WIKIPEDIA

Scheme (programming language)

Scheme is a minimalist dialect of the Lisp family of programming languages. Scheme consists of a small standard core with powerful tools for language extension.[1]

Scheme was created during the 1970s at the MIT AI Lab and released by its developers, Guy L. Steele and Gerald Jay Sussman, via a series of memos now known as the Lambda Papers. It was the first dialect of Lisp to choose lexical scope and the first to require implementations to perform tail-call optimization, giving stronger support for functional programming and associated techniques such as recursive algorithms. It was also one of the first programming languages to support first-class continuations. It had a significant influence on the effort that led to the development of Common Lisp.[2]

The Scheme language is standardized in the official IEEE standard[3] and a *de facto* standard called the *Revised^n Report on the Algorithmic Language Scheme* (R*n*RS). The most widely implemented standard is R5RS (1998);[4] a new standard, R6RS,[5] was ratified in 2007.[6] Scheme has a diverse user base due to its compactness and elegance, but its minimalist philosophy has also caused wide divergence between practical implementations, so much that the Scheme Steering Committee calls it "the world's most unportable programming language" and "a *family* of dialects" rather than a single language.[7]

Scheme

λ

Paradigms	Multi-paradigm: functional, imperative, meta
Family	Lisp
Designed by	Guy L. Steele and Gerald Jay Sussman
First appeared	1975
Stable release	R7RS / 2013
Typing discipline	Dynamic, latent, strong
Scope	Lexical
Filename extensions	.scm, .ss
Website	www.scheme-reports.org (http://www.scheme-reports.org/)

Major implementations

Many
(see Scheme implementations)

Influenced by

ALGOL, Lisp, MDL

Influenced

Clojure, Common Lisp, Dylan, EuLisp, Haskell, Hop, JavaScript, Julia, Lua, MultiLisp, R, Racket, Ruby, Rust, S, Scala, T

Scheme at Wikibooks

Contents

History

Origins

Scheme started in the 1970s as an attempt to understand Carl Hewitt's Actor model, for which purpose Steele and Sussman wrote a "tiny Lisp interpreter" using Maclisp and then "added mechanisms for creating actors and sending messages".[8] Scheme was originally called "Schemer", in the tradition of other Lisp-derived languages such as Planner or *Conniver*. The current name resulted from the authors' use of the ITS operating system, which limited filenames to two components of at most six characters each. Currently, "Schemer" is commonly used to refer to a Scheme programmer.

R6RS

A new language standardization process began at the 2003 Scheme workshop, with the goal of producing an R6RS standard in 2006. This process broke with the earlier R*n*RS approach of unanimity.

R6RS features a standard module system, allowing a split between the core language and libraries. A number of drafts of the R6RS specification were released, the final version being R5.97RS. A successful vote resulted in the ratification of the new standard, announced on August 28, 2007.[5]

Currently the newest releases of various Scheme implementations[9] support the R6RS standard. There is a portable reference implementation of the proposed implicitly phased libraries for R6RS, called psyntax, which loads and bootstraps itself properly on various older Scheme implementations.[10]

A feature of R6RS is the record-type descriptor (RTD). When an RTD is created and used, the record type representation can show the memory layout. It also calculated object field bit mask and mutable Scheme object field bit masks, and helped the garbage collector know what to do with the fields without traversing the whole fields list that are saved in the RTD. RTD allows users to expand the basic RTD to create a new record system.[11]

R6RS introduces numerous significant changes to the language.[12] The source code is now specified in Unicode, and a large subset of Unicode characters may now appear in Scheme symbols and identifiers, and there are other minor changes to the lexical rules. Character data is also now specified in Unicode. Many standard procedures have been moved to the new standard libraries, which themselves form a large expansion of the standard, containing procedures and syntactic forms that were formerly not part of the standard. A new module system has been introduced, and systems for exception handling are now standardized. Syntax-rules has been replaced with a more expressive syntactic abstraction facility (syntax-case) which allows the use of all of Scheme at macro expansion time. Compliant implementations are now *required* to support Scheme's full numeric tower, and the semantics of numbers have been expanded, mainly in the direction of support for the IEEE 754 standard for floating point numerical representation.

R7RS

The R6RS standard has caused controversy because it is seen to have departed from the minimalist philosophy.[13][14] In August 2009, the Scheme Steering Committee, which oversees the standardization process, announced its intention to recommend splitting Scheme into two languages: a large modern programming language for programmers; and a small version, a subset of the large version retaining the minimalism praised by educators and casual implementors.[7] Two working groups were created to work on these two new versions of Scheme. The Scheme Reports Process site has links to the working groups' charters, public discussions and issue tracking system.

The ninth draft of R7RS (small language) was made available on April 15, 2013.[15] A vote ratifying this draft closed on May 20, 2013,[16] and the final report has been available since August 6, 2013, describing "the 'small' language of that effort: therefore it cannot be considered in isolation as the successor to R6RS".[17]

Distinguishing features

Scheme is primarily a functional programming language. It shares many characteristics with other members of the Lisp programming language family. Scheme's very simple syntax is based on s-expressions, parenthesized lists in which a prefix operator is followed by its arguments. Scheme programs thus consist of sequences of nested lists. Lists are also the main data structure in Scheme, leading to a close equivalence between source code and data formats (homoiconicity). Scheme programs can easily create and evaluate pieces of Scheme code dynamically.

The reliance on lists as data structures is shared by all Lisp dialects. Scheme inherits a rich set of list-processing primitives such as `cons`, `car` and `cdr` from its Lisp progenitors. Scheme uses strictly but dynamically typed variables and supports first class procedures. Thus, procedures can be assigned as values to variables or passed as arguments to procedures.

This section concentrates mainly on innovative features of the language, including those features that distinguish Scheme from other Lisps. Unless stated otherwise, descriptions of features relate to the R5RS standard.

In examples provided in this section, the notation "===> result" is used to indicate the result of evaluating the expression on the immediately preceding line. This is the same convention used in R5RS.

Fundamental design features

This subsection describes those features of Scheme that have distinguished it from other programming languages from its earliest days. These are the aspects of Scheme that most strongly influence any product of the Scheme language, and they are the aspects that all versions of the Scheme programming language, from 1973 onward, share.

Minimalism

Scheme is a very simple language, much easier to implement than many other languages of comparable expressive power.[18] This ease is attributable to the use of lambda calculus to derive much of the syntax of the language from more primitive forms. For instance of the 23 s-expression-based syntactic constructs defined in the R5RS Scheme standard, 14 are classed as derived or library forms, which can be written as macros involving more fundamental forms, principally lambda. As R5RS says (R5RS sec. 3.1): "The most fundamental of the variable binding constructs is the lambda expression, because all other variable binding constructs can be explained in terms of lambda expressions."[4]

> **Fundamental forms**: define, lambda, quote, if, define-syntax, let-syntax, letrec-syntax, syntax-rules, set!
> **Derived forms**: do, let, let*, letrec, cond, case, and, or, begin, named let, delay, unquote, unquote-splicing, quasiquote

Example: a macro to implement `let` as an expression using `lambda` to perform the variable bindings.

```
(define-syntax let
  (syntax-rules ()
    ((let ((var expr) ...) body ...)
      ((lambda (var ...) body ...) expr ...))))
```

Thus using `let` as defined above a Scheme implementation would rewrite "`(let ((a 1)(b 2)) (+ b a))`" as "`((lambda (a b) (+ b a)) 1 2)`", which reduces implementation's task to that of coding procedure instantiations.

