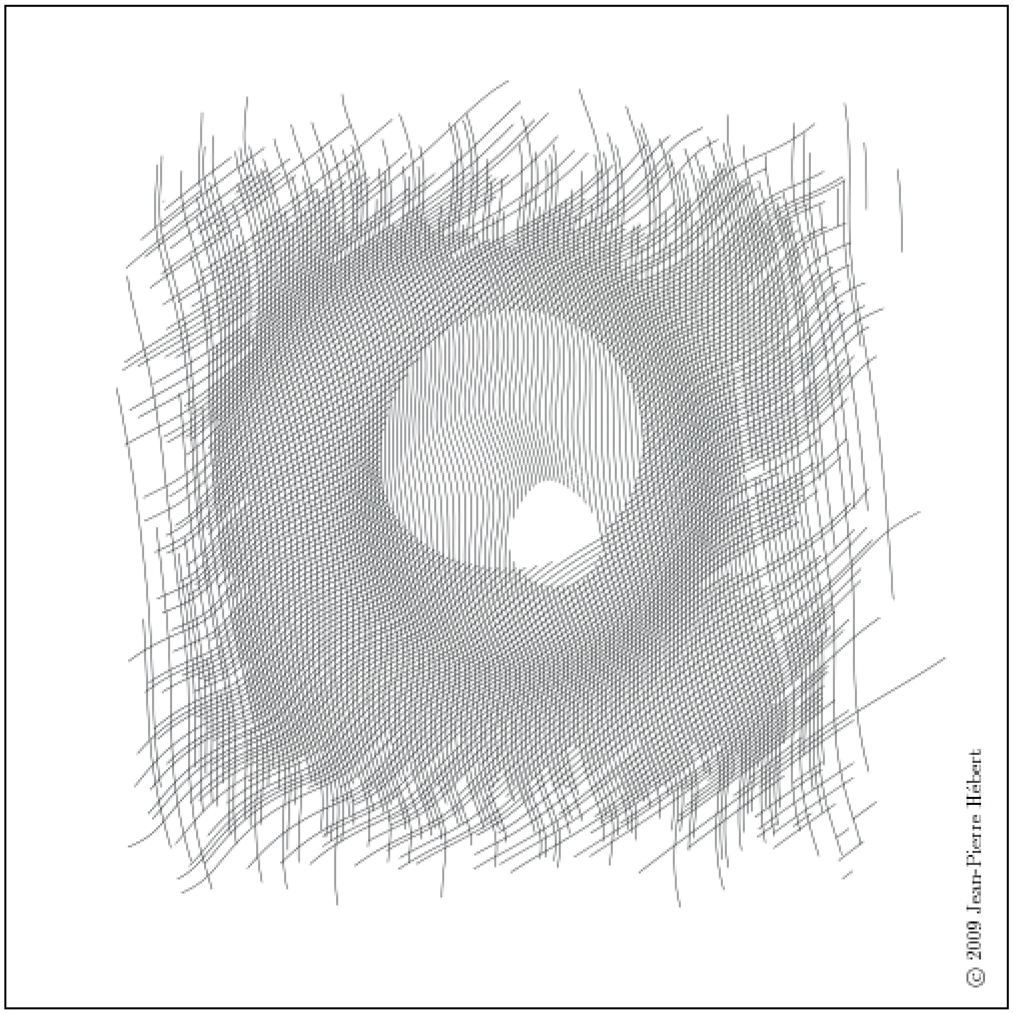

Chapter 1. Introduction

Scheme is a general-purpose computer programming language. It is a high-level language, supporting operations on structured data such as strings, lists, and vectors, as well as operations on more traditional data such as numbers and characters. While Scheme is often identified with symbolic applications, its rich set of data types and flexible control structures make it a truly versatile language. Scheme has been employed to write text editors, optimizing compilers, operating systems, graphics packages, expert systems, numerical applications, financial analysis packages, virtual reality systems, and practically every other type of application imaginable. Scheme is a fairly simple language to learn, since it is based on a handful of syntactic forms and semantic concepts and since the interactive nature of most implementations encourages experimentation. Scheme is a challenging language to understand fully, however; developing the ability to use its full potential requires careful study and practice.

