

Arizona Overview and Notes for Release 2.0

JOHN ALAN McDONALD

MICHAEL SANNELLA *

Depts. of Statistics, Computer Science and Engineering,
University of Washington

August 1993

Abstract

This report gives an overview of the Arizona system and is also a reference manual for the Az package.

*This work was supported in part the Dept. of Energy under contract FG0685-ER25006 and by NIH grant LM04174 from the National Library of Medicine.

Contents

1 Overview	3
1.1 Existing Modules	3
1.1.1 Arizona-Tools	4
1.1.2 Definitions	4
1.1.3 Announcements	4
1.1.4 Actors	5
1.1.5 Geometry	5
1.1.6 Slate	5
1.1.7 Chart	6
1.1.8 Graph	6
1.1.9 Browser	7
1.1.10 Lisp-NPSOL Interface	7
1.1.11 Clay	7
1.1.12 Basic Math	7
1.1.13 Probability	7
1.1.14 Cactus	7
1.2 Current Ports	8
1.3 Getting Arizona	8
2 Style Conventions	8
2.1 Modules and Packages	8
2.2 Abstract Data Types	9
2.3 Immutable Object Types	10
2.4 Limiting garbage: informal Result and Resource protocols	10
2.5 Optional Results	10
2.6 Resources	10
2.7 Miscellaneous	12
2.8 Style Conventions for CLOS	13
3 Assorted Common Lisp Tools	13
3.1 System Compiling and Loading Tools	13
3.2 Runtime type checking	13
3.3 Variations on <code>setf</code>	13
3.4 Iteration tools	14
3.5 Macro Writing Tools	14
3.6 Bug report tools	14
3.7 Program monitoring	14
3.8 Copy protocol	14
3.9 Kill protocol	15
3.10 Resource Protocol	15
4 Reference Manual	16

1 Overview

This document provides an introduction to a system called Arizona, now under development at the U. of Washington. It consists of this overview section, a section outlining suggested style conventions to be followed by contributors to Arizona, followed by a sections describing the most basic module in Arizona, called Arizona-Tools.

Arizona is intended to be a portable, public-domain collection of tools supporting scientific computing, quantitative graphics, and data analysis, implemented in Common Lisp and CLOS (the Common Lisp Object System) [6, 12, 19, 42, 49, 50, 55, 56].

Discussion of the philosophy underlying Arizona can be found in [35, 36, 23, 24, 37, 52, 51]. Briefly, the design is motivated by our belief that an ideal system for scientific computing and data analysis should have:

- One language that can be used for both for line-by-line interaction or defining compiled procedures.
- Minimal overhead in adding new compiled procedures (or other definitions).
- A language that supports a wide variety of abstractions and the definition of new kinds of abstractions.
- Programming tools (editor, debugger, browsers, metering and monitoring tools).
- Automatic memory management (dynamic space allocation and garbage collection).
- Portability over many types of workstations and operating systems.
- A community of users and developers.
- Access to traditional Fortran scientific subroutine libraries or equivalents.
- A representation of scientific data directly in the data structures of the language.
- Comprehensive numerical, graphical, and statistical functionality.
- Device independent static output graphics.
- Window based interactive graphics.
- Support for efficient and concurrent access to large databases.
- Documentation and tutorials, both paper and on-line.

One reason Arizona is based on Common Lisp is that the first nine points (through “access to Fortran”) come for free with standard Common Lisp environments. The remaining six are the research aspects of Arizona.

1.1 Existing Modules

Arizona is divided into a number of modules with limited interdependencies, to permit individual modules to stabilize and be “released” and used independently. The word “module” is used here informally. The definitions in a module will usually be in a package defined for that module and the source code for those definitions will usually be found in files in a single directory.

A number of the modules in Arizona are described below. Current modules in alpha release are Arizona-Tools, Definitions, Announcements, Actors, Geometry, Slate, Chart, Graph, Browser, and an interface between Lisp and the Fortran optimization package NPSOL. Four other modules have seen some use but are likely to have more significant changes before any sort of release: Clay, Basic-Math, Probability, and Cactus.

1.1.1 Arizona-Tools

The Arizona-Tools module, described in more detail in section 3, provides assorted general purpose utilities for Common Lisp programming. Collecting these tools in one place saves users from reinventing many minor variations of the same wheel and encourages programming style consistent with the conventions suggested in section 2.

For example, Arizona-Tools defines protocols for copying, saving to file, and destroying CLOS objects, and default implementations of those protocols, which are some essential facilities for a primitive database using persistent CLOS objects.

1.1.2 Definitions

The Definitions module [27] provides the beginnings of a database for Common Lisp source code objects. The major use of this database at present is in automatically typesetting reference manuals (using Latex [20]), examples of which are the reference manual sections of [27, 27, 41, 40, 34, 28, 29] and this report. It is inspired in part by the Definition Groups of Bobrow et. al [5] and the USER-MANUAL of Kantrowitz [18].

The primary purpose of Definitions is to make possible convenient runtime access to information available in Common Lisp source code, that is lost in the normal process of reading, evaluating, and/or compiling.

Evaluating some Lisp definitions, such as `defclass`, results in a first class Lisp object with a reasonable and reasonably portable protocol for extracting useful information, such as the direct sub- and super-classes. However, most Lisp definitions, such as `defun`, while producing identifiable objects, have limited facilities for extracting useful information; usually the documentation string is all that is available. And other definitions, such as `defstruct`, do not even produce an identifiable object.

The Definitions module provides functions to read source files and create a Definition object for each lisp definition in those files, retaining the complete original defining lisp form.

1.1.3 Announcements

The Announcements module [34] provides a simple mechanism for maintaining dependencies between objects through message broadcasting, similar to the Field [47] and Forest environments. It is derived from a Common Lisp implementation of Events/Mediators [53] by Mark Niehaus for the Prism system [17]. Like Field and Forest, it can be thought of as a mechanism for *implicit invocation* as discussed in [14, 54].

The basic idea is to allow an object (the *announcer*) to notify other objects (the *audience*) of a change of state (or some other event) without the announcer having to know of which objects need to be notified of which events.

An *announcement* is defined with a call to `defannouncement`. Any announcer object can **announce** any defined announcement, which results in every member of the audience for that announcer-announcement pair being notified. Any object can join the audience for a particular announcer-announcement pair with a call to `join-audience`.

1.1.4 Actors

The Actors module [29] is primarily intended as a mechanism for interpreting and distributing asynchronous (input) events in a multiprocessing Lisp environment. It is inspired in part by the actor message passing paradigm of Carl Hewitt (as described in [1]).

An Actor is an object with a message queue; each actor handles its messages in its own, separate, thread of computation, at least conceptually in parallel with other actors and in parallel with any read-eval-print loops or other I/O processes that may exist.

Actors contain multiple Role objects; the *current role* determines how an actor handles a message at any given time. A common response to a message is to simply change the current role. An actor can be thought of as an implementation of a finite state machine for handling (input) events; each role can be thought of as a state in a state-transition diagram and the events that cause changes of role are the transitions.

Actually, a role is more accurately thought of as a fairly independent subgraph of a state-transition diagram. The advantage of encapsulating more-or-less independent chunks of behavior in role objects is that they can then be composed in different ways in various subclasses of Actor, making it easier to implement new graphical user interfaces.

One important subclass of Actor is *Interactor*, in the Slate package. Interactors are actors whose queues receive messages corresponding to the low level mouse and keyboard events received by windows.

Another important subclass is *Coordinator*, in the Clay package. Coordinators are objects that relationships between interactive diagrams and the diagrams' subject — the underlying data that the diagram represents. “Coordinator” is really just another word for “Mediator” as used in [53]. Coordinators receive messages from diagrams requesting operations to be performed on their subjects (which originate from user input) and they also receive messages (via Announcements [34]) from subjects that indicate that diagrams need to be updated to reflect changes in the state of a subject.

1.1.5 Geometry

The Geometry module [39] is intended to support common geometric calculations arising in graphics, numerical linear algebra, optimization, and scientific computing in general. The goal is to allow geometric computation using abstractions that directly represent high level mathematical concepts like affine spaces, vectors, and linear transformations, while retaining the level of performance provided by traditional scientific subroutine packages like Linpack[11]. It will be derived from Cactus [38, 25] and improved with ideas from work by DeRose [8, 9, 10] and Segal [48].

However, the current release of the Geometry module does not support general geometric calculations. It consists of two submodules specialized for high performance in simple graphics calculations, needed by the Slate and Chart modules discussed below. The Screen Geometry submodule supports calculations in a discrete two-dimensional coordinate system called Screen Space, ie. a bitmapped display. The Chart Geometry submodule supports calculations in a continuous (float) two-dimensional coordinate system, called Chart Space, a natural world space for simple scientific diagrams. In addition, the Geometry module provides affine mappings between the Chart and Screen spaces.

1.1.6 Slate

Slate [41] is a low level “device-independent” graphics package that has been ported to a number of Common Lisp platforms, window systems, and other graphics devices. It is intended to be used by developers of higher level scientific and statistical graphics systems rather than end users. Slate is

something like a simplified version of CLX, ported to run on window systems and graphics devices other than X11.

Some of the design goals of Slate are:

- It should be robust against errors in the calls from a higher level graphics system.
- It should be easy to port to a new window system, hardcopy device, or Lisp compiler.
- It should permit high performance implementations.
- Unless there's a good reason to do otherwise, the design should be as close to X as possible, because we expect the X port to be the most important.

The basic abstraction in Slate is) the *slate*. Slates are surfaces that can be drawn on and can receive input of various kinds. To make porting easy, the imaging model is fairly primitive; Slates are essentially bitmaps of some finite depth.

At present the set of drawing operations allows us to outline, fill, or tile simple geometric shapes like points, line segments, or polygons, draw characters and strings in a variety of fonts and colors, and copy rectangular sets of pixels from one slate to another. The behavior of all the drawing operations in Slate can be described in the following way: The drawing operation changes the values of a set of destination pixels in the slate. For each destination pixel, a source pixel value is calculated and the destination pixel is set to some simple function (at present some boolean combination of the bits) of the source and destination pixel values.

A typical drawing operation takes three types of arguments: a *pen*, which is an abstraction that encapsulates boolean operation, color, font, line style, etc., a slate to draw on, and a specification for the set of destination pixels. Sets of destination pixels are specified using the abstractions for discrete screen geometry provided by the Geometry package (see [39]).

Slate provides separate abstractions for the more complex pen parameters, like colors, boolean operations, fonts, line styles, and tiling patterns.

There are two basic kinds of slates, standard, visible slates and *invisible slates*. All drawing operations work “in the same way” on both visible and invisible slates. An invisible slate serves roughly the same purpose as a pixmap in X. It gives us a surface to draw on that can be copied rapidly to multiple places on one or more visible slates.

Another major abstraction in Slate is the *screen*. The screen object captures information about the device a slate is actually displayed on that isn't specific to any slate, such as, the number of bits per pixel. What color the user sees for a given pixel value is determined by the current *colormap* of the slate's screen.

1.1.7 Chart

Chart [40] is a simple example of a (slightly) higher level graphics package built on top of Slate. It is a quick and dirty approximation to S style graphics [2, 3, 4]. What we mean by “S style graphics” is output-only, line and point plots, with labels and tic marks, where locations are specified in an arbitrary 2d world coordinate system and the scaling to screen coordinates is done more or less automatically.

1.1.8 Graph

The Graph module [28] provides a simple protocol for data structures used to represent graphs (networks), graph nodes, and graph edges, and a sample implementation of that protocol.

