WPI-CS-TR-05-07 March 2005
amended, 29 October 2009

Revised^{-1} Report on the Kernel Programming Language

(partial draft of 29 October 2009)

John N. Shutt
`jshutt@cs.wpi.edu`
`http://www.cs.wpi.edu/~jshutt/`
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609

October 29, 2009

Abstract

This report defines the Kernel programming language and documents its design. Kernel is a statically scoped and properly tail-recursive dialect of Lisp, descended from Scheme. It is designed to be simpler and more general than Scheme, with an exceptionally clear, simple, and versatile semantics, only one way to form compound expressions, and no inessential restrictions on the power of that one compound form. Imperative, functional, and message-passing programming styles (to name a few) may be conveniently expressed in Kernel.

All manipulable entities in Kernel are first-class objects. The primary means of computation are *operatives*, which are statically scoped combiners that act on their unevaluated operands; operatives subsume the roles handled in most modern Lisps by special forms and macros. *Applicatives* are combiners that act on their evaluated arguments, hence are roughly equivalent to Scheme procedures; but an applicative is merely a facilitator to computation, inducing evaluation of operands for an underlying operative.

Computer Science Technical Report Series
Worcester Polytechnic Institute
Computer Science Department
100 Institute Road, Worcester, Massachusetts, 01609-2280

Contents

0 Introduction

The Kernel programming language is a statically scoped and properly tail-recursive dialect of Lisp, descended from Scheme. It is designed to be simpler and more general than Scheme, with an exceptionally clear, simple, and versatile semantics, only one way to form compound expressions, and no inessential restrictions on the power of that one compound form. Imperative, functional, and message-passing programming styles (to name a few) may be conveniently expressed in Kernel.

An important property of Kernel is that all manipulable entities in Kernel are first-class objects. In particular, Kernel has no second-class combiners; instead, the roles of special forms and macros are subsumed by *operatives*, which are first-class, statically scoped combiners that act directly on their unevaluated operands.

Kernel also has a second type of combiners, *applicatives*, which act on their evaluated arguments. Applicatives are roughly equivalent to Scheme procedures. However, an applicative is nothing more than a wrapper to induce operand evaluation, around an underlying operative (or, in principle, around another applicative, though that isn't usually done); applicatives themselves are mere facilitators to computation.

This report describes and defines the Kernel language. It specifies the minimal criteria necessary for a language processor to qualify as an implementation of Kernel, criteria for an implementation of Kernel to qualify as supporting or excluding an optional module of the language; and criteria for an implementation of Kernel to qualify as comprehensive and/or robust; and it also documents the derivation of Kernel's design from basic principles.

The remainder of Section 0 discusses Kernel design principles, and briefly explains the status of the report itself. Section 1 provides an overview of the language, and describes conventions used for describing the language. Section 2 describes the lexemes used for writing programs in the language. Section 3 explains basic semantic elements of Kernel, notably the Kernel evaluator algorithm. Sections 4–15 describe the various modules exhibited by Kernel's ground environment. Section 16 provides a formal syntax and semantics for Kernel. Appendix A summarizes past, and suggests possible future, evolution of Kernel. Appendix B discusses first-class objects in depth. Appendix C discusses how formal calculi supporting fexprs can avoid a theoretical pitfall described by Mitchell Wand (in [Wa98]). The report concludes with a list of references and alphabetical index.

0.1 The design of Kernel

The material in this section is not technical. It addresses the underlying purposes of the Kernel design, language, and report.

0.1.1 Attitude toward language design

Kernel is meant to be a pure articulation of certain design principles — what [Ka93] calls a "crystalization of style". The design principles themselves are the subject of §0.1.2, below.

It is a common fate of such crystalization languages that, over the course of years and decades, their pure style is gradually compromised in the name of various practical concerns such as runtime efficiency. The Kernel design posits that the value of a crystalization language lies in its purity — a sort of resonance, that drops off precipitously when impurities are introduced.[1]

From this supposition, it follows that crystalization of style can only be fully effective if the pure style is one that can be reconciled with practical concerns *without compromise*, neither to the style nor to the practicalities. We claim that the Kernel language model is a pure style of this kind, i.e., one that *needn't* be compromised. Embracing this claim in the language design process means, directly, that we should focus entirely on pure articulation of the style; and indirectly, that when we find ourselves being led into compromise, we must conclude that we have strayed from the pure style.

The latter principle —compromise as symptom of impurity— implies that pursuit of the pure style requires patient willingness to go back and correct missteps in the design; but it offers compounded long-term benefits in exchange, as we expect each uncorrected compromise would degrade language potency and lead to further compromises thereafter, i.e., cascading degradation. From our basic claim that the pure style can be reconciled without compromise, it follows that any given facet of the language design will eventually be right and never require further adjustment (up to our choice of paradigm/pure style, anyway). The empirical perception of software design tinkering as an endless process is, in this context, an artifact of founding one's software design on a programming platform that already contains compromises that are not open for reassessment.

If it isn't clear what constitutes a compromise, then even if compromise *can* be avoided, there is little chance that it *will* be avoided. Therefore, as this report defines the Kernel language, it also extensively documents the motivations for the design, from high-level principles that contribute to the pure style (§0.1.2), to the reasons for low-level details of types and features. This motive documentation clarifies the extent, and limitations, of the influence of high-level principles on the entire language design, facilitating maximization of that influence in both the current and future revisions of the report. (We expect the benefits of motive documentation to be cumulative, so that the documentation effort is always worthwhile, even though some motives will be overlooked because they are too subtle to recognize, or too newly formed, or because they are simply lost in the crowd.)

Kernel is expected to form the core of language implementations. It is not, in its

[1]This is a minimal supposition. Although the author further suspects that abstractive power is a prominent beneficiary of this resonance effect, the Kernel supposition re purity does not depend on that suspicion. (A formal criterion for abstractive power is proposed in [Sh08]; for a general discussion of abstractive power, see [Sh09, §1.1].)

current revision, a full general-purpose language (what is often —lamentably, from Kernel's perspective— called a "full-featured" language); nor does it particularly aspire to become such a language in any particular time-frame, although it does seek to evolve in that direction. Pressure to provide additional functionality *promptly* is a significant vector for compromise, and so cannot be reconciled with the Kernel design's no-compromise policy. Each possible addition to the language must be thoroughly vetted for subtle inconsistency with the design principles, or the design principles cannot survive. In particular, compatibility with implemented extensions to one revision of the Kernel report must not be allowed to skew the decisions on whether to absorb those extensions into a later revision of the report.

Outside of the current section (§0.1) and the appendices (A–C), design discussion is isolated from the main text in indented blocks of text headed *Rationale:*, thus:

> *Rationale:*
> While design discussions may contain valuable technical insights, isolating them from the main text gives the reader due notice of possible subjective/editorial content.

0.1.2 Principles of language design

> 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.
>
> — [Sp+07, p. 3] (the *R6RS*);
> [KeClRe98, p. 2] (the *R5RS*);
> [ClRe91b, p. 2] (the *R4RS*);
> [ReCl86, p. 2] (the *R3RS*)

The above sentence declares the basic design philosophy of the Scheme programming language. It is also the starting point for the design of the Kernel programming language, which attempts to carry the philosophy further, creating a language still smaller, stronger, and more general than Scheme. This abstract strategy would be difficult to apply directly to specific tactical design decisions (as called for in §0.1.1), so Kernel refines it with more concrete guidelines.

G1 [uniformity] Special cases should be eliminated from as many aspects of the language as the inherent semantics allow.

- *G1a* [object status] All manipulable entities should be first-class objects (see Appendix B).
- *G1b* [extensibility] Programmer-defined facilities should be able to duplicate all the capabilities and properties of built-in facilities.

G2 [functionality] The language primitives should be capable of implementing all the advanced features of the *R5RS* (such as `dynamic-wind`, here in §7.2.5; or, for that matter, `lambda`, here in §5.3.2).

G3 [usability] Dangerous computation behaviors (e.g., hygiene violations[2]), while permitted on general principle, should be difficult to program *by accident.*

Guideline *G3* was formulated specifically to protect the ideal of 'removing weaknesses and restrictions' from devolving into mere amorphism. It has proven to be a particularly useful heuristic in practice, being explicitly invoked in the report more often (at last count) than all the other guidelines put together.

G4 [encapsulation] The degree of encapsulation of a type should be at the discretion of its designer.

Type encapsulation is in itself a facilitator of *G1a*, because it allows the programmer to freely intermix objects of a new type with objects of other types without fear of losing track of the types. (Cf. encapsulation of type *promise*, §9.) *Selective* encapsulation is instrumental to *G3*, by directly modulating dangerous behaviors. (Cf. encapsulation of type *environment*, §4.8, §6.8.2.) Selective encapsulation is complementary to *G1a*, by allowing the observable shape of a type to be matched to whatever abstract value domain the type is intended to represent. (On the connection between domain representation and first-class-ness, see Appendix B.) The assignment of these selective capabilities to the type designer is based on the theory that the designer of a type is *responsible* for its utility, and so should have the technical ability to address that responsibility.

The Scheme reports also explicitly identify efficiency as a concern. It is a tenet of Kernel design philosophy that

G5 [efficiency] Efficient implementation should *follow* naturally from an elegant language design, without any compromise of elegance being made to achieve it.

This is a refinement of the no-compromise policy described above in §0.1.1. It really has two aspects — that elegance *shouldn't* be compromised for efficiency; and that it *doesn't have to be*, because efficiency will be a natural beneficiary of design decisions made for other reasons. A key example of the latter, efficiency benefiting from other factors, is the ability to restrict mutation of ancestral environments, which is motivated in the Kernel design by *G3* (see for example §6.8.2; a more detailed discussion occurs in [Sh09, §5.3]), but without which Kernel would be substantially unoptimizable since Kernel source expressions have no fixed semantics, i.e., no special forms.

0.2 About the report

Permission to copy this report

This report is intended to belong to the programming community, and so permission is granted to copy it in whole or in part without fee.

[2] Kernel hygiene is discussed in [Sh09, Ch. 5].

Acknowledgements

This report borrows heavily from the *R5RS* for parts of its overall structure, and of its text. The *R5RS*, in turn, borrows heavily from the *R4RS*, and so on. In a few instances, the *R6RS* has also been consulted. The current author is indebted to everyone who has contributed to Scheme and its descriptions over the years —many of whom are named in the Introduction of the *R5RS*— notably to Gerry Sussman and Guy Steele for creating the Scheme language to begin with ([SusSte75]), and to Hal Abelson and Gerry Sussman for writing the book on it ([AbSu96]).

The current author has also benefited from in-person discussions with a number of individuals, many of them members of the NEPLS community;[3] and from feedback and suggestions on the report, ranging from low-level mechanics to high-level design principles and concepts, via email from others. To all these, too, the author is indebted, for broad perspective as well as specific ideas. Where specific ideas have been absorbed into elements of the language design, the sources are noted in the rationale discussions for those elements. Consultees in these capacities include —but are certainly not limited to— Alan Bawden, John Cowan, Herman Ehrenburg, Mike Gennert, Shriram Krishnamurthi, Jim Miller, Marijn Schouten, and Mitchell Wand.

1 Overview

1.1 Semantics

This subsection gives an overview of Kernel's semantics. A detailed informal semantics is the subject of §§3–15. For reference purposes, §16.2 provides a formal semantics of the Kernel evaluator. The main semantic differences from Scheme are summarized in §A.1.

Following Algol and, especially, Scheme, Kernel is a statically scoped programming language. The body of a compound combiner is always evaluated in the static environment where the combiner was constructed (rather than in the dynamic environment where it is called). Ordinarily, each use of a variable is associated with a lexically apparent binding of that variable. In the general case, the evaluation environment may be explicitly computed, making it impossible to identify the binding of a variable without actually evaluating the program. However, once the evaluation environment of an expression is known, scoping proceeds statically within the expression.

Kernel has latent as opposed to manifest types. Types are associated with values (also called objects) rather than with variables. Other languages with latent types include APL, Snobol, and other dialects of Lisp. Languages with manifest types include Algol 60, Pascal, and C.

[3] `http://www.nepls.org/`

All objects created in the course of a Kernel computation have unlimited extent. No Kernel object is ever destroyed. This abstract behavior need not cause actual implementations to rapidly exhaust their storage space, because without violating the required abstract behavior, they can 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 Kernel 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 combiner. Thus with a properly tail-recursive implementation, iteration can be expressed using the ordinary combiner-call mechanics, so that special iteration constructs are useful only as syntactic sugar. See §3.10.

Arguments to Kernel applicatives are passed by value, meaning that the operand expressions are evaluated before the applicative gains control, regardless of whether or not the applicative needs the result of the evaluation. Scheme, ML, C, and APL are languages that always pass procedure 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 is not evaluated unless its value is needed by the procedure.

Operands to Kernel operatives are passed unevaluated, together in each call with the dynamic environment from which the call is made. The operative therefore has complete control over operand evaluation, if any. Consequently, operatives are readily capable of replacing both special forms and macros. A similar facility was provided in LISP 1.5 ([McC+62]), and later in Maclisp ([Pi83]), under the name `FEXPR`s; but historically, `FEXPR`s were dynamically scoped, and thus enjoyed neither the stability that Kernel operatives derive from static scoping,[4] nor the power that Kernel operatives accrue through simultaneous access to both static and dynamic environments.

Kernel applicatives are first-class objects. An essentially unrestricted range of applicatives can be dynamically constructed, stored in data structures, returned as results of combiners, and so on. Other languages with these properties include Scheme, Common Lisp, and ML. However, Kernel operatives are also first-class objects, which is not true of the other languages mentioned.

Continuations, which in most other languages only operate behind the scenes — Scheme being a notable exception— are first-class objects in Kernel. Continuations are useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines. Kernel continuations are also used in exception-handling. See §7.

[4]Stability of this sort manifests formally as strength of equational theory. There has been some popular misconception, based on overgeneralization of the formal result in Mitchell Wand's (broadly titled) 1998 paper "The Theory of Fexprs is Trivial", [Wa98], that all calculi of fexprs necessarily have trivial equational theories. Actually, Wand's formal result applies only to calculi constructed in a certain way; see Appendix C.

Kernel environments are first-class objects as well. Their first-class status is a *sine qua non* of Kernel-style operatives, because it allows operatives to be statically scoped yet readily access their dynamic environments for controlled operand evaluation. Another common use of first-class environments is for the explicit exportation of specific sets of features (see §6.8).

All objects in Kernel are capable of being evaluated. Objects that are neither symbols nor pairs evaluate to themselves, regardless of the environment. Consequently, it is possible to construct combiner-calls that can be evaluated even in an empty environment (i.e., an environment exhibiting no bindings), by building the combiner itself into the expression rather than using a symbol for it. (See for example §5.3.2.)

Kernel uses explicit evaluation, rather than quotation, as a preferred model for specifying the selective evaluation of subexpressions. That is, the facilities of the language are primarily arranged to specify conservatively which subexpressions should be evaluated, rather than assuming by default that all subexpressions are to be evaluated and introducing quasiquotation operators to suppress evaluation in particular cases. To facilitate explicit evaluation, Kernel allows applicatives and operatives to be freely converted into each other by adding and removing wrappers that induce operand evaluation. Kernel has no standard quasiquotation facilities.[5] Most Lisps, including Scheme, use quasiquotation. MetaML ([Ta99]) is a prominent example of a non-Lisp language using quasiquotation.

Kernel's model of arithmetic is designed to provide useful access to different ways of representing numbers within a computer, while minimizing intrusion of those representation choices into the way arithmetic is performed by a program. Every integer is a rational, every rational is a real, and every real is a complex. Thus the distinction between integer and real arithmetic, so important to many programming languages, does not occur. In its place is a distinction between exact arithmetic, which corresponds to the mathematical ideal, and inexact arithmetic on approximations. Exact arithmetic is not limited to integers. Inexact real numbers have associated exact upper and lower bounds, which may be plus and minus infinity if the implementation provides no bounding information on the approximation.

1.2 Syntax

Kernel, like most dialects of Lisp, employs a fully parenthesized prefix notation for

[5]It would be technically straightforward to implement quasiquotation using standard Kernel facilities; the author simply maintains that doing so would be a serious mistake. Quasiquotation is by nature an implicit-evaluation technology, and as such can only compromise the lucidity of the explicit-evaluation strategy of Kernel. Moreover, as emerges empirically from the investigation of hygiene in [Sh09, Ch. 5], use of quotation in Kernel greatly increases the likelihood of unintended hygiene violations, contrary to Guideline *G3* of §0.1.2. Explicit evaluation so permeates the Kernel design that it has been considered for addition to the list of design Guidelines in §0.1.2; but thus far, there has been no clear need to further complicate the Guidelines with it since it has not come up in any particular rationale discussion in the report.

data. As noted above in §1.1, all objects are evaluable; hence in theory all data objects are programs. An important consequence of this simple, uniform representation is the susceptibility of Kernel programs and data to uniform treatment by other Kernel programs.

The ***read*** applicative performs syntactic as well as lexical decomposition of the data it reads; see §15.1.7. Since all objects are evaluable, there is no such thing as a syntactically invalid program. Once a program text has been converted to an object (as by ***read***), any further errors —such as an unbound variable, or a combination whose car doesn't evaluate to a combiner— are semantic errors, and consequently will not occur until and unless the object is evaluated.

Since all objects are potentially programs, the terms "expression" and "object" are used interchangeably throughout the report.

The formal syntax of Kernel is described in §16.1. Note in particular that the syntax "`(`a `. (`$\cdots$`))`" is interchangeable with "`(`a $\cdots$`)`".

1.3 Notation and terminology

1.3.1 Evaluable expressions

A symbol to be evaluated is a *variable* (occasionally called a *symbolic* variable to distinguish it from the *keyed* variable devices of §§10–11).

A pair to be evaluated is a *combination*. The unevaluated car of the pair is an *operator*; its unevaluated cdr is an *operand tree*; and in the usual case that the operand tree is a list, any elements of that list are *operands*. In the common case that all the operands are evaluated, and all other actions use the results rather than the operands themselves, the results of evaluating the operands are *arguments*. The result of evaluating the operator is (if type-correct) a *combiner*, because it specifies how to evaluate the combination. A combiner that acts directly on its operands is an *operative* (or in full, an *operative combiner*). A combiner that acts only on its arguments is an *applicative* (in full an *applicative combiner*), because the ***apply*** combiner (§5.5.1) requires an applicative rather than an operative.

Rationale:

Most of these basic terms are adopted from [AbSu96, §1.1]; those not found there are *operand tree*, *combiner*, *applicative*, and *operative*.

The term *procedure* is avoided by Kernel because its use in the literature is ambiguous, meaning either what is here called an applicative (in discussions of Scheme), or what is here called a combiner (in discussions involving both applicatives and operatives). There is an adjective *call-by-text* in the literature meaning what is here called operative; but the Kernel term *applicative* has no equivalent of the form *call-by-X*. Adjectives *call-by-X* are used in general to specify when the operands are evaluated to arguments, as *call-by-value* (eager) or *call-by-name* (lazy) ([CrFe91]); but *applicative* is intended to mean only that the combiner depends on the arguments, without any implication as to when the arguments are computed.

The terms *combiner* and *operative* were favored for their mnemonic value. *Applicative* was chosen for its mnemonic semi-symmetry with *operative* (see also [Ba78, §2.2.2]); more stringent symmetry would have analogized *operand/operative* with *argument/argumentative*, but the term *argumentative* was too distracting to use.

1.3.2 Modules and features

The features (i.e., bindings) exhibited by the ground environment of Kernel (§3.2) are grouped into modules. Each module contains features supporting a particular aspect of the language, most often an object type or types. Each module may be explicitly specified to *assume* certain other modules. Some modules may be marked as being *optional*; modules not so marked are *required.* A required module never assumes an optional module.

An implementation cannot claim to support a module M unless it both (1) supports all of the features in M, and (2) supports all of the modules assumed by M. Every implementation of Kernel supports all required modules; any system that does not do so is not an implementation of Kernel. An implementation of Kernel is *comprehensive* if it supports all optional modules.

All modules that are supported, and all features that are supported even if the modules containing them are not supported, must conform to their descriptions here. Implementations are permitted to add extensions, provided the extensions are consistent with the language presented here. (See the discussion of encapsulation in §3.4.)

An implementation of Kernel can claim to *exclude* a module M iff ("iff" = "if and only if") it both (1) omits all of the features in M, and (2) does not add any extension that provides a capability provided in the report by M. The latter condition will be clarified in §3.4.

Within each module, features are either *primitive* or *library.* Library features could be implemented using the primitive features of their own module and other modules (possibly including modules not assumed; see, for example, the derivation of ***$binds?***, §6.7.1). Code for implementing library features *in the ground environment* (§3.2) is usually included with their descriptions; that is, given availability of suitable primitives, evaluating the provided code in the ground environment (which Kernel programs are not permitted to do) would mutate the ground environment so that it would exhibit those features. The derivation code is not considered part of the definition of the feature, so implementations are not expected to duplicate the exact behavior of the code. However, the code is meant to conform to the description of the feature, so that in principle it could be used internally by a (non-robust, per §1.3.3) implementation of Kernel.

Rationale:

The purpose of optional modules is to allow aspects of the language to be standardized without requiring them to be implemented in situations where those aspects are irrelevant.

Optionality of modules should be reserved for that purpose: not to make the implementor's life easier, nor because the design of some module is tentative (either it's worth including in the report or it isn't), but because requiring the module simply wouldn't make sense in some of the situations where Kernel might reasonably be implemented. Which modules ought to be optional is thus a reflection of the range of situations toward which the language is targeted.

The assumption relation between modules serves two purposes. It documents dependencies in the language design, and it constrains when an implementation can claim to support a module. The design dependency is considered dominant; when an implementation claims to support a module M, it is asserting its own compliance to the *design* of M, and so design assumptions of M are encompassed within the assertion. When the assumed module is required, the assumption has no immediate significance beyond design documentation — though it might gain significance later if the assumed module were later made optional. If the assumed module were optional but the *assuming* module were required, the assumption would be tantamount to requiring the assumed module, and so we avoid pointless confusion by disallowing this case.

1.3.3 Exceptions to normal computation

Errors

When speaking of an error situation, this report uses the phrase "an error is signaled" to indicate that implementations must detect the error and report it. The specifics of error signaling are described in §7. 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 strongly encouraged to do so. An error situation that implementations are not required to detect is referred to simply as "an error."

For example, it is an error for a combiner to be passed an object (operand or argument) that the combiner is not explicitly specified to handle, even though such domain errors are seldom mentioned in the descriptions of the combiners. In most cases, implementations are permitted to fail to detect and signal such errors.

An implementation of Kernel is *robust* if it signals all errors described in this report for all features that it supports.

In an error situation that is not required to be signaled, implementations that do not signal the error should not provide any deliberately useful behavior: no implemented behavior should detect these situations except to signal them, and the implementor should not choose any facet of the implemented behavior for the sake of non-signaling error behavior.

Conversely, whenever an implementation provides an extension not described in this report (as discussed in §3.4), it is strongly encouraged to provide the extension under a name that is not used for any feature in this report. (In some cases the report may expand the domain of a primitive feature by later introducing a library feature of the same name, as with the two versions of ***$vau*** in §§4.10.3 and 5.3.1. This is not meant to be a paradigm for implementing extensions to the language, but rather to illustrate the derivative nature of the library feature.)

Rationale:

Deliberately useful behavior in a no-signal-required error situation is literally an invitation to write one of the most anti-portable kinds of code: implementation-dependent code that isn't readily identifiable because the nonstandard feature it uses is camouflaged under the name of a standard feature. Of course, all behavior *can* be made use of; the point here is that since it's bad programming practice, the implementor shouldn't be promoting it.

Serious consideration was given to simply requiring *all* errors to be signaled. This extreme measure was not taken only because it was felt implementations should have leeway with which to prevent eager error-checking from becoming burdensome. Notably, the provided derivations of library features in the report are sometimes lax in non-required error signaling — mostly because those derivations also serve as supplementary illustrations of how the features work, and additional error-checking would make the code significantly less readily understandable.

Implementation restrictions

This report uses the phrase "may report a violation of an implementation restriction" 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 required to report violations of implementation restrictions.

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

Rationale:

An implementation restriction pertains to a permissible incapability that should not disqualify the implementation from being "comprehensive"; otherwise, the permissible incapability would be more appropriately treated as an optional module (§1.3.2).

1.3.4 Entry format

Sections 4–15 consist mostly of entries, each entry describing one language feature. Each entry begins with one or more template lines showing supported formats for using the feature.

If the feature is not a combiner, there is just one template, which is simply the name of the object. The template line is followed by an explanation of the feature and its purpose.

Throughout the report, literal text is written in `monospace lettering`. In the template for a feature, the name of the feature appears as literal text. Library feature derivations are also presented as literal text. The object bound to a name (in the environment of interest, most often a Kernel standard environment) is indicated by writing the name in ***`italicized monospace lettering`***. Thus, for example, symbol `cons` is bound in a standard environment to object ***`cons`***.

In the template for an operative, indefinite components are written using angle brackets; for example, ⟨expression⟩, ⟨symbol⟩. In the template for an applicative, indefinite components are *italicized*. In either case, the components always stand for actual objects to be substituted into the template. The description specifies any type restrictions on the components, immediately following the template line(s). Type restrictions pertain to the values that are actually passed to the combiner; thus, type restrictions for an operative refer to the (unevaluated) operands, while type restrictions for an applicative refer to the (evaluated) arguments.

It is an error for a combiner to be presented with an object (operand or argument) that it is not specified to handle. For succinctness, if a component name is also the name of any type defined in any module of this report (§§4–15), optionally followed by one or more digits (e.g., *number2*), then that object must be of the named type. If a component name is the plural of the name of any type defined in any module of this report (e.g., *strings*), then that object must be a list of objects of the named type. Type *list* will be defined in §3.9. Type *object* is understood to include all first-class objects; so component name *objects* implies the same type constraint as component name *list*.

When an entry extends the behavior of a previous entry, both entries refer to each other, but the extending entry does not repeat templates from the earlier entry.

1.3.5 Expression equivalences

The semantics of a language feature are sometimes clarified, or even defined, in its entry by specifying that two expressions are equivalent. For example, the semantics of applicative ***`list*`*** are defined, in §5.2.2, by the equivalences

```
(list* arg1)                ≡  arg1
(list* arg1 . more-args)    ≡  (cons arg1 (list* . more-args))
```

Broadly speaking, the two expressions in such an equivalence are interchangeable in any situation where they would be evaluated; but certain pathological cases are excepted. Interchangeability means that evaluating either of the two expressions *in the same environment*, *starting from the same state of the entire runtime platform*, will have the same consequences — both resultant value (if any) and side-effects. Unless otherwise stated, there are four pathological cases excepted:

- Subexpressions (operands or arguments) are assumed to have the correct types for the combiners on the *left* side of the equivalence (or the first side, if the sides are presented on separate lines). The right/second side of the equivalence may place weaker constraints on the subexpression types, but the equivalence isn't guaranteed under the weaker constraints.

- The expressions themselves are assumed not to be mutated before their structure is used in evaluation. For example, the second equivalence above for ***list**** doesn't cover situations where evaluation of *arg1* causes mutation to *more-args*.

- If the expression evaluations involve subsidiary argument evaluations, then the equivalence only holds if either (1) the argument evaluations don't have side-effects, or (2) the argument evaluations are performed in the same order for both sides of the equivalence. This is significant because, when an expression contains nested combinations, Kernel's eager argument evaluation may prohibit some orderings. For example, in the second equivalence above for ***list****, the arguments to ***list**** on the left may be evaluated in any order at all, while in the nested expression on the right, *arg1* will be evaluated either before all of the other arguments or after all the other arguments.

- If some named subexpression occurs multiple times on one side of an equivalence, evaluating that subexpression multiple times is assumed to produce identically the same object each time, without side-effects from the redundant evaluation. This arises, for example, in an equivalence for the rationale discussion of ***map***, §5.9.1.

If the conditions of a particular equivalence vary from these defaults, the variance should be specified along with the equivalence.

1.3.6 Evaluation examples

The symbol "$\Longrightarrow$" used in program examples should be read "evaluates to". For example,

```
(* 5 8)                              ⟹  40
```

means that the expression `(* 5 8)` evaluates to the expression `40`. Or, more precisely: an object represented externally by the sequence of characters "`(* 5 8)`" evaluates, in a standard environment, to an object represented externally by the sequence of characters "40". See §3.6 for a discussion of external representations of objects.

1.3.7 Naming conventions

By convention, the names of operatives always begin with "`$`". (This was originally meant to be a stylized letter *s*, for ***special form.***)

By convention, the names of combiners that always return a boolean value always end in "?". Such combiners are called predicates.

By convention, the names of combiners whose purpose is to modify previously existing objects always end in "`!`". Such combiners are called mutators. By convention, the value returned by a mutator is inert (§4.5).

By convention, "->" appears within the names of combiners whose purpose is to take an object of one type and return an analogous object of another type. For example, ***continuation->applicative*** (§7.2.5) takes a continuation and returns an applicative that passes its argument list to that continuation.

Rationale:

Latent typing together with clear typing-based naming conventions promote design Guideline *G3* of §0.1.2, that dangerous actions should be permitted but difficult to do by accident. (Cf. the rationale for partitioning of types, §3.5.) Hence, to maximize their design value the conventions should be chosen for quick and accurate recognition. To that end, all the basic conventions use non-alphanumeric characters, as in Scheme and unlike most other Lisps. The Kernel conventions have also been simplified from Scheme by eliminating exceptions; notably, the names of Kernel boolean operations (§6.1) and number comparison predicates (§12) all end with `?`.

Quick and accurate recognition also dictates that the *number* of basic naming conventions should not be increased without genuine need. Experience with an early prototype implementation of Kernel forcefully demonstrated that, while a small number of special forms may simply be learned by rote, and macros may be marginally infrequent enough to handle likewise, a language that grants full first-class status to compound operatives genuinely needs a naming convention to keep track of which combiners are meant to be operative and which applicative.

The uniformity Guideline (*G1* of §0.1.2) is not applied directly to symbolic names of features. The *point* of uniformity is, broadly, that it promotes free interactions between language elements; but the symbolic names of features are targeted at a human audience, and as such their interactions take place within the murky realm of natural language. Discerning just what promotes free interactions within that realm seems scarcely less formidable than full-blown automated natural-language translation; it is, for example, a noted phenomenon in the field of conlanging (the artificial construction of pseudo-natural languages) that excessive superficial uniformity of symbolic names can actually *promote* mistakes[6] — contrary to the usual synergism that, based on their behavior in programming-language semantics, we would expect between properly applied *G1* and *G3*. Given this murkiness, the uniformity Guideline has been judged for the current work to be uninformative, and symbolic names of features are addressed directly through accident-avoidance Guideline *G3* (as above).

Specialized Kernel operations involving a particular type are often given names that contain (or at least strongly suggest) the name of the type involved —in some non-core modules, most or all symbolic names follow this pattern— but the practice is applied

[6]This is a standard criticism of the constructed language Ro, which was begun in 1904. The vocabulary of Ro is arranged taxonomically, building words from sequences of sounds in much the same way that the Dewey Decimal System builds library classification numbers — which, unfortunately, maximizes the likelihood of confusion between words for which distinguishing clues are least likely to be provided by context. (For example, words starting with *bofo-* are colors; red is *bofoc*, orange *bofod*, yellow *bofof*, green *bofog*, and so on.) Ro was sponsored for some years by, among others, Doctor Melvil Dewey, creator of the Dewey Decimal System.

selectively, as judged to best promote accident-avoidance on a case-by-case basis. Type-based names are usually inappropriate to generic operations (for most of which type names would have to be invented anyway),[7] and usually inappropriate to ubiquitous features (whose names the programmer presumably needs no help remembering, and for which type names would create a significant burden of verbosity). Natural languages tend to a similar pattern, with regularity and length being more pervasive in less common vocabulary. At the extreme of the pattern for natural languages, typically the handful of most basic, general-purpose words are the most irregular and shortest, such as English *be*, *do*, *go*, *have*. Lisp also exhibits this extreme case of the pattern: the most basic operators, across the entire Lisp family, are `lambda`, `cons`, `car`, and `cdr`. This also has, one suspects, a psychological feedback effect, akin to the raised dots commonly placed on the `F` and `J` keys of QWERTY keyboards, which serve as an aid to touch typists: language users are cued to the key elements of the language by where terseness and irregularity are most concentrated.

2 Lexemes

This section gives an informal account of the syntax of lexemes used in writing Kernel programs. For a formal syntax of Kernel, see §16.1.

Upper and lower case forms of a letter are never distinguished except within character and string constants. For example, `Foo` is the same identifier as `FOO`, and `#x1AB` is the same number as `#X1ab`.

Rationale:

The *R6RS* breaks from a long-standing Scheme tradition by making identifiers case-sensitive.[8] The Kernel design principles (§0.1.2) are not readily consistent with identifier case-sensitivity. Distinguishing long identifiers by case alone is error-prone, hence contrary to *G3*. Such confusions could be forcibly prevented by requiring that identifiers bound by the same environment must differ by more than case, but that approach would clash with the removal-of-restrictions design philosophy (a clash that would manifest as difficulty in arranging the lookup behavior of multi-parented environments, §3.2). Alternatively, one might require all identifiers to conform to an unambiguous casing policy, which would eliminate any difficulty in arranging multi-parent environment lookup since two identifiers could never differ only by case; this alternative could only be philosophically acceptable if the policy could not be made simpler, and there are exactly two maximally simple policies of this kind: either all letters in identifiers must be lower-case, or all letters in identifiers must be upper-case. Of these two, lower-case would be preferred since it is known to be the less error-prone of the two; but, at least for the current revision of the report, case-insensitivity has been retained from the *R5RS*.

[7]On the desirability of generic operations, see [Sus07].

[8]Prior to the *R6RS*, identifier case-insensitivity had been explicitly specified by every Scheme report back to the *R2RS* ([KeClRe98, §2], [ClRe91b, §2], [ReCl86, §2], [Cl85, §I.1]). The first two Scheme reports apparently have Scheme built on Maclisp, which was case-insensitive on most but not all platforms ([Pi07, Introduction]).

It might be argued that short —say, single-letter— identifiers differing only by case might be mistakenly treated as different. (This differs qualitatively from the confusion induced by long identifiers differing only by case, in that the long-identifier confusion would be due to overlooking particular details, whereas the short-identifier confusion would be due to overlooking a general policy.) Accepting this argument about short identifiers would put case-insensitivity, too, on the wrong side of *G3*. The same policy is used for short and long identifiers, per *G1*; so if the case-insensitive policy is also ruled out, only the one-case policy would remain.

2.1 Identifiers

Most identifiers allowed by other programming languages are also acceptable to Kernel. A sequence of letters, digits, and "extended alphabetic characters" that begins with a character that cannot begin a number is an identifier. In addition, `+` and `-` are identifiers. Here are some examples of identifiers:

```
$lambda                              q
list->vector                         soup
+                                    V17a
<=?                                  a34kTMNs
the-word-recursion-has-many-meanings
```

Extended alphabetic characters may be used within identifiers as if they were letters. The following are extended alphabetic characters:

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

See §16.1.3 for a formal syntax of identifiers.

Identifiers in Kernel are external representations of symbols.

2.2 Whitespace and comments

Whitespace characters are spaces and newlines. (Implementations may provide additional whitespace characters, such as tab or page break.) Whitespace is used for improved readability, and as necessary to separate tokens from each other, a token being an indivisible lexical unit such as an identifier or number; but whitespace is otherwise insignificant. Whitespace may occur between any two tokens, but not within a token. Whitespace may also occur inside a string, where it is significant.

A semicolon (`;`) indicates the start of a comment. The comment continues to the end of the line on which the semicolon appears. Comments are invisible to Kernel, but the end of the line is visible as whitespace. This prevents a comment from appearing in the middle of an identifier or number.

```
;;; The FACT procedure computes the factorial
;;; of a non-negative integer.
```

```
($define! fact
   ($lambda (n)
      ($if (=? n 0)
           1       ;Base case: return 1
           (* n (fact (- n 1))))))
```

2.3 Other notations

For a description of the notations used for numbers, see §12.

`. + -` These are used in numbers, and may also occur anywhere in an identifier except as the first character. A delimited plus or minus sign by itself is also an identifier. A delimited period (not occurring within a number or identifier) is used in the notation for pairs (§4.6).

`( )` Parentheses are used to delimit lists (§4.6).

`"` The double-quote character is used to delimit strings (§13).

`\` Backslash is used in the syntax for character constants (§14) and as an escape character within string constants (§13).

`#` Sharp sign is used for a variety of purposes depending on the character that immediately follows it:

`#t #f` These are the boolean constants (§4.1).

`#ignore` This is the ignore constant (§4.8).

`#inert` This is the inert constant (§4.5).

`#\` This introduces a character constant (§14).

`#e #i #b #o #d #x` These are used in the notation for numbers (§12.4).

`[ ] { } |` Left and right square brackets and curly braces and vertical bar are reserved for possible future use in the language.

`' ` , ,@` Single-quote character, backquote character, comma, and the sequence comma at-sign are reserved as illegal lexemes in Kernel, to avoid confusion with their use in other Lisps for quasiquotation.

3 Basic concepts

This section describes basic semantic elements of Kernel, notably its evaluator algorithm.

3.1 References

An object may, depending on its type, contain one or more *references* to other objects. All references are given initial referents (choices of object referred to) when the object is created. The referent of a reference may sometimes be set at some time after object creation; the reference refers thereafter to the new referent, until the next time, if any, that the referent of that reference is set. The act of setting the referent of a reference after object creation constitutes a mutation of the object containing the reference (§3.8), regardless of whether the new referent is the same as the old referent.

3.2 Environments

A *binding* is an association of a symbol with a reference to a value. The symbol is said to be *bound to* the value, or the value *bound by* the symbol.

An environment consists of a set of bindings, and a list of zero or more references to other environments called its *parents*. The transitive closure of the *parent* relation is *ancestor*; the reflexive transitive closure is *improper ancestor*.

When a symbol s is looked up in an environment e, a depth-first search is conducted on the improper ancestors of e, taking the parents in listed order, to find a binding of s to some value. The first such value found is returned as the result of the search; if no such value is found, an error is signaled. (See also §4.8.4.)

The bindings that are actually part of an environment are *local* to it (they *locally bind* their symbols to values). Those bindings that may provide the result of a symbol lookup in an environment are *visible* in it (*visibly bind*, or simply *bind*, symbols to values). An environment *contains* its local bindings, and *exhibits* its visible bindings.

One environment is assumed to exist: the *ground* environment. The ground environment exhibits all the language features that the implementation supports (minimally, all features of all required modules in this report). Implementations are expressly forbidden to provide any means by which a Kernel program could capture (e.g., §6.7.2) or attempt to mutate (§§3.8, 4.9) any improper ancestor of the ground environment.

An environment is *standard* if it is a child of the ground environment, and contains no local bindings.

Rationale:

It is both formally convenient to assume, and practically convenient to provide, a single ancestor for all standard environments: because the ground environment cannot be mutated, it need only be constructed once, and standard environments can then be created at will (§6.7.3) with, in principle, no more overhead than an ordinary compound combiner call.

The ability to construct an environment with multiple parents (§4.8.4) could be used to merge the exported facilities of several separate modules, and make them all visible to the internal environment of another module that depends on them (similarly to Java's `import`).

Kernel symbol lookup uses depth-first search because (1) depth-first search respects the encapsulation of the parent environments, not requiring the child to know its parents' ancestry, and (2) exactly because depth-first search respects the encapsulation of the parents, it is the *simplest* algorithm for searching a tree, in accordance with Kernel's design philosophy.

3.3 The evaluator

The Kernel evaluator algorithm refers to the following data types:

- An *environment* maps symbols to values. (§§3.2, 4.8)
- A *pair* contains (i.e., references) two values, called its car and cdr. (§4.6)
- An *operative* takes an object and an environment as input, and produces an object as output. (§4.10)
- An *applicative* is simply a wrapper around another combiner, called its *underlying combiner*. (§4.10)

The environment passed to an operative is the *dynamic environment* in which the combiner was invoked (as opposed to the *static environment* in which, if compound, it was constructed). The purpose of passing the dynamic environment to the operative is so that, if it chooses to evaluate any or all of its operands, it may use the dynamic environment to do so.

The Kernel evaluator uses the following algorithm to evaluate an object o in an environment e. The algorithm is simplified here only in that it doesn't mention continuations (§7). Top-level expressions, such as those input to an interactive Kernel interpreter, are evaluated in an initially standard environment ('initially' because, as evaluations proceed, it may be mutated).

1. If o isn't a symbol and isn't a pair, the evaluator returns o.
2. If o is a symbol, the evaluator returns the value bound by o in e.
3. Otherwise, o is a pair. Let a and d be the car and cdr of o. a is evaluated in e; call its result f.
   (a) If f is an operative, then f is called with input object d and input environment e, and the result is the result of the combination.
   (b) Otherwise, f must be an applicative, and d must be a list (§3.9). Let f' be the underlying combiner of f. The elements of list d are evaluated in e; let d' be a list of their values. The cons of f' with d' is evaluated in e, and the result is the result of the combination.

In Step 2, if o is not bound in e, an error is signaled. In Step 3, if f is neither an applicative nor an operative, an error is signaled. In Step 3b, if d is not a list, an error is signaled.

In Step 3b, the operands d may be evaluated in any order (provided the results end up back in the correct order in d'); and the case of cyclic operand lists will be addressed in §3.9. The evaluated arguments list d' is constructed using mutable pairs (although any of the elements of the list could be immutable; mutation will be addressed in §3.8). The cons of f' with d' is evaluated *as a tail context* (§3.10).

It is a noteworthy property of this evaluation algorithm that operatives do all the work, while applicatives, which are a fairly close analog of Scheme procedures, are nothing more than wrappers, doing of themselves no direct work at all.

Rationale:

Consideration was given to the alternative of constructing argument list d' with *immutable* pairs. This, however, would have introduced an irregularity into the properties of operand trees as seen by compound operatives (on the uniformity of which, see under `$define!`, §4.9.1); and also would have gratuitously introduced an otherwise unavailable language capability (see the rationale for `copy-es-immutable`, §4.7.2).

Eager argument evaluation, and sequencing of expressions in the body of a compound operative, are ubiquitous devices for the programmer to impose order on subsidiary computations. (See §5.1.1.) The only similarly ubiquitous device for expressing indifference to order of subsidiary computations is the evaluation of multiple arguments to an applicative. Thus, not specifying the order of argument evaluation is key to the expressiveness of the language. (See also the rationale for §3.9.)

3.4 Encapsulation of types

This report specifies that certain of the object types presented here are "encapsulated". This means, in essence, that implementations are not permitted to support any operation on objects of that type that wouldn't be supported by a comprehensive implementation without extensions.

The concept of encapsulation is rather slippery, especially when dealing with a language as flexible as Kernel. Some further amplification is therefore appropriate as to what constitutes a violation of encapsulation. The following is an exhaustive list of kinds of features that do not violate the required encapsulation of the language. (Here, features are counted regardless of whether they are provided by the ground environment or elsewhere. To emphasize: any system that supports an operation not included in this list is not an implementation of Kernel.)

1. A feature is permitted if it is presented here, and provides only facilities presented here.

2. A feature is permitted if it is not presented here, and does not involve any encapsulated object type presented here. Referencing an object, as is done when

storing it in a data structure (see §3.1 and Appendix B), does not "involve" the referent's type, only the fact that the referent is first-class; so mere reference is always permitted. (Note that extension types must also satisfy the partitioning requirements of §3.5.)

3. A feature is permitted if it could in principle be implemented using only features covered under (1) and (2), without such implementation causing modification of any non-error behavior, nor any required error-signaling behavior, of any feature covered under (1). The features that would be used to implement it needn't be included in the actual implementation; however, even if the features that would be used are themselves omitted, the implementation cannot claim to *exclude* the module that contains them (§1.3.2).

4. **Subtypes.** A type predicate is permitted for a new subtype (that is, a subtype not described in this report) of an encapsulated type, provided that none of the features described in this report, behaving as described in this report, constructs any object of the new subtype. All the operations on the encapsulated supertype must behave normally (as specified in this report) on objects of the new subtype; but for purposes of permitting additional operations, the new subtype is treated as a *separate type*.

An important example of encapsulation is that of compound operatives (§4.10.3). Data type *operative* is encapsulated, and the features described in this report do not provide a way to extract the *static environment* of a compound operative — nor do they even provide any way to determine whether a given operative is compound. So an implementation of Kernel is not permitted to provide facilities to do either of these things. Moreover, even if an implementation exercises its right to provide a subtype of compound operatives whose static environments can be determined, none of the primitive features described in this report will construct compound operatives of the new subtype.

The prohibition against extracting static environments is commonly used to create combiners with local state that only they can access. See for example the derivation code for promises in §9.1.3. Permitting an extension that extracts the static environments of arbitrary compound combiners would thus undermine a key information-hiding device of the language.[9]

Rationale:

Encapsulation of built-in types is the base case of Guideline *G4* of §0.1.2, which awards regulation of type access to the designer of the type.