Scheme programs are highly portable across versions of the same Scheme implementation on different machines, because machine dependencies are almost completely hidden from the programmer. They are also portable across different implementations because of the efforts of a group of Scheme language designers who have published a series of reports, the "Revised Reports" on Scheme. The most recent, the "Revised^6 Report" [24], emphasizes portability through a set of standard libraries and a standard mechanism for defining new portable libraries and top-level programs.

Although some early Scheme systems were inefficient and slow, many newer compiler-based implementations are fast, with programs running on par with equivalent programs written in lower-level languages. The relative inefficiency that sometimes remains results from run-time checks that support generic arithmetic and help programmers detect and correct various common programming errors. These checks may be disabled in many implementations.

Scheme supports many types of data values, or *objects*, including characters, strings, symbols, lists or vectors of objects, and a full set of numeric data types, including complex, real, and arbitrary-precision rational numbers.

The storage required to hold the contents of an object is dynamically allocated as necessary and retained until no longer needed, then automatically deallocated, typically by a *garbage collector* that periodically recovers the storage used by inaccessible objects. Simple atomic values, such as small integers, characters, booleans, and the empty list, are typically represented as immediate values and thus incur no allocation or deallocation overhead.

Regardless of representation, all objects are *first-class* data values; because they are retained indefinitely, they may be passed freely as arguments to procedures, returned as values from procedures, and combined to form new objects. This is in contrast with many other languages where composite data values such as arrays are either statically allocated and never deallocated, allocated on entry to a block of code and unconditionally deallocated on exit from the block, or explicitly allocated *and* deallocated by the programmer.

Scheme is a call-by-value language, but for at least mutable objects (objects that can be modified), the values are pointers to the actual storage. These pointers remain behind the scenes, however, and programmers need not be conscious of them except to understand that the storage for an object is not copied when an object is passed to or returned from a procedure.

At the heart of the Scheme language is a small core of syntactic forms from which all other forms are built. These core forms, a set of extended syntactic forms derived from them, and a set of primitive procedures make up the full Scheme language. An interpreter or compiler for Scheme can be quite small and potentially fast and highly reliable. The extended syntactic forms and many primitive procedures can be defined in Scheme itself, simplifying the implementation and increasing reliability.

Scheme programs share a common printed representation with Scheme data structures. As a result, any Scheme program has a natural and obvious internal representation as a Scheme object. For example, variables and syntactic keywords correspond to symbols, while structured syntactic forms correspond to lists. This representation is the basis for the syntactic extension facilities provided by Scheme for the definition of new syntactic forms in terms of existing syntactic forms and procedures. It also facilitates the implementation of interpreters, compilers, and other program transformation tools for Scheme directly in Scheme, as well as program transformation tools for other languages in Scheme.

Scheme variables and keywords are *lexically scoped*, and Scheme programs are *block-structured*. Identifiers may be imported into a program or library or bound locally within a given block of code such as a library, program, or procedure body. A local binding is visible only lexically, i.e., within the program text that makes up the particular block of code. An occurrence of an identifier of the same name outside this block refers to a different binding; if no binding for the identifier exists outside the block, then the reference is invalid. Blocks may be nested, and a binding in one block may *shadow* a binding for an identifier of the same name in a surrounding block. The *scope* of a binding is the block in which the bound identifier is visible minus any portions of the block in which the identifier is shadowed. Block structure and lexical scoping help create programs that are modular, easy to read, easy to maintain, and reliable. Efficient code for lexical scoping is possible because a compiler can determine before program evaluation the scope of all bindings and the binding to which each identifier reference resolves. This does not mean, of course, that a compiler can determine the values of all variables, since the actual values are not computed in most cases until the program executes.

In most languages, a procedure definition is simply the association of a name with a block of code. Certain variables local to the block are the parameters of the procedure. In some languages, a procedure definition may appear within another block or procedure so long as the procedure is invoked only during execution of the enclosing block. In others, procedures can be defined only at top level. In Scheme, a procedure definition may appear within another block or procedure, and the procedure may be invoked at any time thereafter, even if the enclosing block has completed its execution. To support lexical scoping, a procedure carries the lexical context (environment) along with its code.

