Dylan™
Interim Reference Manual

June 3, 1994

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
408-996-1010

About this Manual

When the first Dylan manual was published in 1992, it was met with an enthusiastic response. People agreed with our goals for the language, and many people suggested ways in which Dylan could better meet its goals.

Since that time, an expanded group of engineers both inside and outside Apple have worked to refine the language design in response to those suggestions. The language has become simpler and more efficient. Loose ends have been tied up. The language has been given a new syntax. Throughout this process, changes to the language design have been published electronically in the form of design notes.

With the exception of the macro system, the current round of language design is now essentially complete. A new Dylan language reference will be published early in 1995. The new book will be the definitive specification of the Dylan language. Apple is working closely with other Dylan implementors to ensure that the new book will not be Apple-specific, but will apply equally to all Dylan implementations.

We realize that many people want to read about Dylan now! For that reason, to fill the gap until the new book is published, we've put together the *Dylan Interim Reference Manual*. This document is an interim user reference for the Dylan language. It combines the original Dylan book, the previously published design notes, and additional previously unpublished design decisions as of May 1994.

Our goal for this interim document has been to get something useful out to you as soon as possible, not to spend time refining it, so it's important to understand the limitations of this document:

- This is a very rough document, intended only as a temporary user reference until the new Dylan book is ready. The writing and formatting are messy in places. We hope you'll excuse us.
- This document is not intended as a language specification. There are places where it is imprecise or inconsistent. We hope you will understand that these are shortcomings of this document, and not of the language design as a whole.
- This document is not intended as a tutorial or introduction to Dylan programming. We assume that you are already somewhat familiar with

object oriented programming. You will often need to devise your own examples and flip back and forth through the manual as you read.

Despite these limitations, we think that this document will be useful to people who want to learn about Dylan, especially to those people who are using an early Dylan implementation. Enjoy, and let us know what you think!

Contents

Foreword to the First Edition

by Larry Tesler

In the mid-1980's, a number of researchers in Apple's Advanced Technology Group (ATG) discovered that exploration of their software ideas was increasingly constrained by languages and tools. They migrated a significant number of projects from Object Pascal, C, and C++ to Smalltalk, Lisp, and various knowledge-engineering environments.

Soon management became concerned that technologies developed in these unrelated environments would not be able to play together. Boldly, we proposed adoption of a single dynamic programming environment throughout the group. The goal was acknowledged, but we soon found that none of the environments we were using could meet every need. For some projects, Common Lisp was too big. For others, Smalltalk-80's simplicity was limiting. And of course, there were forcefully stated differences in personal preference.

Humbled, we decided to search outside the company for an environment that satisfied all requirements—at least technical considerations, if not matters of taste. We encountered some interesting languages, including Eiffel, Self, Beta, and OakLisp, to name just a few, but we could find no environment that met our requirements.

Reluctantly, we entertained the thought of designing yet another language. We envisioned a language that would:

- combine the best aspects of Lisp and Smalltalk
- provide static language users an attractive dynamic alternative
- be practical on small machines
- provide high dynamism during rapid prototyping and development
- provide tools to achieve commercial performance in production code

During this exercise, we were approached by an Apple engineering group seeking an object-oriented dynamic language for a contemplated product. Instead of developing a language just for research, we agreed that ATG would strive to create a system practical for delivery of commercial applications.

After some analysis, we concluded that a language that would meet all our requirements would be more like Lisp than Smalltalk, except that, like Smalltalk, it would be "objects all the way down". By providing alternate syntaxes atop the extremely neutral base of Lisp, we believed we could present a familiar face not only to Lisp programmers, but also to those accustomed to other syntaxes, including Smalltalk's.

We felt we could not design a good language without concurrently implementing it. But we did not have the skills in house to implement the kind of language we envisioned. Around that time we were approached by Massachusetts-based Coral Software, Inc., which was seeking to be acquired. Coral had created a popular Common Lisp implementation distinguished by its small memory footprint, very usable speed, interesting object system, and thorough integration into the Macintosh. A few months later, we acquired the assets of the company, hired most of the staff, and created "ATG East" in Cambridge.

We asked the new group to accept two challenges: (1) continue to develop their Common Lisp implementation, adding CLOS, ephemeral garbage collection, and other features; and (2) develop a new language and environment meeting the aforementioned requirements. They accepted both challenges. The first led to the very popular product known as Macintosh Common Lisp (MCL) 2.0. The second led to Dylan.

Dylan is not yet a frozen language. Debates about proposed improvements continue. But enough of Dylan has been designed, implemented, and experienced that now is the time to take it public and benefit from the wisdom of a wider programming community. We welcome your comments.

Larry Tesler
Cupertino, California
March 1992

Preface to the First Edition

by Ike Nassi

Since its founding, Apple Computer has been dedicated to the cause of moving the power of computers into the hands of computer users. Our most successful product to date has been the Apple® Macintosh™. Since its introduction eight years ago, Macintosh has defined the state of the art in computer ease of learning and ease of use. Macintosh, almost single-handedly, has transformed the way we use computers.

It's time to transform the way we program them, as well.

In 1984, before Macintosh, computer users were asked to make their work patterns conform to the requirements of the machine in front of them. All aspects of computing were presented in terms defined by the computer. The user was given little control over the organization of data, the order in which tasks were performed, and the manner in which work was presented. The machine presented a model which had little to do with the user's world, and the user had no choice but to conform to this model. Rather than facilitate work, the computer came between the user and the user's work. The user had to overcome the computer before getting any work done.

In 1992, the situation is still much the same for programmers. There is still a large gap between software product conception and software product realization. This gap is filled with bits, bytes, arrays, machine-level debuggers, dangling pointers, hours-long recompilations, and months-long design and redesign cycles. Programmers are still asked to present their high-level ideas of program behavior in low-level terms defined by CPU architectures of the early 1970's. Programming work flow is still dictated by the historical limitations of compilers, linkers, and debuggers. Program design in 1992 is reminiscent of typographical design in 1982: design and execution are performed by a series of specialists, resulting in an awkward, lengthy and expensive design, test, and redesign process.

Time-to-market is consistently rated very high in lists of factors affecting the success of high technology products. The end result of poor programming tools is either poor time-to-market or poor programs. Software companies are faced with a dilemma: they can take years, and do the job right, or they can cut corners and get their software into the market before their competitors. Neither solution results in a healthy software industry. Neither solution allows our products to keep up with our visions.

Software development today also has a large *barrier to entry*. Only companies with deep financial resources can bring a full fledged application to market. The result is a loss of healthy competition and fresh ideas in the software field.

Finally, our computing engines are becoming more diverse. In addition to the usual dimensions of speed, memory capacity, screen resolution, interconnectedness, power consumption, and size, the underlying operating systems and toolboxes are becoming broader in scope. Specific implementations can also become more focused, requiring a higher degree of tailorability. For example, with the introduction of mobile computing, software requirements may change as locations change. Software used on mobile computers needs to be able to rapidly respond to spontaneously changing conditions in a reliable way.

Object oriented programming takes important steps towards fixing these problems. By letting programmers structure the text of their programs in terms of the problem at hand, object oriented programming narrows the gap between conception and realization. However, object oriented programming by itself is insufficient. It addresses how programs are described, but it does not address many problems of the day-to-day activity of programming. It doesn't change programming's awkward work flow, nor does it make any guarantees about robustness, nor does it relieve the programmer of many tedious bookkeeping tasks, tasks which are better performed by a computer than by a human.

Today's most popular object oriented languages are still *static languages*. In such languages, most of the information about objects is discarded during compilation, so programs cannot be modified without recompilation, and debugging is more likely to occur at machine level than the level at which the program was designed. In addition, these languages encourage mixing objects with non-object oriented bits and bytes. Even objects can be treated as undifferentiated bits, leading to the possibility of protocol violations and obscure errors. Finally, there is no attempt to put the process of programming into the hands of the programmer. Generally speaking, large programs must still be written in their entirety before they can be compiled and tested.

Static object oriented languages provide only half of a solution. The other half is provided by *dynamism*, yielding what Apple calls *object oriented dynamic languages*, or *OODLs*. In addition to supporting the object oriented methodology, OODLs must support a number of features which guarantee that programming takes place in terms defined by the programmer, rather than in the terms of the hardware.

OODLs must support rapid creation, delivery, and subsequent modification of ambitious, reliable, and efficient software. Among the specific requirements an ideal OODL should satisfy are the following:

Automatic Memory Management

Memory management bugs are among the most common and difficult errors in static programming languages. Bugs involving dangling pointers and twice-freed objects are notoriously hard to track down.

The language run-time, and not the programmer, should be responsible for allocating storage for objects and reclaiming the storage of objects which are no longer used. There should be no explicit procedure calls for allocating or deallocating memory, or for deallocating objects.

In a well engineered implementation, automatic memory management should be robust and scalable. It should not create memory fragmentation, or fail in the presence of large (possibly virtual) address spaces. It should not cause seemingly arbitrary and unpredictable delays for end users.

In a true OODL, there should be no machine-level pointers, only objects. Once freed from dealing with pointers, the programmer can begin to think of objects at a higher level and the primitives become comparably richer. For example, a programmer working with collections of objects does not need to worry about memory leaks as collections expand and contract. The programmer can concentrate on the task at hand, rather than on the bookkeeping details.

Many large programming projects begin with the design of a memory management subsystem. There is no reason that this task should not be performed once, and the corresponding implementation of that design embodied in the language run-time, and thereby made available to all programmers.

Dynamic Linking/ Incremental Development

Programmers should have the ability to build up their programs piece by piece, integrating preexisting pieces when possible and where available. The transition between rapid prototyping and mainstream development should be continuous rather than discrete. It should not require changing languages or tools.

This requirement affects the programming process in at least four ways:

- During the initial construction of a program, classes and functions can be compiled and tested individually. This gives programmers the freedom to use a bottom-up programming style, if they so choose, obviating the need for the construction of a complex superstructure for initial testing.

- During debugging, individual functions and classes can be redefined without resorting to a full recompilation of the program and perhaps without even halting the execution of the program.
- Programs can be delivered in components, which can be linked together on either the development machine or the end-user machine, using built-in language features.
- Program patches (i.e., field upgrades) can be distributed to end-users using built-in language features. Applying the patch should be a very low overhead operation. There should be virtually no performance penalty for executing patched code. Because you don't need source code for the original application in order to apply the patch, traditional intellectual property issues arising from propagating source code simply need not arise.

Self Identifying Objects / Introspection

Operations should be checked for type safety before they are performed. If possible, this check should be performed at compile-time, otherwise it should be performed at run-time. This feature guarantees that type errors are noticed as soon as they occur, before they can propagate and cause system corruption. Because the integrity of the object model is maintained, error reporting can occur in terms of programmer objects and end-user objects, rather than in machine-level terms. In many cases, complete error recovery is possible.

The language should contain features for introspection. This means that the language run-time should have sufficient power to answer questions about itself and the objects it manages. For example, it should be possible at execution time to analyze the structure of an object, find the subclasses of a class, etc.

To facilitate type-safety and introspection, objects are self-identifying in memory. Unless all uses of an object can be analyzed at compile-time, the run-time memory for the object should contain enough information to identify its class and value.

Object Oriented Programming Environment

The programming environment should present all debugging information at an object oriented level. Errors should be described in high-level terms similar to those used by the programmer in constructing the program. Inspection facilities should show a program as a collection of objects, not as a mass of undifferentiated bits. There should be tools for performance analysis and monitoring of single objects as well as collections of objects.

There should be rich libraries of components, and the means to navigate within and between them and to organize and administer them.

There should be a well thought out distinction between development environment and execution environment, and where they are different, the development environment should manage the communication between them in as transparent a manner as possible. This feature is missing from current OODLs, but it is essential for the delivery of OODL-based applications to end-users.

An important aspect of these features is that they are mutually supporting, forming an *organic whole.* The simplest implementation of each depends on the existence of the others. For example, automatic memory management relies on the ability to walk memory and identify objects. Another example is that rapid prototyping requires rapid modification of the program, and so incremental dynamic linking becomes essential. Incremental dynamic relinking utilizes automatic reclamation of storage occupied by the functions, methods, and data being replaced.

Many people in commercial industry and in the research community have recognized the problems with static languages, and some have started to build products that provide some of the features of OODLs. For example, interactive programming environments for static languages are beginning to appear. We applaud this development. However, we believe that starting with a static language requires too many compromises. If each OODL feature were to be added in isolation, the mutually supporting characteristics would be lost, creating redundancy, conflicts, and inefficiency. When instead these features are built together into the core of a programming system, they provide a simple and secure foundation for growth.

The common criticism of OODLs is that we cannot afford them. Most programmers view OODLs as slow and as memory hogs. The common wisdom is that OODLs do not make good use of machine resources. Fortunately, this view is out of date. The combination of improved OODL implementation technology and increasingly powerful hardware make OODLs eminently practical. Every year or two our hardware gets twice as fast and has twice as much memory. Would anyone say that the quality of our software has increased at the same rate? Programming must be made easier, or the fastest hardware in the world will only give us incremental software improvements. By investing a few cycles we can enable a new generation of applications.

When Macintosh was first released, many people thought that windows, menus, and a bit mapped display were a waste of machine resources. We see in retrospect that this was not true. What good is the power of a computer, if it can't be accessed by a user?

This book describes Dylan™, a new object oriented dynamic language designed by Apple. Dylan is our attempt at a language which is simple, yet powerful, one which keeps programming at a high level but which can be compiled efficiently and has a relatively modest memory footprint.

Apple already has one OODL product: Macintosh Common Lisp. Dylan is intended to complement Common Lisp, not to replace it. Common Lisp is a rich environment defined by a standard and available in compatible implementations on a broad range of platforms. Dylan is lean and stripped down to a minimum feature set. At present Dylan is not available on any platform (outside Apple), but is intended to run on a wide variety of machines, including very small machines that don't have the horsepower to support a modern Common Lisp. Common Lisp is aimed primarily at the Lisp community, while Dylan is accessible to application developers unfamiliar with Lisp. Common Lisp is oriented more towards exploratory programming with delivery capability, while Dylan is oriented more towards delivery with exploratory capability.

In our research and development labs, Apple has its own implementations of Dylan, which are being produced hand in glove with the language definition as a design assurance measure. We would like to see others create additional implementations.[1] We would be particularly interested in working with a group to create a reference implementation optimized for portability rather than performance, so that anyone could try out Dylan no matter what brand of computer they use. Because Dylan is a trademark of Apple Computer, anyone implementing the language must obtain permission to use the name. However, it is our intention that permission will be granted to anyone with a conforming implementation.

Apple Eastern Research and Technology[2] will be the focal point for discussion of the Dylan language, its implementations, and its future evolution. Current and prospective users or implementors of Dylan, as well as programming language researchers who are interested in object oriented dynamic languages,

---

[1]Since this was written, such implementations are well underway on a variety of platforms.
[2]Now Apple Cambridge Engineering.

are invited to comment on the design of the language and suggest improvements that are consistent with the goals noted above. We will also continue our past policy of driving the evolution of the language from the needs and comments of our users, who are system and application developers whose previous experience is largely with conventional languages. We hope to integrate all inputs to make Dylan the best language we can and use it to bring the benefits of OODLs to as many programmers as we can possibly reach.

Ike Nassi
Cambridge, Massachusetts
March 1992

Acknowledgments

Designing Dylan has been a work of many hands.

The primary contributors to the language design were Kim Barrett, Glenn S. Burke, Robert Cassels, Bill Chiles, Jerome T. Coonen, Scott Fahlman, Paul Gleichauf, James Grandy, Paul Haahr, John Hotchkiss, Jeremy A. Jones, Michael Kahl, William Lott, Rob MacLachlan,David A. Moon, Ike Nassi, Jeffrey Piazza, Kent Pitman, Keith Playford, Andrew Shalit, Jonathan Sobel, Walter R. Smith, Bill St. Clair, Orca Starbuck, Oliver Steele, Robert Stockton, Steve Strassmann, Larry Tesler, Derek White, and Gail Zacharias.

Additional design work and oodles of helpful comments were provided by Jim Allard, Peter Alley, Jonathan Bachrach, Richard Barber, David Betz, John Barr, Alan Bawden, Stoney Ballard, Ernie Beernink, Brent Benson, Eric Benson, Rasha Bozinovic, Bill Campbell, Steve Capps, Yu-Ying Chow, Doug Cutting, Ken Dickey, Richard Duncan, Mikel Evins, Marc Feeley, Gregg Foster, Tom Gordon, Jed Harris, Alice K. Hartley, Alan Kay, Larry Kenyon, Mike Lockwood, Matthew MacLaurin, Robin Mair, Neil Mayle, Tim McInerney, Scott McKay, Jim Meehan, John Meier, Jim Miller, Richard Mlynarik, Robert Muller, Carl Nelson, Julian Padgett, Paige Parsons, Ed Petrus, Peter Potrebic, Jonathan Rees, Kalman Reti, David Rosenfeld, Carl Schwarcz, David Singer, Andy Sizer, David C. Smith, Andy Stadler, Joshua Susser, Michael Tibbott, Tom Vrhel, John Wainwright, Bob Welland, Paul Wilson, and Dan Zigmond.

Moral and logistical support were provided by Donna Auguste, Chrissy Boggs, Mickey Dennison, Gina Field, Rick Fleischman, James Joaquin, Rick LeFaivre, Ross Knights, Becky Mulhearn, David Nagel, Wendy Phillips, Mark Preece, Mary Reagan, Shane Robison, and Susan M. Whittemore.

The Dylan project is directed by Ike Nassi and Carl Schwarcz.

The first edition of the Dylan manual was written by Andrew Shalit with contributions from Jeffrey Piazza, David Moon, and Steve Strassmann. It was extensively revised by Orca Starbuck, with editorial help from Kim Barrett, David Moon, and Steve Strassmann to incorporate recent language changes.

The first edition of the manual was designed by Scott Kim and Steve Strassmann. The original cover art was designed by Scott Kim.

The Dylan project is funded by Apple Computer's AppleSoft Division.

1. Introduction

Manual Notation

This manual uses a small number of typographic conventions:

Body text appears in Roman.

First uses of terms appear in **Bold**.

Text which appears as it would be entered into the computer appears in `Courier`.

In the body of the manual, the following notation is used to describe syntax forms:

{...}	Curly braces indicate a group of items.
{…}*	Curly braces followed by an asterisk indicate that the contents can appear zero or more times.
{…}+	Curly braces followed by a plus sign indicate that the contents can appear one or more times.
[…]	Square brackets indicate that the contents are optional.
… | …	Items separated by a vertical bar are mutually exclusive. One or the other may appear.

The BNF grammar at the end of the manual uses its own notation which is different from the above.

Formal parameters (i.e., place holders for the actual text you would enter) appear in *Italic*. The names of parameters are often descriptive of the type of value acceptable as a value of the parameter. They will often match the names of classes, indicating a general instance of the class. For example, *number* indicates a general instance of the class `<number>`, and *string* indicates a general instance of the class `<string>`.

Interactions between a user and a Dylan interpreter are shown in a mixture of `Courier` and *`Courier Italic`*. `Courier` shows the text entered by the user, and `Courier Italic` shows the text printed by the interpreter. The question mark used in these sections is the interpreter's prompt.[3]

The error messages in the manual have been edited for brevity. The details of interactions with Dylan (prompt text, error message texts, etc.) will differ among implementations.

The functions `format` and `print` are used in some examples but are not part of Dylan.

[3]The term *interpreter* is used, even though most implementations of Dylan are expected to be compiled. In most cases, an interpreter will be implemented by compiling Dylan expressions and immediately executing the compiled code.

Background and Goals

Dylan is a general-purpose high-level programming language, designed for use both in application and systems programming. Dylan includes garbage collection, run-time type checking, selective dynamism, error recovery, and a module system. These features simplify programming and support attractive debugging and development tools.

The motivation for Dylan is that programs and large software systems have become too complex for traditional static programming languages. To create a new generation of software—and to ensure that this software is maintainable and extensible—requires a new, friendlier set of programming tools. The core of these tools is a simple and expressive language, one that is efficient but which protects the programmer from crashes and machine-level debugging. All programming should take place at an object-oriented level. Improved implementation techniques and increasingly powerful computer hardware make it possible to use fully dynamic object-oriented languages for a wide range of programming tasks.

Dylan is built from the ground up with a thoroughly integrated object model. Like some object-oriented extensions to Lisp, Dylan implements polymorphism through generic functions and class dispatch, rather than through a Smalltalk-style message-passing mechanism.[4] Generic functions provide a very convenient model for a broad range of programming tasks.

The target audience of Dylan is application developers now using languages such as C, C++, and Pascal who are ready to move up to Object Oriented Dynamic Languages. Dylan is not intended primarily for the Lisp or Smalltalk community, thus compatibility with Smalltalk or with Lisp dialects such as Scheme and Common Lisp is not a primary goal. Dylan has been rethought from the ground up, on a fully object-oriented foundation. A primary goal was to make the language as simple as possible, but no simpler. We have tried to avoid multiple ways to do the same thing, to omit features that are difficult for the average application developer to understand and use effectively, to leave out anything that we do not know how to implement efficiently in both space and time, and to provide a clear separation of the language executed at run-time from the development tools. At the same time, we've moved complicated low-level bookkeeping underneath the language and taken it out

[4] Notable Lisp systems that use generic functions are CLOS (Common Lisp Object System), Oak Lisp, and EuLisp.

of the hands of the application programmers, to make them more productive. The overriding goal of Dylan is rapid development and delivery of applications and application components on very small computers.

