***** DRAFT n August 31, 1989 *****

[

Revised Report on the Algorithmic Language

Scheme

William Clinger
and Jonathan Rees
(*Editors*)

H. Abelson	R. K. Dybvig	C. T. Haynes	G. J. Rozas
N. I. Adams iv	D. P. Friedman	E. Kohlbecker	G. L. Steele Jr.
D. H. Bartley	R. Halstead	D. Oxley	G. J. Sussman
G. Brooks	C. Hanson	K. M. Pitman	M. Wand

Dedicated to the Memory of ALGOL 60

]

Chapter

Summary

The report gives a defining description of the programming language Scheme. Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme.

The introduction offers a brief history of the language and of the report.

The first three chapters present the fundamental ideas of the language and describe the notational conventions used for describing the language and for writing programs in the language.

Chapters 4 and 5 describe the syntax and semantics of expressions, programs, and definitions.

Chapter 6 describes Schemejs built-in procedures, which include all of the languagejs data manipulation and input/output primitives.

Chapter 7 provides a formal syntax for Scheme written in extended BNF, along with a formal denotational semantics.

The report concludes with an example of the use of the language and an alphabetic index.

*** DRAFT***

August 31, 1989

Contents

[

Introduction

]

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

Scheme was one of the first programming languages to incorporate first class procedures as in the lambda calculus, thereby proving the usefulness of static scope rules and block structure in a dynamically typed language. Scheme was the first major dialect of Lisp to distinguish procedures from lambda expressions and symbols, to use a single lexical environment for all variables, and to evaluate the operator position of a procedure call in the same way as an operand position. By relying entirely on procedure calls to express iteration, Scheme emphasized the fact that tail-recursive procedure calls are essentially gotojs that pass arguments. Scheme was the first widely used programming language to embrace first class escape procedures, from which all known sequential control structures can be synthesized. More recently, building upon the design of generic arithmetic in Common Lisp, Scheme introduced the concept of exact and inexact numbers.

Background

The first description of Scheme was written in 1975 [51]. A revised report [46] appeared in 1978, which described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler [43]. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University [31, 24, 10]. An introductory computer science textbook using Scheme was published in 1984 [1].

As Scheme became more widespread, local dialects began to diverge until students and researchers occasionally found it difficult to understand code written at other sites. Fifteen representatives of the major implementations of Scheme therefore met in October 1984 to work toward a better and more widely accepted standard for Scheme. Their report [4] was published at MIT and Indiana University in the summer of 1985. Another round of revision took place in the spring of 1986 [33]. The present report reflects further revisions agreed upon in a meeting that preceded the 1988 ACM Conference on Lisp and Functional Programming and in subsequent electronic mail.

Acknowledgements

We would like to thank the following people for their help: Alan Bawden, Michael Blair, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Robert Hieb, Paul Hudak, Richard Kelsey, Morry Katz, Chris Lindblad, Mark Meyer, Jim Miller, Jim Philbin, John Ramsdell, Mike Shaff, Jonathan Shapiro, Julie Sussman, Perry Wagle, Daniel Weise, and Henry Wu. We thank Carol Fessenden, Daniel Friedman, and Christopher Haynes for permission to use text from the Scheme 311 version 4 reference manual. We thank Texas Instruments, Inc. for permission to use text from the *TI Scheme Language Reference Manual*. We gladly acknowledge the influence of manuals for MIT Scheme, T, Scheme 84, Common Lisp, and Algol 60.

We also thank Betty Dexter for the extreme effort she put into setting this report in TEX, and Donald Knuth for designing the program that caused her troubles.

The Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, the Computer Science Department of Indiana University, and the Computer and Information Sciences Department of the University of Oregon supported the preparation of this report. Support for the MIT work was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505. Support for the Indiana University work was provided by NSF grants NCS 83-04567 and NCS 83-03325.

[

Description of the language

]

Chapter 1

Overview of Scheme

1 Semantics

This section gives an overview of Schemejs semantics. A detailed informal semantics is the subject of chapters 3 through 6. For reference purposes, section 7.2 provides a formal semantics of Scheme.

Following Algol, Scheme is a statically scoped programming language. 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 values (also called objectsindexfile(index-entry "object" "rm" main 2)) rather than with variables. (Some authors refer to languages with latent types as weakly typed or dynamically typed languages.) Other languages with latent types are APL, Snobol, and other dialects of Lisp. Languages with manifest types (sometimes referred to as strongly typed or statically typed languages) include Algol 60, Pascal, and C.

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 APL and other Lisp dialects.

Implementations of Scheme are required to 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 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 procedures are 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 and ML.