In 1998, Sussman and Steele remarked that the minimalism of Scheme was not a conscious design goal, but rather the unintended outcome of the design process. "We were actually trying to build something complicated and discovered, serendipitously, that we had accidentally designed something that met all our goals but was much simpler than we had intended....we realized that the lambda calculus—a small, simple formalism—could serve as the core of a powerful and expressive programming language."[8]

Lexical scope

Like most modern programming languages and unlike earlier Lisps such as Maclisp, Scheme is lexically scoped: all possible variable bindings in a program unit can be analyzed by reading the text of the program unit without consideration of the contexts in which it may be called. This contrasts with dynamic scoping which was characteristic of early Lisp dialects, because of the processing costs associated with the primitive textual substitution methods used to implement lexical scoping algorithms in compilers and interpreters of the day. In those Lisps, it was perfectly possible for a reference to a free variable inside a procedure to refer to quite distinct bindings external to the procedure, depending on the context of the call.

The impetus to incorporate lexical scoping, which was an unusual scoping model in the early 1970s, into their new version of Lisp, came from Sussman's studies of ALGOL. He suggested that ALGOL-like lexical scoping mechanisms would help to realize their initial goal of implementing Hewitt's Actor model in Lisp.[8]

The key insights on how to introduce lexical scoping into a Lisp dialect were popularized in Sussman and Steele's 1975 Lambda Paper, "Scheme: An Interpreter for Extended Lambda Calculus",[19] where they adopted the concept of the lexical closure (on page 21), which had been described in an AI Memo in 1970 by Joel Moses, who attributed the idea to Peter J. Landin.[20]

Lambda calculus

Alonzo Church's mathematical notation, the lambda calculus, has inspired Lisp's use of "lambda" as a keyword for introducing a procedure, as well as influencing the development of functional programming techniques involving the use of higher-order functions in Lisp. But early Lisps were not suitable expressions of the lambda calculus because of their treatment of free variables.[8]

A formal lambda system has axioms and a complete calculation rule. It is helpful for the analysis using mathematical logic and tools. In this system, calculation can be seen as a directional deduction. The syntax of lambda calculus follows the recursive expressions from x, y, z, ...,parentheses, spaces, the period and the symbol λ.[21] The function of lambda calculation includes: First, serve as a starting point of powerful mathematical logic. Second, it can reduce the requirement of programmers to consider the implementation details, because it can be used to imitate machine evaluation. Finally, the lambda calculation created a substantial meta-theory.[22]

The introduction of lexical scope resolved the problem by making an equivalence between some forms of lambda notation and their practical expression in a working programming language. Sussman and Steele showed that the new language could be used to elegantly derive all the imperative and declarative semantics of other programming languages including ALGOL and Fortran, and the dynamic scope of other Lisps, by using lambda expressions not as simple procedure instantiations but as "control structures and environment modifiers".[23] They introduced continuation-passing style along with their first description of Scheme in the first of the Lambda Papers, and in subsequent papers, they proceeded to demonstrate the raw power of this practical use of lambda calculus.

Block structure

Scheme inherits its block structure from earlier block structured languages, particularly ALGOL. In Scheme, blocks are implemented by three *binding constructs*: `let`, `let*` and `letrec`. For instance, the following construct creates a block in which a symbol called `var` is bound to the number 10:

```
(define var "goose")
;; Any reference to var here will be bound to "goose"
(let ((var 10))
  ;; statements go here.  Any reference to var here will be bound to 10.
  )
;; Any reference to var here will be bound to "goose"
```

Blocks can be nested to create arbitrarily complex block structures according to the need of the programmer. The use of block structuring to create local bindings alleviates the risk of namespace collision that can otherwise occur.

One variant of `let`, `let*`, permits bindings to refer to variables defined earlier in the same construct, thus:

```
(let* ((var1 10)
       (var2 (+ var1 12)))
  ;; But the definition of var1 could not refer to var2
  )
```

The other variant, `letrec`, is designed to enable mutually recursive procedures to be bound to one another.

```
;; Calculation of Hofstadter's male and female sequences as a list of pairs

(define (hofstadter-male-female n)
  (letrec ((female (lambda (n)
                     (if (= n 0)
                     1
                     (- n (male (female (- n 1)))))))
       (male (lambda (n)
                   (if (= n 0)
                     0
                     (- n (female (male (- n 1))))))))
    (let loop ((i 0))
      (if (> i n)
      '()
      (cons (cons (female i)
                  (male i))
        (loop (+ i 1)))))))

(hofstadter-male-female 8)

===> ((1 . 0) (1 . 0) (2 . 1) (2 . 2) (3 . 2) (3 . 3) (4 . 4) (5 . 4) (5 . 5))
```

(See Hofstadter's male and female sequences for the definitions used in this example.)

All procedures bound in a single `letrec` may refer to one another by name, as well as to values of variables defined earlier in the same `letrec`, but they may not refer to *values* defined later in the same `letrec`.

A variant of `let`, the "named let" form, has an identifier after the `let` keyword. This binds the let variables to the argument of a procedure whose name is the given identifier and whose body is the body of the let form. The body may be repeated as desired by calling the procedure. The named let is widely used to implement iteration.

Example: a simple counter

```
(let loop ((n 1))
  (if (> n 10)
      '()
      (cons n
        (loop (+ n 1)))))

===> (1 2 3 4 5 6 7 8 9 10)
```

Like any procedure in Scheme, the procedure created in the named let is a first class object.

Proper tail recursion

Scheme has an iteration construct, `do`, but it is more idiomatic in Scheme to use tail recursion to express iteration. Standard-conforming Scheme implementations are required to optimize tail calls so as to support an unbounded number of active tail calls (R5RS sec. 3.5)[4]—a property the Scheme report describes as *proper tail recursion*—making it safe for Scheme programmers to write iterative algorithms using recursive structures, which are sometimes more intuitive. Tail recursive procedures and the *named* `let` form provide support for iteration using tail recursion.

```
;; Building a list of squares from 0 to 9:
;; Note: loop is simply an arbitrary symbol used as a label. Any symbol will do.

(define (list-of-squares n)
  (let loop ((i n) (res '()))
    (if (< i 0)
        res
        (loop (- i 1) (cons (* i i) res)))))

(list-of-squares 9)
===> (0 1 4 9 16 25 36 49 64 81)
```

First-class continuations

Continuations in Scheme are first-class objects. Scheme provides the procedure `call-with-current-continuation` (also known as `call/cc`) to capture the current continuation by packing it up as an escape procedure bound to a formal argument in a procedure provided by the programmer. (R5RS sec. 6.4)[4] First-class continuations enable the programmer to create non-local control constructs such as iterators, coroutines, and backtracking.