1.1.9 Browser

The Browser module [28] provides a generic graph browser, implemented using Clay, and three examples of how the browser can be specialized to particular applications: a CLOS class hierarchy browser, a browser for graph of related CLOS objects, where the edges in the graph correspond to slot references from one object to another, and a pedigree browser.

1.1.10 Lisp-NPSOL Interface

The NPSOL package [26] provides a number of functions for calling the Fortran optimization package NPSOL [15, 16, 43, 44, 45] from Common Lisp, using the foreign function interface provided with Franz Allegro Common Lisp [13].

1.1.11 Clay

Clay [32] is the beginnings of a system designed to support drawing and interacting with 2d (and 3d) pictures for visualizing scientific data, both the observational data that's the usual domain of statisticians and the computed "data" arising in computer experiments and algorithm animation. It provides a toolkit of standard plot components and mechanisms for pasting components together to make it easy for the user to improvise new kinds of plots. All plot components obey a clearly defined protocol. This means that there is a consistent user interface and that users can reliably define new types of plot components that serve in the existing user interface.

1.1.12 Basic Math

The Basic Math module [30] consists of things that can be reasonably implemented with Common Lisp functions and primitive Common Lisp data structures; it does not use CLOS. Included in Basic Math are: machine constants, special functions (eg. beta, gamma) extended vector operations (analogous to the BLAS [21] used in Linpack [11]), evaluation and interpolation (eg. generic continued fractions) 1d numerical integration, and basic random number generators.

1.1.13 Probability

The Probability module [33] supports inference and Monte Carlo simulation (including bootstrapping) in a unified framework through a protocol for `Probability-Measure` classes. Probability measure objects are responsible for generating samples from themselves, computing their quantiles, and computing the probabilities of appropriate sets, including tail probabilities. The defined probability measure classes includes the standard one- and higher-dimensional parametric densities and discrete distributions, and non-parametric measures, either resulting from density estimates or the empirical measure of a data set. (It's worth noting that simple descriptive statistics like mean, median, etc., are generic functions in the probability measure protocol and are applied to data sets by viewing them as empirical distributions.)

1.1.14 Cactus

Cactus [25, 31] is a system for numerical linear algebra and optimization implemented in CLOS. Cactus is designed to closely model the abstractions used in a course on finite dimensional (vector) spaces [46]. It provides representations for a variety of *spaces*, *points* (elements of some space), and *mappings* between spaces. The linear algebra core of Cactus deals with vector spaces, vectors, and

linear transformations; more general spaces, points, and mappings are used are used to support constrained optimization at a similar level of abstraction [22].

The implementation of Cactus is, in part, an experiment to see how closely a program can follow natural mathematical abstractions, without losing the level of performance in large numerical problems provided by traditional Fortran subroutine libraries. To evaluate the results of this experiment, I have compared, in [25], the design, implementation, and performance of Cactus on certain standard numerical linear algebra problems with Linpack [11], a classic high-quality Fortran package for solving systems of linear equations and related problems.

The result of the experiment was that the overhead in using a high level of abstraction is modest — the runtimes of Cactus versions of common matrix decompositions range from about the same as the corresponding Linpack routines to perhaps about 50% longer. At the same time, the object-oriented design of Cactus eases code reuse and customization in ways that are not possible in Fortran.

1.2 Current Ports

This release has had very limited testing in:

- Franz Allegro CL 4.0 with CLXr5, X11r4, on Sun Microsystems Sparcstation 2 and 330.

1.3 Getting Arizona

This release of Arizona is available via anonymous FTP from `belgica.stat.washington.edu` (128.95.17.57) in the directory `/pub/az`. Each subsystem is available in its own compressed tar file (eg. `geometry.tar.Z`). In addition, the Latex files making up this document are in `doc.tar.Z`.

Remember to use binary mode when ftping the compressed tar files. Copy the tar files into the directory you want to be the Arizona root directory. Uncompress and extract the tar files to create a copy of the Arizona directory tree. Edit `files.lisp` in the Arizona root directory to be consistent with your choice. Edit all the `make.lisp` and `load.lisp` files in the various subdirectories to be consistent with your choice of Arizona root directory. You may also need to make changes to reflect what version of PCL you have and where it is kept, if it's not automatically part of your lisp image.

Then start up your lisp and load (don't compile it) a `make.lisp` from one of the subdirectories to compile and load all that subsystem and all the other subsystems it depends on. In this release, loading `chart/make.lisp` will compile and load everything.

Once everything is compiled, users without permission to modify the Arizona root directory can safely load Chart, Slate, or any other system by loading the appropriate `load.lisp`. See the Tools module described section 3 for more details on making and loading.

2 Style Conventions

Some of these points are trivial; a few are actually very important. These are NOT intended to be rules, but rather suggestions for default conventions, to be followed unless there's a good reason not to. Most are pretty arbitrary, even the important ones—what's important is not that particular choice, but making one choice and being consistent about it. The real point is to to make it easier to understand little pieces of each other's code, to avoid misunderstanding.

2.1 Modules and Packages

- Each module should have its own directory and its own package.

- The package should be defined in a separate `package.lisp` file in the module's directory.
- The module's directory should contain a `load.lisp` which, when loaded, loads the module, and a `compile.lisp` which, when loaded, compiles and loads the module.
- In general, a module's package should not use other packages, except Lisp (and maybe CLOS or PCL).
- Packages should have reasonable short nicknames, to encourage people not to use them in another package.

2.2 Abstract Data Types

An abstract data type is defined by a functional interface, its (user) protocol. An abstract data type can be implemented by a CLOS class, a `defstruct`, or other Lisp data type(s). In general, the user should not be able to depend on an abstract type being implemented by `defclass`, `defstruct`, or some other means, but should stick to the provided functional interface and complain to the author if the interface isn't rich enough. The fact that a type is defined by a `defclass` should only be made public when the idea is to allow users to specialize the type by defining their own subclasses.

- An abstract data type should be documented with two protocols: a user's protocol and an implementor's protocol.
- The user's protocol should consist of safe functions; that is, they should report an error rather than give unpredictable results when called with invalid arguments.
- The user's protocol (for the type `Foo`) should include:
 - `(foo? foo0)` (or perhaps `(foo-p foo0)`) tests whether `foo0` is a valid representation of the `Foo` type.
 - `(make-foo &rest options)` makes a new `Foo`.
`options` is a "property" list of keyword-value pairs that determine how the new `Foo` is to be initialized.
 - `(equal-foos? foo0 foo1)` tells if `foo0` and `foo1` are equivalent in whatever sense is appropriate for `Foo`'s.
 - `(foo-bla foo0)` for getting the `bla` property of `foo0`.
 - `(setf (foo-bla foo0) new-bla)` for setting the `bla` property of `foo0`.
 - `(copy-foo foo0 &key result)` returns a copy of `foo0`.
 If `result` is not supplied, `copy-foo` makes a new `Foo` for the copy. If `result` is supplied, and is capable of holding a copy of `foo0`, the state of `result` is modified to make it equivalent to `foo0` (in the sense of `equal-foos?`).
 If `result` is not capable of holding a copy of `foo0`, an error should usually be signalled.
 However, in some cases, it may be desirable to have an `error-p` argument which, when null, indicates that a new `Foo` should be made and returned silently when the supplied `result` is incorrect.
 Note that `result` need not be the same type of object as `foo0`. For example, `(copy-sequence a-list a-vector)`.
- Type names should be capitalized in code for readability.

2.3 Immutable Object Types

It's often useful to define an abstract type so that its instances can be assumed to be immutable, that is, whose internal state cannot be changed (or, at least, is very difficult to change). For example, we can easily cache the value(s) of a frequently called, otherwise expensive function whose arguments are all immutable objects.

2.4 Limiting garbage: informal Result and Resource protocols

Scientific functions frequently produce as results large data structures (usually arrays) that are only needed temporarily. For example, consider evaluating an expression like:

```
(add-vector (scale-vector a x) y)
```

where `a` is a `Single-Float` and `x` and `y` are `(Vector Single-Float (n))`, for some large `n`. The result of `(scale-vector a x)` will be a large vector of floats that immediately becomes garbage. The `Series` extension to `CL` (Appendix A, [50]) is one attempt to solve this problem; it makes it possible to evaluate expressions like the one above as implicit loops, without actually ever constructing the intermediate vector. However, `Series` is not a solution for us, because it only solves the problem for `Sequences`. We need a standard idiom for dealing with similar expressions in arbitrary user-defined data types. Our proposed answer is a style convention, followed in `Arizona`, based on optional result arguments and resources of instances of data types that are frequently used on a temporary fashion.

2.5 Optional Results

- Functions that return a sizable result may take an `&key` result argument. For example: `(compose map0 map1 :result map2)`
- A function that follows the result protocol should test a supplied `:result` to ensure that it is valid and to signal an error if it is not.
- The value returned should be `eq` to the supplied `:result` if an error is not signaled.
(This may be violated in some circumstances. For example, it might be convenient to have `compose` accept arguments that are either arrays or numbers and to have `(compose x y :result z)` simply ignore `z` if `x`, `y`, and `z` are all numbers.)
- The user can indicate a destructive operation by supplying a `:result` that is `eq` to one of the other arguments. That is, `(compose map0 map1 :result map1)`, is supposed to overwrite `map1` with the composition of `map0` and `map1`. The function is not required to support such destructive operations, but it should test for `eq`-ness between the `:result` and the other arguments, and signal an error if it does not support the corresponding destructive operation.
(This may also be violated when the result is a number or something similar.)

We may extend the `Result` protocol to functions returning multiple values.

2.6 Resources

Suppose `Bla` is an abstract data type whose instances are sizable objects. Then `Bla` may elect to follow the `Resource` protocol:

- (`borrow-bla &rest options`) gets a `Bla` from the `Bla` resource, or makes a new one if the resource is empty. `options` is a “property” list of keyword–value pairs that determine how the borrowed `Foo` is to be initialized.
- (`return-bla bla0`) returns the `bla0` to the resource. This must be used with care, because any further reference to `bla0` will be incorrect. However, most functions will not test to see if their arguments have been returned to a resource.
- (`in-bla-resource? bla0`) can be used to test if `bla0` thinks it’s in the `Bla` resource, which implies that something’s wrong, the reference to it that was passed to `in-bla-resource?` persisted after `bla0` was returned.
- (`with-borrowed-bla (name &rest options) &body`) is a macro that borrows a `Bla`, initializes it using `options`, binds it to `name`, and then returns it to the resource on exit, returning the values returned by the last form in `body`. Like `return-bla`, it must be used with care.
- (`with-borrowed-blas (names &rest options) &body`) is a macro that borrows a `Bla` for each `name` in `names`, initializes each one using `options`, binds each to the corresponding name, and returns them to the resource on exit, returning the values returned by the last form in `body`. Like `with-borrowed-bla`, it must be used with care.

