Overview of the Revised^7 Algorithmic Language Scheme

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

***** EDITOR'S DRAFT *** September 6, 2022**

OVERVIEW OF SCHEME

This paper gives an overview of the small language of R^{7}RS. The purpose of this overview is to explain enough about the basic concepts of the language to facilitate understanding of the R^{7}RS report, which is organized as a reference manual. Consequently, this overview is not a complete introduction to the language, nor is it precise in all respects or normative in any way.

Following Algol, Scheme is a statically scoped programming language. After macros are expanded, each use of a variable is associated with a lexically apparent binding of that variable.

Scheme has latent as opposed to manifest types. Types are associated with objects (also called values) rather than with variables. (Some authors refer to languages with latent types as untyped, weakly typed or dynamically typed languages.) Other languages with latent types are Python, Ruby, Smalltalk, and other dialects of Lisp. Languages with manifest types (sometimes referred to as strongly typed or statically typed languages) include Algol 60, C, C#, Java, Haskell, and ML.

All objects created in the course of a Scheme computation, including procedures and continuations, have unlimited extent. No Scheme object is ever destroyed. The reason that implementations of Scheme do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they can prove that the object cannot possibly matter to any future computation. Other languages in which most objects have unlimited extent include C#, Java, Haskell, most Lisp dialects, ML, Python, Ruby, and Smalltalk.

Implementations of Scheme must be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure. Thus with a properly tail-recursive implementation, iteration can be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are useful only as syntactic sugar.

Scheme was one of the first languages to support procedures as objects in their own right. Procedures can be created dynamically, stored in data structures, returned as results of procedures, and so on. Other languages with these properties include Common Lisp, Haskell, ML, Ruby, and Smalltalk.

One distinguishing feature of Scheme is that continuations, which in most other languages only operate behind the scenes, also have "first-class" status. First-class continuations are useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines.

In Scheme, the argument expressions of a procedure call are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not. C, C#, Common Lisp, Python, Ruby, and Smalltalk are other languages that always evaluate argument expressions before invoking a procedure. This is distinct from the lazy-evaluation semantics of Haskell, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure.

Scheme's model of arithmetic provides a rich set of numerical types and operations on them. Furthermore, it distinguishes *exact* and *inexact* numbers: Essentially, an exact number object corresponds to a number exactly, and an inexact number is the result of a computation that involved rounding or other approximations.

1. Basic types

Scheme programs manipulate *objects*, which are also referred to as *values*. Scheme objects are organized into sets of values called *types*. This chapter gives an overview of the fundamentally important types of the Scheme language.

Note: As Scheme is latently typed, the use of the term *type* in the Scheme context differs from the use of the term in the context of other languages, particularly those with manifest typing.

Numbers Scheme supports a rich variety of numerical data types, including integers of arbitrary precision, rational numbers, complex numbers, and inexact numbers of various kinds.

Booleans A boolean is a truth value, and can be either true or false. In Scheme, the object for "false" is written `#f`. The object for "true" is written `#t`. In most places where a truth value is expected, however, any object different from `#f` counts as true.

Pairs and lists A pair is a data structure with two components. The most common use of pairs is to represent (singly linked) lists, where the first component (the "car") represents the first element of the list, and the second component (the "cdr") the rest of the list. Scheme also has a distinguished empty list, which is the last cdr in a chain of pairs that form a list.

Symbols A symbol is an object representing a string, the symbol's *name*. Unlike strings, two symbols whose names are spelled the same way are never distinguishable. Symbols are useful for many applications; for instance, they can be used the way enumerated values are used in other languages.

In R^{7}RS, unlike R^{5}RS, symbols and identifiers are case-sensitive.

Characters Scheme characters mostly correspond to textual characters. More precisely, they are isomorphic to a subset of the *scalar values* of the Unicode standard, with possible implementation-dependent extensions.

Strings Strings are finite sequences of characters with fixed length and thus represent arbitrary Unicode texts.

Vectors Vectors, like lists, are linear data structures representing finite sequences of arbitrary objects. Whereas the elements of a list are accessed sequentially through the chain of pairs representing it, the elements of a vector are addressed by integer indices. Thus, vectors are more appropriate than lists for random access to elements.

Bytevectors Bytevectors are similar to vectors, except that their contents are *bytes*, exact integers in the range 0 to 255 inclusive.

Procedures Procedures are values in Scheme.

