The Koioslisp Specification and Notes, version 0.0

Yash Tulsyan

January 2, 2013

Contents

Chapter 1

Preface

Chapter 2

Acknowledgements

If Koioslisp is a better language than others (a questionable proposition), it is, proverbially, because it stood on the shoulders of giants. First and foremost, I must acknowledge all the Lispers who came before me–the hackers of MIT, BBN, Xerox, the Lisp Machine world, and the Lisp community of today–for without their work, none of this would be possible. In particular, I must acknowledge the great implementors and theorists–giants of the LISP community: Professor John McCarthy, Steve Russell, Jonl White, Guy L. Steele, David Moon, Richard P. Gabriel, Richard M. Stallman, Gerald Jay Sussman, and more. I must also acknowledge the giants of recent times such as Paul Graham, Peter Seibel, Pascal Costanza, Edi Weitz, Zach Beane, François-René Rideau, James Kalenius, Nikodemus Siivola, Shriram Krishnamurthi, Stanislav Datskovskiy and Rich Hickey.

Chapter 3

Inspiration and Design Features

Primary inspiration for KOIOSLISP came from the COMMON LISP programming language (indeed, the document you are reading right now is structured based on the authoritative COMMON LISP reference, *Common Lisp, the Language* by Guy L. Steele), with influences from SCHEME, CLOJURE, and PROLOG A design influence comes from the language PYTHON and other modern scripting languages–in that "batteries" (i.e. utilities to connect to the rest of the world) should come included in the standard library, which is indeed part of the reason features which seem superfluous are included in KOIOSLISP. In the specification, batteries will be marked clearly, as such utilities are far more likely to change over time. KOIOSLISP attempts to unify the strengths of these Lisp dialects into one–COMMON LISP for its completeness, macros, object system, historical significance,large amounts of libraries (hopefully, the amount of compatibility given by KOIOSLISP will encourage people to port useful libraries to it, such as ASDF, or SWANK(which is used for SLIME, the free Lisp IDE on Emacs)), and continuity; SCHEME for its elegance, continuations, and assorted innovations; CLOJURE for its concurrency support, clean hash tables, and lazy evaluation.It also features PROLOG-style logic programming. In doing so, it tries to avoid the problems of the languages above mentioned. While COMMON LISP and SCHEME have aged relatively well, COMMON LISP has non-lispy hash tables and some marks of age, while SCHEME, though good for pedagogy, is crippled by its size and lack of powerful macros (though it does have "hygenic" macros), as well as some marks of age. Meanwhile, CLOJURE is dependent on the JVM (or, in the case of its new ports, the CLR and JavaScript–which create some incompatibility between implementations, as CLOJURE feeds off of those engines and absorbs their libraries and object systems) and departs from the "mud ball" metaphor of lisp, and PROLOG is entirely unusable for programming-in-the-large, being useful mainly as an extension to Lisp. Throughout this document, some compatibility notes will be recorded.

Chapter 4

Introduction

This is a specification for a KOIOSLISP, a LISP dialect, which will evolve as time goes on and the fickle attitudes of the BDFL, Yash Tulsyan, change. If you are a programmer from C-like languages who expects a C-like syntax and a C++ (or Java/C#) like object-oriented system imposed upon the user, then please either accept that this will not cater to that palate, or leave. If you are a programmer from Haskell or ML-like languages who is similarly put-off by s-expressions, want a functional style to be imposed upon the user, and caution against side-effects, we advise a similar course of action. As of this moment, this specification does not come with any standard implementation, due to the fact that the BDFL is currently a simple student who has not yet learned how to create a fully-featured compiler or an interpreter.

4.1 Do We Really Need Another Dialect?

This question appears whenever a new language appears, and in Lisp it is especially important, due to both its history and COMMON LISP macros. My answer to this: Maybe. If so, this is going to be it. If not, we (or at least, I) will probably use Common Lisp.

4.2 Conventions Used in This Book

Chapter 5

Data Types

5.1 Class Hierarchy of Koioslisp

5.2 Numbers

As in most programming languages, there are several different types of numbers in KOIOSLISP, and, as in Common Lisp, they are divided into four main categories: *integers*, *ratios*, *floating-point numbers*, and *complex numbers*. Integers and ratios are contained by the type *rational*. Rational numbers and floating-point numbers are of the type *real*, and finally real numbers and complex numbers are of type *number*.

5.2.1 Integers

Integers function as in Common Lisp–as representations of mathematical integers. There are no standard restrictions on the magnitude of an integer, as storage should be automatically allocated for representing large integers. However, there might still be a low-level distinction between very efficient integers of smaller magnitude (`fixnums`) and larger integers, called `bignums`. Though this distinction may exist in implementations of the language, Koioslisp is designed, like Common Lisp, to hide the distinction whenever possible. Because the numbers which may be considered fixnums and those which may be considered bignums will vary from implementation to implementation, the constants `+most-positive-fixnum+` and `+most-negative-fixnum+` are provided.As in most programming languages, integers are canonically written as they are canonically in arithmetic–with decimal digits, possibly preceded by a sign and followed by a decimal point. Integers can be written in bases other than the default 10. The syntax `#nnrddddd`, case insensitively, will refer to the integer represented in *nn*-radix by *ddddd*, where *nn* is between 2 and 36, inclusive, and *ddddd* is an integer. When representing numbers in radices above 10, use of letters to represent numbers is case-insensitive–that is, `#16rFACADE` is the same as `#16RfacaDE`. Binary, octal, and hexadecimal are relatively common, and so they are granted the abbreviations `#b`, `#o` and `#x`.

5.2.2 Ratios

A *ratio* is, as in COMMON LISP, a representation of a mathematical ratio, and the other part (along with integers) that constitute the type `rational`. The canonical representation of a ratio is either an integer if it is equal to an integer, or the ratio of a *numerator* and *denominator*, which are both integers, and whose greatest common denominator is one. The character \ separates the numerator from the denominator. One can input a non-canonical value, but a ratio will always be printed by `print` in its canonical form. As all other reals, ratios may be signed, (with the same syntax as other real numbers). Internally, ratios in non-canonical form will be converted to canonical form. The denominator of a ratio must not be zero; if it is zero, an error is signaled. As with integers, ratios can be used in different radices, with the same syntax.

5.2.3 Floating Point Numbers

5.2.4 Complex Numbers

5.3 Characters

5.3.1 Standard Characters

Koioslisp uses a standard character set in order to enhance portability–Koioslisp programs that use only the standard set of characters can be (assuming no other implementation-specific (mis)features are used) read by any Koioslisp implementation. The standard character set of KOIOSLISP is nearly identical to the Unicode character set. As the Unicode character set has several thousands of characters, it will not be displayed here. It also includes ways to denote non-printing characters, such as

```
#\newline #\space #\backspace #\tab #\linefeed #\page #\return
```

. Of the non-blank printing characters, most are not used in KOIOSLISP syntax. For brevity, we shall note the subset of the subset of the Unix code subset "Basic Latin" which denotes punctuation which is not commonly used in KOIOSLISP syntax: [ ] { } ! ^_~$ %. Almost all of these, however, are used as format directives. Other than that, the following characters are explicitly reserved for the user's syntactic extenstion purposes: [,], {, }, $, %, !, . This preserves the syntactic purity and clarity of Lisp, unlike some recent dialects which insist on the introduction of other forms of brackets, allegedly for purposes of clarity. See chapter "Input/Output" for greater detail on transforming constituent characters into macro characters

5.3.2 Line Divisions

Unlike the nightmare of implementing a standard method of line division in previous decades, the structure of line divisions has simplified from a variety of schemas involving linefeeds, carriage returns, and other such characters into two schemes–either a bare linefeed (LF) character, used by UNIX and its descendants; or a linefeed following a carriage return (CR), used by MS-DOS and Microsoft Windows. Like Common Lisp and C, KOIOSLISP uses the abstract requirement of requiring a single character,

```
#\newline
```

to act as a delimiter between lines. This character may be externally translated to whatever sort of line-division schema the underlying operating system uses, however, it must internally use one character, and one character only. Another issue is raised. Suppose one types the following:

```
 #\Return#\Newline
```

. Implementations should not suppress the call to

```
#\Return
```

, for there are certain standards (such as the Internet Relay Chat (IRC) protocol) which require CRLF sequences.