A sample implementation for vectors:

```
(defparameter *vector-resource* (make-hash-table :test #'eql))

(defun borrow-vector (length)
  (declare (special *vector-resource*))
  (check-type length Fixnum)
  (let ((v-list (gethash length *vector-resource* ())))
    (cond (v-list
           (setf (gethash length *vector-resource*) (rest v-list))
           (first v-list))
          (t (make-long-enough-vector length)))))

(defun return-vector (v)
  (declare (special *vector-resource*))
  (check-type v Vector)
  (push v (gethash (length v) *vector-resource* ())))

(defun in-vector-resource? (v)
  (find v (gethash (length v) *vector-resource* ())))

(defmacro with-borrowed-vector ((name length) &body body)
  ;; use {\sf return-name} so we deallocate the right thing,
  ;; even if the user re-assigns {\sf name}.
  (let ((return-name (gensym)))
    `(let* ((,return-name (borrow-vector ,length))
            (,name ,return-name))
       (multiple-value-prog1
        (progn ,@body))
```

```

      (return-vector ,return-name))))))

(defmacro with-borrowed-vectors ((names length) &body body)
  (let* ((return-names (mapcar #'(lambda (name) (gensym (string name)))
                                names))
         (borrowings (mapcar #'(lambda (return-name)
                                 `(',return-name (borrow-vector ,length))
                                return-names))
         (returnings (mapcar #'(lambda (return-name)
                                 `('return-vector ,return-name))
                                return-names))
         (bindings (mapcar #'(lambda (name return-name) `(',name ,return-name))
                            names return-names)))
    `(let* (,@borrowings ,@bindings)
      (multiple-value-prog1
        (progn ,@body)
        ,@returnings))))

```

2.7 Miscellaneous

- Integer intervals should be specified by a `start—end` pair as in the CL sequence functions; the convention is that the interval includes `start` but not `end`. For example, `start=1, end=5` specifies the sequence 1 2 3 4.

It is very important to abide by this strictly. Otherwise it's impossible to eradicate fencepost errors (off by 1), especially from code dealing with bitmap displays.

This convention may seem unnatural at first, but, in my experience, it tends to simplify code. For example, the length of the interval is `(- end start)`. If you partition an interval into subintervals, by `i0 i1 i2 i3` then the first subinterval is `i0 i1`, the second is `i1 i2`, and so on.

- Names should be zero-origin wherever at all reasonable. For example, `(defun plus (a0 a1) ...)` rather than `(defun plus (a1 a2) ...)`. The reason for this is that, in my opinion, zero-origin is unnatural, but CL forces us to use it for arrays and sequences. Because it's easy to get confused, it's important to be consistent everywhere.
- Predicate names should end with a `#/?`. The usual CL convention, ending with `-p` is not as expressive, and is easily confused with an accessor for a `p` property (eg. the parameter of a binomial distribution object).
- Don't mix `&key` and `&optional` arguments; in general, don't use `&optional` arguments.
- Top-level (user entry) functions should use `&key` freely. If a frequently called function, `foo`, has an elaborate `&key-&rest` argument list, make a separate `%foo` function that is called by `foo` and has all required arguments.
- Top-level (user entry) functions should be safe against invalid arguments; that is, they should signal errors for invalid arguments rather than just returning unpredictable results. Because it's hard to know just what assumptions are crucial, they should `assert` as many predicates as possible about their argument values. For efficiency, provide an `unsafe %foo`, that is called by the safe `foo` and can be called by other functions that take the responsibility for passing valid arguments.

- Common errors should be signaled by calling a specially defined function (eg. `missing-method-error`), rather than just calling `error`, `error`, or `warn`. In addition to making for more readable code, this should make it possible to adapt to more sophisticated CL support for signalling and handling conditions that's likely to appear in the future.

2.8 Style Conventions for CLOS

- Generic functions that dispatch on only one argument should have that as the 1st argument, unless there's a good reason not to.
- A generic function that has only one method should be an ordinary function that uses `az:type-check` to ensure the validity of its argument(s).
- Use of `:before`, `:after` and other combined methods should be avoided. The only clearly valid use is in initialization methods, because the initialization protocol was designed with them in mind.

3 Assorted Common Lisp Tools

The Arizona-Tools module provides assorted general purpose utilities for Common Lisp programming. Collecting these tools in one place saves users from reinventing many minor variations of the same wheel and encourages programming style consistent with the conventions suggested in section 2.

3.1 System Compiling and Loading Tools

The Arizona-Tools module provides several functions that can be used as the basis for a primitive `defsystem` facility. They are in the `:User` package, so that they can be loaded before the `:Arizona-Tools` package is defined. If the source file (extension `“.lisp”` only) is newer than the binary (extensions `“.bin”` and `“.fasl”` are supported), `user:compile-if-needed` compiles the source file. It loads the binary whether the source is compiled or not. `user:compile-all-if-needed` iterates over files concatenating a directory string onto the front before calling `user:compile-if-needed`. `user:load-all` simply loads all the binaries without compiling.

3.2 Runtime type checking

`az:declare-check` has a syntax like `declare`; it is a macro that expands into type checking forms for each type declaration. It does not also expand into declarations, because macros are not allowed to expand into declarations (see [50], p. 217). `az:declare-check` forms are analyzed by the Definitions module [27] as though they were `declare` forms.

3.3 Variations on `setf`

`az:mulf`, `az:divf`, `az:maxf`, and `az:minf` are obvious variations on `incf`, `decf`, etc. `az:multiple-value-setf` is to `multiple-value-setq` as `setf` is to `setq`.

3.4 Iteration tools

`az:with-collection` returns a list of whatever is collected within its scope by calls to `az:collect`, in the order that `az:collect` is called. This is cleaner than a common idiom for the same purpose that uses `let`, `push`, and `nreverse`. This was extracted somehow from the iteration constructs in PCL [7]; we should probably come up with a more precise definition of what it does.

3.5 Macro Writing Tools

`az:once-only` assures that the forms given as vars are evaluated in the proper order, once only. It's useful for writing macros. It was also extracted from PCL[7] somehow.

3.6 Bug report tools

These functions are useful for generating text in output files for recording for bug reports, benchmarking, etc.: `az:day-of-week-name` translates a zero-based day number to a 3 character day name abbreviation (eg. "Mon"). `az:month-number` translates a zero-based month number to a 3 character month name abbreviation (eg. "Jul"). `az:print-date` prints the current date in a human readable format. `az:print-system-description` attempts to print a verbose comprehensive description of the current software/hardware environment.

3.7 Program monitoring

`az:if-reentered` provides a very simple way of detecting whether a piece of code has been reentered or not. This was inspired by the desire to have MAC window event functions call Slate event functions, without having to worry that errors in the canvas event functions would cause unbreakable infinite loops.

`az:time-body` is a macro that returns the "internal run time" taken by the execution of the forms in its body. Note that this means that the values returned by these forms are lost.

3.8 Copy protocol

Copying is something that many think should be provided automatically by an object system. Copying is harder than it first appears, because it's impossible to define, in general, where the copying should stop. There are two extreme alternatives, (1) to merely copy pointers at the top level and (2) to follow all pointers, copying recursively. The second isn't practical; it will almost always result in copying the entire image, if you are really serious about following all pointers, of all kinds. Unfortunately, the first doesn't usually capture what you want.

- `az:copy` is a base generic function that users can specialize to control the copying Lisp objects, including, but not restricted to, instances of CLOS classes. However, we expect that most cases will be dealt with by specializing one of the functions below, that are called by the default method for `az:copy`, rather than `az:copy` itself.

A method for `az:copy` should produce an object that is similar to `original`, in whatever sense of "similar" is appropriate for `original` (and `result`).

The keyword `result` argument allows the user to pass in a preallocated object to hold the copy. In this case, a method for `az:copy` changes the state of `result` to make it similar to `original`. This is done in the default method by calling `az:copy-to!` (see below).

If no `result` is supplied, a method for `az:copy` creates a new object. The default method does this by calling `az:new-copy` (see below).

Note that it is not assumed that `original` and `result` are the same type. When `result` is not the same type as `original`, `az:copy` is interpreted as coercion.

A method for `az:copy` is expected to signal an error if given an `original` and `result` that it doesn't know how to handle. This is done in the default method by calling `az:verify-copy-result` (see below).

- `az:new-copy` differs from `az:copy` in that always creates a new object to hold the copy.
- A method for `az:copy-to!` destructively changes the state of `result` to be similar to `original`, in whatever sense is appropriate for the combination of `original` and `result`. It should signal an error if given a combination it does not understand.
- A method for `az:verify-copy-result?` tests to see if it's valid to copy `original` to `result`. If `errorp` is `nil`, it acts as a predicate, returning `t` or `nil`, if the copying is valid or not. If `errorp` is `t`, it signals an error if the copying is not valid, and returns `t` if the copying is valid.
- `az:copy-slots-to!` is called by the default method for `az:copy-to!`. It copies the slots (the pointers) from `original` to `result`, by extracting an `initarg` list from `original` and calling `reinitialize-instance` on `result`.

3.9 Kill protocol

The purpose of killing is to prevent an object from being used again, and possibly to free up resources associated with that object.

A typical example is when a window in some non-lisp window system is destroyed; the lisp object that represented that window in the lisp environment should be killed to prevent lisp from hanging up the window system by trying to use a destroyed window. Other objects related to the window may want to be killed as well to free resources no longer needed.

The default method for `az:kill` for CLOS instances is to change the class of the instance to `az:Dead-Object`. This will cause undefined method errors the next time a generic function is called on the dead instance, but generic functions that encounter dead objects frequently can be given methods to allow them to deal with the situation more gracefully. Changing the class to `az:Dead-Object` will cause the data in the slots of the killed instance to become unreferenced and eventually get reclaimed by the garbage collector.

3.10 Resource Protocol

The generic functions, `az:borrow-instance` and `az:return-instance`, and the macro `az:with-borrowed-instance`, are used to implement a specializable protocol that allows a class to provide a resource of instances of itself. See the discussion of the Resource protocol in section 2.6.

4 Reference Manual

az:-2pi-	Constant
----------	----------

Usage: az:-2pi-

az:-double-float-digits-	Constant
--------------------------	----------

Usage: az:-double-float-digits-

az:-double-float-radix-	Constant
-------------------------	----------

Usage: az:-double-float-radix-

az:-large-double-float-	Constant
-------------------------	----------

Usage: az:-large-double-float-

az:-largest-magnitude-	Constant
------------------------	----------

Documentation: r1mach(2)

Usage: az:-largest-magnitude-

az:-largest-relative-spacing-	Constant
-------------------------------	----------

Documentation: r1mach(4)

Usage: az:-largest-relative-spacing-

az:-lnpi-	Constant
-----------	----------

Usage: az:-lnpi-

az:-lnsqrt2pi-	Constant
----------------	----------

Usage: az:-lnsqrt2pi-

az:-log-largest-magnitude-	Constant
----------------------------	----------

Usage: az:-log-largest-magnitude-

az:-log-smallest-positive-magnitude-	Constant
--------------------------------------	----------

Usage: az:-log-smallest-positive-magnitude-

az:-log-smallest-relative-spacing-	Constant
------------------------------------	----------

Usage: az:-log-smallest-relative-spacing-

az:-log10-double-float-radix-	Constant
-------------------------------	----------

Documentation: r1mach(5)

Usage: az:-log10-double-float-radix-

az:-pi/2-	Constant
-----------	----------

Usage: az:-pi/2-

az:-small-double-float-	Constant
-------------------------	----------

Usage: az:-small-double-float-

az:-smallest-positive-magnitude-	Constant
----------------------------------	----------

Documentation: r1mach(1)

Usage: az:-smallest-positive-magnitude-

az:-smallest-relative-spacing-	Constant
--------------------------------	----------

Documentation: r1mach(3)

Usage: az:-smallest-relative-spacing-

az:-sqrt2pi-	Constant
--------------	----------

Usage: az:-sqrt2pi-

az:abs-difference	Function
-------------------	----------

Usage: (az:abs-difference x1 x2)

Arguments:

x1 — T

x2 — T

az:add-float-vectors	Function
----------------------	----------

Usage: (az:add-float-vectors v0 v1 &key result)

Arguments:

v0 — Az:Float-Vector
 v1 — Az:Float-Vector
 result — Az:Float-Vector

az:add-int16-vectors	Function
----------------------	----------

Usage: (az:add-int16-vectors v0 v1 &key result)

Arguments:

v0 — Az:Int16-Vector
 v1 — Az:Int16-Vector
 result — Az:Int16-Vector

Az:Alist-Table	Type
----------------	------

Usage: (typep x 'Az:Alist-Table)

az:all-subclasses	Generic Function
-------------------	------------------

Documentation: Returns a list of all the subclasses of <c>.