By extensibility Guideline *G1b* of §0.1.2, the programmer should be able to create new types that (1) are encapsulated, (2) do not satisfy any of the built-in type predicates, §3.5,

[9]MIT/GNU Scheme, [MitGnu], provides a procedure `procedure-environment` for determining the static environment of a non-compiled compound procedure.

and (3) are supported by customized behavior of the standard operations on first-class objects — centrally, ***read*** §15.1.7, ***write*** §15.1.8, ***eq?*** §4.2.1, and ***equal?*** §4.3.1.

Facility (1), by itself, can be achieved by various techniques using just the core modules; all such techniques ultimately involve operatives, because operatives are the only compound objects in the core modules that can be freely distributed without granting direct access to their contents. Facilities (1) and (2) together can be achieved using the *Encapsulations* module, §8. (Actually, with a good deal of fancy footwork, the primitives of the *Encapsulations* module could be derived as library features — using exception-handling from the *Continuations* module, §7, but building ultimately on the access restrictions of, again, type *operative*.)

Facility (3) is not directly supported by the report. The report could have been extended to support it in a variety of ways — for example, customizations could be attached either to operations, to types, or to instances of types— but supporting it in any one way would seem to lack generality, while supporting it in several ways would lack simplicity. However, facility (3) is supported indirectly in that, while Kernel does not provide support within the scope of the ground environment, it provides ready means for the programmer to create variant languages that *do* provide direct support, in whatever way the programmer chooses. Creation of such variant languages is quite straightforward in Kernel (and, BTW, incurs little overhead even in a fairly naive implementation), because of Kernel's mixture of first-class environments and first-class operatives. With exclusively first-class operatives, nearly all the language semantics are derived from the ground environment; and with articulate support for first-class environments, the programmer can readily construct alternatives to the ground environment. Consequently, sweeping semantic variations of the language that, in Scheme, would require the programmer to write an explicit meta-circular evaluator, may be straightforwardly achieved in Kernel by applying the built-in evaluator algorithm with a different environment.

3.5 Partitioning of types

In each module, certain of its primitive applicatives are identified as *primitive type predicates*. Each of these predicates takes zero or more arguments, and returns true iff all of its arguments have the type tested by that predicate. (Thus, on zero arguments the predicate returns true.)

The primitive type predicates must *partition* all possible objects; that is, every object must satisfy exactly one primitive type predicate. The primitive type predicates described in this report partition all objects described in, and all objects constructible by means described in, this report. While an extension of Kernel (§3.4) may introduce objects that do not satisfy any primitive type predicate described here, it must introduce sufficient additional primitive type predicates to partition the objects possible under the extension.

Rationale:

Kernel's typing policy is an exercise in balancing the two halves of guideline *G3* of §0.1.2 — that dangerous behaviors should be tolerated but not invited. Tolerance rules out traditional manifest typing, whose basic purpose is to proactively exclude from the language behaviors that *might* later lead to illegal acts. Partitioning latently typed objects by primitive type promotes the 'not invited' half of the equation, by clarifying the role of each object at the level of the base language, so that ambiguities of purpose only occur if they are deliberately introduced by the programmer.

The canonical illustration of the single-role principle is type *boolean*, whose essential purpose in the language is to serve as the result of a conditional test (as for operative ***$if***, §4.5.2). Most Lisps, including *R5RS* Scheme, provide a *boolean* type for this purpose, but actually allow any object whatever to occur as a conditional test result, the evident purpose of this convention being that the programmer will omit the explicit predicate in conditional tests where the value to be predicated is `#f` exactly when the predicate would return false. This deliberate muddling of object roles in the base language leaves it vulnerable to the accidental use of an expression as a conditional test that was *never intended* to fill such a role, and cannot in fact ever produce a false result; in such an accidental use, an illegal act actually *has* been performed, but is not detected. Requiring a boolean result —which Kernel does— would presumably cause the error to be pinpointed the first time the code is run, rather than lying dormant until it must be laboriously hunted down when the code is discovered to misbehave; and (not forgetting the permissive half of the equation), the programmer is neither prevented nor even significantly inconvenienced from any activity since all it takes to guarantee a boolean result is that the programmer explicitly state his intention in the form of a predicate.

In principle, primitive type predicates could be defined to be unary, and then generalized to zero or more arguments as library features. That, however, would significantly increase the number of library features for the sake of a single derivation technique that would simply be repeated on each type predicate. Instead we merely note here that, given a unary predicate ***foo?***, its generalization to zero or more arguments could be defined by the following expression, using the unary predicate, and primitive and library features of the core modules. (Re the behavior on zero arguments, see the rationale for applicative ***and?***, §6.1.2.)

```
($define! foo?
   ($let ((foo?  foo?)) ; save the unary predicate
      ($lambda args
         (apply and? (map foo? args)))))
```

3.6 External representations

An important concept in Kernel (and Lisp generally) is that of the *external representation* of an object as a sequence of characters. For example, an external representation of the integer 28 is the sequence of characters "`28`", and an external representation of a list consisting of the integers 8 and 13 is the sequence of characters "`(8 13)`".

The external representation of an object is not necessarily unique. The integer 28 also has representations "`#e28.000`" and "`#x1c`", and the list in the previous paragraph also has the representations "`( 08 13 )`" and "`(8 . (13 . ()))`".

External representations can be used for input and output. The applicative ***read*** (§15.1.7) parses all external representations, and cannot parse anything that is not an external representation. Program source files are always parsed (e.g., by applicative ***load***, §15.2.2) as if by ***read***, so that each program source file is just a series of external representations. The applicative ***write*** (§15.1.8) generates external representations whenever possible, that is, whenever the object to be written has an external representation.

An external representation might not capture all available information about the object, but all external representations of an object capture the *same* information about the object, and this information is preserved by applicatives ***read*** and ***write***. When different external representations of an object are read, or when the same external representation of an object is read multiple times, all the generated objects will be ***equal?*** (§4.3.1) to each other; and if all these generated objects are then written and the external representations are read in again, the newly generated objects will be ***equal?*** to the previously generated objects.

Many objects have external representations, but some, such as combiners and environments, do not. However, all objects are legal arguments to applicative ***write***; in those cases where the object does not have an external representation, the sequence of characters generated by ***write*** is an *output-only representation.* When ***read*** attempts to parse an output-only representation, an error is signaled; thus, representations by ***write*** may be rejected or consistently parsed by ***read***, but never misparsed.

The representations generated by ***write*** are also subject to a constraint similar to (but, in this instance, stronger than) the rules governing encapsulation from §3.4: the representation of an object o by ***write*** cannot reveal any information about o that couldn't be determined without ***write***.

A clear distinction must be observed between the object represented by an external representation, and the object that results from *evaluating* the represented object. The sequence of characters "`(+ 2 6)`" is *not* an external representation of the integer 8. Rather, it is an external representation of a three-element list, the elements of which are the symbol `+` and the integers 2 and 6. Similarly, the sequence of characters "`($lambda x x)`" is an external representation of a three-element list whose first element is the symbol `$lambda` and whose second and third elements are both the symbol `x`.

The syntax of external representations of various types of objects accompanies the descriptions of the types in the modules that support them, in §§4–15.

3.7 Diagnostic information

Another important concept in Kernel is that of limiting what information is accessible *from within a Kernel program.* Such limits are imposed notably by Kernel's type encapsulation (§3.4), and the limits are in turn used to constrain Kernel's treatment of output representations (§15.1.8) and object equivalence (§§4.2.1, 4.3.1).

Because implementations must only match the *abstract* behavior described in this report, there is nothing to keep them from maintaining additional information internally, as long as it isn't accessible from within a program. Moreover, by the same reasoning, this additional information can be shared with the *user* of the Kernel system, as long as it remains unavailable to the *program.* For example, objects parsed from a source file could be tagged with their source positions, and the tags could be used to generate diagnostic messages that pinpoint the source locations of errors, even though objects with different source-position tags might be ***eq?*** (that is, when observed from within the Kernel system they might be the same object).

Rationale:

This report does not currently include a way for a Kernel program to run its own implementing platform as a subprocess. If it did include such a facility (as some future revision of the report might conceivably wish to do), it could in principle capture diagnostic messages from the subsidiary process and extract encapsulated information from them. Such a practice might be admissible, under the Kernel design guideline that 'dangerous things should be difficult to do *by accident*' (*G3* of §0.1.2), but admitting it without undermining the definition of type encapsulation would be a neat trick. Since no such facility is currently provided, for the moment this subsection merely points out admissible uses of internal information.

3.8 Mutation

In some cases (e.g. in §4.10.3), the description of a Kernel combiner may stipulate that an object constructed by that combiner is *immutable.* Constructed objects are always mutable unless otherwise stipulated. If an attempt is made to mutate an immutable object, an error is signaled.

Because objects can be designated immutable, there is a need to clearly delineate what constitutes mutation. Most mutations consist of setting the referent of a reference after creation of the referring object (§3.1); other actions described in §§4–15 constitute mutations only when explicitly specified.

Note that mutation of the object referred to by a reference does not, in itself, constitute mutation of the referring object (unless they're the same object, of course). For example, suppose ***p*** is a list `(1 2 3)`. Then `(set-car! p 4)` mutates object ***p***. However, while `(set-car! (cdr p) 5)` mutates the object referred to by the cdr of ***p***, it does not mutate ***p*** itself.

Rationale:

Kernel objects can have two different kinds of internal state, with different purposes and different characteristic hazards.

- Some internal state represents potential resultant data; there is no reason to suppose, in general, that such state will ever reach a *final* result. Mutable pairs (§4.7) are the archetype for objects with this kind of state.

- Other internal state represents the administrative status of some activity; often this will include one or more "dead" states, which indicate that the activity cannot proceed. Ports (§15) are the archetype for objects with this kind of state: once a port object is closed, trying to perform i/o on it is an error.

For each of these two kinds of state, there are situations in which further evolutions of the state should not occur. For administrative state, these situations are determined by the state itself (as with a closed port, which cannot be read/written); while for data state, the difference is whether the state is *intended* to be final. Managing dead administrative states is discussed in the rationale at the beginning of §15. Managing final data states is the purpose of the immutability device, which identifies objects whose data state is final so that the Kernel system knows to signal an error when they are mutated. The purpose of designating certain operations as mutations is to identify them as evolutions that immutability should prohibit. Accordingly, operations that evolve only administrative state of an object —such as the ***read*** operation on ports (§15.1.7)— are not considered mutations of that object. If an object were to have state with separable components of both kinds, operations affecting data components of state would be mutation while operations affecting only administrative components of state would not.

3.9 Self-referencing data structures

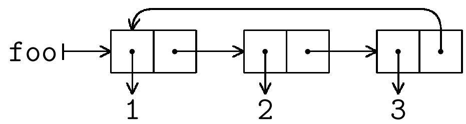
By a *data structure* we will mean, precisely: a collection of objects, with some set of the references they contain (§3.1) designated as *internal* and referring to objects of the structure, and one of the objects of the structure designated as the *starting object*. References contained in the objects but not internal are *external*, and may refer to objects outside the structure.

An example of particular interest is the class of *list* structures (see also §6.3.9). The empty list has just one object, nil, and thus has no internal (nor external) references. A nonempty list starts with a pair. Its objects are the start, all pairs reachable from the start through cdr references, and nil if reachable through cdr references. Its internal references are the cdr references of its pairs. For this to constitute a data structure under the definition, internal references must refer to objects of the structure, so the cdr of each pair of the list must refer either to nil or to a pair of the list.

A data structure is *self-referencing* (or, *cyclic*) if there is some object *o* within the structure from which, by following a nonempty chain of internal references, one can get to *o*. For example, one could create a self-referencing list by evaluating the sequence:

```
($define! foo (list 1 2 3))
(append! foo foo)
```

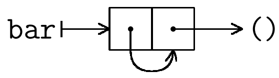
The resulting list data structure would consist of three pairs, with the cdr of the third pair referring back to the first:



Self-referencing structures are sometimes used to represent infinite abstract structures (and the actual structures may then be elliptically called "infinite"). The above list *foo* could be used as a finite representation of the infinite list `(1 2 3 1 2 3 ...)`.

Note that the designation of internal references can determine whether a data structure qualifies as self-referencing. For example, the following is an acyclic, a.k.a. *finite*, list exactly because the car references in a list are external. The same objects with different internal reference designations could constitute a cyclic data structure; but that structure wouldn't be a *list*.

```
($define! bar (list 1))
(set-car! bar bar)
```

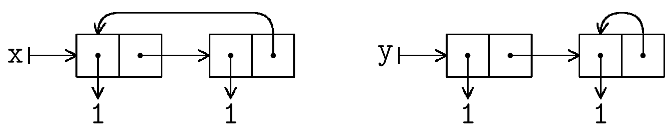


Kernel combiners that involve data structure traversal, such as predicate ***equal?*** (§4.3.1), are required to terminate and give correct results in the presence of cycles unless specifically otherwise stated.

Discussions of behavior on self-referencing structures often use the auxiliary concept of *isomorphically structured* data structures. Data structures S and S' are structurally isomorphic iff there is a one-to-one correspondence between objects of S and objects of S', and between references from corresponding objects, such that (1) the starting object of S corresponds to the starting object of S', (2) corresponding objects have the same type, (3) corresponding internal references refer to corresponding objects, and (4) corresponding external references refer to objects that correspond under some relation that is specified as part of the isomorphism. If no relation is specified between corresponding external referents, they are unconstrained (i.e., all objects are related).

For example, the following two cyclic lists are ***equal?***, and have the same number of pairs, but are not structurally isomorphic.

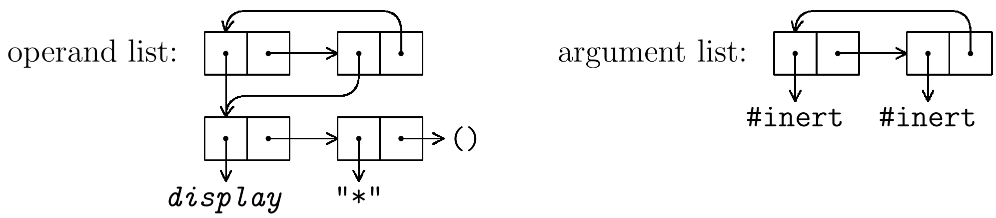
```
($define! x (list 1 1))
(append! x x)
($define! y (list 1 1))
(append! y (cdr y))
```



If the combiner of a combination is applicative, and the cdr of the combination is not a list, an error is signaled. However, in general the operand list is not required to be acyclic. Given a cyclic operand list to an applicative, the evaluator algorithm (§3.3) is required to evaluate the referent of the car of each pair of the list once, (that is, the number of operand evaluations is exactly equal to the number of pairs in the list,) and construct an argument list structurally isomorphic to the operand list. For example:

```
($define! foo (list (list display "*")))
($define! foo (cons (car foo) foo))
(append! foo foo)
(eval (cons list foo) (get-current-environment))
```

In the last expression, the operand list `foo` of applicative `list` is a cyclic list of two pairs, whose two elements are both identically the same object `(display "*")`. That object is evaluated twice, once for each time it occurs in the list. Exactly two asterisks are displayed, and the result returned is a cyclic list of two pairs whose elements are both `#inert` (the value returned by `display`).



Rationale:

The *R5RS* specifically allows Scheme's `equal?` predicate to not terminate when comparing self-referencing structures. This presents a danger when working with self-referencing structures, since forgetfully using `equal?` may cause the program to diverge. Such pitfalls inhibit the programmer's free use of self-referencing structures, degrading the practical status of self-referencing structures as first-class objects (*G1a* of §0.1.2).

Given that robust structure-traversal algorithms are provided in Kernel, there is a temptation to provide naive traversal algorithms as well, because naive traversal is faster in those cases where it works. (The time penalty for robust structure-traversal is actually just a constant factor, i.e., linear in the structure size, provided the objects in the structure can be temporarily marked during the traversal.) Since the motivation for providing naive traversal as well is efficiency, *G5* of §0.1.2 will forbid it *if* it compromises any other design principle.

After studying several possible approaches to simultaneous support for both naive and robust traversal, it was judged that such support would, in fact, compromise two different design principles. One compromise is to *G3* of §0.1.2. Naive traversal algorithms are dangerous; and, inevitably, the more readily available naive traversal algorithms become, the more readily the programmer can use them by accident. So, whatever measures are used to reduce the additional risk, there *is* an additional risk, and the guideline is perturbed.

The second compromise is more elemental. The basic design philosophy (top of §0.1.2) precludes unnecessary features.[10] But the introduction of a naive-traversal feature isn't necessary. In those cases where a tightly coded robust traversal isn't fast enough, and the compiler is unable to prove acyclicity and substitute a naive traversal, the programmer can *hand-code* naive traversals; as explicitly coded algorithms go, naive depth-first traversals are quite simple — in contrast to the robust traversal algorithms which, if omitted from the standard language, would be quite troublesome for the programmer to hand-code.

There are three ways for a list-handling combiner to address a cyclic list.

- The combiner may treat the cyclic list as a type error (which might not be signaled in a non-robust implementation). This should be done iff the action of the combiner is naturally undefined when the list is cyclic (e.g., a cyclic list as a non-final argument to applicative ***append***, §6.3.3).
- The combiner may traverse the cyclic list indefinitely, until and unless its termination criterion, if any, is met. This should be done iff the action of the combiner does not require a list termination point, and entails stepping through the list, in order, with a potential for side-effects from the steps (e.g., the list of operands to operative ***$and?***, §6.1.4).
- The combiner may process the cyclic list in finite time (modulo nontermination of subsidiary computations, over which the combiner has no control). This should be done iff the action of the combiner does not require a list termination point, and does not entail processing the list side-effect-fully in order (e.g., the association-list argument to applicative ***assoc***, §6.3.6).

The governing principle in all three cases is *G3* of §0.1.2, that dangerous activities should be permitted but difficult to undertake by accident. It is presumed that, when cycles occur in lists, they are usually the result of deliberate action rather than accident; therefore, error-signaling is limited to cases where the action of the combiner would be undefined, rather than merely unlikely to be useful (so as not to second-guess a deliberate decision by the programmer). On the other hand, accidents with cyclic lists may still occur if a combiner's behavior contradicts the programmer's expectation; hence, surprises are avoided by ensuring that all combiners in the report adhere strictly to the simple policy presented above.

Under this policy, the assertion that a combiner will process some list in no particular order is a substantial statement. That is, it makes a positive contribution to the design, by determining the behavior of the combiner when the list is cyclic. This means that, to preserve the integrity of the language design, implementations of Kernel must refrain from endorsing any particular order of processing for the combiner, regardless of whether they *use* any particular order. (This is in contrast to Scheme, where particular implementations routinely "extend" the *RxRS* by specifying order of processing in cases where the *RxRS* doesn't.)

---

[10]In effect, the design philosophy is itself a refinement of the *principle of necessity*, the medieval scholastic principle that "Entities should not be multiplied unnecessarily." Toward the end of the scholastic period, William of Occam used the principle of necessity so effectively in cutting his opponents' arguments to ribbons, that today the principle is commonly called Occam's Razor.

The behavior of the evaluator algorithm on a cyclic operand list to an applicative is largely determined by the general policy: arguments don't have to be evaluated in any particular order (§3.3), so a cyclic operand list must be handled in finite time. (So too, in accordance with the above reasoning, implementations should refrain from endorsing any particular order of argument evaluation.) Details of the semantics are chosen to be the same as if argument evaluation were done using *map* (§5.9.1). For example, the following two expressions are equivalent.

```
(apply (wrap list) operands (get-current-environment))
(map (wrap ($vau (x) e (eval x e))) operands)
```

3.10 Proper tail recursion

Implementations of Kernel, like implementations of Scheme, are required to be *properly tail recursive*.

A combiner call is *active* if the called combiner may still return. This includes calls that may be returned from either by the current continuation or by continuations captured earlier (such as by *call/cc*, §7.2.2) that are later invoked. In the absence of captured continuations, calls could return at most once and the active calls would be those that had not yet returned.

Within a combiner call, an expression may be evaluated as a *tail context*. The evaluator algorithm specifies one tail context (§3.3), and the descriptions of Kernel combiners in §§4–15 specify some additional tail contexts. Implementations may recognize other tail contexts in addition to those specified; but no expression can be evaluated as a tail context unless its result will be trivially returned as the result of the containing combiner call (i.e., returning the result to the containing call has no consequences except to return it *from* the containing call).

A *tail call* is a combiner call that is evaluated as a tail context. A Kernel implementation is properly tail-recursive iff it supports an unbounded number of active tail calls.

A formal definition of proper tail recursion in Scheme can be found in [Cl98]. Proper tail recursion is more straightforward in Kernel because of the absence of special forms.

Rationale:

Intuitively, no space is needed for an active tail call because the continuation that is used in the tail call has the same semantics as the continuation passed to the combiner containing the call. Although an improper implementation might use a new continuation in the tail call, a return to this new continuation would be followed immediately by a return to the continuation of the original combiner call. A properly tail-recursive implementation returns to that continuation directly.

Proper tail recursion was one of the central ideas in Steele and Sussman's original version of Scheme. Their first Scheme interpreter implemented both functions and actors. Control flow was expressed using actors, which differed from functions in that they passed their results on to another actor instead of returning to a caller. In the terminology of this section, each actor finished with a tail call to another actor.

Steele and Sussman later observed that in their interpreter the code for dealing with actors was identical to that for functions and thus there was no need to include both in the language. [SusSte75, p. 40]

4 Core types and primitive features

The most fundamental features of Kernel are grouped into ten core modules, supporting nine primitive types (unevenly distributed amongst the modules). Three of the core modules are optional (*Equivalence under mutation*, *Pair mutation*, and *Environment mutation*, §§4.2, 4.7, 4.9). All modules in the report assume all the required core modules; only optional and non-core assumptions will be specified under particular modules.

This section describes the primitive types and primitive features of all the core modules. §§5–6 describe the associated library features.

Rationale:

The rough criteria for a module to be considered core are that (1) nontrivial Kernel programming can't be done without it, or (2) the Kernel evaluator algorithm can't be understood without it. Optional modules *Pair mutation* and *Environment mutation* were judged to fall within the close penumbrae of their respective core types. The two most notable omissions from the core are modules *Continuations* and *Numbers* (§§7 and 12); they were judged not to meet either criterion for core modules, although module *Continuations* is used in the library derivation of core operative `$binds?` for error-handling (§6.7.1), and module *Numbers* is formally assumed by core module *Pairs and lists* for list measuring and indexing.

The division of the required language core into modules is somewhat arbitrary, and is chosen for clarity of presentation. We also wish to present the core library features in such an order that none of their derivations depends on another that occurs later in the report; and this causes some difficulty for both the modular division and the presentation, because it leads us to present several of the most basic core library features in a rather scrambled order, with little respect for the data types involved. In order to minimize —and clarify the character of— this perturbation, the language core is cross-cut into three sections of the report: the type descriptions and primitive features are in §4 (here); the perturbed basic library features are in §5; and the remaining majority of library features are in §6. This arrangement allows §§4 and 6 to be ordered for presentation, while showcasing the minimality of the core primitives, and the ubiquity of certain core library features (in §5).

4.1 Booleans

The *boolean* data type consists of two values, which are called *true* and *false*, and have respectively external representations `#t` and `#f`. There are no possible mutations of either of these two values, and the *boolean* type is encapsulated.

A combiner that always returns a boolean value is called a *predicate*. Conversely, specifying that a combiner is a predicate means that it always returns a boolean value. By convention, the names of predicates always end in "?".

Library features associated with type *boolean* will be described in §6.1.

Rationale:

The essential purpose of boolean values in the language is to serve as the results of conditional tests, for which purpose *only* boolean values are permitted by Kernel. (See rationale under *Partitioning of types*, §3.5.)

4.1.1 `boolean?`

`(boolean? .` *objects*`)`

The primitive type predicate for type *boolean*.

Because there are only two values in the type, predicate ***`boolean?`*** could be constructed as a library combiner. However, designating it a primitive type predicate satisfies the requirements for partitioning of types from §3.5.

4.2 Equivalence under mutation (optional)

Rationale:

Kernel has two general-purpose equivalence predicates, whereas *R5RS* Scheme has three. The two Kernel predicates correspond to the abstract notions of equivalence *up to mutation* (*`equal?`*, §4.3), and *in the presence of mutation* (*`eq?`*, in this module). Scheme assigns the abstract notion of equivalence in the presence of mutation to an intermediate predicate *`eqv?`*, and uses predicate *`eq?`* for the technically stronger equivalence between objects whose *`eqv?`*-ness can be verified especially quickly in any particular implementation. In language design terms, Scheme introduces a third equivalence predicate for the express purpose of promoting implementation-dependent intrusion of concrete performance issues on the abstract semantics of the language — which directly violates Kernel's principles on simplicity and generality as well as its guideline on efficiency (*G5* of §0.1.2).

The criterion for another module to assume this one is that the assuming module supports mutation that could cause objects to be *`equal?`* but not *`eq?`*. For example, *Pair mutation* assumes this module, but *Environment mutation* does not.

For cross-implementation compatibility, the behavior of *`eq?`* is defined in terms of a comprehensive implementation of Kernel. For example, two pairs returned by different calls to *`cons`* are not *`eq?`*, even if they have the same car and cdr and the implementation doesn't support pair mutation; and two empty environments returned by different calls to *`make-environment`* are not *`eq?`*, even if the implementation doesn't support environment mutation. The latter case shows how the implementation-independence can impact implementations even if they don't support *`eq?`*, since the behavior of required predicate *`equal?`* on environments is tied to that of *`eq?`* (which is, in turn, why module *Environment mutation* does not require this module).

4.2.1 `eq?`

`(eq?` *object1 object2*`)`

Predicate that returns true iff *object1* and *object2* are effectively (§3.7) the same object, even in the presence of mutation. The following universal rules constrain, but do not uniquely determine, its behavior. Its behavior for objects of a particular type described in this report may be further constrained in the description of the type.

1 The ***eq?*** predicate must be reflexive, symmetric, and transitive.

2 If the two objects (*object1* and *object2*) are non-interchangeable in any way that could affect the behavior of a Kernel program, using any features in the implementation or the report, but without first postulating that ***eq?*** distinguishes them, ***eq?*** must return false. For example:

- If the two objects are observably not of the same type, ***eq?*** must return false.
- If one of the objects is mutable and the other is immutable, ***eq?*** must return false.
- If both objects are mutable, and mutating one will not necessarily cause the same mutation to the other, ***eq?*** must return false.
- If the objects have different external representations (as do, for example, the two boolean values), ***eq?*** must return false.

3 For any particular two objects, the result returned by ***eq?*** is always the same.

This applicative will be generalized to handle zero or more arguments in §6.5.1.

4.3 Equivalence up to mutation

4.3.1 `equal?`

`(equal?` *object1 object2*`)`

Predicate that returns true iff *object1* and *object2* "look" the same as long as nothing is mutated. This is a weaker predicate than ***eq?***; that is, ***equal?*** must return true whenever ***eq?*** would return true. The following universal rules constrain, but do not uniquely determine, the behavior of ***equal?***. Its behavior for objects of each type described in this report will be uniquely determined, modulo the behavior of ***eq?***, by further constraints as necessary in the description of the type.

1 The ***equal?*** predicate must be reflexive, symmetric, and transitive.

2 If ***eq?*** would return true, ***equal?*** must return true.

3 If the two objects (*object1* and *object2*) are non-interchangeable in any way that could affect the behavior of a Kernel program that (a) performs no mutation and (b) doesn't use ***eq?*** (neither directly nor indirectly), then ***equal?*** must return false. For example:

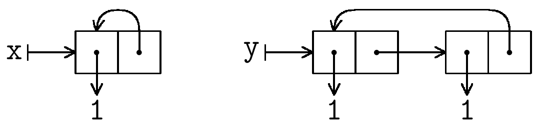
- If the two objects are observably not of the same type, ***equal?*** must return false.
- If the objects have different external representations, ***equal?*** must return false.
- If the objects are both numbers, and numerically equal, but have different inexactness bounds (e.g., one is exact and the other isn't; §12.2), ***equal?*** must return false.

4 If ***equal?*** is not required to return false by the preceding rule, and this fact can be determined with certainty by a (correct) Kernel program that (a) is independent of the objects (they don't refer to it and it only refers to them as parameters), (b) examines the objects only passively (doesn't use them or parts of them as combiners in evaluation), (c) performs no mutation, and (d) always terminates (provided the quantity of actual data within the runtime system is finite), then ***equal?*** must return true. For example,

- Suppose variables `x` and `y` are set up by evaluating the following sequence of expressions.

  ```
  ($define! x (list 1))
  ($define! y (list 1 1))
  (append! x x)
  (append! y y)
  ```

  Then `(equal? x y)` would evaluate to `#t`.



5 For any particular two objects, the result returned by ***equal?*** is always the same during a period of time over which no mutation occurs.

It is generally recommended that ***equal?*** return false in all cases where these rules do not require it to return true.

This applicative will be generalized to handle zero or more arguments in §6.6.1.

Rationale:

The Kernel predicate ***equal?***, unlike its Scheme counterpart, has to terminate for all possible arguments (since it isn't given dispensation to do otherwise). The set of cases in which ***equal?*** is required, by Rule *3* above, to return false is formally undecidable; that doesn't interfere with termination of ***equal?***, but does guarantee that the terminating predicate returns false in some cases where it isn't required to. (Terminating predicate ***eq?*** must similarly return false in some unrequired cases; see §4.10.) The set of cases in which ***equal?*** is required to return true, by Rule *4* above, might appear at first glance to be undecidable but, in practice, it is unproblematically decidable. Because Rule *4* stipulates that the determining program can only examine the objects *passively*, the determining program cannot get bogged down in comparing the formally undecidable active behavior of algorithms; degree of encapsulation doesn't actually matter to this point, as, for example, if the body of a compound combiner were made publicly visible so that it could be used in the determination, different algorithms that do the same thing would be un-***equal?*** (hence un-***eq?***) exactly because the combiners could by supposition be distinguished via their syntactically distinct bodies.

4.4 Symbols

Two symbols are ***eq?*** (§4.2.1) iff they have the same external representation. Symbols are immutable, and the *symbol* type is encapsulated.

The external representations of symbols are usually identifiers (§2.1). However, symbols with other external representations may be created; see §13.1.1.

Rationale:

Symbols are useful as lookup keys for environments (§§3.2, 4.8), thus providing the base case for nontrivial evaluation (§3.3), exactly because they are isomorphic to their external representations.

4.4.1 `symbol?`

`(symbol? .` *objects*`)`

The primitive type predicate for type *symbol*.

4.5 Control

The *inert* data type is provided for use with control combiners. It consists of a single immutable value, having external representation `#inert`. The *inert* type is encapsulated.

Library features of the core *Control* module will be described in §§5.1, 5.6, and 6.9.

Rationale:

Some combiners are called for their side effects, not their results. In the C family of languages, functions called for effect have return type ***void***. The later Scheme reports describe the results of for-effect procedures as 'unspecified', which is a politically necessary hedge because different Scheme implementations already in place follow a variety of conventions concerning the return values of such procedures. Unfortunately, some Scheme implementations allow for-effect procedures to return useful information, which creates a temptation for programmers to write anti-portable code by using the result. Kernel avoids this regrettable turn of events by explicitly requiring the result of each for-effect combiner to be inert. Since the *inert* type is encapsulated, its one instance doesn't carry any usable information beyond its identity and type, which are isomorphic. (But see §3.7.)

Merely replacing "unspecified" with "inert" in the descriptions of standard combiners would violate the spirit of *G1b* of §0.1.2, which calls for duplicability of built-in facilities by the programmer. Hence the inert value must also have a full external representation (as opposed to an output-only representation, §3.6), to facilitate explicit programmer declaration of for-effect combiners.

4.5.1 `inert?`

`(inert? .` *objects*`)`

The primitive type predicate for type *inert*.

4.5.2 `$if`

`($if` ⟨test⟩ ⟨consequent⟩ ⟨alternative⟩`)`

The ***$if*** operative first evaluates ⟨test⟩ in the dynamic environment (that is, the environment in which the `($if ...)` combination is evaluated). If the result is not of type boolean, an error is signaled. If the result is true, ⟨consequent⟩ is then evaluated in the dynamic environment as a tail context (§3.10). Otherwise, ⟨alternative⟩ is evaluated in the dynamic environment as a tail context.

Rationale:

On the exclusion of non-*boolean* results from conditional tests, see the rationale under *Partitioning of types*, §3.5.

In *R5RS* Scheme, the ⟨alternative⟩ operand to ***if*** is optional; and if it is omitted, and ⟨test⟩ evaluates to false, the result is 'unspecified' — which would mean, in Kernel, that the result would be inert. For consistency with the design purpose of `#inert` — which is to convey no information— two-operand ***$if*** ought to return `#inert` regardless of whether ⟨consequent⟩ is evaluated; but at that point, it becomes evident that the two- and three-operand operations are really separate, and by rights ought not to be lumped into a single operative (which lumping doesn't square well with the uniformity guideline, *G1* of §0.1.2, anyway); instead, if both operations are supported they should be given different names. The two-operand form, though, is just a specialized shorthand; so both clarity (thus accident-avoidance, *G3* of §0.1.2) and simplicity are against its inclusion in the language. (Similar issues arise for ***$cond***, §5.6.1.)

4.6 Pairs and lists

A *pair* is an object that refers to two other objects, called its car and cdr. The Kernel data type *pair* is encapsulated.

The *null* data type consists of a single immutable value, called *nil* or *the empty list* and having external representation `()`, with or without whitespace between the parentheses. It is immutable, and the *null* type is encapsulated.

If a and d are external representations of respectively the car and cdr of a pair p, then `(`a `.` d`)` is an external representation of p. If the cdr of p is nil, then `(`a`)` is also an external representation of p. If the cdr of p is a pair p_2, and `(`r`)` is an external representation of p_2, then `(`a r`)` is an external representation of p.

When a pair is output (as by ***write***, §15.1.8), an external representation with the fewest parentheses is used; in the case of a finite list, only one set of parentheses is required beyond those used in representing the elements of the list. For example, an object with external representation `(1 . (2 . (3 . ())))` would be output using, modulo whitespace, external representation `(1 2 3)`.

Combiners for mutating pairs will be presented in a separate, optional *Pair mutation* module (§4.7). Otherwise, library features associated with these types will be described in §§5.2, 5.4, 5.7, and 6.3.

This module assumes the *Numbers* module (§12). (See §§5.7, 6.3.)

Rationale:

The *Numbers* module is used to measure and index lists.

4.6.1 `pair?`

`(pair? .` *objects*`)`

The primitive type predicate for type *pair*.

4.6.2 `null?`

`(null? .` *objects*`)`

The primitive type predicate for type *null*.

4.6.3 `cons`

`(cons` *object1 object2*`)`

A new pair object is constructed and returned, whose car and cdr referents are respectively *object1* and *object2*.

Note that the general laws governing mutation (§3.8: constructed objects are mutable unless otherwise stated) and the general laws governing `eq?` (§4.2.1: independently mutable objects aren't `eq?`) conspire to guarantee that the objects returned by two different calls to `cons` are not `eq?`.

4.7 Pair mutation (optional)

This module consists of those standard combiners that either mutate pairs, or are only needful when pair mutation is possible. It assumes the *Equivalence under mutation* module (§4.2), and the *Numbers* module (§12). Library features of the module will be described in §§5.8 and 6.4.

> *Rationale:*
> The *Equivalence under mutation* module is assumed because pairs may be `equal?` without being `eq?`. The *Numbers* module is used to index lists.

4.7.1 `set-car!`, `set-cdr!`

(`set-car!` *pair object*)
(`set-cdr!` *pair object*)

These applicatives set the referent of, respectively, the car reference or the cdr reference of *pair* to *object*. The result of the expression is inert.

Recall, from §3.1, that the act of setting the referent of a reference after object creation constitutes a mutation of the object containing the reference; and from §3.8, that when an attempt is made to mutate an immutable object, the error must be signaled.

4.7.2 `copy-es-immutable`

(`copy-es-immutable` *object*)

The short description of this applicative is that it returns an object `equal?` to *object* with an immutable evaluation structure. The "`-es-`" in the name is short for "evaluation structure".

The evaluation structure of an object o is defined to be the set of all pairs that can be reached by following chains of references from o without ever passing through a non-pair object. The evaluation structure of a non-pair object is empty.

If *object* is not a pair, the applicative returns *object*. Otherwise (if *object* is a pair), the applicative returns an immutable pair whose car and cdr would be suitable results for (`copy-es-immutable` (`car` *object*)) and (`copy-es-immutable` (`cdr` *object*)), respectively. Further, the evaluation structure of the returned value is isomorphic (§3.9) to that of *object* at the time of copying, with corresponding non-pair referents being `eq?`.

These constraints imply that the result returned by `copy-es-immutable` must be initially `equal?` to *object*. They also imply that if *object* is a mutable pair, then the result is not `eq?` to *object*. However, if *object* is an *im*mutable pair, then (given that there is no way in Kernel for a mutable pair to be the car or cdr of an immutable pair) the result may or may not be `eq?` to *object* at the discretion of the implementation. (This is in contrast to library applicative `copy-es`, §6.4.2, which always returns a non-`eq?` pair when given a pair as argument.)

Rationale:

Whenever unexpected operand capturing occurs (i.e., an unevaluated operand is acquired by a combiner that the caller thought was applicative), there is a risk of unexpected operand *mutation*. Most algorithms, however, are intended by the programmer to be immutable, and therefore, when an object is primarily meant to represent an algorithm, mutating it is a dangerous activity that ought to be difficult to do by accident (*G3* of §0.1.2). The notion of the 'evaluation structure' of an object is meant to correspond to the algorithm that the object represents. Combiners that are used particularly to construct representations of algorithms acquire immutable copies of the given evaluation structures: `$vau` (§4.10.3) and `load` (§15.2.2) do this; but `eval` (§4.8.3) does not, since it should be able to induce arbitrary evaluations, and arbitrary evaluations must admit mutable structures (else one could never pass an argument for mutation, as to `set-car!` and `set-cdr!`, §4.7.1). Applicative `copy-es-immutable` empowers the programmer to duplicate these built-in Kernel facilities (*G1b* of §0.1.2).

Symbols aren't copied by the applicative because, although they clearly play a direct role in specifying algorithms, they are immutable so there is never any need to make immutable copies. Alternatively, one could claim that they *are* copied, but because they are immutable, the copies are `eq?` to the originals.

The possibility was considered of providing a primitive applicative `cons-immutable`, which would return an immutable pair with given car and cdr. However, `cons-immutable` would support an altogether different capability from `copy-es-immutable`. On one hand, `cons-immutable` would not support library derivation of `copy-es-immutable`, which can handle self-referencing structures that cannot be copied one pair at a time without mutating some pair after construction to create a cycle. On the other hand, `cons-immutable` would empower the programmer to create an immutable pair whose car or cdr referent is a mutable pair, which cannot be done with `copy-es-immutable`. The selective power of `cons-immutable` would thus allow much more intricate patterns of structure immutability, and ought not to be introduced into a simplicity-oriented language without a compelling reason; the Kernel design calls for generality in the service of simplicity, not generality for its own sake.

Strictly speaking, `copy-es-immutable` could have been listed as a library feature. It can be derived using `$vau`, by exploiting the fact that `$vau` immutably copies the evaluation structures of the bodies of compound operatives. We prefer to list `copy-es-immutable` as a primitive prior to `$vau`, so that we can explain and discuss immutable copies of evaluation structure separately from the more central aspects of *$vau*. For perspective, though, here is a derivation of *copy-es-immutable*.

```
($define! copy-es-immutable
   ($lambda (object)
      (((wrap $vau) ()
                    #ignore
                    (cons (unwrap list) object)))))
```

4.8 Environments

An environment consists of a set of bindings, and a list of zero or more references to other environments called its parents; detailed terminology, the symbol lookup algorithm, and the ground environment were explained in §3.2.

Changing the set of bindings of an environment, or setting the referent of the reference in a binding, is a mutation of the environment. (Changing the parent list, or a referent in the list, would be a mutation of the environment too, but there is no facility provided to do it.)

The Kernel data type *environment* is encapsulated. Among other things, there is no facility provided for enumerating all the variables exhibited by an environment (which is not required, after all, to be a finite set), and no facility for identifying the parents of an environment.

Two environments are ***equal?*** iff they are ***eq?*** (§§4.2.1, 4.3.1).

Combiners for mutating environments will be presented in a separate, optional *Environment mutation* module (§4.9). Otherwise, library features associated with type *environment* will be described in §§5.10 and 6.7.

An auxiliary data type used by combiners that perform binding is *ignore*. The *ignore* type consists of a single immutable value, having external representation `#ignore`. The *ignore* type is encapsulated.

Rationale:

First-class environments offer a tremendous amount of control over what can be accessed from where — but only if there are limitations carefully placed on what can be done without explicit permission. In particular, whenever a combiner is called, it has the opportunity, in principle, to mutate the dynamic environment in which it was called. This power is balanced by omitting any general facility for determining the parents of a given environment, and also omitting any other general facility for mutating the parents, or other ancestors, of a given environment. (For an example of articulate environment-access control, see ***$provide!***, §6.8.2.)

The behavior of ***equal?*** is tied to that of ***eq?*** to forestall the possibility of an implementation compromising the encapsulation of the type by allowing a program to determine, in finite time, that *all* bindings for one environment are the same as those for another. (Cf. the rationale discussion for the derivation of library predicate ***$binds?***, §6.7.1.)

Type *ignore* is provided specifically for use in parameter matching; see §4.9.1, below. (Contrast type *inert*, §4.5, also an encapsulated type with a single value, but provided specifically for *non*-use.)

4.8.1 `environment?`

`(environment? .` *objects*`)`

The primitive type predicate for type *environment*.

4.8.2 `ignore?`

`(ignore? .` *objects*`)`

The primitive type predicate for type *ignore*.

4.8.3 `eval`

`(eval` *expression environment*`)`

The ***`eval`*** applicative evaluates *expression* as a tail context (§3.10) in *environment*, and returns the resulting value.

Rationale:
The ***`eval`*** applicative provides two notable facilities to the language: the useful facility of evaluating an expression twice, i.e., evaluating the result of evaluating the operand, and the *essential* facility of explicitly specifying what environment will be used for (the second) evaluation. Evaluating twice could be be achieved without ***`eval`***, by ***`wrap`***ping an applicative (§4.10.4); but ***`eval`*** is the *only* primitive combiner in the core modules that supports explicit specification of the evaluation environment. Without that facility, user-defined operatives would be unable to effectively regulate evaluation of their operands.

4.8.4 `make-environment`

`(make-environment .` *environments*`)`

The applicative constructs and returns a new environment, with initially no local bindings, and parent environments the environments listed in *environments*. The constructed environment internally stores its list of parents independent of the first-class list *environments*, so that subsequent mutation of *environments* will not change the parentage of the constructed environment. If the provided list *environments* is cyclic, the constructed environment will still check each of its parents at most once, and signal an error if no binding is found locally or in any of the parents.

As with the *cons* applicative (§4.6.3), the general laws governing mutation (§3.8) and *eq?* (§4.2.1) conspire to guarantee that the objects returned by two different calls to *make-environment* are not *eq?*.

Rationale:

Symbol lookup in an environment is by depth-first search of the environment's improper ancestors (§3.2). If there is no local binding for the symbol, the parents are searched; and if at least one of the parents exhibits a binding for the symbol, the binding is used whose exhibiting parent occurs first on the list of parents. Because searching a parent has no side-effects (i.e., nothing is mutated by the search), the ordered search of the parents must terminate in finite time even if the list of parents is cyclic (per the rationale discussion in §3.9); cf. *assoc*, §6.3.6.

4.9 Environment mutation (optional)

This module consists of just those standard combiners that mutate environments. Library features of the module will be described in §6.8.

Rationale:

There is no need for this module to assume the *Equivalence under mutation* module, because environments are *eq?* iff they are *equal?*.

It isn't clear to what extent one can do serious Kernel programming without mutating environments; but separating the mutators into an optional module allows language implementors to explore this question within the bounds of Kernel specified by this report. In the absence of environment mutators as such, the programmer would presumably fall back on Kernel's rich vocabulary of environment constructors (notably the *$let* family, as §5.10.1), which the report does not class as mutators, although environment initialization is routinely *described* in terms of adding bindings. (See especially the definition of *$letrec*, in §6.7.5.)

Per §3.4, language extensions are judged against features described in the report *rather than* against features actually supported by the extending implementation; so, failure to support features in the *Environment mutation* module as a whole does not prevent a non-comprehensive implementation from providing alternative means for doing some of the same things.

For an example of the subtle interplay between environment mutation, recursion, and sequencing, see the derivation of *$sequence*, in §5.1.1.

4.9.1 `$define!`

`($define!` ⟨definiend⟩ ⟨expression⟩`)`

⟨definiend⟩ should be a formal parameter tree, as described below; otherwise, an error is signaled.

The *$define!* operative evaluates ⟨expression⟩ in the dynamic environment (that is, the environment in which the `(`*$define!* `...)` combination is evaluated), and matches ⟨definiend⟩ to the result in the dynamic environment, binding each symbol in ⟨definiend⟩ in the dynamic environment to the corresponding part of the result; the matching process will be further described below. The ancestors of the dynamic environment, if any, are unaffected by the matching process, as are all bindings, local to the dynamic environment, of symbols not in ⟨definiend⟩. The result returned by ***$define!*** is inert.

A formal parameter tree has the following context-free structure.