Continuations can be used to emulate the behavior of return statements in imperative programming languages. The following function `find-first`, given function `func` and list `lst`, returns the first element `x` in `lst` such that `(func x)` returns true.

```
(define (find-first func lst)
  (call-with-current-continuation
   (lambda (return-immediately)
     (for-each (lambda (x)
         (if (func x)
             (return-immediately x)))
           lst)
     #f)))

(find-first integer? '(1/2 3/4 5.6 7 8/9 10 11))
===> 7
(find-first zero? '(1 2 3 4))
===> #f
```

The following example, a traditional programmer's puzzle, shows that Scheme can handle continuations as first-class objects, binding them to variables and passing them as arguments to procedures.

```
(let* ((yin
         ((lambda (cc) (display "@") cc) (call-with-current-continuation (lambda (c) c))))
       (yang
         ((lambda (cc) (display "*") cc) (call-with-current-continuation (lambda (c) c)))))
    (yin yang))
```

When executed this code displays a counting sequence: `@*@**@***@****@*****@******@*******@********...`

Shared namespace for procedures and variables

In contrast to Common Lisp, all data and procedures in Scheme share a common namespace, whereas in Common Lisp functions and data have separate namespaces making it possible for a function and a variable to have the same name, and requiring special notation for referring to a function as a value. This is sometimes known as the "Lisp-1 vs. Lisp-2" distinction, referring to the unified namespace of Scheme and the separate namespaces of Common Lisp.[24]

In Scheme, the same primitives that are used to manipulate and bind data can be used to bind procedures. There is no equivalent of Common Lisp's `defun` and `#'` primitives.

```
;; Variable bound to a number:
(define f 10)
f
===> 10
;; Mutation (altering the bound value)
(set! f (+ f f 6))
f
===> 26
;; Assigning a procedure to the same variable:
(set! f (lambda (n) (+ n 12)))
(f 6)
===> 18
;; Assigning the result of an expression to the same variable:
(set! f (f 1))
f
===> 13
;; functional programming:
(apply + '(1 2 3 4 5 6))
===> 21
(set! f (lambda (n) (+ n 100)))
(map f '(1 2 3))
===> (101 102 103)
```

Implementation standards

This subsection documents design decisions that have been taken over the years which have given Scheme a particular character, but are not the direct outcomes of the original design.

Numerical tower

Scheme specifies a comparatively full set of numerical datatypes including complex and rational types, which is known in Scheme as the numerical tower (R5RS sec. 6.2[4]). The standard treats these as abstractions, and does not commit the implementor to any particular internal representations.

Numbers may have the quality of exactness. An exact number can only be produced by a sequence of exact operations involving other exact numbers—inexactness is thus contagious. The standard specifies that any two implementations must produce equivalent results for all operations resulting in exact numbers.

The R5RS standard specifies procedures `exact->inexact` and `inexact->exact` which can be used to change the exactness of a number. `inexact->exact` produces "the exact number that is numerically closest to the argument". `exact->inexact` produces "the inexact number that is numerically closest to the argument". The R6RS standard omits these procedures from the main report, but specifies them as R5RS compatibility procedures in the standard library (rnrs r5rs (6)).

In the R5RS standard, Scheme implementations are not required to implement the whole numerical tower, but they must implement "a coherent subset consistent with both the purposes of the implementation and the spirit of the Scheme language" (R5RS sec. 6.2.3).[4] The new R6RS standard does require implementation of the whole tower, and "exact integer objects and exact rational number objects of practically unlimited size and precision, and to implement certain procedures...so they always return exact results when given exact arguments" (R6RS sec. 3.4, sec. 11.7.1).[5]

Example 1: exact arithmetic in an implementation that supports exact rational complex numbers.

```
;; Sum of three rational real numbers and two rational complex numbers
(define x (+ 1/3 1/4 -1/5 -1/3i 405/50+2/3i))
x
===> 509/60+1/3i
;; Check for exactness.
(exact? x)
===> #t
```

Example 2: Same arithmetic in an implementation that supports neither exact rational numbers nor complex numbers but does accept real numbers in rational notation.

```
;; Sum of four rational real numbers
(define xr (+ 1/3 1/4 -1/5 405/50))
;; Sum of two rational real numbers
(define xi (+ -1/3 2/3))
xr
===> 8.48333333333333
xi
===> 0.333333333333333
;; Check for exactness.
(exact? xr)
===> #f
(exact? xi)
===> #f
```

Both implementations conform to the R5RS standard but the second does not conform to R6RS because it does not implement the full numerical tower.

Delayed evaluation

Scheme supports delayed evaluation through the `delay` form and the procedure `force`.

```
(define a 10)
(define eval-aplus2 (delay (+ a 2)))
(set! a 20)
(force eval-aplus2)
===> 22
(define eval-aplus50 (delay (+ a 50)))
(let ((a 8))
  (force eval-aplus50))
===> 70
(set! a 100)
(force eval-aplus2)
===> 22
```

The lexical context of the original definition of the promise is preserved, and its value is also preserved after the first use of `force`. The promise is only ever evaluated once.

These primitives, which produce or handle values known as promises, can be used to implement advanced lazy evaluation constructs such as streams.[25]

In the R6RS standard, these are no longer primitives, but instead, are provided as part of the R5RS compatibility library (rnrs r5rs (6)).

In R5RS, a suggested implementation of `delay` and `force` is given, implementing the promise as a procedure with no arguments (a thunk) and using memoization to ensure that it is only ever evaluated once, irrespective of the number of times `force` is called (R5RS sec. 6.4).[4]

SRFI 41 enables the expression of both finite and infinite sequences with extraordinary economy. For example, this is a definition of the fibonacci sequence using the functions defined in SRFI 41:[25]

```
;; Define the Fibonacci sequence:
(define fibs
  (stream-cons 0
    (stream-cons 1
      (stream-map +
        fibs
        (stream-cdr fibs)))))
;; Compute the hundredth number in the sequence:
(stream-ref fibs 99)
===>  218922995834555169026
```

Order of evaluation of procedure arguments

Most Lisps specify an order of evaluation for procedure arguments. Scheme does not. Order of evaluation—including the order in which the expression in the operator position is evaluated—may be chosen by an implementation on a call-by-call basis, and the only constraint is that "the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with some sequential order of evaluation." (R5RS sec. 4.1.3)[4]

```
(let ((ev (lambda(n) (display "Evaluating ")
                     (display (if (procedure? n) "procedure" n))
                     (newline) n)))
  ((ev +) (ev 1) (ev 2)))
===> 3
```

ev is a procedure that describes the argument passed to it, then returns the value of the argument. In contrast with other Lisps, the appearance of an expression in the operator position (the first item) of a Scheme expression is quite legal, as long as the result of the expression in the operator position is a procedure.

In calling the procedure "+" to add 1 and 2, the expressions (ev +), (ev 1) and (ev 2) may be evaluated in any order, as long as the effect is not as if they were evaluated in parallel. Thus the following three lines may be displayed in any order by standard Scheme when the above example code is executed, although the text of one line may not be interleaved with another because that would violate the sequential evaluation constraint.

```
Evaluating 1
Evaluating 2
Evaluating procedure
```

Hygienic macros

In the R5RS standard and also in later reports, the syntax of Scheme can easily be extended via the macro system. The R5RS standard introduced a powerful hygienic macro system that allows the programmer to add new syntactic constructs to the language using a simple pattern matching sublanguage (R5RS sec 4.3).[4] Prior to this, the hygienic macro system had been relegated to an appendix of the R4RS standard, as a "high level" system alongside a "low level" macro system, both of which were treated as extensions to Scheme rather than an essential part of the language.[26]

Implementations of the hygienic macro system, also called `syntax-rules`, are required to respect the lexical scoping of the rest of the language. This is assured by special naming and scoping rules for macro expansion and avoids common programming errors that can occur in the macro systems of other programming languages. R6RS specifies a more sophisticated transformation system, `syntax-case`, which has been available as a language extension to R5RS Scheme for some time.

```
;; Define a macro to implement a variant of "if" with a multi-expression
;; true branch and no false branch.
(define-syntax when
  (syntax-rules ()
    ((when pred exp exps ...)
      (if pred (begin exp exps ...)))))
```

Invocations of macros and procedures bear a close resemblance—both are s-expressions—but they are treated differently. When the compiler encounters an s-expression in the program, it first checks to see if the symbol is defined as a syntactic keyword within the current lexical scope. If so, it then attempts to expand the macro, treating the items in the tail of the s-expression as arguments without compiling code to evaluate them, and this process is repeated recursively until no macro invocations remain. If it is not a syntactic keyword, the compiler compiles code to evaluate the arguments in the tail of the s-expression and then to evaluate the variable represented by the symbol at the head of the s-expression and call it as a procedure with the evaluated tail expressions passed as actual arguments to it.