Language Overview

Dylan is built around two central concepts: **objects** and **functions**.

Objects are the data that exist in Dylan programs. Objects are mapped into a **class heterarchy**.[5] The class heterarchy describes the inheritance relationships among **classes**. The class `<object>` is the root of the heterarchy. Every class inherits from its **direct superclasses** (except for `<object>` which has no superclasses). Classes may also have **subclasses**, which inherit from them, either directly or indirectly. The **superclasses** of a class are the class itself, its direct superclasses, their direct superclasses, and so on back to the class `<object>`.

Every object in the Dylan world is a **direct instance** of exactly one class. This class is referred to as "the class of" the instance. For each object *O* that is a direct instance of class *C*, the object *O* is also an **indirect instance** of all the superclasses of *C* (except *C* itself). (It follows that every object in Dylan is an indirect instance of `<object>`.) The term **general instance** means "either a direct or indirect instance."

Dylan defines four categories of classes:

Abstract Class — A class that is not intended to have direct instances. Its general instances obey an interface implemented by subclasses. The opposite of an abstract class is a **concrete class**.

Instantiable Class — A class that can be used as the first argument to `make`. The opposite of an instantiable class is an **uninstantiable class**. Note that an abstract class may be instantiable.

Sealed Class — A class that cannot be subclassed, other than those subclasses explicitly defined in the same library. The opposite of a sealed class is an **open class**.

---

5 A heterarchy—also called a directed acyclic graph (DAG)—is like a hierarchy, except that nodes can have multiple parents. Dylan classes are in a heterarchy because Dylan supports multiple inheritance.

Primary Class A class that may be used only as the primary superclass in multiple inheritance. It is illegal for a class to have two primary superclasses unless one is a subclass of the other. The opposite of a primary class is a **free class**.

Functions are a class of objects used for performing actions in Dylan. Functions correspond to the functions, procedures, methods, and messages of other languages. Dylan supports two classes of functions: methods and generic functions.

A **method** is a basic callable unit of code. It contains a typed argument list and a code body. The types in the argument list are called the **specializers** of the method. A method can only be called with arguments that match the specializers of the argument list. (In the most common case, the specializer is a class, and the corresponding argument must be a general instance of the class.) When the specializers of a method match a set of arguments, the method is said to be **applicable** to the arguments. If a method is called with inappropriate arguments, an error is signaled.

A **generic function** contains zero or more methods. When a generic function is called, the arguments are compared to the specializers of all the methods in the generic function. The method with the most appropriate specializers is run. In this way, a generic function chooses a method based on the classes and identities of its arguments. This is the basic technique for object-oriented dispatch in Dylan.

In contrast to methods in other object-oriented languages, Dylan methods can also be called directly. They do not need to be added to generic functions. Used this way, methods play the role of blocks in Smalltalk, closures in Lisp, and functions in C.

Classes and functions are themselves first-class objects, which can be manipulated by programs.

2. Syntax

Lexical Notation

Here are the lexical conventions used in Dylan programs.

White space (including spaces, tabs, newlines, and newpage characters) serves as a delimiter. The amount of contiguous white space is not significant. Thus, any number of contiguous white space characters can be used for formatting.

`foo` — Variable names are a series of alphanumeric characters, and characters from the set **`! & * < = > | ^ $ % @ _ - + ~ ? /`**, that do not begin with one of the special characters **`- + ~ ? /`**. If a variable name begins with a numeric character, the name must contain two alphabetic characters in a row. Variable names are not case-sensitive.

`=` — Infix operators are the following one- or two-character sequences: **`+ - * / - = == < > <= >= ~= ~ & | ^`**.

`\=` — An infix operator variable name is the corresponding infix operator, prefixed with the infix operator escape character **`\`**. This allows an infix operator to be used wherever a variable name is allowed (for example, as the variable name in a **`define method`** or as a function argument).

`123`
`1.5`
`-4.0`
`+57`
`#x1f4e` — The syntax of numbers is described in the numerics section of this manual.

`"abc"` — Literal strings are delimited by double-quote marks. The characters between the quotes are the elements of the string. Within the string, white space is significant (it contributes to the string). Double quotes can be included in strings by preceding them with a backslash character. Within a string, backslash (**`\`**) has the general effect of quoting the following character. To include a backslash in

	a string, the backslash must be preceded by another backslash.
`#"Hello"`	Symbols are interned strings. They are preceded by a .i.sharp-sign. See the section on symbols and keywords for more details.
`Hello:`	An alternative syntax for symbols is "keyword syntax." See the section on symbols and keywords for more details.
`#(1, 2, 3)`	Literal lists are delimited by open and close parentheses, and preceded by a sharp-sign. The items between the parentheses are the elements of the list. They are separated with commas.
`#[1, 2, 3]`	Literal vectors (one-dimensional arrays) are delimited with square brackets, and preceded by a sharp-sign. The items between the square brackets are the elements of the vector. They are separated with commas.
`'M'`	Character constants are represented by delimiting the character with single quotes.
`//`	A pair of slashes indicates the start of a comment. The comment continues until the end of the line. At the start of a new line, regular parsing begins again.
`/* ... */`	Slash-star pairs delimit extended comments. These comments can run over multiple lines. Extended comments can be nested.
`#t`	The hash-sign t syntax is used to indicate the canonical true value.
`#f`	The hash-sign f syntax is used to indicate the canonical false value.

`#key`
`#rest`
`#next`
`#all-keys`
	`#key`, `#rest`, `#next` and `#all-keys` are used to indicate special tokens in parameter lists.

Naming Conventions

Several conventions for naming variables help programmers identify the purposes of variables. The names of variables do not affect the semantics of a program, but are simply used to improve readability.

Dylan uses the following naming conventions:

Variables used to hold classes begin and end with angle brackets.

```
<window>        <object>        <character>
<number>        <stream>        <list>
```

Module variables that are designed to have their values change in the course of a program (i.e., variables that are not read-only) begin and end with asterisks.

```
*parse-level*
*incremental-search-string*
*machine-state*
*window-count*
```

Program constants (read-only module variables) begin with a dollar sign.

```
$pi
$end-of-file
```

The names of most predicate functions end with a question mark. Predicates are functions which return a true or false value.[6]

```
subclass?       even?
instance?
```

The names of most operations that destructively modify data structures end with an exclamation point.[7] These names sometimes contrast with versions that allocate new memory for the result.

`reverse!` *(non-destructive version is* `reverse`*)*
`sort!` *(non-destructive version is* `sort`*)*

Operations that retrieve a value from a location are called **getters**. Operations that store into a location are called **setters**. In general, getters and setters come

---

[6]Three notable exceptions to the terminal question mark convention are <, >, and =.
[7]Notable exceptions to the terminal exclamation point convention are := and all the setter variable names.

in pairs. Setter variable names are derived by appending "`-setter`" to the corresponding getter variable name. Actually, this is not simply a convention. The rule is exploited to generate setter names from getter names automatically, and it is used to expand calls to `:=`.

Getter	*Setter*
`window-position`	`window-position-setter`
`table-size`	`table-size-setter`
`window-color`	`window-color-setter`

These two expressions are equivalent; they both set the color of `my-window` to `green`:

```
window-color-setter (green, my-window)
my-window.window-color := green
```

Expressions

Dylan programs are composed of **expressions**. When an expression is **evaluated**, it **returns** zero or more values. Evaluating an expression may have side effects and may involve evaluating sub-expressions.

An **outer expression** is an expression that is not a sub-expression of any expression. A **top-level expression** is either an outer expression or a direct sub-expression of a `begin` syntax form that itself is a top-level expression. (Such a `begin` cannot have local declarations in it, although the grammar allows it.) Definitions (see Syntax Forms, below) are restricted to be top-level expressions. Other expressions may appear either as top-level expressions, or as sub-expressions of other expressions.

There are four kinds of expressions: literal constants, variable references, function calls, and syntax forms.[8]

Literal Constants

A **literal constant** is an object that is specified explicitly in program text.

```
? "abc"
"abc"
? 123
123
```

[8]The term "syntax form" describes both special forms and macros. They are equivalent to macros in other languages.

```
? foo:
foo:
? 'M'
'M'
? #t
#t
? #f
#f
? #(1, 2, 3)
#(1, 2, 3)
```

Literal constants are immutable. The keys and elements of collections that are literal constants are immutable. Attempting to modify an immutable object has undefined consequences. Immutable objects may share structure. Literal constants that are = may or may not be ==.

Variable References

When an expression is a variable name, the expression indicates a variable reference. The variable name evaluates to the value of the variable.

Dylan supports two kinds of variables: lexical variables and module variables. Lexical variables are created locally. They roughly correspond to local variables in other languages. Module variables are created by using a definition such as `define variable` or `define method`. They roughly correspond to global variables in other languages.

```
? <window>
{the class <window>}
? concatenate
{the generic function concatenate}
? define variable my-variable = 25;
my-variable
? my-variable
25
? begin
    let x = 50;
    x + x;
  end;
100
```

Note that Dylan classes and functions are first-class objects and are named by variables, like other objects.

See the chapter on Variables for detailed information about lexical and module variables.

Function Calls

Function calls generally have the syntax

expression (arg_1, arg_2, … arg_n)

expression must evaluate to a function. This is the function to be called. The arguments to the function are the values of the elements in the list.

The arguments to a function are evaluated in left-to-right order. The function is evaluated before the arguments.[9] Once the function and all the arguments are evaluated, the function is called with the arguments.

A function call evaluates to the values returned by the function.

expression is commonly a variable name. In the following example, the function being called is the value of the variable `factorial`:

```
factorial(x, y)
```

However, the expression in the function position does not have to be a variable reference; it can be any expression that evaluates to a function. In this way, Dylan is like C and Scheme, and unlike Common Lisp. The following example has a complex expression in the "function position." The complex expression creates and returns a function which adds one to its argument. This function is then applied to 99.

```
? (method(x) x + 1 end) (99)
100
```

Syntax Forms

A **syntax form** is a form which has its own rule for evaluation. Syntax forms are introduced by **syntax operators**. Some examples of syntax operators are `define variable`, `for`, `method`, and `:=`.

Syntax forms may have different evaluation rules than function calls. In a function call, all the subexpressions are evaluated and passed to the function as arguments. In contrast, a syntax form can examine its subexpressions literally and can choose to evaluate some and give special meaning to others. The special evaluation rules for each syntax form are listed along with that form's definition.

[9] Except in the special slot reference syntax `argument.function`, where `argument` is evaluated first, followed by `function`.

Generally, the syntax operator is the first word or words appearing in the syntax form. For example, the `define variable` syntax form begins with the words `define variable`, and the`method` syntax form begins with the word `method`.

```
define variable foo = 3;

method (a :: <number>, b :: <number>)
   list (a - b, a + b);
end method;
// Creates an anonymous method, which expects 2
// numeric arguments.
```

However, the syntax operator does not have to be the first word in the form. For example, the assignment operator, :=, is an infix operator that is also a syntax form with special evaluation rules.

```
my-variable := 12;
// Sets the value to 12
```

There are three kinds of syntax forms:

- Most syntax forms are **macros** (introduced by **macro operators**). Some macros (like the `for` iteration construct) are part of the core Dylan language. Others are created by individual implementations or by programmers. Macros can be thought of as expanding into other, equivalent, expressions.
- **Special forms** (introduced by **special form operators**) are basic, built-in syntax forms that cannot be reduced to other expressions. Examples are `:=` and `method`.
- **Definitions** (introduced by **definition operators**) are declarative parts of a program. Definitions are restricted to be top level expressions (see the next section). They do not return values. Examples are `define method` and `define variable`.

Statement bodies

Many kinds of expressions include **implicit bodies**, which may contain one or more other expressions, separated by semicolons. When an implicit body is evaluated, the expressions in the implicit body are evaluated in order (left to right). The value of the implicit body is the value of the last expression.

The simplest expression containing an implicit body is `begin`.

`begin` [Special Form]
body
`end`
⇒ *values*

The *body* consists of any number of expressions separated by semicolons. `begin` executes the expressions in the *body* sequentially. The values of the last expression are returned. If there are no expressions in the *body*, `#f` is returned.

```
begin
  close(airlock.outer-door);    // this
  pressurize(airlock);          // is an
  open(airlock.inner-door);     // implicit body
end;
```

Programs only need to use `begin` explicitly in order to limit the scope of a lexical variable, or in situations where a single expression is expected, such as an argument to a function call. Many other expressions contain implicit bodies. For example, the body of a method definition is an implicit body, and the `if` statement has an implicit body in each sub-part.

```
if (moon-phase == #"full")
  wolf-form(werewolf);          // this is an
  howl-at-moon(werewolf);       // implicit body
else
  human-form(werewolf);
end if;
```

Some statements use the word `end` to mark the end of an implicit body. Other statements use other markers, such as the `else` that marks the end of the first implicit body in an `if` statement. The syntax description for each statement defines where implicit bodies may appear, and how the boundary is marked.

Semicolons

Multiple expressions, such as top-level expressions appearing in source files, are separated by a semicolon "`;`". Semicolons are also required to separate all but the last expression in implicit bodies. The semicolon after the last expression is optional.

Special function call syntax

Infix Operators

Certain predefined operators are called using standard algebraic syntax. Operators and their arguments must be separated by whitespace or parentheses. All binary operators are left-associative, except for the assignment operator `:=`, which is right-associative.

These are listed in descending order of precedence. Operators within a group share the same precedence.

```
unary          -              negative
unary          ~              not (boolean)

binary         ^              exponentiation

binary         *              multiplication
binary         /              division

binary         +              addition
binary         -              subtraction

binary         =              equality
binary         ==             identity
binary         ~=             not equals
binary         <              less than
binary         >              greater than
binary         <=             less than or equals
binary         >=             greater than or equals

binary         &              and
binary         |              or

binary         :=             assign
```

Except for the last three, all of these infix operators are functions. These functions are first class, like all Dylan functions, but in order to use one where a variable name is expected (for example, as the variable name in a `define method`, or as a function argument) the operator must be prefixed with the infix operator escape character (\).

The order of evaluation for general infix expressions is as follows:

In an expression of the form

```
arg1 op … op argn
```

$\texttt{arg}_1$ … $\texttt{arg}_n$ are evaluated in that order. The evaluation time of module variables that `op`'s are defined to invoke (e.g. the plus function for +), is unspecified.

The last three infix operators listed here (`&`, `|`, and `:=`) are syntax forms. They have special evaluation rules which are described where these forms are defined.

Slot Reference

Dylan provides a shorthand syntax for slot reference. The syntax `argument.function` applies *function* to *argument*. This syntax is usually used for slot reference, to access the *function* slot of the object *argument*.
This can be cascaded and is left associative (i.e., `moo.goo.gai.pan` gets the *pan* slot of the *gai* slot of the *goo* slot of *moo*.).
Examples:

```
? america.capital
"Washington, D.C."
? me.mother.current-husband.favorite-child == me
#t
```

Dylan's left-to-right evaluation rule applies here: `argument` is evaluated first, followed by `function`.

Array Reference

Dylan provides a shorthand syntax for array reference. The syntax `foo[i]` is equivalent to the function call `element(foo, i)`. The syntax foo[i_1, i_2, ... i_n] where $n \neq 1$ is equivalent to the function call aref (foo, i_1, i_2, ... i_n).

```
? define constant vect = #[7, 8, 9]
defined vect
? vect[0]
7
```

In expressions of the form

```
sequence[arg1, …, argn]
```

`sequence` is evaluated first, then `arg1 … argn` in that order.

The evaluation time of the variable <`element` in Dylan module> or <`aref` in Dylan module>, which this expression is defined to invoke, is unspecified.

Syntax of Dylan Files

A **Dylan source code file** is a standard portable file format for publishing Dylan source code. Such a file has two parts, the **header** and the **body**. The header comes before the body.

The body consists of zero or more outer expressions and comments.

The header consists of one or more keyword-value pairs, as follows:

1) The syntax of a keyword is as follows: a keyword is a letter, followed by zero or more letters, digits, and hyphens, followed by a colon, contains only characters from the ISO 646 7-bit character set, and is case-independent.

2) A keyword begins on a new line, and cannot be preceded by whitespace.

3) All text (excluding padding whitespace) between the keyword and the newline is considered to be the value. Additional lines can be added by having the additional lines start with whitespace. Leading whitespace is ignored on all lines.

4) Interpretation of the value is determined by the keyword.

5) Implementations must recognize and handle standardized keywords properly, unless the specification for a keyword explicitly states that it can be ignored.

6) When importing a file, implementations are free to ignore any non-standard keyword/value pair that they do not recognize.

7) When exporting a file, implementations must use standard keywords properly. Implementations are free to use non-standard keywords.

8) The definition of a keyword may specify that the keyword may occur more than once in a single file header. If it does not, then it is an error for the keyword to occur more than once. If it does, it should specify the meaning of multiple occurences.

The header cannot contain comments.

Blank lines may not appear in the header. A blank line defines the end of the header and the beginning of the code body. The blank line is not part of the code body. (A "blank line" is a line consisting of zero or more space or tab characters, ending in a newline character.)

The following standard keywords are defined:

`module:` *module-name* [Header keyword]

Expressions in the file are associated with the named module. This keyword is required.

`author:` *arbitrary text* [Header keyword]

`copyright:` *arbitrary text* [Header keyword]

`version:` *arbitrary text* [Header keyword]

These are provided for standardization. These are optional, and can be ignored by the implementation.

A typical Dylan source file might look like this:

```
module:         quickdraw
author:         J. Random Rect
                Linear Wheels, Inc., "Where quality is a slogan!"
                rect@linear.com
copyright: (c) 1993 Linear Wheels, Inc., All rights reserved
version:        1.3 alpha (not fully tested)

define constant $black-color = ...
...
```

3. Variables

Dylan supports two kinds of variables: **lexical variables** and **module variables**.

The term **environment** is used to refer to the set of variables that are available to a given part of a program. It includes both module variables and lexical variables.

Variables can be set to new values with the `:=` special form, which is described in this chapter's section on Assignment.

Module Variables

Module variables can be referenced from anywhere inside the module. They play the role assumed by global variables in other languages.

`define variable` *bindings* **`=`** *init* [Definition]

`define variable` creates module variables in the current module.

bindings are the variable(s) to be created, and mayhave the following forms:

variable

or **`(`** { *variable* }* [**`#rest`** *rest-variable-name*] **`)`**

where *variable* may be either *variable-name* or *variable-name* `::` *type*. The *variable-names* and *rest-variable-name* must have the syntax of variable names, and are not evaluated. The values returned by *init* provide the initial values for the variable(s) specified by *bindings*.

In the simplest case, *bindings* is just a variable name, and *init* returns one value which is used to initialize that variable. See Checking Types and Binding Multiple Values for more complex cases.

```
? define variable foo = 10;
foo
? foo + foo;
20
```

define constant *bindings* **=** *init* [Definition]

`define constant` creates read-only module variables in the current module. This form has the same syntax and semantics as `define variable` except that the variables created are read-only.

Several other forms create module variables. See `define class`, `define generic`, and `define method` for more information.

Source code is associated with a specific module through the programming environment. This association occurs at development time and cannot be changed at run-time. See the Modules chapter for more details about modules.

A given module variable can only be defined once, except that multiple `define method` definitions with different specializers are allowed, together with at most one `define generic` definition.

Lexical Variables

Lexical variables are created locally and can be referenced only in a limited range of program text. They correspond to local variables in other languages.

let *bindings* = *init* [Special Form]

`let` creates new lexical variables within the smallest enclosing implicit body containing the `let`.

bindings are the variable(s) to be created, and may have the following forms:

variable

or **(** { *variable* }* [**#rest** *rest-variable-name*] **)**

where *variable* may be either *variable-name* or *variable-name* **::** *type*. The *variable-names* and *rest-variable-name* must have the syntax of variable names, and are not evaluated. The values returned by *init* provide the initial values for the variable(s) specified by *bindings*.

In the simplest case, *bindings* is just a variable name, and *init* returns one value which is used to initialize that variable. See Checking Types and Binding Multiple Values for more complex cases.

```
? begin
    let foo = 35;
    foo + foo
  end
70
```

A lexical variable shadows any module variable with the same name and any surrounding lexical variable with the same name. This rule means that the innermost version of a variable is the one referenced.

```
? define variable foo = 10;
foo
? foo
10
? begin
    let foo = 20;
    let foo = 50;
    foo + foo
  end
100
? foo
10
```

The bindings introduced by the `let` are in effect until the end of the smallest enclosing implicit body containing the `let`.

Several other syntax forms, including `local` and `block`, also create lexical variables as part of their operation.

Method parameters are another example of lexical variables; they can be referenced only from the body of the method. See the chapter on Functions for more information on method parameters.

Checking Types

Variable bindings appearing in parameter lists and in statements like `let` and `define variable` may be specialized or unspecialized. Specializing a variable indicates that the variable may only hold values of the specified type. Specialized variables have the syntax *variable-name* `::` *type* . Leaving a variable unspecialized indicates that it may hold values of any type. Unspecialized variables simply appear as *variable-name*.