⟨ptree⟩ → ⟨symbol⟩ | `#ignore` | `()` | `(`⟨ptree⟩ `.` ⟨ptree⟩`)`

That is, a formal parameter tree is either a symbol, or `#ignore`, or nil, or a pair whose car and cdr referents are formal parameter trees.

A formal parameter tree must also be acyclic, and no one symbol can occur more than once in it. It is not an error for a pair in the tree to be reachable from the root by more than one path, as long as there is no cycle; but if any particular *symbol* were reachable from the root by more than one path, that would count as occurring more than once. Thus, if a pair is reachable by more than one path, there must be no symbols reachable from it.

Matching of a formal parameter tree t to an object o in an environment e proceeds recursively as follows. If the matching process fails, an error is signaled.

- If t is a symbol, then t is bound to o in e.
- If t is `#ignore`, no action is taken.
- If t is nil, then o must be nil (else matching fails).
- If t is a pair, then o must be a pair (else matching fails). The car of t is matched to the car of o in e, and the cdr of t is matched to the cdr of o in e.

Rationale:

In *R5RS* Scheme, ***define*** can only be used in certain contexts. No attempt was made in Kernel to imitate this context-sensitivity, as it was considered philosophically incompatible with making the ***$define!*** combiner first-class.

Formal parameter trees were first developed for Kernel's ***$vau*** operative (§4.10.3), as a generalization of the formal parameter lists of Scheme's ***lambda*** operative. Formal parameter trees are permitted uniformly in every situation (*G1* of §0.1.2) where a definiend is given, i.e., where binding is specified.[11] By empowering versatile interaction between the separately versatile devices of pair-based data structures and first-class environments,

[11]For the natural-linguistically curious: The suffix *-end* in English mathematical terms such as *addend*, *dividend*, etc., is a simple shortening of the Latin gerundive suffix *-endum*. From *addere* to add, *addendum* thing to be added; from *subtrahere* to take away, *subtrahendum* thing to be taken away; *dividere* to divide, *dividendum* thing to be divided. The preceding are all *-ere* Latin verbs, though. For *-ire* verbs the gerundive suffix is *-iendum*; hence, from *definire* to define, *definiendum* thing to be defined.

the uniform generalization of definiends expands the practical versatility of both. A case in point —as well as a notable effect in its own right— is the convergence of consequences by which uniform generalized definiends in Kernel eliminate the motivation for one of the more semantically baroque features of many Lisps: so-called "multiple-value returns".

The idea of multiple-value returns is to view the basic functional model of computation, which says that each function call returns a value, as a *special case* of the ostensibly more general notion that a function call may return a sequence of values. The Kernel design is inconsistent with the illusion that multiple-value returns are more general than single-value returns. The inconsistency engages, in itself, several of the most primal semantic differences between Kernel and Scheme. In Scheme, the syntax for passing value x to captured continuation `c` is "`(c` x`)`", in which the argument tree of applicative `c` is actually `(`x`)` — a list of length one whose sole element is x. In the spirit of removing arbitrary restrictions, one naturally wonders what would happen if `c` were given an argument list of some other length, and the answer is, essentially, "multiple-value return". However, this Scheme scenario has another arbitrary restriction built into it, because Scheme requires the argument tree of `c` to be a list. Since Kernel eliminates this restriction (rationale in §4.10.5), one must then ask what would happen if `c` were given an argument tree that isn't a list at all — and in *that* case, one is back to passing just one value, an argument tree. Since one is passing a single value to `c` either way, the question of whether to allow multiple-value returns is effectively reduced to whether the superficial syntax for single-value return should be "`(c` x`)`", which make single-value return a special case of multiple-value return; or "`(`***apply*** `c` x`)`", which makes multiple-value return a special case of single-value return. (See also the first rationale discussion in §7.2.5.)

Multiple-value return has long been part of Common Lisp ([Ste90, §7.10]), and was more recently added to Scheme ([KeClRe98, Notes]). The usual practical argument against multiple-value return is that there is no reason to complicate the functional model with it, because the model already allows one to simply return, as a single value, a data structure containing however many values are to be returned; but this argument has often failed to convince because, in most languages, it is syntactically clumsy to return a data structure and then decompose it to get at the multiple values within. By the use of Kernel's generalized formal parameter trees, especially in conjunction with operatives of the ***$let*** family (§5.10.1), one can bind variables directly to the parts of a structure as it is returned, rather than returning it first and then decomposing it in a separate operation. For example, one might call ***get-list-metrics*** (§5.7.1), which returns a list of four values, by:

```
($let (((p n a c)  (get-list-metrics ⟨exp⟩)))
   ⟨body⟩)
```

The ***$define!*** operative is most often used with a single symbol as definiend, but the ability to use more general definiends at will sometimes considerably simplifies algorithmic expressions, and is occasionally quite powerful; see for example the derivation of ***$letrec***, §6.7.5.

The convenience of generalized formal parameter trees, while it allows ***car***, ***cdr***, ***cadr***, etc. to be conveniently derived as library features (§§5.4.1, 5.4.2), also greatly reduces the demand for them; see for example the derivations of operative ***$cond***, §5.6.1.

The use of `#ignore` in formal parameter trees is partly a matter of uniformity (*G1* of §0.1.2) with the ⟨eformal⟩ operand of ***$vau*** (first rationale discussion in §4.10.3); but it also serves to retard accidents (*G3* of §0.1.2), in two ways: it improves clarity of code by explicitly acknowledging that certain information will not be used (see §§5.4.1, 5.4.2); and, it eliminates an opportunity for careless errors by not providing a name with which to access data that is not intended to be accessed.

The possibility of allowing duplicate formal parameter names was considered. The matching algorithm would require that all instances of a given name match the same object, thus guaranteeing that there is just one unambiguous choice of value for the binding of the duplicated name. However, it was deemed more likely that duplicate names would occur by accident. (Duplicates could be particularly badly behaved in the case of ***$vau***, since they could not be treated as errors when the compound operative is constructed, and so would instead manifest later as peculiar dynamic errors when the compound operative is called.) Moreover, even if the duplicates did occur deliberately, they might easily be overlooked when studying the code. It would therefore be both clearer and less error-prone to require distinct symbol names and, when ***eq?***-ness is needed, explicitly test for it.

Cyclic formal parameter trees were also considered. Unlike duplicate names, cyclic trees are unlikely to be accidental; the programmer would likely have to go to some trouble to arrange such a thing, and the arrangements would likely be obvious to anyone reading the program. However, while the *fact* of a cyclic formal parameter tree may be evident, its *meaning* is not (as the author discovered in successive attempts to develop a credible matching algorithm for cyclic formal parameter structures); and, barring some kind of additional features introduced for specifying elaborate structural patterns, none of the semantics considered for cyclic formal parameter structures were observed to provide capabilities of much use to a programmer. So it was decided that clarity would be better served by forbidding cycles and letting the programmer explicitly check for any elaborate patterns.

Scheme provides syntactic sugar for ***define***ing procedures, in which a compound definiend is a template for combinations, with its operator being the simple definiend ⟨symbol⟩, and its operands the formal parameters of an implicit ***lambda*** expression. Thus,

```
   (define (square x) (* x x))
⇒  (define square (lambda (x) (* x x)))
```

This syntactic sugar is, evidently, omitted from Kernel since it is incompatible with uniform support of formal parameter trees.

4.10 Combiners

There are two types of combiners in Kernel, *operative* and *applicative*. Both types are encapsulated. Their roles in evaluation are described in §3.3.

All combiners are immutable. Two applicatives are ***eq?*** iff their underlying combiners are ***eq?***. However, ***eq?***-ness of operatives is only constrained by the general rules for ***eq?***, §4.2.1, which leave considerable leeway for variation between implementations. The only relevant constraints are, in fact, that ***eq?*** must be reflexive symmetric and transitive, and that operatives cannot be ***eq?*** if they can ever exhibit different behavior. The following expressions, for example, may evaluate to true or false depending on the implementation.

```
(eq? ($vau (x) #ignore x) ($vau (x) #ignore x))
(eq? ($vau (x) #ignore x) ($vau (y) #ignore y))
```

Two combiners are ***equal?*** iff they are ***eq?*** (§§4.2.1, 4.3.1).

Library features associated with these types will be described in §§5.3, 5.5, 5.9, and 6.2.

Rationale:

The stipulation that combiners are ***equal?*** iff ***eq?*** avoids a loophole in the general rules for predicate ***equal?***. The general rules do not require the predicate to distinguish objects of the same type unless they can be otherwise observably distinguished by a program that doesn't perform any mutation (Rule *3* of §4.3.1); but the only way to distinguish between two operatives is to *call* them, so under the general rules, if each of several operatives would immediately cause mutation when called, predicate ***equal?*** would be permitted to equate all of them.

4.10.1 `operative?`

`(operative? .` *objects*`)`

The primitive type predicate for type *operative*.

4.10.2 `applicative?`

`(applicative? .` *objects*`)`

The primitive type predicate for type *applicative*.

4.10.3 `$vau`

`($vau` ⟨formals⟩ ⟨eformal⟩ ⟨expr⟩`)`

⟨formals⟩ should be a formal parameter tree, as described for the ***$define!*** operative, §4.9.1. ⟨eformal⟩ should be either a symbol or `#ignore`.

If ⟨formals⟩ does not have the correct form for a formal parameter tree, or if ⟨eformal⟩ is a symbol that also occurs in ⟨formals⟩, an error is signaled.

A vau expression evaluates to an operative; an operative created in this way is said to be *compound*. The environment in which the vau expression was evaluated is remembered as part of the compound operative, called the compound operative's *static environment*. When the compound operative is later called with an object and an environment (as per §3.3), here called respectively the *operand tree* and the *dynamic environment*,

1. A new, initially empty environment is created, with the static environment as its parent. This will be called the *local environment*.

2. A stored copy of the formal parameter tree ⟨formals⟩ is matched in the local environment to the operand tree, locally binding the symbols of ⟨formals⟩ to the corresponding parts of the operand tree. (The matching process was defined in §4.9.1, for operative ***$define!*** .) ⟨eformal⟩ is matched to the dynamic environment; that is, if ⟨eformal⟩ is a symbol then that symbol is bound in the local environment to the dynamic environment.

3. A stored copy of the expression ⟨expr⟩ is evaluated in the local environment as a tail context (§3.10).

(On the copying of objects ⟨formals⟩ and ⟨expr⟩, see under *Immutability*, below.)

This operative will be generalized to allow an arbitrary list of expressions to be evaluated sequentially in the local environment in §5.3.1.

Rationale:

Without the ability to ignore the dynamic environment, every compound combiner application would create a local environment containing a reference to the dynamic environment of the combination. Consequently, the dynamic environment of a tail call wouldn't become garbage (in a straightforward implementation) until the call returned. Proper tail recursion would thus be undermined.

The central constructor of compound combiners for Kernel was named with a classical Greek letter in imitation of the traditional Lisp constructor ***lambda***. The letter *vau* was originally chosen because it is the immediate ancestor of the Roman letter *f*, as in *special **f**orm*, and also in part because, in comparison to other classical Greek letters, it is relatively unencumbered by competing uses. Oddly, it was not observed until long after the Kernel nomenclature had stabilized that, since the $ prefix was originally a stylized *s* as in ***s**pecial form*, the entire name `$vau` of Kernel's "constructor of special forms" is itself a roundabout acronym for ***s**pecial **f**orm*.

Immutability

When the ***$vau*** operative constructs a compound operative, it stores in that compound operative references to

- The dynamic environment in which ***$vau*** was invoked, which becomes the static environment of the compound operative, as explained above.

- ⟨eformal⟩.

- An immutable copy of the evaluation structure of ⟨formals⟩, as by applicative ***copy-es-immutable*** (§4.7.2).

- An immutable copy of the evaluation structure of ⟨expr⟩.

Rationale:

Immutable copies of the evaluation structures of ⟨formals⟩ and ⟨expr⟩ freeze the semantics of the compound operative when it is constructed. This provides stability —hence predictability— of behavior, facilitating both effective software engineering practice, and efficient implementation of calls to the compound operative after construction. Copying will often incur no additional cost at runtime, because objects ***load***ed from a source file already have immutable evaluation structures. If an operative with mutable behavior is desired, that is readily accomplished by means of bindings in its static environment.

Encapsulation of compound operatives

Because compound operatives are not a distinct type in Kernel, they are covered by the encapsulation of type *operative*. In particular, an implementation of Kernel cannot provide a feature that supports extracting the static environment of any given compound operative,[12] nor that supports determining whether or not a given operative is compound.

Rationale:

It's common practice to limit access to privileged information by exporting combiners from a local environment (§6.8.2). The exported combiners then have exclusive access to the information, because there is no way for anyone else to determine their static environment. This limitation is broadly dual to the limitation on the *environment* data type, noted in the rationale in §4.8, that a combiner cannot in general affect the parent of its dynamic environment. (That is, encapsulation of *operative* protects the called from the caller, while encapsulation of *environment* protects the caller from the called.)

For an example of the use of static environments to hide local state, see the rationale in §6.8.2.

4.10.4 `wrap`

(`wrap` *combiner*)

The ***wrap*** applicative returns an applicative whose underlying combiner is *combiner*.

Rationale:

As the primitive constructor for type *applicative*, ***wrap*** has the virtue of orthogonality with the primitive constructor for type *operative* (***$vau***, §4.10.3); whereas the more commonly used library constructor ***$lambda*** (§5.3.2) mixes concerns from both types.

[12] MIT Scheme provides a primitive procedure `procedure-environment` for extracting the static environment of a compound procedure.

4.10.5 `unwrap`

(`unwrap` *applicative*)

If *applicative* is not an applicative, an error is signaled. Otherwise, the ***unwrap*** applicative returns the underlying combiner of *applicative*.

Rationale:

It is *almost* possible to simulate the behavior of type *applicative* using only operatives, thus bypassing ***wrap*** and ***unwrap*** altogether. Given an operative O, one would simulate (***wrap*** O) by constructing a new operative O' that requires a list of operands, evaluates the operands in its dynamic environment, and passes them on to O.

The flaw in the simulation arises when one then attempts to simulate (***unwrap*** O'). One could construct an operative O'' that requires a list of operands, quotes them all, and passes them on to O'. When O' evaluates its operands, their evaluation removes the quotes, and the operands passed from O' to O are the same ones that were originally passed in to O''.

If O requires a list of operands, then O'' has the same behavior as O, modulo ***eq?***-ness of the pairs in the operand list. However, operatives do not necessarily require a list of operands. An *applicative* combination must have a list of operands, but that is only because the operands must be evaluated singly to produce arguments; there is no reason it should be inherent to the underlying operative, and Kernel maintains orthogonality between these two issues — evaluator handling of an applicative combination, and operative handling of its operand tree. The difference between the two is evident in the following example using the ***apply*** applicative (whose principal advantage is, as noted in §5.5.1, that it overrides the usual rule for evaluating arguments).

```
(apply ($lambda x x) 2)
```

In Kernel, this expression evaluates to the value `2`. Scheme disallows the expression (using `lambda` instead of `$lambda`, of course); but to do so it must artificially constrain either ***apply*** or the *procedure* type itself, neither of which has any inherent interest in the form of the second argument to ***apply***, diverting responsibility for the constraint away from the rule for applicative combination evaluation, where the limitation really is inherent. (Why would `($lambda x x)`, internally, care about the form of the value bound to `x`?)

5 Core library features (I)

This section describes some of the most basic library features of the core modules. They are presented in an order that allows each to be derived from those that precede it. The resulting order does not well respect the grouping of features into modules by type. Once these features have been derived, the remaining majority of core library features can and will be ordered by module in §6.

Rationale:

On the division of the language core into sections, see the rationale discussion at the beginning of §4.

5.1 Control

5.1.1 `$sequence`

`($sequence . ⟨objects⟩)`

The ***$sequence*** operative evaluates the elements of the list ⟨objects⟩ in the dynamic environment, one at a time from left to right. If ⟨objects⟩ is a cyclic list, element evaluation continues indefinitely, with elements in the cycle being evaluated repeatedly. If ⟨objects⟩ is a nonempty finite list, its last element is evaluated as a tail context. If ⟨objects⟩ is the empty list, the result is inert.

Rationale:

We prefer the name `$sequence` for this operative, as a forthright and precise description of what it does. The operative has had a variety of names in other Lisps. In standard Scheme, it has been called `begin` since the *R2RS* ([Cl85]), before which it was called `block` ([SteSu78, SusSte75]); but these names have mnemonic value primarily in reference to the Algol family of languages, from which we prefer to remain independent. In Common Lisp ([Ste90]) it is (and in MacLisp it was) called `progn`, because it returns the result of evaluating its nth operand, as opposed to, e.g., ***prog2*** which also evaluates the operands left-to-right but then returns the result from the 2nd operand. The mnemonic value of `progn` is thus largely dependent on its belonging to a set of similar names; and Kernel, which prefers to minimize its set of primitives, doesn't include any of the other operatives in the set. The name `sequence` was used in the first edition of the Wizard Book ([AbSu85]; but it was changed to `begin` in the second edition, [AbSu96], for compatibility with standard Scheme).

Derivation

The following expression defines ***$sequence*** using only previously defined features (which at this point means only core primitives).

```
($define! $sequence
   ((wrap ($vau ($seq2) #ignore
            ($seq2
               ($define! $aux
                  ($vau (head . tail) env
                     ($if (null? tail)
                          (eval head env)
                          ($seq2
                             (eval head env)
```

```
                        (eval (cons $aux tail) env)))))
              ($vau body env
                 ($if (null? body)
                      #inert
                      (eval (cons $aux body) env))))))

    ($vau (first second) env
       ((wrap ($vau #ignore #ignore (eval second env)))
        (eval first env)))))
```

Operative *$seq2* takes two operands and evaluates them in order from left to right, by exploiting eager argument evaluation, which guarantees that the expression `(eval first env)` in its body will be evaluated strictly prior to evaluation of `(eval second env)`. The capability to sequence two evaluations is then leveraged to create a recursive operative *$aux*, by defining it locally using *$define!* and then calling it; and *$aux* evaluates its arguments left-to-right, also by using *$seq2*.

This derivation has the useful property that every instance of symbol `$vau` in it is looked up at the time the derivation is evaluated. Consequently, when symbol `$vau` is rebound by the derivation of compound *$vau* in §5.3.1, that derivation can use this derived operative *$sequence* without causing infinite recursion.

For perspective, here is how the same derivation might be paraphrased if *$let*, *$letrec*, *$lambda*, and *apply* (§§5.10.1, 6.7.5, 5.3.2, 5.5.1) were available.

```
($define! $sequence
   ($let (($seq2  ($vau (first second) env
                     (($lambda #ignore (eval second env))
                        (eval first env)))))
      ($letrec ((aux  (wrap ($vau (head . tail) env
                        ($if (null? tail)
                             (eval head env)
                             ($seq2
                                (eval head env)
                                (apply aux tail env)))))))
         ($vau body env
            ($if (null? body)
                 #inert
                 (apply aux body env))))))
```

Note that the use of *$letrec* eliminates a mixture of sequencing with explicit call to *$define!* (which are the two devices that will be used to define *$letrec* in §6.7.5).

Rationale:

Eager argument evaluation affords unencumbered sequencing: operative

```
($vau (first second) env
   ((wrap ($vau #ignore #ignore (eval second env)))
      (eval first env)))
```

is reasonably construed to require *only* that the two arguments be evaluated sequentially in the dynamic environment, and the second result be returned. Sequencing can also be extracted from primitive operative *$if*, as via

```
($vau (first second) env
   (eval ($if (null? (eval first env)) second second) env))
```

but this carries the additional conceptual burden of requesting an irrelevant test on the result of the first operand evaluation.

Lazy argument evaluation was originally rejected for Scheme because when naively implemented it undermines proper tail recursion [SusSte75, p. 40]. For Kernel, this overtly practical consideration is still indirectly relevant, as naivety is preferred as a protection against accidents (*G3* of §0.1.2) in a language with side-effects.

5.2 Pairs and lists

5.2.1 list

(list . *objects*)

The *list* applicative returns *objects*.

The underlying operative of *list* returns its undifferentiated operand tree, regardless of whether that tree is or is not a list. The behavior of the applicative is therefore determined by the way the Kernel evaluator algorithm evaluates arguments; see §3.9.

Rationale:

Specifically excluding the underlying operative from the list constraint guarantees equivalence

$$(\textit{apply list } x) \equiv x$$

(See also the rationale discussion for *unwrap*, §4.10.5).

Derivation

The following expression defines the *list* applicative, using only previously defined features.

```
($define! list (wrap ($vau x #ignore x)))
```

Recall that mutability of constructed argument lists is guaranteed by the evaluator algorithm (§3.3), thus providing the correct behavior for this derivation without need for explicit use of *cons*.

The implementation would be even simpler if ***$lambda*** (§5.3.2) were available:

```
($define! list ($lambda x x))
```

5.2.2 list*

(list* . *objects*)

objects should be a finite nonempty list of arguments. The following equivalences hold:

(*list** *arg1*)	≡	*arg1*
(*list** *arg1* *arg2* . *args*)	≡	(*cons* *arg1* (*list** *arg2* . *args*))

For example,

(*list** 1)	⟹	1
(*list** 1 2)	⟹	(1 . 2)
(*list** 1 2 3)	⟹	(1 2 . 3)
(*list** 1 2 3 ())	⟹	(1 2 3)

Rationale:

It is fairly common —it happens several times in the library derivations in this report— that one wants to construct a list by prepending several elements onto the front of an existing list. We could write an expression such as

(append (list *x* *y* *z*) *list*)

This is a rather indirect way to express the intended operation, since we really have no interest in constructing a list of the elements x, y, z. Alternatively, we could write

(cons *x* (cons *y* (cons *z* *list*)))

This is more direct, but harder to read; we have lost sight of the unity of the overall operation we are performing.

R5RS Scheme provides a shorthand for such a construction in the special case that the intended use of the new list is to apply a procedure to it. The shorthand uses combinations of the form

(apply *proc* *x* *y* *z* *list*)

From a Kernel design perspective, this shorthand has two basic drawbacks. It complicates the semantics of *apply*, thus lacks simplicity; while, as primary support for multiple-prepend, it lacks generality. Kernel cleanly separates the operations of application and multiple-prepend, by eliminating the extended form of *apply* from the *R5RS* and introducing instead the *list** applicative.

*list** requires a finite nonempty list of arguments because its behavior is defined by the way it treats its *last* argument differently (per the rationale discussion in §3.9; contrast applicative *list*, §5.2.1).

Derivation

The following expression defines the *list** applicative, using only previously defined features.

```
($define! list*
   (wrap ($vau args #ignore
            ($sequence
               ($define! aux
                  (wrap ($vau ((head . tail)) #ignore
                           ($if (null? tail)
                                head
                                (cons head (aux tail))))))
               (aux args)))))
```

This could be implemented more cleanly if *$lambda* (§5.3.2) and *apply* (§5.5.1) were available:

```
($define! list*
   ($lambda (head . tail)
      ($if (null? tail)
           head
           (cons head (apply list* tail)))))
```

Rationale:

Like many library derivations in the report, this one (written either way) isn't robust, because it fails to signal a type error; in this case, if the argument list is cyclic the compound applicative will loop through it indefinitely (until the implementation runs out of memory, or, perhaps, until somebody decides the program has hung and kills it).

5.3 Combiners

5.3.1 `$vau`

`($vau` ⟨formals⟩ ⟨eformal⟩ `.` ⟨objects⟩`)`

This operative generalizes primitive operative *$vau*, §4.10.3, so that the constructed compound operative will evaluate a sequence of expressions in its local environment, rather than just one.

⟨formals⟩ and ⟨eformal⟩ should be as for primitive operative *$vau*. If ⟨objects⟩ has length exactly one, the behavior is identical to that of primitive *$vau*. Otherwise, the expression

`(`*$vau* ⟨x⟩ ⟨y⟩ `.` ⟨z⟩`)`

is equivalent to

`(`*$vau* ⟨x⟩ ⟨y⟩ `(`*$sequence* `.` ⟨z⟩`))`

Derivation

The following expression defines the *$vau* library operative using only previously defined features.

```
($define! $vau
   ((wrap ($vau ($vau) #ignore
             ($vau (formals eformal . body) env
                (eval (list $vau formals eformal
                           (cons $sequence body))
                      env))))
      $vau))
```

This could be implemented somewhat more cleanly if *$let* (§5.10.1) were available:

```
($define! $vau
   ($let (($vau $vau)) ; save the primitive
      ($vau (formals eformal . body) env
         (eval (list $vau formals eformal
                     (cons $sequence body))
               env))))
```

5.3.2 $lambda

($lambda ⟨formals⟩ . ⟨objects⟩)

⟨formals⟩ should be a formal parameter tree as described for operative *$define!*, §4.9.1.

The expression

(*$lambda* ⟨formals⟩ . ⟨objects⟩)

is equivalent to

(*wrap* (*$vau* ⟨formals⟩ #ignore . ⟨objects⟩))

Rationale:

In ordinary Kernel programming, applicatives that care about their dynamic environment are rare. Moreover, such applicatives pose a potential hazard to proper tail recursion (as noted in §4.10.3); so, in the interest of making dangerous things difficult to do by accident (*G3* of §0.1.2), *$lambda* constructs compound applicatives that ignore their dynamic environments, encouraging the programmer to flag out the rare dynamic applicatives by explicit use of *wrap* and *$vau*. See, for example, §6.7.2.

Derivation

The following expression defines the ***$lambda*** operative, using only previously defined features.

```
($define! $lambda
   ($vau (formals . body) env
      (wrap (eval (list* $vau formals #ignore body)
                  env))))
```

5.4 Pairs and lists

5.4.1 car, cdr

(car *pair*)
(cdr *pair*)

Both applicatives require *pair* to be a pair, else an error is signaled. They return, respectively, the car and cdr of *pair*.

Rationale:

Feature names car/cdr are sometimes criticized on grounds of non-mnemonicity. These names are among the most universally supported in the Lisp language family, currently supported by all the Scheme versions (including *R6RS*) and Common Lisp. The non-mnemonicity criticism is therefore somewhat dubious within the context of the Lisp language family. Even the objection that they have no meaning outside the Lisp community is somewhat undermined by their occasional occurrence, as common nouns, in discussions of lists in non-Lisp languages. General ergonomic/psychological properties of these names are placed in the broader context of linguistic vocabularies by the rationale discussion in §1.3.7.

Specific possible alternative names typically have mnemonicity problems of their own. For example, either first/rest or head/tail would be a conceptual type error, because they suggest parts of a list — which may be the data type that pairs are most often used to represent, but in fact the features described here are accessors for type *pair*, not type *list*. At an opposite extreme, left/right are so general in meaning that they are probably best left available for local use in specific situations.

Derivation

The following expressions define the ***car*** and ***cdr*** applicatives, using only previously defined features.

```
($define! car ($lambda ((x . #ignore)) x))
($define! cdr ($lambda ((#ignore . x)) x))
```

5.4.2 caar, cadr, ... cddddr

(caar *pair*)
...
(cddddr *pair*)

These applicatives are compositions of ***car*** and ***cdr***, with the a's and d's in the same order as they would appear if all the individual car's and cdr's were written out in prefix order. Arbitrary compositions up to four deep are provided. There are twenty-eight of these applicatives in all.

Rationale:

These tools are an asset when one wants to explicitly access components of a tree (here, any subtree in the first four generations below the root). They compactly and straightforwardly articulate element selections that, if expressed as compositions of ***car*** and ***cdr***, would be quite bulky. The tools are therefore well worth including in the language — but, interestingly enough, they are not often wanted, nor are ***car*** and ***cdr*** themselves. Experience programming in Kernel suggests that Kernel formal parameter trees (in various constructs, especially ***$let***) are a more lucid idiom for many of the situations where a Scheme programmer might resort to explicit tree selectors. See for example the derivations of ***$cond***, §5.6.1; for perspective, contrast the derivation of ***$let***, §5.10.1.

Derivation

There are a couple of obvious ways to define ***caar*** ... ***cddddr*** using previously defined features. The naive way would be by explicit compositions, as follows.

```
($define! caar ($lambda (x) (car (car x))))
($define! cadr ($lambda (x) (car (cdr x))))
...
($define! cddddr ($lambda (x) (cdr (cdr (cdr (cdr x))))))
```

The more sophisticated alternative is to use deep formal parameter trees, as in the derivations of ***car*** and ***cdr*** in §5.4.1:

```
($define! caar ($lambda (((x . #ignore) . #ignore)) x))
($define! cadr ($lambda ((#ignore x . #ignore)) x))
...
($define! cddddr ($lambda
   ((#ignore #ignore #ignore #ignore . x)) x))
```

5.5 Combiners

5.5.1 apply

(apply *applicative object environment*)
(apply *applicative object*)

When the first syntax is used, applicative ***apply*** combines the underlying combiner of *applicative* with *object* in dynamic environment *environment*, as a tail context. The expression

(***apply*** *applicative object environment*)

is equivalent to

(***eval*** (***cons*** (***unwrap*** *applicative*) *object*) *environment*)

The second syntax is just syntactic sugar; the expression

(***apply*** *applicative object*)

is equivalent to

(***apply*** *applicative object* (***make-environment***))

Derivation

The following expression defines ***apply*** using only previously defined features.

```
($define! apply
   ($lambda (appv arg . opt)
      (eval (cons (unwrap appv) arg)
            ($if (null? opt)
                 (make-environment)
                 (car opt)))))
```

Rationale:

The ***apply*** applicative is designed to guarantee equivalence between the following two expressions, for all choices of ⟨operator⟩ that do not cause an error when evaluating either expression.

(⟨operator⟩ . ⟨operands⟩)

(***apply*** ⟨operator⟩ (***list*** . ⟨operands⟩) (***get-current-environment***))

In order for the first expression to evaluate without error, ⟨operator⟩ must evaluate to a combiner. If it evaluates to an operative, in the first expression ⟨operands⟩ will be passed to it unevaluated; but in the second expression, the operands are always evaluated. So these two expressions cannot be equivalent unless the first argument to ***apply*** is an applicative, and the behavior of ***apply*** is defined such that a type error in the first argument must be signaled.

Although the above equivalence is a useful litmus test for the well-behavedness of ***apply***, the principle advantage of ***apply*** is that it provides a convenient way to override the usual rule for evaluating arguments (§3.3) with an arbitrary alternative computation. There is no similar advantage to an analogous ***operate*** applicative for use with operatives, because the operands of an operative aren't evaluated anyway, so there is no usual rule to override. Looking at it from another angle, if there *were* an ***operate*** applicative, its litmus test would be equivalence of expressions

(⟨operator⟩ . ⟨operands⟩)

(*operate* ⟨operator⟩ (*$quote* ⟨operands⟩) (*get-current-environment*))

(where *$quote* ≡ (*$vau* (x) #ignore x)), and the general behavior of ***operate*** would be defined by stipulating that

(*operate* c t e)

is equivalent to

(*eval* (*cons* c t) e)

which in turn is equivalent to

(*apply* (*wrap* c) t e)

The general equivalence is true for *all* combiners c, and the natural behavior for ***operate*** would make the litmus equivalence true for all combiners as well. So nothing about the behavior of ***operate*** is specific to operatives, and it really ought to be called ***combine***; but at that point, why bother with it at all? It *isn't* the analog for operatives of ***apply*** for applicatives, and its implementation is so simple that using it would only serve to slightly obscure what is actually being done.

The use of an empty environment as the default for ***apply*** is motivated by accident avoidance (*G3* of §0.1.2). (On avoiding environment-capture, see also ***$provide!***, §6.8.2.)

5.6 Control

5.6.1 $cond

($cond . ⟨clauses⟩)

⟨clauses⟩ should be a list of clause expressions, each of the form (⟨test⟩ . ⟨body⟩), where ⟨body⟩ is a list of expressions.

The expression

(*$cond* (⟨test⟩ . ⟨body⟩) . ⟨clauses⟩)

is equivalent to

```
($if ⟨test⟩ ($sequence . ⟨body⟩) ($cond . ⟨clauses⟩))
```

while the expression `($cond)` is equivalent to `#inert`.

Rationale:

Because evaluation of test expressions may have side-effects, the policy on handling of cyclic lists (rationale in §3.9) requires that, if ⟨clauses⟩ is a cyclic list, ***$cond*** will continue looping through it indefinitely until some test evaluates to true. Note how this policy harmonizes with the assumption that, if a cyclic list of clauses is passed to ***$cond***, it was done deliberately. The programmer would hardly go to such trouble to induce a dynamic error; and if ***$cond*** were simply going to terminate after testing each clause once, that could be achieved far more easily with a finite list of clauses. So the only reason to use a cyclic list of clauses would be to induce indefinite looping.

In *R5RS* Scheme, the test on the last clause of a ***cond*** expression may be replaced by the keyword `else`. This is something like a local second-class special form operator (flouting *G1a* of §0.1.2); and, moreover, it would encourage the embedding of unevaluated symbols in expressions constructed for evaluation, and thus (indirectly) the likelihood of accidental bad hygiene (flouting *G3* of §0.1.2); so Kernel omits it. The same effect can be achieved at least as clearly, and more uniformly, by specifying `#t` as the ⟨test⟩ on the last clause.

In the no-clause base case (which comes into play whenever ⟨clauses⟩ is acyclic and all the ⟨test⟩'s evaluate to false), the Kernel analog of traditional Lisp (particularly, Scheme) behavior is to return `#inert`; while the most straightforward alternative would be to signal an error. When ***$cond*** is called for effect, the programmer would prefer the former; when for value, the latter. If the language provided two separate operatives for those cases, ***$cond-for-effect*** would always return `#inert`, regardless of whether any clause is fired; while ***$cond-for-value*** would require at least one clause, and signal an error if no clause is fired.

However, if ***$cond*** were split in two for effect/value, uniformity of design would suggest splitting *every* standard operative that performs a ⟨body⟩, starting with ***$vau***, and notably including the entire ***$let*** family, bloating the language vocabulary. Returning `#inert` for value is dangerous, and relatively easy to do if it's the base case for ***$cond*** (hence, disfavored by *G3* of §0.1.2); but that base-case behavior does have the virtue of admitting both for-effect and for-value use. So, until some major new insight recommends itself, we prefer to follow the traditional behavior.

Derivation

The following expression defines the ***$cond*** operative, using only previously defined features.

```
($define! $cond
   ($vau clauses env
```

```
    ($define! aux
       ($lambda ((test . body) . clauses)
          ($if (eval test env)
               (apply (wrap $sequence) body env)
               (apply (wrap $cond) clauses env))))

    ($if (null? clauses)
         #inert
         (apply aux clauses))))
```

The auxiliary applicative ***aux*** could be cleanly eliminated if ***$let*** (§5.10.1) were available:

```
($define! $cond
   ($vau clauses env
      ($if (null? clauses)
           #inert
           ($let ((((test . body) . clauses)  clauses))
              ($if (eval test env)
                   (apply (wrap $sequence) body env)
                   (apply (wrap $cond) clauses env))))))
```

5.7 Pairs and lists

5.7.1 `get-list-metrics`

(`get-list-metrics` *object*)

By definition, an *improper list* is a data structure (per §3.9) whose objects are its start together with all objects reachable from the start by following the cdr references of pairs, and whose internal references are just the cdr references of its pairs. Every object, of whatever type, is the start of an improper list. If the start is not a pair, the improper list consists of just that object.

The *acyclic prefix length* of an improper list L is the number of pairs of L that a naive traversal of L would visit only once. The *cycle length* of L is the number of pairs of L that a naive traversal would visit repeatedly. Two improper lists are structurally isomorphic (§3.9) iff they have the same acyclic prefix length and cycle length and, if they are terminated by non-pair objects rather than by cycles, the non-pair objects have the same type.

Applicative ***get-list-metrics*** constructs and returns a list of exact integers of the form `(`p n a c`)`, where p, n, a, and c are, respectively, the number of pairs in, the number of nil objects in, the acyclic prefix length of, and the cycle length of, the improper list starting with *object*. n is either 0 or 1, $a + c = p$, and n and c cannot both be non-zero. If $c = 0$, the improper list is acyclic; if $n = 1$, the improper list is a finite list; if $n = c = 0$, the improper list is not a list; if $a = c = 0$, *object* is not a pair. (Lists are defined in §3.9.)

Rationale:

The classical Lisp idiom of *cdr*ing down a list is inadequate if the list may be cyclic and must be processed in finite time. Often, the programmer can be insulated from the danger of cycles by intermediate-level tools, such as *map* (§5.9.1), that handle cyclic lists robustly (as in the derivation of *combiner?*, §6.2.1). *get-list-metrics* is a lower-level tool for those contingencies that the intermediate-level tools don't cover. It provides a complete characterization of the shape of the list, in a format suitable for detailed analysis (as in the derivation of *length*, §6.3.1, or, most elaborately, in the derivation of *map*).

Actually, most of the (by intent, very few) contingencies that escape the intermediate-level tools will only need to bound their list traversal by the number of pairs in the list, and won't even care whether the list is cyclic. Some will care about the prefix/cycle breakdown, though; and in the acyclic case, it's trivial for *get-list-metrics* to determine whether or not the terminator is nil, whereas fetching the same information later would require more effort.

In theory, *get-list-metrics* doesn't provide any capability that the programmer wouldn't be able to reproduce without it (i.e., it's a *library* feature); but in practice, the programmer caught without it would find it troublesome to reimplement from scratch (easy to get wrong, thus contrary to the spirit of *G3* of §0.1.2).

Derivation

The following expression defines the *get-list-metrics* applicative, using previously defined features, and number primitives (§12).

```
($define! get-list-metrics
   ($lambda (ls)

      ($define! aux
         ($lambda (kth k nth n)
            ($if (>=? k n)
                 ($if (pair? (cdr nth))
                      (aux ls 0 (cdr nth) (+ n 1))
                      (list (+ n 1)
                            ($if (null? (cdr nth)) 1 0)
                            (+ n 1)
                            0))
                 ($if (eq? kth nth)
                      (list n 0 k (- n k))
                      (aux (cdr kth) (+ k 1) nth n)))))
```

```
      ($if (pair? ls)
           (aux ls 0 ls 0)
           (list 0 ($if (null? ls) 1 0) 0 0))))
```

Rationale:

For expository purposes, the above derivation has two merits: it's (relatively) simple, and it works. Unfortunately, it also takes quadratic time in the number of pairs in the list, because it compares each pair to *all* of its predecessors before moving on to the next pair.

A linear-time (but also more complicated) alternative is to compare the n^{th} pair only to the $(2^{\lfloor \log_2 (n-1) \rfloor})^{\text{th}}$ element. (Thus, element 2 is compared to element 1, elements 3–4 to element 2, 5–8 to 4, etc.) Any cycle will eventually be detected along with its exact length, though perhaps not until after the traversal has been looping through it for a while; and we can then go back to the beginning of the list and work forward, knowing the cycle length, to find the point where the cycle is first entered.

Other kinds of cycle-handling traversal may achieve asymptotic speed-up by temporarily marking visited objects, but list traversal doesn't need marking to achieve linear time because the structure being traversed is innately linear. The non-marking algorithm outlined above might even be faster than marking, since cache considerations can make writing memory much more expensive than reading it.

5.7.2 `list-tail`

`(list-tail` *object integer*`)`

integer should be exact and non-negative. The ***list-tail*** applicative follows *integer* cdr references starting from *object*. Thus, *object* must be the start of an improper list containing at least *integer* pairs. The following equivalences hold:

(***list-tail*** *object* 0) ≡ *object*
(***list-tail*** *object* (+ *k* 1)) ≡ (***list-tail*** (***cdr*** *object*) *k*)

Derivation

The following expression defines the ***list-tail*** applicative, using previously defined features, and number primitives (§12).

```
($define! list-tail
   ($lambda (ls k)
      ($if (>? k 0)
           (list-tail (cdr ls) (- k 1))
           ls)))
```

5.8 Pair mutation (optional)

5.8.1 `encycle!`

`(encycle!` *object integer1 integer2*`)`

integer1 and *integer2* should be exact and non-negative. The improper list starting at *object* must contain at least *integer1* + *integer2* pairs.

If *integer2* = 0, the applicative does nothing. If *integer2* > 0, the applicative mutates the improper list starting at *object* to have acyclic prefix length *integer1* and cycle length *integer2*, by setting the cdr of the $(\textit{integer1} + \textit{integer2})^{\text{th}}$ pair in the list to refer to the $(\textit{integer1} + 1)^{\text{th}}$ pair in the list.

The result returned by ***encycle!*** is inert.

Cf. ***get-list-metrics***, §5.7.1.

Rationale:

This tool complements ***get-list-metrics***. The general idiom for using them is to measure the input with ***get-list-metrics***, perform one's operations (whatever they are) robustly by counting pairs rather than expecting a null terminator, assemble an acyclic list of output elements, and encycle the output list just before returning it. If it were really that simple, of course, the client programmer would be using ***map***, §5.9.1, instead of fussing with list metrics. ***encycle!*** isn't provided to the programmer to manage an anticipated situation; it's provided in the hope that, by naturally complementing ***get-list-metrics***, it will help to manage *un*anticipated situations.

Derivation

The following expression defines the ***encycle!*** applicative, using previously defined features, and number primitives (§12).

```
($define! encycle!
   ($lambda (ls k1 k2)
      ($if (>? k2 0)
           (set-cdr! (list-tail ls (+ k1 k2 -1))
                     (list-tail ls k1))
           #inert)))
```

5.9 Combiners

5.9.1 `map`

`(map` *applicative* . *lists*`)`

lists must be a nonempty list of lists; if there are two or more, they must all have the same length (§6.3.1). If *lists* is empty, or if all of its elements are not lists of the same length, an error is signaled.

The *map* applicative applies *applicative* element-wise to the elements of the lists in *lists* (i.e., applies it to a list of the first elements of the lists, to a list of the second elements of the lists, etc.), using the dynamic environment from which *map* was called, and returns a list of the results, in order. The applications may be performed in any order, as long as their results occur in the resultant list in the order of their arguments in the original lists.

If *lists* is a cyclic list, each argument list to which *applicative* is applied is structurally isomorphic to *lists*. If any of the elements of *lists* is a cyclic list, they all must be, or they wouldn't all have the same length. Let $a_1 \dots a_n$ be their acyclic prefix lengths, and $c_1 \dots c_n$ be their cycle lengths. The acyclic prefix length a of the resultant list will be the maximum of the a_k, while the cycle length c of the resultant list will be the least common multiple of the c_k. In the construction of the result, *applicative* is called exactly $a + c$ times.

Cf. applicative *for-each*, §6.9.1.

Rationale:

The *map* applicative is designed to guarantee equivalence between the following two expressions, for $n > 0$ and $m \geq 0$.

$$
\begin{array}{ll}
& (\textit{map}\ \ c\ \ (\textit{list}\ \ a_{1,1} \dots a_{1,m}) \\
& \qquad\qquad \vdots \\
& \qquad\qquad (\textit{list}\ \ a_{n,1} \dots a_{n,m})) \\
\equiv & \\
& (\textit{list}\ \ (c\ \ a_{1,1}\ \dots a_{n,1}\ ) \\
& \qquad\qquad \vdots \\
& \qquad\quad (c\ \ a_{1,m} \dots a_{n,m}))
\end{array}
$$

This equivalence implies that the first argument to *map* must be an applicative, never an operative, by the same reasoning as for *apply*, §5.5.1. It also requires *map* to use its own dynamic environment for the applications it performs, since that is the dynamic environment of the combinations in the second half of the equivalence.

The decision that *map* will not work with an operative does not cause any practical difficulty because, in the unusual but certainly conceivable event that the programmer wants to combine an operative element-wise with the elements of one or more lists, this effect is readily achieved by simply *wrap*ping the operative, thus:

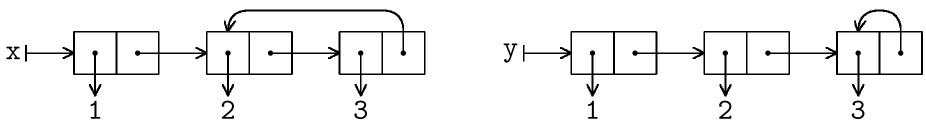
(map (wrap *operative*) . *lists*)

The treatment of dynamic environments differs from that of *apply*. Both behaviors are based on the governing principle of accident avoidance (*G3* of §0.1.2); in differentiating the two, the decisive factor was whether there would be a danger of contradicting the programmer's expectation. *map* is conceptually a way of constructing a series of ordinary applicative combinations, and ordinary applicative combinations can access their dynamic environments. *apply*, on the other hand, is principally a vehicle for calling applicatives *abnormally*, overriding the usual rule for argument evaluation. Hence, the programmer who uses *apply* does not have an expectation of normalcy; in fact, in derivations in this report, *apply* is rarely called *without* specifying its optional environment argument. The default behavior for *apply* is therefore treated as a conscious decision, by definition not an accident, and is chosen for safety on that basis. (See the rationale for *apply*.)

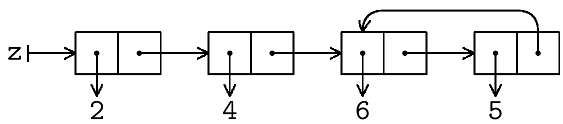
When the list arguments are cyclic, the metrics (acyclic prefix length, cycle length) of the result list are chosen so that the result list will have the smallest number of pairs that will produce the correct infinite sequence of application results — *if* the mapped applicative is well-behaved, always producing the same result when applied to the same arguments.