Records Records are structured values, and are aggregations of zero or more *fields*, each of which holds a single location. Records are organized into *record types*. A predicate, a constructor, and field accessors and mutators can be defined for each record type.

Ports Ports represent input and output devices. To Scheme, an input port is a Scheme object that can deliver data upon command, while an output port is a Scheme object that can accept data.

2. Expressions

The most important elements of Scheme code are *expressions*. Expressions can be *evaluated*, producing a *value* (actually, any number of values.) The most fundamental expressions are literal expressions:

```
#t                              ⟹ #t
23                              ⟹ 23
```

This notation means that the expression `#t` evaluates to `#t`, that is, the value for "true", and that the expression `23` evaluates to a number representing the number 23.

Compound expressions are formed by placing parentheses around their subexpressions. The first subexpression identifies an operation; the remaining subexpressions are operands to the operation:

```
(+ 23 42)                       ⟹ 65
(+ 14 (* 23 42))                ⟹ 980
```

In the first of these examples, `+` is the name of the built-in operation for addition, and `23` and `42` are the operands. The expression `(+ 23 42)` reads as "the sum of 23 and 42". Compound expressions can be nested—the second example reads as "the sum of 14 and the product of 23 and 42".

As these examples indicate, compound expressions in Scheme are always written using the same prefix notation. As a consequence, the parentheses are needed to indicate structure. Consequently, "superfluous" parentheses, which are often permissible in mathematical notation and also in many programming languages, are not allowed in Scheme.

As in many other languages, whitespace (including line endings) is not significant when it separates subexpressions of an expression, and can be used to indicate structure.

3. Variables and binding

Scheme allows identifiers to stand for locations containing values. These identifiers are called variables. In many cases, specifically when the location's value is never modified after its creation, it is useful to think of the variable as standing for the value directly.

```
(let ((x 23)
      (y 42))
  (+ x y))                      ⟹ 65
```

In this case, the expression starting with `let` is a binding construct. The parenthesized structure following the `let` lists variables alongside expressions: the variable `x` alongside `23`, and the variable `y` alongside `42`. The `let` expression binds `x` to 23, and `y` to 42. These bindings are available in the *body* of the `let` expression, `(+ x y)`, and only there.

4. Definitions

The variables bound by a `let` expression are *local*, because their bindings are visible only in the `let`'s body. Scheme also allows creating top-level bindings for identifiers as follows:

```
(define x 23)
(define y 42)
(+ x y)                         ⟹ 65
```

(These are actually "top-level" in the body of a top-level program or library.)

The first two parenthesized structures are *definitions*; they create top-level bindings, binding `x` to 23 and `y` to 42. Definitions are not expressions, and cannot appear in all places where an expression can occur. Moreover, a definition has no value.

Bindings follow the lexical structure of the program: When several bindings with the same name exist, a variable refers

to the binding that is closest to it, starting with its occurrence in the program and going from inside to outside, and referring to an outermost binding if no local binding can be found along the way:

```
(define x 23)
(define y 42)
(let ((y 43))
  (+ x y))                    ⟹ 66

(let ((y 43))
  (let ((y 44))
    (+ x y)))                 ⟹ 67
```

5. Procedures

Definitions can also be used to define procedures:

```
(define (f x)
  (+ x 42))

(f 23)                        ⟹ 65
```

A procedure is, slightly simplified, an abstraction of an expression over objects. In the example, the first definition defines a procedure called `f`. (Note the parentheses around `f x`, which indicate that this is a procedure definition.) The expression `(f 23)` is a procedure call meaning, roughly, "evaluate `(+ x 42)` (the body of the procedure) with `x` bound to 23".

As procedures are objects, they can be passed to other procedures:

```
(define (f x)
  (+ x 42))

(define (g p x)
  (p x))

(g f 23)                      ⟹ 65
```

In this example, the body of `g` is evaluated with `p` bound to `f` and `x` bound to 23, which is equivalent to `(f 23)`, which evaluates to 65.

In fact, many predefined operations of Scheme are provided not by syntax, but by variables whose values are procedures. The `+` operation, for example, which receives special syntactic treatment in many other languages, is just a regular identifier in Scheme, bound to a procedure that adds numbers. The same holds for `*` and many others:

```
(define (h op x y)
  (op x y))

(h + 23 42)                   ⟹ 65
(h * 23 42)                   ⟹ 966
```

Procedure definitions are not the only way to create procedures. A `lambda` expression creates a new procedure as an object, with no need to specify a name:

```
((lambda (x) (+ x 42)) 23)  ⟹ 65
```

The entire expression in this example is a procedure call; `(lambda (x) (+ x 42))`, evaluates to a procedure that takes a single number and adds 42 to it.