5.3.3 Non-standard Characters

KOIOSLISP implementations may add extra characters, however, this may render programs using such characters unportable.

5.3.4 Character Attributes

5.4 Symbols

5.5 Arrays

5.6 Vectors

5.7 Strings

5.8 Bit-Vectors

5.9 Hash Tables

5.10 Readtables

5.11 Packages

5.12 Pathnames

5.13 Sockets

5.14 Streams

5.15 Random-States

5.16 Structures

5.17 Functions

5.18 Objects

5.19 Continuations

KIOSLISP, like Scheme, provides *continuations*, convenient and abstract mechanisms for controlling the control state of a program. Unlike Scheme, however, these are *delimited* continuations. Though they can be discussed here as structures, due to their nature as GOTO-like mechanisms (though theoretically much more fundamental), they are discussed in greater detail in the Continuations chapter of this book.

5.20 Laziness and Lazy Types

Though *lazy evaluation* refers to an evaluation strategy, it is useful to speak of *lazy* forms of data structures, of which the most central is the lazy list.

5.21 Unreadable Data Objects

5.22 Overlap, Inclusion, and Disjointness of Types

Chapter 6

Scope and Extent

Scope and *extent* are important when discussing features of a language. *Scope* is the spatial region on which a variable is defined–*extent* is the temporal region. Koioslisp, like Common Lisp, utilizes two systems of scope and two systems of extent–*lexical scope*, *indefinite scope*, *dynamic extent*, and *indefinite extent*.

Chapter 7

Type Specifiers

Chapter 8

Program Structure

Chapter 9

Predicates

[*Function*]
`eq?`
[*Function*]
`eqv?`
[*Generic Function*]
`=?`
[*Function*]
`not` *x*
[*Function*]
`type?` *object type*

Chapter 10

Control Structure

[*Macro*]
`and` &rest *forms*
[*Macro*]
`or` &rest *forms*
[*Function*]
`xor` &rest *forms*
[*Macro*]
`zap!`
[*Macro*]
`alias`
[*Macro*]
`partial` *function args-or-placeholders*
A *placeholder* is represented the atom _ and the placeholder arguments form the arguments of the returned function
[*Function*]
`memoize`
[*Function*]
`compose` *f* &rest *funs*
[*Function*]
`conjoin` &rest *predicates*
[*Function*]
`disjoin` &rest *predicates*

10.1 Constants and Variables

10.1.1 Reference

10.1.2 Assignment and Bindings

Low-Level Assignment

Like TAGBODY, PROG and GO, many of these functions are not used often in casual code, but they are necessary in the implementation of some of the higher-level functions and indispensible in the implementation of new languages upon Koios. [*Function*]
`%set`
[*Function*]

`%setq`

Generalized References

[*Macro*]
`set!`
[*Macro*]
`pset!`
[*Macro*]
`rotate!`
[*Macro*]
`shift!`
[*Macro*]
`defzap`
[*Macro*]
`defsetter`
[*Macro*]
`defsetter*`

Establishing New Bindings

[*Special form*]
`let` ({*var* | (*var value*)}*) {*declaration*}* {*form*}*
A `let` form introduces new variable bindings to be used in the forms it contains.

Unlike previous lisps (except Clojure), in Koioslisp, `let` supports *destructuring*, of the sort found in the `destructuring-bind` construct in COMMON LISP. It also supports the functionality previously in the COMMON LISP construct `multiple-value-bind`. See also the Koioslisp constraint-programming facilities.

10.2 Sequencing

[*Special form*]
`begin` {*form*}*
[*Special form*]
`%progx` *integer* {*form*}*

10.3 Iteration

[*Macro*]
`do` ({*var* | (*var* [*init* [*step*]])}*) (*end-test* {*result*}*) {*declaration*}* {*tag* | *statement*}*
`do` is a very powerful, useful, and arcane (at least, to newcomers) macro meant to facilitate complex iterations.
[*Macro*]
`do*`({*var* | (*var* [*init* [*step*]])}*) (*end-test* {*result*}*) {*declaration*}* {*tag* | *statement*}*
As the name suggests, `do*` is a variant of `do`
[*Macro*]
`doseq` (*var listform* [*resultform*]) {*declaration*}* {*tag* | *expression*}*

`doseq`
[*Macro*]
`dotimes` (*var listform* [*countform*]) {*declaration*}* {*tag* | *expression*}*
`dotimes`

10.3.1 Low-Level Iteration

Like %SET and %SETQ, many of these functions are not used often in casual code, but they are necessary in the implementation of some of the higher-level functions and indispensible in the implementation of new languages upon Koios.
[*Special form*]
`%tagbody` {*tag* | *expression*}*
[*Special form*]
`%go` *tag*

[*Macro*]
`%prog` ({*var* | (*var* [*init*])}*) {*declaration*}* {*tag* | *expression*}*
[*Macro*]
`%prog*` ({*var* | (*var* [*init*])}*) {*declaration*}* {*tag* | *expression*}*

10.3.2 LOOP

The syntax and semantics of the LOOP macro in Koioslisp is very different than that of Common Lisp, which has been criticized for being inelegant, inextensible, and generally un-Lispy; though regarded as an excellent demonstration of Lisp's ability to create domain-specific languages. Instead, it is based upon the related ITERATE library for Common Lisp.
[*Macro*]
`loop`
[*Loop clause*]
`repeat` *n*
[*Loop clause*]
`for` *var* &sequence
&sequence is one of the following symbols: `:from`, `:upfrom`, `:downfrom`, `:to`, `:downto`, `:above`, `:below`, and `:by`.
[*Loop clause*]
`for` *var* `:in` *sequence* &sequence
[*Loop clause*]
`for` *key val* `:in-table` *table*
[*Loop clause*]
`for` *var* `:in-package` *package* &optional `:external-only` *external-only?*
[*Loop clause*]
`for` *var* `:in-packages` *package* &optional `:having-access` *symbol-types*
[*Loop clause*]
`for` *var* `:in-file` *filename* &optional `:using` *reader*
[*Loop clause*]
`for` *var* `:in-stream` *stream* &optional `:using` *reader*
[*Loop clause*]
`for` *var* `:next` *expr*
[*Loop clause*]
`for` *var* `:do-next` *form*
[*Loop clause*]