As a simple example of the treatment of cyclic list metrics, consider mapping applicative *+* onto the following two cyclic lists.

```
($define! x (list 1 2 3))
(append! x (cdr x))
($define! y (list 1 2 3))
(append! y (cddr y))
```




```
($define! z (map + x y))
```



The infinite sequences represented by these lists are:

```
x : (1 2 3 2 3 2 3 ...)
y : (1 2 3 3 3 3 3 ...)
z : (2 4 6 5 6 5 6 ...)
```

If *map* were to allow its list arguments to have different lengths, there are a bewildering variety of ways it might handle that case. Quite a lot of them were considered for Kernel, starting with those that preserve the formulae developed for cyclic lists: maximum of acyclic prefix lengths, least common multiple of cycle lengths. However, each approach was found to have anomalies in its behavior, and eliminating one anomaly usually produced several others. It was finally decided that, since the behavior of *map* is really well-defined only when the list arguments are congruent, generalizations beyond that behavior should be viewed with extreme skepticism. The handling of uniformly cyclic list arguments is not too much of a stretch, since the result list can still preserve the sequence of results in the unbounded parallel traversal of the list arguments, as noted above; but when one of the lists runs out of elements while another does not, any behavior other than signaling an error is potentially unexpected, hence facilitates accidents (*G3* of §0.1.2).

Having decided to disallow differently lengthed list arguments, we have the invariant that the length of the result list is the length of each of the list arguments; but then, if no list arguments are provided, there is no way to choose a result length that will be compatible with all nontrivial cases: the result length would have to be simultaneously equal to every nonnegative integer, and to positive infinity as well. Hence the requirement that *map* signal an error when given fewer than two arguments.

Derivation

The following expression defines *map* using previously defined features, and number features (§12; not all are primitive, but there is no circular dependency). Note that this is, easily, the most complicated derivation in the language core.

```
($define! map
   (wrap ($vau (appv . lss) env

      ($define! acc
         ($lambda (input (k1 k2) base-result head tail sum)
            ($define! aux
               ($lambda (input count)
                  ($if (=? count 0)
                       base-result
                       (sum (head input)
                            (aux (tail input) (- count 1))))))
            (aux input (+ k1 k2))))

      ($define! enlist
         ($lambda (input ms head tail)
            ($define! result (acc input ms () head tail cons))
            (apply encycle! (list* result ms))
            result))

      ($define! mss (cddr (get-list-metrics lss)))

      ($define! cars ($lambda (lss) (enlist lss mss caar cdr)))
      ($define! cdrs ($lambda (lss) (enlist lss mss cdar cdr)))

      ($define! result-metrics
         (acc lss mss (cddr (get-list-metrics (car lss)))
              ($lambda (lss) (cddr (get-list-metrics (car lss))))
              cdr
              ($lambda ((j1 j2) (k1 k2))
                 (list (max j1 k1)
```

```
                        ($cond ((=? j2 0)  k2)
                               ((=? k2 0)  j2)
                               (#t  (lcm j2 k2)))))))

    (enlist lss
            result-metrics
            ($lambda (lss) (apply appv (cars lss) env))
            cdrs))))
```

This applicative cannot be constructed using *$lambda*, because it needs to access its dynamic environment.

The required error signaling has been arranged without explicit use of error continuations by tweaking the algorithm for computing the cycle length of the result, so that *map* is certain to run off the end of some list unless they are all the same length. If two finite lists have different lengths, the acyclic prefix length will be at least the greater of them; and if one list is finite while another is cyclic, the cycle length will be non-zero and *map* will try to build a nontrivial cycle *following* an acyclic prefix at least as long as the finite list (due to the way *lcm*, §12.5.14, treats zeros).

For readers with a passing familiarity with category theory, it may be of interest to note that auxiliary applicative *acc* is approximately the counit of the usual adjunction from **Set** to **Mon** — of which *map* is, approximately, the unit.

Rationale:

There are two reasons to include a library feature in this section (§5). The first reason is that a number of other library features scattered through the core modules cannot readily be derived without it, so that trying to fit the feature into §6 would interfere with the modular presentation of that section. The second reason is that it facilitates the derivation of another feature with a prior reason to be included in this section. (For purposes of inclusion we treat the entire family of applicatives *caar*... *cddddr*, §5.4.2, as a single feature.)

map is the focal point for both of these reasons for inclusion. Every other feature in this section contributes, directly or indirectly, to its derivation — except for *$let*, §5.10.1, which *would* have facilitated a number of the earlier features if only it had been available. *$let* is probably more ubiquitous than any other feature in this section except *$lambda*; it is left to the end of the section only because, owing to the way it handles cyclic binding lists, its derivation would be far more disarranged by not being able to use *map*, than earlier derivations are by not being able to use *$let*. But once *map* has been implemented, everything else is "easy".

5.10 Environments

5.10.1 $let

(`$let` ⟨bindings⟩ `.` ⟨objects⟩)

$\langle$bindings$\rangle$ should be a finite list of formal-parameter-tree/expression pairings, each of the form `(`$\langle$formals$\rangle$ $\langle$expression$\rangle$`)`, where each $\langle$formals$\rangle$ is a formal parameter tree as described for the ***$define!*** operative, §4.9.1, and no symbol occurs in more than one of the $\langle$formals$\rangle$.

The expression

`(`*`$let`* `((`$\langle$form$_1\rangle$ $\langle$exp$_1\rangle$`)` ... `(`$\langle$form$_n\rangle$ $\langle$exp$_n\rangle$`))` . $\langle$objects$\rangle$`)`

is equivalent to

`((`*`$lambda`* `(`$\langle$form$_1\rangle$ ... $\langle$form$_n\rangle$`)` . $\langle$objects$\rangle$`)` $\langle$exp$_1\rangle$ ... $\langle$exp$_n\rangle$`)`

Thus, the $\langle$exp$_k\rangle$ are first evaluated in the dynamic environment, in any order; then a child environment e of the dynamic environment is created, with the $\langle$form$_k\rangle$ matched in e to the results of the evaluations of the $\langle$exp$_k\rangle$; and finally the subexpressions of $\langle$objects$\rangle$ are evaluated in e from left to right, with the last (if any) evaluated as a tail context, or if $\langle$objects$\rangle$ is empty the result is inert.

Rationale:

Because the $\langle$exp$_k\rangle$ are evaluated in the dynamic environment of the call to *$let* but matched in a child of that environment, the $\langle$exp$_k\rangle$ can't see each other's bindings; and if any $\langle$exp$_k\rangle$ is a *$vau* or *$lambda* expression, the resulting combiner can't be recursive because it can't see its own binding.

These constraints may sometimes be intended, or at least unproblematic. For occasions when they are not wanted, three variants are provided: *$let** (§6.7.4), *$letrec* (§6.7.5), and *$letrec** (§6.7.6). The `*` variants evaluate the $\langle$exp$_k\rangle$ from left to right, and allow each to see the results of its predecessors. The `rec` variants support recursive combiners by matching each $\langle$form$_k\rangle$ in the same environment where $\langle$exp$_k\rangle$ was evaluated.

All operatives in the *$let* family consider it an error for $\langle$bindings$\rangle$ to be a cyclic list. In all cases, this is because the behavior of the operative is undefined for cyclic $\langle$bindings$\rangle$ (per the general policy on handling cyclic lists, in the rationale discussion for §3.9); but the undefinedness has two different sources. In the unordered variants (principally *$let* and *$letrec*), the structure of $\langle$bindings$\rangle$ is mapped onto a single formal parameter tree, so the acyclicity constraint on $\langle$bindings$\rangle$ follows from the acyclicity constraint on formal parameter trees (§4.9.1). In the ordered variants (*$let** and *$letrec**), the ordered sequence of bindings is followed by expression sequence $\langle$objects$\rangle$, and their chronological concatenation is undefined when the former sequence is cyclic by the same reasoning that forbids cyclic non-final arguments to applicative *append* (§6.3.3).

Derivation

The following expression defines *$let* using only previously defined features.

```
($define! $let
   ($vau (bindings . body) env
      (eval (cons (list* $lambda (map car bindings) body)
```

```
                    (map cadr bindings))
          env)))
```

Rationale:

Note that coping with cyclic binding lists is hidden seamlessly within the calls to *map*. This is why *$let* is withheld to the end of the section, even though it would have simplified several of the earlier derivations (see, for example, the alternative derivation of *$cond*, §5.6.1): without *map*, *$let* would have to implement a subset of the same functionality itself — which would be appallingly difficult unless one used the same tools as *map* itself, in which case *$let* would arrive too late to be used for anything else except *map* itself.

6 Core library features (II)

This section describes the remaining library features of the core modules, after those few library features that were isolated in §5. The features in this section are arranged by module.

Rationale:

On the division of the language core into sections, see the rationale discussion at the beginning of §4.

6.1 Booleans

6.1.1 `not?`

`(not?` *boolean*`)`

Applicative *not?* is a predicate that returns the logical negation of its argument.

Derivation

The following expression defines the *not?* applicative, using only previously defined features.

```
($define! not? ($lambda (x) ($if x #f #t)))
```

6.1.2 `and?`

`(and? .` *booleans*`)`

Applicative *and?* is a predicate that returns true unless one or more of its arguments are false.

Rationale:

Because ***and?*** doesn't process its arguments in any particular order, it must terminate in finite time even if *arguments* is cyclic (per the rationale discussion in §3.9).

Returning true on zero arguments is deemed the most uniform, hence least error-prone, behavior for that case. One measure of its uniformity is that it allows the entire behavior of the applicative, including the zero-argument case, to be captured by a very simple statement (above). Another is that that it preserves the following equivalence:

(***and?*** *h* . *t*) ≡ (***and?*** *h* (***and?*** . *t*))

Derivation

The following expression defines the ***and?*** applicative, using only previously defined features.

```
($define! and?
   ($lambda x

      ($define! aux
         ($lambda (x k)
            ($cond ((<=? k 0)  #t)
                   ((car x)    (aux (cdr x) (- k 1)))
                   (#t         #f))))

      (aux x (car (get-list-metrics x)))))
```

6.1.3 `or?`

(`or?` . *booleans*)

Applicative ***or?*** is a predicate that returns false unless one or more of its arguments are true.

Rationale:

See the rationale for ***and?***, §6.1.2. The equivalence preserved here is:

(***or?*** *h* . *t*) ≡ (***or?*** *h* (***or?*** . *t*))

Derivation

The following expression defines the ***or?*** applicative, using only previously defined features.

```
($define! or?
   ($lambda x
      (not? (apply and? (map not? x)))))
```

6.1.4 `$and?`

`($and? . ⟨objects⟩)`

The ***$and?*** operative performs a "short-circuit and" of its operands: It evaluates them from left to right, until either an operand evaluates to false, or the end of the list is reached. If the end of the list is reached (which is immediate if ⟨objects⟩ is nil), the operative returns true. If an operand evaluates to false, no further operand evaluations are performed, and the operative returns false. If ⟨objects⟩ is acyclic, and the last operand is evaluated, it is evaluated as a tail context (§3.10). If ⟨objects⟩ is cyclic, an unbounded number of operand evaluations may be performed.

If any of the operands is evaluated to a non-*boolean* value, it is an error; and if the operand evaluated to a non-*boolean* is not the last operand, the error is signaled.

Cf. the ***and?*** applicative, §6.1.2.

Rationale:

Because the behavior of ***$and?*** is still definable when ⟨objects⟩ is cyclic, and the operands are processed in a fixed order, and the processing of any operand may have side-effects, ***$and?*** continues processing operands in a cyclic list indefinitely (per the rationale discussion in §3.9).

When a compound computation has a well-defined last unbounded step, and the result of the last step is to be the result of the larger computation, it would be counterintuitive (hence error-prone) for that last step not to be a tail context. In the case of ***$and?***, the results of all but the last operand evaluation are used internally as booleans, and we require type errors on these values to be signaled, to protect against degradation of the role of type *boolean* (on which, see the rationale in §3.5). A final operand evaluation, performed as a tail context, is also expected to return a boolean, that expectation being the meaning of the predicate suffix on the name "`$and?`" of the operative. However, we do not require implementations to signal a dynamic type error on the result of a tail context. If error signaling were sufficiently important to the design to require the error to be signaled (which it isn't, here), we would lift the tail-context requirement; although a dynamic *boolean* type check certainly *can* be imposed on a tail context without compromising its tail-context status, doing so is a burden that we would not lightly impose on non-robust implementations of Kernel.

Derivation

The following expression defines the ***$and?*** operative, using only previously defined features.

```
($define! $and?
   ($vau x e
      ($cond ((null? x)         #t)
             ((null? (cdr x))   (eval (car x) e)) ; tail context
             ((eval (car x) e)  (apply (wrap $and?) (cdr x) e))
```

```
                (#t                 #f))))
```

6.1.5 `$or?`

`($or? . ⟨objects⟩)`

The ***$or?*** operative performs a "short-circuit or" of its operands: It evaluates them from left to right, until either an operand evaluates to true, or the end of the operand list is reached. If the end of the operand list is reached (which is immediate if ⟨objects⟩ is nil), the operative returns false. If an operand evaluates to true, no further operand evaluations are performed, and the operative return true. If ⟨objects⟩ is acyclic, and the last operand is evaluated, it is evaluated as a tail context (§3.10). If ⟨objects⟩ is cyclic, an unbounded number of operand evaluations may be performed.

If any of the operands is evaluated to a non-*boolean* value, it is an error; and if the operand is not the last operand, the error is signaled.

Cf. the ***or?*** applicative, §6.1.3.

Rationale:
See the rationale discussion for ***$and?***, §6.1.4.

Derivation

The following expression defines the ***$or?*** operative, using only previously defined features.

```
($define! $or?
   ($vau x e
      ($cond ((null? x)         #f)
             ((null? (cdr x))   (eval (car x) e)) ; tail context
             ((eval (car x) e)  #t)
             (#t                (apply (wrap $or?) (cdr x) e)))))
```

6.2 Combiners

6.2.1 `combiner?`

`(combiner? . objects)`

This is the type predicate for type *combiner*. Returns true iff all of its arguments are combiners, i.e., of type *operative* or *applicative*.

Rationale:
See the rationale for applicative ***and?*** (§6.1.2).

Derivation

The following expression defines the ***combiner?*** applicative using only previously defined features.

```
($define! combiner?
   ($lambda x
      (apply and? (map ($lambda (x)
                          (or? (applicative? x)
                               (operative? x)))
                       x))))
```

6.3 Pairs and lists

6.3.1 `length`

`(length` *object*`)`

Applicative ***length*** returns the (exact) improper-list length of *object*. (Re improper lists, see §5.7.1.) That is, it returns the number of consecutive cdr references that can be followed starting from *object*. If *object* is not a pair, it returns zero; if *object* is a cyclic list, it returns positive infinity.

Derivation

The following expression defines the ***length*** applicative, using previously defined features, and number primitives (§12).

```
($define! length
   ($lambda (object)
      ($let (((#ignore #ignore a c)  (get-list-metrics object)))
         ($if (>? c 0)
              #e+infinity
              a))))
```

6.3.2 `list-ref`

`(list-ref` *list* *integer*`)`

integer should be exact and non-negative. The ***list-ref*** applicative returns the *integer*$^{\text{th}}$ element of *list*, zero-indexed.

Derivation

The following expression defines the *list-ref* applicative, using previously defined features.

```
($define! list-ref
   ($lambda (ls k)
      (car (list-tail ls k))))
```

6.3.3 append

(append . *lists*)

Here, all the elements of *lists* except the last element (if any) must be acyclic lists. The *append* applicative returns a freshly allocated list of the elements of all the specified lists, in order, except that if there is a last specified element of *lists*, it is not copied, but is simply referenced by the cdr of the preceding pair (if any) in the resultant list.

If *lists* is cyclic, the cycle of the result list consists of just the elements of the lists specified in the cycle in *lists*. In this case, the acyclic prefix length of the result is the sum of the lengths of the lists specified in the acyclic prefix of *lists*, and the cycle length of the result is the sum of the lengths of the lists specified in the cycle of *lists*.

The following equivalences hold.

$$\begin{array}{rcl} (\mathit{append}) & \equiv & () \\ (\mathit{append}\ h) & \equiv & h \\ (\mathit{append}\ ()\ h\ .\ t) & \equiv & (\mathit{append}\ h\ .\ t) \\ (\mathit{append}\ (\mathit{cons}\ a\ b)\ h\ .\ t) & \equiv & (\mathit{cons}\ a\ (\mathit{append}\ b\ h\ .\ t)) \end{array}$$

Rationale:

The behavior of *append* on acyclic *lists* generalizes at least three mathematically simple behaviors. If *append* required its last argument to be an acyclic list, and there were no mutation in the language (so that *equal?* pairs were effectively identical), the acyclic behavior would be fully defined by the zero-argument base case and a simplified recursive step. If *append* required its last argument to be an acyclic list, and copied the pairs of that list as well as those of preceding argument lists, its acyclic behavior would again be defined by the zero-argument base case and simplified recursive step. If *append* required at least one argument, then the special treatment of the last argument would seem less noteworthy since it would simply be the (sole) base case. The acyclic behavior provided here supports all of these as special cases, but also supports the sometimes-useful construction of improper lists (see also *list**, §5.2.2). For uniform acyclic behavior with respect to mutation, one can explicitly add a nil final argument; an expression

(append $l_1 \cdots l_n$ ())

is guaranteed to signal an error if any l_k is not an acyclic list, and returns an acyclic list with none of the same pairs as the original lists l_k.

Derivation

The following expression defines the ***append*** applicative, using only previously defined features.

```
($define! append
   ($lambda lss

      ($define! set-last!
         ($lambda (ls tail) ; set cdr of last pair of ls to tail
            ($let ((next  (cdr ls)))
               ($if (pair? next)
                    (set-last! next tail)
                    (set-cdr! ls tail)))))

      ($define! aux2
         ($lambda (ls tail) ; prepend ls onto tail
            ($if (null? ls)
                 tail
                 (cons (car ls) (aux2 (cdr ls) tail)))))

      ($define! aux1
         ($lambda (k lss tail) ; prepend k elmts of lss onto tail
            ($if (>? k 0)
                 (aux2 (car lss)
                       (aux1 (- k 1) (cdr lss) tail))
                 tail)))

      ($if (null? lss)
           ()
           ($let (((#ignore #ignore a c)
                     (get-list-metrics lss)))
              ($if (>? c 0)
                   ($let ((cycle  (aux1 c (list-tail lss a) ())))
                      ($cond ((pair? cycle)
                                (set-last! cycle cycle)))
                      (aux1 a lss cycle))
                   (aux1 (- a 1) lss (list-ref lss (- a 1))))))))
```

Rationale:

Helper applicative ***aux2*** really ought to signal an error when its first argument is a cyclic list, because that could happen by accident; so, even though we don't usually do non-required error signaling in our expository library derivations, we would be tempted to do so here, if not that it would depend on details of error handling that haven't been finalized yet in this revision of the report.

6.3.4 `list-neighbors`

(`list-neighbors` *list*)

The ***list-neighbors*** applicative constructs and returns a list of all the consecutive sublists of *list* of length 2, in order. If *list* is nil, the result is nil. If *list* is non-nil, the length of the result is one less than the length of *list*. If *list* is cyclic, the result is structurally isomorphic to it (i.e., has the same acyclic prefix length and cycle length).

For example,

```
(list-neighbors (list 1 2 3 4))   ⟹   ((1 2) (2 3) (3 4))
```

Rationale:

This applicative is one of Kernel's intermediate-level tools for handling potentially cyclic lists (as opposed to the lower-level tools that use explicit list metrics). It addresses list handling that involves considering consecutive elements two at a time (whereas most other intermediate-level tools act element-wise, e.g. ***map***, §5.9.1). For examples of its use, see the derivations of library applicatives ***append!***, §6.4.1, and ***eq?***, §6.5.1.

Some consideration was given to providing a more general tool ***list-tails*** that would return a list of nonempty suffixes of its argument. However, thus far it appears that most cases addressable by ***list-tails*** are more cleanly addressed by ***list-neighbors***; so ***list-neighbors*** is included here, while ***list-tails*** is omitted pending a stronger case for its utility.

Derivation

The following expression defines the ***list-neighbors*** applicative, using only previously defined features.

```
($define! list-neighbors
   ($lambda (ls)

      ($define! aux
         ($lambda (ls n) ; get n sets of neighbors from ls
            ($if (>? n 0)
                 (cons (list (car ls) (cadr ls))
                       (aux (cdr ls) (- n 1)))
                 ())))

      ($let (((p #ignore a c)  (get-list-metrics ls)))
         ($if (=? c 0)
```

```
                    (aux ls (- a 1))
                    ($let ((ls  (aux ls p)))
                       (encycle! ls a c)
                       ls)))))
```

6.3.5 filter

(filter *applicative list*)

Applicative ***filter*** passes each of the elements of *list* as an argument to *applicative*, one at a time in no particular order, using a fresh empty environment for each call. The result of each call to *applicative* must be boolean, otherwise an error is signaled. ***filter*** constructs and returns a list of all elements of *list* on which *applicative* returned true, in the same order as in *list*.

applicative is called exactly as many times as there are pairs in *list*. The resultant list has a cycle containing exactly those elements accepted by *applicative* that were in the cycle of *list*; if there were no such elements, the result is acyclic.

Rationale:

Because ***filter*** doesn't process the list elements in any particular order, it must terminate in finite time even if *list* is cyclic (per the rationale discussion in §3.9; the possibility that *applicative* could have side-effects would only matter to the policy if the elements were to be processed in order).

The two paradigmatic examples of a standard applicative that takes an applicative argument are ***apply*** and ***map*** (§§5.5.1, 5.9.1). ***apply*** allows the caller to specify an environment to be used when calling its applicative argument; and this makes sense for ***apply***, because ***apply***'s purpose is to facilitate general calls to its argument; but the purpose of ***filter*** is list processing, so such a general interface would be tangential. ***map*** calls its applicative argument using the dynamic environment of the call to ***map***, because that behavior is followed by the code equivalence that ***map*** seeks to preserve; but ***filter*** has no comparable equivalence to preserve. So instead, ***filter*** follows the simple and hygienic precedent of the default behavior of ***apply***, by using a fresh empty environment for each argument call.

For examples of the use of applicative ***filter*** in robust list-processing without explicit use of list metrics, see the derivations of library applicatives ***assoc***, §6.3.6, and ***append!***, §6.4.1.

Derivation

The following expression defines the ***filter*** applicative, using only previously defined features.

```
($define! filter
   ($lambda (accept? ls)
      (apply append
```

```
(map ($lambda (x)
        ($if (apply accept? (list x))
             (list x)
             ()))
     ls))))
```

Note the use of *apply* when calling argument *accept?*, which provides the call with its expected fresh empty environment.

6.3.6 assoc

(assoc *object pairs*)

Applicative *assoc* returns the first element of *pairs* whose car is *equal?* to *object*. If there is no such element in *pairs*, nil is returned.

Cf. *assq* (§6.4.3).

Derivation

The following expression defines the *assoc* applicative, using only previously defined features.

```
($define! assoc
   ($lambda (object alist)
      ($let ((alist  (filter ($lambda (record)
                                (equal? object (car record)))
                              alist)))
         ($if (null? alist)
              ()
              (car alist)))))
```

Rationale:

This isn't a particularly efficient way to implement *assoc*, but it is interesting in that it emphasizes that *assoc* doesn't depend on processing the list in any particular order. Because of this order-independence, it must handle cyclic lists in finite time (per the rationale discussion in §3.9).

The choice of what value *assoc* should return on failure touches on a number of deep issues in the language design. We first consider why several alternatives would be unsuitable.

In most Lisp languages, the value returned by *assoc* to indicate failure is determined by the language policy for handling conditional test values. In those languages, arbitrary objects can be conditional test values, and all but one or a few designated values count as true. By selecting a designated-false value as the value returned by *assoc* on failure, one can write

```
(let ((result  (assoc key alist)))
  (if result
      ⟨consequent⟩
      ⟨alternative⟩))
```

and have ⟨consequent⟩ executed if *key* is found in *alist*, ⟨alternative⟩ if it isn't found. Accordingly, *assoc* in Scheme returns `#f` on failure ([KeClRe98]), while in more traditional Lisps (as [Pi83]) it returns nil. However, Kernel forbids non-boolean values for conditional tests (§3.5 rationale). The above expression in Kernel (replacing `let/if` with `$let/$if`) would cause a type error if *key* is found in *alist*; and so, in practice, it is desirable that the failure value of Kernel *assoc* should be non-boolean, so that the dynamic type error will occur consistently rather than intermittently.

The failure value of Kernel *assoc* signifies, broadly speaking, *nothing*, that being what was found. It would be a conceptual type error to use `#inert` (§4.5) for this purpose, since `#inert` signifies *no information*, and thus ought not to be used to convey positive information (namely, the definite fact that nothing was found).

One might introduce a new encapsulated object for the purpose, similarly to `#ignore` and `#inert` (§§4.8, 4.5) — perhaps `#search-failure`, or more generally, `#failure`. This, however, would set an unfortunate precedent. The abstract domain of return codes can never be fully represented by one, or even a finite fixed set, of specialized objects (re full representation of abstract domains, see Appendix B); and an *infinite* primitive type of return codes would be a major, probably elaborate, addition to the language. It could therefore only be admitted after a very compelling case had been made for it; and it seems unlikely that such a case could be made, since there are already *two* significant facilities in the language that address much the same purpose: the exception handling supported by Kernel's *continuation* type (§7), and the data structure parsing supported by Kernel's generalized matching algorithm (§4.9.1).

Generalized matching bears on the general problem of return codes by effectively supporting simultaneous return of multiple values. If a combiner always returns a data structure of a certain shape, the caller can immediately decompose the returned structure into its constituent parts. In the case of *assoc*, one wants to return either one or two values: always, a boolean indicating whether *object* was found in *alist*; and on success, also the *alist* element that was found. It's undesirable to return a second value on failure, since that would make it easier to accidently use the second value as if it had been found. Let's call the combiner that behaves this way `assoc*`. One could define it by:

```
($define! assoc*
   ($lambda (object alist)
      ($let ((alist  (filter ($lambda (record)
                                (equal? object (car record)))
                              alist)))
         ($if (null? alist)
              (cons #f ())
              (cons #t (car alist))))))
```

To use *assoc**, one could then write

```
($let (((found . result)  (assoc* key alist)))
   ($if found
        ⟨consequent⟩
        ⟨alternative⟩))
```

Here, formal parameter tree `(found . result)` is locally matched to the data structure returned by *assoc**, binding `found` to `#f` or `#t`, and `result` to `()` or the matching element of *alist*. An optimizing Kernel compiler may recognize that the returned data structure cannot actually be accessed, and therefore needn't actually be constructed as long as its two parts are properly exported from *assoc**.

The entire technique works because of the special status that the matching algorithm awards to the list-constituent types, *pair* and *null*. The special status of type *null*, in particular, is the reason why it is natural to substitute nil for *result* on failure, in the absence of a useful value.

However, the data structure returned by *assoc** is gratuitously complicated, forcing the caller to use either a compound definiend to bind two formal parameters as in the above *$let*, or explicit accessors to extract the desired parts of the structure. Either device would be perfectly acceptable with cause; but here, neither is warranted, because the boolean car of the structure is entirely redundant information: we can easily factor *assoc** into two stages, one that simply returns the element found or nil, and another that wraps the data structure and boolean around that result:

```
($define! assoc*-aux
   ($lambda (object alist)
      ($let ((alist  (filter ($lambda (record)
                                (equal? object (car record)))
                             alist)))
         ($if (null? alist)
              ()
              (car alist)))))

($define! assoc*
   ($lambda (object alist)
      ($let ((result  (assoc*-aux object alist)))
         (cons (pair? result) result))))
```

The underlying function *assoc*-aux* is obviously just *assoc* as defined in this report; and we could —and do— eliminate both the indirection, and the redundancy that caused it, by providing the simpler *assoc* rather than the needlessly elaborate *assoc**. The use of nil to represent nothing in a return value is thus seen to be neither arbitrary nor (in itself) *ad hoc*, but rather a natural consequence of the special status afforded to types *pair* and *null* by Kernel's matching algorithm (§4.9.1).

6.3.7 `member?`

`(member?` *object list*`)`

Applicative ***member?*** is a predicate that returns true iff some element of *list* is ***equal?*** to *object*.

Cf. ***memq?*** (§6.4.4).

Rationale:

On the handling of cyclic lists, see the rationale discussion in §3.9.

Derivation

The following expression defines the ***member?*** applicative, using only previously defined features.

```
($define! member?
   ($lambda (object ls)
      (apply or?
             (map ($lambda (x) (equal? object x))
                  ls))))
```

6.3.8 `finite-list?`

`(finite-list? .` *objects*`)`

This is the type predicate for type *finite-list*. It returns true iff all its arguments are acyclic lists (§3.9).

Rationale:

See the rationale for applicative ***and?*** (§6.1.2).

Since Scheme programmers expect predicate ***list?*** to test for acyclic lists, but Kernel lists can be cyclic, it would be potentially confusing for Scheme programmers if Kernel were to assign the unadorned predicate name ***list?*** to either type. So Kernel qualifies its names for both: `finite-list?` for the acyclic type (here), and `countable-list?` for the potentially cyclic type (§6.3.9).

Derivation

The following expression defines ***finite-list?*** using only previously defined features.

```
($define! finite-list?
   ($lambda args
      (apply and?
```

```
            (map ($lambda (x)
                    ($let (((#ignore n . #ignore)
                            (get-list-metrics x)))
                      (>? n 0)))
                 args))))
```

6.3.9 `countable-list?`

(`countable-list? .` *objects*)

This is the type predicate for type *list*. Returns true iff all its arguments are lists (§3.9).

Rationale:
See the rationales for applicatives ***and?*** (§6.1.2) and ***finite-list?*** (§6.3.8).

Derivation

The following expression defines ***countable-list?*** using only previously defined features.

```
($define! countable-list?
   ($lambda args
      (apply and?
            (map ($lambda (x)
                    ($let (((#ignore n #ignore c)
                            (get-list-metrics x)))
                      ($or? (>? c 0)
                            (>? n 0))))
                 args))))
```

6.3.10 `reduce`

(`reduce` *list binary identity*)
(`reduce` *list binary identity precycle incycle postcycle*)

list should be a list. *binary* should be an applicative. If the first call syntax is used, *list* should be an acyclic list. If the second call syntax is used, *precycle*, *incycle*, and *postcycle* should be applicatives.

If *list* is empty, applicative ***reduce*** returns *identity*.

If *list* is nonempty but acyclic, applicative ***reduce*** uses binary operation *binary* to merge all the elements of *list* into a single object, using any associative grouping of the elements. That is, the sequence of objects initially found in *list* is repeatedly decremented in length by applying *binary* to a list of any two consecutive objects, replacing those two objects with the result at the point in the sequence where they occurred; and when the sequence contains only one object, that object is returned.

If *list* is cyclic, the second call syntax must be used. The elements of the cycle are passed, one at a time (but just once for each position in the cycle), as arguments to unary applicative *precycle*; the finite, cyclic sequence of results from *precycle* is reduced using binary applicative *incycle*; and the result from reducing the cycle is passed as an argument to unary applicative *postcycle*. Binary operation *binary* is used to reduce the sequence consisting of the elements of the acyclic prefix of *list* followed by the result returned by *postcycle*. The only constraint on the order of calls to the applicatives is that each call must be made before its result is needed (thus, parts of the reduction of the acyclic prefix may occur before the contribution from the cycle has been completed).

Each call to *binary*, *precycle*, *incycle*, or *postcycle* uses the dynamic environment of the call to ***reduce***.

If *list* is acyclic with length $n \geq 1$, *binary* is called $n - 1$ times. If *list* is cyclic with acyclic prefix length a and cycle length c, *binary* is called a times; *precycle*, c times; *incycle*, $c - 1$ times; and *postcycle*, once.

Rationale:

Because `reduce` uses an unspecified associative order of binary operations, the guidelines in §3.9 require it to reduce a cyclic list in finite time *if* the reduction behavior can be naturally defined in the cyclic case. However, an arbitrary binary operation can't be automatically generalized to reduce an infinite sequence of elements in finite time. So, either the client must provide additional information on how to handle cycles, or cyclic-list reduction must be an error.

Cyclic-list reduction is sometimes desirable (addition is a paradigmatic commutative case); and would be significantly more onerous if, in addition to the technicalities of each particular case of reduction, one also had to cope with explicit list metrics; therefore, the cyclic case is worth supporting at an intermediate level (i.e., above the list-metric level). The technicalities of any such reduction are sufficiently ticklish, even in simple cases, that providing a single interface general enough to cover elaborate cases does not significantly increase the burden on the less elaborate. On the other hand, the added programming burden and source-code complexity of the cycle-handling arguments may sometimes be undesirable —or irrelevant, or impossible— so the simpler cycle-rejecting interface is also worth supporting.

For the single general interface to specify cycle-handling, it is envisioned that, as a most-elaborate case, *precycle* will convert each term to a record that tracks multiple facets of the reduction; *incycle* will use and synthesize these facets; and *postcycle* will convert the reduced record back to a term of the type expected by *binary*.

The applicative arguments are all called with the dynamic environment of the call to ***reduce*** by analogy with ***map*** (§5.9.1).

Two variant tools are under consideration (in a leisurely fashion, not to rush into vocabulary growth), ***reduce-left*** and ***reduce-right***, which would use specific associative groupings and thus cater primarily to non-associative binary operations. Given a cyclic list, *reduce-left* would perform the binary operation an unbounded number of times, while *reduce-right* would signal an error.

Derivation

The following expression defines *reduce* using only previously defined features.

```
($define! reduce
   ($let ()

      ($define! reduce-acyclic
         ($lambda (ls bin id)
            ($cond ((null? ls)        id)
                   ((null? (cdr ls))  (car ls))
                   (#t
                      (bin (car ls)
                           (reduce-acyclic (cdr ls) bin id))))))

      ($define! reduce-n
         ($lambda (ls bin n)
            ($if (=? n 1)
                 (car ls)
                 (bin (car ls)
                      (reduce-n (cdr ls) bin (- n 1))))))

      (wrap ($vau (ls bin id . opt) env

         ($define! fixenv
            ($lambda (appv)
               ($lambda x (apply appv x env))))

         ($define! bin (fixenv bin))

         ($let (((p n a c)  (get-list-metrics ls)))
            ($if (=? c 0)
                 (reduce-acyclic ls bin id)
                 ($sequence
                    ($define! (pre in post) (map fixenv opt))
                    ($define! reduced-cycle
                       (post (reduce-n (map pre (list-tail ls a))
                                       in
                                       c)))
                    ($if (=? a 0)
```

```
                    reduced-cycle
                    (bin (reduce-n ls bin a)
                         reduced-cycle))))))))
```

6.4 Pair mutation (optional)

6.4.1 append!

(append! . *lists*)

lists must be a nonempty list; its first element must be an acyclic nonempty list, and all of its elements except the last element (if any) must be acyclic lists. The ***append!*** applicative sets the cdr of the last pair in each nonempty list argument to refer to the next non-nil argument, except that if there is a last non-nil argument, it isn't mutated.

It is an error for any two of the list arguments to have the same last pair.

The result returned by this applicative is inert.

The following equivalences hold.

(append! *v*) ≡ #inert

(append! *u* *v* . *w*) ≡ ($sequence (append! *u* *v*)
(append! *u* . *w*))

Rationale:

append! is pointedly *not* a more space-efficient version of *append*. Efficiency is a proscribed design motivation in Kernel (per *G5* of §0.1.2), so the design of *append!* must be driven entirely by its role as a mutator. Mutators are called for effect, rather than for value, hence they always return #inert; and this in turn steers the programmer away from confusion between *append!* and *append* (per *G3* of §0.1.2).

Whereas the acyclic behavior of *append* naturally builds leftward from its rightmost argument by means of *cons*, the acyclic behavior of *append!* naturally builds *right*ward from its *left*most argument by means of *set-cdr!*. Hence, *append!* must have at least one argument, and the leftmost argument must be nonempty so that it *can* have things appended to it by mutation. Note that, after completion of the operation, the mutated leftmost argument is the one structure that is certain to encompass all of the elements of all of the *append!*ed lists.

Derivation

The following expression defines the *append!* applicative, using only previously defined features.

```
($define! append!
   ($lambda lss
```

```
($define! set-last!
   ($lambda (ls tail)
      ($let ((next  (cdr ls)))
         ($if (pair? next)
              (set-last! next tail)
              (set-cdr! ls tail)))))

(map ($lambda (x) (apply set-last! x))
     (list-neighbors (filter ($lambda (x)
                                (not? (null? x)))
                             lss)))
#inert))
```

If applicative ***for-each*** (§6.9.1) were available at this point, the derivation would have used that, rather than construct a list of inert values with ***map*** only to discard it.

6.4.2 copy-es

(copy-es *object*)

Briefly, applicative ***copy-es*** returns an object initially ***equal?*** to *object* with a freshly constructed evaluation structure (§4.7.2) made up of mutable pairs.

If *object* is not a pair, the applicative returns *object*. If *object* is a pair, the applicative returns a freshly constructed pair whose car and cdr would be suitable results for (***copy-es*** (***car*** *object*)) and (***copy-es*** (***cdr*** *object*)), respectively. Further, the evaluation structure of the returned value is structurally isomorphic (§3.9) to that of *object* at the time of copying, with corresponding non-pair referents being ***eq?***.

Cf. ***copy-es-immutable*** (§4.7.2).

Derivation

The following expression defines ***copy-es*** using only previously defined features.

```
($define! copy-es
   ($lambda (x)

      ($define! aux
         ($lambda (x alist)
            ($if (not? (pair? x))
                 (list x alist)
                 ($let ((record (assoc x alist)))
                  ($if (pair? record)
                       (list (cdr record) alist)
```

```
                                    ($let ((y (cons () ())))
                                     ($let ((alist (cons (cons x y) alist)))
                                      ($let (((z alist) (aux (car x) alist)))
                                       (set-car! y z)
                                       ($let (((z alist) (aux (cdr x) alist)))
                                        (set-cdr! y z)
                                        (list y alist))))))))))

          (car (aux x ())))))
```

The depth of nesting of this code could be greatly reduced if operative *$let** (§6.7.4) were available; one could rewrite the alternative clause of the inner *$if* combination as:

```
($let* ((y          (cons () ()))
        (alist      (cons (cons x y) alist))
        ((z alist)  (aux (car x) alist))
        (#ignore    (set-car! y z))
        ((z alist)  (aux (cdr x) alist))
        (#ignore    (set-cdr! y z)))
   (list y alist))
```

Rationale:

The above derivation of *copy-es* is messy and slow (quadratic time) because it performs its traversal *without* temporarily marking visited pairs. Internal robust traversals can readily achieve linear time using temporary marks.

6.4.3 assq

(assq *object pairs*)

Applicative *assq* returns the first element of *pairs* whose car is *eq?* to *object*. If there is no such element in *pairs*, nil is returned.

Rationale:

Applicative *assq* behaves as *assoc*, except that when comparing the cars of elements to *object*, it uses *eq?* instead of *equal?*. See the rationale discussion for *assoc*, §6.3.6.

Derivation

The following expression defines the *assoc* applicative, using only previously defined features.

```
($define! assq
   ($lambda (object alist)
      ($let ((alist (filter ($lambda (record)
                                (eq? object (car record)))
                             alist)))
         ($if (null? alist)
              ()
              (car alist)))))
```

6.4.4 memq?

(memq? *object list*)

Applicative ***memq?*** is a predicate that returns true iff some element of *list* is ***eq?*** to *object*.

Cf. ***member?*** (§6.3.7).

Derivation

The following expression defines the ***memq?*** applicative, using only previously defined features.

```
($define! memq?
   ($lambda (object ls)
      (apply or?
             (map ($lambda (x) (eq? object x))
                  ls))))
```

6.5 Equivalence under mutation (optional)

6.5.1 eq?

(eq? . *objects*)

This applicative generalizes primitive predicate ***eq?*** (§4.2.1) to zero or more arguments. It is a predicate that returns true unless some two of its arguments are different as judged by the primitive predicate.

Rationale:

Because the applicative doesn't process its arguments in any particular order, it must terminate in finite time even if *objects* is cyclic (per the rationale discussion in §3.9).

Returning true on zero or one arguments is deemed the most uniform, hence least error-prone, behavior for those cases. One measure of its uniformity is that it allows the entire behavior of the applicative, including the zero/one-argument cases, to be captured by a very simple statement (above; cf. the behavioral statement for ***and?***, §6.1.2). Another is that it preserves the following implication:

($eq?$ h . t) $\Rightarrow$ ($eq?$. t)

Derivation

The following expression defines the ***eq?*** library applicative using only previously defined features. (This is a bit tricky because some previously defined features — notably ***get-list-metrics***— use binary ***eq?***, creating hidden circularities when `eq?` is rebound to the library applicative.)

```
($define! eq?
   ($let ((old-eq?  eq?))
      ($lambda x
         ($if ($and? (pair? x) (pair? (cdr x)) (null? (cddr x)))
              (apply old-eq? x)
              (apply and?
                     (map ($lambda (x) (apply old-eq? x))
                          (list-neighbors x)))))))
```

6.6 Equivalence up to mutation

6.6.1 `equal?`

`(equal? .` *objects*`)`

This applicative generalizes primitive predicate ***equal?*** (§4.3.1) to zero or more arguments. It is a predicate that returns true unless some two of its arguments are different as judged by the primitive predicate.

Rationale:
See the rationale for library ***eq?***, §6.5.1.

Derivation

The following expression defines the ***equal?*** library applicative using only previously defined features. (As a somewhat paranoid precaution, special provisions are taken, as with library ***eq?***, §6.5.1, to truncate any hidden circularities induced by previously defined features using binary ***equal?***.)

```
($define! equal?
   ($let ((old-equal?  equal?))
      ($lambda x
         ($if ($and? (pair? x) (pair? (cdr x)) (null? (cddr x)))
```

```
              (apply old-equal? x)
              (apply and?
                     (map ($lambda (x) (apply old-equal? x))
                          (list-neighbors x)))))))
```

6.7 Environments

6.7.1 `$binds?`

`($binds?` ⟨exp⟩ `.` ⟨symbols⟩`)`

Operative ***$binds?*** evaluates ⟨exp⟩ in the dynamic environment; call the result *env*. *env* must be an environment. The operative is a predicate that returns true iff all its later operands, ⟨symbols⟩, are visibly bound in *env*.

Rationale:

The choice to make this feature operative, rather than applicative, is ultimately derived from accident-avoidance Guideline *G3* of §0.1.2. In Kernel programming, unevaluated symbols are opportunities for accidental hygiene violations. The likelihood of such accidents is low in sufficiently focused situations (most operative core library feature derivations are examples of this, notably including the library derivation here of ***$binds?***), but the likelihood rises sharply with increasing accessibility of the symbols — since precautions for handling unevaluated symbols become a ready source of problems as they become incidental to the purpose of the code. One way to discourage accidents of this sort in Kernel is to omit any facilities that purposefully return unevaluated symbols, such as quasiquotation (cf. §1.1), thus avoiding any convenient means for general use of unevaluated symbols. The other side of this is to avoid features that *motivate* general use of unevaluated symbols. Normal use of an applicative ***binds?*** would presuppose operands that evaluate to unevaluated symbols — which may occur in a focused setting in the body of an operative when the symbols are operands, but which in a general setting is suggestive of some quasiquotation-like device. Hence, to avoid this accident-prone motivation, the provided feature is operative.

Derivation

The following expression defines the ***$binds?*** operative, using previously defined features, and features from the *Continuations* module (§7). (The latter features aren't all primitive, but they don't use ***binds?*** directly or indirectly, so there's no circularity in the derivation.)

```
($define! $binds?
   ($vau (exp . ss) dynamic
      (guard-dynamic-extent
         ()
```

```
     ($lambda ()
              ($let ((env  (eval exp dynamic)))
                 (map ($lambda (sym) (eval sym env))
                      ss))
              #t)
     (list (list error-continuation
                 ($lambda (#ignore divert)
                    (apply divert #f)))))))
```

Rationale:

Presenting this predicate as a library feature drives home the point that it doesn't introduce any capability that wasn't already provided by the language. In particular, for purposes of type encapsulation (§3.4), there is still no way for a Kernel program to generate a complete list of the variables exhibited in an arbitrary environment. Predicate ***$binds?***, or the techniques used to derive it, could be used to formally *enumerate* such a list, by sequentially generating all possible variables (an infinite sequence) and testing each one; but there would be no way to know when all variables in the environment had been enumerated. This nontermination is critical because it means that no Kernel program can prove in finite time that two non-***eq?*** environments are equivalent up to mutation; therefore, the rules governing predicate ***equal?*** (§4.3.1) do not require it to return false in any cases where predicate ***eq?*** returns true.

6.7.2 get-current-environment

(get-current-environment)

The *get-current-environment* applicative returns the dynamic environment in which it is called.

Rationale:

Operatives should be used only when there is a specific reason to do so, so that the programmer can assume that $'s always flag out exceptions to the usual rules of argument evaluation. Accordingly, throughout this report zero-ary combiners, such as this one, are always *wrap*ped: *get-current-environment* rather than *$get-current-environment*.