Each *type* is any expression that evaluates to a valid Dylan type (that is, an instance of `<type>`). The *types* are evaluated before the *init*, in the same environment as *init*. Each *type* specifies the type of the corresponding *variable*. Attempts to initialize or assign the *variable* to a value which is not an instance of the corresponding *type* results in a type error being signalled.

```
? begin
    let x :: <integer> = sqrt (2);
    x
  end

error: 1.4142135623730951 is not an instance of <integer>

? define variable foo :: <integer> = 10;
foo
? foo := sqrt (2)
10
error: 1.4142135623730951 is not an instance of <integer>
```

Binding Multiple Values

Evaluating a Dylan expression can yield one value, more than one value, or no values at all. This capability is called **multiple values**.

Multiple values are supported through two capabilities:

- The function `values` is used to return multiple values.
- In the *bindings* parts of statements such as `let, define variable`, and `define constant`, *bindings* may have one of the following forms:

  variable or **(** { *variable* }* [**#rest** *rest-variable-name*] **)**

  The second form is used to receive multiple values.

`values` `#rest` *the-values* ⇒ *the-values* [Function]

Returns *the-values* as multiple values.

```
? values(1, 2, 3);
1               // first value returned
2               // second value returned
3               // third value returned
```

When variables are initialized to values returned by an *init* in a statement such as `let`, `define variable`, and `define constant`, the number of *variables* and the number of values are compared:

- If there are the same number of *variables* and values, the variables are initialized to the corresponding values.

```
? begin
    let (foo, bar, baz) = values (1, 2, 3);
    list (foo, bar, baz)
  end
#(1, 2, 3)
```

```
? define method opposite-edges (center :: <number>,
                                radius :: <number>);
    let (min, max) = edges (center, radius);
    values (max, min);
  end method;
opposite-edges
? opposite-edges (100, 2);
102
98
```

- If there are more *variables* than there are values returned by *init* , the remaining *variables* are initialized to `#f`. (If a specialized *variable* defaults to `#f`, and `#f` is not an instance of that variable's type, an error is signaled.)
- If there are more values returned than there are *variables*, the excess values are placed in a sequence which is used as the initial value for *rest-variable*; if there is no *rest-variable*, these excess values are discarded.

```
? begin
    let (#rest nums) = edges (100, 2);
    nums;
  end
#(98, 102)
```

- If there is a *rest-variable* but there are no excess values, *rest-variable* is initialized to an empty sequence.

Multiple values can be used to perform parallel binding:

```
? begin
    let x = 10;
    let y = 20;
    let (x, y) = values (y, x);
    list (x, y);
  end
#(20, 10)
```

Assignment

The Dylan special form `:=` is used to set variables to new values. It can also be used as an alternate syntax for calling setter functions and macros.

place **:=** *new-value* ⇒ *new-value* [Special Form]

:= stores *new-value* in *place*. Subsequent evaluations of *place* will yield *new-value*. **:=** returns *new-value*.

place commonly has the syntax of a variable name. Dylan also allows extended formats (described below). If *place* is not a variable name or one of these extended formats, an error is signaled.

If *place* is a variable name, then *new-value* is stored in the corresponding variable (which may be a lexical or module variable). An error is signaled if there is no lexical or module variable corresponding to the *place* , if the corresponding variable is a read-only variable, or if the corresponding variable is specialized to some type and the *new-value* is not an instance of that type.

```
? define variable foo = 10;
10
? foo              // this is a variable
10                 // this is the variable's contents
? foo := 10 + 10;
20
? foo
20
```

Extended form

If *place* is not a variable name, then it may have the syntax of a call to a function. This is called the **extended form** of **:=**. In this case, the function call indicates an accessible location. The call to **:=** should change what future accesses of the location will return.

```
? define variable foo = vector (10, 6, 8, 5);
foo
? element(foo, 2)
8
? element(foo, 2) := "bar"
"bar"
? foo
#[10, 6, "bar", 5]
```

In the general case,

name (*arg1, ... argn*) := *new-value*

means roughly

name-setter (*new-value, arg1, ... argn*)

In functional := expressions

```
function-name(arg1, ..., argn) := value
```

where `function-name` is not a macro name, $\texttt{arg}_1$ ... $\texttt{arg}_n$ are evaluated first in that order, followed by `new-value`.

The evaluation time of the variable `function-name-setter`, which this expression is defined to invoke, is unspecified.

Extended form using array reference or slot reference syntax

The special syntactic shorthands for array reference and slot reference are also allowed as part of the extended form of :=.

For example, the two forms

```
foo[2] := "quux"
element (foo, 2) := "quux"
```

are equivalent to

```
element-setter ("quux", foo, 2).
```

In . := expressions

```
argument.function-name := value
```

where `function-name` is not a macro name, `argument` is evaluated first, followed by `value`.

The evaluation time of the variable `function-name-setter`, which this expression is defined to invoke, is unspecified.

In implicit element := expressions

```
sequence[arg1, …, argn] := value
```

`sequence` is evaluated first, then $\texttt{arg}_1$ … $\texttt{arg}_n$ in that order, then `value`. The evaluation time of the variable `element-setter`, which this expression is defined to invoke, is unspecified.

4. Control Constructs

True and False

In Dylan, there is a single object that counts as false. This object is indicated by the syntax **`#f`**[10]. All other values are considered true for the sake of true/false testing. The canonical true object, indicated by the syntax **`#t`**, can be used for clarity of code.

Throughout the manual, the phrases "true," "false," "returns true," and "returns false" are used. It's important to remember that "returns true" does not necessarily mean "returns the object `#t`." It simply means "returns any object besides `#f`." Sometimes for efficiency, debuggability, or informational value, some value besides `#t` is returned.

`#t` [Object]

This is the canonical true value. In Dylan, all objects besides `#f` count as true, not just `#t`.

`#f` [Object]

This is the only false value.

`~` *thing* ⇒ *boolean* [Function]

Returns true if *thing* is `#f`; otherwise returns false.

Conditionals

The following syntax forms are used to perform conditional execution.

[10]This is in sharp distinction to C, which equates 0 and the false value, and some dialects of Lisp, which equate the empty-list and the false value.

if **(** *test* **)** *consequent*
  [**elseif** **(** *elseif-test* **)** *elseif-consequent*]*
  [**else** *alternate*]
  end [**if**] ⇒ *values* [Special Form]

`if` evaluates *test*. If *test* evaluates to a true value, *consequent* is evaluated.

If *test* evaluates to `#f` (the only false value), and any `elseif` clauses were supplied, the first *elseif-test* is evaluated. If that *elseif-test* evaluates to a true value, its *elseif-consequent* is evaluated. If it evaluates to `#f`, the next *elseif-test* is evaluated, and so on.

Finally, if *test* and each *elseif-test* evaluates to #f, and *alternate* was supplied, the *alternate* is evaluated. If *alternate* was not supplied, #f is returned.

The *consequent* , *elseif-consequent*, and *alternate* sections of an `if` statement are all implicit bodies (may contain any number of expressions, separated by semicolons). The values of the last expression in the appropriate *consequent* , *elseif-consequent*, or *alternate* form (whichever is evaluated) are returned.

```
? define method test (thing :: <object>)
    if (thing)
      #t
    else
      #f
    end if
  end method;
test
? test ("hello")
#t
? test (#t)
#t
? test (#f)
#f


? define method show-and-tell (thing :: <object>);
     if (thing)
       print (thing);
       #t
     else
       #f
     end if
  end method;
show-and-tell
? show-and-tell("hello")
hello
#t
```

unless (*test* **)** [Macro]

body

end [**unless**]

⇒ *values*

`unless` evaluates *test*. *body* is an implicit body. If *test* evaluates to a true value, then none of the expressions in the *body* are evaluated and `#f` is returned. If *test* evaluates to `#f`, then all the expressions in the *body* are evaluated and the values of the last expression are returned.

If there are no expressions in the *body*, then `#f` is returned.

```
unless(detect-gas? (nose))
    light(match)
end unless
```

case [Macro]

test$_1$ **=>** *body*$_1$ **;**

...

test$_n$ **=>** *body*$_n$ **;**

[**otherwise =>** *otherwise-body*] [;]

end [**case**]

⇒ *values*

`case` evaluates the *tests* in order.

If a *test* returns true, the corresponding *body* is evaluated, and the last expression in the *body* is returned. If there is no *body* in the *test* that succeeds, then the first value of the *test* is returned. Subsequent *tests* are not evaluated.

If no *test* is true, then `case` returns `#f`.

As a special case, the keyword `otherwise` may appear as a *test*. This *test* always succeeds if there are no other successful *tests*.

```
case
   player1.money <= 0
     => end-game(player1);
   player2.money <= 0
     => end-game(player2);
   otherwise
     => move(player1);
        move(player2);
end case;
```

select (*target-form* [**by** *test*] **)** [Macro]

match$_{1a}$, … *match*-$_{1n}$ **=>** *body*$_{1}$;

…

match$_{ma}$, … *match*-$_{mn}$ **=>** *body*$_{m}$;

[**otherwise =>** *otherwise-body*] [;]

end [**select**]

⇒ *values*

Each *match-list* must be a sequence of expressions separated by commas, or the keyword `otherwise`. The *bodies* are implicit bodies (may be any number of expressions separated by semicolons).

When the `select` form is entered, the *target-form* is evaluated, generating a **target value**.

The clauses are tested in order, using the following procedure:

- Each *match* is evaluated to produce a match object. The match expressions are evaluated in order.
- If a *test* function was supplied, this is used to compare the target object to the match objects. Otherwise, the match objects are compared to the target value with ==.
- If a match is found, the corresponding *body* form is evaluated and the values of the last expression in the *body* are returned. If there is no *body* form in the matching clause, `#f` is returned.
- Once a match is found, the following match-object and body expressions, in this and subsequent clauses, are not evaluated at all.
- If there is no matching clause, an error is signaled.

As a special case, the keyword `otherwise` may appear instead of a match-list. This clause will be considered a match if no other match is found. Because an `otherwise` clause matches when no other clause matches, a `select` form that includes an `otherwise` clause will never signal an error for failure to match.

Since testing stops when the first match is found, it is irrelevant whether the test function would also have returned true if called on later match-list elements of the same clause or on match-list elements of later clauses.

```
select ( career-choice(student) )
   art:, music:, drama:
     => "Don't quit your day job";
   literature:, history:, linguistics:
     => "That really is fascinating";
   science:, math:, engineering:
     => "Say, can you fix my VCR?";
   otherwise => "I wish you luck";
end select;
```

Dylan's `select` with a test of `instance?` is similar in effect to a Common Lisp `typecase` statement.

```
select ( my-object by instance? )
  <window>, <view>, <rectangle> => "a graphical object";
  <number>, <string>, <list> => "a computational object";
  otherwise => "I don't know";
end select
```

form1 **|** *form2* ⇒ *values* [Macro]

| ("or") evaluates the *forms* from left to right. If *form1* returns true as its first value, that value is returned and *form2* is not evaluated. If *form1* returns `#f` as its first value, *form2* is evaluated and its values are returned.

form1 **&** *form2* ⇒ *values* [Macro]

`&` ("and") evaluates the *forms* from left to right. If *form1* returns `#f` as its first value, `#f` is returned and *form2* is not evaluated. If *form1* returns true as its first value, *form2* is evaluated and its values are returned.

Iteration Constructs

`while ( ` *test* **`)`** [Macro]
body
`end` [**`while`**]
⇒ `#f`

`while` evaluates the expressions in the *body* over and over until the *test* returns false.

Each pass through the loop begins by evaluating *test*. If *test* evaluates to a true value, the expressions in the *body* are evaluated and the looping continues. If *test* evaluates to `#f`, the loop terminates and `while` returns `#f`.

`until` **(** *test* **)** [Macro]
body
`end` [**`until`**]

⇒ `#f`

`until` evaluates the expressions in the *body* over and over until the *test* returns true.

Each pass through the loop begins by evaluating *test*. If *test* evaluates to `#f`, the expressions in the *body* are evaluated and the looping continues. If *test* evaluates to a true value, the loop terminates and `until` returns `#f`.

`for` **(** *clauses* [Special Form]
[{**`until`** | **`while`**}*end-test*] **)**
body
[**`finally`** *result-body*]
`end` [**`for`**]

⇒ *values*

`for` allows one or more *clauses*. Each *clause* controls one **iteration variable** throughout the iteration. The optional *end-test* controls whether the iteration continues or terminates. It does not control any iteration variables.

The kinds of *clauses* include **explicit step clauses**, **collection clauses**[11], and **numeric clauses**:

<u>Explicit step clauses</u>

{*variable* | *variable* **`::`** *type*} **`=`** *init-value* **`then`** *next-value*

<u>Collection clauses</u>

{*variable* | *variable* **`::`** *type*} **`in`** *collection*

<u>Numeric clauses</u>

{*variable* | *variable* **`::`** *type*} **`from`** *start*
[{**`to`** | **`above`** | **`below`**} *bound*]
[**`by`** *increment*]

[11]Note that collection clauses are implemented using forward-iteration-protocol.

Iteration with `for` proceeds through the following steps:

1) Evaluate the expressions that are evaluated just once, in left to right order as they appear in the `for` statement.
   - For explicit step clauses, these expressions are *type* and *init-value*.
   - For collection clauses, these are *type* and *collection*. If the value of *collection* is not a collection, signal an error.
   - For numeric clauses, these are *type*, *start*, *bound* if it is supplied, and *increment* if it is supplied. If *increment* is not supplied, it defaults to 1.

2) Bind the iteration variables of explicit step and numeric clauses.
   - For each explicit step clause, bind *variable* to the value of *init-value*. If *type* is supplied and the value of *init-value* is not of the specified type, signal an error.
   - For each numeric clause, bind *variable* to the value of *start*. If *type* is supplied and the value of *start* is not of the specified type, signal an error.

3) Check numeric and collection clauses for exhaustion. If a clause is exhausted, go to step 9.
   - A collection clause is exhausted if its collection has no next element.

   - Numeric clauses cannot be exhausted if *bound* is not supplied. If *bound* is supplied, the following table gives the conditions for exhaustion:

	increment >= 0	*increment* < 0
to	*variable* > *bound*	*variable* < *bound*
above	*variable* <= *bound*	*variable* <= *bound*
below	*variable* >= *bound*	*variable* >= *bound*

4) Bind the iteration variables of collection clauses. Fresh bindings are created each time through the iteration.
   - For each collection clause, bind *variable* to the next element of the collection for that clause. If *type* is supplied and this next element of the collection is not of the specified type, signal an error.

5) If *end-test* is supplied, evaluate it. The termination conditions depend on the symbol used to introduce the *end-test* in the `for` statement.
   - If the value of *end-test* is false and the symbol is `while`, go to step 9.
   - If the value of *end-test* is true and the symbol is `until`, go to step 9.

6) Execute the expressions in the *body* in sequence. The expressions in the *body* are used to produce side-effects.

7) Obtain the next values for explicit step and numeric clauses. Values are obtained in left to right order, in the environment produced by step 6.
   - For each explicit step clause, evaluate *next-value*.
   - For each numeric clause, add the values of *variable* and *increment*.

8) Bind the iteration variables of explicit step and numeric clauses to the values obtained in step 7. For each clause, if *type* is supplied and the next value for that clause is not of the specified type, signal an error. Fresh bindings are created each time through the iteration. After variables are bound, go to step 3.

9) Evaluate the expressions in the *result-body* in sequence. Bindings created in step 2 and 8 are visible during the execution of *result-body*. Bindings created in step 4 (i.e. the iteration variables of collection clauses) are not visible during the execution of *result-body*. The values of the last expression in the *result-body* are returned by the `for` statement. If there are no expressions in the *result-body*, `for` returns `#f`.

Examples of `for`:

```
for ( thing = first-thing then next(thing),
      until done?(thing) )
  do-some(thing)
end;
```

```
for (j :: <integer> from 0 to height)
  for (i :: <integer> from 0 to width)
   erase(i,j);
   plot (i,j);
  end for;
end for;
```

```
for (city in olympic-cities,
     year from start-year by 4)
  schedule-olympic-game(city, year)
  finally notify(press);
          sell(tickets);
end;
```

```
for (i from 0 below 100,
     zombies from 0 below 100,
     normals from 100 above 0 by -1)
   population[i] := zombies + normals
end;
```

Non-local Exits

`block ( `[*exit-var*]**` )`** [Special Form]
body
[**`cleanup`** *cleanup-clause* | **`exception`** *exception-clause*]*
`end` [**`block`**]
⇒ *values*

`block` is a construct which combines the functionality of non-local exits, protected forms, and exception handling. block executes the expressions in the *body* in sequence, and then the optional *cleanup-clauses*. Any number of *cleanup-clauses* may be provided, and they may be arbitrarily interleaved with *exception-clauses* (described in the chapter on Conditions). Normally, the value returned by `block` is the value of the last expression in the *body*. If there are no expressions in the *body*, `#f` is returned.

If *exit-var* is provided, *exit-var* is bound to an **exit procedure** during the execution of the *body* and *clauses*. At any point in time before the last *clause* returns, the exit procedure can be called. Calling the exit procedure has the effect of immediately terminating the execution of the *body*. The exit procedure accepts any number of arguments. These arguments are used as the return values of block. Calling an exit procedure is known as performing a **non-local exit.**

Generally, the *cleanup-clauses* are guaranteed to be executed. Even if one of the expressions in the *body* is terminated by a non-local exit out of the `block`, the *cleanup-clauses* are executed before the non-local exit can complete.

For example, the following code fragment ensures that files are closed even in the case of an error causing a non-local exit from the block body:

```
block (return)
  open-files();
  ...
  if (error)
    return(#f);
  end if;
  ...
  result
cleanup
  close-files();
end block
```

However, if one of the *cleanup-clauses* is terminated by a non-local exit out of the `block`, any following *cleanup-clauses* within the same `block` are not executed.

Restrictions on the use of exit procedures

The exit procedure is a first-class object. Specifically, it can be passed as an argument to functions, stored in data structures, etc. Its use is not restricted to the lexical body of the **establishing block** (the block in which that exit procedure was established). However, invocation of the exit procedure is valid only during the execution of the establishing block. It is an error to invoke an exit procedure after its establishing block has returned, or after execution of the establishing block has been terminated by a non-local exit.

In the following example, the `block` establishes an exit procedure and binds `bar` to that exit procedure. The `block` returns an anonymous method containing a call to `bar`. The anonymous method is then bound to `foo`. Calling the `foo` method is an error because it is no longer valid to invoke `bar` after its establishing block returns.

```
? define constant foo =
    block (bar)
       method (n) bar(n) end;
    end block;
? foo(5)
error or other undefined consequences
```

When an exit procedure is called, it initiates a non-local exit out of its establishing block. Before the non-local exit can complete, however, the cleanup clauses of **intervening blocks** (blocks that have been entered, but not exited, since the establishing block was entered) must be executed, beginning with the most recently entered intervening block. Once the cleanup clauses of an intervening block have been executed, it is an error to invoke the exit procedure established by that block. Finally, the cleanup clauses of the establishing block are executed, further invocation of the exit procedure becomes invalid, and the establishing block returns with the values that were passed to the exit procedure.

During the process of executing the cleanup clauses of the intervening blocks, any valid exit procedure may be invoked and may interrupt the current non-local exit.

5. Comparisons

Dylan provides an identity function, as well as a group of equality and magnitude comparison functions that can be extended for user classes. The methods ~=, >, <=, and >= are defined in terms of = and <. By extending the behavior of = and <, programs can extend the behavior of the other methods.

For the protocol to work, all methods on = and < must preserve the following properties:

Identity:	If (a == b), then (a = b).
Transitivity:	If (a < b) and (b < c), then (a < c).
	If (a = b) and (b = c), then (a = c).
Trichotomy:	Exactly one of: (a < b), (a = b), (b < a) always holds (on the assumption that these two operations are defined for the objects in question). [12,13]

Equality Comparisons

object1 **==** *object2* ⇒ *boolean* [Function]

== returns true if *object1* and*object2* are identical. Otherwise, it returns false.

Objects are considered identical if they are computationally equivalent. That is, there is no way for any possible Dylan program to distinguish them.[14]

---

[12]Trichotomy also implies antisymmetry [if `(a < b)`, then `~(b < a)`] and antireflexivity [if `(a == b)`, then `~(a < b))`]. It also implies commutativity for = [if `(a = b)`, then `(b = a)`].
[13]The trichotomy rule does not hold for IEEE floating-point comparisons. IEEE requires all comparison operations to return false if one of the operands is a NaN. This means that the generic Dylan equality and magnitude predicates will not be IEEE compliant.
[14]At an implementation level, this will usually mean that the objects are pointers to the same storage or are the same immediate value. An extension is made for built-in number classes and characters. Because these objects are not mutable (i.e., cannot be changed), two with the same value will always be the same (and will thus be indistinguishable to programs).

object1 **=** *object2* ⇒ *boolean* [Generic Function]

Programmers may define methods for = specialized on classes they define. A programmer may be required to provide a = method when defining subclasses of some predefined classes in order to fullfill the protocol of the class, as described below.

- Two collections are equal if they have identical `key-test` functions, they have the same keys (as determined by their `key-test` functions), the elements at corresponding keys are =, and neither is a dotted list.

-Two dotted lists are equal if they are the same size, corresponding elements are =, and the final tails are =.

- Numbers are equal if they have the same mathematical value.

- For objects which do not have a more specific = method, = returns the same as ==.

= is not guaranteed to return (for example, when called on circular structures or unbounded structures).

object1 **~=** *object2* ⇒ *boolean* [Function]

~= calls = on *object1* and *object2*. Returns true if = returns false.

~= is not a generic function, so you can't add methods to it. To extend the protocol, define methods on =.