Usage: (az:all-subclasses class)

Arguments: class — (Or Symbol Class)

Returns: List

az:all-subclasses Class	Primary Method
-------------------------	----------------

Documentation: Returns a list of all the subclasses of <class>.

Usage: (az:all-subclasses class)

Arguments: class — Class

Returns: List

az:all-subclasses Symbol	Primary Method
--------------------------	----------------

Documentation: Find the Class with name <class-name> and recurse.

Usage: (az:all-subclasses class-name)

Arguments: class-name — Symbol

Returns: List

az:all-superclasses	Generic Function
---------------------	------------------

Documentation: Returns a list of all the superclasses of <c>.

Usage: (az:all-superclasses class)

Arguments: class — (Or Symbol Class)

Returns: List

az:all-superclasses Class	Primary Method
---------------------------	----------------

Documentation: Returns a list of all the superclasses of <c>.

Usage: (az:all-superclasses class)

Arguments: class — Class

Returns: List

az:all-superclasses Symbol	Primary Method
----------------------------	----------------

Documentation: Find the class with name <class-name> and recurse.

Usage: (az:all-superclasses class-name)

Arguments: class-name — Symbol

Returns: List

Az:Arizona-Object	Class
-------------------	-------

Documentation: A base class for all Arizona objects, just to help organize the class hierarchy a little.

Usage: (typep x 'Az:Arizona-Object)

Parents: Standard-Object

Children: Az:Dead-Object

az:array-data	Function
---------------	----------

Documentation: This function should return a simple 1d array of the same element type as <array> occupying the same memory locations as <array>. The correspondence between elements should be the same as for displaced arrays. If this is not possible, a call to <array-data> should produce an error.

Usage: (az:array-data array)

Arguments: array — Simple-Array

Az:Array-Index	Type
----------------	------

Documentation: The subset of Integer that can be used as an array index.

Usage: (typep x 'Az:Array-Index)

Az:Array-Index-Array	Type
----------------------	------

Usage: (typep x 'Az:Array-Index-Array)

Arguments: dims — T

Az:Array-Index-Matrix	Type
-----------------------	------

Usage: (typep x 'Az:Array-Index-Matrix)

Arguments:

d0 — T

d1 — T

Az:Array-Index-Vector	Type
-----------------------	------

Usage: (typep x 'Az:Array-Index-Vector)

Arguments: len — T

:Az	Package
-----	---------

Usage: (in-package :Az)

Az:Bit-Array	Type
--------------	------

Usage: (typep x 'Az:Bit-Array)

Arguments: dims — T

Az:Bit-Matrix	Type
---------------	------

Usage: (typep x 'Az:Bit-Matrix)

Arguments:

d0 — T

d1 — T

Az:Boolean	Type
------------	------

Usage: (typep x 'Az:Boolean)

az:borrow-float-vector	Function
------------------------	----------

Usage: (az:borrow-float-vector len)

Arguments: len — T

az:borrow-instance	Generic Function
--------------------	------------------

Documentation: Borrow an instance of the class <c> from a global resource, initializing it with the supplied <options>.

Usage: (az:borrow-instance c &rest options)

Arguments:

c — (Or Symbol Class)
options — List

Returns: Az::C

az:borrow-instance Class	Primary Method
--------------------------	----------------

Documentation: If an instance of <class> is in the resource, pop it out and apply <reinitialize-instance> to it and <options>. If no instance is in the resource, apply <make-instance> to <class> and <options>.

Usage: (az:borrow-instance class &rest options)

Arguments:

class — Class
options — List

Returns: Class

az:borrow-instance Symbol	Primary Method
---------------------------	----------------

Documentation: Find the class corresponding to <class> and recurse.

Usage: (az:borrow-instance class &rest options)

Arguments:

class — Symbol
options — List

Returns: Class

az:bound	Macro
----------	-------

Documentation: Clip <x> to the closed interval [xmin,xmax].

Usage: (az:bound xmin x xmax)

az:cache-value	Macro
----------------	-------

Documentation: This assumes that the function should give the same answer for the same arguments (in the sense of #'eq). It won't work, for example, on functions of arrays that depend on the values of the array elements, unless you are willing to assume that the array elements will not be changed from call to call.

See Abelson and Sussman, SICP, ex 3.27.

Usage: (az:cache-value arg-list &body body)

Az:Card16	Type
-----------	------

Usage: (typep × 'Az:Card16)

Az:Card16-Array	Type
-----------------	------

Usage: (typep × 'Az:Card16-Array)

Arguments: dims — T

Az:Card16-List	Type
----------------	------

Documentation: A list of Card16s.

Usage: (typep × 'Az:Card16-List)

Az:Card16-Matrix	Type
------------------	------

Usage: (typep × 'Az:Card16-Matrix)

Arguments:

d0 — T

d1 — T

Az:Card16-Vector	Type
------------------	------

Usage: (typep × 'Az:Card16-Vector)

Arguments: len — T

Az:Card24	Type
-----------	------

Usage: (typep × 'Az:Card24)

Az:Card24-Array	Type
-----------------	------

Usage: (typep × 'Az:Card24-Array)

Arguments: dims — T

Az:Card24-Matrix	Type
------------------	------

Usage: (typep × 'Az:Card24-Matrix)

Arguments:

d0 — T

d1 — T

Az:Card24-Vector	Type
------------------	------

Usage: (typep x 'Az:Card24-Vector)

Arguments: len — T

Az:Card28	Type
-----------	------

Usage: (typep x 'Az:Card28)

Az:Card28-Array	Type
-----------------	------

Usage: (typep x 'Az:Card28-Array)

Arguments: dims — T

Az:Card28-Matrix	Type
------------------	------

Usage: (typep x 'Az:Card28-Matrix)

Arguments:

d0 — T

d1 — T

Az:Card28-Vector	Type
------------------	------

Usage: (typep x 'Az:Card28-Vector)

Arguments: len — T

Az:Card32	Type
-----------	------

Usage: (typep x 'Az:Card32)

Az:Card32-Array	Type
-----------------	------

Usage: (typep x 'Az:Card32-Array)

Arguments: dims — T

Az:Card32-Matrix	Type
------------------	------

Usage: (typep x 'Az:Card32-Matrix)

Arguments:

d0 — T

d1 — T

Az:Card32-Vector	Type
------------------	------

Usage: (typep x 'Az:Card32-Vector)

Arguments: len — T

Az:Card4	Type
----------	------

Usage: (typep x 'Az:Card4)

Az:Card4-Array	Type
----------------	------

Usage: (typep x 'Az:Card4-Array)

Arguments: dims — T

Az:Card4-Matrix	Type
-----------------	------

Usage: (typep x 'Az:Card4-Matrix)

Arguments:

d0 — T

d1 — T

Az:Card4-Vector	Type
-----------------	------

Usage: (typep x 'Az:Card4-Vector)

Arguments: len — T

Az:Card8	Type
----------	------

Usage: (typep x 'Az:Card8)

Az:Card8-Array	Type
----------------	------

Usage: (typep x 'Az:Card8-Array)

Arguments: dims — T

Az:Card8-Matrix	Type
-----------------	------

Usage: (typep x 'Az:Card8-Matrix)

Arguments:

d0 — T

d1 — T

Az:Card8-Vector	Type
-----------------	------

Usage: (typep x 'Az:Card8-Vector)

Arguments: len — T

az:clean-declare	Function
------------------	----------

Usage: (az:clean-declare &rest clauses)

Arguments: clauses — T

az:copy	Generic Function
---------	------------------

Documentation: Copying is something that many think should be provided automatically by an object system. Copying is harder than it first appears, because it's impossible to define, in general, where the copying should stop. There are two extreme alternatives, (1) to merely copy pointers at the top level and (2) to follow all pointers, copying recursively. The second isn't practical; it will almost always result in copying the entire image, if you are really serious about following all pointers, of all kinds. Unfortunately, the first doesn't usually capture what you want.

<copy> is a base generic function that users can specialize to control the copying Lisp objects, including, but not restricted to, instances of CLOS classes. However, we expect that most cases will be dealt with by specializing one of the functions below, that are called by the default method for <copy>, rather than <copy> itself.

A method for <copy> should produce an object that is similar to <original>, in whatever sense of "similar" is appropriate for <original> (and <result>).

The keyword <result> argument allows the user to pass in a preallocated object to hold the copy. In this case, a method for <copy> changes the state of <result> to make it similar to <original>. This is done in the default method by calling <copy-to!> (see below).

If no <result> is supplied, a method for <copy> creates a new object. The default method does this by calling <new-copy> (see below).

Note that it is not assumed that <original> and <result> are the same type. When <result> is not the same type as <original>, <copy> is interpreted as coercion.

A method for <copy> is expected to signal an error if given an <original> and <result> that it doesn't know how to handle. This is done in the default method by calling <verify-copy-result> (see below).

Usage: (az:copy original &key result)

Arguments:

original — T

result — T

Returns: result

az:copy T	Primary Method
-----------	----------------

Documentation: If <result> is supplied (and <result> is not eq to <original>), then the default method for <copy> does a destructive <copy-to!> of <original> to <result>. <result> must be an object that is capable of holding a copy of <original>, but it need not be exactly the same type as <original>. For example, one might write a method for copying Lists to Vectors. If <result> is not supplied, the default method calls <new-copy>.

Usage: (az:copy original &key result)

Arguments:

original — T

result — T

Returns: result

az:copy Az:Timestamp	Primary Method
----------------------	----------------

Documentation: Copier function for Timestamps.

Usage: (az:copy timestamp &key result)

Arguments:

timestamp — Az:Timestamp

result — Az:Timestamp

Returns: result

az:copy-array-contents	Function
------------------------	----------

Usage: (az:copy-array-contents a0 &optional a1)

Arguments:

a0 — Array

a1 — Array

az:copy-float-array	Function
---------------------	----------

Usage: (az:copy-float-array a0 a1)

Arguments:

a0 — Az:Float-Array

a1 — Az:Float-Array

az:copy-to!	Generic Function
-------------	------------------

Documentation: <copy-to!> overwrites <result> with the state of <original>. Methods should signal an error if <result> is not able to hold a copy of <original>, for what ever reason.

Usage: (az:copy-to! original result)

Arguments:

original — T
result — T

Returns: result

az:copy-to! Array Array	Primary Method
-------------------------	----------------

Usage: (az:copy-to! a result)

Arguments:

a — Array
result — Array

az:copy-to! Standard-Object Standard-Object	Primary Method
---	----------------

Documentation: The default method for Standard-Objects calls the generic function <verify-copy-result?> to ensure that <result> is an object able to hold a copy of <original> and <copy-slots-to!> to do a shallow copy.