Most Scheme implementations also provide additional macro systems. Among popular ones are syntactic closures, explicit renaming macros and `define-macro`, a non-hygienic macro system similar to `defmacro` system provided in Common Lisp.

The inability to specify whether or not a macro is hygienic is one of the shortcomings of the macro system. Alternative models for expansion such as scope sets provide a potential solution.[27]

Environments and eval

Prior to R5RS, Scheme had no standard equivalent of the `eval` procedure which is ubiquitous in other Lisps, although the first Lambda Paper had described `evaluate` as "similar to the LISP function EVAL"[19] and the first Revised Report in 1978 replaced this with `enclose`, which took two arguments. The second, third and fourth revised reports omitted any equivalent of `eval`.

The reason for this confusion is that in Scheme with its lexical scoping the result of evaluating an expression depends on where it is evaluated. For instance, it is not clear whether the result of evaluating the following expression should be 5 or 6:[28]

```
(let ((name '+))
  (let ((+ *))
    (evaluate (list name 2 3))))
```

If it is evaluated in the outer environment, where `name` is defined, the result is the sum of the operands. If it is evaluated in the inner environment, where the symbol "+" has been bound to the value of the procedure "*", the result is the product of the two operands.

R5RS resolves this confusion by specifying three procedures that return environments and providing a procedure `eval` that takes an s-expression and an environment and evaluates the expression in the environment provided. (R5RS sec. 6.5)[4] R6RS extends this by providing a procedure called environment by which the programmer can specify exactly which objects to import into the evaluation environment.

With modern scheme (usually compatible with R5RS) to evaluate this expression, you need to define function evaluate which can look like this:

```
(define (evaluate expr)
   (eval expr (interaction-environment)))
```

interaction-environment is global environment from your interpreter. That's why + still point to plus operation.

Treatment of non-boolean values in boolean expressions

In most dialects of Lisp including Common Lisp, by convention the value NIL evaluates to the value false in a boolean expression. In Scheme, since the IEEE standard in 1991,[3] all values except #f, including NIL's equivalent in Scheme which is written as '(), evaluate to the value true in a boolean expression. (R5RS sec. 6.3.1)[4]

Where the constant representing the boolean value of true is T in most Lisps, in Scheme it is #t.

Disjointness of primitive datatypes

In Scheme the primitive datatypes are disjoint. Only one of the following predicates can be true of any Scheme object: boolean?, pair?, symbol?, number?, char?, string?, vector?, port?, procedure?. (R5RS sec 3.2)[4]

Within the numerical datatype, by contrast, the numerical values overlap. For example, an integer value satisfies all of the integer?, rational?, real?, complex? and number? predicates at the same time. (R5RS sec 6.2)[4]

Equivalence predicates

Scheme has three different types of equivalence between arbitrary objects denoted by three different *equivalence predicates*, relational operators for testing equality, eq?, eqv? and equal?:

- eq? evaluates to #f unless its parameters represent the same data object in memory;
- eqv? is generally the same as eq? but treats primitive objects (e.g. characters and numbers) specially so that numbers that represent the same value are eqv? even if they do not refer to the same object;
- equal? compares data structures such as lists, vectors and strings to determine if they have congruent structure and eqv? contents.(R5RS sec. 6.1)[4]

Type dependent equivalence operations also exist in Scheme: string=? and string-ci=? compare two strings (the latter performs a case-independent comparison); char=? and char-ci=? compare characters; = compares numbers.[4]

Comments

Up to the R5RS standard, the standard comment in Scheme was a semicolon, which makes the rest of the line invisible to Scheme. Numerous implementations have supported alternative conventions permitting comments to extend for more than a single line, and the R6RS standard permits two of them: an entire s-expression may be turned into a comment (or "commented out") by preceding it with `#;` (introduced in SRFI 62[29]) and a multiline comment or "block comment" may be produced by surrounding text with `#|` and `|#`.

Input/output

Scheme's input and output is based on the *port* datatype. (R5RS sec 6.6)[4] R5RS defines two default ports, accessible with the procedures `current-input-port` and `current-output-port`, which correspond to the Unix notions of standard input and standard output. Most implementations also provide `current-error-port`. Redirection of input and standard output is supported in the standard, by standard procedures such as `with-input-from-file` and `with-output-to-file`. Most implementations provide string ports with similar redirection capabilities, enabling many normal input-output operations to be performed on string buffers instead of files, using procedures described in SRFI 6.[30] The R6RS standard specifies much more sophisticated and capable port procedures and many new types of port.

The following examples are written in strict R5RS Scheme.

Example 1: With output defaulting to (current-output-port):

```
(let ((hello0 (lambda() (display "Hello world") (newline))))
  (hello0))
```

Example 2: As 1, but using optional port argument to output procedures

```
(let ((hello1 (lambda (p) (display "Hello world" p) (newline p))))
  (hello1 (current-output-port)))
```

Example 3: As 1, but output is redirected to a newly created file

```
;; NB: with-output-to-file is an optional procedure in R5RS
(let ((hello0 (lambda () (display "Hello world") (newline))))
  (with-output-to-file "helloworldoutputfile" hello0))
```

Example 4: As 2, but with explicit file open and port close to send output to file

```
(let ((hello1 (lambda (p) (display "Hello world" p) (newline p)))
      (output-port (open-output-file "helloworldoutputfile")))
  (hello1 output-port)
  (close-output-port output-port))
```

Example 5: As 2, but with using call-with-output-file to send output to a file.

```
(let ((hello1 (lambda (p) (display "Hello world" p) (newline p))))
  (call-with-output-file "helloworldoutputfile" hello1))
```

Similar procedures are provided for input. R5RS Scheme provides the predicates `input-port?` and `output-port?`. For character input and output, `write-char`, `read-char`, `peek-char` and `char-ready?` are provided. For writing and reading Scheme expressions, Scheme provides `read` and `write`. On a read operation, the result returned is the end-of-file object if the input port has reached the end of the file, and this can be tested using the predicate `eof-object?`.

In addition to the standard, SRFI 28 defines a basic formatting procedure resembling Common Lisp's `format` function, after which it is named.[31]

Redefinition of standard procedures

In Scheme, procedures are bound to variables. At R5RS the language standard formally mandated that programs may change the variable bindings of built-in procedures, effectively redefining them. (R5RS "Language changes")[4] For example, one may extend + to accept strings as well as numbers by redefining it:

```
(set! +
      (let ((original+ +))
        (lambda args
          (apply (if (or (null? args) (string? (car args)))
                     string-append
                     original+)
                 args))))
(+ 1 2 3)
===> 6
(+ "1" "2" "3")
===> "123"
```

In R6RS every binding, including the standard ones, belongs to some library, and all exported bindings are immutable. (R6RS sec 7.1)[5] Because of this, redefinition of standard procedures by mutation is forbidden. Instead, it is possible to import a different procedure under the name of a standard one, which in effect is similar to redefinition.