`for` *pvar* `:previous` *var* &optional `:initially` *init* `:back` *n*
[*Loop clause*]
`with` *var* &optional `:=` *value*
[*Loop clause*]
`for` *var* `:=` *expr*
[*Loop clause*]
`for` *var* `:initially` *init-expr* `:then` *then-expr*
[*Loop clause*]
`for` *var* `:first` *first-expr* `:then` *then-expr*
[*Loop clause*]
`sum` *expr* &optional `:into` *var*
[*Loop clause*]
`multiply` *expr* &optional `:into` *var*
[*Loop clause*]
`counting` *expr* &optional `:into` *var*
[*Loop clause*]
`maximize` *expr* &optional `:into` *var*
[*Loop clause*]
`reducing` *expr* `:by` *func* &optional `:initial-value` *init-val* `:into` *var*
[*Loop clause*]
`collecting` *exptr* &optional `:into` *var* `:at` *place* `:result-type` *type*
place is one of `:start`, `:beginning` (a synonym for `:start`) or `:end`.
[*Loop clause*]
`adjoining` *exptr* &optional `:into` *var* `:at` *place* `:result-type` *type*
place is one of `:start`, `:beginning` (a synonym for `:start`) or `:end`.
[*Loop clause*]
`appending` *exptr* &optional `:into` *var* `:at` *place* `:result-type` *type*
place is one of `:start`, `:beginning` (a synonym for `:start`) or `:end`.
[*Loop clause*]
`appending!` *exptr* &optional `:into` *var* `:at` *place* `:result-type` *type*
place is one of `:start`, `:beginning` (a synonym for `:start`) or `:end`.
[*Loop clause*]
`unioning` *exptr* &optional `:into` *var* `:at` *place* `:result-type` *type*
place is one of `:start`, `:beginning` (a synonym for `:start`) or `:end`.
[*Loop clause*]
`unioning!` *exptr* &optional `:into` *var* `:at` *place* `:result-type` *type*
place is one of `:start`, `:beginning` (a synonym for `:start`) or `:end`.
[*Loop clause*]
`accumulate` *expr* `:by` *func* &optional `:initial-value` *init-val* `:into` *var*
[*Loop clause*]
`finding` *expr* `:such-that` *test* &optional `:into` *var* `:on-failure` *failure-value*
[*Loop clause*]
`finding` *expr* `:maximizing` *m-expr* &optional `:into` *var*
[*Loop clause*]
`finding` *expr* `:minimizing` *m-expr* &optional `:into` *var*
[*Loop clause*]
`first-iteration?`
[*Loop clause*]
`first-time?`
[*Loop clause*]
`always` *expr*
[*Loop clause*]

`never` *expr*
[*Loop clause*]
`there-is` *expr*
[*Loop clause*]
`finish`
[*Loop clause*]
`leave` &optional *value*
[*Loop clause*]
`next-iteration`
[*Loop clause*]
`while` *expr*
[*Loop clause*]
`until` *expr*
[*Loop clause*]
`if-first-time` *then* &optional *else*
[*Loop clause*]
`initially` &rest *forms*
[*Loop clause*]
`after-each` &rest *forms*
[*Loop clause*]
`else` &rest *forms*
[*Loop clause*]
`finally` &rest *forms*
[*Loop clause*]
`finally-protected` &rest *forms*
[*Loop clause*]
`in` *name* &rest *forms*
[*Function*]
`display-loop-clauses` &optional *clause-spec*
[*Macro*]
`defloop` *arglist* &body *body*
[*Macro*]
`defloop-driver` *arglist* &body *body*
[*Macro*]
`defloop-alias` *syn word*
[*Macro*]
`defloop-sequence` *element-name index-name* &key *access-fn size-fn sequence-type element-type element-doc-string index-doc-string*

10.4 List Comprehension

List comprehension refers to a convenient method of generating collections of items that fit a specific criterion. [*Function*]
`iota` *start* &optional *end* &key `:step`
This function has the alias ι, that is, the literal small iota character (U+03B9 and optionally U+2373). If an *end* is not specified, it returns a series; otherwise, it returns a list.
[*Macro*]
`collect`
[*Collect option*]
`while`

[*Collect option*]
`in`
[*Collect option*]
`when`

10.5 Catch, Throw, and Unwind-Protect

[*Special form*]
`catch` *tag* {*form*}*
[*Special form*]
`unwind-protect` *protected-form* {*cleanup-form*}*
[*Special form*]
`throw` *tag result*

Chapter 11

Macros

11.1 Definition of Macros

[*Function*]
`macro-function` *symbol*
[*Macro*]
`defmacro` *name lambda-list* [[{*declaration*}* | *doc-string*]]{*form*}*
[*Macro*]
`defmacro*` *name lambda-list* [[{*declaration*}* | *doc-string*]]{*form*}* `defmacro*` is like `defmacro`, except that it ensures that the macros are *hygienic*, such as to prevent variable capture.
[*Macro*]
`define-syntax`
[*Macro*]
`syntax-rules`

11.2 Expansion of Macros

[*Function*]
`macroexpand` *form* &optional *env*
[*Function*]
`macroexpand-1` *form* &optional *env*
[*Variable*]
`*macroexpand-hook*`

11.3 Compiler Macros

11.4 Environments

Chapter 12

Declarations

[*Special form*]
`declare` {*decl-spec*}*
[*Special form*]
`locally` {*declaration*}* {*form*}*
[*Function*]
`proclaim` *decl-spec*
[*Macro*]
`declaim` {decl-spec}*
[*Macro*]
`define-declaration` *decl-name lambda-list* {*form*}*

12.1 Declaration Specifiers

`special`
`type`
type
`pure`
`multithreaded`
`return-type`
`ftype`
`inline`
`notinline`
`ignore`
`optimize`
`declaration`
`dynamic-extent`

12.2 Forms and Type Declaration

[*Special form*]
`the` *value-type form*

Chapter 13

Symbols

Chapter 14

Packages and Systems

The Common Lisp package system is *not* inherited, as it turned out to be quite a mess. [*Function*]
`load`
[*Function*]
`provide`
[*Function*]
`require`
[*Function*]
`use-system`
[*Function*]
`shadow`
[*Function*]
`intern`
[*Function*]
`unuse-system`
[*Function*]
`unintern`
[*Function*]
`import` *module symbols*
[*Function*]
`export`
[*Function*]
`defsystem`
[*System option*]
`:name`
[*System option*]
`:version`
[*System option*]
`:maintainer`
[*System option*]
`:author`
[*System option*]
`:licence`
Because some projects may be multi-licenced, or certain parts of projects can be licenced differently than others, the item following the `:licence` option can be a list, and modules (see the `:module` form) can include this option. A few pre-defined licences are provided:
`:gpl` The GNU General Public Licence
`:llgpl` The Lisp-GNU Lesser General Public Licence

`:agpl` The Affero General Public Licence
`:x11` The X11 Licence (often referred to as the MIT Licence)
`:bsd3` The 3 Clause BSD Licence (BSD Licence sans advertising clause, also referred to as the "Modified BSD Licence")
`:bsd2` The 2 Clause BSD Licence (FreeBSD licence)
`:public` Public domain

[*System option*]
`:description`
[*System option*]
`:long-description`
[*System option*]
`:serial`
[*System option*]
`:depends-on`
[*System option*]
`:components`
[*System-components form*]
`:file`
[*System-components form*]
`:depends-on`
[*System option*]
`:module`
[*System-module form*]
`:pathname`
[*System-module form*]
`:components`
[*System-module form*]
`:aliases`
[*System-module form*]
`:licence`
[*System option*]
`:aliases`

14.1 Colon syntax

`foo:bar`

`foo::bar`

`foo::(bar quux frob)`
When read, evaluates `(bar quux frob)` as if it were read in the `FOO` module. This feature is currently provided in Steel Bank Common Lisp and Allegro Common Lisp. `foo::bar::quux`
Modules can be nested, as in this example.
`:bar`
This interns the symbol `BAR` as an external symbol in the `keyword` module.
`#:bar`

14.2 Built-in Modules

`koios-lisp`

`koios-lisp-user`

`keyword`

`system`

`math`

Chapter 15

Mathematics

Koioslisp's standard mathematical features have been changed somewhat from Common Lisp's in light of the IEEE standard for floating-point arithmetic. However, they are on the whole fairly similar, and because of the presence of complex numbers in Koioslisp, some NaN (not a number results) are removed.