Usage: (az:copy-to! original result)

Arguments:

original — Standard-Object
result — Standard-Object

Returns: result

az:copy-vector	Function
----------------	----------

Documentation: Assumes <result> is at least as long as <v> and of a type that can hold the elements of <v>. Copies (the pointer) to each element of <v> to the corresponding place in <result>. If <result> is longer than <v>, elements beyond the length of <v> are unchanged.

Usage: (az:copy-vector v &key result)

Arguments:

v — Vector
result — Vector

Returns: result

az:day-of-week-name	Function
---------------------	----------

Documentation: Translate the day number to a 3 char abbreviation.

Usage: (az:day-of-week-name day-number)

Arguments: day-number — (Integer 0 6)

Returns: (String 3)

Az:Dead-Object	Class
----------------	-------

Documentation: The default method for <kill-object> changes the class to Dead-Object.

Usage: (typep x 'Az:Dead-Object)

Parents: Az:Arizona-Object

az:declare-check	Macro
------------------	-------

Documentation: This macro generates type checking forms and has a syntax like <declare>. Unfortunately, we can't easily have it also generate the declarations.

Usage: (az:declare-check &rest decls)

az:deletef	Macro
------------	-------

Documentation: Delete in place.

Usage: (az:deletef item place)

az:empty-slot?	Function
----------------	----------

Documentation: Is specified slot unbound or is its value nil?

Usage: (az:empty-slot? object slot-name)

Arguments:

object — Standard-Object

slot-name — Symbol

Returns: Az:Boolean

az:equal-array-dimensions?	Function
----------------------------	----------

Usage: (az:equal-array-dimensions? a0 &rest others)

Arguments:

a0 — Array

others — List

Returns: boolean

az:equal?	Generic Function
-----------	------------------

Documentation: Are the two objects equal, in whatever sense is appropriate for the two types?

Usage: (az:equal? o0 o1)

Arguments:

o0 — T

o1 — T

az:equal? T T	Primary Method
---------------	----------------

Documentation: The default sense of equivalence is <eq>.

Usage: (az:equal? o0 o1)

Arguments:

o0 — T

o1 — T

az:equal? Az:Timestamp Az:Timestamp	Primary Method
-------------------------------------	----------------

Documentation: Equality predicate for Timestamps.

Usage: (az:equal? timestamp0 timestamp1)

Arguments:

timestamp0 — Az:Timestamp

timestamp1 — Az:Timestamp

Returns: Az:Boolean

az:fill-array-index-array	Function
---------------------------	----------

Usage: (az:fill-array-index-array array value)

Arguments:

array — (Az:Array-Index-Array *)

value — Az:Array-Index

az:fill-card16-array	Function
----------------------	----------

Usage: (az:fill-card16-array array value)

Arguments:

array — Az:Card16-Array

value — Az:Card16

az:fill-card24-array	Function
----------------------	----------

Usage: (az:fill-card24-array array value)

Arguments:

array — Az:Card24-Array

value — Az:Card24

az:fill-card28-array	Function
----------------------	----------

Usage: (az:fill-card28-array array value)

Arguments:

array — Az:Card28-Array
value — Az:Card28

az:fill-card32-array	Function
----------------------	----------

Usage: (az:fill-card32-array array value)

Arguments:

array — Az:Card32-Array
value — Az:Card32

az:fill-card8-array	Function
---------------------	----------

Usage: (az:fill-card8-array array value)

Arguments:

array — Az:Card8-Array
value — Az:Card8

az:fill-fixnum-array	Function
----------------------	----------

Usage: (az:fill-fixnum-array array value)

Arguments:

array — (Az:Fixnum-Array *)
value — Fixnum

az:fill-float-array	Function
---------------------	----------

Usage: (az:fill-float-array array value)

Arguments:

array — Az:Float-Array
value — Double-Float

az:fill-int16-array	Function
---------------------	----------

Usage: (az:fill-int16-array array value)

Arguments:

array — Az:Int16-Array
value — Az:Int16

az:fill-random-float-vector	Function
-----------------------------	----------

Usage: (az:fill-random-float-vector v &key min max)

Arguments:

v — T
 min — T
 max — T

az:fill-real-array	Function
--------------------	----------

Usage: (az:fill-real-array array value)

Arguments:

array — (Az:Real-Array *)
 value — Real

az:first-elt	Function
--------------	----------

Documentation: A <first> that works on Sequences.

Usage: (az:first-elt seq)

Arguments: seq — Sequence

Returns: T

Az:Fixnum-Array	Type
-----------------	------

Usage: (typep x 'Az:Fixnum-Array)

Arguments: dims — T

Az:Fixnum-Matrix	Type
------------------	------

Usage: (typep x 'Az:Fixnum-Matrix)

Arguments:

d0 — T
 d1 — T

Az:Fixnum-Vector	Type
------------------	------

Usage: (typep x 'Az:Fixnum-Vector)

Arguments: len — T

az:fl	Macro
-------	-------

Documentation: Coerce a number to a floating point number of the default precision (usually Double-Float).

Usage: (az:fl x)

Az:Float-Array	Type
----------------	------

Usage: (typep x 'Az:Float-Array)

Arguments: dims — T

Az:Float-List	Type
---------------	------

Documentation: A list of Double-Floats.

Usage: (typep x 'Az:Float-List)

Az:Float-Matrix	Type
-----------------	------

Usage: (typep x 'Az:Float-Matrix)

Arguments:

d0 — T

d1 — T

az:float-matrix*matrix	Function
------------------------	----------

Documentation: Overwrites <result> with the matrix product of <a0> and <a1>

Usage: (az:float-matrix*matrix a0 a1 &key result)

Arguments:

a0 — Az:Float-Matrix

a1 — Az:Float-Matrix

result — Az:Float-Matrix

Returns: result

az:float-matrix*vector	Function
------------------------	----------

Documentation: Overwrites <result> with the matrix product of <a> and <v>

Usage: (az:float-matrix*vector a v &key result)

Arguments:

a — Az:Float-Matrix

v — Az:Float-Vector

result — Az:Float-Vector

Returns: result

Az:Float-Vector	Type
-----------------	------

Usage: (typep x 'Az:Float-Vector)

Arguments: len — T

az:float-vector	Function
-----------------	----------

Documentation: Like <vector> or <list>.

Usage: (az:float-vector &rest initial-contents)

Arguments: initial-contents — List

Returns: Az:Float-Vector

az:float-vector-len2	Function
----------------------	----------

Documentation: Returns the squared norm of <v>.

Usage: (az:float-vector-len2 v)

Arguments: v — Az:Float-Vector

az:flsq	Function
---------	----------

Usage: (az:flsq x)

Arguments: x — Double-Float

az:for	Macro
--------	-------

Usage: (az:for specs &body body)

Az:Funcallable	Type
----------------	------

Usage: (typep x 'Az:Funcallable)

az:getalist	Function
-------------	----------

Documentation: This is like a gethash for association list tables.

Usage: (az:getalist key table &optional default)

Arguments:

key — T

table — List

default — T

(setf az:getalist)	Setf
--------------------	------

Documentation: This permits (setf (getalist —) —).

Usage: (setf (az:getalist key table &optional default) value)

az:if-reentered	Macro
-----------------	-------

Documentation: <if-reentered> provides a very simple way of detecting whether a piece of code has been reentered or not. This was inspired by the desire to have mac window event fns call canvas event fns, without having to worry that errors in the canvas event fns would cause unbreakable infinite loops.

Michael Sannella, 1993

Usage: (az:if-reentered reentered-yes-code reentered-no-code)

Az:Int16	Type
----------	------

Usage: (typep x 'Az:Int16)

Az:Int16-Array	Type
----------------	------

Usage: (typep x 'Az:Int16-Array)

Arguments: dims — T

Az:Int16-List	Type
---------------	------

Documentation: A list of Int16s.

Usage: (typep x 'Az:Int16-List)

Az:Int16-Matrix	Type
-----------------	------

Usage: (typep x 'Az:Int16-Matrix)

Arguments:

d0 — T

d1 — T

Az:Int16-Sequence	Type
-------------------	------

Documentation: A sequence of Int16s.

Usage: (typep x 'Az:Int16-Sequence)

Az:Int16-Vector	Type
-----------------	------

Usage: (typep x 'Az:Int16-Vector)

Arguments: len — T

az:integrate-card28-vector	Function
----------------------------	----------

Documentation: Note that <vec> and <integral> may be the same array.

Usage: (az:integrate-card28-vector vec integral)

Arguments:

vec — Az:Card28-Vector
 integral — Az:Card28-Vector

az:iota-card28-vector	Function
-----------------------	----------

Usage: (az:iota-card28-vector v)

Arguments: v — Az:Card28-Vector

az:kill-object	Generic Function
----------------	------------------

Documentation: The purpose of killing is to prevent an object from being used again, and possibly to free up resources associated with that object.

A typical example is when a window in some non-lisp window system is destroyed; the lisp object that represented that window in the lisp environment should be killed to prevent lisp from hanging up the window system by trying to use a destroyed window. Other objects related to the window may want to be killed as well to free resources no longer needed.

Usage: (az:kill-object object)

Arguments: object — T

Returns: object — T

az:kill-object Null	Primary Method
---------------------	----------------

Documentation: Don't need to do anything to kill nil.

Usage: (az:kill-object object)

Arguments: object — Null

Returns: nil

az:kill-object Standard-Object	Primary Method
--------------------------------	----------------

Documentation: The default method for <kill-object> for CLOS instances is to change the class of the instance to Dead-Object. This will cause undefined method errors the next time a generic function is called on the dead instance, but generic functions that encounter dead objects frequently can be given methods to allow them to deal with the situation more gracefully. Changing the class to Dead-Object will permit the data in the slots of the killed instance to become unreferenced garbage and eventually get reclaimed.

Usage: (az:kill-object object)

Arguments: object — Standard-Object

Returns: object — Az:Dead-Object

az:l2-dist-xy	Function
---------------	----------

Usage: (az:l2-dist-xy x0 y0 x1 y1)

Arguments:

x0 — T

y0 — T

x1 — T

y1 — T

az:l2-dist2-xy	Function
----------------	----------

Usage: (az:l2-dist2-xy x0 y0 x1 y1)

Arguments:

x0 — T

y0 — T

x1 — T

y1 — T

az:l2-norm-xy	Function
---------------	----------

Usage: (az:l2-norm-xy x y)

Arguments:

x — T

y — T

az:l2-norm2-xy	Function
----------------	----------

Usage: (az:l2-norm2-xy x y)

Arguments:

x — T

y — T

az:large-allocation	Macro
---------------------	-------

Usage: (az:large-allocation &body body)

az:large?	Function
-----------	----------

Usage: (az:large? x y &optional big)

Arguments:

x — T

y — T

big — T

az:last-elt	Function
-------------	----------

Documentation: Get the last element in a Sequence (not the last Cons in a List like <last>).

Usage: (az:last-elt seq)

Arguments: seq — Sequence

Returns: T

az:lazy-getalist	Macro
------------------	-------

Documentation: This is like getalist, except that constructor is a expression, rather than a default value, and the value returned by funcall-ing the constructor is stored into the table if the lookup returns nil. Note that this means that nil cannot be stored in a lazy alist — at least, not efficiently, which is the whole point.

Usage: (az:lazy-getalist key table constructor)

az:lazy-gethash	Macro
-----------------	-------

Documentation: This is like gethash, except that constructor is a function, rather than a default value, and the value returned by funcall-ing the constructor is stored into the table if the lookup returns nil. Note that this means that nil cannot be stored in a lazy hashtable — at least, not efficiently, which is the whole point.