Furthermore, Scheme procedures are not always named. Instead, procedures are first-class data objects like strings or numbers, and variables are bound to procedures in the same way they are bound to other objects.

As with procedures in most other languages, Scheme procedures may be recursive. That is, any procedure may invoke itself directly or indirectly. Many algorithms are most elegantly or efficiently specified recursively. A special case of recursion, called tail recursion, is used to express iteration, or looping. A *tail call* occurs when one procedure directly returns the result of invoking another procedure; *tail recursion* occurs when a procedure recursively tail-calls itself, directly or indirectly. Scheme implementations are required to implement tail calls as jumps (gotos), so the storage overhead normally associated with recursion is avoided. As a result, Scheme programmers need master only simple procedure calls and recursion and need not be burdened with the usual assortment of looping constructs.

Scheme supports the definition of arbitrary control structures with *continuations*. A continuation is a procedure that embodies the remainder of a program at a given point in the program. A continuation may be obtained at any time during the execution of a program. As with other procedures, a continuation is a first-class object and may be invoked at any time after its creation. Whenever it is invoked, the program immediately continues from the point where the continuation was obtained. Continuations allow the implementation of complex control mechanisms including explicit backtracking, multithreading, and coroutines.

Scheme also allows programmers to define new syntactic forms, or *syntactic extensions*, by writing transformation procedures that determine how each new syntactic form maps to existing syntactic forms. These transformation procedures are themselves expressed in Scheme with the help of a convenient high-level pattern language that automates syntax checking, input deconstruction, and output reconstruction. By default, lexical scoping is maintained through the transformation process, but the programmer can exercise control over the scope of all identifiers appearing in the output of a transformer. Syntactic extensions are useful for defining new language constructs, for emulating language constructs found in other languages, for achieving the effects of in-line code expansion, and even for emulating entire languages in Scheme. Most large Scheme programs are built from a mix of syntactic extensions and procedure definitions.

Scheme evolved from the Lisp language and is considered to be a dialect of Lisp. Scheme inherited from Lisp the treatment of values as first-class objects, several important data types, including symbols and lists, and the representation of programs as objects, among other things. Lexical scoping and block structure are features taken from Algol 60 [21]. Scheme was the first Lisp dialect to adopt lexical scoping and block structure, first-class procedures, the treatment of tail calls as jumps, continuations, and lexically scoped syntactic extensions.

Common Lisp [27] and Scheme are both contemporary Lisp languages, and the development of each has been influenced by the other. Like Scheme but unlike earlier Lisp languages, Common Lisp adopted lexical scoping and first-class procedures, although Common Lisp's syntactic extension facility does not respect lexical scoping. Common Lisp's evaluation rules for procedures are different from the evaluation rules for other objects, however, and it maintains a separate namespace for procedure variables, thereby inhibiting the use of procedures as first-class objects. Also, Common Lisp does not support continuations or require proper treatment of tail calls, but it does support several less general control structures not found in Scheme. While the two languages are similar, Common Lisp includes more specialized constructs, while Scheme includes more general-purpose building blocks out of which such constructs (and others) may be built.

The remainder of this chapter describes Scheme's syntax and naming conventions and the typographical conventions used throughout this book.

Section 1.1. Scheme Syntax

Scheme programs are made up of keywords, variables, structured forms, constant data (numbers, characters, strings, quoted vectors, quoted lists, quoted symbols, etc.), whitespace, and comments.

Keywords, variables, and symbols are collectively called identifiers. Identifiers may be formed from letters, digits, and certain special characters, including `?`, `!`, `.`, `+`, `-`, `*`, `/`, `<`, `=`, `>`, `:`, `$`, `%`, `^`, `&`, `_`, `~`, and `@`, as well as a set of additional Unicode characters. Identifiers cannot start with an at sign (`@`) and normally cannot start with any character that can start a number, i.e., a digit, plus sign (`+`), minus sign (`-`), or decimal point (`.`). Exceptions are `+`, `-`, and `...`, which are valid identifiers, and any identifier starting with `->`. For example, `hi`, `Hello`, `n`, `x`, `x3`, `x+2`, and `?$&*!!!` are all identifiers. Identifiers are delimited by whitespace, comments, parentheses, brackets, string (double) quotes (`"`), and hash marks(`#`). A delimiter or any other Unicode character may be included anywhere within the name of an identifier as an escape of the form `\x`*`sv`*`;`, where *`sv`* is the scalar value of the character in hexadecimal notation.