Nomenclature and naming conventions

In Standard Scheme, procedures that convert from one datatype to another contain the character string "->" in their name, predicates end with a "?", and procedures that change the value of already-allocated data end with a "!". These conventions are often followed by Scheme programmers.

In formal contexts such as Scheme standards, the word "procedure" is used in preference to "function" to refer to a lambda expression or primitive procedure. In normal usage, the words "procedure" and "function" are used interchangeably. Procedure application is sometimes referred to formally as *combination*.

As in other Lisps, the term "thunk" is used in Scheme to refer to a procedure with no arguments. The term "proper tail recursion" refers to the property of all Scheme implementations, that they perform tail-call optimization so as to support an indefinite number of active tail calls.

The form of the titles of the standards documents since R3RS, "Revised^n Report on the Algorithmic Language Scheme", is a reference to the title of the ALGOL 60 standard document, "Revised Report on the Algorithmic Language Algol 60," The Summary page of R3RS is closely modeled on the Summary page of the ALGOL 60 Report.[32][33]

Review of standard forms and procedures

The language is formally defined in the standards R5RS (1998) and R6RS (2007). They describe standard "forms": keywords and accompanying syntax, which provide the control structure of the language, and standard procedures which perform common tasks.

Standard forms

This table describes the standard forms in Scheme. Some forms appear in more than one row because they cannot easily be classified into a single function in the language.

Forms marked "L" in this table are classed as derived "library" forms in the standard and are often implemented as macros using more fundamental forms in practice, making the task of implementation much easier than in other languages.

Standard forms in the language R5RS Scheme

Purpose	Forms
Definition	define
Binding constructs	lambda, do (L), let (L), let* (L), letrec (L)
Conditional evaluation	if, cond (L), case (L), and (L), or (L)
Sequential evaluation	begin (*)
Iteration	lambda, do (L), named let (L)
Syntactic extension	define-syntax, let-syntax, letrec-syntax, syntax-rules (R5RS), syntax-case (R6RS)
Quoting	quote('), unquote(,), quasiquote(`), unquote-splicing(,@)
Assignment	set!
Delayed evaluation	delay (L)

Note that `begin` is defined as a library syntax in R5RS, but the expander needs to know about it to achieve the splicing functionality. In R6RS it is no longer a library syntax.

Standard procedures

The following two tables describe the standard procedures in R5RS Scheme. R6RS is far more extensive and a summary of this type would not be practical.

Some procedures appear in more than one row because they cannot easily be classified into a single function in the language.

Standard procedures in the language R5RS Scheme

Purpose	Procedures
Construction	vector, make-vector, make-string, list
Equivalence predicates	eq?, eqv?, equal?, string=?, string-ci=?, char=?, char-ci=?
Type conversion	vector->list, list->vector, number->string, string->number, symbol->string, string->symbol, char->integer, integer->char, string->list, list->string
Numbers	*See separate table*
Strings	string?, make-string, string, string-length, string-ref, string-set!, string=?, string-ci=?, string<? string-ci<?, string<=? string-ci<=?, string>? string-ci>?, string>=? string-ci>=?, substring, string-append, string->list, list->string, string-copy, string-fill!
Characters	char?, char=?, char-ci=?, char<? char-ci<?, char<=? char-ci<=?, char>? char-ci>?, char>=? char-ci>=?, char-alphabetic?, char-numeric?, char-whitespace?, char-upper-case?, char-lower-case?, char->integer, integer->char, char-upcase, char-downcase
Vectors	make-vector, vector, vector?, vector-length, vector-ref, vector-set!, vector->list, list->vector, vector-fill!
Symbols	symbol->string, string->symbol, symbol?
Pairs and lists	pair?, cons, car, cdr, set-car!, set-cdr!, null?, list?, list, length, append, reverse, list-tail, list-ref, memq. memv. member, assq, assv, assoc, list->vector, vector->list, list->string, string->list
Identity predicates	boolean?, pair?, symbol?, number?, char?, string?, vector?, port?, procedure?
Continuations	call-with-current-continuation (call/cc), values, call-with-values, dynamic-wind
Environments	eval, scheme-report-environment, null-environment, interaction-environment (optional)
Input/output	display, newline, read, write, read-char, write-char, peek-char, char-ready?, eof-object? open-input-file, open-output-file, close-input-port, close-output-port, input-port?, output-port?, current-input-port, current-output-port, call-with-input-file, call-with-output-file, with-input-from-file(optional), with-output-to-file(optional)
System interface	load (optional), transcript-on (optional), transcript-off (optional)
Delayed evaluation	force
Functional programming	procedure?, apply, map, for-each
Booleans	boolean? not

String and character procedures that contain "-ci" in their names perform case-independent comparisons between their arguments: upper case and lower case versions of the same character are taken to be equal.

Standard numeric procedures in the language R5RS Scheme

Purpose	Procedures
Basic arithmetic operators	+, -, *, /, abs, quotient, remainder, modulo, gcd, lcm, expt, sqrt
Rational numbers	numerator, denominator, rational?, rationalize
Approximation	floor, ceiling, truncate, round
Exactness	inexact->exact, exact->inexact, exact?, inexact?
Inequalities	<, <= , >, >=, =
Miscellaneous predicates	zero?, negative?, positive? odd? even?
Maximum and minimum	max, min
Trigonometry	sin, cos, tan, asin, acos, atan
Exponentials	exp, log
Complex numbers	make-rectangular, make-polar, real-part, imag-part, magnitude, angle, complex?
Input-output	number->string, string->number
Type predicates	integer?, rational?, real?, complex?, number?

Implementations of - and / that take more than two arguments are defined but left optional at R5RS.

Scheme Requests for Implementation

Because of Scheme's minimalism, many common procedures and syntactic forms are not defined by the standard. In order to keep the core language small but facilitate standardization of extensions, the Scheme community has a "Scheme Request for Implementation" (SRFI) process by which extension libraries are defined through careful discussion of extension proposals. This promotes code portability. Many of the SRFIs are supported by all or most Scheme implementations.

SRFIs with fairly wide support in different implementations include:[34]

- 0: feature-based conditional expansion construct
- 1: list library
- 4: homogeneous numeric vector datatypes
- 6: basic string ports
- 8: receive, binding to multiple values
- 9: defining record types
- 13: string library
- 14: character-set library
- 16: syntax for procedures of variable arity
- 17: generalized set!
- 18: Multithreading support
- 19: time data types and procedures
- 25: multi-dimensional array primitives
- 26: notation for specializing parameters without currying
- 27: sources of random bits
- 28: basic format strings
- 29: localization

- 30: nested multi-line comments
- 31: a special form for recursive evaluation
- 37: args-fold: a program argument processor
- 39: parameter objects
- 41: streams
- 42: eager comprehensions
- 43: vector library
- 45: primitives for expressing iterative lazy algorithms
- 60: integers as bits
- 61: a more general cond clause
- 66: octet vectors
- 67: compare procedures

Implementations

The elegant, minimalist design has made Scheme a popular target for language designers, hobbyists, and educators, and because of its small size, that of a typical interpreter, it is also a popular choice for embedded systems and scripting. This has resulted in scores of implementations,[35] most of which differ from each other so much that porting programs from one implementation to another is quite difficult, and the small size of the standard language means that writing a useful program of any great complexity in standard, portable Scheme is almost impossible.[7] The R6RS standard specifies a much broader language, in an attempt to broaden its appeal to programmers.