Usage: (az:lazy-gethash key table constructor)

az:lazy-lookup	Macro
----------------	-------

Documentation: This is like lookup, except that constructor is a expression, rather than a default value, and the value returned by funcall-ing the constructor is stored into the table if the lookup returns nil. Note that this means that nil cannot be stored in a lazy alist — at least, not efficiently, which is the whole point.

Usage: (az:lazy-lookup key table constructor)

az:linear-mix-float-vectors	Function
-----------------------------	----------

Usage: (az:linear-mix-float-vectors x0 v0 x1 v1 &key result)

Arguments:

- x0 — Double-Float
- v0 — Az:Float-Vector
- x1 — Double-Float
- v1 — Az:Float-Vector
- result — Az:Float-Vector

az:linear-mix-int16-vectors	Function
-----------------------------	----------

Usage: (az:linear-mix-int16-vectors x0 v0 x1 v1 &key result)

Arguments:

x0 — Double-Float
 v0 — Az:Int16-Vector
 x1 — Double-Float
 v1 — Az:Int16-Vector
 result — Az:Int16-Vector

az:lookup	Generic Function
-----------	------------------

Documentation: Lookup provides a generic way to get and set entries in association lists and hashables (like gethash).

Usage: (az:lookup key table &optional default)

Arguments:

key — T
 table — (Or List Hash-Table)
 default — T

az:lookup T Hash-Table	Primary Method
------------------------	----------------

Usage: (az:lookup key table &optional default)

Arguments:

key — T
 table — Hash-Table
 default — T

az:lookup T List	Primary Method
------------------	----------------

Usage: (az:lookup key table &optional default)

Arguments:

key — T
 table — List
 default — T

(setf az:lookup)	Generic Function
------------------	------------------

Documentation: Lookup provides a generic way to get and set entries in association lists and hashables.

Usage: (setf (az:lookup key table &optional default) new-value)

Arguments:

new-value — T
 key — T
 table — (Or List Hash-Table)
 default — T

az:lookup T T Hash-Table Primary Method

Usage: (setf (az:lookup key table &optional default) new-value)

Arguments:

- new-value — T
- key — T
- table — Hash-Table
- default — T

az:lookup T T List Primary Method

Usage: (setf (az:lookup key table &optional default) new-value)

Arguments:

- new-value — T
- key — T
- table — List
- default — T

az:make-alist-table Function

Documentation: Creates an association list that can be used like a hash-table. The test function is kept in the first element of the list and the actual associations are kept in the rest of the list.

Usage: (az:make-alist-table &key test)

Arguments: test — Az:Funcallable

az:make-array-index-array Function

Usage: (az:make-array-index-array dims &key initial-element)

Arguments:

- dims — List
- initial-element — Az:Array-Index

Returns: (Az:Array-Index-Array *)

az:make-array-index-matrix Function

Usage: (az:make-array-index-matrix d0 d1 &key initial-element)

Arguments:

- d0 — Az:Array-Index
- d1 — Az:Array-Index
- initial-element — Az:Array-Index

Returns: Az:Array-Index-Matrix

az:make-array-index-vector	Function
----------------------------	----------

Usage: (az:make-array-index-vector len &key initial-element)

Arguments:

len — Az:Array-Index
 initial-element — Az:Array-Index

Returns: Az:Array-Index-Vector

az:make-card16-array	Function
----------------------	----------

Documentation: Make a Simple-Array of Card16 of the desired dimensions, initializing the elements by default to 0.

Usage: (az:make-card16-array dims &key initial-element)

Arguments:

dims — List
 initial-element — Az:Card16

Returns: Az:Card16-Array

az:make-card16-matrix	Function
-----------------------	----------

Usage: (az:make-card16-matrix d0 d1 &key initial-element)

Arguments:

d0 — Az:Array-Index
 d1 — Az:Array-Index
 initial-element — Az:Card16

Returns: Az:Card16-Matrix

az:make-card16-vector	Function
-----------------------	----------

Usage: (az:make-card16-vector len &key initial-element)

Arguments:

len — Az:Array-Index
 initial-element — Az:Card16

Returns: Az:Card16-Vector

az:make-card24-array	Function
----------------------	----------

Documentation: Make a Simple-Array of Card24 of the desired dimensions, initializing the elements by default to 0.

Usage: (az:make-card24-array dims &key initial-element)

Arguments:

dims — List
 initial-element — Az:Card24

Returns: Az:Card24-Array

az:make-card24-matrix	Function
-----------------------	----------

Usage: (az:make-card24-matrix d0 d1 &key initial-element)

Arguments:

d0 — Az:Array-Index
d1 — Az:Array-Index
initial-element — Az:Card24

Returns: Az:Card24-Matrix

az:make-card24-vector	Function
-----------------------	----------

Usage: (az:make-card24-vector len &key initial-element)

Arguments:

len — Az:Array-Index
initial-element — Az:Card24

Returns: Az:Card24-Vector

az:make-card28-array	Function
----------------------	----------

Documentation: Make a Simple-Array of Card28 of the desired dimensions, initializing the elements by default to 0.

Usage: (az:make-card28-array dims &key initial-element)

Arguments:

dims — List
initial-element — Az:Card28

Returns: Az:Card28-Array

az:make-card28-matrix	Function
-----------------------	----------

Usage: (az:make-card28-matrix d0 d1 &key initial-element)

Arguments:

d0 — Az:Array-Index
d1 — Az:Array-Index
initial-element — Az:Card28

Returns: Az:Card28-Matrix

az:make-card28-vector	Function
-----------------------	----------

Usage: (az:make-card28-vector len &key initial-element)

Arguments:

len — Az:Array-Index
initial-element — Az:Card28

Returns: Az:Card28-Vector

az:make-card32-array	Function
----------------------	----------

Documentation: Make a Simple-Array of Card32 of the desired dimensions, initializing the elements by default to 0.

Usage: (az:make-card32-array dims &key initial-element)

Arguments:

dims — List
initial-element — Az:Card32

Returns: Az:Card32-Array

az:make-card8-array	Function
---------------------	----------

Usage: (az:make-card8-array dims &key initial-element)

Arguments:

dims — List
initial-element — Az:Card8

Returns: Az:Card8-Array

az:make-card8-matrix	Function
----------------------	----------

Usage: (az:make-card8-matrix d0 d1 &key initial-element)

Arguments:

d0 — Az:Array-Index
d1 — Az:Array-Index
initial-element — Az:Card8

Returns: Az:Card8-Matrix

az:make-card8-vector	Function
----------------------	----------

Usage: (az:make-card8-vector len &key initial-element)

Arguments:

len — Az:Array-Index
initial-element — Az:Card8

Returns: Az:Card8-Vector

az:make-fixnum-array	Function
----------------------	----------

Usage: (az:make-fixnum-array dims &key initial-element)

Arguments:

dims — List
initial-element — Fixnum

Returns: (Az:Fixnum-Array *)

az:make-fixnum-matrix	Function
-----------------------	----------

Usage: (az:make-fixnum-matrix d0 d1 &key initial-element)

Arguments:

d0 — Az:Array-Index
 d1 — Az:Array-Index
 initial-element — Fixnum

Returns: Az:Fixnum-Matrix

az:make-fixnum-vector	Function
-----------------------	----------

Usage: (az:make-fixnum-vector len &key initial-element)

Arguments:

len — Az:Array-Index
 initial-element — Fixnum

Returns: Az:Fixnum-Vector

az:make-float-array	Function
---------------------	----------

Documentation: Make a Simple-Array of Double-Float of the desired dimensions, initializing the elements by default to 0.

Usage: (az:make-float-array dims &key initial-element)

Arguments:

dims — List
 initial-element — Double-Float

Returns: Az:Float-Array

az:make-float-matrix	Function
----------------------	----------

Usage: (az:make-float-matrix d0 d1 &key initial-element)

Arguments:

d0 — Integer
 d1 — Integer
 initial-element — Double-Float

Returns: Az:Float-Matrix

az:make-float-vector	Function
----------------------	----------

Documentation: Make a 1d Simple-Array of Double-Float of the desired length, initializing the elements by default to 0.

Usage: (az:make-float-vector length &key initial-element)

Arguments:

length — T
 initial-element — T

az:make-int16-array	Function
---------------------	----------

Documentation: Make a Simple-Array of Int16 of the desired dimensions, initializing the elements by default to 0.

Usage: (az:make-int16-array dims &key initial-element)

Arguments:

dims — List
initial-element — Az:Int16

Returns: Az:Int16-Array

az:make-int16-matrix	Function
----------------------	----------

Usage: (az:make-int16-matrix d0 d1 &key initial-element)

Arguments:

d0 — Az:Array-Index
d1 — Az:Array-Index
initial-element — Az:Int16

Returns: Az:Int16-Matrix

az:make-int16-vector	Function
----------------------	----------

Usage: (az:make-int16-vector len &key initial-element)

Arguments:

len — Az:Array-Index
initial-element — Az:Int16

Returns: Az:Int16-Vector

az:make-real-array	Function
--------------------	----------

Usage: (az:make-real-array dims &key initial-element)

Arguments:

dims — List
initial-element — Real

Returns: (Az:Real-Array *)

az:make-real-matrix	Function
---------------------	----------

Usage: (az:make-real-matrix d0 d1 &key initial-element)

Arguments:

d0 — Az:Array-Index
d1 — Az:Array-Index
initial-element — Real

Returns: Az:Real-Matrix

az:make-real-vector	Function
---------------------	----------

Usage: (az:make-real-vector len &key initial-element)

Arguments:

len — Az:Array-Index
initial-element — Real

Returns: Az:Real-Vector

az:make-timestamp	Function
-------------------	----------

Documentation: Constructor function for Timestamps.

Usage: (az:make-timestamp &key timestamp-info)

Arguments: timestamp-info — List

Returns: Az:Timestamp

az:max-card28-vector	Function
----------------------	----------

Usage: (az:max-card28-vector v)

Arguments: v — Az:Card28-Vector

az:month-name	Function
---------------	----------

Documentation: Translate the month number to a 3 char abbreviation.

Usage: (az:month-name month-number)

Arguments: month-number — (Integer 0 11)

Returns: (String 3)

az:multiple-value-setf	Macro
------------------------	-------

Documentation: <multiple-value-setf> is to <multiple-value-setq> as <setf> is to <setq>.

Usage: (az:multiple-value-setf places expression)

az:negate-float-vector	Function
------------------------	----------

Usage: (az:negate-float-vector v0 &key result)

Arguments:

v0 — Az:Float-Vector
result — Az:Float-Vector

az:negate-int16-vector	Function
------------------------	----------

Usage: (az:negate-int16-vector v0 &key result)

Arguments:

v0 — Az:Int16-Vector
 result — Az:Int16-Vector

az:new-copy	Generic Function
-------------	------------------

Documentation: <new-copy> returns a new object that is a copy of <original>.

Usage: (az:new-copy original)

Arguments: original — T

Returns: (Type-Of Az::Original)

az:new-copy Array	Primary Method
-------------------	----------------

Usage: (az:new-copy a)

Arguments: a — Array

az:new-copy Standard-Object	Primary Method
-----------------------------	----------------

Documentation: The default method for Standard-Object does a shallow copy, ie., copies the slot pointers from <original> to the new object.