There is no inherent limit on the length of a Scheme identifier; programmers may use as many characters as necessary. Long identifiers are no substitute for comments, however, and frequent use of long identifiers can make a program difficult to format and consequently difficult to read. A good rule is to use short identifiers when the scope of the identifier is small and longer identifiers when the scope is larger.

Identifiers may be written in any mix of upper- and lower-case letters, and case is significant, i.e., two identifiers are different even if they differ only in case. For example, `abcde`, `Abcde`, `AbCdE`, and `ABCDE` all refer to different identifiers. This is a change from previous versions of the Revised Report.

Structured forms and list constants are enclosed within parentheses, e.g., `(a b c)` or `(* (- x 2) y)`. The empty list is written `()`. Matched sets of brackets (`[ ]`) may be used in place of parentheses and are often used to set off the subexpressions of certain standard syntactic forms for readability, as shown in examples throughout this book. Vectors are written similarly to lists, except that they are preceded by `#(` and terminated by `)`, e.g., `#(this is a vector of symbols)`. Bytevectors are written as sequences of unsigned byte values (exact integers in the range 0 through 255) bracketed by `#vu8(` and `)`, e.g., `#vu8(3 250 45 73)`.

Strings are enclosed in double quotation marks, e.g., `"I am a string"`. Characters are preceded by `#\`, e.g., `#\a`. Case is important within character and string constants, as within identifiers. Numbers may be written as integers, e.g., -123, as ratios, e.g., 1/2, in floating-point or scientific notation, e.g., 1.3 or 1e23, or as complex numbers in rectangular or polar notation, e.g., 1.3-2.7i or -1.2@73. Case is not important in the syntax of a number. The boolean values representing *true* and *false* are written `#t` and `#f`. Scheme conditional expressions actually treat `#f` as false and all other objects as true, so `3`, `0`, `()`, `"false"`, and `nil` all count as true.

Details of the syntax for each type of constant data are given in the individual sections of Chapter 6 and in the formal syntax of Scheme starting on page 455.

Scheme expressions may span several lines, and no explicit terminator is required. Since the number of whitespace characters (spaces and newlines) between expressions is not significant, Scheme programs should be indented to show the structure of the code in a way that makes the code as readable as possible. Comments may appear on any line of a Scheme program, between a semicolon (`;`) and the end of the line. Comments explaining a particular Scheme expression are normally placed at the same indentation level as the expression, on the line before the expression. Comments explaining a procedure or group of procedures are normally placed before the procedures, without indentation. Multiple comment characters are often used to set off the latter kind of comment, e.g., `;;; The following procedures ....`

Two other forms of comments are supported: block comments and datum comments. Block comments are delimited by `#|` and `|#` pairs, and may be nested. A datum comment consists of a `#;` prefix and the datum (printed data value) that follows it. Datum comments are typically used to comment out individual definitions or expressions. For example, `(three #;(not four) element list)` is just what it says. Datum comments may also be nested, though `#;#;(a)(b)` has the somewhat nonobvious effect of commenting out both `(a)` and `(b)`.

Some Scheme values, such as procedures and ports, do not have standard printed representations and can thus never appear as a constant in the printed syntax of a program. This book uses the notation `#<`*`description`*`>` when showing the output of an operation that returns such a value, e.g., `#<procedure>` or `#<port>`.

Section 1.2. Scheme Naming Conventions

Scheme's naming conventions are designed to provide a high degree of regularity. The following is a list of these naming conventions:

- Predicate names end in a question mark (`?`). Predicates are procedures that return a true or false answer, such as `eq?`, `zero?`, and `string=?`. The common numeric comparators `=`, `<`, `>`, `<=`, and `>=` are exceptions to this naming convention.

- Type predicates, such as `pair?`, are created from the name of the type, in this case `pair`, and the question mark.

- The names of most character, string, and vector procedures start with the prefix `char-`, `string-`, and `vector-`, e.g., `string-append`. (The names of some list procedures start with `list-`, but most do not.)