Some combiner names are nouns, while others are verbs. When a combiner acts on one or more operands, it's clear that it describes action, so we consider it acceptably clear to name the combiner for its result (e.g., *lcm*, §12.5.14, which returns the lcm of its arguments). Often such a combiner *can* be called with no operands, but usually isn't, so the degenerate case shouldn't interfere with understanding the nomenclature. However, when the combiner is primarily, or even exclusively, called without operands, there is some danger that because its name is a noun, the programmer might forget to put parentheses around it; so, in the interest of preventing accidents (*G3* of §0.1.2), the names of zero-ary combiners are always verbs. Hence, in this case, get-current-environment rather than simply current-environment.

Derivation

The following expression defines *get-current-environment* using only previously defined features.

```
($define! get-current-environment (wrap ($vau () e e)))
```

6.7.3 make-kernel-standard-environment

`(make-kernel-standard-environment)`

The *make-kernel-standard-environment* applicative returns a standard environment; that is, a child of the ground environment with no local bindings. (See §3.2.)

Derivation

The following expression defines *make-kernel-standard-environment* using only previously defined features.

```
($define! make-kernel-standard-environment
   ($lambda () (get-current-environment)))
```

Rationale:

This derivation works exactly because, per §1.3.2, library derivations are postulated to be evaluated in the ground environment. Thus, the ground environment becomes the static environment of the compound combiner; and when the combiner is called, its local environment —which it returns— is a child of the ground environment, with no local bindings since the combiner has no formal parameters.

6.7.4 $let*

`($let*` ⟨bindings⟩ `.` ⟨body⟩`)`

⟨bindings⟩ should be a finite list of formal-parameter-tree/expression pairings, each of the form `(`⟨formals⟩ ⟨expression⟩`)`, where each ⟨formals⟩ is as described for the *$define!* operative, §4.9.1. ⟨body⟩ should be a list of expressions.

The expression

(*$let** () . ⟨body⟩)

is equivalent to

(*$let* () . ⟨body⟩)

and the expression

(*$let** ((⟨form⟩ ⟨exp⟩) . ⟨bindings⟩) . ⟨body⟩)

is equivalent to

($let ((⟨form⟩ ⟨exp⟩)) ($let* ⟨bindings⟩ . ⟨body⟩))

Rationale:

The *$let** operative provides a different combination of capabilities and constraints than do the other operatives in the `$let` family (on which see the rationale for *$let*, §5.10.1). The binding expressions are guaranteed to be evaluated from left to right, and each of these evaluations has access to the bindings of previous evaluations. Because the result of each binding expression is matched separately, there is nothing to prevent the same symbol from occurring in more than one ⟨formals⟩. (For an example of this technique, see the alternative derivation in §6.4.2.) However, each of these evaluations takes place in a child of the environment of the previous one, and bindings for the previous evaluation take place in the child, too. So, if one of the binding expressions is a *$vau* or *$lambda* expression, the resulting combiner still can't be recursive; and only the first binding expression is evaluated in the dynamic environment, so if the dynamic environment is to be bound, only the first binding can do it.

Derivation

The following expression defines *$let** using only previously defined features.

```
($define! $let*
   ($vau (bindings . body) env
      (eval ($if (null? bindings)
                 (list* $let bindings body)
                 (list $let
                       (list (car bindings))
                       (list* $let* (cdr bindings) body)))
            env)))
```

6.7.5 `$letrec`

($letrec ⟨bindings⟩ . ⟨body⟩)

⟨bindings⟩ and ⟨body⟩ should be as described for *$let*, §5.10.1.
The expression

($letrec ((⟨form$_1$⟩ ⟨exp$_1$⟩) ... (⟨form$_n$⟩ ⟨exp$_n$⟩)) . ⟨body⟩)

is equivalent to

($let ()
   ($define! (⟨form$_1$⟩ ... ⟨form$_n$⟩)
      (list ⟨exp$_1$⟩ ... ⟨exp$_n$⟩))
   . ⟨body⟩)

Rationale:

The *$letrec* operative provides a different combination of capabilities and constraints than the other operatives in the `$let` family (on which see the rationale for *$let*, §5.10.1). The binding expressions may be evaluated in any order. None of them are evaluated in the dynamic environment, so there is no way to capture the dynamic environment using *$letrec*; and none of the bindings are made until after the expressions have been evaluated, so the expressions cannot see each others' results; but since the bindings are in the same environment as the evaluations, they can be recursive, and even mutually recursive, combiners.

The *R5RS* requires its special form `letrec` to provide dummy bindings for the $\langle \mathrm{sym}_k \rangle$ (bindings to "undefined values") while the $\langle \mathrm{exp}_k \rangle$ are being evaluated, but then goes on to say that *it is an error* for the evaluation of any $\langle \mathrm{exp}_k \rangle$ to actually look up any of the $\langle \mathrm{sym}_k \rangle$ in e'. So the bindings have to be there, and you're supposed to pretend they aren't.

Derivation

The following expression defines *$letrec* using only previously defined features.

```
($define! $letrec
   ($vau (bindings . body) env
      (eval (list* $let ()
                   (list $define!
                         (map car bindings)
                         (list* list (map cadr bindings)))
                   body)
            env)))
```

6.7.6 `$letrec*`

`($letrec*` ⟨bindings⟩ `.` ⟨body⟩`)`

⟨bindings⟩ and ⟨body⟩ should be as described for *$let**, §6.7.4.

The expression

(*$letrec** `()` `.` ⟨body⟩`)`

is equivalent to

(*$letrec* `()` `.` ⟨body⟩`)`

and the expression

(*$letrec** `((`⟨form⟩ ⟨exp⟩`))` `.` ⟨bindings⟩`)` `.` ⟨body⟩`)`

is equivalent to

(*$letrec* `((`⟨form⟩ ⟨exp⟩`))` (*$letrec** ⟨bindings⟩ `.` ⟨body⟩`))`

Rationale:
The *$letrec** operative provides a different combination of capabilities and constraints than do the other operatives in the `$let` family (on which see the rationale for *$let*, §5.10.1). The binding expressions are guaranteed to be evaluated from left to right; each of these evaluations has access to the bindings of previous evaluations; and the result of each evaluation is matched in the same environment where it was performed, so if the result is a combiner, it can be recursive. Further, the result of each binding expression is matched separately, so there is nothing to prevent the same symbol from occurring in more than one ⟨formals⟩. However, since each evaluation takes place in a child of the previous one, and even the first does not take place in the dynamic environment, there is no way to capture the dynamic environment using *$letrec**, and no way for combiners resulting from different binding expressions to be *mutually* recursive.

Derivation

The following expression defines *$letrec** using only previously defined features.

```
($define! $letrec*
   ($vau (bindings . body) env
      (eval ($if (null? bindings)
                 (list* $letrec bindings body)
                 (list $letrec
                       (list (car bindings))
                       (list* $letrec* (cdr bindings) body)))
            env)))
```

6.7.7 `$let-redirect`

`($let-redirect` ⟨exp⟩ ⟨bindings⟩ `.` ⟨body⟩`)`

⟨bindings⟩ and ⟨body⟩ should be as described for *$let*, §5.10.1.
The expression

(*$let-redirect* ⟨exp⟩
((⟨form$_1$⟩ ⟨exp$_1$⟩) ... (⟨form$_n$⟩ ⟨exp$_n$⟩))
. ⟨body⟩)

is equivalent to

((*eval* (*list* *$lambda* (⟨form$_1$⟩ ... ⟨form$_n$⟩) ⟨body⟩)
⟨exp⟩)
⟨exp$_n$⟩ ... ⟨exp$_n$⟩)

Rationale:
An ordinary *$let* expression depends on its dynamic environment in two different ways: the binding expressions are evaluated in the dynamic environment; and the local

environment, where the results are bound and the body is processed, is a child of the dynamic environment. *$let-redirect* is a general tool for eliminating the latter dependence: the binding expressions are evaluated in the dynamic environment, but the local environment is a child of some other environment specified by the programmer. This promotes semantic stability, by protecting the meaning of expressions in the body from unexpected changes to the client's environment (much as static scoping protects explicitly constructed compound combiners).

In the interests of maintaining clarity and orthogonality of semantics, there are no variants of *$let-redirect* analogous to the variants *$let**, etc., of *$let*. The variants of *$let* modulate its role in locally *augmenting* the current environment, whereas the primary purpose of *$let-redirect* is presumed to be locally *replacing* the current environment; so it was judged better to provide just one environment replacement device, insulating it as much as possible from complexities of environment augmentation.

Derivation

The following expression defines *$let-redirect* using only previously defined features.

```
($define! $let-redirect
   ($vau (exp bindings . body) env
      (eval (list* (eval (list* $lambda (map car bindings) body)
                         (eval exp
                               env))
                   (map cadr bindings))
            env)))
```

6.7.8 `$let-safe`

`($let-safe` ⟨bindings⟩ `.` ⟨body⟩`)`

⟨bindings⟩ and ⟨body⟩ should be as described for *$let*, §5.10.1.

The expression

(*$let-safe* ⟨bindings⟩ . ⟨body⟩)

is equivalent to

(*$let-redirect* (*make-kernel-standard-environment*)
⟨bindings⟩ . ⟨body⟩)

Rationale:

This is a common case of *$let-redirect*; providing a shorthand for it unclutters one's source code.

Derivation

The following expression defines *$let-safe* using only previously defined features.

```
($define! $let-safe
   ($vau (bindings . body) env
      (eval (list* $let-redirect
                   (make-kernel-standard-environment)
                   bindings
                   body)
            env)))
```

6.7.9 `$remote-eval`

`($remote-eval` ⟨exp1⟩ ⟨exp2⟩`)`

Operative *$remote-eval* evaluates ⟨exp2⟩ in the dynamic environment, then evaluates ⟨exp1⟩ as a tail context (§3.10) in the environment that must result from the first evaluation.

Rationale:

Operative *$remote-eval* provides a convenient way to evaluate an expression built into source code in an environment computed at run-time. This is a sometimes-useful task when working with first-class environments; if not provided as a standard feature, the task would have to be reprogrammed when needed, creating various preventable opportunities for accidents (cf. *G3* of §0.1.2).

Derivation

The following expression defines *$remote-eval* using only previous defined features.

```
($define! $remote-eval
   ($vau (o e) d
      (eval o (eval e d))))
```

6.7.10 `$bindings->environment`

`($bindings-environment .` ⟨bindings⟩`)`

⟨bindings⟩ should be as described for *$let*, §5.10.1.

The expression

(*$bindings->environment* . ⟨bindings⟩)

is equivalent to

(*$let-redirect* (*make-environment*) ⟨bindings⟩
(*get-current-environment*))

Rationale:

Operative *$bindings->environment* provides a convenient way to construct a first-class environment containing computed bindings for a predetermined set of symbols. It might, for example, be useful in constructing a suitable second argument for applicative *get-module*, §15.2.3.

Derivation

The following expression defines *$bindings->environment* using only previous defined features.

```
($define! $bindings->environment
   ($vau bindings denv
      (eval (list $let-redirect
                  (make-environment)
                  bindings
                  (list get-current-environment))
            denv)))
```

6.8 Environment mutation (optional)

6.8.1 $set!

`($set!` ⟨exp1⟩ ⟨formals⟩ ⟨exp2⟩`)`

⟨formals⟩ should be as described for the *$define!* operative, §4.9.1.

The *$set!* operative evaluates ⟨exp1⟩ and ⟨exp2⟩ in the dynamic environment; call the results *env* and *obj*. If *env* is not an environment, an error is signaled. Then the operative matches ⟨formals⟩ to *obj* in environment *env*. Thus, the symbols of ⟨formals⟩ are bound in *env* to the corresponding parts of *obj*. (The matching process was defined in §4.9.1.) The result returned by *$set!* is inert.

Rationale:

On support of arbitrary formal parameter trees as definiends, see the rationale discussion of §4.9.1.

The (second-class) Scheme operative *set!* allows Scheme code to modify any binding that is visible, regardless of whether or not the binding is local. Binding mutation in Kernel is more controlled, because the environment mutators can only affect capturable environments — *$set!* mutates an environment captured by the client, while equi-powerful primitive *$define!* does the capturing itself. Encapsulation of the *environment* type allows ancestor environments to be visible without being capturable, making them effectively read-only. (On restricting environment-mutation, see also the rationale for *$provide!*, §6.8.2.)

Derivation

The following expression defines *$set!*, using only previously defined features.

```
($define! $set!
   ($vau (exp1 formals exp2) env
      (eval (list $define! formals
                  (list (unwrap eval) exp2 env))
            (eval exp1 env))))
```

The key to this implementation is its use of *unwrap*. In the constructed combination

($define! ⟨formals⟩ ($eval ⟨exp2⟩ env))

(where we write *$eval* for the operative that underlies applicative *eval*), *$define!* evaluates its second operand in its dynamic environment, and then matches ⟨formals⟩ to the result, again in its dynamic environment. Here, its dynamic environment is the result of *$set!*'s local evaluation of `(eval exp1 env)`, call it *e1*; but because the second operand (*$eval* ⟨exp2⟩ *env*) is an operative combination, ⟨exp2⟩ is evaluated only in *env*, the dynamic environment of the call to *$set!* — not in *e1*.

6.8.2 $provide!

`($provide!` ⟨symbols⟩ `.` ⟨body⟩`)`

⟨symbols⟩ must be a finite list of symbols, containing no duplicates. ⟨body⟩ must be a finite list.

The *$provide!* operative constructs a child e of the dynamic environment d; evaluates the elements of ⟨body⟩ in e, from left to right, discarding all of the results; and exports all of the bindings of symbols in ⟨symbols⟩ from e to d, i.e., binds each symbol in d to the result of looking it up in e. The result returned by *$provide!* is inert.

The expression

($provide! ⟨symbols⟩ . ⟨body⟩)

is equivalent to

($define! ⟨symbols⟩
   ($let ()
      ($sequence . ⟨body⟩)
      (list . ⟨symbols⟩)))

Rationale:

An important encapsulation technique in Kernel is to store private information in a local environment, and then construct combiners in that environment and export them. If only one combiner is being exported, the easiest way to do this is to simply *$define!* the combiner, and put a *$let* around the definition expression, as in

```
($define! count
   ($letrec ((self     (get-current-environment))
             (counter  0))
      ($lambda ()
         ($set! self counter (+ counter 1))
         counter)))
```

When more than one combiner is exported to the surrounding environment, it would be technically possible to locally capture the surrounding environment and use *$set!* for the exportation:

```
($let ((outside  (get-current-environment)))
   ...
   ($set! outside (foo bar quux) (list foo bar quux)))
```

but this would be bad style because the local binding of `outside` is visible to everything in the local block, granting power to mutate the surrounding environment to everything in the local environment, and therefore (probably) to anything that captures a descendent of the local environment. Permission to mutate an environment should be conferred sparingly, to avoid accidents (*G3* of §0.1.2). Another technically possible approach would be to *$define!* a compound definiend with a *$let*-expression definition, and end the *$let*-expression with a *list* of values to match the definiend:

```
($define! (foo bar quux)
   ($let ()
      ...
      (list foo bar quux)))
```

but this too would be accident-prone, because the programmer would have to manually maintain coordination between the list of symbols at the top of the construct, and the list of combiners at the bottom of the construct. The corresponding *$provide!*-expression,

```
($provide! (foo bar quux)
   ...)
```

accomplishes the same multiple exportation, without either widely distributing permission to mutate the surrounding environment or requiring the programmer to coordinate two lists at opposite ends of the construct. For a full example of its use, see the derivation code for promises, §9.1.3.

Derivation

The following expression defines *$provide!*, using only previously defined features.

```
($define! $provide!
   ($vau (symbols . body) env
      (eval (list $define! symbols
                  (list $let ()
```

```
                (list* $sequence body)
                (list* list symbols)))
        env)))
```

6.8.3 `$import!`

(`$import!` ⟨exp⟩ . ⟨symbols⟩)

⟨symbols⟩ must be a list of symbols.

The *$import!* operative evaluates ⟨exp⟩ in the dynamic environment; call the result *env*. *env* must be an environment. Each distinct symbol *s* in ⟨symbols⟩ is evaluated in *env*, and *s* is bound in the dynamic environment to the result of this evaluation.

The expression

(*$import!* ⟨exp⟩ . ⟨symbols⟩)

is equivalent to

(*$define!* ⟨symbols⟩ (*$remote-eval* (*list* ⟨symbols⟩) ⟨exp⟩))

Rationale:

Extracting bindings from a first-class environment is of immediate interest in conjunction with *get-module* (§15.2.3).

Derivation

The following expression defines *$import!*, using only previous defined features.

```
($define! $import!
   ($vau (exp . symbols) env
      (eval (list $set!
                  env
                  symbols
                  (cons list symbols))
            (eval exp env))))
```

6.9 Control

6.9.1 `for-each`

(`for-each` *applicative* . *lists*)

lists must be a nonempty list of lists; if there are two or more, they should all be the same length. If *lists* is empty, or if all of its elements are not lists of the same length, an error is signaled.

for-each behaves identically to ***map***, except that instead of accumulating and returning a list of the results of the element-wise applications, the results of the applications are discarded and the result returned by ***for-each*** is inert.

Rationale:

The Scheme procedure *for-each* is defined to perform its applications from left to right ([KeClRe98, §6.4]). By Kernel's general policy on list handling (rationale for §3.9), though, if *for-each* were required to process from left to right, it would have to loop forever on cyclic lists. Handling cyclic lists in finite time was judged a more useful behavior, and uniformity with *map* was preferred, so the order of processing is pointedly unspecified.

Derivation

The following expression defines ***for-each*** using only previously defined features. (Since we're only trying to show derivability, we can take the mathematician's way out by reducing it to a previously solved problem.)

```
($define! for-each
   (wrap ($vau x env
            (apply map x env)
            #inert)))
```

7 Continuations

A continuation is a plan for all future computation, parameterized by a value to be provided, and contingent on the states of all mutable data structures (which notably may include environments, §4.9). When the Kernel evaluator is invoked, the invoker provides a continuation to which the result of the evaluation will normally be returned.

For example, when ***$if*** (§4.5.2) evaluates its ⟨test⟩ operand, the continuation provided for the result expects to be given a boolean value; and, depending on which boolean it gets, it will evaluate either the ⟨consequent⟩ or the ⟨alternative⟩ operand as a tail context — that is, the continuation provided for the result of evaluating the selected operand is the same continuation that was provided for the result of the call to ***$if***.

A Kernel program may sometimes *capture* a continuation; that is, acquire a reference to it as a first-class object. The basic means of continuation capture is applicative ***call/cc*** (§7.2.2).

Given a first-class continuation c, a combiner can be constructed that will *abnormally pass* its operand tree to c (as opposed to the *normal return* of values to continuations, which was mentioned above, and which will be discussed below in §7.1). In the simplest case, the abnormally passed value arrives at c as if it had been normally returned to c. In general, continuations bypassed by the abnormal pass may have *entry/exit guards* attached to them, and these guards can intercept the abnormal pass before it reaches c. Each entry/exit guard consists of a *selector* continuation, which designates which abnormal passes the guard will intercept, and an *interceptor* applicative that performs the interception when selected; see §§7.2.4 and 7.2.5 (applicatives ***guard-continuation*** and ***continuation->applicative***).

Continuations are immutable, and are ***equal?*** iff ***eq?***. The *continuation* type is encapsulated.

Rationale:

First-class continuations can be used to implement a wide variety of advanced control structures. Their inclusion in Kernel may therefore be justified from the generality design goal. (Justification from first-class-ness Guideline *G1a* of §0.1.2 has a suggestion of circularity about it, because it's problematic whether continuations would qualify as 'manipulable entities' if they weren't capturable.) However, the ability to capture and invoke first-class continuations leads to a subtle partial undermining of the ability of algorithms to regulate *themselves* (which falls under the aegis of accident-prevention, Guideline *G3* of §0.1.2): when an algorithm initiates a subcomputation, while it gives up a great deal of control over subsequent events, there is usually an expectation that the subcomputation will return (and thus, the algorithm resume from that point) *at most once.* Kernel redresses this problem, while not-incidentally supporting a generalization of conventional exception-handling facilities, by means of a flexible *entry/exit guard* device (§§7.2.4, 7.3.3) that allows the initiating algorithm to regulate abnormal passes into or out of the dynamic extent of the subcomputation.

Scheme first-class continuations are procedures. That design choice is defensible in Scheme, where anything that can be done with first-class continuations can also be done with first-class procedures, and vice versa. In Kernel, though, each of the two types supports operations that the other does not: arbitrary combiners cannot act as guard selectors, nor as parents in continuation construction; while the concept of ***wrap***ping and ***unwrap***ping has no relevance at all to the internal semantics of continuations, and therefore ought to be orthogonal to the language interface of the *continuation* type.

7.1 Dynamic extents

Each call to the Kernel evaluator is provided with a continuation to which it would normally return its result; hence, at each point in a Kernel computation, there is a unique stack of continuations waiting for normal return of results. Continuation c_2 is a *child* of continuation c_1 iff normal receipt of a result passed to c_2 is scheduled by an evaluator call that would normally return its result to c_1. (That is, whenever c_2 is on the normal continuation stack, c_1 is immediately below it on the stack.)

Each call to a Kernel operative is provided with a continuation to which it would normally return its result; but this is not a separate case from evaluator calls. Each operative call is initiated by the evaluator because the object being evaluated is a combination whose combiner is that operative (evaluator Step 3a in §3.3); and since the result of the operative call will become the result of the evaluator call, the continuation provided to the operative call is just the continuation provided to the evaluator call. In effect, the operative call is a suffix of the processing of the evaluator call.

Some examples:

- When ***$if*** is called, the continuation for the evaluation of the ⟨test⟩ is a child of the continuation for the call to ***$if***.

- When a compound operative is called, the continuations for the evaluations of expressions in the body of the operative (via ***$sequence***, §5.1.1) are children of the continuation for the call to the operative (except that, if the body is acyclic, the last expression-evaluation has no distinct continuation because it's a tail context).

- When the evaluator evaluates the operands to an applicative combination, the continuation for the evaluation of each operand is a child of the continuation for the evaluation of the combination. It doesn't matter whether the applicative happens to be a converted continuation (via ***continuation->applicative***, §7.2.5), whose underlying operative will abnormally pass the argument list; the operand evaluations take place before that, and are part of the combination evaluation, and that is all the child-continuation relation reflects.

Since the set of ancestors of a continuation is determined solely by the identity of the evaluator call whose computation would resume on normal return to the continuation, the set of ancestors of the continuation is fixed for all time at the moment the continuation is created.

The *dynamic extent* of a continuation c consists of c, all its descendants, and all evaluator calls whose results would normally be passed to c or to any of its descendants.

Rationale:

Exception-handling facilities in most modern programming languages involve a logical hierarchy of exceptional conditions. If the language has manifest types, the type hierarchy is used for the purpose; in languages with no manifest type hierarchy, a special type-like logical hierarchy of exceptions may be introduced. While the introduction of an entirely new hierarchy for exceptions would not sit well with Kernel's simplicity design goal, the natural ordering of continuations in a Kernel computation provides a pre-existing hierarchy suitable for the purpose. The ability to extend the hierarchy explicitly (i.e., by specifying the extension without having to be within the dynamic extent of the continuation to be extended), which in a manifestly typed language would be accomplished via programmer-defined subtyping, is supported in Kernel by applicatives ***extend-continuation*** and ***guard-continuation*** (§§7.2.3, 7.2.4).

7.2 Primitive features

7.2.1 `continuation?`

(`continuation?` . *objects*)

The primitive type predicate for type *continuation*.

Rationale:

The possibility was considered of providing a distinct type for guard selectors. It would then be possible to export a selector for a dynamic extent without exporting the continuation of that dynamic extent. However, there did not appear to be sufficient need for such an arrangement to justify the additional complexity. Worse, the complexity of the feature would tend to take on a life of its own: once selectors were made a separate type, there would be a natural temptation to introduce additional selector constructors — first union, then intersection, and by that point it would be hard to justifying stopping short of Turing power.

7.2.2 `call/cc`

(`call/cc` *combiner*)

Calls *combiner* in the dynamic environment as a tail context, passing as sole operand to it the continuation to which ***`call/cc`*** would normally return its result. (That is, constructs such a combination and evaluates it in the dynamic environment.)

Cf. operative ***`$let/cc`***, §7.3.2.

Rationale:

Since the operand list of the call to *combiner* is a list whose one element is self-evaluating and whose one pair is freshly allocated, applying argument-evaluation to the operand list has no observable effect; so any restriction on the argument type beyond type *combiner* would be arbitrary.

The name of the analogous feature in Scheme is `call-with-current-continuation`; the *R5RS* notes that this name was coined in 1982 and "opinions differ on the merits of such a long name." In choosing between these two alternative names in Kernel, accident avoidance and its correlate clarity were first considered, but failed to produce a compelling resolution (the long name is more self-documenting, the short name allows more uncluttered code). The choice was finally based on a gedankenexperiment: if the standard name had been made `call/cc` back in 1982, how many programmers today would be renaming it to `call-with-current-continuation`?

7.2.3 `extend-continuation`

(`extend-continuation` *continuation applicative environment*)
(`extend-continuation` *continuation applicative*)

When the first syntax is used, the ***extend-continuation*** applicative constructs and returns a new child of *continuation* that, when it normally receives a value v, calls the underlying combiner of *applicative* with dynamic environment *environment* and operand tree v, the result of the call normally to be returned to *continuation*.

The second syntax is syntactic sugar, equivalent to specifying an empty environment to the first syntax; that is,

```
   (extend-continuation c a)
≡  (extend-continuation c a (make-environment))
```

Rationale:

The decision that v should be passed as the entire operand tree to the underlying combiner of *applicative*, rather than as a single operand, parallels the design of applicative ***continuation->applicative***, §7.2.5.

In order to wield Kernel's continuation hierarchy effectively for exception handling, the programmer must be able to extend the hierarchy of possible destination continuations without triggering entry/exit guards, which are meant to intercept actual exceptions, and oughtn't get in the way of defining potential destinations when no exceptional condition yet exists. Applicative ***call/cc*** cannot do this, because it must be used from inside the dynamic extent of the continuation that it captures.

Applicative ***extend-continuation*** allows the programmer to remotely create a child of an existing continuation c by prepending some arbitrary computation (embodied by argument *applicative*) to the computation by c. The complementary ability to place entry/exit guards on a remotely created continuation (extending abnormal computation as ***extend-continuation*** does normal computation) is provided by ***guard-continuation***, §7.2.4.

7.2.4 `guard-continuation`

(`guard-continuation` *entry-guards continuation exit-guards*)

entry-guards and *exit-guards* should each be a list of clauses; each clause should be a list of length two, whose first element is a continuation, and whose second element is an applicative whose underlying combiner is operative.

Applicative ***guard-continuation*** constructs two continuations: a child of *continuation*, called the *outer continuation*; and a child of the outer continuation, called the *inner continuation*. The inner continuation is returned as the result of the call to ***guard-continuation***.

When the inner continuation normally receives a value, it passes the value normally to the outer continuation; and when the outer continuation normally receives a value, it passes the value normally to *continuation*. Thus, in the absence of abnormal passing, the inner and outer continuations each have the same behavior as *continuation*.

The two elements of each guard clause are called, respectively, the *selector* and the *interceptor*. The selector continuation is used in deciding whether to intercept a given abnormal pass, and the interceptor applicative is called to perform customized action when interception occurs. (Selection and interception are explained in §7.2.5.) At the beginning of the call to ***guard-continuation***, internal copies are made of the evaluation structures (§4.7.2) of *entry-guards* and *exit-guards*, so that the selectors and interceptors contained in the arguments at that time remain fixed thereafter, independent of any subsequent mutations to the arguments.

Rationale:

guard-continuation copies its *entry-guards* and *exit-guards* arguments for the same reason that ***$vau*** (§4.10.3) copies its operands: they represent algorithmic information that may persist indefinitely, and should not be subject to internal change. (Algorithm behavior can be modulated through interaction with mutable structures external to the algorithm, which is less accident-prone than mutation of the algorithm itself.)

Early in the design, it was envisioned that dynamic-extent guards would be specified by an operative, with the guard lists being special syntax, similarly to the binding list to ***$let***. Thus, instead of

```
(guard-continuation (list (list s1 i1) ... (list sn in))
                    c
                    (list (list s'1 i'1) ... (list s'n' i'n'))
```

one would write

```
($guard-continuation ((s1 i1) ... (sn in))
                     c
                     ((s'1 i'1) ... (s'n' i'n'))
```

However, as noted elsewhere (§6.7.2), we prefer to reserve the use of operatives for situations where they are really needed. Binding clauses *have* to be handled operatively because they contain parameter trees, which cannot be evaluated; but here, every single sublist element s_k, i_k, s'_k, and i'_k is to be evaluated in the dynamic environment.

The type constraint on interceptors follows mainly from Guidelines *G1* (uniformity) and *G3* (accident avoidance) of §0.1.2. It is generally more convenient, and safer, to work with applicatives; but the value passed to an interceptor usually should not be evaluated — and when it should be evaluated, to avoid accidents its evaluation should be performed within the jurisdiction of the interceptor. So: when constructing an interceptor call we expect to unwrap the interceptor, and, in order to handle all interceptors in a uniform way, we unwrap each interceptor *exactly once* (which means the interceptor can't have been operative to begin with); *but*, if the underlying combiner of an interceptor were itself applicative, the value passed to it would still be evaluated outside the jurisdiction of the interceptor; so we don't allow the interceptor to be multiply wrapped, and leave it to the interceptor to explicitly call ***eval*** at its discretion.

7.2.5 `continuation->applicative`

`(continuation->applicative` *continuation*`)`

Returns an applicative whose underlying operative abnormally passes its operand tree to *continuation*, thus: A series of interceptors are selected to handle the abnormal pass, and a continuation is derived that will *normally* perform all the interceptions in sequence and pass some value to the destination of the originally abnormal pass. The operand tree is then normally passed to the derived continuation.

The processes of selection and interception are detailed below.

Rationale:

A continuation receives one object, normally the value returned by a subcomputation for which the continuation was constructed; an operative receives one object, called an *operand tree*. Naturally, when a continuation is converted to an operative, the continuation receives the object received by the operative, i.e., its operand tree.

There is a mild mismatch of expectations in Kernel between operatives, which usually expect lists because the evaluator algorithm provides lists of arguments when evaluating applicative combinations (which most combinations are); and continuations, which tend by nature to expect the result of a single computation, rather than a list of results of separate computations. Both expectations are readily overcome —again, in Kernel— the former by deconstructing a list received by a continuation, via a list-structured definiend (see the rationale of §4.9.1); the latter by passing an atomic value in place of an argument list (see the rationale of §4.10.5).

In Scheme, however, the mismatch isn't mild, because neither expectation is readily overcome. Deconstructing a list received by a continuation is laborious, since formal parameter lists are only supported for deconstructing the argument list received by a procedure; and passing an atomic value in place of an argument list is specifically forbidden (e.g., `(apply c 5)` is illegal in Scheme). Scheme gets around the problem by skewing the correspondence between argument list received by a procedure and result received by a continuation: a Scheme first-class continuation is a procedure that takes one argument, and behaves as if its argument were the returned value. Thus, for continuation `c`, Kernel `(`***`apply-continuation`*** `c 5)` becomes Scheme `(`***`apply`*** `c (list 5))`. (The point here being structural, we use Scheme ***`apply`***; in practice a Scheme programmer would write `(c 5)`, which has no close analog in Kernel since Kernel continuations are not combiners.)

The Scheme single-operand approach was considered for Kernel. Once historical precedent is set aside (the compatibility motive being toxic to the Kernel design, as noted in §0.1.1), there remains a tenuous argument from accident-avoidance, that the single-operand approach would reduce the amount of structural intervention needed, and thus the number of opportunities for error. Ultimately, though, the single-operand approach was rejected on the more primordial grounds that it treats continuations as a special case — a conclusion that is subtly affirmed by its fostering of the illusion of multiple-value returns (on which see, again, the rationale discussion of §4.9.1.)

Selection

The selection of interceptors for an abnormal pass is based on

- the continuation to which the value is being abnormally passed, called the *destination* of the abnormal pass; and
- the continuation that would have normally received the result of the abnormal combination (i.e., the combination whose combiner was constructed by ***continuation->applicative***, and whose evaluation initiated the abnormal pass), called the *source* of the abnormal pass.

In selecting guards to intercept the abnormal pass, an exit-guard list is considered iff the abnormal pass exits the dynamic extent of the associated *inner* continuation (that is, if the source is within that dynamic extent, and the destination is not within it); and an entry-guard list is considered iff the abnormal pass enters the dynamic extent of the associated *outer* continuation (the destination is within that extent and the source isn't). Exit-guard lists are considered first, proceeding from smallest to largest dynamic extent (thus, outward from the source), followed by entry-guard lists proceeding from largest to smallest dynamic extent (thus, inward to the destination). Note that no extents are exited if the destination is within the extent of the source, and no extents are entered if the source is within the extent of the destination.

For each exit-guard list considered, the first interceptor (if any) is selected whose selector's dynamic extent contains the destination. Symmetrically, for each entry-guard list considered, the first interceptor (if any) is selected whose selector's dynamic extent contains the source. Thus, at most one interceptor is selected from each list.

Rationale:

It doesn't matter whether a guard list is cyclic, because testing a selector has no side-effects; cf. the handling of cyclic parent lists by ***make-environment***, §4.8.4.

Interceptors are selected in the order that the guarded dynamic-extent barriers are passed through — exit from successively larger extents, followed by entrance to successively smaller extents. Since *all* relevant guard lists are considered along the path from source to destination, selecting at most *one* match from each list maximizes the programmer's control over the interception process — analogously to taking a conjunction of disjunctions in boolean algebra.

In the case of exit guards, this conjunction-of-disjunctions algorithm is just the way exception-handling facilities in other languages typically work. When an exception is thrown (to use Java terminology), it is caught first by the first compatible `catch` clause of the most recent enclosing `try` block; and then, if the exception is rethrown, no other `catch` clause is considered for that `try`, but control continues outward to successively less recent blocks, stopping (if repeatedly rethrown) at no more than one `catch` clause for each `try`.

The handling of entry guards is ruthlessly symmetric to that of exit guards. Languages with type-based exception hierarchies have no clear analog to Kernel's entry guards, which occur as a concept only because the hierarchy of exception destinations is alike in kind to the hierarchy of exception sources; there is therefore, within the author's experience, no precedent for the entry-guard facility, and its design is based entirely on the strength of the exit/entry symmetry.

There was some deliberation over the possibility that, when specifying a nontrivial selector for an entry guard (i.e., a selector that doesn't simply intercept everything), the guarded destination extent will usually have a trusted source extent from which it allows unobstructed abnormal passing, and will want to intercept just those abnormal entries that *do not* come from the trusted extent. One could imagine reversing the polarity of selection on entry guards, so that an *exit* guard is selected if the destination *belongs* to the selector's extent, but an *entry* guard is selected if the source *does not belong* to the selector's extent. However, even if this does turn out to be the most common usage pattern for entry guards, it was judged more symmetrical and straightforward (thus, less accident-prone) to use the same selection algorithm for entry as for exit. Complementary behavior can be achieved, when wanted, by specifying a pass-through guard for the trusted case followed by a restrictive guard for all other cases; for example,

```
(guard-dynamic-extent
   (list (list trusted           ($lambda (x #ignore) x))
         (list root-continuation ($lambda (x divert) ···)))
   ($lambda () ···)
   ())
```

Interception

In the derived normal handling of an abnormal pass, each selected interceptor is called with two arguments (that is, its underlying operative is called with two operands), in the dynamic environment of the associated call to ***guard-continuation***. The two arguments are: first, the value being passed; and second, an applicative based on the associated outer continuation (as constructed by ***continuation->applicative***). The call takes place within the dynamic extent of the associated outer continuation, but outside the dynamic extent of the associated inner continuation; thus, if the interceptor calls its second argument, no interceptions will be caused (since exit guards only care about abnormal exit from the *inner* continuation; to be precise, no interceptions will be caused external to the interceptor call). If the interceptor normally returns a value, that value becomes the first argument to the next interceptor, or becomes the value normally passed to the destination if the returning interceptor was the last selected.

To make this behavior happen, for each selected interceptor a new continuation is constructed to normally receive its result. The constructed continuation is a child of the associated outer continuation (so that the call is inside the outer extent but outside the inner extent); but when it receives a value, it *normally* sends the received value to wherever it should go next — passing the value to the destination, or initiating the call to the next interceptor. One additional continuation is then constructed to normally receive the value that was abnormally passed, and initiate the first interceptor call.

Rationale:

In making the design of Kernel extent guarding work smoothly, a key insight was that, when an abnormal pass occurs, all the consequent interceptions must be scheduled at once, with the *normal* return of a result from each interceptor causing the value to be passed on to the next interceptor. That is, passing of the value from one interceptor to the next is not itself subject to interception (*normal = not subject to interception*). Exception-handling facilities in other languages typically require a caught exception to be rethrown; in our terminology, normal return from an interceptor would go to the outer continuation, and an abnormal pass would be required in order to send the value further along its way toward the destination. However, that approach only works when all interceptors are selected by destination. In Kernel, where entry interceptors are selected by source, initiating a new abnormal pass from within an interceptor might cause entirely different entry-guard selections, because the new abnormal pass has the earlier interceptor as its source. Thus, the entire chain of selected interceptions is scheduled to occur as a *normal* process; and, in case the interceptor wants to cut the process short, it is given a route to the associated outer continuation as its second argument. (The possibility was considered of also providing the destination as an argument — but then, by symmetry one ought also to provide the source, and at that point the whole mechanism is getting awfully elaborate.) The handling of outer and inner continuations is then carefully arranged so that, when an interceptor does call its second argument, no interceptions will occur (external to the interceptor call).

If an interceptor doesn't simply ignore its second argument, its interest will almost always be to pass an object to the outer continuation. At need, the interceptor could construct a continuation with equivalent behavior to the outer continuation, by evaluating the following expression (assuming the local name for the applicative is `divert`):

```
($let/cc c (extend-continuation c divert))
```

This expression only works properly if it is evaluated in a dynamic extent that has no entry/exit guards between it and the outer continuation being simulated. Thus, not only is the applicative form of the outer continuation more often wanted, but it is also *safer to work with*, because if it should somehow fall into the hands of a computation in some remote dynamic extent, it would grant less power to that remote extent.

Providing the abnormally passed value as the first argument to the interceptor and the applicative as the second argument, rather than the other way around, avoids illusory cues to the programmer. If the applicative were the first argument and the value second, the programmer might be tempted to think of the argument tree as a combination; but if the argument tree were taken as a combination, the value would be the cadr of the combination when it ought to be the cdr, and even if that detail were fixed, the combination would be *applicative*, when one would almost never intend the value or, worse, parts of it to be evaluated as arguments before being passed to the outer continuation.

R5RS Scheme supports a limited form of extent guarding through its procedure *dynamic-wind*, which unconditionally intercepts all entry/exit of its dynamic extent. The *R5RS* doesn't fully define how *dynamic-wind* interacts with first-class continuations (specifically, what happens when a continuation is captured from inside an interceptor), and says nothing about how ***dynamic-wind*** interacts with error-signaling. For perspective, here is a Kernel implementation of ***dynamic-wind***.

```
($define! dynamic-wind
   ($lambda (before thunk after)
      (guard-dynamic-extent
         (list (list root-continuation
                     ($lambda (value #ignore)
                        (before)
                        value)))
         ($lambda ()
            (before)
            ($let ((result  (thunk)))
               (after)
               result))
         (list (list root-continuation
                     ($lambda (value #ignore)
                        (after)
                        value))))))
```

7.2.6 `root-continuation`

`root-continuation`

This continuation is the ancestor of all other continuations. When it normally receives a value, it terminates the Kernel session. (For example, if the system is running a read-eval-print loop, it exits the loop.)

Cf. applicative ***exit***, §7.3.4.

Rationale:

If the hierarchy of continuations didn't have a maximal element, or if that element weren't made available to the programmer, there would be no way to create a guard clause that is always selected. See for example the implementation of ***dynamic-wind*** in the rationale discussion of §7.2.5.

7.2.7 `error-continuation`

`error-continuation`

The dynamic extent of this continuation is mutually disjoint from the dynamic extent in which Kernel computation usually occurs (such as the dynamic extent in which the Kernel system would run a read-eval-print loop).

When this continuation normally receives a value, it provides a diagnostic message to the user of the Kernel system, on the assumption that the received value is an attempt to describe some error that aborted a computation; and then resumes operation of the Kernel system at some point that is outside of all user-defined computation. (For example, if the system is running a read-eval-print loop, operation may resume by continuing from the top of the loop.)

The diagnostic message is not made available to any Kernel computation, and is therefore permitted to contain information that violates abstractions within the system (as discussed in §3.7).

When an error is signaled during a Kernel computation, the signaling action consists of an abnormal pass to some continuation in the dynamic extent of *error-continuation*.

Rationale:

Making *error-continuation* available to the programmer serves two purposes: it allows the programmer to duplicate the built-in action of error-signaling (modulo the format of the value used to describe the error, which this preliminary report does not specify), per extensibility Guideline *G1b* of §0.1.2; and it allows for an exit-guard that will be selected whenever an error is signaled within the guarded dynamic extent (see for example the derivation of predicate *$binds?*, §6.7.1.)

7.3 Library features

7.3.1 `apply-continuation`

(`apply-continuation` *continuation object*)

Applicative *apply-continuation* converts its first argument to an applicative as if by *continuation->applicative*, and then applies it as usual. That is,

(*apply-continuation continuation object*)
≡ (*apply* (*continuation->applicative continuation*) *object*)

Rationale:

This applicative provides a less cluttered way to abnormally pass values to continuations than by explicitly calling *continuation->applicative*.

There is also an element of added programmer safety in using *apply-continuation* instead of *continuation->applicative*, because *apply-continuation* specifically does not support an optional third, environment argument. Since an abnormal pass ignores its dynamic environment, the programmer who specifies an environment argument when applying a continuation is making a conceptual error, which *apply-continuation* signals.

Derivation

The following expression defines the *apply-continuation* applicative, using only previously defined features.

```
($define! apply-continuation
   ($lambda (c o)
      (apply (continuation->applicative c) o)))
```

7.3.2 $let/cc

(`$let/cc` ⟨symbol⟩ `.` ⟨objects⟩)

A child environment e of the dynamic environment is created, containing a binding of ⟨symbol⟩ to the continuation to which the result of the call to ***$let/cc*** should normally return; then, the subexpressions of ⟨objects⟩ are evaluated in e from left to right, with the last (if any) evaluated as a tail context, or if ⟨objects⟩ is empty the result is inert. That is,

(***$let/cc*** ⟨symbol⟩ . ⟨objects⟩)
≡ (***call/cc*** (***$lambda*** (⟨symbol⟩) . ⟨objects⟩))

Rationale:
Almost all continuation capture follows the pattern abstracted by ***$let/cc***. It is equi-powerful with ***call/cc***, either being easily rewritten using the other. (Derivations are given below in both directions.) Including ***call/cc*** in Kernel forestalls incompatibilities that could result from its omission; and making ***call/cc*** the primitive underscores that capturing the current continuation does not require an operative.

Derivation

The following expression defines the ***$let/cc*** applicative, using only previously defined features.

```
($define! $let/cc
   ($vau (symbol . body) env
      (eval (list call/cc (list* $lambda (list symbol) body))
            env)))
```

For perspective, here is an implementation of ***call/cc*** using ***$let/cc***:

```
($define! call/cc
   (wrap ($vau (appv) env
            ($let/cc cont
               (apply appv (list cont) env)))))
```

7.3.3 guard-dynamic-extent

(`guard-dynamic-extent` *entry-guards combiner exit-guards*)

This applicative extends the current continuation with the specified guards, and calls *combiner* in the dynamic extent of the new continuation, with no operands and the dynamic environment of the call to ***guard-dynamic-extent***.

Derivation

The following expression defines the *guard-dynamic-extent* applicative, using only previously defined features.

```
($define! guard-dynamic-extent
   (wrap ($vau (entry-guards combiner exit-guards) env

      ($let ((local  (get-current-environment)))
         ($let/cc  bypass
            ($set! local bypass bypass)
            (apply-continuation
               (car ($let/cc  cont
                        ($set! local enter-through-bypass
                           (continuation->applicative cont))
                        (list bypass)))
               #inert)))

      ($let/cc  cont
         (enter-through-bypass
            (extend-continuation
               (guard-continuation
                  (cons (list bypass ($lambda (v . #ignore) v))
                        entry-guards)
                  cont
                  exit-guards)
               ($lambda #ignore
                  (apply combiner () env))))))))
```

To the client, *guard-dynamic-extent* appears to call *combiner* from *inside the guarded continuation* without triggering the *entry-guards* to get there. Internally, though, it uses *guarded-continuation* to construct the new continuation and then abnormally passes control to it. To avoid interception by *entry-guards*, a prepended entry guard overrides all of them (by being first on the list) to allow unimpeded abnormal entry from local dedicated continuation *bypass*. (In principle, a clever interpreter might correctly deduce that *bypass* cannot possibly be used this way again, and remove the prepended entry guard, making *bypass*, and *local*, available for garbage collection.)