6. Procedure calls and syntactic keywords

Whereas `(+ 23 42)`, `(f 23)`, and `((lambda (x) (+ x 42)) 23)` are all examples of procedure calls, `lambda` and `let` expressions are not. This is because `let`, even though it is an identifier, is not a variable, but is instead a *syntactic keyword*. A list that has a syntactic keyword as its first subexpression obeys special rules determined by the keyword. The `define` identifier in a definition is also a syntactic keyword. Hence, definitions are also not procedure calls.

The rules for the `lambda` keyword specify that the first sublist is a list of parameters, and the remaining sublists are the body of the procedure. In `let` expressions, the first sublist is a list of binding specifications, and the remaining sublists constitute a body of expressions.

Procedure calls can be distinguished from these *expression types* by looking for a syntactic keyword in the first position of a list: if the first position does not contain a syntactic keyword, the expression is a procedure call. The set of syntactic keywords of Scheme is fairly small, which usually makes this task fairly simple. It is possible, however, to create new bindings for syntactic keywords.

7. Assignment

Scheme variables bound by definitions or `let` or `lambda` expressions are not actually bound directly to the objects specified in the respective bindings, but to locations containing these objects. The contents of these locations can subsequently be modified destructively via *assignment*:

```
(let ((x 23))
  (set! x 42)
  x)                          ⟹ 42
```

In this case, the body of the `let` expression consists of two expressions which are evaluated sequentially, with the value of the final expression becoming the value of the entire `let` expression. The expression `(set! x 42)` is an assignment, saying "replace the object in the location referenced by `x` with 42". Thus, the previous value of `x`, 23, is replaced by 42.

8. Derived syntax and macros

Many of the expression types specified as part of the R[7]RS small language can be translated into more basic expression types. For example, a `let` expression can be translated into a procedure call and a `lambda` expression. The following two expressions are equivalent:

```
(let ((x 23)
      (y 42))
  (+ x y))                    ⟹ 65

((lambda (x y) (+ x y)) 23 42)
          ⟹ 65
```

Syntax expressions like `let` expressions are called *derived* because their semantics can be derived from that of other kinds of expressions by a syntactic transformation. Some procedure definitions are also derived expressions. The following two definitions are equivalent:

```
(define (f x)
  (+ x 42))

(define f
  (lambda (x)
    (+ x 42)))
```

In Scheme, it is possible for a program to create its own derived expressions by binding syntactic keywords to macros:

```
(define-syntax def
  (syntax-rules ()
    ((def f (p ...) body)
     (define (f p ...)
       body))))

(def f (x)
  (+ x 42))
```

The `define-syntax` construct specifies that a parenthesized structure matching the pattern `(def f (p ...) body)`, where `f`, `p`, and `body` are pattern variables, is translated to `(define (f p ...) body)`. Thus, the `def` expression appearing in the example gets translated to:

```
(define (f x)
  (+ x 42))
```

The ability to create new syntactic keywords makes Scheme extremely flexible and expressive, allowing many of the features built into other languages to be implemented directly in Scheme: any Scheme programmer can add new expression types.

9. Syntactic data and datum values

Datum values constitute a subset of Scheme objects. These include booleans, numbers, characters, symbols, and strings as well as lists, vectors, and bytevectors whose elements are datum values. Each datum value can be represented textually as a *syntactic datum*, which can be written out and read back in without loss of information. There is in general more than one syntactic datum corresponding to each datum value. Moreover, each datum value can be trivially translated to a literal expression in a program by prepending a `'` to a corresponding syntactic datum:

```
'23                           ⟹ 23
'#t                           ⟹ #t
'foo                          ⟹ foo
'(1 2 3)                      ⟹ (1 2 3)
'#(1 2 3)                     ⟹ #(1 2 3)
```

The `'` shown in the previous examples is not needed for representations of literal constants other than symbols and lists. The syntactic datum `foo` represents a symbol with name "foo", and `'foo` is a literal expression with that symbol as its value. `(1 2 3)` is a syntactic datum that represents a list with elements 1, 2, and 3, and `'(1 2 3)` is a literal expression with this list as its value. Likewise, `#(1 2 3)` is a syntactic datum that represents a vector with elements 1, 2 and 3, and `'#(1 2 3)` is the corresponding literal.