- The names of procedures that convert an object of one type into an object of another type are written as $type_1$`->`$type_2$, e.g., `vector->list`.

- The names of procedures and syntactic forms that cause side effects end with an exclamation point (`!`). These include `set!` and `vector-set!`. Procedures that perform input or output technically cause side effects, but their names are exceptions to this rule.

Programmers should employ these same conventions in their own code whenever possible.

Section 1.3. Typographical and Notational Conventions

A standard procedure or syntactic form whose sole purpose is to perform some side effect is said to return *unspecified*. This means that an implementation is free to return any number of values, each of which can be any Scheme object, as the value of the procedure or syntactic form. Do not count on these values being the same across implementations, the same across versions of the same implementation, or even the same across two uses of the procedure or syntactic form. Some Scheme systems routinely use a special object to represent unspecified values. Printing of this object is often suppressed by interactive Scheme systems, so that the values of expressions returning unspecified values are not printed.

While most standard procedures return a single value, the language supports procedures that return zero, one, more than one, or even a variable number of values via the mechanisms described in Section 5.8. Some standard expressions can evaluate to multiple values if one of their subexpressions evaluates to multiple values, e.g., by calling a procedure that returns multiple values. When this situation can occur, an expression is said to return "the values" rather than simply "the value" of its subexpression. Similarly, a standard procedure that returns the values resulting from a call to a procedure argument is said to return the values returned by the procedure argument.

This book uses the words "must" and "should" to describe program requirements, such as the requirement to provide an index that is less than the length of the vector in a call to `vector-ref`. If the word "must" is used, it means that the requirement is enforced by the implementation, i.e., an exception is raised, usually with condition type `&assertion`. If the word "should" is used, an exception may or may not be raised, and if not, the behavior of the program is undefined.

The phrase "syntax violation" is used to describe a situation in which a program is malformed. Syntax violations are detected prior to program execution. When a syntax violation is detected, an exception of type `&syntax` is raised and the program is not executed.

The typographical conventions used in this book are straightforward. All Scheme objects are printed in a `typewriter` typeface, just as they are to be typed at the keyboard. This includes syntactic keywords, variables, constant objects, Scheme expressions, and example programs. An *italic* typeface is used to set off syntax variables in the descriptions of syntactic forms and arguments in the descriptions of procedures. Italics are also used to set off technical terms the first time they appear. In general, names of syntactic forms and procedures are never capitalized, even at the beginning of a sentence. The same is true for syntax variables written in italics.

In the description of a syntactic form or procedure, one or more prototype patterns show the syntactic form or forms or the correct number or numbers of arguments for an application of the procedure. The keyword or procedure name is given in typewriter font, as are parentheses. The remaining pieces of the syntax or arguments are shown in italics, using a name that implies the type of expression or argument expected by the syntactic form or procedure. Ellipses are used to specify zero or more occurrences of a subexpression or argument. For example, `(or` *`expr`* `...)` describes the `or` syntactic form, which has zero or more subexpressions, and `(member` *`obj list`*`)` describes the `member` procedure, which expects two arguments, an object and a list.

A syntax violation occurs if the structure of a syntactic form does not match its prototype. Similarly, an exception with condition type `&assertion` is raised if the number of arguments passed to a standard procedure does not match what it is specified to receive. An exception with condition type `&assertion` is also raised if a standard procedure receives an argument whose type is not the type implied by its name or does not meet other criteria given in the description of the procedure. For example, the prototype for `vector-set!` is

```
(vector-set! vector n obj)
```

and the description says that *`n`* must be an exact nonnegative integer strictly less than the length of *`vector`*. Thus, `vector-set!` must receive three arguments, the first of which must be a vector, the second of which must be an exact nonnegative integer less than the length of the vector, and the third of which may be any Scheme value. Otherwise, an exception with condition type `&assertion` is raised.

In most cases, the type of argument required is obvious, as with *`vector`*, *`obj`*, or *`binary-input-port`*. In others, primarily within the descriptions of numeric routines, abbreviations are used, such as *`int`* for integer, *`exint`* for exact integer, and *`fx`* for fixnum. These abbreviations are explained at the start of the sections containing the affected entries.