Magnitude Comparisons

object1 **<** *object2* ⇒ *boolean* [Generic Function]

The predefined methods on < behave as follows:

— Built-in real numbers are compared by mathematical value.

— Characters are compared by the ordinal value of the underlying character set. Character case is significant.

— Strings are compared lexicographically, using < on corresponding elements. If one string is a strict prefix of the other, the shorter string is considered the “smaller” one.

object1 **>** *object2* ⇒ *boolean* [Function]

> compares the *objects* with < (with appropriate argument reordering). If *object1* is greater than *object2*, > returns true. Otherwise, > returns false.

> is not a generic function, so you can’t add methods to it. To extend the protocol, define methods on <.

object1 **<=** *object2* ⇒ *boolean* [Function]

<= compares the *objects* with < (with appropriate argument reordering). If *object1* is less than or equal to *object2*, <= returns true. Otherwise, <= returns false.

<= is not a generic function, so you can’t add methods to it. To extend the protocol, define methods on <.

object1 **>=** *object2* ⇒ *boolean* [Function]

>= compares the *objects* with <. If *object1* is greater than or equal to *object2*, >= returns true. Otherwise, >= returns false.

>= is not a generic function, so you can’t add methods to it. To extend the protocol, define methods on <.

6. Functions

Function Defining Forms

Generic Functions and G. F. Methods

The basic tools for defining generic functions and methods are the defining forms `define generic` and `define method`.

`define generic` is used to declare information about the generic function as a whole. It is not strictly necessary to use `define generic`; generic functions can be created with `define method` alone.

`define` [*adjectives*] **`generic`** *name parameter-list* [Definition]

`define generic` creates a new generic function object. *name* is a variable name, and is defined as a read-only variable in the current module, containing the new generic function.

The *adjectives* are words separated by spaces. The allowed *adjectives* are `open` and `sealed`. The *adjectives* control the level of dynamism of the generic function. See the Controlling Dynamism chapter for more details.

The *parameter-list* specifies what arguments are acceptable to the generic function, and constrains which methods may be added to the generic function. In its simplest form, the *parameter-list* is a list of variable names, separated by commas, which represent the required parameters of the generic function. Methods added to the generic function must have parameter lists that are congruent with the generic function's parameter list.

A generic function with no required parameters can contain a single method. Adding a method has the effect of replacing the old method.

A complete description of parameter lists and congruency is given in the Function Applicability section.

`define method` is used to add methods to a generic function. It can also be used to create a new generic function that does not already exist.

define [*adjectives*] **method** *name* *parameter-list* [Definition]

implicit-body

end [**method** [*name*]]

`define method` creates a method and adds it to the generic function in *name*. If the module variable *name* is not already defined, it is defined as with `define generic`. Thus, `define method` will create a new generic function or extend an old one, as needed. Methods added to a generic function must have parameter lists that are congruent with the generic function's parameter list. A complete description of congruency is given in the Function Applicability section.

The *adjectives* are words separated by spaces. The allowed *adjectives* are `open` and `sealed`. The *adjectives* control the level of dynamism of the named generic function. See the Controlling Dynamism chapter for more details.

The parameters of the method—and the specializers of the parameters—are described in the *parameter-list*. The specializers declare what kinds of arguments are acceptable to the method. The method can be called only with arguments that match the specializers of the parameters. A complete description of parameter lists is given in the Function Applicability section.

When the method is invoked, it executes the *implicit-body*. Statements in the *implicit-body* are executed in order. The method parameters are bound as lexical variables over the scope of the *implicit-body*.

The following example shows how a generic function `double` could be defined to work with some built-in classes.

You can define a method for `double` that is applicable when `double` is called on a number.

```
? define method double (thing :: <number>)
    thing + thing;
  end method;
double
? double
{the generic function double}
? double(10)
20
```

The generic function `double` is now defined, but because it has a method only for the class `<number>`, it can be called only on instances of `<number>`. If you try to call `double` with another class of argument, you will get an error.

```
? double("the rain in Spain.")
error: no method for {the generic function double} was found
       for the arguments ("the rain in Spain.")
```

To operate on a second class of arguments, you have to add another method to `double`.

```
? define method double (thing :: <sequence>)
    concatenate (thing, thing);
  end method;
double
? double("the rain in Spain.")
"the rain in Spain.the rain in Spain."
```

Bare Methods

In Dylan, methods can also be created and used directly:

- To perform a local calculation.
- For a function that does not require any class dispatch.
- To build up generic functions programmatically.

The basic tool for creating methods is the special form `method`.

`method` *parameter-list* [Special Form]

implicit-body

`end` [**`method`**]

⇒ *method*

`method` creates and returns *method*. *method* is a method that accepts the arguments described by *parameter-list* and then executes the *implicit-body*. Statements in the *implicit-body* are executed in order. The *parameter-list* describes the parameters for *method*. A complete description of parameter lists is given in the Function Applicability section.

Here is an example:

```
? method(num1, num2)
    num1 + num2
  end method
{an anonymous method}

;the second argument to SORT is the test function
? sort(person-list,
       method(person1, person2)
          person1.age < person2.age
       end method)

? begin
    let double = method(number) number + number end;
    double(double(10))
  end
40
```

The following example defines a method for `double` that works on functions. When you double a function, you get back a method that accepts arguments and calls the function twice, passing the arguments both times.

```
? define method double (my-method :: <function>)
    method (#rest args)
      apply (my-method, args);
      apply (my-method, args);
      #f
    end method
  end method;
double
? define constant print-twice = double(print)
print-twice
? print-twice
{an anonymous method}
? print-twice("The rain in Spain. . .")
The rain in Spain. . .The rain in Spain. . .
#f
? print-twice(55)
5555
#f
```

Bare methods may be called directly or they may be added to generic functions.[15] Usually, however, when you want to add a method to a generic function, you create and add the method in a single step, with `define method`.

[15]When methods are called directly, the `method` special form corresponds to the `lambda` special form of Lisp and to blocks in Smalltalk.

`local` *method-spec*$_1$, *method-spec*$_2$ `, …;` [Special Form]

`local` can be used for creating self- and mutually-recursive methods.

Each *name* is bound to a method specified by the corresponding *method-spec* . Each *method-spec* has the same syntax as the `method` form, except that the word `method` is optional, and a variable name precedes the parameter list.

`local` is similar to `let`, in that it binds variables within the current scope (usually the immediately enclosing `begin...end` block). The scope of the *names* also includes the parameter lists and the bodies of the methods. This means that the methods can refer to themselves and to the other methods created by the `local` form.

```
? define method root-mean-square (s :: <sequence>)
     local method average (nums)
             reduce1 (\+, nums) / size (nums)
           end average,
           method square (n)
             n * n
           end square;
     sqrt (average (map (square, s)))
  end method root-mean-square

? root-mean-square (#(5, 6, 6, 7 ,4))
5.692099788303083

? define method newtons-sqrt (x)
     local method sqrt1 (guess)
             if (close? (guess))
                guess
             else
                sqrt1 (improve (guess))
             end if
           end sqrt1,
           method close? (guess)
             abs (guess * guess - x) < .0001
           end close?,
           method improve (guess)
             (guess + (x / guess)) / 2
           end improve;
      sqrt1 (1)
   end method newtons-sqrt

? newtons-sqrt (25)
5.000000000053723
```

Parameter Lists

Dylan parameter lists support **required parameters**, **rest parameters**, **keyword parameters**, and sometimes a special **next-method parameter**. They also may include **return type declarations**.

Required parameters correspond to arguments supplied when a function is called. The arguments are supplied in a fixed order and must appear before any other arguments.

Each required parameter may be a variable name, or a variable name **specialized** by a type (using the syntax *variable* `::` *type*). This has the effect of declaring that supplied arguments must be general instances of that type. The singleton specializer syntax can also be used (*variable* `==` *expression*, which is equivalent to *variable* `:: singleton (`*expression*`)`).

A rest parameter allows a function to accept an unlimited number of arguments.[16] After the required arguments of a function have been supplied, any additional arguments are collected in a new sequence, which is passed as the value of the rest parameter.

Keyword parameters correspond to arguments that are optional and may be given in any order. Symbols are used among the arguments to guide matching of argument values to parameter variables. We say that "the arguments are keyed by symbols." These symbols are usually written in keyword syntax and so they are informally known as keywords. The arguments and parameters are matched up by keyword name. Keyword arguments can only be supplied after all required arguments are supplied.

Required parameters come first in the parameter list, followed by the rest parameter, if any, and then the keyword parameters, if any. A rest parameter is indicated by the token `#rest`, followed by the name of the parameter. Keyword parameters are indicated by the token `#key`, followed by the keyword parameter specifiers, optionally followed by the token `#all-keys`.

If `#rest` and `#key` are used in the same parameter list, `#rest` must come first. The rest parameter will be bound to a sequence containing all the keywords and their corresponding values.

A next-method parameter is indicated by the token `#next`, followed by the name of the parameter. It is not normally necessary to specify a next-method parameter explicitly. If a next-method parameter is not specified by the

[16]In practice, an implementation may place a reasonable limit on the number of arguments that may be passed to any function.

programmer, `define method` inserts one with the name `next-method`. If an explicit next-method parameter is given, it must come after the required parameters and before the rest and keyword parameters. Details of using `next-method` are given in the section on Calling More General Methods.

The following are sample parameter lists as they would appear in a method definition.

`(a, b, c)`	three required parameters
`(a :: <list>, b, #rest c)`	two required parameters and a rest parameter
`(#rest a)`	a rest parameter, no required parameters
`(a, b, #key foo, bar)`	two required and two keyword parameters
`(#key bar, baz, bim)`	three keyword and no required parameters
`(#rest all, #key fee, fi)`	a rest parameter and two keyword parameters

Argument Passing Protocol

The argument passing protocol for a generic function or method can be described in one of the following ways, depending on its parameter list:

- It is said to **accept keyword arguments** if its parameter list specifies `#key`. The parameter list could also specify `#rest`.
- A function that accepts keyword arguments is said to **accept all keyword arguments** if its parameter list specifies `#all-keys`.
- It is said to **accept a variable number of arguments** if its parameter list specifies `#rest` but does not specify `#key`.
- It is said to **require a fixed number of arguments** if its parameter list does not specify either `#rest` or `#key`.

A method that accepts keyword arguments is said to **recognize** the keywords mentioned in its parameter list. (A method may, of course, mention them in the parameter list and then explicitly ignore the values.) It is possible for a method to accept keyword arguments in general, but not recognize any particular keywords.

A generic function that accepts keyword arguments may specifiy a set of keywords which must be recognized by every method added to the generic function. These are the **mandatory keywords** for the generic function.

A function that accepts keyword arguments is said to **permit** a keyword argument if the function is a method that recognizes the keyword, the function is a generic function and the keyword is recognized by any of the applicable methods, or the function accepts all keyword arguments.

If a function that accepts keyword arguments is called, it will signal an error if called with a keyword argument that it does not permit, or if the arguments following the required arguments are not keyword/value pairs.

If a method is called via a generic function or via next-method (rather than directly), the method itself does not check whether it received any keyword arguments it does not recognize, nor does it check that the arguments following the required arguments are keyword/value pairs.

A call to a function may supply the same keyword argument more than once. When this is done, the value of the leftmost occurrence is used.

Specializing Required Parameters

When you define a generic function or method, you may specify the classes or identities of the arguments appropriate for the generic function or method. This is called **specializing** the generic function or method.

The following example defines a method on `double` which is applicable when `double` is called with a general instance of `<number>`. The instance can be a direct instance or an indirect instance of the specializer class. In the example, the argument can be an integer, a float, or any other kind of number.

```
? define method double (thing :: <number>)
    thing + thing
  end method

? double(10)
20
? double(4.5)
9.0
```

Specialization restricts the arguments that may be passed as the value of the parameter. The function can be called only with arguments that match the specializers of the corresponding parameters. If a specializer is a class, the corresponding argument must be a general instance of the class. If the specializer is a **singleton** (used to indicate an individual object), the corresponding argument must be the singleton's object.

Providing this information offers two key benefits:

- It indicates to the compiler (and to anyone reading your code) that an error is signalled if arguments of the wrong type are provided.
- It allows the compiler to optimize your code and possibly improve its performance, due to the restrictions it can infer from the type information you provide.

Specialized parameters are also used in method dispatch. A generic function chooses among its methods on the basis of the methods' specializers. The generic function chooses the method whose specializers most closely match the classes and identities of the actual parameters.

The following are examples of parameter lists that include specializers:

`(a :: <window>, b, c)`	Three required arguments. The first must be a window.

`(a :: <window>,`
` b :: <character>,`
` c)`
	Three required arguments.
	The first must be a window and the second a character.

`(a :: <window>,`
` b :: <character>,`
` c :: singleton (0))`
	Three required arguments.
	The first must be a window, the second a character, the third the integer 0.

`(a, b :: <string>,` ` #rest c)`	Two required arguments and a rest argument. The second argument must be a string.
`(a :: <vector>, b,` ` #key foo, bar)`	Two required and two keyword arguments. The first required argument must be a vector.

Specializers are evaluated once, when a method is created. They are not reevaluated each time the method or containing generic function is called.

Specializers will usually be module-variables or singleton specializers. However, they are not restricted to these forms. A specializer can be any expression that evaluates to a type.

Dylan also allows a special syntax for **singleton specializers**, which is equivalent to explicitly using a singleton as a specializer:

`(a :: <window>,`
` b :: <character>,`
` c == 0)`
	Three required arguments.
	The first must be a window, the second a character, the third the integer 0.

Keyword Parameters

When defining a method that includes keyword parameters, each keyword specifier must have one of the following forms:

	name
or	*name* (*default*)
or	*keyword parameter*
or	*keyword parameter* (*default*)

In the first two forms, the *name* is used to indicate both the keyword and the parameter. In the last two forms, the keyword and parameter are given independently. The keyword is used when calling the method, and the parameter is used to refer to the value inside the body of the method.

The *default* supplies a default value for the argument. It is used when the method is called and the keyword is not supplied. The *default* should be an expression. It is evaluated each time the method is called and the corresponding keyword argument is not supplied. If no *default* is specified, the parameter corresponding to an unsupplied keyword argument is initialized to `#f`. The *default* is evaluated in a scope that includes all the preceding parameters, including required parameters, the rest parameter (if any), the preceding keyword parameters, and the next-method parameter (if any).

For example:

```
define method percolate (#key brand (#"maxwell-house"),
                              cups (4),
                              strength (#"strong"))
  make-coffee (brand, cups, strength);
end method;

define method layout (widget, #key position: the-pos,
                                   size: the-size)
  let the-sibling = sibling (widget);
  unless (the-pos = position (the-sibling))
    align-objects (widget, the-sibling, the-pos, the-size);
end method;
```

These methods could be called as follows:

```
percolate (brand: #"folgers", cups: 10);
percolate (strength: #"weak",
           brand: #"tasters-choice",
           cups: 1);

layout (my-widget, position: point (10, 10),
                   size: point (30, 50));
layout (my-widget, size: query-user-for-size() );
```

The extended syntax for declaring keyword arguments (in which the keyword name and parameter name are given separately) is needed to allow keyword names such as position: without forcing the method to use position as a parameter name. If a method uses position as a parameter name, it cannot access the function stored in the module variable position. The lexical variable will shadow the module variable.

All required arguments must be supplied before any keyword arguments can be supplied.

Result Values

Parameter lists may include value declarations, which must come at the end of the parameter list and are separated from the parameters by =>. A value declaration can be of the form *variable-name* :: *type-expression*, or just *variable-name* if the *type-expression* is <object>, just like a required parameter. The result of evaluating the *type-expression* at the time the function is defined is a type, called a “value type.” The *variable-name* never comes into scope, so it is just there for documentation and for syntactic consistency with parameters. It is valid for the same variable name to be used in both one parameter and one value declaration in the same parameter list; this is useful as documentation that a function returns one of its arguments, for example.

The last value declaration can be preceded by #rest to indicate a variable number of values returned. A value declaration preceded by #rest is called a “rest value declaration.” A value declaration not preceded by #rest is called a “required value declaration.” The value type in a rest value declaration is the type of each one of the remaining individual values, not the type of a conceptual sequence of multiple values.

If a parameter-list does not contain =>, it defaults to => #rest x :: <object>, i.e. the function can return any number of values of any type.

A function must always return the number and types of values declared in its parameter-list. More precisely:

- Each value returned by a function must be an instance of the corresponding value type, or else an error of type `<type-error>` will be signalled.
- If fewer values are returned by the function's body (or by the applicable method if the function is a generic function) than the number of required value declarations in the function's parameter-list, the missing values are defaulted to `#f` and returned. If `#f` is not an instance of the corresponding value type, an error of type `<type-error>` will be signalled.
- If a function does not have a rest value declaration, and more values are returned by the function's body (or by the applicable method if the function is a generic function) than the number of required value declarations in the function's parameter-list, the extra values are discarded and not returned.

Generic functions can have value declarations in the parameter list used in `define generic`. The values returned by the generic function will be instances of the value types. Rather than adding run-time checking to the generic function dispatch, the parameter list congruency rules are augmented to require each method added to the generic function to have congruent value declarations. `add-method`, `define method`, `define class`, etc. signal an error if this requirement is violated.

If a generic function is implicitly defined by `define method` or `define class`, it is not given any value declarations, so it defaults to `=> #rest x :: <object>`, which imposes no restrictions on its methods.

Parameter List Congruency

For any given generic function, the generic function and all methods for that function must have **congruent parameter lists**. Two parameter lists are congruent if they satisfy the following conditions:

1. They have the same number of required arguments.
2. Each of the method's parameter specializers is a subtype of the corresponding parameter specializer of the generic function.
3. Either
    - both accept keyword arguments,
    - both accept a variable number of arguments,
    - or both require a fixed number of arguments.

4. If the generic function accepts keyword arguments, each method must recognize the mandatory keywords of the generic function.
5. If a method accepts all keyword arguments, then the generic function must accept all keyword arguments.
6. In addition, the value declarations must be congruent, defined as follows:

   If the generic function's parameter list does not contain a rest value declaration, then:

   - the method's parameter list must not contain a rest value declaration.
   - the two parameter lists must contain the same number of required value declarations.
   - each value type in the method's parameter list must be a subtype of the corresponding value type in the generic function's parameter list.

   If the generic function's parameter list contains a rest value declaration, then:

   - the method's parameter list is permitted, but not required, to contain a rest value declaration.
   - the method's parameter list must contain at least as many required value declarations as the generic function's parameter list.
   - each value type in the method's parameter list must be a subtype of the corresponding value type in the generic function's parameter list. If the method has a rest value type, it corresponds to the generic function's rest value type. If the method has more required value types than the generic function, the extra ones correspond to the generic function's rest value type.

Method Dispatch

When a generic function is called, the generic function uses the classes and identities of the arguments to determine which methods to call. This process is called method dispatch.

Method dispatch occurs in three phases. First, all the applicable methods are selected. Next, the applicable methods are sorted by specificity. Then the most specific method is called.

Method specificity[17]

This section covers how methods are sorted by specificity.

For any two methods A and B that are applicable to a given generic function call, one method may be **more specific than** the other, or the methods may be **ambiguous methods**.

To order two methods A and B with respect to a particular set of arguments, compare each of A's specializers with B's specializer in the corresponding position using the corresponding argument. The comparison works in the following way.

- If the specializers are type equivalent, then A and B are unordered at the current argument position. That is, this argument position provides no information about the order of the two methods.
- Otherwise, if the specializer of A is a subtype of the specializer of B, then A precedes B at the current argument position. Similarly, B precedes A at this position if B's specializer is a subtype of A's specializer.
- Otherwise, if both specializers are classes, then their order in the **class precedence list** of the argument's class is used to determine which is more specific. (See the next section, Computing the Class Precedence List, for more detail.) If A's specializer precedes B's specializer in the class precedence list of the argument's class, then A precedes B at the current argument position. Similarly, B precedes A at this position if B's specializer precedes A's in the class precedence list of the argument's class.
- Otherwise, the methods are ambiguous methods.

The method A is more specific than the method B if and only if A precedes B or is unordered with respect to B in all required argument positions, and precedes B in at least one argument position. Similarly, B is more specific than A if and only if B precedes A or is unordered with respect to A in all required argument positions, and precedes A in at least one argument position. If

[17] This section is similar to the approach taken in Craig Chambers' language Cecil (U. Washington). This approach is different from CLOS in that there is no reference to the lexicographic order of arguments when multimethods are sorted. See the examples for more detail on these differences.

neither of these cases apply, i.e. neither method is more specific than the other, then A and B are ambiguous methods.

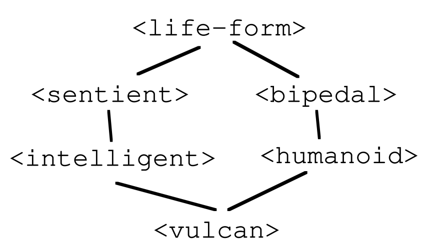
When the applicable methods are sorted by specificity, the sorted list is divided into two parts, each possibly empty. The first part contains methods that are more specific than every method that follows them. The second part (which cannot be sorted itself) begins at the first point of ambiguity; there are at least two methods that could equally well go first in the second part. If the first part is empty, generic function dispatch signals an "ambiguous method" error. If the last method in the first part calls `next-method`, the call signals an "ambiguous next method" error.