Usage: (az:new-copy original)

Arguments: original — Standard-Object

Returns: new-instance — (Type-Of Az::Original)

az:once-only	Macro
--------------	-------

Documentation: <once-only> assures that the forms given as vars are evaluated in the proper order, once only. Used in the body of macro definitions.

Usage: (az:once-only vars &body body)

Az:Positive-Fixnum	Type
--------------------	------

Documentation: Actually, this means non-negative Fixnum.

Usage: (typep x 'Az:Positive-Fixnum)

Az:Positive-Float	Type
-------------------	------

Documentation: A non-negative Double-Float.

Usage: (typep x 'Az:Positive-Float)

Az:Positive-Float-List	Type
------------------------	------

Documentation: A list of Positive-Floats.

Usage: (typep x 'Az:Positive-Float-List)

Az:Positive-Real	Type
------------------	------

Documentation: Actually, non-negative Real.

Usage: (typep x 'Az:Positive-Real)

az:print-array	Function
----------------	----------

Usage: (az:print-array a &optional stream)

Arguments:

a — T
stream — T

az:print-date	Function
---------------	----------

Documentation: Print the current time and date in a human readable format.

Usage: (az:print-date &optional stream)

Arguments: stream — (Or Stream Az:Boolean)

Returns: String

print-object Az:Arizona-Object T	Primary Method
----------------------------------	----------------

Documentation: This method for <print-object> provides a special “look” to make it a little easier to recognize Arizona objects.

Usage: (print-object object stream)

Arguments:

object — Az:Arizona-Object
stream — Stream

Returns: String

az:print-system-description	Function
-----------------------------	----------

Documentation: Print a comprehensive description of the current hardware and software environment.

Usage: (az:print-system-description &optional stream)

Arguments: stream — (Or Stream Az:Boolean)

Returns: String

az:printing-random-thing	Macro
--------------------------	-------

Documentation: Similar to printing-random-object in the lisp machine but much simpler and machine independent. (Borrowed from PCL).

Usage: (az:printing-random-thing (thing s) &body body)

az:random-float-vector	Function
------------------------	----------

Usage: (az:random-float-vector dim &key min max)

Arguments:

dim — T
 min — T
 max — T

az:read-card8-array	Function
---------------------	----------

Usage: (az:read-card8-array path nskip nbytes array)

Arguments:

path — Simple-String
 nskip — Az:Card28
 nbytes — Az:Card28
 array — (Az:Card8-Array *)

Az:Real-Array	Type
---------------	------

Usage: (typep x 'Az:Real-Array)

Arguments: dims — T

Az:Real-Matrix	Type
----------------	------

Usage: (typep x 'Az:Real-Matrix)

Arguments:

d0 — T
 d1 — T

Az:Real-Vector	Type
----------------	------

Usage: (typep x 'Az:Real-Vector)

Arguments: len — T

az:remove-entry	Generic Function
-----------------	------------------

Documentation: Remove-entry provides a generic way to delete entries from association lists and hashtables (like remhash).

Usage: (az:remove-entry key table)

Arguments:

key — T
table — (Or List Hash-Table)

az:remove-entry T Hash-Table	Primary Method
------------------------------	----------------

Usage: (az:remove-entry key table)

Arguments:

key — T
table — Hash-Table

az:remove-entry T List	Primary Method
------------------------	----------------

Usage: (az:remove-entry key table)

Arguments:

key — T
table — List

az:return-float-vector	Function
------------------------	----------

Usage: (az:return-float-vector v)

Arguments: v — T

az:return-instance	Generic Function
--------------------	------------------

Documentation: Return the <instance> of class <c> to the resource. If <errorp> is not nil and <instance> is not an instance of <c>, an error is signalled.

Usage: (az:return-instance instance c &optional errorp)

Arguments:

instance — Standard-Object
c — (Or Symbol Class)
errorp — T

Returns: t

az:return-instance Standard-Object Class	Primary Method
--	----------------

Documentation: Return the <instance> of class <class> to the resource. If <errorp> is not nil and <instance> is not an instance of <class>, an error is signalled.

Usage: (az:return-instance instance class &optional errorp)

Arguments:

instance — Standard-Object
class — Class
errorp — T

Returns: t

az:return-instance Standard-Object Symbol	Primary Method
---	----------------

Documentation: Find the class corresponding to <class> and recurse.

Usage: (az:return-instance instance class &optional errorp)

Arguments:

instance — Standard-Object
class — Symbol
errorp — T

Returns: t

az:scale-float-vector	Function
-----------------------	----------

Usage: (az:scale-float-vector x0 v0 &key result)

Arguments:

x0 — Double-Float
v0 — Az:Float-Vector
result — Az:Float-Vector

az:scale-int16-vector	Function
-----------------------	----------

Usage: (az:scale-int16-vector x0 v0 &key result)

Arguments:

x0 — Double-Float
v0 — Az:Int16-Vector
result — Az:Int16-Vector

az:sign	Function
---------	----------

Usage: (az:sign a)

Arguments: a — T

az:small?	Function
-----------	----------

Usage: (az:small? x &optional y)

Arguments:

x — T

y — T

az:sq	Function
-------	----------

Usage: (az:sq a)

Arguments: a — T

az:stamp-time!	Function
----------------	----------

Documentation: Mark the timestamp with the current time (actually increment the global counter and mark the timestamp with the current count).

Usage: (az:stamp-time! timestamp)

Arguments: timestamp — Az:Timestamp

Returns: timestamp

az:sub-float-vectors	Function
----------------------	----------

Usage: (az:sub-float-vectors v0 v1 &key result)

Arguments:

v0 — Az:Float-Vector

v1 — Az:Float-Vector

result — Az:Float-Vector

az:sub-int16-vectors	Function
----------------------	----------

Usage: (az:sub-int16-vectors v0 v1 &key result)

Arguments:

v0 — Az:Int16-Vector

v1 — Az:Int16-Vector

result — Az:Int16-Vector

Az:Table	Type
----------	------

Usage: (typep x 'Az:Table)

az:time-body	Macro
--------------	-------

Documentation: Returns the “internal run time” taken by the execution of <body>. Note that this means that the value returned by <body> is lost.

Usage: (az:time-body &body body)

Az:Timestamp	Structure
--------------	-----------

Documentation: Timestamps are used for, among other things, lazy maintenance of dependencies. Typically, the dependent object will compare its timestamp to the timestamp of the independent object and only do updating computation if the timestamp of the independent object is newer. For such uses, we only need to record the order of critical events and not the absolute time. So we implement timestamps with a global counter, rather than attempting record realtime. This also has the advantage of eliminating problems caused by the finite and sometimes coarse resolution of system clocks. Timestamps can record up to $\langle \text{most-positive-fixnum} \rangle^2$ without overflow problems, which should be enough for most applications.

Usage: (typep x 'Az:Timestamp)

az:timestamp<	Function
---------------	----------

Documentation: Did <timestamp0> come strictly before <timestamp1>?

Usage: (az:timestamp< timestamp0 timestamp1)

Arguments:

timestamp0 — Az:Timestamp

timestamp1 — Az:Timestamp

Returns: Az:Boolean

az:timestamp<=	Function
----------------	----------

Documentation: Did <timestamp0> come before or at the same time as <timestamp1>?

Usage: (az:timestamp<= timestamp0 timestamp1)

Arguments:

timestamp0 — Az:Timestamp

timestamp1 — Az:Timestamp

Returns: Az:Boolean

az:timestamp=	Function
---------------	----------

Documentation: Equality predicate for Timestamps.

Usage: (az:timestamp= timestamp0 timestamp1)

Arguments:

timestamp0 — Az:Timestamp

timestamp1 — Az:Timestamp

Returns: Az:Boolean

az:timestamp>	Function
---------------	----------

Documentation: Did <timestamp0> come after <timestamp1>?

Usage: (az:timestamp> timestamp0 timestamp1)

Arguments:

timestamp0 — Az:Timestamp

timestamp1 — Az:Timestamp

Returns: Az:Boolean

az:timestamp>=	Function
----------------	----------

Documentation: Did <timestamp0> come after or at the same time as <timestamp1>?

Usage: (az:timestamp>= timestamp0 timestamp1)

Arguments:

timestamp0 — Az:Timestamp

timestamp1 — Az:Timestamp

Returns: Az:Boolean

az:transpose-float-matrix	Function
---------------------------	----------

Usage: (az:transpose-float-matrix a0 a1)

Arguments:

a0 — Az:Float-Matrix

a1 — Az:Float-Matrix

az:type-check	Macro
---------------	-------

Documentation: A <check-type> that takes arguments more like declarations, eg, (declare (type Integer x y)).

Usage: (az:type-check type &rest args)

az:under/overflow-danger?	Function
---------------------------	----------

Usage: (az:under/overflow-danger? x &optional scale)

Arguments:

x — Float
scale — T

az:unit-card8-array	Function
---------------------	----------

Usage: (az:unit-card8-array array)

Arguments: array — (Az:Card8-Array *)

az:unit-card8-vector	Function
----------------------	----------

Usage: (az:unit-card8-vector adv)

Arguments: adv — Az:Card8-Vector

az:until	Macro
----------	-------

Usage: (az:until test &body body)

az:verify-copy-result?	Generic Function
------------------------	------------------

Documentation: <verify-copy-result?> checks to see that <result> is an object able to hold a copy of <original>. In general, <result> need not be of the same class (or type) as <original>.

Usage: (az:verify-copy-result? original result)

Arguments:

original — T
result — T

Returns: Az:Boolean

az:verify-copy-result? Array Array	Primary Method
------------------------------------	----------------

Usage: (az:verify-copy-result? a result)

Arguments:

a — Array
result — Array

az:verify-copy-result? Standard-Object Standard-Object	Primary Method
--	----------------

Documentation: The default method for Standard-Objects only allows copying to instances of the same class.

Usage: (az:verify-copy-result? original result)

Arguments:

original — Standard-Object
result — Standard-Object

Returns: Az:Boolean

az:while	Macro
----------	-------

Usage: (az:while test &body body)

az:with-borrowed-float-vector	Macro
-------------------------------	-------

Usage: (az:with-borrowed-float-vector (vname min-length) &body body)

az:with-borrowed-float-vectors	Macro
--------------------------------	-------

Usage: (az:with-borrowed-float-vectors (vnames min-length) &body body)

az:with-borrowed-instance	Macro
---------------------------	-------

Usage: (az:with-borrowed-instance (name class &rest options) &body body)

az:with-collection	Macro
--------------------	-------

Documentation: <az:with-collection> returns a list of whatever is collected within its scope by calls to <az:collect>, in the order that {sf az:collect} is called. This is cleaner than a common idiom for the same purpose that uses <let>, <push>, and <nreverse>.

Usage: (az:with-collection &body body)

az:with-gc-tuned-for-large-allocation	Macro
---------------------------------------	-------

Documentation: <with-gc-tuned-for-large-allocation> is supposed to adjust the Lisp implementation's gc parameters to be temporarily more efficient at allocating a lot of data with a long lifetime, as in system loading. For example, the kind of thing one wants to do is turn off scavenging in a multi-generation gc and arrange for the data that's allocated in during the extent of <with-gc-tuned-for-large-allocation> to be immediately tenured.

