ ⟨expression⟩)` syntax

Semantics: ⟨Expression⟩ is evaluated, and the resulting value is stored in the location to which ⟨variable⟩ is bound. It is an error if ⟨variable⟩ is not bound either in some region enclosing the `set!` expression or else globally. The result of the `set!` expression is unspecified.

```
(define x 2)
(+ x 1)          ⟹  3
(set! x 4)       ⟹  unspecified
(+ x 1)          ⟹  5
```

4.1.7. Inclusion

`(include ⟨string₁⟩ ⟨string₂⟩ ...)` syntax
`(include-ci ⟨string₁⟩ ⟨string₂⟩ ...)` syntax

Semantics: Both `include` and `include-ci` take one or more filenames expressed as string literals, apply an implementation-specific algorithm to find corresponding files, read the contents of the files in the specified order as if by repeated applications of `read`, and effectively replace the `include` or `include-ci` expression with a `begin` expression containing what was read from the files. The difference between the two is that `include-ci` reads each file as if it began with the `#!fold-case` directive, while `include` does not.

Note: Implementations are encouraged to search for files in the directory which contains the including file, and to provide a way for users to specify other directories to search.

4.2. Derived expression types

The constructs in this section are hygienic, as discussed in section ??. For reference purposes, section 7.3 gives syntax definitions that will convert most of the constructs described in this section into the primitive constructs described in the previous section.

4.2.1. Conditionals

`(cond` ⟨clause$_1$⟩ ⟨clause$_2$⟩ `...)` syntax
`else` auxiliary syntax
`=>` auxiliary syntax

Syntax: ⟨Clauses⟩ take one of two forms, either

`(`⟨test⟩ ⟨expression$_1$⟩ `...)`

where ⟨test⟩ is any expression, or

`(`⟨test⟩ `=>` ⟨expression⟩`)`

The last ⟨clause⟩ can be an "else clause," which has the form

`(else` ⟨expression$_1$⟩ ⟨expression$_2$⟩ `...)`.

Semantics: A `cond` expression is evaluated by evaluating the ⟨test⟩ expressions of successive ⟨clause⟩s in order until one of them evaluates to a true value (see section **??**). When a ⟨test⟩ evaluates to a true value, the remaining ⟨expression⟩s in its ⟨clause⟩ are evaluated in order, and the results of the last ⟨expression⟩ in the ⟨clause⟩ are returned as the results of the entire `cond` expression.

If the selected ⟨clause⟩ contains only the ⟨test⟩ and no ⟨expression⟩s, then the value of the ⟨test⟩ is returned as the result. If the selected ⟨clause⟩ uses the `=>` alternate form, then the ⟨expression⟩ is evaluated. It is an error if its value is not a procedure that accepts one argument. This procedure is then called on the value of the ⟨test⟩ and the values returned by this procedure are returned by the `cond` expression.

If all ⟨test⟩s evaluate to `#f`, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its ⟨expression⟩s are evaluated in order, and the values of the last one are returned.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))          ⟹  greater
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))            ⟹  equal
(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f))                ⟹  2
```

5. Program structure

5.1. Programs

5.2. Import declarations

5.3. Variable definitions

6. Standard procedures

This chapter describes Scheme's built-in procedures.

6.1. Equivalence predicates

A *predicate* is a procedure that always returns a boolean value (`#t` or `#f`). An *equivalence predicate* is the computational analogue of a mathematical equivalence relation; it is symmetric, reflexive, and transitive. Of the equivalence predicates described in this section, `eq?` is the finest or most discriminating, `equal?` is the coarsest, and `eqv?` is slightly less discriminating than `eq?`.

7. Formal syntax and semantics

This chapter provides formal descriptions of what has already been described informally in previous chapters of this report.

7.1. Formal syntax

This section provides a formal syntax for Scheme written in an extended BNF.

7.1.1. Lexical structure

This section describes how individual tokens (identifiers, numbers, etc.) are formed from sequences of characters. The following sections describe how expressions and programs are formed from sequences of tokens.

7.2. Formal semantics

7.3. Derived expression types

This section gives syntax definitions for the derived expression types in terms of the primitive expression types (literal, variable, call, `lambda`, `if`, and `set!`), except for `quasiquote`.

Appendix A. Standard Libraries

This section lists the exports provided by the standard libraries. The libraries are factored so as to separate features which might not be supported by all implementations, or which might be expensive to load.

The `scheme` library prefix is used for all standard libraries, and is reserved for use by future standards.

Base Library

The `(scheme base)` library exports many of the procedures and syntax bindings that are traditionally associated with Scheme. The division between the base library and the other standard libraries is based on use, not on construction. In particular, some facilities that are typically implemented as primitives by a compiler or the run-time system rather than in terms of other standard procedures or syntax are not part of the base library, but are defined in separate libraries. By the same token, some exports of the base library are implementable in terms of other exports. They are redundant in the strict sense of the word, but they capture common patterns of usage, and are therefore provided as convenient abbreviations.

`*`	`+`
`-`	`...`
`/`	`<`
`<=`	`=`
`=>`	`>`
`>=`	`_`
`abs`	`and`
`append`	`apply`
`assoc`	`assq`
`assv`	`begin`
`binary-port?`	`boolean=?`
`boolean?`	`bytevector`
`bytevector-append`	`bytevector-copy`

Case-Lambda Library

The `(scheme case-lambda)` library exports the `case-lambda` syntax.

`case-lambda`

Char Library

The `(scheme char)` library provides the procedures for dealing with characters that involve potentially large tables when supporting all of Unicode.

`char-alphabetic?`	`char-ci<=?`
`char-ci<?`	`char-ci=?`

Complex Library

The `(scheme complex)` library exports procedures which are typically only useful with non-real numbers.

`angle`	`imag-part`

Appendix B. Standard Feature Identifiers

An implementation may provide any or all of the feature identifiers listed below for use by `cond-expand` and `features`, but must not provide a feature identifier if it does not provide the corresponding feature.

`r7rs`

All R^7RS Scheme implementations have this feature.

`exact-closed`

All algebraic operations except `/` produce exact values given exact inputs.

`exact-complex`

Exact complex numbers are provided.

LANGUAGE CHANGES

Incompatibilities with R^5RS

This section enumerates the incompatibilities between this report and the "Revised^5 report" [20].

This list is not authoritative, but is believed to be correct and complete.

ADDITIONAL MATERIAL

The Scheme community website at `http://schemers.org` contains additional resources for learning and programming, job and event postings, and Scheme user group information.

EXAMPLE

The procedure `integrate-system` integrates the system

$$y'_k = f_k(y_1, y_2, \ldots, y_n),\ k = 1, \ldots, n$$

of differential equations with the method of Runge-Kutta.

Infinite streams are implemented as pairs whose car holds the first element of the stream and whose cdr holds a promise to deliver the rest of the stream.

```
(define head car)
(define (tail stream)
  (force (cdr stream)))
```

The following illustrates the use of `integrate-system` in integrating the system

$$C\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

which models a damped oscillator.

```
(define (damped-oscillator R L C)
  (lambda (state)
    (let ((Vc (vector-ref state 0))
          (Il (vector-ref state 1)))
      (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
              (/ Vc L)))))

(define the-states
  (integrate-system
     (damped-oscillator 10000 1000 .001)
     '#(1 0)
     .01))
```

REFERENCES

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs, second edition.* MIT Press, Cambridge, 1996.

[2] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 86–95.

[3] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. `http://www.ietf.org/rfc/rfc2119.txt`, 1997.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

The principal entry for each term, procedure, or keyword is listed first, separated from the other entries by a semicolon.