15.1 The Math library

[*Function*]
`rref`
[*Function*]
`mean` *sample*
[*Function*]
`median` *sample*
[*Function*]
`variance` *sample* &key *biased*
[*Function*]
`standard-deviation` *sample* &key *biased*
[*Function*]
`map-iota` *function n* &key *start step*
[*Function*]
`clamp` *number min max*
[*Function*]
`binomial-coefficient` *n k*
[*Function*]
`count-permutations` *n* &optional *k*
[*Function*]
`lerp` *v a b*
[*Function*]
`factorial` *n*
[*Function*]
`subfactorial` *n*
[*Function*]
`gaussian-random` &optional *min max*
[*Function*]
`invert-matrix`
[*Function*]
`transpose`

[*Function*]
`conjugate-transpose`
[*Function*]
`eigenvalues`
[*Function*]
`eigenvectors`

Chapter 16

Characters

Chapter 17

Sequences

17.1 Basic Sequence Functions

[*Generic Function*]
`nth` *sequence index*
[*Generic Function*]
`with` *object sequence*
[*Generic Function*]
`with!` *object sequence*
[*Generic Function*]
`subseq` *sequence start* &optional *end*
[*Generic Function*]
`copy-seq` *sequence*
[*Generic Function*]
`length` *sequence*
[*Generic Function*]
`reverse` *sequence*
[*Generic Function*]
`reverse!` *sequence*
[*Generic Function*]
`make-sequence` *type size* &key *initial-element*
[*Generic Function*]
`union`
[*Generic Function*]
`union!`
[*Generic Function*]
`intersection`
[*Generic Function*]
`intersection!`
[*Generic Function*]
`set-difference`
[*Generic Function*]
`set-difference!`
[*Function*]
`complement` *function*
[*Generic Function*]
`flatten` *sequence*

17.2 Advanced Sequence Functions

All of the pure advanced sequence generic functions (save `map`) should have an implementation where the *sequence* arguments are of type `sequence`. They may optionally provide implementations for sub-classes of `sequence` in the interests of efficiency.

[*Generic Function*]
`map` *result-type function sequence* &rest *more-sequences*
[*Generic Function*]
`map!` *result-sequence function* &rest *sequences*
[*Generic Function*]
`append`*result-type* &rest *sequences*
[*Function*]
`mappend`*result-type function sequence* &rest *more-seqences*
[*Function*]
`some` *predicate sequence* &rest *more-sequences*
[*Function*]
`every` *predicate sequence* &rest *more-seqences*
[*Function*]
`notany` *predicate sequence* &rest *more-sequences*
[*Function*]
`notevery` *predicate sequence* &rest *more-sequences*
[*Function*]
`reduce` *function sequence* &key `:from-end :start :end :initial-value :key`
[*Generic Function*]
`map-reduce`
[*Generic Function*]
`fill` *sequence item* &key `:start :end`
[*Generic Function*]
`fill!` *sequence item* &key `:start :end`
[*Generic Function*]
`replace` *sequence1 sequence2* &key `:start1 :end1 :start2 :end2`
[*Generic Function*]
`replace!` *sequence1 sequence2* &key `:start1 :end1 :start2 :end2`
[*Generic Function*]
`remove` *item sequence* &key `:from-end :test :start :end :count :key`
[*Generic Function*]
`remove-if` *predicate sequence* &key `:from-end :start :end :count :key`
[*Generic Function*]
`remove-duplicates` *sequence* &key `:from-end :test :start :end :key`
[*Generic Function*]
`remove!` *item sequence* &key `:from-end :test :start :end :count :key`
[*Generic Function*]
`remove-if!` *predicate sequence* &key `:from-end :start :end :count :key`
[*Generic Function*]
`filter` *item sequence* &key `:from-end :test :start :end :count :key`
[*Generic Function*]
`filter-if` *predicate sequence* &key `:from-end :start :end :count :key`
[*Generic Function*]
`filter!` *item sequence* &key `:from-end :test :start :end :count :key`
[*Generic Function*]
`filter-if!` *predicate sequence* &key `:from-end :start :end :count :key`
[*Generic Function*]

`partition` *item sequence* &key `:from-end :test :start :end :count :key`
[*Generic Function*]
`partition-if` *predicate sequence* &key `:from-end :start :end :count :key`
[*Generic Function*]
`remove-duplicates!` *sequence* &key `:from-end :test :start :end :key`
[*Generic Function*]
`substitute` *newitem olditem sequence* &key `:from-end :test :start :end :count :key`
[*Generic Function*]
`substitute-if` *new-item test sequence* &key `:from-end :start :end :count :key`
[*Generic Function*]
`substitute!` *newitem olditem sequence* &key `:from-end :test :start :end :count :key`
[*Generic Function*]
`substitute-if!` *new-item test sequence* &key `:from-end :start :end :count :key`

17.3 Searching Sequences

[*Generic Function*]
`member`
[*Generic Function*]
`member-if`
[*Generic Function*]
`find` *item sequence* &key `:from-end :test :test-not`
[*Generic Function*]
`find-if` *predicate sequence* &key `:from-end :start :end :key`
[*Generic Function*]
`position` *item sequence* &key `:from-end :test :test-not :start :end :key`
[*Generic Function*]
`position-if` *predicate sequence* &key `:from-end :start :end :key`
[*Generic Function*]
`count` *item sequence* &key `:from-end :test :test-not :start :end :key`
[*Generic Function*]
`count-if` *predicate sequence* &key `:from-end :start :end :key`
[*Generic Function*]
`mismatch` *sequence1 sequence2* &key `:from-end :test :test-not :key :start1 :start2 :end1 :end2`
[*Generic Function*]
`search` *sequence1 sequence2* &key `:from-end :test :test-not :key :start1 :start2 :end1 :end2`

17.4 Sorting and Merging

[*Generic Function*]
`sort` *sequence predicate* &key `:key`
[*Generic Function*]
`stable-sort` *sequence predicate* &key `:key`
[*Generic Function*]
`sort!` *sequence predicate* &key `:key`
[*Generic Function*]
`stable-sort!` *sequence predicate* &key `:key`
[*Generic Function*]
`merge` *result-type sequence1 sequence2 predicate* &key *:key*

[*Generic Function*]
`merge!` *result-type sequence1 sequence2 predicate* &key *:key*

Chapter 18

Lists

Lists are the primary–and the first–data structure of Lisp. A list is a chain of *conses* (which are data structures with two fields, the *car* and the *cdr*)–linked to each other via *cdr*s.