Usage: (az:with-gc-tuned-for-large-allocation &body body)

az:without-gc	Macro
---------------	-------

Usage: (az:without-gc &body body)

az:zero-array-index-array	Function
---------------------------	----------

Usage: (az:zero-array-index-array array)

Arguments: array — (Az:Array-Index-Array *)

az:zero-card16-array	Function
----------------------	----------

Usage: (az:zero-card16-array array)

Arguments: array — Az:Card16-Array

az:zero-card16-vector	Function
-----------------------	----------

Usage: (az:zero-card16-vector adv)

Arguments: adv — Az:Card16-Vector

az:zero-card24-array	Function
----------------------	----------

Usage: (az:zero-card24-array array)

Arguments: array — Az:Card24-Array

az:zero-card24-vector	Function
-----------------------	----------

Usage: (az:zero-card24-vector v)

Arguments: v — Az:Card24-Vector

az:zero-card28-array	Function
----------------------	----------

Usage: (az:zero-card28-array array)

Arguments: array — Az:Card28-Array

az:zero-card28-vector	Function
-----------------------	----------

Usage: (az:zero-card28-vector v)

Arguments: v — Az:Card28-Vector

az:zero-card32-array	Function
----------------------	----------

Usage: (az:zero-card32-array array)

Arguments: array — Az:Card32-Array

az:zero-card32-vector	Function
-----------------------	----------

Usage: (az:zero-card32-vector v)

Arguments: v — Az:Card32-Vector

az:zero-card8-array	Function
---------------------	----------

Usage: (az:zero-card8-array array)

Arguments: array — (Az:Card8-Array *)

az:zero-card8-vector	Function
----------------------	----------

Usage: (az:zero-card8-vector array)

Arguments: array — Az:Card8-Vector

az:zero-fixnum-array	Function
----------------------	----------

Usage: (az:zero-fixnum-array array)

Arguments: array — Az:Fixnum-Array

az:zero-fixnum-vector	Function
-----------------------	----------

Usage: (az:zero-fixnum-vector v)

Arguments: v — Az:Fixnum-Vector

az:zero-float-array	Function
---------------------	----------

Usage: (az:zero-float-array array)

Arguments: array — Az:Float-Array

az:zero-float-vector	Function
----------------------	----------

Usage: (az:zero-float-vector v)

Arguments: v — Az:Float-Vector

az:zero-int16-array	Function
---------------------	----------

Usage: (az:zero-int16-array array)

Arguments: array — Az:Int16-Array

az:zero-int16-vector	Function
----------------------	----------

Usage: (az:zero-int16-vector adv)

Arguments: adv — Az:Int16-Vector

az:zero-real-array	Function
--------------------	----------

Usage: (az:zero-real-array a)

Arguments: a — Az:Real-Array

az:zero-real-vector	Function
---------------------	----------

Usage: (az:zero-real-vector v)

Arguments: v — Az:Real-Vector

References

- [1] Gul Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, 1986.
- [2] R.A. Becker and J.M. Chambers. Design of the S system for data analysis. *CACM*, 27(5):486–495, 1984.
- [3] R.A. Becker and J.M. Chambers. *S: An Interactive Environment for Data Analysis and Graphics*. Wadsworth, Belmont, Ca., 1984.
- [4] R.A. Becker, J.M. Chambers, and A.R. Wilks. *The New S Language*. Wadsworth and Brooks/Cole, Pacific Grove, CA, 1988.
- [5] Daniel Bobrow, David Fogelsong, and Mark Miller. Definition Groups: Making sources into first-class objects. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge MA, 1987.
- [6] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S. Keene, G. Kiczales, and D.A. Moon. Common Lisp Object System specification X3J13 document 88-002R. *SIGPLAN Notices*, 23, 1988.
- [7] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and object-oriented programming. *SIGPLAN Notices*, 21(11):17–29, 1986. Proceedings OOPSLA’86.
- [8] Tony D. DeRose. Geometric programming: a coordinate-free approach. ACM SIGGRAPH Tutorial Course No. 25, 1988.
- [9] Tony D. DeRose. A coordinate-free approach to geometric programming. In W. Strasser, editor, *Theory and Practice of Geometric Modeling*. Springer Verlag, 1989.
- [10] Tony D. DeRose. A coordinate-free approach to geometric programming (a geometric algebra and its implementation). Technical report, Dept. of Computer Science, U. of Washington, 1989. course notes for CSCI557, Spring 1989.
- [11] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. *LINPACK Users’ Guide*. SIAM, Philadelphia, 1979.
- [12] Franz, Inc. *Common Lisp: The Reference*. Addison-Wesley, Reading, MA, 1988.
- [13] Franz, Inc. *Allegro CL, An Extended Common Lisp, User’s Guide, Release 4.0*. Franz Inc., Berkeley, CA, January 1991.
- [14] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of VDM’91: Formal Software Development Methods*, 1991.
- [15] Philip E. Gill, Sven J. Hammarling, Walter Murray, Michael A. Saunders, and Margaret H. Wright. User’s guide for LSSOL (version 1.0): a Fortran package for constrained linear least squares and convex quadratic programming. Technical Report SOL 86-1, Systems Optimization Laboratory, Dept. of Operations Research, Stanford, 1986.
- [16] Philip E. Gill, Walter Murray, Michael A. Saunders, and Margaret H. Wright. User’s guide for NPSOL (version 4.0): a Fortran package for nonlinear programming. Technical Report SOL 86-2, Systems Optimization Laboratory, Dept. of Operations Research, Stanford, 1986.

- [17] Ira Kalet, Christine Sweeney, and Jonathan Jacky. Software design for interactive graphic radiation treatment simulation systems. In *Proceedings of the Fourteenth Annual Symposium on Computer Applications in Medical Care*, pages 594–598, Washington, D.C., November 1990. IEEE Computer Society Press.
- [18] Mark Kantrowitz. Portable Utilities for Common Lisp, User Guide and Implementation Notes. Technical Report CMU-CS-91-143, School of Computer Science, CMU, 1991.
- [19] Sonya E. Keene. *Object-oriented programming in Common Lisp: a programmer's guide to CLOS*. Symbolics Press and Addison-Wesley, Reading, MA, 1988.
- [20] Leslie Lamport. *Latex, a Document Preparation System*. Addison-Wesley, Reading, MA, 1985.
- [21] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM TOMS*, 5(3):308–371, 1979.
- [22] David G. Luenberger. *Optimization by vector space methods*. Wiley, NY NY, 1969.
- [23] John Alan McDonald. Antelope: data analysis with object-oriented programming and constraints. In *Proc. of the 1986 Joint Statistical Meetings, Stat. Comp. Sect.*, 1986.
- [24] John Alan McDonald. Object-oriented design in numerical linear algebra. Technical Report 109, Dept. of Statistics, U. of Washington, 1987.
- [25] John Alan McDonald. Object-oriented programming for linear algebra. *SIGPLAN Notices (Proceedings OOPSLA '89)*, 24(10):175–184, 1989.
- [26] John Alan McDonald. Calling NPSOL from Common Lisp. Technical report, Dept. of Statistics, U. of Washington, October 1991.
- [27] John Alan McDonald. Definitions: a simple database for typesetting documentation. Technical report, Dept. of Statistics, U. of Washington, October 1991.
- [28] John Alan McDonald. A simple graph browser. Technical report, Dept. of Statistics, U. of Washington, October 1991.
- [29] John Alan McDonald. Actors, interactors, and coordinators. Technical report, Dept. of Statistics, U. of Washington, 1992. in preparation.
- [30] John Alan McDonald. Basic Math: a package of assorted numerical analysis functions in Common Lisp. Technical report, Dept. of Statistics, U. of Washington, 1992. in preparation.
- [31] John Alan McDonald. Cactus: a object-oriented system for computational linear algebra and optimization. Technical report, Dept. of Statistics, U. of Washington, 1992. in preparation.
- [32] John Alan McDonald. Clay: a high performance user interface toolkit. Technical report, Dept. of Statistics, U. of Washington, 1992. in preparation.
- [33] John Alan McDonald. Probability Measure Objects in CLOS. Technical report, Dept. of Statistics, U. of Washington, 1992. in preparation.
- [34] John Alan McDonald and Mark Niehaus. Announcements: an implementation of implicit invocation. Technical report, Dept. of Statistics, U. of Washington, October 1991.

- [35] John Alan McDonald and Jan O. Pedersen. Computing environments for data analysis I: Introduction. *SIAM J. Scientific and Statistical Computing*, 6(4):1004–1012, 1985.
- [36] John Alan McDonald and Jan O. Pedersen. Computing environments for data analysis II: Hardware. *SIAM J. Scientific and Statistical Computing*, 6(4):1013–1021, 1985.
- [37] John Alan McDonald and Jan O. Pedersen. Computing environments for data analysis III: Programming environments. *SIAM J. Scientific and Statistical Computing*, 9(2):380–400, 1988.
- [38] John Alan McDonald and Jan O. Pedersen. Geometric abstractions for constrained optimization of layouts. In Andreas Buja and Paul Tukey, editors, *Computing and Graphics in Statistics*, volume IMA 36. Springer-Verlag, NY NY, 1991.
- [39] John Alan McDonald and Michael Sannella. Geometry: a package for basic geometric calculation in Common Lisp, release 1.0. Technical report, Dept. of Statistics, U. of Washington, October 1991.
- [40] John Alan McDonald and Michael Sannella. Chart: a simple Common Lisp package for plotting data and functions, release 1.0. Technical report, Dept. of Statistics, U. of Washington, 1992. in preparation.
- [41] John Alan McDonald and Michael Sannella. Slate: a portable bitmap graphics package for Common Lisp, release 1.0. Technical report, Dept. of Statistics, U. of Washington, 1992. in preparation.
- [42] Molly M. Miller and Eric Benson. *Lisp Style and Design*. Digital Press, 1990.
- [43] Bruce A. Murtagh and Michael A. Saunders. Large-scale linearly constrained optimization. *Mathematical Programming*, 14:41–72, 1978.
- [44] Bruce A. Murtagh and Michael A. Saunders. A projected Lagrangian algorithm and its implementation for sparse nonlinear constraints. *Mathematical Programming Study*, 16:84–117, 1978.
- [45] Bruce A. Murtagh and Michael A. Saunders. MINOS 5.1 users' guide. Technical Report SOL 83-20R, Systems Optimization Laboratory, Dept. of Operations Research, Stanford, 1987.
- [46] Walter Noll. *Finite-Dimensional Spaces (Algebra, Geometry, and Analysis)*, volume I. Martinus Nijhoff, Dordrecht, 1987.
- [47] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.
- [48] Mark G. Segal. *Programming Language Support for Geometric Computations*. PhD thesis, Computer Science, U.C. Berkeley, 1989.
- [49] G.L. Steele. *Common Lisp, The Language*. Digital Press, 1984.
- [50] G.L. Steele. *Common Lisp, The Language*. Digital Press, second edition, 1990.
- [51] Werner Stuetzle. Design and implementation of plot windows. In *Proceedings of the Statistical Computing Section of the American Statistical Association*, pages 32–40, 1987.
- [52] Werner Stuetzle. Plot windows. *JASA*, 82:466–475, 1987.

- [53] Kevin J. Sullivan and David Notkin. Reconciling environment integration and component independence. In *SIGSOFT'90: Fourth Symposium on Software Development Environments, Irvine CA*, pages 208–225, June 1990.
- [54] Kevin J. Sullivan and David Notkin. Behavioral relationships in object-oriented analysis. Technical Report 91-09-03, Dept. of Computer Science and Engineering, U. of Washington, 1991.
- [55] Deborah G. Tatar. *A Programmer's Guide to Common Lisp*. Digital Press, 1987.
- [56] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP*. Addison-Wesley, Reading, MA, 3rd edition, 1988.