Examples of method specificity

Consider the following class definitions

```
define class <sentient> (<life-form>) ... end class;
define class <bipedal> (<life-form>) ... end class;
define class <intelligent> (<sentient>) ... end class;
define class <humanoid> (<bipedal>) ... end class;
define class <vulcan> (<intelligent>, <humanoid>) ... end class;
```

which form the class relationships shown in the following graph:



Computing the class precedence list for `<vulcan>` yields

```
(<vulcan>,<intelligent>,<sentient>,<humanoid>,<bipedal>,<life-form>)
```

In this class precedence list, note that classes in the simple superclass chains `(<intelligent>,<sentient>)` and `(<humanoid>,<bipedal>)` are kept adjacent.

The class precedence lists computed for two different classes may have different precedence orders for some intermediate superclasses. This is not a problem as long as there is no class which inherits from both classes. For example, we could define a class `<human>` as follows:

```
define class <human> (<humanoid>, <intelligent>) ... end class;
```

For the class <human> defined as above, the class precedence list would be

```
(<human>,<humanoid>,<bipedal>,<intelligent>,<sentient>,<life-form>)
```

It is not a problem that the two class precedence lists give different orders to some of the intermediate superclasses such as <bipedal> and <sentient> as long as no class is added which inherits from both <vulcan> and <human>.

When sorting the applicable methods, each specializer pair needs to be compared with respect to the class precedence list for the class of the argument passed to the generic function in that argument position, because the class precedence list might be different depending on which class it was computed from. For example, given the following definitions

```
define class <vulcan> (<intelligent>, <humanoid>) end class;
define class <human> (<humanoid>, <intelligent>) end class;
define method psychoanalyze (being :: <intelligent>) ...
   end method;
define method psychoanalyze (being :: <humanoid>) ...
   end method;
```

calling the generic function psychoanalyze on a being of type <human> would cause the method for <humanoid> to be called first, while calling the generic function on a being of type <vulcan> would cause the method for <intelligent> to be called first.

There is no reference to lexicographic order in sorting multimethods.[18] Given the above class definitions, the following methods are unambiguous when superior-being is called on two beings of type<vulcan> or two beings of type <human>, but the methods are ambiguous when superior-being is called on a being of type <vulcan> and a being of type <human> (because it is ambiguous whether to choose the most-intelligent-being, or the best-looking-being):

[18]This is a difference from CLOS. Under the CLOS system, the example would work as follows: when superior-being is called on a being of type <vulcan> and a being of type <human>, the best-looking-being is chosen when the <human> is the first argument, and the most-intelligent-being is chosen when the <vulcan> is the first argument.

```
define method superior-being (a :: <intelligent>,
                               b :: <intelligent>)
  most-intelligent-being (a, b)
  end method;

define method superior-being (a :: <humanoid>,
                               b :: <humanoid>)
  best-looking-being (a, b)
  end method;
```

Computing the Class Precedence List[19]

The definition of a class specifies a total ordering on that class and its direct superclasses. This ordering is called the **local precedence order**. In the local precedence order, the class precedes its direct superclasses, and each direct superclass precedes all other direct superclasses following it in the sequence of direct superclasses of the class definition.

The **class precedence list** for a class C is a total ordering on C and its superclasses that is consistent with the local precedence orders for each of C and its superclasses.

Sometimes there are several possible total orderings on C and its superclasses that are consistent with the local precedence orders for each of C and its superclasses. Dylan uses a deterministic algorithm to compute the class precedence list, which chooses one of the possible total orderings. The algorithm has the effect that the classes in a simple superclass chain are adjacent in the class precedence list, and classes in each relatively separated subgraph are adjacent in the class precedence list.

Sometimes there is no possible total ordering on C and its superclasses that is consistent with the local precedence orders for each of C and its superclasses. In this case, the class precedence list cannot be computed, and Dylan signals an error.

To compute the class precedence list:

Let S be the set of C and all of its superclasses.

For each class c in S, let

$$R_c = \{(c, c_1), (c_1, c_2), \ldots, (c_{n-1}, c_n)\}$$

where $c_1, \ldots, c_n$ are the direct superclasses of c in the order in which they are mentioned in the class definition for c.

[19] This section is intended to present the same algorithm as in CLOS.

Let R be the union of all R_c for every class c in S. The set R might or might not generate a partial ordering, depending on whether the R_c are consistent; it is assumed that they are consistent and that R generates a partial ordering. When the R_c are not consistent, it is said that R is inconsistent.

To compute the class precedence list for C, topologically sort C and its superclasses with respect to R. Topological sorting proceeds in the following way:

1. Find a class N in S such that no other class precedes N according to the elements in R.

   If there are several classes from S with no predecessors, select the one that has a direct subclass rightmost in the partial class precedence list computed so far. In more precise terms, let $\{N_1,...N_m\}$, $m \geq 2$, be the classes from S with no predecessors. Let $(C_1,...,C_n)$, $n \geq 1$, be the partial class precedence list computed so far. C_1 is the most specific class, and C_n is the least specific. Let $1 \leq j \leq n$ be the largest number such that there exists an i where $1 \leq i \leq m$ and N_i is a direct superclass of C_j. Select N_i.

2. Place N first in the result. Remove N from S, and remove all pairs of the form (N, M), such that M is an element of S, from R.

3. Repeat the process, adding classes with no predecessors to the end of the result. Stop when no element can be found that has no predecessor.

If S is not empty and the process has stopped, R is inconsistent because it contains a loop. That is, there is a chain of classes $C_1,....,C_n$ in S such that C_i precedes C_{i+1}, $1 \leq i < n$, and C_n precedes C_1. When R is inconsistent, there is no possible total ordering on C and its superclasses that is consistent with the local precedence orders for each of C and its superclasses, and an error is signaled.

Calling More General Methods

In many situations, a subclass wants to modify the behavior of a method, rather than replace it completely; it wants to perform some work but also use the inherited behavior. This can be accomplished with **`next-method`**. `next-method` is a function that, when called, invokes the next most specific method applicable in the generic function. The next-method is the value of the `#next` parameter. Normally this parameter is named `next-method`, though it can have other names at the programmer's discretion.

In the following example, the `double` method for `<float>` prints out a notice and then calls `next-method`, which invokes the next most specific method, in this case the method for `<number>`.[20]

```
? define method double (num :: <float>)
    print("doubling a floating-point number")
    next-method()
  end method;

? double(10.5)
doubling a floating-point number
21.0
```

If there are no more methods available, the next-method parameter will be bound to the value `#f` instead of to a method.

Passing Different Arguments to Next-Method

In the usual case, `next-method` is called with no arguments. This indicates that the next-method should be passed the same arguments that were supplied to the current method.

It is valid to supply arguments, including different arguments, when calling next-method. However, if you pass different arguments, the new arguments must result in the same ordered sequence of applicable methods, in the same order, as the original arguments. Otherwise, the behavior of Dylan is undefined.

In some cases, the methods in a generic function accept different keyword arguments. In such cases, it's convenient for the methods also to accept a rest parameter. That way, *all* the non-required arguments to the generic function are captured in the rest parameter. By using `apply`, the next-method can be invoked with the complete set of arguments.

The Next-Method Parameter

The next-method parameter is passed behind the scenes. When a method is called by its generic function, the generic function dispatch mechanism automatically passes the appropriate value for next-method. There is no way for a user program to specify the next-method argument when calling a method.

[20]Another way to think of `next-method` is that it invokes the method that would have been invoked if the current method did not exist.

If you create a method directly (i.e., with `method` rather than with `define method`) and you want this method to accept a next-method parameter, then you should insert a `#next` into the parameter list explicitly. You would do this if you are creating a method that you plan to add to a generic function, and you want this method to be able to call next-method. You can also supply the next-method parameter when using `define method`, in cases where you want to give the parameter a different name.

Reflective Operations on Functions

The following functions don't need to be called in routine programming, but are useful for implementation of the language and in the construction of the programming environment.

`generic-function-methods` *generic-function* [Function]

⇒ *sequence*

`generic-function-methods` returns a sequence of all of the methods in *generic-function*. The order of the methods in the sequence is not significant.[21]

The sequence returned should never be destructively modified. Doing so may cause unpredictable behavior.

If *generic-function* is sealed, an implementation may choose not to return a sequence of methods, but instead signal an error of type `<sealed-object-error>`.

`add-method` *generic-function method* [Function]

⇒ *new-method old-method*

`add-method` adds *method* to *generic-function*. This operation modifies the generic function object.

In general, you do not need to call `add-method` directly. It is called by `define method`.

[21] A method may be removed from a generic function if it is proved that it will never be called. This will be the case if any of the objects on which the method specializes are garbage collected.

If you add a method to a generic function, and the generic function already has a method with the exact same specializers, then the old method is replaced with the new one.

A single method may be added to any number of generic functions.

`add-method` returns two values. The first is the new *method*. The second will be either the method in *generic-function* which is being replaced by *method*, or it will be `#f`, if no method is being replaced.

`generic-function-mandatory-keywords` *generic-function* [Function]

If *generic-function* accepts keyword arguments, returns a collection of the mandatory keywords for *generic-function.* Returns `#f` if *generic-function* does not accept keyword arguments.

The collection returned should never be destructively modified. Doing so may cause unpredictable behavior.

`function-specializers` *function* ⇒ *sequence* [Function]

`function-specializers` returns a sequence of the specializers for *function*. The length of the sequence will equal the number of required arguments of *function*. The first element of the sequence will be the specializer of the first argument of *function*, the second will be the specializer of the second argument, etc.

The sequence returned should never be destructively modified. Doing so may cause unpredictable behavior.

`function-arguments` *function* [Function]
⇒ *required-number rest-boolean kwd-sequence*

`function-arguments` returns three values:

1. The number of required arguments accepted by the function.
2. A boolean that indicates whether the function accepts a variable number of arguments.
3. If the value is `#f` then the function does not accept keyword arguments. Otherwise, the function does accept keyword arguments, and the value is

either a collection of the keywords which are permissible for any call to the function, or the symbol `all` if all keywords are permitted by the function.

Note that particular calls to a generic function may accept additional keywords not included in the third value returned by `function-arguments`, by virtue of their being recognized by applicable methods.

`applicable-method?` *function* `#rest` *sample-arguments* [Function]
⇒ *boolean*

`applicable-method?` returns true if *function* is a method or contains a method that would be applicable for *sample-arguments*.

`sorted-applicable-methods` [Function]
generic-function `#rest` *sample-arguments*
⇒ *sequence1 sequence2*

`sorted-applicable-methods` returns two sequences that, taken together, contain the methods in *generic-function* that are applicable for the *sample-arguments*. *sequence1* contains methods that are more specific than every method that follows them. *sequence2* (which cannot be sorted itself) begins at the first point of ambiguity; there are at least two methods that could equally well go first in *sequence2*.

The sequences returned should never be destructively modified. Doing so may cause unpredictable behavior.

`find-method` *generic-function specializer-list* [Function]
⇒ *{method or* `#f`*}*

`find-method` returns the method in *generic-function* that has the specializers in *specializer-list* as its specializers. The specializers must match exactly for a method to be returned. If *generic-function* is sealed, an implementation may choose to signal an error of type `<sealed-object-error>` (instead of returning any value at all) .

`remove-method` *generic-function method* ⇒ *method* [Function]

`remove-method` removes *method* from *generic-function* and returns *method*. `remove-method` will signal an error if *method* is not in *generic-function*. If *generic-function* is sealed, or if *method* is a sealed method of *generic-function* , then an error of type `<sealed-object-error>` is signaled.

The classes <function>, <generic-function>, and <method>

`<function>` [Abstract Class]

`<function>` is the class of all objects that can be applied to arguments. It is the superclass of generic functions and methods. `<function>` inherits from `<object>`.

`<generic-function>` [Instantiable Class]

`<generic-function>` is the class of functions which contain methods. Generic functions inherit from `<function>`.

The class `<generic-function>` supports the following init-keywords:

`required:` *number-or-sequence*

This represents the required arguments that the generic function accepts. If a sequence is supplied, the size of the sequence is the number of required arguments, and the elements of the sequence are the specializers. If a number is supplied, it is the number of required arguments, and the specializers default to `<object>`. If the argument is not supplied, or the supplied argument is neither a sequence nor a non-negative integer, an error is signaled.

`rest?:` *boolean*

A true value indicates that the generic function accepts a variable number of arguments. The default is `#f`.

`key:` *collection-of-keywords-or-#f* .

If the value is a collection, then the generic function accepts keyword arguments, and the collection specifies the set of mandatory keywords for the generic function. A value of `#f` indicates that the generic function does not accept keyword arguments. The default is `#f`.

`all-keys?:` *boolean*

A true value indicates that the generic function accepts all keyword arguments. The default is `#f`.

An error is signaled if the value of `rest?:` is true and the value of `key:` is a collection.

An error is signaled if the value of `all-keys?:` is true and the value of `key:` is `#f`.

The new generic function initially has no methods. An error will be signaled if the generic function is called before methods are added to it. Once a generic function is created, you can give it behavior by adding methods to it with `add-method` or `define method`.

Generic functions are not usually created directly. Most often they are created by `define generic` or indirectly by `define method`.

`<method>` [Class]

Methods inherit from `<function>`.

7. Classes

Introduction

Classes are used to categorize Dylan objects. Classes specify the structure of their instances. In concert with methods, classes help determine the behavior of their instances. Every object is a direct instance of exactly one class.

A class determines which **slots** its instances have. Slots are the local storage available within instances. They are used to store the state of objects.

A class also helps determine the behavior of its instances. When an object is passed as an argument to a generic function, the generic function looks at the class (and perhaps identity) of the object to determine which method should be run.

Singletons are specializers used to indicate an individual object. A singleton for an object is created by passing the object to the function `singleton`. Singletons are used to customize the behavior of individual objects. Singletons can be used for specialization, like classes, but the specialization will only apply to the singleton object. When determining whether a method is applicable for a set of arguments, an argument with a singleton specializer must be == to the object used to create the singleton.

Slots and slot access

Slots correspond to the *fields* or *instance variables* of other object-oriented programming languages. For most slots, each instance of the class has private storage for the slot, so one instance can have one value in the slot and another instance can have another value.

Dylan follows the lead of some newer object-oriented languages by always accessing slots through function calls, rather than variable references.[22] In Dylan, slots are accessed through methods. The method that returns the value of a slot is called the **getter method**, and the method that sets the value of a slot is called the **setter method**. The getter and setter methods are added to generic functions. When defining a class, you specify slots by specifying to which generic functions the getter and setter methods should be added.

---

[22]For example, Self, and to some degree, CLOS.

For example, the class definition for `<point>` might be

```
define class <point> (<object>)
  slot horizontal;
  slot vertical;
end class;
```

This definition indicates that instances of `<point>` should have two slots, `horizontal` and `vertical`. The getter method for the first slot is added to the generic function `horizontal`, and the getter method for the second slot is added to the generic function `vertical`. The setter method for the first slot is added to the generic function `horizontal-setter`, while the setter method for the second slot is added to the generic function `vertical-setter`.

To get the horizontal coordinate of a point, make the call

```
horizontal(my-point)
```

To set the horizontal coordinate to 10, use the corresponding setter function:

```
horizontal-setter(10, my-point)
```

or

```
horizontal(my-point) := 10;
```

Singleton types

Singletons provide a way to add methods to single instances. This lets programs specialize a single object, without changing other objects of the same class, and without defining a whole new class just for the object.

A singleton is a type, but it is not a class. It is little more than a pointer to a single object. The singleton's sole purpose is to indicate that object, so that a method can be created that specializes on the object. By defining a method that specializes on a singleton (rather than on a class), you have defined a method that discriminates on the singleton's object.

Singleton methods are considered more specific than methods defined on an object's original class. Singletons are the most specific specializer.

```
? define method double (thing :: singleton(#"cup"))
    #"pint"
  end method

? double (#"cup")
#"pint"
? double (10)
20
```

Dylan provides a shorthand for singletons used as method specializers. The folowing definition is equivalent to the one above.

```
? define method double (thing == #"cup")
    #"pint"
  end method

? double (#"cup")
#"pint"
```

Defining New Classes

The `double` example given in the Functions chapter showed how to define methods for classes that already exist. A large portion of Dylan programming consists of defining new classes. New classes are created with `define class`.

`define` [*adjectives*] **`class`** *class-name* (*superclasses*) [Definition]
slot-spec$_1$ `;` *slot-spec*$_2$ …
`end` [**`class`**] [*class-name*]

`define class` is used to create classes. `define class` defines the module variable *class-name* and creates a class to store in the variable. The variable is made read–only.

The *adjectives* are words that describe features of the class. The permitted adjectives are `sealed`, `open`, `primary`, `free`, `abstract`, and `concrete`. See the Controlling Dynamism chapter for more information.

The new class inherits from the *superclass*es, which is a sequence of expressions separated by commas which evaluate to classes. The elements of the list are evaluated. The list cannot contain duplicates, and the class heterarchy cannot be circular—that is, a class cannot be its own direct or indirect superclass. At least one superclass must be specified.

In addition to inheriting slots from its superclasses, the new class is given a slot for each of the *slot-spec* arguments. In the simplest format, a slot-spec is just:

`slot` *variable-name*

A getter method is defined on the generic function *name*, and a setter function is defined on the generic function `name-setter`. The full syntax for slot-specs is given in the Slot Options section below. The full syntax allows many more options when defining slots.

The following definition creates a new class and stores it in the module variable `<menu>`. Instances of the class will have two slots. The first slot is read with the generic function `title` and set with the generic function `title-setter`. The second slot is read with the generic function `action` and set with the generic function `action-setter`.

```
define class <menu> (<object>)
  slot title;
  slot action;
end class;
```

The creation and initialization of instances is controlled by the generic functions `initialize` and `make`. See the section on instance creation for more details.

Slot Uniqueness

The collection of all the getter and setter generic functions for slots specified in a class or inherited from its superclasses must not contain any duplicates. This implies that an inherited slot cannot be overridden.

If a superclass is inherited through multiple paths, its slots are only counted once. For example, if class A has direct superclasses B and C, and both B and C have D as a direct superclass, A inherits from D both through B and through C, but D's slots are only counted once so this multiple inheritance does not by itself create any duplicates among the getters and setters.

Note that if two classes each specify a slot and the two slots have the same getter and/or setter generic function, these two classes are *disjoint* —they can never have a common subclass and no object can be an instance of both classes. The same is true if one slot's getter function is the other slot's setter function (this would also cause a parameter list congruency error).

Slot Options

There are three kinds of *slot-specs* that are allowed in class definitions. These are **slot specifications**, **initialization argument specifications**, and **inherited slot specifications.**

Slot Specifications

A slot specification describes a slot.

The syntax of a slot specification is as follows:

[*adjectives*] [*allocation*] `slot` *getter-name* [`::` *type*] `#key` *setter init-keyword required-init-keyword init-value init-function*

getter-name specifies the getter name for the slot. It must be a variable name.

The *adjectives* are words separated by spaces. Currently, the allowed *adjectives* are `open` and `sealed`. These *adjectives* control the level of dynamism of the getter, and (if present) the setter for this slot. See the Controlling Dynamism chapter for more details.

allocation controls the allocation of the slot. See the Specifying Allocation section for details.

The keywords *setter*, *init-keyword*, *init-value*, and *init-function* respectively specify a setter name, an init-keyword, and (if the allocation is not `virtual`) a default value specified with `init-value:` or `init-function:`. See the Keywords Allowed in Slot Specs section for more detail.

If there is an init-keyword specified with `init-keyword:` or `required-init-keyword:`, the slot is said to be ***keyword initializable***

The following example defines a class with two keyword initializable slots accessed by the generic functions `bar-x` and `bar-y`.

```
define class <bar> (<object>)
  slot bar-x, init-keyword: x:;
  slot bar-y, init-keyword: y:;
end class <bar>;
```

In this example, the call `make (<bar>, x: 2, y: 3)` will make an instance in which these slots are initialized to `2` and `3` respectively; the call `make (<bar>)` will make in instance in which these slots are not initialized (attempting to get the values by calling the getter functions signals an error).

Initialization Argument Specifications

An initialization argument specification does not describe a slot; it describes how a particular initialization argument is to be handled. It allows the type of the initialization argument to be restricted, it allows the initialization argument to be declared to be required, and it allows the specification of a default value for an initialization argument which is not required.

There are two kinds of initialization argument specifications: required initialization argument specifications, and optional initialization argument specifications.

A required initialization argument specification has the form

`required keyword` *keyword* `#key` *type*

A required initialization argument specification asserts that the initialization argument is required in a call to `make` — the default `make` method will signal an error if no such initialization argument is supplied.

An optional initialization argument specification has the form

`keyword` *keyword* `#key` *type init-value init-function*

An optional initialization argument specification can be used to specify a default value for the initialization argument. When a call to make does not specify the keyword, the default value is used by the default `make` method in computing the *defaulted initialization arguments* , and indirectly to initialize a slot. (See the next section on slot initialization for more details.)

The type argument has the same meaning in both kinds of initialization argument specification: It restricts the type of that initialization argument. (Note that this is not the same thing as restricting the type of the slot.)

The following example defines a class similar to the previous example class `<bar>`, but with modified initialization behavior.

```
define class <baz> (<bar>)
   required keyword x:;
   keyword y:, init-value: #t;
end class <baz>;
```

The `x:` initialization argument is specified as being required: `make` will signal an error if an `x:` keyword argument is not supplied. The `y:` initialization argument , and consequently the initial value for the `bar-y` slot, now defaults to `#t` if not specified in the call to `make`.