Almost all implementations provide a traditional Lisp-style read–eval–print loop for development and debugging. Many also compile Scheme programs to executable binary. Support for embedding Scheme code in programs written in other languages is also common, as the relative simplicity of Scheme implementations makes it a popular choice for adding scripting capabilities to larger systems developed in languages such as C. The Gambit, Chicken, and Bigloo Scheme interpreters compile Scheme to C, which makes embedding particularly easy. In addition, Bigloo's compiler can be configured to generate JVM bytecode, and it also features an experimental bytecode generator for .NET.

Some implementations support additional features. For example, Kawa and JScheme provide integration with Java classes, and the Scheme to C compilers often make it easy to use external libraries written in C, up to allowing the embedding of actual C code in the Scheme source. Another example is Pvts, which offers a set of visual tools for supporting the learning of Scheme.

Usage

Scheme is widely used by a number[36] of schools; in particular, a number of introductory Computer Science courses use Scheme in conjunction with the textbook *Structure and Interpretation of Computer Programs* (SICP).[37] For the past 12 years, PLT has run the ProgramByDesign (formerly TeachScheme!) project, which has exposed close to 600 high school teachers and thousands of high school students to rudimentary Scheme programming. MIT's old introductory programming class 6.001 was taught in Scheme,[38] Although 6.001 has been replaced by more modern courses, SICP continues to be taught at MIT.[39] Likewise, the introductory class at UC Berkeley, CS 61A, was until 2013 taught entirely in Scheme, save minor diversions into Logo to demonstrate dynamic scope. Today, like MIT, Berkeley has replaced the syllabus with a more modern version that is primarily taught in Python 3, but the current syllabus is still based on the old curriculum, and parts of the class are still taught in Scheme.[40]

The textbook *How to Design Programs* by Matthias Felleisen, currently at Northeastern University, is used by some institutes of higher education for their introductory computer science courses. Both Northeastern University and Worcester Polytechnic Institute use Scheme exclusively for their introductory courses Fundamentals of Computer Science (CS2500) and Introduction to Program Design (CS1101), respectively.[41][42] Rose-Hulman uses Scheme in its more advanced Programming Language Concepts course.[43] Indiana University's introductory class, C211, is taught entirely in Scheme. A self-paced version of the course, CS 61AS, continues to use Scheme.[44] The introductory computer science courses at Yale and Grinnell College are also taught in Scheme.[45] Programming Design Paradigms,[46] a mandatory course for the Computer science Graduate Students at Northeastern University, also extensively uses Scheme. The former introductory Computer Science course at the University of Minnesota - Twin Cities, CSCI 1901, also used Scheme as its primary language, followed by a course that introduced students to the Java programming language;[47] however, following the example of MIT, the department replaced 1901 with the Python-based CSCI 1133,[48] while functional programming is covered in detail in the third-semester course CSCI 2041.[49] In the software industry, Tata Consultancy Services, Asia's largest software consultancy firm, uses Scheme in their month-long training program for fresh college graduates.

Scheme is/was also used for the following:

- The Document Style Semantics and Specification Language (DSSSL), which provides a method of specifying SGML stylesheets, uses a Scheme subset.[50]
- The well-known open source raster graphics editor GIMP uses TinyScheme as a scripting language.[51]
- Guile has been adopted by GNU project as its official scripting language, and that implementation of Scheme is embedded in such applications as GNU LilyPond and GnuCash as a scripting language for extensions. Likewise, Guile used to be the scripting language for the desktop environment GNOME,[52] and GNOME still has a project that provides Guile bindings to its library stack.[53] There is a project to incorporate Guile into GNU Emacs, GNU's flagship program, replacing the current Emacs Lisp interpreter.
- Elk Scheme is used by Synopsys as a scripting language for its technology CAD (TCAD) tools.[54]
- Shiro Kawai, senior programmer on the movie *Final Fantasy: The Spirits Within*, used Scheme as a scripting language for managing the real-time rendering engine.[55]
- Google App Inventor for Android uses Scheme, where Kawa is used to compile the Scheme code down to byte-codes for the Java Virtual Machine running on Android devices.[56]

See also

- Stalin compiler, a compiler for Scheme.
- *Essentials of Programming Languages*, another classic computer science textbook.
- SXML, an illustrative representation for XML in Scheme that provides a straightforward approach to XML data processing in Scheme.

References

1. "The Scheme Programming Language" (https://groups.csail.mit.edu/mac/projects/scheme/). *MIT*.
2. Common LISP: The Language, 2nd Ed., Guy L. Steele Jr. Digital Press; 1981. ISBN 978-1-55558-041-4. "Common Lisp is a new dialect of Lisp, a successor to MacLisp, influenced strongly by ZetaLisp and to some extent by Scheme and InterLisp."

3. 1178-1990 (Reaff 2008) IEEE Standard for the Scheme Programming Language. IEEE part number STDPD14209, unanimously reaffirmed (http://standards.ieee.org/board/rev/308minutes.html) at a meeting of the IEEE-SA Standards Board Standards Review Committee (RevCom), March 26, 2008 (item 6.3 on minutes), reaffirmation minutes accessed October 2009. NOTE: this document is only available for purchase from IEEE and is not available online at the time of writing (2009).
4. Richard Kelsey; William Clinger; Jonathan Rees; et al. (August 1998). "Revised^5 Report on the Algorithmic Language Scheme" (http://www.schemers.org/Documents/Standards/R5RS/). *Higher-Order and Symbolic Computation*. **11** (1): 7–105. doi:10.1023/A:1010051815785 (https://doi.org/10.1023%2FA%3A1010051815785). Retrieved 2012-08-09.
5. Sperber, Michael; Dybvig, R. Kent; Flatt, Matthew; Van Straaten, Anton; et al. (August 2007). "Revised^6 Report on the Algorithmic Language Scheme (R6RS)" (http://www.r6rs.org). Scheme Steering Committee. Retrieved 2011-09-13.
6. "R6RS ratification-voting results" (http://www.r6rs.org/ratification/results.html). Scheme Steering Committee. 2007-08-13. Retrieved 2012-08-09.
7. Will Clinger, Marc Feeley, Chris Hanson, Jonathan Rees and Olin Shivers (2009-08-20). "Position Statement *(draft)*" (http://scheme-reports.org/2009/position-statement.html). Scheme Steering Committee. Retrieved 2012-08-09.
8. Gerald Jay Sussman and Guy L. Steele, Jr. (December 1998). "The First Report on Scheme Revisited" (https://web.archive.org/web/20060615225746/http://www.brics.dk/~hosc/local/HOSC-11-4-pp399-404.pdf) (PDF). *Higher-Order and Symbolic Computation*. **11** (4): 399–404. doi:10.1023/A:1010079421970 (https://doi.org/10.1023%2FA%3A1010079421970). ISSN 1388-3690 (https://www.worldcat.org/issn/1388-3690). Archived from the original (http://www.brics.dk/~hosc/local/HOSC-11-4-pp399-404.pdf) (PDF) on 2006-06-15. Retrieved 2012-08-09.
9. "R6RS Implementations" (http://www.r6rs.org/implementations.html). r6rs.org. Retrieved 2017-11-24.
10. Abdulaziz Ghuloum (2007-10-27). "R6RS Libraries and syntax-case system (psyntax)" (https://www.cs.indiana.edu/~aghuloum/r6rs-libraries/). Ikarus Scheme. Retrieved 2009-10-20.
11. Keep, Andrew W.; Dybvig, R. Kent (2014). "A run-time representation of scheme record types". *Journal of Functional Programming*. **24** (6): 675–716. doi:10.1017/S0956796814000203 (https://doi.org/10.1017%2FS0956796814000203). ISSN 0956-7968 (https://www.worldcat.org/issn/0956-7968).
12. "Revised^6 Report on the Algorithmic Language Scheme, Appendix E: language changes" (http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-19.html#node_chap_E). Scheme Steering Committee. 2007-09-26. Retrieved 2009-10-20.
13. "R6RS Electorate" (http://www.r6rs.org/ratification/electorate.html). Scheme Steering Committee. 2007. Retrieved 2012-08-09.
14. Marc Feeley (compilation) (2007-10-26). "Implementors' intentions concerning R6RS" (http://lists.r6rs.org/pipermail/r6rs-discuss/2007-October/003351.html). Scheme Steering Committee, r6rs-discuss mailing list. Retrieved 2012-08-09.
15. "R7RS 9th draft available" (http://trac.sacrideo.us/wg/raw-attachment/wiki/WikiStart/r7rs-draft-9.pdf) (PDF). 2013-04-15.
16. Will Clinger (2013-05-10). "extension of voting period" (https://web.archive.org/web/20130721162308/http://lists.scheme-reports.org/pipermail/scheme-reports/2013-May/003401.html#). Scheme Language Steering Committee, scheme-reports mailing list. Archived from the original (http://lists.scheme-reports.org/pipermail/scheme-reports/2013-May/003401.html) on 2013-07-21. Retrieved 2013-07-07.
17. "R7RS final available" (http://trac.sacrideo.us/wg/raw-attachment/wiki/WikiStart/r7rs.pdf) (PDF). 2013-07-06.