7.3.4 exit

`(exit)`

Applicative *exit* initiates an abnormal transfer of `#inert` to *root-continuation* (§7.2.6). That is,

```
(exit ) ≡ (apply-continuation root-continuation #inert)
```

Rationale:

The main advantage of *exit* over *root-continuation* is that it is an applicative, rather than a continuation. An obvious benefit is that combiners are easy to call (it's their reason for being, after all), whereas a continuation can only be invoked through an auxiliary call to another combiner (typically *apply-continuation*, occasionally *continuation->applicative*). A more subtle benefit (though it comes into play only mildly in this case) is that combiners induce type errors *eagerly.* If an operative receives an operand tree of the wrong type, it signals an error from within the dynamic extent of the call to the operative. If a continuation receives a value of the wrong type, it too signals an error — but by the time the continuation actually receives a value, the source dynamic extent of the abnormal pass has been lost, reducing Kernel's ability to provide useful diagnostic information about the source of the problem. It is therefore often desirable, for safety, to build known type constraints into an applicative shell around a continuation, and call the shell rather than the continuation.

A secondary advantage of *exit* over *root-continuation* is that its name is both shorter, thus less code-cluttering; and more descriptive of its function in terminating computation, whereas `root-continuation` primarily characterizes the role of the continuation as a guard selector.

Derivation

The following expression defines the *exit* applicative, using only previously defined features.

```
($define! exit
   ($lambda ()
      (apply-continuation root-continuation #inert)))
```

8 Encapsulations

An *encapsulation* is an object that refers to another object, called its content. The Kernel data type *encapsulation* is encapsulated.

Two encapsulations are *equal?* iff they are *eq?* (§§4.2.1, 4.3.1).

Encapsulations are immutable.

Rationale:

Type *encapsulation* is not a primitive type. Rather, it is the union of arbitrarily many generated primitive types, each with its own matching set of constructor, primitive predicate, and accessor. Primitive applicative *make-encapsulation-type* generates a fresh such set each time it is called. These generated sets of tools are minimalist, but, by isolating the tools in a local environment, the programmer can channel access to them through a customized interface, effectively synthesizing arbitrary new encapsulated types (per Guidelines *G1b* and *G4* of §0.1.2).

Programmer-defined encapsulated types are discussed in the rationale of §3.4. For an example of the practical use of type encapsulation to create customized primitive types, see the derivation in §9.1.3.

The behavior of predicate ***equal?*** on encapsulations is tied to that of ***eq?*** to prevent a more permissive predicate ***equal?*** from exposing information about the contents of encapsulations, and thus weakening the encapsulation of programmer-defined types.

The concept for Kernel encapsulated types was inspired by *seals* from [Mor73] (on which see also [SumPi04]).

8.1 Primitive features

8.1.1 `make-encapsulation-type`

`(make-encapsulation-type)`

Returns a list of the form `(`*e p? d*`)`, where *e*, *p?*, and *d* are applicatives, as follows. Each call to ***make-encapsulation-type*** returns different applicatives *e*, *p?*, and *d*.

- *e* is an applicative that takes one argument, and returns a fresh encapsulation with the argument as content. Encapsulations returned on different occasions are not ***eq?***.
- *p?* is a primitive type predicate, that takes zero or more arguments and returns true iff all of them are encapsulations generated by *e*.
- *d* is an applicative that takes one argument; if the argument is not an encapsulation generated by *e*, an error is signaled, otherwise the content of the encapsulation is returned.

That is, the predicate *p?* only recognizes, and the decapsulator *d* only extracts the content of, encapsulations created by the encapsulator *e* that was returned by *the same call* to ***make-encapsulation-type***.

9 Promises

A *promise* is an object that represents the potential to determine a value. The value may be the result of an arbitrary computation that will not be performed until the value must be determined (constructor ***$lazy***, §9.1.3); or, in advanced usage, the value may be determined before the promise is constructed (constructor ***memoize***, §9.1.4).

The value determined by a promise is obtained by *forcing* it (applicative ***force***, §9.1.2). A given promise cannot determine different values on different occasions that it is forced. Also, if a promise determines its value by computation, and that computation has already been completed, forcing the promise again will produce the previously determined result without re-initiating the computation to determine it.

The Kernel data type *promise* is encapsulated.

The general rules for predicate ***eq?*** (§4.2.1) only require it to distinguish promises if they can exhibit different behavior; the resulting leeway for variation between implementations is similar, in both cause and effect, to that for ***eq?***-ness of operatives (§4.10). For example, if two promises, constructed on different occasions, would perform the same computation to determine their values, and that computation has no side-effects and must always return the same value, the promises may or may not be ***eq?***. Two promises are ***equal?*** iff they are ***eq?*** (§4.3.1).

Rationale:

Kernel promises have a distinctly different character from Scheme promises, because the encapsulated status of Kernel promises allows Kernel to partition objects into promises and non-promises. This in turn allows promises to be intermixed freely with non-promise objects, without losing track of which is which — so that, in this case, encapsulation promotes first-class-ness (*G1a* of §0.1.2). The distinct character of Kernel promises will be elaborated further in rationale discussions throughout the section.

There are two versions of Scheme promises. Promises in the *R5RS* are supported by two combiners, ***delay*** (which is operative) and ***force***; but their implementation in the *R5RS* contains a memory leak — accumulation of continuations in a situation that ought to have been tail-recursive (§3.10). SRFI-45, [vT04], fixes the memory leak, introducing in the process two additional combiners, ***lazy*** (another operative) and ***eager***. Kernel promises more nearly resemble those of SRFI-45 than of the *R5RS*; but their encapsulated status induces further changes to the set of primitives, renaming ***eager*** to ***memoize*** and eliminating ***delay***. (See the rationale discussions for ***$lazy*** and ***memoize***, §§9.1.3, 9.1.4.)

The general rules for predicate ***equal?*** would sometimes permit it to equate two promises — if, for example, both have previously been forced and contain the same memoized value. However, if such disagreements between ***equal?*** and ***eq?*** were allowed, they might sometimes give the programmer a way to determine whether or not a promise has ever been forced. Hence the stipulation tying the behavior of ***equal?*** to that of ***eq?***.

9.1 Library features

9.1.1 `promise?`

`(promise? .` *objects*`)`

The primitive type predicate for type *promise*.

A library derivation of this combiner will be provided as part of the derivation in §9.1.3.

9.1.2 `force`

`(force` *object*`)`

If *object* is a promise, applicative ***force*** returns the value determined by *promise*; otherwise, it returns *object*.

The means used to force a promise depend on how the promise was constructed. The description of each promise-constructor specifies how to force promises constructed by that constructor.

A library derivation of this combiner will be provided as part of the derivation in §9.1.3.

9.1.3 `$lazy`

`($lazy` ⟨expression⟩`)`

Operative ***$lazy*** constructs and returns a new object of type *promise*, representing potential evaluation of ⟨expression⟩ in the dynamic environment from which ***$lazy*** was called.

When the promise is forced, if a value has not previously been determined for it, ⟨expression⟩ is evaluated in the dynamic environment of the constructing call to ***$lazy***. If, when the evaluation returns a result, a value is found to have been determined for the promise during the evaluation, the result is discarded in favor of the previously determined value; otherwise, the result is forced, and the value returned by that forcing becomes the value determined by the promise.

Forcing an undetermined lazy promise (i.e., a promise constructed by ***$lazy*** for which no value has yet been determined) may cause a sequential series of evaluations, each of which returns a promise that is forced and thus initiates the next evaluation in the series. The implementation must support series of this kind with unbounded length (i.e., unbounded number of sequential evaluations).

Note that forcing concerns the value determined by a given promise, not the result of evaluating a given expression in a given environment. Distinct promises (judged by ***eq?***, §4.2.1) represent different occasions of evaluation; so, even if they do represent evaluation of the same expression in the same environment, forcing one does not necessarily determine the value for the other, and actual evaluation will take place the first time each of them is forced.

Rationale:

In SRFI-45, promise constructor *lazy* (which is a macro) requires the programmer to guarantee that when its operand is eventually evaluated in the dynamic environment where it was constructed, the result of the evaluation will be a promise. This is necessary because if the result is an undetermined lazy promise, it should be iteratively forced (much as described here for ***$lazy***) — and there is no way to distinguish promises from other objects in the Scheme implementation, so SRFI-45 ***force*** can only *assume* that the resulting value is always a promise, and access its parts accordingly.

SRFI-45 also provides a second operative constructor of lazy promises, ***delay***, meant to be compatible with the like-named constructor in the *R5RS*. ***delay*** has a different type signature than ***lazy***: where the operand to ***lazy*** must be an expression that evaluates to a promise, the operand to ***delay*** is an expression that can evaluate to any arbitrary value; by definition, `(delay` ⟨x⟩`)` ≡ `(lazy (eager` ⟨x⟩`))`.

In Kernel, where type *promise* is an encapsulated type with its own primitive predicate (thanks, in the library derivation, to the *Encapsulations* module, §8), there is no need to prohibit the operand to ***$lazy*** from evaluating to a non-promise object; so we remove that restriction (per the core statement of design philosophy, §0.1.2). Since the operand to ***$lazy*** can then yield an arbitrary value, it has the same type signature as Scheme's ***delay***, eliminating the possible type-signature motive for a Kernel constructor ***$delay*** (compatibility with Scheme being already a non-starter).

History shows that promises are exceedingly easy to mis-implement, and the detailed description of Kernel's ***$lazy*** operative is chosen to preclude several problems that have occurred in published implementations.

When, as described above, forcing an undetermined lazy promise causes a sequential series of evaluations, support for unbounded lengths of series means, simply, iteration in constant space. The form of the requirement emulates that of the proper-tail-recursion requirement in §3.10, which is modeled in turn on that of the like requirement in the Scheme reports. Iterative forcing is not supported by the standard Scheme implementation of promises, and was the immediate motive for SRFI-45.

The rule for forcing an undetermined lazy promise is phrased in such a way that, when it does cause a sequential series of evaluations, all of the intermediate promises involved (whose forcing initiates the evaluations) are forced during the process. Thus, when the sequence of evaluations is complete, all of the promises involved have determined values, so that subsequently forcing any of them will not cause further evaluations. The version of SRFI-45 finalized in April 2004 had failed to meet this requirement, and was corrected in the official post-finalization update of August 2004.

The forcing rule specifies that, when the expression evaluation returns a result, the result is discarded if a value was determined for the promise during the evaluation. While this does curtail needless iteration (in case the result of evaluation is an undetermined lazy promise), its primary purpose is to guarantee that any given promise will always determine the same value. The *R3RS* implementation of promises didn't check for a previous determination before storing the result of evaluation, and consequently violated this invariant; the *R4RS* corrected this, but accompanied its bug-fix with test code that did not actually test for the bug, and the bug reappeared in the April version of SRFI-45. Again, the problem was corrected in the August post-finalization update.

(The rule for forcing an undetermined lazy promise is also phrased so that it explicitly requires each forced evaluation, once initiated, to complete even after a value has already been determined despite the fact that the result of that completion will certainly be discarded. This clarifies the phrasing in the *R5RS*, which so strongly emphasized determination of values that the status of the evaluations themselves was somewhat understated.)

Derivation

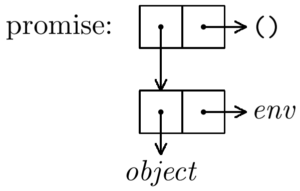
The following expression defines the ***promise?***, ***force***, ***$lazy***, and ***memoize*** combiners, using only previously defined features.

```
($provide! (promise? memoize $lazy force)

   ($define! (encapsulate promise? decapsulate)
             (make-encapsulation-type))

   ($define! memoize
      ($lambda (value)
         (encapsulate (list (cons value ())))))

   ($define! $lazy
      ($vau (exp) env
         (encapsulate (list (cons exp env)))))

   ($define! force
      ($lambda (x)
         ($if (not? (promise? x))
              x
              (force-promise (decapsulate x)))))

   ($define! force-promise
      ($lambda (x)
         ($let ((((object . env))  x))
            ($if (not? (environment? env))
                 object
                 (handle-promise-result x (eval object env))))))

   ($define! handle-promise-result
      ($lambda (x y)
         ($cond ((null? (cdar x))    ; check for earlier result
                   (caar x))
                ((not? (promise? y))
                   (set-car! (car x) y)       ;
                   (set-cdr! (car x) ())      ; memoize
```

```
          y)
       (#t
          (set-car! x (car (decapsulate y)))  ; iterate
          (force-promise x))))))
```

Rationale:

This derivation represents a promise internally as a data structure of the form



If *env* is an environment, the promise is lazily waiting to evaluate *object* in environment *env*; otherwise, *env* is nil, and the promise has memoized (thus, determines) the value *object*.

The implementation of applicative `force` only directly handles the case that the forced object is not a promise; all other cases it delegates to applicative `force-promise`, which acts on the content of the promise, not on the encapsulated object itself. `force-promise` disposes of promises already in memoized state, and advances processing of undetermined lazy promises by performing the previously suspended evaluation, but then delegates handling the result of evaluation to `handle-promise-result`.

Most of the core logic of the implementation, including the subtleties that have caused problems historically, is in `handle-promise-result`.

The first cond-clause handles the case that the promise was actually forced *during the evaluation*, so that on entry to `handle-promise-result` a result for this promise has already been memoized. Failing to make this check was the bug in the *R3RS* that was corrected by the *R4RS*. Here is (ugly) Kernel code to test this behavior.

```
($provide! (get-count p)

   ($define! count 5)

   ($define! get-count ($lambda () count))

   ($define! p
      ($let ((self  (get-current-environment)))
         ($lazy
            ($if (<=? count 0)
                 count
                 ($sequence
                    ($set! self count (- count 1))
                    (force p)
                    ($set! self count (+ count 2))
                    count))))))
```

Following these definitions,

`(get-count)`	$\Longrightarrow$	`5`
`(force p)`	$\Longrightarrow$	`0`
`(get-count)`	$\Longrightarrow$	`10`

Evaluation of `(force p)` causes *five* evaluations of the operand of ***$lazy***, nested within each other. Each evaluation of the operand returns a different result; but all five calls to ***force*** return 0, because that is the result of the innermost operand evaluation, which returns first. The fact that the other four operand evaluations *do* complete is evident from the fact that ***count*** is left at twice its original value.

The second cond-clause in ***handle-promise-result*** is the ordinary terminal case of forcing, where the result of evaluation is not a promise; in which case, the result is memoized in the promise, so that forcing this promise again later will not cause further computation.

The third cond-clause handles the iterative case of forcing, where the result of evaluation is itself a promise. By mutating the forcing promise ***x*** to reference the same second pair as the resultant promise ***y***, the clause guarantees that ***y*** will not be forced more than once. This was the subject of a subtle bug in the April version of SRFI-45: That version copied the content of ***y*** back into ***x*** and then discarded ***y***; and in doing so, it effectively created a clone of the internal state of ***y***. The clone was then forced; but the original ***y*** remained unforced, so that, if ***y*** had not been determined before cloning, its evaluation might be re-initiated later. Here is a sequence of expressions that test this behavior.

```
($define! p1 ($lazy (display "*")))
($define! p2 ($lazy p1))
(force p2)
(force p1)
```

The correct behavior on this sequence is that `(force p2)` displays an asterisk, but then `(force p1)` does not.

The third clause also enables promises to be forced iteratively: at an intermediate point during computation, when forcing the original promise (here, ***x***) has been reduced to forcing another promise (here, ***y***), the code that orchestrates the forcing process makes a tail call. In the implementation of promises in the *R5RS*, the orchestrating code failed to make a tail call, resulting in the memory leak that motivated SRFI-45. (Correcting the problem involved a fairly drastic internal rearrangement of the implementation, so one can't point to any one place in the *R5RS* implementation where a tail call should have occurred.) Here is a sequence of expressions that test iterative forcing.

```
($define! stream-filter
   ($lambda (p? s)
      ($lazy
         ($let ((v  (force s)))
            ($if (null? v)
                 v
                 ($let ((s  (stream-filter p? (cdr v))))
                    ($if (p? (car v))
```

```
                                    (cons (car v) s)
                                    s)))))))

($define! from
   ($lambda (n)
      ($lazy (cons n (from (+ n 1))))))

(force (stream-filter ($lambda (n) (=? n 10000000000))
                      (from 0)))
```

Here, a stream represents a possibly infinite list by using promises to defer elaboration of later elements of the list. (*from n*) represents the infinite list of integers ascending from n. *stream-filter* takes a predicate and a stream, and constructs a new stream representing the list of elements of the given stream that satisfy the given predicate. In an iterative implementation of promises, the final expression in this sequence should iteratively force ten billion and one promises, finally producing a cons cell whose car is ten billion, and whose cdr is a stream. (Forcing the cdr stream would never terminate, since none of the integers greater than ten billion are equal to ten billion — but shouldn't run out of memory, either, since the implementation is properly tail-recursive.)

A subtlety, BTW, of the last example is that it only works in bounded space because the stream generated by `(from 0)` is only used by *stream-filter*. Thus, at any given moment in the iterative forcing, only one segment of the stream of integers is reachable, and everything to its left is garbage. If, in place of the last expression above, we had written

```
($define! s (from 0))

(force (stream-filter ($lambda (n) (=? n 10000000000))
                      s))
```

then we would eventually run out of memory, because the entire stream of integers from zero on up would have to be retained in memory. (Even the "bounded space" version isn't really bounded, because the increasing integers themselves take space logarithmic in their magnitude; but since occupying all of memory that way would take longer than the age of the universe, we choose to overlook it.)

9.1.4 memoize

(memoize *object*)

Applicative *memoize* constructs and returns a new object of type *promise*, representing memoization of *object*. Whenever the promise is forced, it determines *object*.

A library derivation of this combiner was provided as part of the derivation in §9.1.3.

Rationale:

A basic right of first-class objects is the right to be the result of a general computation

(§B); and promises are supposed to represent the potential to do general computation. So if there were some particular type t of objects that could be the result of general computation, but *couldn't* be the result of forcing a promise, one would have to conclude either that type *promise* wasn't entirely living up to its obligations, or (by extrapolation of the usual right of first-class objects) that type t wasn't quite as first-class as it ought to be. (Moreover, the inability to return objects of type t from lazy computations would in turn weaken their right to be *stored* in lazily expanded data structures (as illustrated below) — again casting doubt on their first-class status.)

If ***$lazy*** were the only constructor of promises, then forcing a promise could never result in a promise — so that type *promise* would be either short on its obligations, or short on first-class status. Constructor ***memoize*** provides the wherewithal for a promise to determine a promise. That is, it allows a promise to be the result of lazy computation.

As a concrete example: Define a *stream* to be a promise that will determine either nil, or a pair whose cdr is a stream. One could then define the following function ***stream-ref*** that returns the element at index k of a stream (zero-indexed):

```
($define! stream-ref
   ($lambda (s k)
      ($let ((v  (force s)))
         ($if (>? k 0)
              (stream-ref (cdr v) (- k 1))
              (car v)))))
```

Applicative ***stream-ref*** immediately forces the stream to determine a result. If one wants instead to construct a *promise* to access that element of the stream later, one might (naively) write:

```
($define! lazy-stream-ref
   ($lambda (s k)
      ($lazy (stream-ref s k))))
```

When the result of ***lazy-stream-ref*** is later forced, the operand `(stream-ref s k)` is immediately evaluated, and returns whatever object was at the indexed position in the stream; and this value is then iteratively forced (per the specification for constructor ***$lazy***). If the value stored at that position in the stream is a non-promise, forcing it will have no effect on it, and it will be returned as intended. If, however, the value stored at that position in the stream is a promise, it will be forced — which may not be what was intended, and is certainly inconsistent with what would have happened if we had used ***stream-ref*** instead of ***lazy-stream-ref***. To bring the behavior of ***lazy-stream-ref*** into line with that of ***stream-ref***, one would instead write:

```
($define! lazy-stream-ref
   ($lambda (s k)
      ($lazy (memoize (stream-ref s k)))))
```

Now, forcing the result will again evaluate `(stream-ref s k)`, but the value returned is memoized, and the *memoizing promise* is iteratively forced — so that the final result of forcing the promise is the k-indexed element of the stream *even if that element is a promise.*

In SRFI-45, a constructor of promises substantially identical to *memoize* is called `eager`. Because SRFI-45 lacks a type predicate *promise?* for distinguishing promises from non-promise objects, its support for lazy promises requires the programmer to write code that always generates a promise, either eager or lazy; so *eager* is in fact the sole primitive alternative to *lazy*, which lends plausibility to the name. However, since Kernel can always distinguish promises from non-promise objects, lazy promises are commonly managed without ever constructing an eager promise (as above in *stream-ref*); and the name `eager` seems singularly inappropriate for a constructor whose net effect is to allow forcing to stop short of resolving all promises; so we choose instead to call it `memoize`, which is after all what it does internally.

In those uses of promises where *memoize is* needed, a common idiom is that the call to *memoize* is nested immediately inside a call to *$lazy*, as in the above derivation of *lazy-stream-ref*. This idiom provides the exact semantics of the sole promise constructor in standard Scheme, macro *delay*: evaluation of the operand to *delay* (the operand to *memoize* in the Kernel idiom) is postponed, but once performed its result will be returned without iterative forcing. SRFI-45, whose whole purpose is to support iterative forcing, defines *delay* by equivalence `(delay` $\langle$x$\rangle$`)` $\equiv$ `(lazy (eager` $\langle$x$\rangle$`))`. The corresponding derivation in Kernel would be

```
($define! $delay
   ($vau (x) e
      ($lazy (memoize (eval x e)))))
```

Serious consideration was given to including such an operative in Kernel. The name `$delay` was rejected on the grounds that that name would tend to somewhat obscure the relationship of what is being done to the underlying iterative-forcing mechanism (thus hiding information that the programmer should be aware of, rather than information that the programmer shouldn't be bothered by). A more explicit name, such as `$lazy-memoize`, was considered, but seems scarcely more readable that the idiomatic nested calls, so would smack of feature bloat. There would be no feature bloat in including *$lazy-memoize* if *memoize* itself were omitted from the language; but guaranteeing that iterative forcing won't pass a certain point is logically distinct from postponing computation, so it seems that *memoize* ought to be provided.

10 Keyed dynamic variables

A *keyed dynamic variable* is a device that associates a non-symbolic key (in the form of an accessor applicative) with a value depending on the dynamic extent (§7.1) in which lookup occurs.

Rationale:

Most modern Lisps are "statically scoped", meaning that the binding of a variable depends on *where* it is evaluated (the surrounding static extent, i.e., the current environment) rather than on *when* it is evaluated (the surrounding dynamic extent, i.e., the current continuation). Most dynamic information needed by an algorithm is supplied to it by means of explicit parameter passing. However, explicit parameter passing only makes sense if the information to be passed is locally relevant to both caller and callee. When dynamic information must be propagated more-or-less universally through a large dynamic extent, but is locally relevant at only a few static locations, one uses a dynamic variable.

In Common Lisp ([Ste90]), the value of a symbolic variable in an environment may be determined statically or dynamically. This approach was rejected for Kernel, on the grounds that it degrades orthogonality of function between types *environment* and *continuation* — those types being the embodiments of, respectively, static and dynamic context. While Kernel supports dynamic lookup, it preserves orthogonality by using applicative accessors instead of symbolic keys; thus, there is just one lookup rule for symbols.

The particular form of the provided support is parallel to that of the *Encapsulations* module (§8): A factory applicative (here, *make-keyed-dynamic-variable*) generates unique matching sets of tools (here, a binder and an accessor). The tools themselves are then subject to being statically bound by symbols in a local environment, and are thus subject to hiding by just the same means as any other Kernel facilities.

10.1 Primitive features

10.1.1 `make-keyed-dynamic-variable`

`(make-keyed-dynamic-variable)`

Returns a list of the form `(`b a`)`, where b and a are applicatives, as follows. Each call to *make-keyed-dynamic-variable* returns different b and a.

- b is an applicative that takes two arguments, the second of which must be a combiner. It calls its second argument with no operands (nil operand tree) in a fresh empty environment, and returns the result.

- a is an applicative that takes zero arguments. If the call to a occurs within the dynamic extent of a call to b, then a returns the value of the first argument passed to b in the smallest enclosing dynamic extent of a call to b. If the call to a is *not* within the dynamic extent of any call to b, an error is signaled.

As an illustration of how this facility might be used, here is a hypothetical derivation of the current-input-port facilities from the *Ports* module (§15, where the features hypothetically derived here are primitive).

```
($provide! (with-input-from-file get-current-input-port)

   ($define! (binder accessor)  (make-keyed-dynamic-variable))

   ($define! with-input-from-file
      ($lambda (filename appv)
```

```
         (call-with-input-file
            filename
            ($lambda (port)
               ($let ((result  (apply binder (list port appv))))
                 (close-input-port port)
                 result)))))

   ($define! get-current-input-port  accessor))
```

Using this code as written, the Kernel interpreter would have to be called through *with-input-from-file*, in order to specify the default input port. Alternatively, one might modify the code by building a compound version of *get-current-input-port* that substitutes a default if the primitive accessor fails:

```
   ($define! get-current-input-port
      ($lambda ()
         (guard-dynamic-extent
            ()
            ($lambda () (accessor))
            (list (list error-continuation
                        ($lambda (#ignore divert)
                           (apply divert
                                  default-input-port)))))))
```

Rationale:

The possibility was considered of allowing *make-keyed-dynamic-variable* to take an optional argument that would serve as a default value when the accessor is called outside of the dynamic extent of any call to the binder. It was decided that such a default value is not obviously part of the mandate of the keyed-dynamic-variable device; so, to maintain the orthogonality of the device, imposition of default values was left to the programmer, with an example provided showing how it may be accomplished.

Consideration was also given to whether a keyed-variable binder ought to call its second argument in its dynamic environment, rather than in an empty environment. However, our expectation is that the combiner argument will usually be customized for the particular binder-combination; in such a case, the combiner is constructed during the combination's argument-evaluation, at which time the dynamic environment could be deliberately captured. We therefore prefer the more hygienic, hence less accident-prone, behavior; also cf. *call/cc*, §7.2.2. The caller's dynamic environment would be preferred if general call semantics were fundamental to the operation (as with *apply*, §5.5.1, or, even more so, *map*, §5.9.1). Also, the dynamic environment would naturally be provided if, instead of providing an algorithm to the binder through a combiner argument, one made the binder operative and provided the algorithm an expression operand, similarly to *$let* (§5.10.1) — since in that case there would be no argument-evaluation during which to deliberately capture the dynamic environment.

As long as Kernel doesn't support concurrency, it would be possible to implement dynamic variables as a library facility; for each dynamic variable, one would simply maintain a global stack of values, which at any given moment would simulate the calls to b enclosing the current continuation. This approach, however, would lose miserably in the presence of concurrency, since there would then be more than one current continuation. Therefore, `make-dynamic-variable` is presented here as a primitive, in anticipation of concurrency support to be added later.

11 Keyed static variables

A *keyed static variable* is a device that binds data in an environment by a non-symbolic key, where the key is an accessor applicative.

Cf. module *Keyed dynamic variables*, §10.

Rationale:

Statically scoped symbolic variables are ideally suited as a basis for regulating access to Kernel facilities: being statically scoped, they vary mostly with location in the source-code, which is the basis on which programs are naturally organized; and symbols being publicly accessible keys, they are a natural basis for general-purpose information distribution.

A binding (of a symbol or other key) is *shadowed* when it isn't visible in an environment e despite being contained in an ancestor a of e, because another binding for the same key is located in some ancestor a' of e and, during lookup in e, a' is visited before a. The very fact that symbols are publicly accessible keys, which makes them suitable for general-purpose access regulation, also means that a symbolic binding in a might be shadowed in e by anyone with access to a'. Sometimes, though, we may want to regulate access to the *key*, so that access to a' does not necessarily confer unlimited power to bind that key in a'; this would allow the construction of tools that interact with their dynamic environments in advanced but strictly regulated ways. Some cross-cutting concerns in aspect-oriented programming may call for a tool to behave in a certain way just when it is called from within a certain static extent; or, as a more traditional example, one might want to build a Scheme-style environment-mutator, that affects pre-existing non-local bindings (cf. Kernel `$set!`, §6.8.1).

The particular form of the provided support is parallel to that of the *Encapsulations* and *Keyed dynamic variables* modules (§§8, 10): A factory applicative (here, `make-keyed-static-variable`) generates unique matching sets of tools (here, a binder and an accessor). The tools themselves are then subject to being statically bound by symbols in a local environment, and are thus subject to Kernel's general-purpose access-regulation techniques.

11.1 Primitive features

11.1.1 `make-keyed-static-variable`

`(make-keyed-static-variable)`

Returns a list of the form `(`b a`)`, where b and a are applicatives, as follows. Each call to ***make-keyed-static-variable*** returns different b and a.

- b is an applicative that takes two arguments, the second of which must be an environment. It constructs and returns a child-environment of its second argument, with initially no local bindings.

- a is an applicative that takes zero arguments. If the dynamic environment e of the call to a has an improper ancestor that was constructed by a call to b, then a returns the value of the first argument passed to b in the first such environment encountered by a depth-first traversal of the improper ancestors of e. If e has no improper ancestors constructed via b, an error is signaled.

Rationale:

Traditionally in Lisp, unique limited-access static keys are achieved via *gensyms* (short for *generated symbols*). Gensyms are generated one-off by a generator procedure, typically called `gensym`, and cannot be created from input representations (in common parlance, they are *uninterned* symbols); so they are not publicly accessible; but because they are still considered symbols by the Lisp system, they are acceptable to all the usual facilities for static bindings. (E.g., [Ste90, §10.3].) Allowing the unique/limited-access key facility to tap into the usual static-symbolic-binding facilities is, at least, somewhat parsimonious of features; but it clearly does not foster orthogonality between the features. So for the Kernel design we prefer a strictly non-symbolic approach.

The uses identified, to date, for non-symbolic static variables are related to environment mutation; but the ability to regulate shadowing of bindings —thus allowing a sort of "perfect hygiene" in tools that remotely manipulate environments— is not obviously tied to mutability; so, for the integrity of the underlying concepts involved, we prefer that the new facility be orthogonal to environment mutation.

Moreover, if the new facility were directly based on mutation, it would confer a capability that is not obviously related to non-symbolic static variables at all: the capability to determine *in an effectively non-destructive way* whether any given environment is an ancestor of another given environment. Given candidate ancestor a and candidate descendant d, one could reliably find a symbol s that isn't bound in d (using ***string->symbol*** and ***$binds?***; §§13.1.1, 6.7.1); then, if s is bound in a, a isn't an ancestor of d, otherwise mutate a by binding s in it, and look to see whether the new binding is visible in d. The process is destructive because there is then no way to get rid of the binding of s in a; however, if a key k could be generated one-off for the occasion (whether a non-symbol or a gensym), its binding in a could be left *in situ*, where it would have no further effect on computation since no part of future computation would have access to k. So the facility intended to support regulated extensions to environment manipulation would actually support unregulated (and unregulable) ancestor-testing.

The non-mutation-based alternative is to build the desired non-symbolic static binding into an environment at the time the environment is constructed — just as dynamic bindings are built into continuations at construction time in §10. As in the dynamic case, even though the binding itself refers to a fixed object, the effect of binding-mutation can still be achieved by mutating the referent. This sort of construction might have taken the form of a *$let*-like operative construct; but that approach was rejected because it is highly non-orthogonal to the primary information-hiding facility of the language — *symbolic* static variables.

To illustrate, suppose the keyed-static-variable facility is used to construct a child environment e of the surrounding current environment e', with the bound value being a compound combiner c. If the constructor for e followed the model of *$let* (§5.10.1), then the expression constructing c would be evaluated in e'; then c could capture e', as by

```
($let-keyed  ($let ((parent-env  (get-current-environment)))
                ($lambda ...))
   ...)
```

but then c would not have access to e. If the constructor for e followed the model of *$letrec* (§6.7.5), the expression constructing c would be evaluated in e, so could capture e by

```
($letrec-keyed  ($let ((child-env  (get-current-environment)))
                   ($lambda ...))
   ...)
```

but then, in order to grant c access to e', one would have to capture e' with a binding that would be visible throughout e. A general keyed-static-variable facility oughtn't have to wrestle with this access issue at all (neither to choose between e and e', nor to provide both constructors and thus bloat the feature interface); so, instead, we model the constructor on applicative *make-environment* (§4.8.4) rather than any operative of the *$let* family. One could then give c access to either or even both of e and e', without use of bindings in either environment, by

```
($let ((parent-env  (get-current-environment)))
   ($letrec ((child-env  (binder ($letrec ((c  ($lambda ...)))
                                     c)
                                  (get-current-environment))))
      child-env)))
```

(an arrangement that also, for good measure, gives c the wherewithal to call itself recursively).

12 Numbers

This section describes the required *Numbers* module and five optional modules that assume it. Module *Numbers* supports type *number* and its subtype *integer*; between them, the five optional modules define another five subtypes of *number* (including subtype *exact*, which is defined by module *Inexact* but contains just those numbers that would be supported without that module). Features are grouped into separate subsections by module, according to which module is necessary to the purpose of each feature; but, having placed the entry for a feature with one module, the behavior of that feature on numbers of subtypes from all five optional modules is specified in that one entry. For example, applicative ***sin*** is grouped with module *Real*, but the entry there includes its behavior on complex numbers.

We distinguish between a *mathematical number*, which is an abstract idea; a *Kernel number*, or simply *number*, which is a Kernel object that determines (exactly), or approximates (inexactly), a mathematical number; an external representation of a Kernel number, which is a sequence of characters that describes, or partially describes, a Kernel number (below, §12.4); and an *internal number*, which is a presumed data structure of the Kernel implementation that determines a mathematical number (below, §12.3). Note that an internal number determines a mathematical number, *not* a Kernel number; determining a Kernel number may require more than one internal number, if the Kernel number is inexact (below, §12.2).

The numerical sublanguage described in this section is designed to be largely independent of both what internal number formats are used, and how large a domain of mathematical numbers is modeled. It draws freely on its *R5RS* Scheme counterpart ([KeClRe98, §6.2]), but modifies its approach for Kernel's design policies, and incorporates two significant extensions to the *R5RS* Scheme notion of number: infinities, and bounds on inexact real numbers.

All numbers are immutable, and ***equal?*** iff ***eq?***. The *number* type is encapsulated.

Rationale:

The *R6RS* introduces concrete numeric representation issues (such as fixnums, flonums, and NaNs) into the purely abstract treatment of numbers from the previous Scheme reports ([Sp+07]).[13] One reason suggested for doing so is that, while attempting to be independent of internal number formats, the abstract Scheme treatment of numbers allowed so many variant behaviors that it left itself open to rabid non-portability. The Kernel design agrees with this objection (on grounds that non-portable number support degrades the programmer's ability to use numbers freely, contrary to first-class-ness Guideline *G1a* of §0.1.2); but the *R6RS* alternative multiplies the number of numeric features, violating Kernel's core design philosophy (top of §0.1.2). There is a profound difference between constraining low-level behavior, which promotes portability; and inflating the programmer interface with low-level details, which, besides the feature bloat itself, promotes fragmentation of the language semantics and thereby degrades first-class-ness. Kernel provides an abstract programmer interface to numbers, but seeks to avoid the somewhat *ad hoc* character of the abstract *R5RS* interface by moderating its interface through explicit design

[13] The *R5RS* abstract treatment of numbers dates back to the *R2RS*, [Cl85].

principles (a lesser echo of the overall Kernel design strategy). Kernel admits flexibility for problem-domain-driven variations in internal number formats, but seeks to avoid rampant non-portability by disallowing variations that would be merely stylistic.

The Kernel design purpose of infinities is to bound sets of finite real numbers that have no finite bounds. For example, the *max* of an empty list of real numbers —which, for smooth behavior of *max*, must be no greater than the *max* of any non-empty list of real numbers— is negative infinity; the *min*, positive infinity (§12.5.13). An important case is bounds on inexact real numbers: if no finite upper or lower bound can be be placed on the set of mathematical numbers that might be approximated by an inexact real, we can still assign it upper bound positive infinity, and lower bound negative infinity.

When an arithmetic operation involves an infinity, either among its arguments or as its result, it should be understood as a limit, as with the arctangent of positive infinity, which is exactly pi over two (though an implementation of module *Real* doesn't have to support exact modeling of that mathematical number). This principle also implies that certain operations are *not* defined, such as division by zero, which has no determinate primary value since it has different one-sided limits as zero is approached from above or from below (or, worse, from an arbitrary direction in the complex plane; re `/`, see §12.8.2).

The Kernel design purpose of inexact numbers is to support not only approximations of mathematical numbers (as in *R5RS* Scheme), but also bounds on those approximations. See the rationale discussion of §12.2.

Encapsulating the *number* type does not interfere with support for extended subtypes of number. For example, if a number subtype *quaternion* were added, encapsulation would prohibit any standard feature from returning a non-complex quaternion when given complex arguments, because its behavior on complex arguments is described in the report; but its behavior on non-complex quaternions is not described in the report, so given any non-complex quaternion argument it could return a non-complex quaternion.

12.1 Kinds of mathematical numbers

The only mathematical numbers modeled by the *Numbers* module are the integers (positive, zero, and negative), and positive and negative infinity (countable). Successively larger domains of mathematical numbers are modeled via optional modules:

Rational — admits finite ratios of integers.
Real — admits finite real numbers that might not be rational.
Complex — admits sums and products of numbers with complex i.

Module *Real* assumes module *Rational.* Module *Complex* assumes module *Real.*

Each of these modules supports a subtype of type *number*, with the same name as the module. Subtype *rational* is a supertype of *integer*, *real* is a supertype of *rational*, and *complex* is a supertype of *real.*

Rationale:

Integers and infinities are included in required module *Numbers* because they are needed for the core modules, e.g. `length` (§6.3.1). Module *Complex* assumes module

Real because both rectangular and polar accessors are provided, so that trigonometry is involved even if only Gaussian integers are constructed. It would seem a daunting task to implement module *Real* without module *Inexact*, but in case someone has a reason to do so, the report doesn't preclude it, i.e., module *Real* doesn't assume module *Inexact*.

12.2 Inexactness

An exact Kernel number models some particular mathematical number, and is represented in the Kernel implementation by an internal number. However, sometimes there may be no way for an internal number to capture a mathematical number that the client wants to reason about, either because the intended mathematical number cannot be represented by an internal number (as with exclusively rational internal number formats confronted with an irrational mathematical number), or because the client doesn't know exactly which mathematical number to model (as with an experimental measurement, or the result of arithmetic on numbers that were already inexact). Optional module *Inexact* associates to each real Kernel number a mathematical upper and lower bound, between which the intended mathematical number is certain to fall *if* the intended mathematical number exists at all. It also partitions type *number* into subtypes *exact* and *inexact*, each of which intersects, but is not a sub- or supertype of, the other subtypes of *number* described in the report (*integer*, *rational*, etc.).

An inexact real number is represented internally by an internal real *upper bound*; an internal real *lower bound*; optional internal real *primary value*; and optional *robust* tag. The intended mathematical number, if it exists, is certain to be no less than the lower bound, and no greater than the upper bound. If the number is robust (i.e., is tagged so), the intended mathematical number is certain to exist. The primary value is a best guess at the intended mathematical number, within the bounded interval. If there is no primary value, the "best guess" is that the intended mathematical number does not exist; but if it does exist it must still fall within the bounded interval. An inexact number with no primary value cannot be robust. An inexact real number has lower bound no greater than its upper bound. When an inexact "real" number is created with lower bound greater than its upper bound, the number is said to be *undefined*; there is only one undefined number (i.e., all undefined numbers are ***eq?***), which does not belong to type *real*, nor even to type *complex*; the only subtype of *number* that it belongs to is *inexact*. If an arithmetic operation with any undefined argument returns a result, that result must be undefined.

Because types *inexact* and *exact* are mutually exclusive, an inexact real is not an exact real, even if the inexact real is robust and its upper and lower bounds are the same as its primary value. However, an exact real is considered to be robust, and to have itself as its primary value and upper and lower bounds. When an arithmetic operation in module *Numbers* or *Rational* is given only exact arguments, the result of the operation must be exact.

When any real arithmetic operation is performed ("real" in the sense of real arguments and real result),

- The primary value of the result is based solely on the primary values of the arguments, taking into account the internal number formats of those primary values. (Therefore, the primary value of the result cannot depend on the bounds, robustness, nor exactness, of the arguments.) If any of the arguments doesn't have a primary value, the result cannot have a primary value.

  If the result doesn't have a primary value (and even if all of the arguments do have primary values), an error is or is not signaled depending on the current value of the strict-arithmetic keyed dynamic variable (§12.6.6).

- The closed interval between the upper and lower bounds of the result must include every possible result of the operation from any mathematical real primary values within the bounds of the arguments. For example, if an integer two, with upper bound three and lower bound one, is added to itself, the result is four with an upper bound of at least six and a lower bound of at most two.

- If some choice of primary argument values within the argument bounds would result in failure of the operation, the result cannot be robust. For example, if exact 1 is divided by an inexact 1 with lower bound less than zero, the result has primary value 1, upper bound positive infinity, lower bound negative infinity, and is not robust.

When any arithmetic operation is performed that involves complex numbers, the above rules are applied by treating separately each real component of each non-real complex number (in whatever internal coordinate system is used for each complex number; see §12.3.1, below). When an operation takes a complex argument, it behaves as a specialized operation taking the two real components separately; and when an operation returns a complex result, it behaves as if it were two operations, each returning one real number. (Moreover, if an implementation supports any other kind of hypercomplex number, again the above rules are applied by treating real components separately.)

An implementation can fully support module *Inexact* without making any effort to maintain finite bounds or robustness on inexact real numbers, because the requirements of the module —excepting exact operations on exact arguments— only prevent inexact numbers from claiming greater precision than is actually certain. The implementation might simply take all inexact real numbers to be non-robust with upper bound positive infinity and lower bound negative infinity, and describe each inexact real number by a single internal real number and a tag indicating that it is inexact (since the implementation is required to distinguish between exact and inexact numbers). Optional module *Narrow inexact* lets the client instruct the implementation to maintain bounding and robustness information, through the narrow-arithmetic keyed dynamic variable (§12.7.1). Module *Narrow inexact* assumes module *Inexact*.

If an operation on numeric arguments depends on their values to determine behavior other than a numeric result, and the operation is given a numeric argument with no primary value, the operation signals an error. (As earlier, a complex argument is treated here as if it were two real arguments.) The numeric comparison predicates (§§12.5.2, 12.5.3) are paradigmatic of such operations. Type predicates notably do not belong to this class of operations because, although some of them may use a primary value when present, they do not *depend* on its presence: absence of a primary value is at most a cause for returning false.

Rationale:

When a Scheme arithmetic operation produces an inexact result, that result conveys no certain knowledge at all. However, if one has certain bounds on the arguments of the operation, and knows the internal number formats being used, one can deduce certain upper and lower bounds on the result. The implementation knows the internal number formats, while the client should worry about them as little as possible; so we provide means for the implementation to deduce the bounds and provide that information to the client, empowering the client to accommodate differences between internal number formats while maintaining the abstract treatment of numbers. Since this kind of deduction by the implementation is a substantial effort that may be irrelevant to some applications, we separate out the means by which the client requests it to a dependent optional module, *Narrow inexact*.

In bounding an inexact arithmetic result, the bounded interval is required to include results from all choices of *mathematical* primary values of the arguments, rather than merely all *supported internal* primary values of the arguments. Requiring coverage of all choices of *real* primary values prohibits the bounds from taking advantage of non-support for optional modules (e.g., even if only integers are supported, arbitrary real choices must be covered by the interval). Requiring coverage of all choices of *mathematical*, rather than internal, real primary values prohibits the bounds from taking advantage of limitations on support for internal number formats.

This revision of the report leaves flexibility for implementations of module *Narrow inexact* to decide for themselves just how tightly to bound the results of narrow-arithmetic operations. Flexibility here entails non-portability of narrow-inexact bounding behavior; but the numerical analysis issues involved appear to be quite deep (especially when contemplating complex arithmetic), and were not studied in depth for this revision; so, for the moment, we leave implementations free to address these deep issues for themselves, rather than impose ill-considered constraints now and then have to repair it later after forcing implementations to choose between non-conformant behavior and poor behavior.

When the strict-arithmetic keyed dynamic variable is false (§12.6.6), and the numeric result of an operation depends on the value of a numeric argument that has no primary value, the operation has the option of returning a number with no primary value instead of signaling an error. However, non-numeric types generally do not have states signifying 'probably, but not necessarily, non-existent'; so when a non-numeric result depends on the value of a numeric argument with no primary value, there is no non-strict alternative to an error signal. In the case of predicates on numbers, such as numeric comparisons (`=?`, §12.5.2, etc.), one might be tempted to handle lack of primary argument value by simply returning false; however, the use of such predicates on inexact numbers is already, by nature, a *guess* based on best information available, and lack of a primary value means that the best information available indicates that the question has no answer (cf. *mu* in [Ra03]). Framing the situation in purely practical terms, the results of numeric predicates are used to determine branching of control flow (the design purpose of type *boolean*), and it would be merely a coincidence if, in a particular case, false causes a more useful no-primary-value behavior than true; so returning either boolean value on no-primary-value is essentially a random guess, and accident avoidance (*G3* of §0.1.2) favors signaling a dynamic type error.