More than one keyword initializable slot may be initialized from the same initialization argument (i.e., more than one keyword initializable slot may specify the same init-keyword). However, an error is signaled if a single `define-class` form has more than one initialization argument specification for the same keyword. An error will also be signaled if a single `define-class` form has keyword initializable slots which specify `init-value:` or `init-function:` and an initialization argument specification for the same keyword which either requires a value (`required-init-keyword:`) or provides a default value with `init-value:` or `init-function:` — i.e., the initial value specification for the slot can never be used.

Inherited Slot Specifications

An inherited slot specification specifies a getter generic function, and optionally an `init-value:` or `init-function:` slot option, but not both.

The syntax of an inherited slot specification is as follows:

`inherited slot` *getter-name* `#key` *init-value init-function*

getter-name identifies the slot. It must be a variable name. A superclass of the class being defined must specify a slot with the same getter.

If neither `init-value:` nor `init-function:` is specified, the only function of the inherited slot specification is to require that some superclass of the class being defined specify a slot with the same getter. It could also be useful as documentation, perhaps. Because the `init-value:` and `init-function:` slot options are not allowed for virtual slots, this is the only valid form of inherited slot specification for virtual slots.

If either `init-value:` or `init-function:` is specified, the superclass's slot's `init-value:` and `init-function:` slot options are ignored and the options in the inherited specification are used instead. This allows the `init-value:` and `init-function:` slot-options of an inherited slot to be replaced in a subclass so the default initial value of the slot can be changed.

```
define class <animal> (<object>)
  slot n-legs, init-value: 4;
end class;

define class <spider> (<animal>)
  inherited slot n-legs, init-value: 8;
end class;
```

Specifying Allocation

allocation in a slot specification specifies how storage for the slot is allocated. *allocation* should be one of the symbols `instance`, `class`, `each-subclass`, `virtual`, or `constant`, or it may be an implementation-dependent value. This argument is not evaluated.

`instance`	Indicates that each instance gets its own storage for the slot. This is the default.
`class`	Indicates there is only one storage location used by all the direct and indirect instances of the class. All the instances share a single value for the slot. If the value is changed in one instance, all the instances see the new value.
`each-subclass`	Indicates that the class gets one storage location for the slot, to be used by all the direct instances of the class. In addition, every subclass of the class gets a storage location for the slot, for use by its direct instances.
`constant`	Indicates a constant value for the slot. There will be no setter method. The value of the slot must be specified as an init-value.
`virtual`	Indicates that no storage will be automatically allocated for the slot. If allocation is `virtual`, then it is up to the programmer to define methods on the getter and setter generic functions to retrieve and store the value of the slot. Dylan will ensure the existence of generic functions for any specified getter and setter but will not add any methods to them.

The values of virtual slots are not automatically initialized when a new instance is created. The programmer must perform any necessary initialization. This would usually be done inside a method on `initialize`. Because the values of virtual slots are often computed from other values at run-time, many virtual slots will not require any explicit initialization.

To support the `slot-initialized?` protocol in a virtual slot, programmers must define a method for `slot-initialized?` that shares a protocol with the getter method for the slot.

Keywords allowed in slot-specs

Each keyword can be specified no more than once and must be followed by a value. Implementations may add additional keyword arguments.

`setter:` The name of a module variable to which the setter method should be added, or `#f` if no setter method should be defined. This argument is not evaluated. If it is not supplied, it will default to the symbol whose name is the concatenation of *getter-name* and "-setter", unless the allocation of the slot is `constant` in which case it defaults to `#f`.

`type:` Should be a type specifier that limits the types of values that can be stored in the slot. This type specification is enforced by the low-level slot storage mechanisms. It is not enforced for virtual slots, which do not use the low-level slot storage mechanisms.

This argument defaults to `<object>`, which allows any object to be stored in the slot. The argument is evaluated once, after the variable containing the class is defined and before the slot is first added to an instance.

`init-value:` Supplies a default initial value for the slot. The argument is evaluated once, after the variable containing the class is defined and before the slot is first added to an instance. The resultant value is used as the initial value when an instance is created. If you want to create a new value for every instance, you should supply an `init-function:` rather than an `init-value:`. There is no default value for this argument.

`init-value:` may not be specified with `init-function:`.

`init-function:` Should be a function of no arguments. It will be called to generate an initial value for the slot when a new instance is created. There is no default value for this argument. The argument is evaluated once, after the variable containing the class is defined and before the slot is first added to an instance.

`init-function:` may not be specified with `init-value:`.

`init-keyword:` Should be a keyword. It permits an initial value for the slot to be passed to `make`, as a keyword argument using this keyword. For use, see "Slot Initialization," below. This argument is not evaluated.

There is no default for the `init-keyword:` argument. If none is specified, there will be no init-keyword for the slot.

`init-keyword:` and `required-init-keyword:` cannot both be specified for a single slot.

`required-init-keyword:`
Like `init-keyword:`, except it indicates an init-keyword that must be provided when the class is instantiated. If `make` is called on the class and a required init-keyword is not provided, an error is signaled.

If `required-init-keyword:` is specified for a slot, then `init-keyword:`, `init-value:`, and `init-function:` cannot be specified.

Filtered Slots

In some situations, the value of a slot will be stored directly in instances of one class but will require some computation in a subclass. For example, the `position` slot could be stored as a value in direct instances of `<view>` while requiring some computation in direct instances of the `<view>` subclass `<displaced-view>`.

Both classes provide the same interface to `position` (the `position` generic function). If the implementor of either class decides to change the implementation, users of the classes will not need to change or recompile code, because there is no change in the interface. Consistent function call syntax helps hides implementation details.

In this case, the `<view>` class supports `position` as an instance slot, and the `<displaced-view>` class supports `position` as a filtered slot. Methods for `position` and `position-setter` are added automatically by the slot definition in `<view>`; these methods access the raw value of the slot in the instance. In contrast, the implementor of `<displaced-view>` must explicitly add methods to the two generic functions. The <displaced-view> methods can call next-method to access (get or set) the stored value of the slot.

```
define class <view> (<object>)
  instance slot position;
  ...
end class;

define class <displaced-view> (<view>)
  ...
end class;

define method position (v :: <displaced-view>)
  displace-transform (next-method (v));
end method;

define method position-setter (new-position),
                               v :: <displaced-view>)
  next-method (undisplace-transform (new-position), v);
end method;
```

In other situations, a programmer will want storage in an instance for a slot value, but will want to perform some auxiliary action whenever the slot is accessed. In this case, the programmer should define two slots: an instance slot to provide the storage and a virtual slot to provide the interface. In general, only the virtual slot will be documented. The instance slot will be an internal implementation used by the virtual slot for storage. An example of such use would be a slot that caches a value.

```
define class <shape> (<view>)
  virtual slot image;
  instance slot cached-image, init-value: #f;
  ...
end class;

define method image (shape :: <shape>)
  cached-image (shape)
    | (cached-image (shape) := compute-image (shape));
end method;

define method image-setter (new-image, shape :: <shape>)
  cached-image (shape) := new-image;
end method;
```

Note that, as in the above example, if you define a virtual slot, you are expected to define an internal instance slot in order to provide a value for the slot accessor.

Instance creation

The creation and initialization of instances is controlled by the generic functions `initialize` and `make`.

Overview

Instance creation and initialization works in the following way:

(1) The user calls `make` specifying a class and a set of keyworded arguments.

(2) Optionally, the system-supplied `make` method may be shadowed by a user-supplied method specialized with a `singleton` specializer. This enables the user method to get at all the arguments to `make`, and to provide actual initializations based on them, perhaps in combination. Typically, the method uses `next-method` with the new or modified keyworded arguments to actually create and initialize the instance.

(3) The default `make` method examines its keyworded arguments, the **supplied initialization arguments**. Next, `make` produces a set of **defaulted initialization arguments**, which will be used in creating and initializing the instance, by augmenting the supplied initialization arguments with any additional initialization arguments for which default values are defined by class or any of its superclasses.

(4) The default `make` method allocates an instance and initializes all slots it can provide values for: those which are keyword initializable are initialized from the corresponding values of the defaulted initialization arguments; slots which are not keyword initializable but have a default initial value specification have the appropriate value assigned.

(5) It then calls `initialize` on the initialized instance and the defaulted initialization arguments.

(6) Each `initialize` method typically calls `next-method()`, and then (having used `#key` to access the initargs it needs) performs the initializations it needs to. (Note that it won't have to initialize slots that were handled by step 4.)

(7) The default `make` method ignores the value of the call to `initialize`, and returns the instance.

Definitions of make and initialize

`make` *class* `#key #all-keys` => *instance* [Generic Function]

`make` returns an instance of *class*, with characteristics specified by the keyword value pairs.

Dylan does not specify whether the value returned must be a newly allocated instance, or whether `make` is permitted to return a previously created instance. If a new instance is allocated, `make` will call `initialize` on the instance before returning it.

The object returned is guaranteed to be a general instance of *class* but not necessarily a direct instance of *class*. This liberality allows `make` to be called on an abstract class; it can instantiate and return a direct instance of one of the concrete subclasses of the abstract class.

(Note that the default method on `make` returns a newly allocated direct instance of *class* .)

Programmers may customize `make` for particular classes by defining methods specialized by `singleton` specializers. These methods may obtain the default `make` behavior, if desired, by calling `next-method`.

In the absence of any customization, the default `make` method is invoked.

`make` *class* `#rest` *supplied-initialization-arguments* `#key` => *instance* [Method]

The default method for `make` does the following:

It signals an error if *class* is abstract. An instantiable abstract class must override this method with its own method for `make`.

Any deferred evaluations which have not yet been performed for *class* and its superclasses are performed first.

If *supplied-initialization-arguments* contains duplicate keywords, `make` will use the leftmost occurence. (This is consistent with `#key` argument conventions in function calls.)

`make` starts with *supplied-initialization-arguments* and constructs the set of defaulted initialization arguments , which will be used in creating and initializing the instance, by augmenting it with any additional initialization arguments for which default values are defined by *class* or any of its superclasses. `make` signals an error if any required initialization argument is absent from the defaulted initialization arguments, or if any of the defaulted initialization arguments are not valid for initialization of that class (that is, some keyword is neither usable for slot initialization nor recognized by some `initialize` method applicable to an instance of *class*). Note that init-functions used to construct the defaulted initialization arguments will only be invoked if the initialization argument is in fact missing from the supplied initialization arguments.

`make` then allocates an instance and initializes all slots for which it can compute values. If a slot is keyword initializable and the corresponding *initialization argument is present in the defaulted initialization arguments , the* slot is set to that value. Otherwise, if the slot has a default initial value specification, that is used to provide the initial value: either the init-value, or the result of calling the init-function. In either case, an error is signaled if the value is not of the type declared for the slot.

The exact order in which instance allocation and init-function invocation occurs in the above is undefined.

`make` calls `initialize` on the initialized instance and the defaulted initialization arguments, to perform any custom or complex initializations. User-defined `initialize` methods may use `#key` parameters to access particular initialization arguments, or `#rest` to access the entire set of initialization arguments. If the entire set of initialization arguments is examined and *supplied initialization arguments* contained duplicate keywords, it is undefined whether any entries other than the leftmost for that keyword will be present.

`make` ignores any values returned by the call to `initialize`, and returns the instance.

`initialize` *instance* `#key #all-keys` [Generic Function]

The `initialize` generic function provides a way for users to handle initialization of instances which cannot be expressed simply by init-values or init-functions. This is typically because a computation requires inputs from initargs or slot values, or a single computation needs to be used to initialize multiple slots.

For instance, the following example shows a `<triangle>` class which presents an interface of storing three sides, but actually stores two sides and an angle:

```
define class <triangle> (<shape>)
  slot side-a, required-init-keyword: side-a:;
  slot side-b, required-init-keyword: side-b:;
  slot angle-C;
  virtual slot side-c, required-init-keyword: side-c:
end class <triangle>;

define method initialize (x :: <triangle>,
                          #key side-a, side-b, side-c)
  next-method();
  x.angle-C := three-sides-to-angle (side-a, side-b, side-c);
end method initialize;
```

By convention, all `initialize` methods should call `next-method` very early, to make sure that any initializations from less specific classes are performed first. Dylan implementations may provide warnings about code style for methods on `initialize` which do not call `next-method`.

The `initialize` generic function *permits* all keywords, and requires none. It does this because the keyword argument checking has already been performed by the default method on `make`.

`initialize` *instance* `:: <object> #key` [Method]

The default `initialize` method does nothing. It is present so that `next-method()` may be used with abandon by modularly written `initialize` methods.

Initialization Argument Inheritance

If a slot is keyword-initializable, it is possible to change the default value for the initialization argument alone, without mention of the slot.

This is illustrated by the modified default for `favorite-beverage` in the following example.

```
define class <person> (...)
  slot favorite-beverage, init-value: #"milk",
                  init-keyword: favorite-beverage:;
  slot name required-init-keyword: name:;
end class <person>;

define class <astronaut> (<person>)
  keyword favorite-beverage: init-value: #"tang";
  keyword name: init-value: "Bud";
end class <astronaut>;
```

Another common use is to supply a default which is required in the superclass, as illustrated with the `name:` init-keyword above; this makes the `name:` keyword no longer required when calling `make` on `<astronaut>`.

Initialization argument specification inheritance is defined as follows. There are four cases:

1. A slot specification which supplies an init-keyword *K* by using `required-init-keyword:` is treated as if the initialization argument specification `required-init-keyword:` *K* had been specified in the class definition.

   A slot specification which supplies both an init-keyword and also an init-value or init-function is *not* equivalent to an initialization argument specification which includes the init-keyword and init-value or init-function. The former is used to default the value of the slot directly, but does not affect the defaulted initialization arguments; the latter is used to default the value of the slot indirectly, by affecting the defaulted initialization arguments.

2. The initialization argument is being specified for the first time — it is not inherited from any superclasses.

   a. The `type:` argument, which defaults to `<object>`, specifies the required type of the initialization argument.
   b. If the initialization argument is specified with `required-init-keyword:` then it is required, otherwise it is optional.
   c. If the initialization argument is specified with `init-keyword:`, then it can provide an initial value specification which is used by the default `make` method to provide a default value for the initialization argument in the defaulted initialization arguments. This is either a value specified with `init-value:` or a function (called each time a value is needed) specified with `init-function:`

3. An initialization argument specification is being specified for an initialization argument which is inherited from a single superclass.

   a. The type must be a subtype of the type of the inherited initialization argument.
   b. The initialization argument is required if the overriding initialization argument specification uses `required-init-keyword:`, or if the inherited initialization argument specification is required and the overriding initialization argument specification uses neither `init-value:` nor `init-function:`. When the the overriding initialization argument specification uses `required-init-keyword:`, any init-value or init-function in the inherited initialization argument specification is discarded.
   c. Otherwise, the initialization argument is optional. If the overriding specification provides an `init-value:` or `init-function:`, then that is used to compute the defaulted initialization argument when the class is instantiated. Otherwise, the inherited initial value specification is used.

4. An initialization argument specification is being inherited from multiple superclasses. If the superclasses have exactly the same definition for the initialization argument, then that definition can simply be inherited. If the definitions differ, then the class which combines these other classes must provide an initialization argument specification which is compatible with all of the inherited ones, as described above.

(3)b. means that a subclass can force an initialization argument used by a superclass to become required, but cannot force a required initialization argument to become optional without specifying a default value to be used.

Initialization of Class Allocated Slots

Initialization of slots of allocations `class` and `each-subclass` works in the following way. There are four cases to consider, depending on whether the slot is keyword initializable, and whether any way to compute a default value was specified:

1. The slot is not keyword initializable, and there is no default value for the slot (i.e., neither `init-value:` nor `init-function:` were specified). The slot value is not changed, and if it is presently uninitialized it remains so.

2. The slot is not keyword initializable, but there is a default value for it (specified with either `init-value:` or `init-function:`). The init-value (or value of the call to the init-function) is used to initialize the slot some time before or during the first call to `make` on a class that shares this slot.
3. The slot is keyword-initializable, but the corresponding initialization argument is absent from the defaulted initialization arguments of the call to `make`. The slot value is not changed, and if it is presently uninitialized it remains so.
4. The slot is keyword-initializable, and the corresponding initialization argument is present in the defaulted initialization arguments of the call to `make`. The slot is unconditionally set to the value of that initialization argument.

Testing the Initialization of a Slot

A program can test to see whether a slot has been initialized.

`slot-initialized?` *instance getter* ⇒ *boolean* [Function]

`slot-initialized?` returns true if the slot in *instance* that would be accessed by the *getter* generic function is initialized. If the slot is not initialized, then false is returned.

`slot-initialized?` will signal an error if the *getter* does not access a slot in the *instance*.

There is no mechanism for resetting a slot to the uninitialized state.

Reflective Operations on Types

The following operations return information about types and objects.

`instance?` *object type* ⇒ *boolean* [Function]

`instance?` returns true if *object* is a general instance of *type*.

`subtype?` $type_1$ $type_2$ ⇒ *boolean* [Function]

`subtype?` returns true if $type_1$ is a subtype—either direct or indirect—of $type_2$, or if $type_1$ and $type_2$ are the same type.

object-class *object* ⇒ *class* [Function]

object-class returns the class of which *object* is a direct instance.

all-superclasses *class* ⇒ *sequence* [Function]

all-superclasses returns all the superclasses of *class* in a sequence. The order of the classes in the sequence is significant. The first element in the sequence will always be *class*, and <object> will always be the last.

The sequence returned should never be destructively modified. Doing so may cause unpredictable behavior. If *class* is sealed, an implementation may choose to signal an error of type <sealed-object-error> rather than returning the sequence of all superclasses.

direct-superclasses *class* ⇒ *sequence* [Function]

direct-superclasses returns the direct superclasses of *class* in a sequence. These are the classes that were passed as arguments to make or define class when the *class* was created. The order of the classes in the sequence is the same as the order in which they were passed to define class, or make when *class* was created.

The sequence returned should never be destructively modified. Doing so may cause unpredictable behavior. If *class* is sealed, an implementation may choose to signal an error of type <sealed-object-error> rather than returning the direct superclasses.

direct-subclasses *class* ⇒ *sequence* [Function]

direct-subclasses returns the direct subclasses of *class* in a sequence. These are the classes that have *class* as a direct superclass. The order of the classes in the sequence is insignificant.

The sequence returned should never be destructively modified. Doing so may cause unpredictable behavior.[23] If *class* is sealed, an implementation may choose to signal an error of type <sealed-object-error> rather than returning the direct subclasses.

[23]Note that the pointer from a class to its subclasses is through a weak link, so subclasses may be garbage collected if there are no other references to them.

Coercing and Copying Objects

These functions are used to perform shallow copies on objects, and to coerce objects to other classes.

`as` *class object* ⇒ *instance* [Generic Function]

`as` coerces *object* to *class*. That is, it returns an instance of *class* that has the same contents as *object*.

If *object* is already an instance of *class*, it is returned unchanged.

Predefined methods allow coercion between numeric types, between integers and characters, between strings and symbols, and between collection types. No methods are defined for other classes.

When converting between collection types, the new collection (the returned object) will have the same number of elements as the source collection (the *object*). If the source and target class are both subclasses of `<sequence>`, the elements will be in the same order. The individual elements may also undergo some conversion.

`shallow-copy` *object* ⇒ *new-object* [Generic Function]

`shallow-copy` returns a new object that has the same contents as *object*. The contents are not copied but are the same objects contained in *object*.

Dylan includes methods for copying collections. The method for `<collection>` creates a new object by calling `make` on the `class-for-copy` of *object*. For other classes, the programmer must provide a method.

`class-for-copy` *object* ⇒ *class* [Generic Function]

`class-for-copy` returns an appropriate collection class for creating mutable copies of the argument. For collections that are already mutable, the collection's actual class is generally the most appropriate, so the `<object>` method of `class-for-copy` can be used. The `class-for-copy` value of a sequence should be a subclass of `<sequence>`, and the`class-for-copy` value of an explicit-key-collection should be a subclass of `<explicit-key-collection>`. In all cases, the`class-for-copy` value must be a mutable collection.

The classes <type>, <class>, and <singleton>

`<type>` [Abstract Class]

All types (including `<type>` and `<class>`) are general instances of `<type>`. `<type>` is a subclass of `<object>`.

`<class>` [Abstract Instantiable Class]

All classes (including `<class>`) are general instances of `<class>`. `<class>` is a subclass of `<type>`.

In most programs the majority of classes are created with `define class`. However, there is nothing to prevent programmers from creating classes by calling `make`, for example, if they want to create a class without storing it in a module variable, or if they want to create new classes at runtime.

If `make` is used to create a new class and creating the new class would violate any restrictions specified by sealing directives, then an error of type `<sealed-object-error>` is signaled.

The class `<class>` supports the following init-keywords:

`superclasses:` Specifies the direct superclasses of the class. `superclasses:` should be a class or a sequence of classes. The default value is `<object>`. The meaning of the order of the superclasses is the same as in `define class`.

`slots:` A sequence of slot specs, where each slot-spec is a sequence of keyword/value pairs.

The following keywords and corresponding values are accepted by all implementations. Implementations may also define additional keywords and values for use within slot specs.

`getter:` A generic function of one argument. Unless the allocation of the slot is virtual, the getter method for the slot will be added to this generic function. This option is required.