One distinguishing feature of Scheme is that continuations, which in most other languages only operate behind the scenes, also have kfirst-classl status. Continuations are useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines. See section 6.9.

Arguments to Scheme procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not. ML, C, and APL are three other languages that always pass arguments by value. 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.

Schemejs model of arithmetic is designed to remain as independent as possible of the particular ways in which numbers are represented within a computer. In Scheme, every integer is a rational number, every rational is a real, and every real is a complex

number. Thus the distinction between integer and real arithmetic, so important to many programming languages, does not appear in Scheme. In its place is a distinction between exact arithmetic, which corresponds to the mathematical ideal, and inexact arithmetic on approximations. As in Common Lisp, exact arithmetic is not limited to integers.

2 Syntax

Scheme, like most dialects of Lisp, employs a fully parenthesized prefix notation for programs and (other) data; the grammar of Scheme generates a sublanguage of the language used for data. An important consequence of this simple, uniform representation is the susceptibility of Scheme programs and data to uniform treatment by other Scheme programs.

The indexfile(index-entry "read" "tt" aux 2)`read` procedure performs syntactic as well as lexical decomposition of the data it reads. The indexfile(index-entry "read" "tt" aux 2)`read` procedure parses its input as data (section 7.1.2), not as program.

The formal syntax of Scheme is described in section 7.1.

3 Notation and terminology

3.1 Essential and non-essential features

It is required that every implementation of Scheme support features that are marked as being indexfile(index-entry "essential" "rm" main 2)*essential*. Features not explicitly marked as essential are not essential. Implementations are free to omit non-essential features of Scheme or to add extensions, provided the extensions are not in conflict with the language reported here. In particular, implementations must support portable code by providing a syntactic mode that preempts no lexical conventions of this report and reserves no identifiers other than those listed as syntactic keywords in section 2.1.

3.2 Error situations and unspecified behavior

indexfile(index-entry "error" "rm" main 2) When speaking of an error situation, this report uses the phrase kan error is signalledl to indicate that implementations must detect and report the error. If such wording does not appear in the discussion of an error, then implementations are not required to detect or report the error, though they are encouraged to do so. An error situation that implementations are not required to detect is usually referred to simply as kan error.l

For example, it is an error for a procedure to be passed an argument that the procedure is not explicitly specified to handle, even though such domain errors are seldom mentioned in this report. Implementations may extend a procedurejs domain of definition to include such arguments.

This report uses the phrase kmay report a violation of an implementation restrictionl to indicate circumstances under which an implementation is permitted to report that it is unable to continue execution of a correct program because of some restriction imposed by the implementation. Implementation restrictions are of course discouraged, but implementations are encouraged to report violations of implementation restrictions.indexfile(index-entry "implementation restriction" "rm" main 2)

For example, an implementation may report a violation of an implementation restriction if it does not have enough storage to run a program.

If the value of an expression is said to be kunspecified,l then the expression must evaluate to some object without signalling an error, but the value depends on the implementation; this report explicitly does not say what value should be returned. indexfile(index-entry "unspecified" "rm" main 2)

3.3 Entry format

Chapters 4 and 6 are organized into entries. Each entry describes one language feature or a group of related features, where a feature is either a syntactic construct or a built-in procedure. An entry begins with one or more header lines of the form

i`=13=0.5em *template* essential *category*if the feature is an essential feature, or simplyi`=13=0.5em *template* *category*if the feature is not an essential feature.If *category* is ksyntaxl, the entry describes an expression type, and the header line gives the syntax of the expression type. Components of expressions are designated by syntactic variables, which are written using angle brackets, for example, ** expression, **variable. Syntactic variables should be understood to denote segments of program text; for example, **expression stands for any string of characters which is a syntactically valid expression. The notation

**thing ?

indicates zero or more occurrences of a **thing, and

**thing **thing ?

indicates one or more occurrences of a **thing.If *category* is kprocedurel, then the entry describes a procedure, and the header line gives a template for a call to the procedure. Argument names in the template are *italicized*. Thus the header linei`=13=0.5em (vector-ref *vector k*) essential procedureindicates that the essential built-in procedure vector-ref takes two arguments, a vector *vector* and an exact non-negative integer *k* (see below). The header linesi`=13=0.5em (make-vector *k*) essential procedure i`=13=0.5em (make-vector *k fill*) procedureindicate that in all implementations, the make-vector procedure must be defined to take one argument, and some implementations will extend it to take two arguments. It is an error for an operation to be presented with an argument that it is not specified to handle. For succinctness, we follow the convention that if an argument name is also the name of a type listed in section