12.3 Internal numbers

Internal numbers are, as stated earlier, *presumed* data structures of the Kernel implementation. Because implementations only have to match the abstract behavior described in this report, internal numbers needn't be realized as concrete data structures, and even if they are, the format of the concrete data structures is not constrained by the report. However, under various circumstances implementations are constrained to behave *as if* internal numbers were concrete data structures with particular formats, and these constraints are the subject of the current section (§12.3).

12.3.1 Complex numbers

An inexact complex number is represented internally by a choice of coordinate system, and real Kernel numbers for the coordinates. The coordinate system must be one of those used by the available primitive constructors (here, `make-rectangular` and `make-polar`, §12.10), although a number returned by one of these constructors does not have to be represented internally in the coordinate system of that constructor. Accessors are provided to extract coordinates under each of these coordinate systems; and if that system is used internally to represent the accessed complex number, the accessor returns the internally stored coordinate.

An exact finite complex number is always internally represented in rectangular coordinates.

An exact infinite complex number (i.e., an exact complex number with infinite magnitude) is represented internally either in rectangular coordinates, or in point-projective form. Rectangular coordinates can be used for the internal representation only if one of the rectangular coordinates is finite. In point-projective form, an exact finite non-zero complex number (the *point*) is specified, which determines the direction from zero of the represented infinite complex; that is, the represented number is exact positive infinity times the point. The point doesn't have to be stored internally in any normal form (it only has to be exact, finite, and non-zero), but any given implementation must normalize the point when `write`ing the infinity, so as not to reveal anything about `eq?` numbers that couldn't be determined without `write` (per §3.6).

Rationale:

Requiring that internal inexact complex numbers correspond with the constructors provides a degree of simple reproducibility to inexact complex arithmetic.

Requiring that an exact finite complex number be internally rectangular means that `real-part` and `imag-part` must return an exact result when given an exact finite complex argument. The purpose of this constraint is to disambiguate the behavior of predicate `exact?` without causing turbulence between exact finite arithmetic and internal real formats. From §12.2, any exact arithmetic operation —i.e., an operation in module *Number* or *Rational*— given only exact arguments must return an exact result; but we don't mean to require any implementation to support exact irrational reals. Consider the complex number $z = 1^i$, where the radix 1 and exponent i are both exact. When we construct z, we know exactly which mathematical number we are modeling: the complex number with magnitude exactly one, and angle exactly one radian. No irrational reals are needed to represent this number exactly in polar coordinates, even though both of its rectangular coordinates are irrational. However, adding exact one to z gives a complex number $z + 1$ whose rectangular and polar coordinates are all irrational. (Of course, an implementation that *does* support exact irrationals won't have a problem with that.)

An exact complex infinity is only internally representable in rectangular coordinates if one of its rectangular coordinates is finite. Representing an exact rectangular complex infinity in polar coordinates would actually lose data, since differences in the finite rectangular coordinate change the angle infinitesimally. Representing a non-rectangular complex infinity (i.e., one whose rectangular coordinates are both infinite) in polar coordinates would force implementations to support exact irrationals (below, §12.3.2).

12.3.2 Exact real numbers

Every implementation of Kernel must support the two exact real infinity objects, positive and negative. Every implementation of Kernel must support exact integers of arbitrary size, and, when supporting module *Rational*, exact finite ratios of integers of arbitrary size (subject only to running out of memory, which is an implementation restriction violation admitted by §1.3.3). Implementations may also provide, at their discretion, internal representations for various exact irrational numbers; but note that the first-class exact Kernel numbers are required to be closed under features of modules *Numbers* and *Rational* (above, §12.2).

As long as only exact numbers occur, the only observable consequence of internal number format is that the exact numbers involved are representable at all.

When discussing a rational internal representation in terms of numerator and denominator, they are assumed to be integers, with least positive denominator. (E.g., rational zero has numerator zero and denominator one).

Rationale:

The report is intended to specify the behavior of each operation sufficiently that, in principle, given exact arguments it has just one possible exact result (or is undefined). The operation usually isn't required to *return* this exact result, the largest class of exceptions being features of modules *Numbers* and *Rational*; and the exact result might not be internally representable by the implementation (if it is irrational, or a complex with an irrational component).

If an implementation supports module *Complex*, and also chooses to support any exact irrationals, it must support a class of exact irrationals that is closed, not only under arithmetic of modules *Numbers* and *Rational*, but also under these operations on rectangular complex numbers with exact components (as discussed above, §12.3.1).

12.3.3 Inexact real numbers

In the internal representation of an inexact real number, the internal numbers — bounds and optional primary value— may use a *restricted format*. A restricted internal real format has finite precision (number of significant digits in the mantissa), may have bounded exponent (of the radix, by which to multiply the mantissa), and is equipped with definitions for all arithmetic operations. If any of the internal numbers in an inexact real number use a restricted format, all of the finite internal numbers for that inexact real must use the same restricted format. When multiple restricted formats are involved in an operation, any result should use the restricted format among the arguments with the greatest precision, and within that precision, the largest range of exponents. If all internal numbers in an operation use the same restricted format, the operation uses its behavior for that format; if multiple formats are involved, and the result format will be restricted, when feasible all internal numbers should be converted to that format before performing the operation.

A numeric *overflow* occurs when the primary value of an inexact result would exceed the largest magnitude representable by its restricted format (either in the positive or in the negative direction); *underflow*, when the primary value would have non-zero magnitude smaller than the smallest non-zero magnitude representable. When the strict-arithmetic keyed dynamic variable is cleared (§12.6.6), numeric overflow and underflow are not errors; in that case, overflow produces an infinite primary value (positive or negative), and underflow produces a zero primary value.

12.4 External representations of numbers

An external representation of an inexact real number describes its primary value, possibly including some information about internal format (which can affect arithmetic behavior on inexact numbers), and indicates that it is inexact, but does not indicate whether it is robust, nor what its bounds are. An external representation of a complex number indicates that it is complex and contains external representations of two real numbers from which the complex number may be constructed, either by rectangular or by polar construction (and indicates which). An external representation of an exact number completely determines the number, so that `write`ing an exact number z and then `read`ing what was written will produce an object `eq?` to z.

The external representation of an undefined number is `#undefined`. All other rules for externally representing numbers pertain only to defined numbers.

An external representation of a real number consists of optional radix and/or exactness prefixes, optional sign (`+` or `-`), and magnitude. The radix prefixes are `#b` (binary), `#o` (octal), `#d` (decimal), and `#x` (hexadecimal); the default is decimal. The exactness prefixes are `#e` (exact) and `#i` (inexact); by default, the number is inexact iff the magnitude representation uses floating point. If both kinds of prefixes are used, they may occur in either order. The magnitude is either `infinity`; an unsigned integer (nonempty sequence of digits); a ratio of unsigned integers (two unsigned integers with a `/` between, of which the second is non-zero); or a floating point representation. If the magnitude is `infinity`, there must be an exactness prefix and a sign, and no radix prefix. Floating point representation can only be used with decimal radix; it consists of nonempty integer part, point (`.`), nonempty fraction part, and optional exponent part. The optional exponent part consists of an exponent letter, and an (optionally signed) integer indicating a power of ten by which to multiply the magnitude. The choice of exponent letter makes no difference in what mathematical number is indicated by the external representation, but does indicate internal representation precision. Exponent letters `s`, `f`, `d`, `l` indicate preference for successively higher internal precision — *short*, *float*, *double*, *long*. When ***`read`***ing an inexact real number, exponent letter `e` accepts the default internal precision, which must be at least double. When ***`write`***ing an inexact real number, exponent letter `e` may be used for the default internal precision, and must be used for any internal number format not indicated by any of the other exponent letters. Float and double must provide, respectively, at least as much precision as IEEE 32-bit and 64-bit floating point standards [IE85].

For example, `#i#xa/c` represents an inexact number using hexadecimal notation, with signed magnitude positive five sixths (ten over twelve). `-3.5l-2` represents an inexact number using decimal notation, with signed magnitude negative thirty five thousandths, and requested long precision (which must be at least IEEE 64-bit floating point).

When ***`read`***ing an external representation of an inexact real, the bounds on the resulting inexact number are chosen in accordance with the narrow-arithmetic keyed dynamic variable (§12.7.1).

A polar external representation of a complex number consists of external representations of magnitude and angle separated by an `@`; the angle must not be infinite. A rectangular external representation of a complex number consists of an optional external representation of the real part (defaults to exact zero if omitted), required sign (`+` or `-`), optional unsigned external representation of the imaginary part (defaults to exact one if omitted), and `i`; either the real part or the imaginary part may be infinite, but not both.

For example, `2.3@8/3` represents a complex number with magnitude inexact 2.3 and angle exactly eight thirds. `-0.7i` represents a complex number with real part exactly zero and imaginary part inexact negative seven tenths. `1.0+i` represents a complex number with real part inexact positive one and imaginary part exact positive one.

A point-projective external representation of a complex infinity consists of required exactness prefix, required sign `+`, `infinity`, `*`, and angle brackets `<>` delimiting an external representation of the point. The external representation of the point must include an imaginary part or angle (`i` or `@`). If the exactness prefix of the whole is `#e`, both real components of the point must be exact; if the exactness prefix of the whole is `#i`, the real components of the point may be exact or inexact.

For example, `#e+infinity*<-1+i>` represents an exact complex infinity with angle $-\frac{\pi}{4}$ radians, while `#i+infinity*<1@0>` represents inexact positive real infinity.

When an external representation is read of a number belonging to an optional numeric module (such as a non-real complex number), any resulting number must be either of the specified type, or an inexact approximation of the represented number (as allowed for the result of an arithmetic operation, above in §12.3). Alternatively, if the optional module is not supported, an error may be signaled; and if the optional module is excluded, an error *must* be signaled. Reading an external representation of a number belonging to an unsupported optional numeric module does not constitute violation of an implementation restriction.

Rationale:

The traditional restriction of floating point to decimal radix cannot easily be dismissed, because the most usual exponent letter, `e`, is a hexadecimal digit.

12.5 Number features

12.5.1 `number?`, `finite?`, `integer?`

(number? . *objects*)
(finite? . *numbers*)
(integer? . *objects*)

Applicative ***number?*** is the primitive type predicate for type *number*.

Applicative ***finite?*** is a predicate that requires *numbers* to be a list of numbers, and returns true unless one or more of its arguments is either a real infinity, or an inexact real with no primary value, or a complex number whose rectangular components are not all finite. If an argument is an inexact real with no primary value, or a complex number any of whose components has no primary value, or undefined, an error is signaled.

Applicative ***integer?*** is the primitive type predicate for number subtype *integer*. A real infinity is not an integer. A rational is an integer iff its denominator is one. An inexact real is an integer iff it has an integer primary value. A complex is an integer iff its real part is an integer and its imaginary part has primary value zero and upper and lower bounds zero.

Rationale:

Since predicate `finite?` does not take arbitrary objects as arguments, it is not a type predicate; therefore, when any of its arguments has no primary value, an error is signaled (per §12.2). Predicates `number?` and `integer?` are type predicates, so they handle no-primary-value without error.

The criterion for predicate `integer?` is superficial on the real part of a complex number (only concerns its primary value), but fundamental on the imaginary part (also concerns its bounds).

The real part of the criterion could have required, instead, certainty that every possible value of the intended mathematical number is an integer. However, given only primary value and upper and lower bounds, the only ways to be certain of this would be (1) the implementation only supports integers, so that every finite number computed is sure to be an integer; or (2) the upper and lower bounds are identical to the primary value. We don't want the behavior of the predicate on inexact numbers to change when additional modules are supported; and we do want to support inexact integers. Another approach would be to maintain an additional *integer* tag on each inexact real number, akin to the *robust* tag, indicating that the intended mathematical number is known to be an integer; but we prefer not to further complicate the inexact type in this way.

The imaginary part of the criterion could have been made superficial, only requiring that the primary value be zero. However, we intend `integer?` to imply `real?`; see the rationale discussion for `real?`, §12.9.1.

12.5.2 `=?`

`(=? . ` *numbers*`)`

Applicative `=?` is a predicate that returns true iff all its arguments are numerically equal to each other. If any of its arguments has no primary value, or has any real component with no primary value, an error is signaled.

Two real numbers are numerically equal iff their primary values model the same mathematical number. Two complex numbers are numerically equal iff their corresponding components, both rectangular and polar, are numerically equal — that is, their real parts are numerically equal, their imaginary parts are numerically equal, their magnitudes are numerically equal, and their angles are numerically equal.

Rationale:

The conditions and rationale for signaling an error on no-primary-value were discussed in §12.2.

When two same-signed real infinities are compared, they should be found numerically equal. Because an infinity is viewed *in arithmetic operations* as a limit, one might suppose that the result of comparing two same-signed infinities is indeterminate, there being no way to decide which limit approached infinity “faster”. However, as set out in the rationale discussion at the top of §12, an infinity is treated as a limit *only* to guide its arithmetic use. The design purpose of an infinity is to bound sets of finite reals — and in that capacity, two different infinities with the same sign bound the same finite reals, so they are numerically equal.

12.5.3 <?, <=?, >=?, >?

(<? . *reals*)
(<=? . *reals*)
(>=? . *reals*)
(>? . *reals*)

Each of these applicatives is a predicate that returns true iff every two consecutive elements of *reals* have primary values in the order indicated by the name of the applicative. If any element of *reals* has no primary value, an error is signaled.

For example,

```
(<=?)                    ⟹  #t
(<=? 1)                  ⟹  #t
(<=? 1 3 7 15)           ⟹  #t
(<=? 1 7 3 15)           ⟹  #f
```

On a cyclic list of real arguments, `<?` or `>?` will always return false, while `<=?` or `>=?` might still return true if all elements of the cycle are numerically equal to each other.

Rationale:

The conditions and rationale for signaling an error on no-primary-value were discussed in §12.2.

In principle, all of these predicates could be derived as library features, given `=?` and a binary version of `<?` (using a similar technique to the library derivations of `eq?` and `equal?`, §§6.5.1, 6.6.1).

12.5.4 +

(+ . *numbers*)

Applicative `+` returns the sum of the elements of *numbers*.

If *numbers* is empty, the sum of its elements is exact zero.

If a positive infinity is added to a negative infinity, the result has no primary value. If a complex number with magnitude infinity is added to another complex number with magnitude infinity, and they don’t have the same angle, the result has no primary value.

If all the elements of a cycle are zero, the sum of the cycle is zero. If the acyclic sum of the elements of a cycle (i.e., the sum of an acyclic list containing just those elements) is non-zero, the sum of the cycle is positive infinity times the acyclic sum of the elements. If the acyclic sum of the elements of a cycle is zero, but some of the elements of the cycle are non-zero, the sum of the cycle has no primary value.

Rationale:

We view the sum of a cycle as the limit sum of an infinite series. When the acyclic sum of the elements of the cycle is zero, but some of the elements are non-zero, the sum of the series isn't convergent.

12.5.5 `*`

`(*` . *numbers*`)`

Applicative `*` returns the product of the elements of *numbers*.

If *numbers* is empty, the product of its elements is exact one.

If an infinity is multiplied by zero, the result has no primary value. If a complex number with magnitude infinity is multiplied by zero, the result has no primary value. If a non-real complex number is multiplied by infinity, the result has no primary value.

If the acyclic product of the elements of a cycle is real greater than one, the product of the cycle is positive infinity. If all the elements of a cycle are positive one, the product of the cycle is positive one. If the acyclic product of the elements of a cycle is positive one, but some of the elements of the cycle are not positive one, the product of the cycle has no primary value. If the acyclic product of the elements of a cycle has magnitude less than one, the product of the cycle is zero. If the acyclic product of the elements of a cycle has magnitude greater than or equal to one, and is not positive real, the product of the cycle has no primary value.

Rationale:

We view the product a cycle as the limit product of an infinite series.

The product of positive infinity with a finite non-zero complex number does not have a unique limit as the finite factor approaches zero, so we do not arbitrarily assign a primary value to the product of positive infinity with zero.

12.5.6 `-`

`(-` *number* . *numbers*`)`

numbers should be a nonempty list of numbers. Applicative `-` returns the sum of *number* with the negation of the sum of *numbers*.

Rationale:

The smooth behavior of applicative `-` on a single argument (i.e., if *numbers* is the empty list) would be to return that argument unchanged. In Scheme, however, `-` on a single argument performs a completely different operation on the single argument (negation). If we wanted a negation applicative in Kernel, we would give it a different name; but we also forgo the extension of smooth behavior to a single argument, to avoid a behavioral inconsistency with Scheme that could trip up Scheme programmers.

12.5.7 `zero?`

(`zero?` . *numbers*)

Applicative ***`zero?`*** is a predicate that returns true iff every element of *numbers* is zero. For this purpose, a real number is zero if its primary value is zero, and a complex number is zero if its real part, imaginary part, and magnitude are all zero. If any element of *numbers* has no primary value, or has any real component with no primary value, an error is signaled.

Rationale:

The conditions and rationale for signaling an error on no-primary-value were discussed in §12.2.

12.5.8 `div`, `mod`, `div-and-mod`

(`div` *real1* *real2*)
(`mod` *real1* *real2*)
(`div-and-mod` *real1* *real2*)

For all three applicatives, if *real1* is infinite or *real2* is zero, an error is signaled.

Let n be the greatest integer such that $real2 \times n \leq real1$. Applicative ***`div`*** returns n. Applicative ***`mod`*** returns $real1 - (real2 \times n)$. Applicative ***`div-and-mod`*** returns a freshly allocated list of length two, whose first element is n and whose second element is $real1 - (real2 \times n)$.

Rationale:

The *R5RS* supports integer division through procedures ***`quotient remainder`*** and ***`modulo`***. There are some difficulties using those three procedures as the principal general tools for integer division, due to which the *R6RS* provides its primary integer-division support through functions div, mod, div_0, and mod_0. Here we prefer the more well-behaved div etc. to the earlier procedures; the *R6RS* discusses its rationale for these functions in a separate document ([Sp+07]).

12.5.9 `div0`, `mod0`, `div0-and-mod0`

(div0 *real1 real2*)
(mod0 *real1 real2*)
(div0-and-mod0 *real1 real2*)

For all three applicatives, if *real1* is infinite or *real2* is zero, an error is signaled.

Let n be the greatest integer such that $real2 \times n \leq real1 + \left|\frac{real2}{2}\right|$. Applicative ***div0*** returns n. Applicative ***mod0*** returns $real1 - (real2 \times n)$. Applicative ***div0-and-mod0*** returns a freshly allocated list of length two, whose first element is n and whose second element is $real1 - (real2 \times n)$.

Rationale:

See the rationale discussion in §12.5.8.

12.5.10 `positive?`, `negative?`

(positive? . *reals*)
(negative? . *reals*)

Applicative ***positive?*** is a predicate that returns true iff every element of *reals* is greater than zero. Applicative ***negative?*** is a predicate that returns true iff every element of *reals* is less than zero. If any argument to either applicative has no primary value, or has any real component with no primary value, an error is signaled.

Rationale:

The conditions and rationale for signaling an error on no-primary-value were discussed in §12.2.

12.5.11 `odd?`, `even?`

(odd? . *integers*)
(even? . *integers*)

Applicative ***odd?*** is a predicate that returns true iff every element of *integers* is odd. Applicative ***even?*** is a predicate that returns true iff every element of *integers* is even. If any argument to either applicative has no primary value, or has any real component with no primary value, an error is signaled.

Rationale:

The conditions and rationale for signaling an error on no-primary-value were discussed in §12.2.

12.5.12 `abs`

(`abs` *real*)

Applicative *`abs`* returns the nonnegative real number with the same magnitude as *real*; that is, if *real* is nonnegative it returns *real*, otherwise it returns the negation of *real*.

12.5.13 `max`, `min`

(`max` . *reals*)
(`min` . *reals*)

If *reals* is nil, applicative *`max`* returns exact negative infinity, and applicative *`min`* returns exact positive infinity. If *reals* is non-nil, applicative *`max`* returns the largest number in *reals*, and applicative *`min`* returns the smallest number in *reals*.

Rationale:
The behaviors on nil argument list preserve equivalences

```
(max h . t)  ≡  (max h (max . t))
(min h . t)  ≡  (min h (min . t))
```

12.5.14 `lcm`, `gcd`

(`lcm` . *impints*)
(`gcd` . *impints*)

impints should be a list of *improper integers*, that is, real numbers each of which is either an integer or an infinity.

Applicative *`lcm`* returns the smallest positive improper integer that is an improper-integer multiple of every element of *impints* (that is, smallest $n \geq 1$ such that for every argument n_k there exists n'_k with $n_k \times n'_k = n$). If any of the arguments is zero, the result of *`lcm`* has no primary value. According to these rules, *`lcm`* with nil argument list returns 1, and *`lcm`* with any infinite argument returns positive infinity.

Applicative *`gcd`* returns the largest positive improper integer such that every element of *reals* is an improper-integer multiple of it (that is, largest $n \geq 1$ such that for every argument n_k there exists n'_k with $n \times n'_k = n_k$). *`gcd`* with nil argument list returns exact positive infinity. If *`gcd`* is called with one or more arguments, and at least one of the arguments is zero, but none of the arguments is a non-zero finite integer, its result has no primary value. According to these rules, if *`gcd`* is called with at least one finite non-zero argument, its result is the same as if all zero and infinite arguments were deleted.

Rationale:

Positive infinity is an improper-integer multiple of every non-zero improper integer (by multiplying that non-zero improper integer by an infinity of like sign). Zero is an improper-integer multiple of every finite integer (by multiplying that finite integer by zero). Therefore, the gcd of any non-zero finite integer n with zero is $abs(n)$, and the gcd of any non-zero finite integer n with any real infinity is $abs(n)$. However, the gcd of zero with an infinity is indeterminate: it can't be positive infinity, because there isn't anything that multiplied by zero gives infinity, but it can't be any particular finite integer either, because *every* finite positive integer can be multiplied by zero to give zero, and by an infinity to give positive infinity. Therefore, if ***gcd*** has a zero argument, but doesn't have a non-zero finite argument, its result has no primary value.

The behaviors on nil argument list preserve equivalences

```
(lcm h . t)  ≡  (lcm h (lcm . t))
(gcd h . t)  ≡  (gcd h (gcd . t))
```

provided, in the latter case, that `(gcd . t)` has a primary value (which excludes the case that t contains a zero, but h was the only non-zero finite argument).

In the *R5RS* (which does not support infinities), ***gcd*** given any empty argument list returns 0. This behavior does, in fact, preserve the above equivalence, and could be extended to a behavior that encompasses infinite arguments without leaving certain cases indeterminate (the ones with a zero argument but no non-zero finite argument). However, the behavior specified here is deemed more uniform. Here, unlike the *R5RS*, `(gcd h . t)` is never greater than `(gcd . t)`; and the entire behavior of the applicative here is captured by a simple statement of the form "the largest ... such that ...", whereas a statement of the *R5RS* behavior is intrinsically more elaborate.

12.6 Inexact features

12.6.1 `exact?`, `inexact?`, `robust?`, `undefined?`

(exact? . *numbers*)
(inexact? . *numbers*)
(robust? . *numbers*)
(undefined? . *numbers*)

Applicative ***exact?*** is a predicate that returns true iff every element of *numbers* is either an exact real number, or a complex number whose rectangular components are all exact. Applicative ***inexact?*** is a predicate that returns true iff every element of *numbers* is either an inexact real number, or a complex number at least one of whose rectangular components is inexact. Applicative ***robust?*** is a predicate that returns true iff every element of *numbers* is either a robust real number, or a complex number all of whose rectangular components are robust. Applicative ***undefined?*** is a predicate that returns true iff every element of *numbers* is undefined.

Rationale:

Although none of these predicates are type predicates (because they don't take arbitrary objects as arguments), none of them depend on their arguments having primary values, either, so they don't signal an error on no-primary-value (per §12.2).

12.6.2 get-real-internal-bounds, get-real-exact-bounds

(get-real-internal-bounds *real*)
(get-real-exact-bounds *real*)

Applicative ***get-real-internal-bounds*** returns a freshly allocated list of reals (x_1 x_2), where the primary value of x_1 is the lower bound of *real*, using the same internal representation as the primary value of *real*, and the primary value of x_2 is the upper bound of *real*, using the same internal representation as the primary value of *real*. The x_k are inexact iff *real* is inexact. The x_k are robust (i.e., tagged if the implementation supports such), and the bounds of each x_k are only required to contain its primary value (i.e., the implementation is allowed to make the bounds equal to the primary value).

Applicative ***get-real-exact-bounds*** returns a freshly allocated list of exact reals (x_1 x_2), where x_1 is not greater than the lower bound of *real*, and x_2 is not less than the upper bound of *real*.

Rationale:

Not all internal numbers are necessarily representable by an exact number, so when converting the bounds of *real* to exact numbers, some error may be introduced. ***get-real-internal-bounds*** avoids introducing this error, while ***get-real-exact-bounds*** guarantees that any error introduced will increase the bounded interval rather than decreasing it. (In practice, one will usually prefer to keep the internal format, simply because converting to exact numbers would be pointlessly expensive — but that is proscribed as a design motive, by *G5* of §0.1.2).

12.6.3 get-real-internal-primary, get-real-exact-primary

(get-real-internal-primary *real*)
(get-real-exact-primary *real*)

If *real* is exact, both applicatives return *real*. If *real* has no primary value, both applicatives signal an error.

If *real* is inexact with a primary value, applicative ***get-real-internal-primary*** returns a real number x_0 whose primary value is the same as, and has the same internal format as, the primary value of *real*. x_0 is robust, and its bounds are only required to contain its primary value.

If *real* is inexact with a primary value, applicative ***get-real-exact-primary*** returns an exact real number x_0 within the exact bounds that would be returned for *real* by applicative ***get-real-exact-bounds*** (§12.6.2). Preferably, x_0 should be as close to the primary value of *real* as the implementation can reasonably arrange. If the implementation does not support any exact real that reasonably approximates *real*, an error may be signaled.

Rationale:

There should be a facility for separating the primary value of *real* from its bounds without changing the format of *real*, hence ***get-real-internal-primary***. See also the rationale discussion in §12.6.2.

When *real* has no primary value, signaling an error preserves the expectations that ***get-real-exact-primary*** always returns an exact real, and that ***get-real-internal-primary*** always returns a real whose primary value is within its bounds.

These applicatives could be widened to support arbitrary number arguments, rather than just reals. For the moment, only real arguments are supported, pending deeper analysis of the complex-representation issues involved.

12.6.4 `make-inexact`

(`make-inexact` *real1 real2 real3*)

Applicative ***make-inexact*** returns an inexact real number, as follows.

If *real2* is inexact, the result has the same primary value as *real2*; and if *real2* has no primary value, the result has no primary value. The result has the same robustness as *real2*. If possible, the result uses the same internal representation as *real2*.

If *real2* is exact, the primary value of the result is as close to *real2* as the implementation can reasonably arrange; overflow and underflow are handled as described in §12.3.3.

The lower bound of the result is no greater than the lower bound of *real1*, the primary value of *real2*, and the primary value of the result. The upper bound of the result is no less than the upper bound of *real3*, the primary value of *real2*, and the primary value of the result.

Rationale:

If this applicative used the general rule for internal representations for arithmetic operations (§12.3.3), the programmer would have to regulate the internal representations of all three arguments in order to regulate the internal representation of the result. Adopting the internal representation of *real2* simplifies the programmer's regulation task.

A seemingly more straightforward behavior for this applicative would base the bounds of the result on the primary values of *real1* and *real3*, rather than the lower bound of *real1* and upper of *real3*. However, this was deemed potentially error-prone, since any error accumulated in calculating *real1* and *real3* probably should be applied to the number constructed from them — and in the presumably less usual case that it shouldn't, the programmer can stipulate the primary values of *real1* and *real3* via applicative ***get-real-internal-primary*** (§12.6.3).

It would be possible to derive ***get-real-internal-primary*** from this applicative, by

```
($define! get-real-internal-primary
   ($lambda (x)
      (make-inexact #e+infinity x #e-infinity))
```

12.6.5 `real->inexact`, `real->exact`

(`real->inexact` *real*)
(`real->exact` *real*)

Applicative ***`real->exact`*** behaves just as ***`get-real-exact-primary`***, §12.6.3.

If *real* is inexact, applicative ***`real->inexact`*** returns *real*.

If *real* is exact, applicative ***`real->inexact`*** returns an inexact real x_0 such that *real* would be a permissible result of passing x_0 to ***`real->exact`***. If the implementation does not support any such x_0, an error may be signaled. Otherwise, x_0 is robust, and its bounds are only required to contain its primary value and *real*.

12.6.6 `with-strict-arithmetic`, `get-strict-arithmetic?`

(`with-strict-arithmetic` *boolean combiner*)
(`get-strict-arithmetic?`)

These applicatives are the binder and accessor of the strict-arithmetic keyed dynamic variable; keyed dynamic variables were described in §10. When this keyed variable is true, various survivable but dubious arithmetic events signal an error — notably, operation results with no primary value (§12.2), and over- and underflows (§12.3.3).

12.7 Narrow inexact features

12.7.1 `with-narrow-arithmetic`, `get-narrow-arithmetic?`

(`with-narrow-arithmetic` *boolean combiner*)
(`get-narrow-arithmetic?`)

These applicatives are the binder and accessor of the narrow-arithmetic keyed dynamic variable; keyed dynamic variables were described in §10. When this keyed variable is true, the implementation is advised to maintain the most restrictive bounding and robustness information it (correctly) can. The only constraint on such information, besides correctness, is that it cannot be *less* restrictive when the variable is true than when the variable is false.

Rationale:
See the rationale discussion in §12.2.

12.8 Rational features

12.8.1 `rational?`

`(rational? .` *objects*`)`

Applicative ***`rational?`*** is the primitive type predicate for number subtype *rational*. A real infinity is not a rational. An inexact real is a rational iff its primary value is a ratio of integers. A complex is a rational iff its real part is a rational and its imaginary part has primary value zero and upper and lower bounds zero.

Rationale:
See the rationale discussion for primitive type predicate ***`integer?`***, §12.5.1.

12.8.2 `/`

`(/` *number* `.` *numbers*`)`

numbers should be a nonempty list of numbers. Applicative `/` returns *number* divided by the product of *numbers*. If the product of *numbers* is zero, an error is signaled. If *number* is infinite and the product of *numbers* is infinite, an error is signaled.

Rationale:
On requiring at least two arguments, see the rationale in §12.5.6.
When a non-zero number is divided by zero, one might expect the result to be infinite; but to understand this result as a limit (per the rationale discussion at the top of §12), one would have to know which direction the zero denominator is approached from, which is not knowable from the arguments to applicative `/`. Infinity divided by infinity is similarly indeterminate. A case could be made that zero divided by zero should be zero, but the essence of that position is that zero divided by anything else is zero, and one could as well argue that anything else divided by zero is indeterminate; since provisions against dividing by zero are already routine, the latter position was judged less anomalous.

12.8.3 `numerator`, `denominator`

`(numerator` *rational*`)`
`(denominator` *rational*`)`

These applicatives return the numerator and denominator of *rational*, in least terms (i.e., chosen for the least positive denominator).

Note that if *rational* is inexact, and either of its bounds is not its primary value, the denominator has upper bound positive infinity, and the numerator must have at least one infinite bound (two infinite bounds if the bounds of *rational* allow values of both signs).

12.8.4 `floor`, `ceiling`, `truncate`, `round`

(floor *real*)
(ceiling *real*)
(truncate *real*)
(round *real*)

Applicative ***floor*** returns the largest integer not greater than *real*.

Applicative ***ceiling*** returns the smallest integer not less than *real*.

Applicative ***truncate*** returns the integer closest to *real* whose absolute value is not greater than that of *real*.

Applicative ***round*** returns the closest integer to *real*, rounding to even when *real* is halfway between two integers.

Rationale:

The behavior of ***round*** is as specified in the *R5RS*, which in turn was chosen for consistency with the IEEE floating point standard.

12.8.5 `rationalize`, `simplest-rational`

(rationalize *real1 real2*)
(simplest-rational *real1 real2*)

A rational number r_1 is *simpler* than another rational r_2 if $r_1 = p_1/q_1$ and $r_2 = p_2/q_2$, both in lowest terms, and $|p_1| \leq |p_2|$ and $|q_1| \leq |q_2|$. Thus 3/5 is simpler than 4/7. Not all rationals are comparable in this ordering, as for example 2/7 and 3/5. However, any interval (that contains rational numbers) contains a rational number that is simpler than every other rational number in that interval. For example, between 2/7 and 3/5 lies the simpler 2/5. Note that $0 = 0/1$ is simpler than any other rational (so that one never has to choose between p/q and $-p/q$).

For applicative ***simplest-rational***, let x_0 be the simplest rational mathematically not less than the primary value of *real1* and not greater than the primary value of *real2*. If no such x_0 exists (because the primary value of *real1* is greater, or because the primary values of the arguments are equal and irrational), or if either argument does not have a primary value, an error is signaled.

For applicative ***rationalize***, let x_0 be the simplest rational mathematical number within the interval bounded by the primary value of *real1* plus and minus the primary value of *real2*. If no such x_0 exists (because the primary value of *real1* is irrational and the primary value *real2* is zero), or if either argument does not have a primary value, an error is signaled.

If *real1* and *real2* are exact, the applicative (whichever it is) returns exact x_0. If one or both of *real1* and *real2* are inexact, the applicative returns an inexact rational approximating x_0 (as by ***real->inexact***, §12.6.5). Note that an inexact result returned is not necessarily bounded by the primary values of the arguments; but the result is an approximation of x_0, which is so bounded, and the bounds of the result include x_0.

Rationale:

The results of these operations are, in principle, distributed arbitrarily over the entire specified interval, with no preference given to any 'central' part of the interval. Therefore, if the bounds of the arguments were included in the interval used for the operation, the primary values of the arguments would be completely irrelevant to the operation; and unintendedly large intervals would tend to have exaggerated consequences (contra *G3* of §0.1.2). Bounds are also a focal point for quantification of differences between implementations, whereas this operation seems not to be intensionally related to such differences.

The *R5RS* provides only ***rationalize***, not ***simplest-rational***. However, providing only the outer bounds of the interval is a likely tactic since this is the form of result provided by the get-real-bounds applicatives (§12.6.2). Converting available arguments from one of these two interfaces to the other carries a risk of imprecision due to fixed-precision internal formats; so instead we provide both interfaces directly to the programmer, deferring the task of dealing with such imprecisions to the implementation. (The name `simplest-rational` is borrowed from MIT/GNU Scheme, [MitGnu].)

12.9 Real features

12.9.1 `real?`

(`real?` . *objects*)

Applicative ***real?*** is the primitive type predicate for number subtype *real*. A complex is a real iff its imaginary part has primary value zero and upper and lower bounds zero. (Moreover, if an implementation supports any other kind of hypercomplex number, a hypercomplex number is real iff all its imaginary parts have zero primary values and bounds.)

Rationale:

If we're told that a number is real, we shouldn't have to think about imaginaries at all; we shouldn't have to ask about bounds on the imaginary part of a complex number, and we certainly shouldn't have to know what non-complex kinds of numbers might be supported by the implementation.

12.9.2 `exp`, `log`

(`exp` *number*)
(`log` *number*)

12.9.3 `sin`, `cos`, `tan`

(`sin` *number*)
(`cos` *number*)
(`tan` *number*)

12.9.4 `asin`, `acos`, `atan`

(`asin` *number*)
(`acos` *number*)
(`atan` *number*)
(`atan` *number1 number2*)

12.9.5 `sqrt`

(`sqrt` *number*)

12.9.6 `expt`

(`expt` *number1 number2*)

12.10 Complex features

12.10.1 `complex?`

(`complex? .` *objects*)

12.10.2 `make-rectangular`, `real-part`, `imag-part`

(`make-rectangular` *real1 real2*)
(`real-part` *complex*)
(`imag-part` *complex*)

12.10.3 `make-polar`, `magnitude`, `angle`

(`make-polar` *real1 real2*)
(`magnitude` *number*)
(`angle` *complex*)

13 Strings

13.1 Primitive features

13.1.1 `string->symbol`

13.2 Library features

14 Characters

14.1 Primitive features

14.2 Library features

15 Ports

A port is an object that mediates character-based input from a source or character-based output to a destination. In the former case, the port is an *input port*, in the latter case, an *output port*.

Although ports are not considered immutable (i.e., a mutation of a port would not have to be an error per §3.8), none of the operations on ports described in this section constitute mutation. Ports are ***equal?*** iff ***eq?***. The *port* type is encapsulated.

Rationale:

Because the file i/o facilities of this module do not specify what constitutes a valid "filename", the facilities can be used to support a very wide variety of character-stream-based i/o.

Ports are the archetype of Kernel objects with administrative state (rationale, §3.8). Objects of this kind can be a serious obstacle to Guideline *G3* (§0.1.2), because once such an object enters a "dead" state, the object becomes literally an accident waiting to happen. In the current case, a reference to a closed port is simply a means by which the programmer could possibly cause an i/o error by trying to use it. Therefore, when the internal state of a data type is administrative, a dominant concern in the design of the type support is to minimize the programmer's dependence on explicit references to objects of the type.

The port-based i/o tools in *R5RS* Scheme are already well suited to this design goal, and so Kernel adopts Scheme's port tools substantially intact. The opening and closing of ports can be handled in three ways:

1. The safest, and therefore preferred, way to open a port is via *with-input-from-file*/*with-output-to-file*. Since the opened port is accessed implicitly within the dynamic extent of the call, and is automatically closed on normal return, it is possible for the programmer to use a port this way without ever once explicitly referencing it.

2. For somewhat more general i/o operations, ***call-with-input-file/call-with-output-file*** provides an explicit reference to the opened port, but still allows the programmer to readily contain the reference within a dynamic extent in which it is valid (as it is closed on return).
3. The programmer can choose to take on the full burden of accident-prevention in exchange for complete generality —in which the lifetime of a port does not coincide with any dynamic extent— by means of ***open-input-file/open-output-file*** (which will also require explicit calls to ***close-input-file/close-output-file***).

Because type *port* has only administrative state, not data state, none of the general operations on ports are considered mutation (per the rationale in §3.8). Conceivably, a subtype of port could have some sort of associated data, so that some internal state of the subtype would be data state, and modifying that would be mutation.

15.1 Primitive features

15.1.1 `port?`

`(port? . `*objects*`)`

The primitive type predicate for type *port*.

15.1.2 `input-port?`, `output-port?`

`(input-port? . `*objects*`)`

Applicative ***input-port?*** is a predicate that returns true unless one or more of its arguments is not an input port. Applicative ***output-port?*** is a predicate that returns true unless one or more of its arguments is not an output port.

Every port must be admitted by at least one of these two predicates.

Rationale:

A port has no purpose if it can't be used for input or output. Although this report does not allow a single port to be used for both, it would not be unreasonable for an extension to want to support that, and therefore *input port* and *output port* were not made primitive types (would would have made them mutually exclusive).

15.1.3 `with-input-from-file`, `with-output-to-file`

15.1.4 `get-current-input-port`, `get-current-output-port`

15.1.5 `open-input-file`, `open-output-file`

15.1.6 `close-input-file`, `close-output-file`

15.1.7 `read`

15.1.8 `write`

15.2 Library features

15.2.1 `call-with-input-file`, `call-with-output-file`

15.2.2 `load`

15.2.3 `get-module`

(`get-module` *string*)
(`get-module` *string* *environment*)

When the first syntax is used, applicative ***`get-module`*** creates a fresh standard environment (§3.2); opens for input (§15.1.5) a file named *string*; ***`read`***s objects (§15.1.7) from the file until the end of the file is reached; evaluates those objects consecutively in the created environment; and, lastly, returns the created environment.

When the second syntax is used, the freshly created standard environment is augmented, prior to evaluating read expressions, by binding symbol `module-parameters` to the optional argument, *environment*.

Rationale:

In any nontrivial programming language, some details of the meaning of a program module can only be determined by studying other program modules on which it depends. In an extensible language, this effect may be so pronounced that a dependent program module may be effectively meaningless without careful study of all its predecessors. [Sta75] identified this as a basic practical limitation on how far an extensible language can be extended (using, of course, the extension technology of the time). If all source files of a Kernel program were processed in a single environment, as via applicative ***`load`*** (§15.2.2), the effect would be highly pronounced, since Kernel has no special forms whose meanings would necessarily remain stable. Applicative ***`get-module`*** provides stability, thus promoting usability Guideline *G3* of §0.1.2, both by starting each module in a fresh standard environment, and by giving each module explicit control over access to its predecessors through use of first-class environments.

The possibility was considered of allowing a single environment to be returned from multiple calls to ***`get-module`*** on the same file, provided no changes had been made in the interim to that file, nor to any other files that it in turn had ***`get-module`***d. This was ultimately rejected as violating the simplicity language goal. A secondary objection was that, since detection of relevant source-file changes, and even the definition of what constitutes a change or when two files are really "the same file", would be platform-dependent, there would be no reliable way to predict whether two different calls to ***`get-module`*** would produce ***`eq?`*** environments, potentially promoting accidents (contra *G3* of §0.1.2).

Because each call to ***`get-module`*** reevaluates the contents of the source file(s), caution is needed when a module constructs facilities that use object identity for access control — such as, notably, encapsulated types (§8) or keyed dynamic/static variables (§§10, 11). A closely related issue is the ability to parameterize the loading process, which is the purpose of optional argument *environment*. Using an environment for this lends a degree of uniformity to the parameter-passing protocol; minimizes intrusion of representation format (versus alternative formats, such as alists, whose structural mechanics are explicitly visible); and provides a high degree of flexibility since Kernel core facilities are strongly oriented toward fluent handling of bindings and first-class environments. Requiring *environment* to be specified as an explicit argument, rather than using the dynamic environment of the call to *get-module*, reduces the likelihood of *accidental* exposure of the dynamic environment. Providing *environment* to the loading module through a single binding with a standard name, rather than using any arrangement that would make its bindings directly visible in the environment of the loading module, preserves the stability afforded by loading in a standard environment. Construction of suitable input-parameter environments is facilitated by *$bindings->environment* (§6.7.10); modules being loaded can check for input parameters using *$binds?* (§6.7.1); parameters when present might be extracted using *$import!* (§6.8.3), or *$remote-eval* (§6.7.9).

There is room, within the functionality of applicative *get-module*, for a sort of "compilation", in which a source file has associated with it preprocessed information about how to rapidly construct an appropriate environment for return by *get-module*.

Derivation

The following expression defines *get-module* using only previous defined features.

```
($define! get-module
   ($lambda (filename . opt)
      ($let ((env  (make-kernel-standard-environment)))
         ($cond ((pair? opt)
                    ($set! env module-parameters (car opt))))
         (eval (list load filename) env)
         env)))
```

16 Formal syntax and semantics

This section provides formal descriptions of what has already been described informally in previous sections.

16.1 Formal syntax

This subsection formally defines the syntax of Kernel. The material is presented, with some redundancy, in four parts: §16.1.1 gives an algorithm for dividing a sequence of characters into lexemes (in words, as a grammar, and as a finite-state machine). §16.1.2 gives an algorithm for classifying lexemes by token type. §16.1.3 presents a grammar for building tokens from sequences of characters (which formally subsumes the contents of §§16.1.1–16.1.2). Finally, §16.1.4 presents a grammar for building expressions from sequences of tokens.

The grammars use an extended BNF. All spaces in the grammars are for legibility. Case is insignificant; for example, `#x1A` and `#X1a` are equivalent. ⟨empty⟩ stands for the empty string. The following extensions to BNF are used to make the description more concise: $\langle\text{thing}\rangle^*$ means zero or more occurrences of ⟨thing⟩; and $\langle\text{thing}\rangle^+$ means one or more occurrences of ⟨thing⟩.

Rationale:

Kernel expression syntax is much simpler than Scheme's, because there is no need to go any further than simple expressions — no special forms, no syntax transformers, and no syntactic sugar for quotation or quasiquotation. However, Kernel lexical structure is nearly identical to that of Scheme because, for the safety of Scheme programmers, it includes Scheme lexemes such as `'` and `,@` that are not actually legal tokens in Kernel. This common Scheme/Kernel lexical structure is fairly simple, but the simplicity is not evident from a pure extended-BNF grammar for tokens. We therefore present the lexical structure in a different form that brings out its internal simplicity, so that future revisions of the report will be better able to preserve the simplicity.

The heart of the simple lexical structure of Scheme/Kernel is that a lexer can group the input character stream into lexemes using only a few distinct classes of characters, with a lookahead of only one character,[14] and without having to know anything about the detailed syntax of particular kinds of tokens (§16.1.1). These lexemes can then be easily classified by token type by looking at a few leading characters, still without worrying about whether the entire lexeme is a syntactically correct token (§16.1.2). The detailed syntax of tokens is only needed after the token class is already known.

16.1.1 Lexemes

Here we describe an algorithm for dividing a sequence of characters into lexemes. The algorithm doesn't classify lexemes by type, not even by which lexemes are legal and which are illegal; that is accomplished by a second algorithm, below in §16.1.2.

There are a few special kinds of Kernel lexemes that have well-defined beginnings or ends, but most lexemes sprawl, each encompassing as many characters as it can in both directions until stopped by a character that cannot belong to it. Lexemes of the sprawling kind are called *middle lexemes.* The special kinds of lexemes come in two varieties: *left lexemes* and *right lexemes.* Simply put, a left lexeme is one that can occur immediately to the left of a middle lexeme, while a right lexeme is one that can occur immediately to the right of a middle lexeme.