Syntactic datums are a superset of Scheme expressions. Thus, data can be used to represent Scheme expressions as data objects. In particular, symbols can be used to represent identifiers.

```
'(+ 23 42)                    ⟹ (+ 23 42)
'(define (f x) (+ x 42))
          ⟹ (define (f x) (+ x 42))
```

This facilitates writing programs that operate on Scheme source code, in particular interpreters and program transformers.

10. Continuations

Whenever a Scheme expression is evaluated there is a *continuation* wanting the result of the expression. The continuation represents an entire (default) future for the computation. For example, informally the continuation of `3` in the expression

```
(+ 1 3)
```

adds 1 to it. Normally these ubiquitous continuations are hidden behind the scenes and programmers do not think much about them. On rare occasions, however, a programmer needs to deal with continuations explicitly. The `call-with-current-continuation` procedure allows Scheme programmers to do that by creating a procedure that reinstates the current continuation. The `call-with-current-continuation` procedure accepts a procedure, calls it immediately with an argument that is an *escape procedure*. This escape procedure can then be called with an argument that becomes the result of the call to `call-with-current-continuation`. That is, the escape procedure abandons its own continuation, and reinstates the continuation of the call to `call-with-current-continuation`.

In the following example, an escape procedure representing the continuation that adds 1 to its argument is bound to `escape`, and then called with 3 as an argument. The continuation of the call to `escape` is abandoned, and instead the 3 is passed to the continuation that adds 1:

```
(+ 1 (call-with-current-continuation
       (lambda (escape)
         (+ 2 (escape 3)))))
          ⟹ 4
```

An escape procedure has unlimited extent: It can be called after the continuation it captured has been invoked, and it can be called multiple times. This makes `call-with-current-continuation` significantly more powerful than typical non-local control constructs such as exceptions in other languages.

11. Libraries

Scheme code can be organized in components called *libraries*. Each library contains definitions and expressions. It can import definitions from other libraries and export definitions to other libraries.

The following library called `(hello)` exports a definition called `hello-world`, and imports the base library and the display library. The `hello-world` export is a procedure that displays `Hello World` on a separate line:

```
(define-library (hello)
  (export hello-world)
  (import (scheme base)
          (scheme display))
  (begin
    (define (hello-world)
      (display "Hello World")
      (newline))))
```

12. Programs

Libraries are invoked by other libraries, but ultimately by a Scheme *program*. Like a library, a program contains imports, definitions and expressions, and specifies an entry point for execution. Thus a program defines, via the transitive closure of the libraries it imports, a Scheme program.

The following program obtains the first argument from the command line via the `command-line` procedure from the process-context library. It then opens the file using `with-input-from-file`, which causes the file to be the current input port, and arranges for it to be closed at the end. Next, it calls the `read-line` procedure to read a line of text from the file, and then `write-string` and `newline` to output the line, then looping until the end of file:

```
(import (scheme base)
        (scheme file)
        (scheme process-context))
(with-input-from-file
  (cadr (command-line))
  (lambda ()
    (let loop ((line (read-line)))
      (unless (eof-object? line)
        (write-string line)
        (newline)
        (loop (read-line))))))
```

13. The REPL

Implementations may provide an interactive session called a *REPL* (Read-Eval-Print Loop), where import declarations, expressions and definitions can be entered and evaluated one at a time. The REPL starts out with the base library imported, and possibly other libraries. An implementation may provide a mode of operation in which the REPL reads its input from a file. Such a file is not, in general, the same as a program, because it can contain import declarations in places other than the beginning.

Here is a short REPL session. The `>` character represents the REPL's prompt for input:

```
> ; A few simple things
> (+ 2 2)
4
> (sin 4)
Undefined variable: sin
> (import (scheme inexact))
> (sin 4)
-0.756802495307928
> (define sine sin)
> (sine 4)
-0.756802495307928
> ; Guy Steele's three-part test
> ; True is true ...
> #t
#t
> ; 100!/99! = 100 ...
> (define (fact n)
    (if (= n 0) 1 (* n (fact (- n 1)))))
> (/ (fact 100) (fact 99))
100
> ; If it returns the *right* complex number,
> ; so much the better ...
> (define (atanh x)
    (/ (- (log (+ 1 x))
          (log (- 1 x)))
       2))
> (atanh -2)
-0.549306144334055+1.5707963267949i
```