18.1 Conses

[*Function*]
`car` *list*
[*Function*]
`cdr` *list*
[*Function*]
`caar` *list*
[*Function*]
`cadr` *list*
[*Function*]
`cdar` *list*
[*Function*]
`cddr` *list*
[*Function*]
`caaar` *list*
[*Function*]
`caadr` *list*
[*Function*]
`cadar` *list*
[*Function*]
`caddr` *list*
[*Function*]
`cdaar` *list*
[*Function*]
`cdadr` *list*
[*Function*]
`cddar` *list*
[*Function*]
`cdddr` *list*
[*Function*]
`caaaar` *list*
[*Function*]

caaadr *list*
[*Function*]
caadar *list*
[*Function*]
caaddr *list*
[*Function*]
cadaar *list*
[*Function*]
cadadr *list*
[*Function*]
caddar *list*
[*Function*]
cadddr *list*
[*Function*]
cdaaar *list*
[*Function*]
cdaadr *list*
[*Function*]
cdadar *list*
[*Function*]
cdaddr *list*
[*Function*]
cddaar *list*
[*Function*]
cddadr *list*
[*Function*]
cdddar *list*
[*Function*]
cddddr *list*
[*Function*]
cons x y
[*Function*]
tree-equal x y

18.2 Lists

[*Function*]
end?
[*Method*]
length
[*Method*]
nth
[*Function*]
first
[*Function*]
second
[*Function*]
third
[*Function*]
fourth

[*Function*]
fifth
[*Function*]
sixth
[*Function*]
seventh
[*Function*]
eighth
[*Function*]
ninth
[*Function*]
tenth
[*Function*]
rest
[*Function*]
nthcdr
[*Method*]
last
[*Function*]
list
[*Function*]
list*
[*Function*]
list? *object*
[*Function*]
make-list
[*Method*]
append
[*Function*]
copy-list
[*Function*]
copy-tree
[*Method*]
revappend
[*Method*]
append!
[*Method*]
revappend!
[*Macro*]
push!
[*Macro*]
pushnew!
[*Macro*]
pop!
[*Function*]
butlast
[*Function*]
nbutlast
[*Function*]
ldiff
[*Method*]
subst

[*Method*]
`subst-if`
[*Method*]
`subst!`
[*Method*]
`subst-if!`
[*Function*]
`sublis`
[*Method*]
`member`
[*Method*]
`member-if`
[*Function*]
`tail?`
[*Function*]
`citcular-list?`

18.3 Association Lists

[*Function*]
`acons`
[*Function*]
`pairlis`
[*Method*]
`assoc`
[*Method*]
`assoc-if`
[*Method*]
`rassoc`
[*Method*]
`rassoc-if`
[*Function*]
`copy-alist`

18.4 Mapping

[*Function*]
`maplist` *function list* &rest *more-lists*

Chapter 19

Sets

[*Function*]
`make-set`
[*Function*]
`adjoin`
[*Function*]
`less`
[*Function*]
`empty-set`
[*Function*]
`subset?`
[*Function*]
`disjoint?`

Chapter 20

Hash Tables

Hash tables in Koioslisp, like in Haskell and Clojure, can be operated similar to linked lists, specifically, association lists; as opposed to the support in Common Lisp, where most of the functions are destructive. Unlike in Common Lisp, hash tables in Koioslisp, hash tables have standard, `read`able, printed representations, borrowed from Racket and Arc. For example, a plausible hash-table is provided

```
#hash((vladimir . estragon) (rosencrantz . guildenstern))
```

In this hash-table, the *symbol* `vladimir` is mapped to the symbol `estragon`, and the symbol `rosencrantz` is mapped to the symbol `guildenstern`. If the hash table had an equality test of `eq?`, then it would be notated as

```
#hash(:eq? (vladimir . estragon)(rosencrantz . guildenstern))
```

[*Function*]

`make-hash-table` &key *initial-contents*

[*Function*]

`empty-hash`

[*Function*]

`hash-table?` *object*

[*Method*]

`assoc` *item table* &key *key test*

[*Method*]

`assoc-if` *predicate table* &key *key*

[*Method*]

`rassoc` *item table* &key *key test test-not*

[*Method*]

`rassoc-if` *predicate table* &key *key*

[*Method*]

`pair` *keys values* &optional *table*

[*Method*]

`remove` *key table*

[*Method*]

`remove!` *key table*

[*Method*]

`reduce`

[*Function*]

`h-cons` *key value table*

`h-cons` takes a key, a value, and a hash table (respectively) as its arguments, and returns a new hash-table with the contents of *table* and a new association between *key* and *value*. If *key* is already contained (as in, the keys are `=`) as a key in *table*, the table returned will use the value *value* instead of the value mapped to *key* in table.

`(h-cons 'foo 'bar baz)` is equivalent to `(pair 'foo 'bar baz)`

[*Function*]
`h-append` *table1 table2* &key *test*
`h-append` is like `h-cons`, except that it takes two hash-tables as arguments instead of a key, a value, and a hash-table. It returns a new hash-table with the contents of both the first table and the second table. If the second hash-table contains an entry with the same (`=`) key as the first hash-table, then the entry from the second hash-table is used. If the optional parameter *new-table-test* (which may be the values EQ or =) is supplied, the hash-table returned has membership specified by the given parameter as a test.
[*Function*]
`h-push!` *key value table*
`h-push` is the destructive counterpart to `h-cons`, modifying *table* as a result. `(h-push 'foo 'bar baz)` is equivalent to `(zap #'(λ (x) (h-cons 'foo 'bar x)) baz)`.
[*Function*]
`h-append!` *table1 table 2*
`h-append!` is the destructive counterpart to `h-append`, modifying *table1* as a result. `(h-append! foo baz)` is equivalent to `(zap #'(λ (x) (h-append x bar)) foo)`, unless *table1* uses a different test than the default (such as =, which will be used for this example), in which case it would be `(zap #'(λ (x) (h-append x bar :test '=)) foo)`
[*Method*]
`map-keys` *result-class function hash-table* &rest *more-sequences*
This is the method of map specialized for *hash-tables*. It behaves like the rest of the methods of *map*,except that the function is called on the set of keys in the hash-table rather than both the keys and the values.
[*Function*]
`maphash` *function hash-table*
The *maphash* function maps over every entry of the hash-table *hash-table*; calling the function *function* with the arguments being the key and the value of the entry; then it returns a list of the results of the evaluation of the function over the entries. In this it is incompatible with Common Lisp's `maphash` function, which unconditionally returns `nil`. If the entries in *hash-table* are destructively modified by *function*, the results are unpredictable.
[*Macro*]
`with-hash-table-iterator`
[*Method*]
`member`
[*Method*]
`member-if`
[*Method*]
`sort` *table*
[*Method*]
`sort!` *table*
[*Method*]
`clear`
[*Function*]
`sxhash` *object*
[*Method*]
`get-in` *table* &rest *keys*
[*Function*]
`keys` *table*
[*Function*]
`hash-values`*table*
[*Method*]
`length`
[*Function*]
`merge-with`

Chapter 21

Arrays

[*Function*]
`make-array`
[*Constant*]
`+array-rank-limit+`
[*Constant*]
`+array-dimension-limit+`
[*Constant*]
`+array-total-size-limit+`
[*Function*]
`aref`
[*Function*]
`array-element-type`
[*Function*]
`array-rank`
[*Function*]
`array-dimension`
[*Function*]
`array-dimensions`
[*Function*]
`array-total-size`
[*Function*]
`array-in-bounds-p`
[*Function*]
`array-row-major-index`
[*Function*]
`row-major-aref`
[*Function*]
`adjustable-array-p`

21.1 Vectors

What Common Lisp called `vector-push-extend` has been rechristened `vector-push`; Common Lisp's `vector-push` (a less-useful case of `vector-push-extend`) has been removed from the language.

Chapter 22

Tuples

In KOIOSLISP, tuples are designed to potentially produce particularly efficient parallel code. They are modelled, in part, after the "conc lists" of the Fortress programing language.