There are only three kinds of right lexemes: left paren, right paren, and string literal (delimited by double quotes). All three kinds of right lexemes are also left

[14] Lookahead of one character means that, when scanning a lexeme from left to right, we can process characters one at a time, without worrying about what, if anything, comes after a given character until after we've decided whether to include it in the current lexeme.

lexemes, because it's perfectly clear when the right lexeme has ended, so if the following character is non-atmosphere (not a space, tab, newline, etc.), it must be the beginning of a new lexeme.

In addition to the three kinds of right lexemes, there are also five other left lexemes:

```
`  '  #(  ,  ,@
```

(In words: backquote, quote, hash-left-paren, comma, and comma-at.) Kernel defines these to be lexemes to avoid accidents that could otherwise result from lexical incompatibilities with Scheme; but the language syntax doesn't use any of them, so they will all be classified as illegal in §16.1.2.

⋮

16.1.2 Classes of lexemes

16.1.3 Tokens

16.1.4 Expressions

16.2 Formal semantics

[The planned strategy is an operational semantics and associated calculus. On the theoretical admissibility of this approach, see Appendix C.]

A Evolution of Kernel

A.1 *R5RS* to *R⁻¹RK*

Principal technical additions from *R5RS* Scheme to *R⁻¹RK* Kernel include:

- First-class operatives and their treatment, especially ***$vau*** (which allows their general construction and is thus necessary to their first-class status, per Appendix B), and ***unwrap*** (which ties the semantics of applicatives to that of operatives).

- First-class environments and their treatment, especially the selective limitations on their mutation (mitigating hygiene concerns).

- Generalized formal parameter trees for binding constructs (effectively supporting simultaneous return of multiple values without Scheme's extra-functional constructs ***values*** and ***call-with-values***; see §4.9.1).

- Object `#inert` (to be returned in lieu of useful information).

- First-class treatment of cyclic structures (§3.9).

- Exception handling via entry/exit guards (§7).

- Real infinities, and exact upper and lower bounds on inexact reals.

- Encapsulated data types (§8).

- Dynamic variables via ***make-dynamic-variable-key*** (§10).

Principal deletions from Scheme include quotation/quasiquotation, hygienic macros, and ***values*** / ***call-with-values***.

A.2 $R^{-1}RK$ partial drafts

Initial construction of the report is a multi-year process. This is to be expected, given the nature of its design philosophy (§0.1.1); but the content of the report is meant to be shared with the programming community at large (§0.2), so it was decided in March 2005 that the existing incomplete draft of the $R^{-1}RK$ should be made public, and updated from time to time thereafter.

The incompleteness of the draft was most prominently because portions simply hadn't been written yet, only planned in rough outline. However, some existing material required further rationale discussion, which could sometimes lead to amendments; the development of omitted materials might also occasionally induce changes to previous materials; and everything —especially, all source code fragments— will have to be systematically rechecked for inconsistencies after the material has stabilized.

The public drafts of the $R^{-1}RK$ are:

- 22 March 2005.

- 21 April 2005. Improved (therefore altered) the interface to interceptors in continuation guards, §7.2.5. Added keyed static variables.

- 5 November 2005. Minor edits (fixed typos, improved phrasing).

- 15 February 2006. Some content on numbers, formal syntax. Corrected ***eq?***-ness demands on ***read*** / ***write*** (§3.6). Bug fixes to ***length***, ***filter***, ***reduce***, ***append!***.

- 13 March 2006. Extensive content on numbers.

- 22 August 2006. Some content on numbers. Bug fixes to ***list****, ***reduce***.

- 9 September 2007. Updates to *First-class objects* Appendix B. Bug fixes to example under ***make-keyed-static-variable***, derivations of ***$sequence*** and compound ***$vau***. Improved phrasing, §1.3.3; expanded rationales, §4.9.1, §5.1.1. More content on numbers (including no-primary-value errors and point-projective form).

- 21 September 2009. Rationales, §1.3.7, §2, §3.8, §4.9.1, §5.4.1. Expanded index entry for "external representation". Some content on ports, notably ***get-module***. Bug fix to derivation of ***append!***, and small emendation to its description. Bug fix to derivations of library ***eq?*** and ***equal?***. Bug fix to derivation of ***guard-dynamic-extent***. Replaced applicative ***bound?*** with operative ***$binds?***. Added operatives ***$remote-eval***, ***$bindings->environment***, ***$import!***.

- 29 October 2009. Added Appendix C.

A.3 $R^{-1}RK$ to *R0RK*

The $R^{-1}RK$ may never have a single authoritatively final version, since "Revised^{-1}" *means* "preliminary". However, there are some weaknesses in the preliminary language that, while they should not be allowed to continue into the *R0RK*, probably will not be given priority in the design process until all the sections of the report have at least been drafted. The most prominent such weakness is that cyclic structures composed of pairs do not currently have an external representation (§4.6; it seems likely that SRFI-38, [Di03], will be adopted). Another weakness that may fall into the same class is the absence of any technical characterization of data passed to the error continuation (§7.2.7).

A.4 Beyond *R0RK*

A major development that lies beyond the scope of the *R0RK* is:

- Elaboration of an error continuation hierarchy. This needs to be done, especially to the extent of completely specifying errors signaled by combiners in the report.

The issue of separate compilation units is under review. Some (possibly) more extensive additions are also under consideration:

- Concurrency. The currently envisioned approach would seek to minimize the number of new concepts introduced, by basing synchronization on the notion of reachability that is already used in Lisp for garbage collection.

- Theorems. Traditional type systems contain a fixed inference engine that fights, and inevitably loses to, the halting problem. An alternative approach is being developed in which inference engines are the responsibility of the programmer, who already confronts the halting problem as a normal part of programming.

- External i/o. Graphical user interfaces are an obvious omission. It is not at all clear that any GUI strategy devised to date has the kind of elegance that Kernel aspires to; but as of this writing, the problem has not been subjected to a direct assault, so no alternative strategy for Kernel is yet under development.

B First-class objects

As a matter of principle, all entities directly involved in the Kernel evaluator algorithm must be first-class objects (*G1a* of §0.1.2). This subsection discusses the concept of *first-class object* in depth, starting from its general sense and then applying it to Kernel.

The concept is due to Christopher Strachey.[15] In [Str00, §3.5.1], he describes first-class objects in Algol:

> In Algol a real number may appear in an expression or be assigned to a variable, and either may appear as an actual parameter in a procedure call. A procedure, on the other hand, may only appear in another procedure call either as the operator (the most common case) or as one of the actual parameters. There are no other expressions involving procedures or whose results are procedures. Thus in a sense procedures in Algol are second class citizens — they always have to appear in person and can never be represented by a variable or expression (except in the case of a formal parameter)

Strachey's detailed enumeration of rights and privileges is specific to Algol, but the concept generalizes naturally. In any given programming language, first-class objects may be freely manipulated, according to certain broad classes of manipulations that are characteristic of the language; second-class objects are those whose general manipulations are hemmed in with special restrictions. The rights and privileges of first-class objects in a language are usually noticed only by their absence when working with second-class objects. For example, based on what *cannot* be done with primitive data types in Java 1.4, one might well adopt the position that first-class objects in Java must belong to class `Object`.[16]

Abelson and Sussman [AbSu96, §1.3.4] recommend the following four criteria as necessary, though by implication not sufficient, for first-class objects in Scheme.

1. They may be named by variables.
2. They may be arguments to procedures.
3. They may be returned by procedures.

[15] Christopher Strachey originally evolved the notion of first-class value from W.V. Quine's principle *To be is to be the value of a variable* ([La00]).

Quine's principle is a criterion for what a statement assumes to exist (*not* a criterion for what actually does exist). He proposes reformulating any given statement using quantified variables and predicates; in the reformulation, whenever a variable must have a value bound to it, some such value that satisfies the statement is assumed to exist. Thus the sentence "Pegasus does not exist," in which "Pegasus" appears to be the name of something, becomes "$\neg(\exists x)$(is-Pegasus(x))", in which x is not bound to anything and so no existence is assumed. ([Qu61].)

[16] This could also be taken as an example of why one shouldn't compromise on the elegance of a language design.

4. They may be components of data structures.

A significant omission from this list may be illustrated by the following thought experiment. Suppose Scheme were slightly modified, by granting these four properties to, say, the built-in special-form combiners ***and*** and ***or***. There would then be two new "first-class" objects in the Scheme programmer's repertory. However, there would be little to gain by it. The new objects would remain anomalies, because they would always be the only two "first-class" examples of a larger type, with its own peculiar abstract properties; in fact, conceptually they seem to belong to some kind of *infinite* domain, most of whose values are unavailable in Scheme, and the rest of which are (in this particular scenario) second-class under criteria 1–4. Although one might then choose to *call* those two objects "first-class", their arbitrary peculiarity would never allow them to, so to speak, fully assimilate into the society of Scheme objects. To correct this problem, here is an additional criterion for first-class objects[17] (still without claiming sufficiency, of course):

5. There are no arbitrary restrictions on the set of objects, in the given object's naturally arising value domain, that satisfy criteria 1–4.

This criterion requires first-class status to be judged collectively for an entire type, rather than individually for each particular value. In applying the criterion, each object type is to be judged according to the standard of the naturally arising value domain to which it corresponds; either all of the values in the domain are first-class, or none of them are.

The new criterion depends inherently on the *intent* of the data type: whether a restriction is arbitrary depends on what abstract domain the type is being used to represent. For example, a 32-bit unsigned integer representation may be first-class if its purpose in the program is to model the nonnegative integers modulo 2^{32}, while the same concrete data type used to represent the countably infinite domain of nonnegative integers fails the criterion. (Any strictly first-class representation of the integers would have to be infinite-precision, i.e., Lisp-style bignums.)

Under Criterion 5, the Scheme special-form combiners can't be made first-class in isolation; a conceptually complete domain of such creatures must be identified, and given first-class status all at once. Thus, type *operative* would not be first-class without its constructor, ***$vau***.

Although Scheme has been remarkable in its support for first-class procedures and first-class continuations, operatives are not the only second-class objects centrally involved in the *R5RS* evaluator algorithm: Scheme *environments* are also second-class. This follows, again, from Criterion 5. The naturally arising domain of Scheme environments is infinite; but in the *R5RS*, only three environments out of this infinite domain can be explicitly denoted (only two if optional procedure ***interaction-environment*** is omitted) [KeClRe98, §6.5], so only these three (or two) satisfy Criteria 1–4.

[17] A similar criterion for first-class objects was recommended by [Gu91].

Other kinds of second-class objects in Scheme, less basic to its evaluator algorithm, include multiple-value return sequences (see rationale discussion of §4.9.1), and cyclic list and tree structures (see rationale discussion of §3.9).

C De-trivializing the theory of fexprs

The use of a term-reduction calculus to model any Lisp with fexprs has, in recent years, acquired a folk reputation for theoretical inadmissibility, due to an overgeneralization based on the (quite correct) central theoretical result of Mitchell Wand's 1998 paper "The Theory of Fexprs is Trivial" ([Wa98]). The paper is precise in stating its formal result:

> We provide a very simple model of a reflective facility based on the pure λ-calculus, and we show that its theory of contextual equivalence is trivial: two terms in the language are contextually equivalent iff they are α-congruent.

Throughout the paper's lucid formal development of this result, no generalization is made beyond this precise statement. However, the informal concluding remarks of the paper do contain the overgeneralization (foreshadowed by the paper's title). They first observe that the provided model, with its trivial theory, is apparently adding no more new capability than is strictly necessary to qualify as supporting fexprs — not really a problematic observation, although the paper notably does not distinguish, when making this point, between properties (such as capabilities) of the modeled Lisp and properties of the model. Then, however, the remarks draw from this observation an inference that the trivialization of theory cannot be avoided without failing to support fexprs. This inference is apparently based on the assumption that the trivialization of theory is *caused by* the new capability.

In fact, it is fairly easy to demonstrate (with the advantage of hindsight, i.e., once one has seen it done) that the observed trivialization of theory is an artifact of properties of the model, not properties of the modeled Lisp.

Any modification of λ-calculus to model fexprs must somehow distinguish between those subterms that are to be evaluated, and those that are not. Wand's treatment supposes that the only difference between these two is that subterms that are not to be evaluated are marked by an evaluation-suppressing context (specifically, they occur within an operand of a fexpr).[18] It also supposes that "evaluating a term" is just another name for reducing it; hence, calculus reduction of a subterm is itself suspended when the subterm occurs within an operand to a fexpr. Because there is a context that suppresses reduction, two terms cannot be interchangeable *in all*

[18]This evaluation-suppressing approach may be seen as an echo of the `quote` operator, which was introduced in [McC60] to distinguish data S-expressions from control S-expressions.

contexts unless they can be recognized as identical without performing any reduction on them — in other words, contextual equivalence is then a trivial relation.

The trivialization can be avoided if one disrupts these suppositions: rather than marking subterms not to be evaluated by means of an evaluation-suppressing context, mark subterms that *are* to be evaluated, by means of an evaluation-*inducing* context; and introduce different syntax for a cons-cell than for an application, so that evaluation becomes clearly separate from β-reduction. To illustrate the general method, here are a series of straightforward successive alterations to pure λ-calculus, each of which evidently preserves its equational strength, at the end of which sequence the result is a pure calculus, called $\wr_x$-calculus.[19]

To start with, here is our rendition of λ-calculus:

λ-calculus.

Syntax:

$$
\begin{array}{rcl}
c & \in & \mathit{Constants} \\
x & \in & \mathit{Variables} \\
T & ::= & c \mid x \mid (TT) \mid (\lambda x.T) \qquad \text{(Terms)}\,.
\end{array}
$$

Schemata:

$$(\lambda x.T_1)T_2 \;\longrightarrow\; T_1[x \leftarrow T_2]\,.$$

Auxiliary function $T_1[x \leftarrow T_2]$ performs hygienic substitution of term T_2 for variable x in term T_1; its definition is a standard exercise, which we omit here.

This is the traditional, call-by-name λ-calculus, which we prefer for the demonstration since it underscores that fexpr support does not in itself force such reduction-ordering decisions. If we wanted the call-by-name λ_v-calculus (due to Gordon Plotkin, [Plo75]), we would restrict the operand in the schema to be a value — for suitable definition of *value*; choosing the correct definition of *value* was key to Plotkin's construction, and further care would be required when extending the definition to cover the additional syntax needed for $\wr_x$-calculus.

Our first alteration to the calculus is a straightforward replacement of its superficial syntax: we replace the combination notation "(TT)" with "[combine T T]", and the function abstraction notation "$(\lambda x.T)$" with "$\langle \wr x.T \rangle$".

Syntax:

$$
\begin{array}{rcl}
c & \in & \mathit{Constants} \\
x & \in & \mathit{Variables} \\
T & ::= & c \mid x \mid [\text{combine } T\ T] \mid \langle \wr x.T \rangle \qquad \text{(Terms)}\,.
\end{array}
$$

Schemata:

$$[\text{combine } \langle \wr x.T_1 \rangle\ T_2] \;\longrightarrow\; T_1[x \leftarrow T_2]\,.$$

The equational strength of the calculus is obviously unaffected; the formal equations of λ-calculus now use the superficially altered syntax.

[19]The choice of glyph $\wr$ for vau in this mathematical context is discussed in an appendix of [Sh09].

Next, we add primitive syntax for cons-cells.

$$T \ ::= \ c \mid x \mid \texttt{(}T \ \texttt{.} \ T\texttt{)} \mid [\text{combine } T\ T] \mid \langle \text{\textrevepsilon} x.T \rangle \qquad (\text{Terms})\,.$$

We assume that the empty list is a constant, $\texttt{()} \in \mathit{Constants}$; and we will freely use the usual abbreviated list notation, $\texttt{(}T \ \texttt{.} \ \texttt{(}\cdots\texttt{))} \equiv \texttt{(}T \ \cdots\texttt{)}$.

We add a primitive context to indicate that a subterm is to be evaluated, $[\text{eval } T]$; and, at the same time, we introduce a separate nonterminal S for terms that we anticipate will be self-evaluating. (We also introduce a separate nonterminal A for "active" terms, which is organizationally convenient although it has no bearing on reduction in this call-by-name variety of the calculus.)

$$\begin{array}{lll} S & ::= \ c \mid \langle \text{\textrevepsilon} x.T \rangle & (\text{Self-evaluating terms}) \\ A & ::= \ [\text{eval } T] \mid [\text{combine } T\ T] & (\text{Active terms}) \\ T & ::= \ x \mid S \mid \texttt{(}T \ \texttt{.} \ T\texttt{)} \mid A & (\text{Terms})\,. \end{array}$$

All this additional syntax has, of course, no effect on the pre-existing formal equations, and adds new formal equations formed by embedding the β-redex pattern in terms that involve the additional syntax, such as $\texttt{(}[\text{combine } \langle \text{\textrevepsilon}\texttt{x.(x . x)}\rangle \ \texttt{()}]\texttt{)} =_\beta \texttt{((()))}$.

Here are reduction schemata for evaluating the syntax we have so far.

$$\begin{array}{rcl} [\text{eval } S] & \longrightarrow & S \\ [\text{eval } \texttt{(}T_1 \ \texttt{.} \ T_2\texttt{)}] & \longrightarrow & [\text{combine } [\text{eval } T_1] \ T_2]\,. \end{array}$$

Variables and active terms naturally don't have their own evaluation rules, since they may represent evaluable data structures yet to be determined.

At this point, we only support operative combinations. To support applicative combinations, we add syntax for applicatives $\langle T \rangle$ (read as "wrap T"), a schema for evaluating applicatives, and schemata for scheduling the argument evaluations of an applicative combination (one schema for each possible number of operands).

Syntax (additional):

$$T \ ::= \ \langle T \rangle \qquad (\text{Terms})\,.$$

Schemata (additional):

$$\begin{array}{rcl} [\text{eval } \langle T \rangle] & \longrightarrow & \langle [\text{eval } T] \rangle \\ [\text{combine } \langle T_0 \rangle \ \texttt{()}] & \longrightarrow & [\text{combine } T_0 \ \texttt{()}] \\ [\text{combine } \langle T_0 \rangle \ \texttt{(}T_1\texttt{)}] & \longrightarrow & [\text{combine } T_0 \ \texttt{(}[\text{eval } T_1]\texttt{)}] \\ [\text{combine } \langle T_0 \rangle \ \texttt{(}T_1 \ T_2\texttt{)}] & \longrightarrow & [\text{combine } T_0 \ \texttt{(}[\text{eval } T_1] \ [\text{eval } T_2]\texttt{)}] \\ & \vdots & \end{array}$$

These schemata evidently presuppose that operatives have some means to decompose an operand list, before binding the operands to variables one at a time using $\langle \text{\textrevepsilon} x.\Box \rangle$. To provide this means we therefore add, finally, two parameterless function abstractions: $\langle \wr_0.T\rangle$, which matches an empty operand list, and $\langle \wr_2.T\rangle$, which curries a nonempty operand list.

Syntax (additional):

$$S \;::=\; \langle \wr_0.T\rangle \mid \langle \wr_2.T\rangle \qquad \text{(Self-evaluating terms)}\,.$$

Schemata (additional):

$$\begin{array}{rcl}
[\text{combine } \langle \wr_0.T\rangle\ \texttt{()}] & \longrightarrow & T \\
[\text{combine } \langle \wr_2.T_0\rangle\ \texttt{(}T_1\ \texttt{.}\ T_2\texttt{)}] & \longrightarrow & [\text{combine } [\text{combine } T_0\ T_1]\ T_2]\,.
\end{array}$$

All together, we have

$\wr_x$-calculus.

Syntax:

$$\begin{array}{rcll}
c & \in & \mathit{Constants} & \\
x & \in & \mathit{Variables} & \\
S & ::= & c \mid \langle \wr x.T\rangle \mid \langle \wr_0.T\rangle \mid \langle \wr_2.T\rangle & \text{(Self-evaluating terms)} \\
A & ::= & [\text{eval } T] \mid [\text{combine } T\ T] & \text{(Active terms)} \\
T & ::= & x \mid S \mid \texttt{(}T\ \texttt{.}\ T\texttt{)} \mid \langle T\rangle \mid A & \text{(Terms)}\,.
\end{array}$$

Schemata:

$$\begin{array}{rcl}
[\text{combine } \langle \wr x.T_1\rangle\ T_2] & \longrightarrow & T_1[x \leftarrow T_2] \\
[\text{combine } \langle \wr_0.T\rangle\ \texttt{()}] & \longrightarrow & T \\
[\text{combine } \langle \wr_2.T_0\rangle\ \texttt{(}T_1\ \texttt{.}\ T_2\texttt{)}] & \longrightarrow & [\text{combine } [\text{combine } T_0\ T_1]\ T_2] \\
[\text{eval } S] & \longrightarrow & S \\
[\text{eval } \texttt{(}T_1\ \texttt{.}\ T_2\texttt{)}] & \longrightarrow & [\text{combine } [\text{eval } T_1]\ T_2] \\
[\text{eval } \langle T\rangle] & \longrightarrow & \langle [\text{eval } T]\rangle \\
[\text{combine } \langle T_0\rangle\ \texttt{()}] & \longrightarrow & [\text{combine } T_0\ \texttt{()}] \\
[\text{combine } \langle T_0\rangle\ \texttt{(}T_1\texttt{)}] & \longrightarrow & [\text{combine } T_0\ \texttt{(}[\text{eval } T_1]\texttt{)}] \\
[\text{combine } \langle T_0\rangle\ \texttt{(}T_1\ T_2\texttt{)}] & \longrightarrow & [\text{combine } T_0\ \texttt{(}[\text{eval } T_1]\ [\text{eval } T_2]\texttt{)}] \\
 & \vdots &
\end{array}$$

Mostly because the left-hand sides of the schemata are mutually exclusive, $\wr_x$-calculus is a *regular CRS* in the sense of [Kl80], so it has the Church-Rosser property; and because a term inside the λ-calculus subset (i.e., not using any of the syntactic extensions) can only be reduced via the β-rule, $\wr_x$-calculus is a *conservative extension* of λ-calculus.

Of course, $\wr_x$-calculus doesn't have symbols or environments, so it's not a very practical tool for reasoning about Lisp with fexprs; but it is a simple demonstration of how a λ-like calculus can support fexprs with a nontrivial equational theory. Environments would make the demonstration less immediate: one of the first things one would do would be to replace syntactic form [combine T T] with [combine T T T] —where the extra subterm is the dynamic environment of the combination— and that would be the end of the "perfect" syntactic correspondence with λ-calculus.

Nevertheless, $\imath_x$-calculus can be used to simulate evaluation of sufficiently tame Lisp expressions, including all those evaluations corresponding to reduction of λ-calculus terms with no free variables; a sufficient condition is that each symbol *that would be evaluated* can be replaced by either a constant or a bound variable (since $\imath_x$-calculus has no symbols as such, and treating symbols as constants only works so long as they aren't evaluated). As an illustration of how this works, consider the Lisp expression `(eval (cons $quote (cons (cons 2 3) ())))`. (Since the calculus doesn't support environments, we simply omit the environment argument.) The combiners in this example can be represented in $\imath_x$-calculus as

$$\begin{array}{rcl} \boldsymbol{\mathtt{eval}} & \equiv & \langle\langle\imath_2.\langle\imath\mathrm{x}.\langle\imath_0.[\mathrm{eval\ x}]\rangle\rangle\rangle\rangle \\ \boldsymbol{\mathtt{cons}} & \equiv & \langle\langle\imath_2.\langle\imath\mathrm{x}.\langle\imath_2.\langle\imath\mathrm{y}.\langle\imath_0.\mathtt{(}\mathrm{x}\ \mathtt{.}\ \mathrm{y}\mathtt{)}\rangle\rangle\rangle\rangle\rangle\rangle \\ \boldsymbol{\mathtt{\$quote}} & \equiv & \langle\imath_2.\langle\imath\mathrm{x}.\langle\imath_0.\mathrm{x}\rangle\rangle\rangle\,. \end{array}$$

Working up gradually to the example, we have

$$\begin{array}{rl} & [\mathrm{eval}\ \boldsymbol{\mathtt{eval}}] \\ \equiv & [\mathrm{eval}\ \langle\langle\imath_2.\langle\imath\mathrm{x}.\langle\imath_0.[\mathrm{eval\ x}]\rangle\rangle\rangle\rangle] \\ \longrightarrow_{\imath x} & \langle[\mathrm{eval}\ \langle\imath_2.\langle\imath\mathrm{x}.\langle\imath_0.[\mathrm{eval\ x}]\rangle\rangle\rangle]\rangle \\ \longrightarrow_{\imath x} & \langle\langle\imath_2.\langle\imath\mathrm{x}.\langle\imath_0.[\mathrm{eval\ x}]\rangle\rangle\rangle\rangle \\ \equiv & \boldsymbol{\mathtt{eval}}\,, \end{array}$$

and similarly $[\mathrm{eval}\ \boldsymbol{\mathtt{cons}}] \longrightarrow^*_{\imath x} \boldsymbol{\mathtt{cons}}$ and $[\mathrm{eval}\ \boldsymbol{\mathtt{\$quote}}] \longrightarrow_{\imath x} \boldsymbol{\mathtt{\$quote}}$. For any term T,

$$\begin{array}{rl} & [\mathrm{eval}\ \mathtt{(}\boldsymbol{\mathtt{eval}}\ T\mathtt{)}] \\ \longrightarrow_{\imath x} & [\mathrm{combine}\ [\mathrm{eval}\ \boldsymbol{\mathtt{eval}}]\ \mathtt{(}T\mathtt{)}] \\ \longrightarrow^*_{\imath x} & [\mathrm{combine}\ \boldsymbol{\mathtt{eval}}\ \mathtt{(}T\mathtt{)}] \\ \equiv & [\mathrm{combine}\ \langle\langle\imath_2.\langle\imath\mathrm{x}.\langle\imath_0.[\mathrm{eval\ x}]\rangle\rangle\rangle\rangle\ \mathtt{(}T\mathtt{)}] \\ \longrightarrow_{\imath x} & [\mathrm{combine}\ \langle\imath_2.\langle\imath\mathrm{x}.\langle\imath_0.[\mathrm{eval\ x}]\rangle\rangle\rangle\ \mathtt{(}[\mathrm{eval}\ T]\mathtt{)}] \\ \longrightarrow_{\imath x} & [\mathrm{combine}\ [\mathrm{combine}\ \langle\imath\mathrm{x}.\langle\imath_0.[\mathrm{eval\ x}]\rangle\rangle\ [\mathrm{eval}\ T]]\ \mathtt{()}] \\ \longrightarrow_{\imath x} & [\mathrm{combine}\ \langle\imath_0.[\mathrm{eval\ x}]\rangle[\mathrm{x} \leftarrow [\mathrm{eval}\ T]]\ \mathtt{()}] \\ \equiv & [\mathrm{combine}\ \langle\imath_0.[\mathrm{eval}\ [\mathrm{eval}\ T]]\rangle\ \mathtt{()}] \\ \longrightarrow_{\imath x} & [\mathrm{eval}\ [\mathrm{eval}\ T]]\,. \end{array}$$

Thus, $[\mathrm{eval}\ \mathtt{(}\boldsymbol{\mathtt{eval}}\ T\mathtt{)}] =_{\imath x} [\mathrm{eval}\ [\mathrm{eval}\ T]]$. Also,

$$\begin{array}{rl} & [\mathrm{eval}\ \mathtt{(}\boldsymbol{\mathtt{\$quote}}\ T\mathtt{)}] \\ \longrightarrow_{\imath x} & [\mathrm{combine}\ [\mathrm{eval}\ \boldsymbol{\mathtt{\$quote}}]\ \mathtt{(}T\mathtt{)}] \\ \longrightarrow_{\imath x} & [\mathrm{combine}\ \boldsymbol{\mathtt{\$quote}}\ \mathtt{(}T\mathtt{)}] \\ \equiv & [\mathrm{combine}\ \langle\imath_2.\langle\imath\mathrm{x}.\langle\imath_0.\mathrm{x}\rangle\rangle\rangle\ \mathtt{(}T\mathtt{)}] \\ \longrightarrow_{\imath x} & [\mathrm{combine}\ [\mathrm{combine}\ \langle\imath\mathrm{x}.\langle\imath_0.\mathrm{x}\rangle\rangle\ T]\ \mathtt{()}] \\ \longrightarrow_{\imath x} & [\mathrm{combine}\ \langle\imath_0.\mathrm{x}\rangle[\mathrm{x} \leftarrow T]\ \mathtt{()}] \\ \equiv & [\mathrm{combine}\ \langle\imath_0.T\rangle\ \mathtt{()}] \\ \longrightarrow_{\imath x} & T\,, \end{array}$$

so $[\mathrm{eval}\ \mathtt{(}\boldsymbol{\mathtt{\$quote}}\ T\mathtt{)}] =_{\imath x} T$. For any terms T_1 and T_2,

$$
\begin{array}{rl}
 & [\mathrm{eval}\ \texttt{(}\textit{\texttt{cons}}\ T_1\ T_2\texttt{)}] \\
\longrightarrow_{f_x} & [\mathrm{combine}\ [\mathrm{eval}\ \textit{\texttt{cons}}]\ \texttt{(}T_1\ T_2\texttt{)}] \\
\longrightarrow^*_{f_x} & [\mathrm{combine}\ \textit{\texttt{cons}}\ \texttt{(}T_1\ T_2\texttt{)}] \\
\equiv & [\mathrm{combine}\ \langle\langle f_2.\langle f\mathrm{x}.\langle f_2.\langle f\mathrm{y}.\langle f_0.\texttt{(x . y)}\rangle\rangle\rangle\rangle\rangle\rangle\ \texttt{(}T_1\ T_2\texttt{)}] \\
\longrightarrow_{f_x} & [\mathrm{combine}\ \langle f_2.\langle f\mathrm{x}.\langle f_2.\langle f\mathrm{y}.\langle f_0.\texttt{(x . y)}\rangle\rangle\rangle\rangle\rangle\ \texttt{(}[\mathrm{eval}\ T_1]\ [\mathrm{eval}\ T_2]\texttt{)}] \\
\longrightarrow^*_{f_x} & [\mathrm{combine}\ \langle f_2.\langle f\mathrm{y}.\langle f_0.\texttt{(}[\mathrm{eval}\ T_1]\ \texttt{. y)}\rangle\rangle\rangle\ \texttt{(}[\mathrm{eval}\ T_2]\texttt{)}] \\
\longrightarrow^*_{f_x} & [\mathrm{combine}\ \langle f_0.\texttt{(}[\mathrm{eval}\ T_1]\ \texttt{.}\ [\mathrm{eval}\ T_2]\texttt{)}\rangle\ \texttt{()}] \\
\longrightarrow_{f_x} & \texttt{(}[\mathrm{eval}\ T_1]\ \texttt{.}\ [\mathrm{eval}\ T_2]\texttt{)}\,,
\end{array}
$$

so $[\mathrm{eval}\ \texttt{(}\textit{\texttt{cons}}\ T_1\ T_2\texttt{)}] =_{f_x} \texttt{(}[\mathrm{eval}\ T_1]\ \texttt{.}\ [\mathrm{eval}\ T_2]\texttt{)}$. Finally,

$$
\begin{array}{rl}
 & [\mathrm{eval}\ \texttt{(}\textit{\texttt{eval}}\ \texttt{(}\textit{\texttt{cons}}\ \textit{\texttt{\$quote}}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{2 3) ()))))}] \\
\longrightarrow^*_{f_x} & [\mathrm{eval}\ [\mathrm{eval}\ \texttt{(}\textit{\texttt{cons}}\ \textit{\texttt{\$quote}}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{2 3) ()))}]] \\
\longrightarrow^*_{f_x} & [\mathrm{eval}\ \texttt{(}[\mathrm{eval}\ \textit{\texttt{\$quote}}]\ \texttt{.}\ [\mathrm{eval}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{2 3) ())}]\texttt{)}] \\
\longrightarrow_{f_x} & [\mathrm{eval}\ \texttt{(}\textit{\texttt{\$quote}}\ \texttt{.}\ [\mathrm{eval}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{2 3) ())}]\texttt{)}] \\
\longrightarrow^*_{f_x} & [\mathrm{eval}\ \texttt{(}\textit{\texttt{\$quote}}\ [\mathrm{eval}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{2 3)}]\ \texttt{.}\ [\mathrm{eval}\ \texttt{()}]\texttt{)}] \\
\longrightarrow_{f_x} & [\mathrm{eval}\ \texttt{(}\textit{\texttt{\$quote}}\ [\mathrm{eval}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{2 3)}]\texttt{)}] \\
\longrightarrow^*_{f_x} & [\mathrm{eval}\ \texttt{(}\textit{\texttt{\$quote}}\ \texttt{(}[\mathrm{eval}\ 2]\ \texttt{.}\ [\mathrm{eval}\ 3]\texttt{))}] \\
\longrightarrow^*_{f_x} & \texttt{(}[\mathrm{eval}\ 2]\ \texttt{.}\ [\mathrm{eval}\ 3]\texttt{)} \\
\longrightarrow^*_{f_x} & \texttt{(2 . 3)}\,,
\end{array}
$$

so $[\mathrm{eval}\ \texttt{(}\textit{\texttt{eval}}\ \texttt{(}\textit{\texttt{cons}}\ \textit{\texttt{\$quote}}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{(}\textit{\texttt{cons}}\ \texttt{2 3) ()))))}] =_{f_x} \texttt{(2 . 3)}$.

References

[AbSu85] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, New York: McGraw-Hill, 1985.

The first edition of the Wizard Book [Ra03, "Wizard Book"]. Mostly superseded by [AbSu96].

[AbSu96] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, 1996. Available (as of October 2009) at URL:
`http://mitpress.mit.edu/sicp/sicp.html`

The second edition of the Wizard Book [Ra03, "Wizard Book"].

[Ba78] John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs', *Communications of the ACM* 21 no. 8 (August 1978), pp. 613–641.

Augmented form of the 1977 ACM Turing Award Lecture, which proposes the function programming paradigm, and coins the term "von Neumann bottleneck".

[Cl85] William Clinger, editor, *The Revised Revised Report on Scheme, or An Uncommon Lisp*, memo 848, MIT Artificial Intelligence Laboratory, August 1985. Also published as Computer Science Department Technical Report 174, Indiana University, June 1985. Available (as of October 2009) at URL:
`http://www.ai.mit.edu/publications/`

This is the first of the *RxRS*s with a huge pile of authors (for two perspectives, see [ReCl86, Introduction], [SteGa93, §2.11.1]). There is, as one might expect from a committee, almost nothing in the way of motivation; however, there is also —as one would not normally expect from a committee— a verse about lambda modeled on J.R.R. Tolkien's verse about the Rings of Power.

[Cl98] William Clinger, "Proper Tail Recursion and Space Efficiency", *SIGPLAN Notices* 33 no. 5 (May 1998) [*Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, 17–19 June 1998], pp. 174-185. Available (as of October 2009) at URL:
`http://www.ccs.neu.edu/home/will/papers.html`

Formally defines Scheme proper tail recursion (but not proper tail recursion in general).

[ClRe91b] William Clinger and Jonathan Rees, editors, "The Revised^4 Report on the Algorithmic Language Scheme", *Lisp Pointers* 4 no. 3 (1991), pp. 1–55.

Available (as of October 2009) at URL:
`http://www.cs.indiana.edu/scheme-repository/doc.standards.html`

[CrFe91] Erik Crank and Matthias Felleisen, “Parameter-Passing and the Lambda Calculus”, *POPL ’91: Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 21–23, 1991, pp. 233–244. Available (as of October 2009) at URL:
`http://www.ccs.neu.edu/scheme/pubs/#popl91-cf`

[Di03] Ray Dillinger, “External Representation for Data with Shared Structure”, *SRFI-38*, finalized 2 April 2003. Available (as of October 2009) at URL:
`http://srfi.schemers.org/srfi-38/`

[Gu91] David Gudeman, “Re: Complexity of syntax”, USENET posting, Message-ID: `<373@coatimundi.cs.arizona.edu>`, Newsgroup: `comp.lang.misc`, January 1991.

Recommends four criteria for a first-class object. Quoting from the article:

“(1) The object can be produced as the value of general expressions of the correct type.

(2) The object can be used in any expression of the correct type.

(3) The lifetime of the object is unbounded (I’m talking high-level semantics here, so I don’t have to mention garbage collection.)

(4) It is not (in general) decidable that a given object is produced by no expression in a program. (In other words, the object may be created at runtime).”

[IE85] *IEEE Standard 754-1985. IEEE Standard for Binary Floating-Point Arithmetic.* IEEE, New York, 1985.

[Ka93] Alan C. Kay, “The Early History of Smalltalk”, *SIGPLAN Notices* 28 no. 3 (March 1993) [Preprints, *ACM Sigplan Second History of Programming Languages Conference*, Cambridge, Massachusetts, April 20–23, 1993], pp. 69–95.

[KeClRe98] Richard Kelsey, William Clinger, and Jonathan Rees, editors, “Revised^5 Report on the Algorithmic Language Scheme”, 20 February 1998. Available (as of October 2009) at URLs:
`http://groups.csail.mit.edu/mac/projects/scheme/index.html`
`http://download.plt-scheme.org/doc/202/html/r5rs/index.htm`

[Kl80] Jan Willem Klop, *Combinatory Reduction Systems*, Ph.D. Thesis, University of Utrecht, 1980. Also, Mathematical Centre Tracts 127. Available (as of October 2009) at URL:
`http://web.mac.com/janwillemklop/Site/Bibliography.html`

[La00] P. J. Landin, "My Years with Strachey", *Higher-Order and Symbolic Computation* 13 no. 1/1 (April 2000), pp. 75–76.

[McC60] John McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine", *Communications of the ACM* 3 no. 4 (April 1960), pp. 184–195.

The original reference for Lisp.

[McC+62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin, *LISP 1.5 Programmer's Manual*, Cambridge, Massachusetts: The MIT Press, 1962. Available (as of October 2009) at URL:
`http://www.softwarepreservation.org/projects/LISP/book/`

The second edition was 1965, same authors.

[MitGnu] MIT/GNU Scheme. Available (as of October 2009) at URL:
`http://www.gnu.org/software/mit-scheme/`

[Mor73] James H. Morris Jr., "Types are not Sets", *POPL '73: Conference Record of ACM Symposium on Principles of Programming Languages*, Boston, Massachusetts, October 1–3, 1973, pp. 120–124.

[Mos00] Peter D. Mosses, "A Foreword to 'Fundamental Concepts in Programming Languages'", *Higher-Order and Symbolic Computation* 13 no. 1/2 (April 2000), pp. 7–9.

[Pi83] Kent M. Pitman, *The Revised Maclisp Manual* (Saturday evening edition), MIT Laboratory for Computer Science Technical Report 295, May 21, 1983.

[Pi07] Kent M. Pitman, *The Revised Maclisp Manual* (Sunday morning edition), published Sunday, December 16, 2007, updated Sunday, July 6, 2008. Available (as of October 2009) at URL:
`http://maclisp.info/pitmanual/`

[Plo75] Gordon D. Plotkin, "Call-by-name, call-by-value, and the λ-calculus", *Theoretical Computer Science* 1 no. 2 (December 1975), pp. 125–159. Available (as of October 2009) at URL:
`http://homepages.inf.ed.ac.uk/gdp/publications/`

[Qu61] W. V. Quine, "On What There Is", in *from a logical point of view*, Second Edition, revised, New York: Harper & Row, 1961, pp. 1–19.

The book is a collection of nine "logico-philosophical essays".

[Ra03] Eric S. Raymond, *The Jargon File*, version 4.4.7, 29 December 2003. Available (as of October 2009) at URL:
`http://www.catb.org/~esr/jargon/`

[ReCl86] Jonathan Rees and William Clinger, editors, "The Revised^3 Report on the Algorithmic Language Scheme", *SIGPLAN Notices* 21 no. 12 (December 1986), pp. 37–43. Available (as of October 2009) at URL:
`http://www.cs.indiana.edu/scheme-repository/doc.standards.html`

The second of the *RxRS*s authored by a committee. Introduces a high-level statement on language-design principles in the *Introduction*, which has been passed on to all the *RxRS*s since.

[Sh08] John N. Shutt, "Abstractive Power of Programming Languages: Formal Definition", technical report WPI-CS-TR-08-01, Worcester Polytechnic Institute, Worcester, MA, March 2008, emended 26 March, 2008. Available (as of October 2009) at URL:
`http://www.cs.wpi.edu/Resources/techreports.html`

[Sh09] John N. Shutt, *Fexprs as the basis of Lisp function application; or, **$vau**: the ultimate abstraction*, Ph.D. Dissertation, WPI CS Department, forthcoming.

[Sp+07] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten, "Revised^6 Report on the Algorithmic Language Scheme", 26 September 2007. Available (as of October 2009) at URL:
`http://www.r6rs.org/`

[Sta75] Thomas A. Standish, "Extensibility in Programming Language Design", *SIGPLAN Notices* 10 no. 7 (July 1975) [*Special Issue on Programming Language Design*], pp. 18–21.

A retrospective survey of the subject, somewhat in the nature of a postmortem. The essence of Standish's diagnosis is that the extensibility features required an expert to use them. He notes that when a system is complex, modifying it is complex. (He doesn't take the next step, though, of suggesting that some means should be sought to reduce the complexity of extended systems.)

He classifies extensibility into three types: *paraphrase* (defining a new feature by showing how to express it with pre-existing features — includes ordinary procedures as well as macros); *orthophrase* (adding new facilities that are orthogonal to what was there — think of adding a file system to a language that didn't have one); and *metaphrase* (roughly what would later be called "reflection").

[Ste90] Guy Lewis Steele Jr., *Common Lisp: The Language*, 2nd Edition, Digital Press, May 1990. Available (as of October 2009) at URL:
`http://www-2.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html`

[SteGa93] Guy L. Steele Jr. and Richard P. Gabriel, "The Evolution of Lisp", *SIGPLAN Notices* 28 no. 3 (March 1993) [Preprints, *ACM SIGPLAN Second*

History of Programming Languages Conference, Cambridge, Massachusetts, April 20–23, 1993], pp. 231–270.

[SteSu78] Guy Lewis Steele Jr. and Gerald Jay Sussman, *The Revised Report on Scheme, a Dialect of Lisp*, memo 452, MIT Artificial Intelligence Laboratory, January 1978.

[Str67] Christopher Strachey, "Fundamental Concepts in Programming", Lecture notes for International Summer School on Computer Programming, Copenhagen, August 7 to 25, 1967.

These notes have been cited commonly when crediting Strachey with coining the term *polymorphism* as it applies to programming languages. The discussion of polymorphism is on page 10.

The discussion of first-class objects is on pp. 9–10.

Strachey later turned these notes into a paper, which remained unpublished until it appeared as [Str00]. Although the finished paper is far more articulate on most topics, some specific points are more clearly expressed by the notes.

[Str00] Christopher Strachey, "Fundamental Concepts in Programming Languages", *Higher-Order and Symbolic Computation* 13 no. 1/2 (April 2000), pp. 11–49.

This is Strachey's paper based on his lectures, [Str67]. On the history of the paper, see [Mos00].

[SumPi04] Eijiro Sumii and Benjamin C. Pierce, "A Bisimulation for Dynamic Sealing", *SIGPLAN Notices* 39 no. 1 (January 2004) [*Proceedings of the 2004 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, Venice, Italy, January 14–16, 2004], pp. 161–172.

[Sus07] Gerald Jay Sussman, "Building Robust Systems", essay, January 13, 2007. Available (as of October 2009) at URL:
`http://groups.csail.mit.edu/mac/users/gjs/6.945/readings/robust-systems.pdf`

[SusSte75] Gerald Jay Sussman and Guy Lewis Steele Jr., *Scheme: an Interpreter for Extended Lambda Calculus*, memo 349, MIT Artificial Intelligence Laboratory, December 1975. Available (as of October 2009) at URL:
`http://www.ai.mit.edu/publications/`

The revised^0 report on Scheme.

[Ta99] Walid Taha, "Multi-Stage Programming: Its Theory and Applications", Ph.D. Dissertation, Technical Report CSE-99-TH-002, Oregon Graduate Institute of Science and Technology, November 1999. Available (as of October 2009) at URL:
`http://www.cs.rice.edu/~taha/publications.html`

[vT04] André van Tonder, "Primitives for Expressing Iterative Lazy Algorithms", *SRFI-45*, post-final version with bug fix, 4 August 2004. Available (as of October 2009) at URL:
`http://srfi.schemers.org/srfi-45/`

[Wa98] Mitchell Wand, "The Theory of Fexprs is Trivial", *Lisp and Symbolic Computation* 10 no. 3 (May 1998), pp. 189–199. Available (as of October 2009) at URL:
`http://www.ccs.neu.edu/home/wand/pubs.html#Wand98`

This paper is a useful elaboration of the basic difficulty caused by mixing fexprs with implicit evaluation in an equational theory. Along the way, the author makes various sound observations. "We care," he notes in the concluding section, "not only when two terms have the same behavior, but we may also care just what that behavior is."

Alphabetical index

Items may be concepts, terms, types, keywords, or combiners. The principal entries for an item are listed first, in **boldface**, separated from the other entries by a semicolon. Keywords prefixed by #, and combiners prefixed by $, are alphabetized without the prefix. (For example, *$vau* is alphabetized under *v*.)