18. The Scheme 48 implementation is so-named because the interpreter was written by Richard Kelsey and Jonathan Rees in 48 hours (August 6th – 7th, 1986. See Richard Kelsey; Jonathan Rees; Mike Sperber (2008-01-10). "The Incomplete Scheme 48 Reference Manual for release 1.8" (http://s48.org/1.8/manual/manual.html). Jonathan Rees, s48.org. Retrieved 2012-08-09.
19. Gerald Jay Sussman & Guy Lewis Steele, Jr. (December 1975). "Scheme: An Interpreter for Extended Lambda Calculus" (https://web.archive.org/web/20160510140804/http://library.readscheme.org/page1.html). *AI Memos*. MIT AI Lab. AIM-349. Archived from the original (http://library.readscheme.org/page1.html) (postscript or PDF) on 2016-05-10. Retrieved 2012-08-09.
20. Joel Moses (June 1970), *The Function of FUNCTION in LISP, or Why the FUNARG Problem Should Be Called the Environment Problem*, hdl:1721.1/5854 (https://hdl.handle.net/1721.1%2F5854), AI Memo 199, "A useful metaphor for the difference between FUNCTION and QUOTE in LISP is to think of QUOTE as a porous or an open covering of the function since free variables escape to the current environment. FUNCTION acts as a closed or nonporous covering (hence the term "closure" used by Landin). Thus we talk of "open" Lambda expressions (functions in LISP are usually Lambda expressions) and "closed" Lambda expressions. [...] My interest in the environment problem began while Landin, who had a deep understanding of the problem, visited MIT during 1966-67. I then realized the correspondence between the FUNARG lists which are the results of the evaluation of "closed" Lambda expressions in LISP and ISWIM's Lambda Closures."
21. "A LAMBDA CALCULUS FOR QUANTUM COMPUTATION: EBSCOhost" (https://eds.b.ebscohost.com/ehost/detail/detail?vid=7&sid=6ca10543-f88a-41e2-91d7-06a3838d8da5@sessionmgr101&bdata=JnNpdGU9ZWhvc3QtbGl2ZQ==#AN=14678913&db=aph). *eds.b.ebscohost.com*. Retrieved 2018-10-26.
22. Niehren, J.; Schwinghammer, J.; Smolka, G. (2006-11-08). "A concurrent lambda calculus with futures" (https://hal.inria.fr/inria-00090434/file/0.pdf) (PDF). *Theoretical Computer Science*. **364** (3): 338–356. doi:10.1016/j.tcs.2006.08.016 (https://doi.org/10.1016%2Fj.tcs.2006.08.016). ISSN 0304-3975 (https://www.worldcat.org/issn/0304-3975).
23. Gerald Jay Sussman & Guy Lewis Steele, Jr. (March 1976). "Lambda: The Ultimate Imperative" (https://web.archive.org/web/20160510140804/http://library.readscheme.org/page1.html). *AI Memos*. MIT AI Lab. AIM-353. Archived from the original (http://library.readscheme.org/page1.html) (postscript or PDF) on 2016-05-10. Retrieved 2012-08-09.
24. Gabriel, Richard P.; Pitman, Kent (1988). "Technical Issues of Separation in Function Cells and Value Cells" (http://www.nhplace.com/kent/Papers/Technical-Issues.html). *Lisp and Symbolic Computation*. **1** (1) (published June 1988). pp. 81–101. doi:10.1007/BF01806178 (https://doi.org/10.1007%2FBF01806178). Retrieved 2012-08-09.
25. Philip L. Bewig (2008-01-24). "SRFI 41: Streams" (http://srfi.schemers.org/srfi-41/srfi-41.html). The SRFI Editors, schemers.org. Retrieved 2012-08-09.
26. William Clinger and Jonathan Rees, editors (1991). "Revised^4 Report on the Algorithmic Language Scheme" (http://www.cs.indiana.edu/scheme-repository/R4RS/r4rs_toc.html). *ACM Lisp Pointers*. **4** (3): 1–55. Retrieved 2012-08-09.
27. Flatt, Matthew (2016-04-08). *Binding as sets of scopes* (https://utah.pure.elsevier.com/en/publications/binding-as-sets-of-scopes-2). *ACM SIGPLAN Notices*. **51**. pp. 705–717. doi:10.1145/2837614.2837620 (https://doi.org/10.1145%2F2837614.2837620). ISBN 9781450335492. ISSN 1523-2867 (https://www.worldcat.org/issn/1523-2867).
28. Jonathan Rees, The Scheme of Things The June 1992 Meeting (http://mumble.net/~jar/pubs/scheme-of-things/june-92-meeting.ps) Archived (https://web.archive.org/web/20110716071317/http://mumble.net/~jar/pubs/scheme-of-things/june-92-meeting.ps#) 2011-07-16 at the Wayback Machine (postscript), in Lisp Pointers, V(4), October–December 1992. Retrieved 2012-08-09
29. Taylor Campbell (2005-07-21). "SRFI 62: S-expression comments" (http://srfi.schemers.org/srfi-62/srfi-62.html). The SRFI Editors, schemers.org. Retrieved 2012-08-09.
30. William D Clinger (1999-07-01). "SRFI 6: Basic String Ports" (http://srfi.schemers.org/srfi-6/srfi-6.html). The SRFI Editors, schemers.org. Retrieved 2012-08-09.