`setter:` A generic function of two arguments. Unless the allocation of the slot is virtual, the setter method for the slot will be added to this generic function. There is no default.

`type:` A type. Values stored in the slot are restricted to be of this type. The default value for this option is <object>.

`deferred-type:` A function of no arguments, which returns a type, and is called once to compute the type of the slot, within the call to `make` which constructs the first instance of that class.

`init-value:` Supplies a default initial value for the slot. This option cannot be specified along with `init-function:`. There is no default.

`init-function:` A function of no arguments. This function will be called to generate an initial value for the slot when new instances are created. This option cannot be specified along with `init-value:`. There is no default

`init-keyword:` A keyword. This option permits an initial value for the slot to be passed to make, as a keyword argument using this keyword. There is no default. This option cannot be specified along with `required-init-keyword:`.

`required-init-keyword:`
A keyword. This option is like `init-keyword:`, except it indicates an init-keyword that must be provided when the class is instantiated. If `make` is called on the

class and a required init-keyword is not provided, an error is signaled. There is no default. This option cannot be specified if `init-keyword:`, `init-value:`, or `init-function:` is specified.

`allocation:`	One of the keywords `instance:`, `class:`, `each-subclass:`, `constant:`, or `virtual:`, or an implementation defined keyword. The meaning of this option is the same as adding the corresponding adjective to a `define class` form.

<singleton> [Instantiable Class]

The class `<singleton>` supports the following init-keyword:

`object:`	The object that the singleton indicates. There is no default for this argument. If it is not supplied, an error will be signaled.

If a singleton for the specified object already exists, implementations are free to fold the two instances into one.

singleton *object* ⇒ *singleton* [Function]

`singleton` returns a singleton for *object*.

`singleton(`*object*`)` is equivalent to `make(<singleton>, object:` *object*`)`.

8. Controlling Dynamism

This section describes how you can selectively control the dynamism of Dylan's functions and classes. The techniques for selectively controlling dynamism in Dylan are:

- Declaring a class to be `sealed` or `open`
- Declaring a class to be `abstract` or `concrete`
- Declaring a class to be `primary` or `free`
- Declaring a generic function to be `sealed` or `open`
- Using the `seal generic` form, or using the abbreviations `define sealed method` and `sealed slot`.

Syntactically, these are all expressed as *adjectives* on the generic function, class, or method definition, or slot specification, with the exception of the `seal generic` form.

Declaring characteristics of classes

A class definition (see the previous chapter on Classes) may include the adjectives `sealed`, `open`, `primary`, `free`, `abstract`, or `concrete`. These adjectives declare characteristics of the class.

- A class can be declared to be either `sealed` or `open`. If a class is sealed then no additional subclasses other than those *explicitly defined* in the same *library* may be created. Thus, it is an error to define a subclass of a sealed class in some *library* other than the one which defined the sealed class, or to use `make` of `<class>` with a sealed class directly or indirectly in the superclasses. An open class does not prohibit such operations. It is an error to define an open subclass of a sealed class.

  When *explicitly defining* a class, the default is for the class to be sealed. This may be overriden by explicitly specifying that it is open. A class created using `make` of `<class>` is open. There is no specified way to create a sealed class using `make`.

- An explicitly defined class may be declared to be either `primary` or `free`. The default is `free`. It is illegal for a class to have two primary superclasses unless one is a subclass of the other.

- An explicitly defined class may be defined to be either `abstract` or `concrete`. The default is `concrete`. The superclasses of an abstract class must be abstract. An abstract class does not usually have slots; although defining slots in an abstract class is not forbidden, it should be done with caution, especially for instance allocated slots.

Declaring sealing of generic functions

A generic function definition (see the previous chapter on Functions) may include the adjective `sealed` or the adjective `open`. These adjectives declare whether the generic function is sealed..

If a generic function is sealed then no additional methods other than those *explicitly defined* in the same *library* may be added to the generic function. Thus, it is an error to define a method for a sealed generic function in some *library* other than the one which defined the sealed generic function, or to apply `add-method` or `remove-method` to a sealed generic function. An open generic function does not prohibit such operations.

When *explicitly defining* a generic function, the default is for the generic function to be sealed. This may be overridden by explicitly specifying that it is open. A generic function that has no *explicit definition* but has implicit definitions provided by *explicit definitions* of *generic function methods* for it is sealed. A generic function created using `make` of `<generic-function>` is open. There is no specified way to create a sealed generic function using `make.`

The seal generic form

There is also a `seal generic` top-level form, with the following syntax:

```
seal generic function ( type, ... );
```

The *function* must be a module variable that refers to a generic function.

The *types* must be expressions that evaluate to instances of `<type>`. The number of *types* must be the same as the number of required arguments accepted by the *function.*

A `seal generic` form in a library L for a generic function G with types $T_1 ... T_n$ imposes the following constraints on programs:

1. A method M which is congruent to G and which is not an explicitly known method in L may be added to G if and only if at least one of the specializers for M is disjoint from the corresponding type in the sealing form.
2. A method M may be removed from G if and only if at least one of the specializers for M is disjoint from the corresponding type in the sealing form.
3. A class C with direct superclasses $D_1 ... D_m$ and which is not an explicitly known class in L may be created if and only if
    a. none of the classes $D_1 ... D_m$ are subtypes of any of the types $T_1 ... T_n$ in the sealing form, or
    b. for every method M in G with specializers $S_1 ... S_n$, if there exists an i such that S_i is disjoint from T_i, then M must satisfy condition (1) below, otherwise M must satisfy condition (2) below.
        i. There exists an i such that S_i is disjoint from T_i and such that there do not exist j and k such that D_j is a subtype of S_i and D_k is a subtype of T_i.
        ii. There do not exist i, j, and k such that D_j and D_k are subtypes of T_i, D_j is disjoint from D_k, D_j is a subtype of S_i, and D_k is not a subtype of S_i.

Abbreviations for seal generic

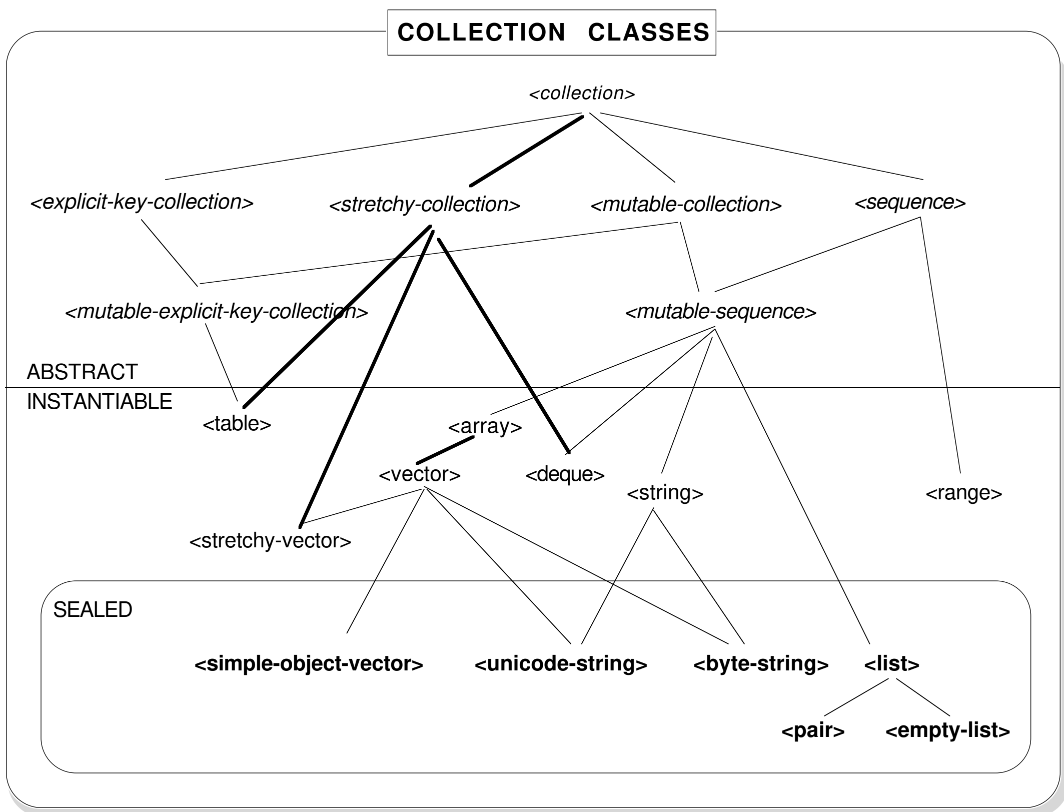
`define sealed method` defines a normal method and then seals the generic function for the types that are the specializers of the method. `define sealed method` is a convenient abbreviation for the `seal generic` form.

The `sealed slot` option to `define class` defines a normal slot and then seals the getter generic function for the class, and seals the setter generic function, if there is one, on `<object>` and the class. The `sealed slot` option to `define class` is a convenient abbreviation for the`seal generic` form.

9. Collections

This section describes the design of the classes of aggregate data structures (collections). In addition to the concrete classes provided by the Dylan language, the heterarchy for collection classes contains several abstract classes, each of which embodies some particular behavior, usually expressed as a protocol of generic functions. Every subclass of an abstract class implements every generic function of the corresponding protocol.

The following diagram shows the collection class heterarchy. Abstract classes are shown in italic and sealed classes are shown in bold.



Functions for Collections

The Dylan run-time system provides default implementations for all of the functions described in this section.

size *collection* ⇒ *{integer or* #f*}* [Generic Function]

size returns the number of keys contained in *collection*. A default method is provided by the Dylan run-time system which simply counts while iterating through the collection. size may return #f for collections of unbounded size.

class-for-copy *mutable-collection* ⇒ *class* [G.F. Method]

class-for-copy returns an appropriate collection class for creating mutable copies of the argument. For collections that are already mutable, the collection's actual class is generally the most appropriate, so the <object> method of class-for-copy can be used. The class-for-copy value of a sequence should be an instantiable subclass of <mutable-sequence>, and the class-for-copy value of an explicit-key-collection should be an instantiable subclass of <mutable-explicit-key-collection>.

empty? *collection* ⇒ *boolean* [Generic Function]

Returns #t if *collection* contains no elements, and #f otherwise.

do *procedure collection* #rest *more-collections* [Function]
⇒ *false*

do applies *procedure* to corresponding elements of all the collections. Returns #f. If all the collections are sequences, do guarantees that they will be processed in their natural order.

```
? do (method (a b) print (a + b) end,
      #(100, 100, 200, 200),
      #(1, 2, 3, 4))
101
102
203
204
#f
```

map *procedure collection* `#rest` *more-collections* [Function]

⇒ *new-collection*

`map` creates a new collection whose elements are obtained by calling *procedure* on corresponding elements of all the collection arguments. If all the collections are sequences, processing is performed in the natural order.

`map` returns a collection whose value is an instance of the `class-for-copy` value of *collection*. The new collection is created by calling make on that class, with a size: initialization argument whose value is the number of corresponding elements in the argument collections.

```
? map (\+,
       #(100, 100, 200, 200),
       #(1, 2, 3, 4))
#(101, 102, 203, 204)
```

map-as *class procedure collection* `#rest` *more-collections* [Function]

⇒ *new-collection*

`map-as` creates a new collection of class *class* whose elements are obtained by applying *procedure* to corresponding elements of the collection arguments. *class* must be an instantiable subclass of `<mutable-collection>` and acceptable as the required argument to `make`. `size:` with a non-negative integer value must be an acceptable initarg for `make` of the class. The new collection is created by calling `make` on the indicated class, with a `size:` initialization argument whose value is the number of corresponding elements in the argument collections. If all the collections are sequences (including the created collection), processing is done in the natural order.

```
? map-as (<vector>, \+,
          #(100, 100, 200, 200),
          #(1, 2, 3, 4))
#(101, 102, 203, 204)
```

map-into *mutable-collection procedure*
collection `#rest` *more-collections* [Function]

⇒ *mutable-collection*

`map-into` returns the *mutable-collection* argument after modifying it by replacing its elements with the results of applying *procedure* to corresponding elements of *collection* and *more-collections*. If *mutable-collection* and all the other collections are sequences, processing is done in the natural order.

When *mutable-collection* is an instance of `<stretchy-collection>`, the usual alignment requirement (described in the section on collection alignment) is relaxed. In this case, the key sequence of *mutable-collection* is not considered during alignment. Rather, only the key sequences for the source collections are aligned, with procedure called on the corresponding elements. The result of each call to *procedure* is then stored into *mutable-collection* with the corresponding key (possibly stretching *mutable-collection* in the process), using `element-setter`. Other keys in *mutable-collection* remain undisturbed.

The target collection for `map-into` must have the same key-test function as the rest of the collections, even if it is a `<stretchy-collection>` and therefore does not get aligned. Otherwise, an error is signalled.

mutable-collection may be the same as *collection* or any of the *more-collections*.

```
? define variable x = list (100, 100, 200, 200)
x
? map-into (x, \+, #(1, 2, 3, 4))
#(101, 102, 203, 204)
? x
#(101, 102, 203, 204)
```

any? *procedure collection* `#rest` *more-collections* [Function]

⇒ *value*

If *procedure* returns a value other than `#f` when applied to some group of corresponding elements of *collection* and *more-collections*, then `any?` returns the (first) value returned by *procedure*. Otherwise, if *procedure* returns `#f` when applied to every such group, then `any?` returns `#f`. If all the collection arguments are sequences, `any?` operates in natural order. In all cases, `any?` stops on the first true value returned by *procedure*.

```
? any? (\>, #(1, 2, 3 ,4), #(5, 4, 3, 2))
#t
? any? (even?, #(1, 3, 5, 7))
#f
```

every? *procedure collection* `#rest` *more-collections* [Function]

⇒ *boolean*

`every?` is similar to `any?`, but `every?` returns `#f` if *procedure* ever returns `#f` and otherwise returns `#t`.

```
? every? (\>, #(1, 2, 3, 4), #(5, 4, 3, 2))
#f
? every? (odd?, #(1, 3, 5, 7))
#t
```

reduce *procedure initial-value collection* ⇒ *value* [Generic Function]

procedure is a binary function used to combine all the elements of *collection* into a single value. If *collection* is empty, `reduce` returns *initial-value*; otherwise, *procedure* is applied to *initial-value* and the first element of *collection* to produce a new value. If more elements remain in the *collection*, then *procedure* is called again, this time with the value from the previous application and the next element from *collection*. This process continues until all elements of *collection* have been processed. Processing is always done in the natural order for *collection*.

```
? define variable high-score = 10
high-score
? reduce (max, high-score, #(3, 1, 4, 1, 5, 9))
10
? reduce (max, high-score, #(3, 12, 9, 8, 8, 6))
12
```

reduce1 *procedure collection* ⇒ *value* [Generic Function]

`reduce1` is just like `reduce`, except that the first element of*collection* is taken as an initial value, and all the remaining elements of *collection* are processed as if by `reduce`. (In other words, the first value isn't used twice.) For unstable collections, "first" element effectively means "an element chosen at random." It is an error for *collection* to be an empty collection. Processing is done in the natural order for *collection*.

```
? reduce1 (\+, #(1, 2, 3, 4, 5))
15
```

member? *value collection* `#key` *test* ⇒ *boolean* [Generic Function]

`member?` returns `#t` if and only if*collection* contains *value* (as determined by *test*, which defaults to ==). The *test* function may be non-commutative: it is always called with *value* as its first argument and an element from *collection* as its second argument.

```
? define constant flavors = #(#"vanilla", #"pistachio", #"ginger")
flavors
? member? (#"vanilla", flavors)
#t
? member? (#"banana", flavors)
#f
```

find-key *collection procedure* #key *skip failure* ⇒ *key* [Generic Function]

This function returns a key value such that (*procedure* (element *collection* *key*)) is true. If no element in the collection satisfies *procedure*, it returns *failure* (default #f).

The *skip* argument (default 0) indicates that the first *skip* matching elements should be ignored. If *skip* or fewer elements of *collection* satisfy *predicate*, then *failure* is returned.

```
? flavors
#(#"vanilla", #"pistachio", #"ginger")
? find-key (flavors, has-nuts?)
1
? flavors[1]
#"pistachio"
```

replace-elements! *mutable-collection predicate* *new-value-fn* #key *count* [Generic Function]
⇒ *mutable-collection*

This function replaces those elements of *mutable-collection* for which *predicate* returns true. The elements are replaced with the value of calling *new-value-fn* on the element. If *count* is specified, no more than *count* elements are replaced.

```
? define variable numbers = list (10, 13, 16, 19)
numbers
? replace-elements! (numbers, odd?, double)
#(10, 26, 16, 38)
```

fill! *mutable-collection value* #key *start end* [Generic Function]

fill! alters *mutable-collection* so that (element *mutable-collection key*) returns *value* for every *key*.

If *mutable-collection* is a mutable sequence, then *start* and *end* keywords may be specified to indicate that only a part of the sequence should be filled. *start* is considered an inclusive bound and defaults to 0; *end* is an exclusive bound and defaults to the length of the sequence.

```
? define variable numbers = list (10, 13, 16, 19)
numbers
? fill! (numbers, 3, start: 2)
#(10, 13, 3, 3)
```

key-test *collection* => *test-function* [Generic Function]

Returns the function used by the collection to compare keys. All collection classes must provide or inherit a method that returns a result consistent with their iteration protocol and `element` methods. A given method for `key-test` must return the same value (compared with ==) each time it is called.

Functions for Sequences

Dylan provides default implementations for all of the functions described in this section.

The default methods for `add`, `add-new`, `remove`, `choose`, `choose-by`, `intersection`, `union`, `remove-duplicates`, `copy-sequence`, `concatenate`, `reverse`, and `sort` all return new sequences that are instances of the `class-for-copy` of the primary sequence argument. However, more specialized methods are permitted to choose a more appropriate result class; for example, `copy-sequence` of a range returns another range, even though the `class-for-copy` value of a range is the `<list>` class.

size-setter *n stretchy-sequence* ⇒ *n* [Generic Function]

Sets the size of *stretchy-sequence* to be *n*. *stretchy-sequence* is destructively modified.

If *n* is less than or equal to the original size of *stretchy-sequence*, then the first *n* elements of *stretchy-sequence* are retained at the same positions. If *n* is greater than the original size of *stretchy-sequence*, then the previous elements of the *stretchy-sequence* are retained at the same positions, and enough new elements are added to reach the new size. The value of each new element is the same as would have been used if *stretchy-sequence* had been created with `make`, specifying `size:` *n* but not `fill:`.