[*Function*]
`tuple`
[*Function*]
`tuple?`
[*Function*]
`conc`
[*Method*]
`empty?`
[*Function*]
`singleton?`
[*Function*]
`item`
[*Function*]
`left`
[*Function*]
`right`
[*Function*]
`rebalance`
[*Function*]
`add-left`
[*Function*]
`add-right`
[*Function*]
`split`
[*Function*]
`divide-and-conc`

Chapter 23

Strings

A string is a sequence of characters. [*Generic Function*]
`str`

23.1 String Pattern Matching

Koioslisp supports pattern matching both at the string level (here), and the tree level (the "Pattern Matching" chapter), with the inspiration for the string-based pattern matching being SNOBOL.
[*Function*]
`match`
[*Function*]
`replace`
[*Function*]
`replace!`
[*Function*]
`replace*`
[*Function*]
`replace*!`
[*Function*]
`make-pattern`
[*Function*]
`str-span`
[*Function*]
`str-break`
[*Function*]
`l-pos`
[*Function*]
`r-pos`
[*Function*]
`str-or`
[*Function*]
`str-not`
[*Function*]
`match-len`
[*Function*]
`all-matches`

[*Function*]
`str-fail!`
[*Function*]
`arbstr`
[*Function*]
`arb-num-str`

23.1.1 Regular Expressions

Another, often quicker, method for recognizing patterns in strings is through the use of *regular expressions*.
[*Function*]
`splitre`

Chapter 24

The Evaluator

Chapter 25

Lazy Evaluation

Lazy evaluation is a method for efficiently manipulating large (even theoretically infinite) data structures. Though usually evaluation is eager, Koioslisp provides support for lazy evaluation with explicit annotation. The Koioslisp lazy evaluation facility is partially based on the CLAZY library for Common Lisp, and the Clojure lazy evaluation facility. See also the section on List Comprehension.

[*Macro*]
`deflazy`
[*Function*]
`lazycall`
[*Macro*]
`lazy`
[*Macro*]
`lazylambda`
[*Macro*]
`lazylet`
[*Function*]
`force`

25.1 Series

Series are sequences, and thus all sequence generic functions should "work" on them, in manners defined here. Generally they are to be defined lazily; that is, to generate a result, one must `take` (or `take-all`) elements from them. [*Method*]
`take`
Unlike `take-all`, `take` is useful for large sequences in general–thus it has been promoted to a sequence generic function.
[*Function*]
`take-all`
[*Function*]
`drop`
[*Function*]
`head`
[*Function*]
`tail`

Chapter 26

Continuations

Like Scheme and Qi, KIOSLISP provides continuations, which are mechanisms for abstract control over the flow of execution of a program. Unlike Scheme or Qi, these are *delimited continuations*

[*Function*]
`call/cc` *function*
The argument *function* is a function which takes one parameter. CALL/CC can only be called within a WITH-CALL/CC form or an implicit WITH-CALL/CC form. CALL/CC has the alias CALL-WITH-CURRENT-CONTINUATION
[*Macro*]
`with-call/cc`
[*Macro*]
`let/cc`
[*Macro*]
`lambda/cc`
[*Macro*]
`defun/cc`
[*Macro*]
`without-call/cc`
[*Macro*]
`defmethod/cc`
[*Macro*]
`defgeneric/cc`

Chapter 27

Logic Programming

27.1 Constraints

The features described in this section are based on the Common Lisp library Screamer, which adds constraint-programming constructs to Common Lisp.

27.1.1 Nondeterminism

A nondeterministic context is:
1. The body of a function defined with `def-nondet-fun`.
2. The body of a `for-effects`, `all-values`, `print-values`, `possibly?`, or `necessary?` form
3. The first argument of a `one-value` form
4. The second argument of a `ith-value` form

[*Macro*]
`either` &body *alternatives*
[*Function*]
`fail`
[*Function*]
`trail` *function*
[*Macro*]
`all-values` &body *body*
[*Macro*]
`one-value` *form* &optional *default*
[*Macro*]
`for-effects` &body *body*
[*Macro*]
`ith-value` *i form* &optional *default*
[*Macro*]
`print-values` &body *body*
[*Macro*]
`possibly?` &body *body*
[*Macro*]
`necessarily?` &body *body*

Some macros have been provided to facilitate the creation of non-deterministic functions and generic function, as they cannot, by definition, be called in a deterministic context.
[*Macro*]
`def-nondet-fun`
[*Macro*]

def-nondet-generic
[*Macro*]
def-nondet-method
[*Function*]
a-boolean
[*Generic Function*]
a-member-of *sequence*
[*Function*]
an-integer
[*Function*]
an-integer-above *low*
[*Function*]
an-integer-below *high*
[*Function*]
an-integer-between *low high*
[*Function*]
nondet? *x*
[*Function*]
nondet-generic? *x*
[*Macro*]
when-failing (&body *failing-forms*) &body *body*
[*Macro*]
count-failures &body *body*

27.1.2 The Creation and Manipulation of Constraint Variables

[*Macro*]
assert! *v*
[*Function*]
of-typev *type* &optional *name*
[*Function*]
value-of *x*
[*Function*]
apply-substitution *x*
[*Function*]
bound? *x*
[*Function*]
ground? *x*
[*Function*]
applyv *f x* &rest *more-vars*
[*Function*]
funcallv *f* &rest *vars*
[*Function*]
equalv *x y*
[*Function*]
template *template*

[*Function*]
make-logvar &optional *name*

[*Function*]
`unify` *x y*
`(unify x y)` is syntactic sugar for `(assertv (equalv x y))`

Sequences

[*Function*]
`a-member-ofv` *vals* &optional *name*
[*Function*]
`memberv` *x sequence*

Booleans

[*Function*]
`a-booleanv` &optional *name*
[*Function*]
`booleanpv` *v*
[*Macro*]
`knownp` *x*
[*Macro*]
`decide` *x*
[*Function*]
`notv` *x*
[*Function*]
`andv` &rest *vars*
[*Function*]
`orv` &rest *vars*
[*Function*]
`count-truesv` &rest *vars*

Numbers

[*Function*]
`a-numberv` &optional *name*
[*Function*]
`a-realv` &optional *name*
[*Function*]
`a-real-abovev` *low* &optional *name*
[*Function*]
`a-real-belowv` *high* &optional *name*
[*Function*]
`a-real-betweenv` *low high* &optional *name*
[*Function*]
`an-integerv` &optional *name*
[*Function*]
`a-integer-abovev` *low* &optional *name*
[*Function*]
`a-integer-belowv` *high* &optional *name*
[*Function*]
`a-integer-betweenv` *low high* &optional *name*

[*Function*]
`numberv?` *x*
[*Function*]
`realv?` *x*
[*Function*]
`integerv?` *x*
[*Function*]
`minv` *x* &rest *more-vars*
[*Function*]
`maxv` *x* &rest *more-vars*
[*Function*]
`+v` &rest *vars*
[*Function*]
`-v` *x* &rest *vars*
[*Function*]
`*v` &rest *vars*
[*Function*]
`/v` *x* &rest *vars*
[*Function*]
`<v` *x* &rest *vars*
[*Function*]
`<=v` *x* &rest *vars*
[*Function*]
`>=v` *x* &rest *vars*
[*Function*]
`>v` *x* &rest *vars*
[*Function*]
`/=v` *x* &rest *vars*

27.1.3 Forcing Solutions

[*Function*]
`solution` *arguments ordering-force-function*
[*Function*]
`static-ordering` *force-function*
[*Function*]
`reorder` *cost-function terminate-p order force-function*
[*Function*]
`linear-force` *x*

27.1.4 Side Effects in Constraint Programming

[*Macro*]
`globalc` &body *body*
[*Macro*]
`localc` &body *body*

27.1.5 Relations

For more convenience, KOIOSLISP also provides *relations*, allowing a more Prolog-esque style

Chapter 28

Sockets

Chapter 29

Streams

Chapter 30

Objects

The Koioslisp Object System is based on CLOS (Common Lisp Object System) and TELOS (The EuLisp Object System), and provides reasonably better integration with the rest of Koioslisp than CLOS does with Common Lisp.