31. Scott G. Miller (2002-06-25). "SRFI 28: Basic Format Strings" (http://srfi.schemers.org/srfi-28/srfi-28.html). The SRFI Editors, schemers.org. Retrieved 2012-08-09.
32. J.W. Backus; F.L. Bauer; J.Green; C. Katz; J. McCarthy P. Naur; et al. (January–April 1960). "Revised Report on the Algorithmic Language Algol 60" (http://www.masswerk.at/algol60/report.htm). *Numerische Mathematik, Communications of the ACM, and Journal of the British Computer Society*. Retrieved 2012-08-09.
33. Jonathan Rees; William Clinger, eds. (December 1986). "Revised(3) Report on the Algorithmic Language Scheme (Dedicated to the Memory of ALGOL 60)" (http://groups.csail.mit.edu/mac/ftpdir/scheme-reports/r3rs-html/r3rs_toc.html). *ACM SIGPLAN Notices*. **21** (12): 37–79. CiteSeerX 10.1.1.29.3015 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.3015). doi:10.1145/15042.15043 (https://doi.org/10.1145%2F15042.15043). Retrieved 2012-08-09.
34. "Scheme Systems Supporting SRFIs" (http://srfi.schemers.org/srfi-implementers.html). The SRFI Editors, schemers.org. 2009-08-30. Retrieved 2012-08-09.
35. 75 known implementations of Scheme are listed by "scheme-faq-standards" (http://community.schemewiki.org/?scheme-faq-standards#implementations). Community Scheme Wiki. 2009-06-25. Retrieved 2009-10-20.
36. Ed Martin (2009-07-20). "List of Scheme-using schools" (http://www.schemers.com/schools.html). Schemers Inc. Retrieved 2009-10-20.
37. "List of SICP-using schools" (http://mitpress.mit.edu/sicp/adopt-list.html). MIT Press. 1999-01-26. Retrieved 2009-10-20.
38. Eric Grimson (Spring 2005). "6.001 Structure and Interpretation of Computer Programs" (http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-001Spring-2005/CourseHome/index.htm). MIT Open Courseware. Retrieved 2009-10-20.
39. Alex Vandiver; Nelson Elhage; et al. (January 2009). "6.184 - Zombies drink caffeinated 6.001" (http://web.mit.edu/alexmv/6.001/). MIT CSAIL. Retrieved 2009-10-20.
40. John DeNero (Fall 2019). "Computer Science 61A, Berkeley" (https://cs61a.org/articles/about.html). Department of Electrical Engineering and Computer Sciences, Berkeley. Retrieved 2019-12-17.
41. CS 2500: Fundamentals of Computer Science I (http://www.ccs.neu.edu/course/cs2500/), Northeastern University
42. CS 1101: Introduction to Program Design (A05): course software (http://web.cs.wpi.edu/~cs1101/a05/details.html#software), Worcester Polytechnic Institute
43. "CSSE 304: Programming Language Concepts" (https://www.rose-hulman.edu/Users/faculty/young/CS-Classes/csse304/syllabus.html). Rose-Hulman Institute of Technology.
44. https://berkeley-cs61as.github.io
45. Dana Angluin (Fall 2009). "Introduction to Computer Science (CPSC 201)" (http://zoo.cs.yale.edu/classes/cs201/). The Zoo, Yale University Computer Science Department. Retrieved 2009-10-20.
46. "Programming Design Paradigms CSG107 Course Readings" (http://www.ccs.neu.edu/home/matthias/107-f08/readings.html). Northeastern University College of Computer and Information Science. Fall 2009. Retrieved 2012-08-09.
47. Structure of Computer Programming I (http://www-users.itlabs.umn.edu/classes/Spring-2010/csci1901/) Archived (https://web.archive.org/web/20100619111110/http://www-users.itlabs.umn.edu/classes/Spring-2010/csci1901/#) 2010-06-19 at the Wayback Machine, Computer Science Department, University of Minnesota, Spring 2010 (accessed 2010-01-30).
48. CSci Required Class Course Descriptions and Other Information (https://www.cs.umn.edu/academics/undergraduate/curriculum/required) Archived (https://web.archive.org/web/20191025145152/https://www.cs.umn.edu/academics/undergraduate/curriculum/required) 2019-10-25 at the Wayback Machine, Computer Science Department, University of Minnesota (accessed 2019-10-25)

49. CSCI 2041—New Course (https://www.csdy.umn.edu/~shield/csecc/minutes/2013-4-23/csci2041.html) CSE Curriculum Committee, University of Minnesota (accessed 2019-10-25)
50. Robin Cover (2002-02-25). "DSSSL - Document Style Semantics and Specification Language. ISO/IEC 10179:1996" (http://xml.coverpages.org/dsssl.html). Cover Pages. Retrieved 2012-08-09.
51. "*The major scripting language for the GIMP that has been attached to it today is Scheme.*" From Dov Grobgeld (2002). "The GIMP Basic Scheme Tutorial" (http://www.gimp.org/tutorials/Basic_Scheme/). The GIMP Team. Retrieved 2012-08-09.
52. Todd Graham Lewis; David Zoll; Julian Missig (2002). "GNOME FAQ from Internet Archive" (https://web.archive.org/web/20000522010523/http://www.gnome.org/gnomefaq/html/x930.html). The Gnome Team, gnome.org. Archived from the original (http://www.gnome.org/gnomefaq/html/x930.html) on 2000-05-22. Retrieved 2012-08-09.
53. "guile-gnome" (https://www.gnu.org/software/guile-gnome/). Free Software Foundation. Retrieved 2012-08-09.
54. Laurence Brevard (2006-11-09). "Synopsys MAP-in^{SM} Program Update: EDA Interoperability Developers' Forum" (http://www.synopsys.com/community/interoperability/documents/devforum_pres/2006nov/milkywaysession_mapin_overview.pdf) (PDF). Synopsis Inc. Retrieved 2012-08-09.
55. Kawai, Shiro (October 2002). "Gluing Things Together - Scheme in the Real-time CG Content Production" (http://practical-scheme.net/docs/ILC2002.html). *Proceedings of the First International Lisp Conference, San Francisco*: 342–348. Retrieved 2012-08-09.
56. Bill Magnuson; Hal Abelson & Mark Friedman (2009-08-11). "Under the Hood of App Inventor for Android" (http://googleresearch.blogspot.com/2009/08/under-hood-of-app-inventor-for-android.html). Google Inc, Official Google Research blog. Retrieved 2012-08-09.

Further reading

- An Introduction to Scheme and its Implementation (ftp://ftp.cs.utexas.edu/pub/garbage/cs345/schintro-v14/schintro_toc.html) (a mirror (http://icem-www.folkwang-hochschule.de/~finnendahl/cm_kurse/doc/schintro/schintro_toc.html))
- Christopher T. Haynes (1999-06-22). "The Scheme Programming Language Standardization Experience" (http://acm.org/tsc/sstd.html).
- Guy L. Steele, Jr., Richard P. Gabriel. "The Evolution of Lisp" (http://www.dreamsongs.org/Files/HOPL2-Uncut.pdf) (PDF).
- Gerald Sussman and Guy Steele, *SCHEME: An Interpreter for Extended Lambda Calculus* AI Memo 349 (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.80), MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, December 1975.

External links

- Scheme (https://curlie.org/Computers/Programming/Languages/Lisp/Scheme) at Curlie
- Scheme Programming at Wikibooks
- Write Yourself a Scheme in 48 Hours at Wikibooks
- Media related to Scheme (programming language) at Wikimedia Commons