30.1 Classes

30.1.1 Defining Classes

30.1.2 Instance Creation

30.1.3 Slots

30.1.4 Accessing Slots

30.1.5 Inheritance

Inheritance of Methods

Inheritance of Slots

Examples

30.1.6 Built-in Classes

30.1.7 Determining the Class Precedence List

Sorting the Class Precedence Lists

Examples

30.2 Generic Functions and Methods

30.2.1 Introduction

30.2.2 Agreement on Parameter Specializers and Qualifiers

30.2.3 Keyword Arguments

30.2.4 Method Selection

30.2.5 Determinating the Effective Method

30.2.6 Standard Method Combination

30.2.7 Other Method Combinations

30.2.8 Defining Method Combinations

30.2.9 SET methods

30.3 Structures

30.3.1 Defstruct Options

30.3.2 BOA Constructors

30.4 The Meta-Object Protocol

Chapter 31

Input/Output

```
(defvar *country* ``Eurasia'')
(format t ``Oceania has always been at war with ~A'' *country*)
;→ ``Oceania has always been at war with Eurasia''
```

31.1 Readtables

KOIOSLISP adopts the mechanism of *named readtables*. There are two preregistered readtables: `:koioslisp` and `:current`: they refer to standard KOIOSLISP syntax and the current readtable respectively. [*Macro*]
`defreadtable` *name* &body *options*
`defreadtable`, as indicated by its name, defines a new readtable with its name the symbol *name*, unless *name* is already used for another readtable, in which case it is redefined. `defreadtable` takes several options:
(`:merge` *readtables+*)
(`:merge-overwriting` *readtables+*)
(`:dispatch-macro-char` *macro-char function*)
(`:macro-char` *macro-char function*
[*Function*]
`copy-readtable`
[*Function*]
`ensure-readtable`
[*Function*]
`find-readtable` *name*
Looks for a readtable of name *name* (which is of type `(or readtable symbol)`), and returns it if it is found, or `nil` otherwise.

Chapter 32

File System Interface

Often, programs are packaged in *files*. Because of their use, Koioslisp provides a standard method for interacting with a filesystem. The system is designed with its basis on the Common Lisp filesystem interface because of its generality and ability to handle most, if not all, forms of filesystems. These are useful features because of the historical and current variation in file systems and their conventions in representing files. As an example of such variation, a table of different methods to represent a certain file on different filesystems important (both currently and historically) in the Lisp world is given:

System	File name
TOPS-20	⟨LISPIO⟩FORMAT.FASL.13
TOPS-10	FORMAT.FAS[1,4]
ITS	LISPIO;FORMAT FASL
MULTICS	⟩udd⟩LispIO⟩format.fasl
TENEX	⟨LISPIO⟩FORMAT.FASL;13
VAX/VMS	[LISPIO]FORMAT.FAS;13
Unix, GNU, *BSD, Plan 9, Mac OS X, etc.	/usr/LispIO/format.fasl
MS-DOS, Windows, ReactOS, etc.	C:\Program Files\LispIO\format.fasl

This large proliferation of differing filesystems makes optimizing to one specific type in the core language rather arbitrary and shortsighted. Thus, Koioslisp uses a very general method for dealing with filesystems; namely, adapting the Common Lisp system–providing *namestrings* (which are strings in the customary file system-representing form) and *pathnames* (which are abstract methods of dealing with files) that work in a very similar manner to their predecessors.

32.1 File Names

32.1.1 Pathnames

Filesystems in Koioslisp are seen through the framework of the *pathname* data object, with its components defined below. This is used in order to enhance portability over file systems.

host

A *host* is the name of the file system which contains the file

device
The "file structure", or "device" component of file systems
directory
The "directory" component of most file systems; a structure containing a group of possibly related files
name
The label associated with a file
type The "extension"or "filetype" of a file system–quite literally the type of file, which determines which programs are to handle it.
version
An increasingly rare component; a number which represents which revision the file is in, and how many revisions it has undergone

32.1.2 Case Conventions

32.1.3 Structured Directories

32.1.4 Extended Wildcards and Globbing

32.1.5 Logical Pathnames

Syntax of Logical Pathnames

Parsing of Logical Pathname Namestrings

Using Logical Pathnames

Logical Pathname Examples

32.1.6 Pathname Functions

32.2 File and Directory Operations

32.2.1 File I/O

[*Function*]
`read-file-into-string`
[*Function*]
`rewind`

Loading files

32.2.2 Renaming, Deleting, and Other FIle Operations

32.2.3 Accessing Directories

Chapter 33

Errors

Chapter 34

Advice

[*Macro*]
`defadvice` (*function advice-name time* &key *where documentation*) (*lambda-list*) &body *body*

Chapter 35

Miscellaneous Features

35.1 The Compiler

[*Function*]
`compile`
[*Function*]
`compile-file`
[*Variable*]
`*compile-verbose*`
[*Variable*]
`*compile-print*`
[*Variable*]
`*compile-file-pathname*`
[*Variable*]
`*compile-file-truename*`
[*Special form*]
`load-time-value`
[*Function*]
`disassemble`
[*Function*]
`function-lambda-expression`
[*Macro*]
`with-compilation-unit`

35.1.1 Compiler Diagnostics

35.1.2 Compiled Functions

35.1.3 Compilation Environment

35.1.4 Similarity of Constants

35.2 Documentation

[*Generic Function*]
`documentation`
[*Generic Function*]

apropos

35.3 Debugging Tools

[*Macro*]
trace
[*Macro*]
untrace
[*Macro*]
step
[*Macro*]
time
[*Function*]
describe
[*Generic Function*]
describe-object
[*Method*]
describe-object
[*Function*]
inspect
[*Function*]
room
[*Function*]
ed
[*Function*]
dribble

35.4 Environment

Here there is significant influence from Erik Naggum's paper "A Long, Painful History of Time "
[*Class*]
<local-time>
Superclasses: <standard-object>
Slots:
[*Function*]
make-local-time &key *universal internal unix* (msec 0) (zone 0)
[*Function*]
local-time-day
[*Function*]
local-time-sec
[*Function*]
local-time-msec
[*Function*]
local-time-zone
[*Function*]
local-time<
[*Function*]
local-time<=
[*Function*]

`local-time>`
[*Function*]
`local-time>=`
[*Function*]
`local-time=`
[*Function*]
`local-time/=`
[*Function*]
`local-time-adjust` *source timezone* &optional *destination*
`local-time-adjust` returns the values of the adjusted slots unless *destination* is provided as a `local-time` instance, in which case it will first adjust the slots and then return the *destination* `local-time` instance
[*Function*]
`get-local-time`
[*Function*]
`get-french-time`
This function returns the current time in the French Republican Calendar. The values it returns are: the year, the month, the day of the month, the hour, minute, and second. The current year is the number of years since the establishment of the calendar (in the Gregorian Calendar, 22 September 1792 C.E.; the New Year is thus set at 22 September in the Koioslisp implementation, though in actual usage it varied depending on the Autumn Equinox). The month is one of the following values: "Vendémiaire", "Brumaire", "Frimaire", "Nivôse", "Pluviôse", "Ventôse", "Germinal", "Floréal", "Prairial", "Messidor", "Thermidor", "Fructidor", or "Fêtes"; each month is 30 days long, except for the Fêtes, which are 5 days long on normal years, 6 on leap years. A leap year occurs with the same rule as in the Gregorian calendar, but the origin date is the year 20.
[*Function*]
`get-formatted-french-time` &optional *timestamp*
This function returns a formatted string representing the current French Republican Calendar time, or, if supplied a timestamp argument, formats the time represented by the timestamp. The format is specified as: *time* ::= ⟦*date-of-month month* | *fête* ⟧An *year* de la Révolution, *hour*:*minute*:*second*
[*Function*]
`timestamp->french`
[*Function*]
`sleep` *seconds*
[*Function*]
`syscall`
[*Function*]
`lisp-implementation-type`
Examples: "Seattle Koioslisp", "Alonzo".
[*Function*]
`lisp-implementation-version`
[*Function*]
`os-type`
Returns a list of symbols which identify the operating system upon which Koioslisp runs–a list is chosen, of course, because some OSes implement a larger standard, or are part of a family that can be treated as a single unit. Some are provided here as standards for common systems:
`:gnu` The GNU system, often erroneously called Linux when paired with said kernel. Matches GNU/Linux systems, GNU/kFreeBSD systems, GNU/HURD systems, and any other GNU systems.
`:bsd` The BSD family of operating systems.
`:posix` The POSIX standard for operating systems.
`:loper` The Loper Operating System
`:plan-9` The Plan 9 Operating System
`:os-x` The Mac OS X operating system

`:windows` The Microsoft Windows operating system, and the React Operating System

[*Function*]
`os-version`
[*Function*]
`software-type`
[*Variable*]
`*features*`
[*Function*]
`identity` *object*

Chapter 36

Pretty-Printing

Chapter 37

Concurrency

[*Function*]
`spawn`
[*Macro*]
`recieve`
[*Recieve option*]
`after`
[*Function*]
`send`

[*Function*]
`exit-proc`
[*Function*]
`spawn-remotely`
[*Function*]
`atomic`
[*Atomic anaphor*]
`retry`
[*Macro*]
`or-else` &rest *args*

37.1 Refs

A *ref* is like a Concurrent Haskell *TVar*; it is used for mutable shared state in concurrent programs. [*Function*]
`ref`
[*Macro*]
`defref`
`(defref x y)` is syntactic sugar for `(defvar x (ref y))`
[*Macro*]
`getref`

37.2 The Dining Philosophers Problem

```
(defclass <chopstick> ()
    ((owner :accessor chopstick-owner :initform (ref nil))))
(defvar *chopsticks*
    '((make-instance '<chopstick>)
    (make-instance '<chopstick>)
    (make-instance '<chopstick>)
    (make-instance '<chopstick>)
    (make-instance '<chopstick>)
    (make-instance '<chopstick>)
    (make-instance '<chopstick>)
    (make-instance '<chopstick>)
    (make-instance '<chopstick>)))
(defclass <philosopher> (<thread>)
    ((name :accessor name-of :initarg name)
    (meals-eaten :accessor phil-meals :initform (ref 0))
    (left :accessor phil-left :initarg left)
    (right :accessor phil-right :initarg right)))
(defvar *philosophers*
    '((make-instance '<philosopher> :name ''Chomsky'' :left (nth *chopsticks* 0) :right (nth *chopsticks* 1))
    (make-instance '<philosopher> :name ''Kongfuzi'' :left (nth *chopsticks* 1) :right (nth *chopsticks* 2))
    (make-instance '<philosopher> :name ''Nietzsche'' :left (nth *chopsticks* 2) :right (nth *chopsticks* 3))
    (make-instance '<philosopher> :name ''Marx'' :left (nth *chopsticks* 3) :right (nth *chopsticks* 4))
    (make-instance '<philosopher> :name ''Spinoza'' :left (nth *chopsticks* 4) :right (nth *chopsticks* 5))
    (make-instance '<philosopher> :name ''Wittgenstein'' :left (nth *chopsticks* 5) :right (nth *chopsticks* 6))
    (make-instance '<philosopher> :name ''de Beauvoir'' :left (nth *chopsticks* 6) :right (nth *chopsticks* 7))
    (make-instance '<philosopher> :name ''Dostoyevsky'' :left (nth *chopsticks* 7) :right (nth *chopsticks* 8))
    (make-instance '<philosopher> :name ''de Saussure'' :left (nth *chopsticks* 8) :right (nth *chopsticks* 0))))
(defmethod run ((thread <philosopher>))
    (until (=? (phil-meals philosopher) 30)
       (atomic
          (if (not
             (and (null? (chopstick-owner (phil-left philosopher)))
                  (null? (chopstick-owner (phil-right philosopher)))))
             (retry)
             (begin
                  (set! (get-ref (chopstick-owner (phil-left philosopher))) (name-of philosopher))
                  (set! (get-ref (chopstick-owner (phil-right philosopher))) (name-of philosopher))
                  (format t ''~A has acquired two chopsticks'' (name-of philosopher)))))
       (atomic
          (inc (get-ref (phil-meals philosopher)))
          (format t ''~A has eaten a tasty bite of rice'' (name-of philosopher))
          (set! (get-ref (chopstick-owner (phil-left philosopher))) nil)
          (set! (get-ref (chopstick-owner (phil-right philosopher))) nil)
          (format t ''~A has returned to thinking'' (name-of philosopher))))
    (format t ''~A has left the table'' (name-of philosopher)))
(mapc #'spawn *philosophers*)
```

Chapter 38

Backquote

Chapter 39

Implementation Notes

Some notes as to how to implement this are located here, however, the notes speak from a high-level, design perspective.Koioslisp's syntax and semantics have been designed in such a way that if one has a modern, working implementation of Common Lisp (such as Steel Bank Common Lisp (SBCL)), one can adapt it with little effort to implement much of the Koioslisp standard library. An interesting perspective on implementation, proposed by Richard P. Gabriel in his paper "Lisp: Good News, Bad News, How to Win Big", and followed by the experimental Lisp dialect Shen, is to create a kernel language small enough to implement easily, yet powerful enough to express the language–and implement the rest of the language as a standard library for the kernel (Shen's kernel language is called Kernel Lisp, or 'Kl', which, like Koioslisp, has used '.kl' as an extension. Koioslisp is not compatible with Kernel Lisp, and has no relation to it whatsoever). Another solution would be to implement it as an embedded language within Common Lisp. Whatever the solution turns out to be, we hope that these suggestions will prompt quicker implementation of Koioslisp.

Chapter 40

Batteries--Utility Libraries

Many of these "batteries" are connected to the outside world in such a way that if something were to drastically change, they would be obsoleted and will be updated. They are suggested, but not required, to be included in implementations of KOIOSLISP as standard utility libraries–this is simply a description of a standard "way" to do the things described here. Some more complex utilities, such as a BitTorrent library, are not specified here as standard libraries but are encouraged to be developed by KOIOSLISP users.

40.1 Logging

Standard logging levels include, from least to most severe, `+dribble+`, `+debug+`, `+info+`, `+warn+`, `+error+`, and `+fatal+`

40.2 Data Compression

40.3 Foreign Function Interfaces

40.4 OS Services and Adaptations

40.4.1 UNIX-specific

40.5 Cryptography

40.6 Internet Data Handling

40.7 Markup Language Processing Tools

40.8 Internet Protocols

40.9 Multimedia Services

40.10 GUIs

40.10.1 KLIM

See the KLIM manual.

Chapter 41

Community Things

These are (probably) not to be implemented as part of the language. Rather, they are services designed for the Koioslisp community. In fact, with some of them (such as Lake or LispDoc), there is encouraged to

41.1 CKLAN

CKLAN is a proposed common repository for the hosting of useful software libre (FLOSS) KOIOSLISP libraries, programs, and scripts. It is inspired by the CPAN (Comprehensive Perl Archive Network), which does the same for the Perl programming language.

41.2 Lake

Lake is the Koioslisp build language, similar to Unix make, Apache Ant, and Ruby Rake. It is also the interpreter for such a program

41.3 LispDoc

LispDoc generates documentation (in the form of LaTeXor HTML) based on docstrings in a given selection of Koioslisp code. It is roughly similar to JavaDoc and RDoc in this respect.

41.4 ASDF-like tools

Chapter 42

References