It is not specified how `size-setter` adds new elements to a *stretchy-sequence*. In particular, it is not required to call `add!` or any other predefined Dylan function.

add *sequence new-element* ⇒ *new-sequence* [Generic Function]

add returns a new sequence that contains *new-element* and all the elements of *sequence*. The resultant sequence's size is one greater than the size of *sequence*. The resultant sequence shares no structure with *sequence*, and *sequence* will be unmodified. The generic function add doesn't specify where the new element will be added, although individual methods may do so.

```
? define variable numbers = #(3, 4, 5)
numbers
? add (numbers, 1)
#(1, 3, 4, 5)
? numbers
#(3, 4, 5)
```

add! *sequence1 new-element* ⇒ *sequence2* [Generic Function]

add! returns a sequence made from *sequence1* that includes *new-element*. The result may or may not be == to *sequence1*. The size of *sequence2* is one greater than the size of *sequence1*. *sequence1* may be modified as a side effect of this operation. *sequence2* may share structure with *sequence1*.

```
? define variable numbers = list (3, 4, 5)
numbers
? add! (numbers, 1)
#(1, 3, 4, 5)
```

add-new *sequence new-element* #key *test* ⇒ *new-sequence* [Generic Function]

If *new-element* is not already a member of *sequence* (as determined by the *test* function, which defaults to ==), then add-new operates just as add would. If *new-element* is already a member of *sequence*, then *sequence* is returned. *sequence* is never modified by this operation. The *test* function may be non-commutative: it is always called with an element from *sequence* as its first argument and *new-element* as its second argument.

```
? add-new (#(3, 4, 5), 1)
#(1, 3, 4, 5)
? add-new (#(3, 4, 5), 4)
#(3, 4, 5)
```

add-new! *sequence new-element* #key *test* ⇒ *new-sequence* [Generic Function]

If *new-element* is not already a member of *sequence* (as determined by the *test* function, which defaults to ==), then add-new! operates just as add! would. If *new-element* is already a member of *sequence*, then *sequence* is returned.

sequence may be modified by this operation. The *test* function may be non-commutative: it is always called with an element from *sequence* as its first argument and *new-element* as its second argument.

```
? add-new! (list (3, 4, 5), 1)
#(1, 3, 4, 5)
? add-new! (list (3, 4, 5), 4)
#(3, 4, 5)
```

remove *sequence value* #key *test count* ⇒ *new-sequence* [Generic Function]

remove returns a new sequence consisting of the elements of *sequence* not equal to *value*. *test*, which defaults to ==, is a function which determines whether an element is equal to *value*. If *count* is specified, then no more than *count* copies of *value* are removed (so additional elements equal to *value* might remain). The resultant sequence must share no structure with *sequence* and *sequence* will not be modified by this operation. The *test* function may be non-commutative: it is always called with an element from *sequence* as its first argument and *value* as its second argument.

```
? remove (#(3, 1, 4, 1, 5, 9), 1)
#(3, 4, 5, 9)
```

remove! *sequence value* #key *test count* ⇒ *new-sequence*[Generic Function]

remove! returns a sequence consisting of the elements of *sequence* which are not equal to *value*. *test*, which defaults to ==, is a function which determines whether an element is equal to *value*. If *count* is specified, then no more than *count* copies of *value* are removed (so additional elements equal to *value* might remain). The resultant sequence may or may not be == to *sequence* and may share structure with *sequence*. *sequence* may be modified by this operation. The *test* function may be non-commutative: it is always called with an element from *sequence* as its first argument and *value* as its second argument.

```
? remove! (list (3, 1, 4, 1, 5, 9), 1)
#(3, 4, 5, 9)
```

choose *predicate sequence* ⇒ *new-sequence* [Generic Function]

choose returns a new sequence containing only those elements of *sequence* that satisfy *predicate*.

```
? choose (even?, #(3, 1, 4, 1, 5, 9))
#(4)
```

`choose-by` *predicate test-sequence value-sequence* [Generic Function]
⇒ *new-sequence*

`choose-by` returns a new sequence formed of elements from *value-sequence* whose corresponding elements in *test-sequence* satisfy *predicate*.

```
? choose-by (even?, range (from: 1),
             #("a", "b", "c", "d", "e", "f", "g", "h", "i"))
#("b", "d", "f", "h")
```

`intersection` *sequence1 sequence2* `#key` *test* [Generic Function]
⇒ *new-sequence*

`intersection` returns a new sequence containing only those elements of *sequence1* that also appear in *sequence2*. *test*, which defaults to ==, is used to determine whether an element appears in *sequence2*. It is always called with an element of *sequence1* as its first argument and an element from *sequence2* as its second argument. The order of elements in the result sequence is not specified. The result sequence may or may not share structure with the argument sequences.

```
? intersection (#("john", "paul", "george", "ringo"),
                #("richard", "george", "edward", "charles"),
                test: \=)
#("george")
```

`union` *sequence1 sequence2* `#key` *test* ⇒ *new-sequence* [Generic Function]

`union` returns a sequence containing every element of *sequence1* and *sequence2*. If the same element appears in both argument sequences, this will not cause it to appear twice in the result sequence. However, if the same element appears more than once in a single argument sequence, it may appear more than once in the result sequence. *test*, which defaults to ==, is used for all comparisons. It is always called with an element from *sequence1* as its first argument and an element from *sequence2* as its second argument. The order of elements in the result is not specified. The result sequence may or may not share structure with the argument sequences.

```
? union (#("butter", "flour", "sugar", "salt", "eggs"),
         #("eggs", "butter", "mushrooms", "onions", "salt"),
         test: \=)
#("salt", "butter", "flour", "sugar", "eggs", "mushrooms",
"onions")
```

remove-duplicates *sequence* #key *test* ⇒ *new-sequence* [Generic Function]

This function returns a new sequence that contains all the unique elements from *sequence* but no duplicate elements. The result shares no structure with *sequence*, and *sequence* will not be modified by the operation. *test* , which defaults to ==, is the function used to determine whether one element is a duplicate of another. The *test* argument may be non-commutative; it will always be called with its arguments in the same order as they appear in *sequence*.

```
? remove-duplicates (#("spam", "eggs", "spam",
                       "sausage", "spam", "spam"),
                     test: \=)
#("spam", "eggs", "sausage")
```

remove-duplicates! *sequence* #key *test* ⇒ *new-sequence* [Generic Function]

This function returns a sequence that contains all the unique elements from *sequence* but no duplicate elements. The result may or may not share structure with *sequence*, the result may or may not be == to *sequence*, and *sequence* may or may not be modified by the operation. *test* , which defaults to ==, is the function used to determine whether one element is a duplicate of another. The *test* argument may be non-commutative; it will always be called with its arguments in the same order as they appear in *sequence*.

```
? remove-duplicates! (#("spam", "eggs", "spam",
                       "sausage", "spam", "spam"),
                     test: \=)
#("spam", "eggs", "sausage")
```

copy-sequence *source* #key *start end* ⇒ *new-sequence* [Generic Function]

`copy-sequence` creates a new sequence containing the elements of*source* between *start* (default 0) and *end* (which defaults to the size of *source*).

```
? define constant hamlet = #("to", "be", "or", "not", "to", "be")
hamlet
? hamlet == copy-sequence (hamlet)
#f
? copy-sequence (hamlet, start: 2, end: 4)
#("or", "not")
```

`concatenate-as` *class sequence1* `#rest` *more-sequences* [Function]

⇒ *new-sequence*

`concatenate-as` returns a new sequence, of class *class*, containing all the elements of all the sequences, in order. *class* must be a subclass of `<mutable-sequence>` and acceptable as the required argument to `make`. `size:` with a non-negative integer value must be an acceptable initarg for `make` of the class. The new sequence is created by calling `make` on the indicated class, with a `size:` initialization argument whose value is the sum of the sizes of the arguments.

```
? concatenate-as (<string>, #('n', 'o', 'n'), #('f', 'a', 't'))
"nonfat"
```

`concatenate` *sequence1* `#rest` *sequences* ⇒ *new-sequence* [Function]

`concatenate` returns a new sequence containing all the elements of all the sequences, in order. *new-sequence* may non-destructively share structure with any of the input sequences, but it is not guaranteed to do so. `concatenate` returns a sequence whose value is an instance of the `class-for-copy` value for the first sequence. The `class-for-copy` value for the first sequence must be a subclass of `<mutable-collection>`, and it must support the init keyword `size:` The *new-sequence* is created by calling `make` on the indicated class, with a `size:` initialization argument whose value is the sum of the sizes of the argument sequences.

```
? concatenate ("low-", "calorie")
"low-calorie"
```

replace-subsequence! *sequence insert-sequence* #key *start end* [Generic Function]

⇒ *result-sequence*

This function returns a sequence with the same elements as *sequence*, except that elements of the indicated subsequence are replaced by all the elements of *insert-sequence*. The subsequence to be overridden begins at index *start* and ends at index *end*. If *start* is not supplied, it defaults to 0. If *end* is not supplied, it defaults to `size`(*sequence*). *result-sequence* may or may not share structure with *sequence* , it may or may not be == to *sequence* , and *sequence* may or may not be modified by the operation. *result-sequence* will not share structure with *insert-sequence*. *result-sequence* is not destroyed.

```
? define variable x = list ("a", "b", "c", "d", "e")
// unspecified
? abcde := replace-subsequence! (x, #("x", "y", "z"), end: 1))
#("x", "y", "z", "b", "c", "d", "e")
? abcde := replace-subsequence! (x, #("x", "y", "z"), start: 4))
#("x", "y", "z", "b", "x", "y", "z")
? abcde := replace-subsequence! (x, #("a", "b", "c"),
                                          start: 2, end: 4))
#("x", "y", "a", "b", "c", "x", "y", "z")
```

reverse *sequence* ⇒ *new-sequence* [Generic Function]

`reverse` creates a new sequence containing the same elements as *sequence*, but in reverse order. The result is generally of the same class as the *sequence* argument.

```
? define constant x = #("bim", "bam", "boom")
x
? reverse(x)
#("boom", "bam", "bim")
? x
#("bim", "bam", "boom")
```

reverse! *sequence1* ⇒ *sequence2* [Generic Function]

`reverse!` returns a sequence containing the same elements as *sequence*, but in reverse order, possibly reusing *sequence's* storage to hold the result. Even if the storage is recycled, the result may or may not be == to *sequence*.

sort *sequence* `#key` *test stable* ⇒ *new-sequence* [Generic Function]

`sort` returns a new sequence containing the elements of *sequence* sorted into ascending order. *test*, which defaults to <, is used to determine what an "ascending order" is. If *stable* is supplied and not `#f`, a possibly slower algorithm will be used that will leave in their original order any two elements, *x* and *y*, such that *test(x, y)* and *test(y, x)* are both false.

```
? define variable numbers = #(3, 1, 4, 1, 5, 9)
numbers
? sort (numbers)
#(1, 1, 3, 4, 5, 9)
? numbers
#(3, 1, 4, 1, 5, 9)
```

sort! *sequence1* #key *test stable* ⇒ *sequence2* [Generic Function]

This function is like `sort`, but may reuse the storage from *sequence* to form the result. The result may or may not be == to *sequence*.

first *sequence* #key *default* ⇒ *value* [Function]

second *sequence* #key *default* ⇒ *value* [Function]

third *sequence* #key *default* ⇒ *value* [Function]

Each of these functions returns the indicated element of the *sequence* by calling `element` with the supplied arguments and the corresponding index.

Note that because `element` is zero-based, `first(seq)` is equivalent to `element (seq, 0)` and `seq[0]`.

first-setter *new-value sequence* ⇒ *new-value* [Function]

second-setter *new-value sequence* ⇒ *new-value* [Function]

third-setter *new-value sequence* ⇒ *new-value* [Function]

Each of these functions sets the indicated element of the *sequence* and returns the *new-value*, by calling `element-setter` with the supplied arguments and the corresponding index.

Note that because `element` is zero-based, `first-setter(val, seq)` is equivalent to `element-setter (val, seq, 0)` and `seq[0] := val`.

last *sequence* #key *default* ⇒ *value* [Generic Function]

`last` returns the last element of *sequence*. If the sequence is empty, then the behavior of `last` depends on whether it was called with a *default* argument. If the *default* argument was supplied, its value is returned; otherwise, an error is signaled.

```
? last (#("emperor", "of", "china"))
"china"
```

`last-setter` *new-value mutable-sequence* ⇒ *new-value* [Generic Function]

Replaces the last element of *mutable-sequence* with *new-value*. *new-value* must obey any type restrictions for elements of *mutable-sequence* . An error is signaled if *mutable-sequence* is empty or unbounded.

```
? define variable my-list = list (1, 2, 3)
? my-list
#(1, 2, 3)
? last (my-list) := 4
4
? my-list
#(1, 2, 4)

? define variable my-empty-vector = vector()
? my-empty-vector
#[]
? last (my-empty-vector) := 4
// error
```

`subsequence-position` *big pattern* `#key` *test count* ⇒ *index* [Generic Function]

big and *pattern* must be sequences. Searches *big* for a subsequence that is element-for-element equal to *pattern*, as determined by the *test* argument (which defaults to ==). That is, *test* is applied to the subsequence of *big* and corresponding elements of the *pattern* to determine whether a match has occurred. If *count* is specified, then the *count*th such subsequence is selected. If a subsequence is found, returns the index at which the subsequence starts; otherwise, returns `#f`.

```
? subsequence-position ("Ralph Waldo Emerson", "Waldo")
6
```

sequence$_1$ **=** *sequence*$_2$ ⇒ *boolean* [G.F. Method]

For sequences, = returns `#f` unless *sequence1* and *sequence2* have the same size and elements with = keys are =.

`key-test` *sequence* => *test-function* [G.F. Method]

The method of `key-test` for sequences returns the function ==.

The Instantiable Collection Classes

Dylan provides several standard instantiable collection classes, in addition to classes the user might wish to add. Each provides unique capabilities, strengths, and weaknesses.

`<array>` [Abstract Instantiable Class]

Arrays are structures whose elements are arranged according to a Cartesian coordinate system. An array element is referred to by a (possibly empty) series of indices. The length of the series must equal the rank of the array. Each index must be a non-negative integer less than the corresponding dimension. An array element may alternatively be referred to by an integer, which is interpreted as a row-major index.

Arrays typically use space efficient representations, and the average time required to access a randomly chosen element is typically sublinear in the number of elements.

`make` of `<array>` supports the keywords `dimensions:` and `fill:`. `dimensions:` is required and must be a sequence of non-negative integers. `fill:` (default `#f`) specifies the initial value to be placed in all elements of the array.

Each concrete subclass of `<array>` must either provide or inherit implementations of the functions `element` and `element-setter`.

`<byte-string>` [Sealed Instantiable Class]

This sealed subclass of `<vector>` provides efficient storage for elements that are eight-bit characters. It also inherits lexicographic ordering from the `<string>` class. `make` of `<byte-string>` supports the keywords `size:` and `fill:`. `size:` (default 0) tells how large a `<byte-string>` to create; `fill:` specifies the initial value of every element.

`<deque>` [Instantiable Class]

This subclass of `<mutable-sequence>` implements double-ended queues. `make` of `<deque>` supports the keywords `size:` and `fill:`. `size:` (default 0) tells how large a `<deque>` to create; `fill:` (default `#f`) specifies the initial value of every element.

<list> [Sealed Instantiable Class]

This sealed subclass of `<sequence>` provides efficient support for linked lists. `make` of `<list>` supports the keywords `size:` and `fill:`. `size:` (default 0) tells how large a `<list>` to create; `fill:` (default `#f`) specifies the initial value of every element.

<empty-list> [Sealed Instantiable Class]

The class `<empty-list>` has only one instance, the empty list. The empty list is a direct instance of `<empty-list>` and an indirect instance of `<list>`. Note that `<empty-list>` is not == to `singleton (#())`.

```
? object-class (#())
<empty-list>
? define method f (x :: <empty-list>) 1 end;
? define method f (x == #()) 2 end;
? define method f (x :: <list>) 3 end;
? f (#())
2
? f (#("chocolate", "vanilla"))
3
```

<range> [Instantiable Class]

A subclass of `<sequence>` that represents arithmetic sequences. Some ranges can be infinitely large. The permitted initargs are `from:, to:, above:, below:, by:,` and `size:`.

`from:` (default 0) is the first value in the range. `by:` (default 1) is the step between consecutive elements of the range. `size:` specifies the size of the range. `range:` has three additional initargs, `to:`, `above:` and `below:`, which constrain the range. `to:` is an inclusive bound, `above:` is an exclusive lower bound and `below` is an exclusive upper bound. `above:` and `below:` constrain the range independent of the sign of `by:`. It is permissible to specify a range that runs from a higher to a lower value by specifying a negative value for `by:`.

<simple-object-vector> [Sealed Instantiable Class]

This sealed subclass of `<vector>` supports elements of any class. `make` of `<simple-object-vector>` supports the keywords `size:` and `fill:`. `size:` (default 0) specifies the size of the `<simple-object-vector>`; `fill:` (default `#f`) specifies the initial value of every element.

<stretchy-vector> [Abstract Instantiable Class]

This subclass of `<vector>` supports elements of any class. `make` of `<stretchy-vector>` supports the keywords `size:`, `fill:`. `size:` (default 0) tells how large a `<stretchy-vector>` to create; `fill:` (default `#f`) specifies the initial value of every element.

<string> [Abstract Instantiable Class]

A subclass of `<sequence>` whose elements are all characters. `<string>` supports lexicographic comparison. `<string>` has no direct instances; calling `make` on `<string>` will return an instance of a concrete subclass of `<string>`. `make` of `<string>` supports the init-keywords `size:` and `fill:`. `size:` (default `0`) tells how large a string to create.

<table> [Abstract Instantiable Class]

Also called a hashtable, a table is an unordered mapping between arbitrary keys and elements. Tables are the only predefined collections that are unstable under iteration. `make` of `<table>` supports the init-keyword `size:`. When specified, it provides a hint to the implementation of the class as to the expected number of elements to be stored in the table, which might be used to control how much space to initially allocate for the table. The default value is unspecified.

The class `<table>` provides an implementation of the Iteration Protocol, using the function `table-protocol`. Every concrete subclass of `<table>` must provide or inherit a method for `table-protocol`. See the Table Protocol section for details.

`<table>` has no direct instances; calling `make` on `<table>` will return an instance of `<object-table>`.

<object-table> [Sealed Instantiable Class]

A subclass of `<table>` that compares keys using `==`.

<unicode-string> [Sealed Instantiable Class]

This sealed subclass of `<vector>` provides efficient storage for elements that are sixteen-bit Unicode characters. It inherits lexicographic ordering from the `<string>` class. `make` of `<unicode-string>` supports the keywords `size:` and `fill:`. `size:` (default `0`) tells how large a `<unicode-string>` to create; `fill:` (default `' '`) specifies the initial value of every element.

<vector> [Abstract Instantiable Class]

The subclass of <array> whose instances are of rank one (i.e. have exactly one dimension). All one-dimensional arrays are vectors.

`<vector>` has no direct instances; calling `make` on `<vector>` returns an instance of `<simple-object-vector>`.

Each concrete subclass of `<vector>` must either provide or inherit an implementation of `size` that shadows the method provided by `<array>`.

Operations on Arrays

dimensions *array* ⇒ *sequence* [Function]

Returns the dimensions of *array*, as a sequence of integers. The consequences are undefined if the resulting sequence is modified.
This function forms the basis for all the other array operations. Each concrete subclass of `<array>` must either provide or inherit an implementation of this function.

```
? dimensions (make (<array>, dimensions: #(4, 4)))
#(4, 4)
```

size *array* ⇒ *size* [G. F. Method]

The method for `<array>` is equivalent to `reduce (\*, 1, dimensions (array))`.

There is a standard library for arrays, containing the following generic functions and methods, as defined below: `rank`, `row-major-index`, `aref`, `aref-setter`, and `dimension`.

rank *array* ⇒ *rank* [Generic Function]

Returns the number of dimensions (the rank) of *array*.

rank *array* ⇒ *rank* [G. F. Method]

The method for `<array>` computes `rank` by calling `size` on the `dimensions` of `array`.

row-major-index *array* `#rest` *subscripts* ⇒ *index* [Generic Function]

Computes the position according to the row-major ordering of `array` for the element that is specified by `subscripts`, and returns the offset of the element in the indicated position from the start of the `array`. An error is signaled if the number of subscripts is not equal to the rank of the array. An error is signaled if any of the subscripts are out of bounds for array.

row-major-index *array* `#rest` *subscripts* ⇒ *index* [G. F. Method]

The method for `<array>` computes the `index` using the result of calling `dimensions` on the `array`.

aref *array* `#rest` *indices* ⇒ *element* [Generic Function]

Returns the element of *array* indicated by *indices*. An error is signaled if the number of *indices* is not equal to the rank of the *array*. An error is signaled if any of the *indices* are out of bounds for *array*.

aref *array* `#rest` *indices* ⇒ *element* [G. F. Method]

The method for `<array>` calls `element` on the *array*, using the result of applying `row-major-index` to the *array* and *indices* as the key.

aref-setter *new-value array* `#rest` *indices* ⇒ *new-value* [Generic Function]

Sets the element of *array* indicated by *indices* to the new value and returns the new value. An error is signaled if the number of *indices* is not equal to the rank of the *array*. An error is signaled if any of the *indices* are out of bounds for *array*. An error is signaled if the *array* is limited to hold objects of a particular type and the new value is not an instance of that type.

aref-setter *new-value array* `#rest` *indices* ⇒ *new-value* [G. F. Method]

The method for `<array>` calls `element-setter` on the array and new value, using the result of applying `row-major-index` to the `array` and indices as the key.

dimension *array axis* ⇒ *dimension* [Generic Function]

Returns the `axis` dimension of `array`. `axis` must be a non-negative integer less than the rank of `array`. An error is signaled if `axis` is out of bounds for `array`.

dimension *array axis* ⇒ *dimension* [G. F. Method]

The method for `<array>` calls `element` on the result of calling `dimensions` on the `array`, using the `axis` number as the key.

Operations on Deques

push *deque new-value* ⇒ *deque* [Generic Function]

`push` augments *deque* by adding *new-value* to the front of the deque.

pop *deque* ⇒ *first-element* [Generic Function]

`pop` removes the first element from *deque* and returns it.

push-last *deque new-value* ⇒ *deque* [Generic Function]

`push-last` augments *deque* by adding *new-value* to the end of the deque.

pop-last *deque* ⇒ *last-element* [Generic Function]

`pop-last` removes the last element from *deque* and returns it.

add! *deque new-value* ⇒ *deque* [G.F. Method]

This method is the same as `push`. That is, the result is == to *deque*, and *deque* is destructively modified by this operation.

remove! *deque value* `#key` *test count* ⇒ *deque* [G.F. Method]

The result of `remove!` on a deque is == to the *deque* argument. *deque* is destructively modified by this operation.

Operations on Lists

The `<list>` class is partitioned into two subclasses, `<pair>` and `<empty-list>`. The classes `<list>`, `<pair>`, and `<empty-list>` are sealed; users cannot create new subclasses of `<list>`.

An **improper list** is a list that is not terminated by the empty list, either because it is terminated by something that is not a list, or because it is circular and thus non-terminating. Except when their behavior on improper lists is documented explicitly, collection or sequence functions are not guaranteed to return an answer when an improper list is used as a collection or a sequence. At the implementation's option, these functions may return the correct result, signal a `<type-error>`, or (in the case of a circular list) fail to return.

pair *head tail* ⇒ *pair* [Function]

`pair` creates a new pair whose head and tail values are as indicated.

```
? pair (1, 2)
#(1 . 2)
? pair (1, #(2, 3, 4, 5))
#(1, 2, 3, 4, 5)
```

list `#rest` *args* ⇒ *list* [Function]

`list` returns a list of the *args*, in order.

```
? list (1, 2, 3)
#(1, 2, 3)
? list (4 + 3, 4 - 3)
#(7, 1)
```

head *list* ⇒ *object* [Function]

If *list* is a pair, this function returns the value of the head slot. Otherwise, *list* is the empty list, and `head` returns the empty list.

```
? head (#(4, 5, 6))
4
? head (#())
#()
```

tail *list* ⇒ *object* [Function]

If *list* is a pair, this function returns the value of the tail slot. Otherwise, *list* is the empty list, and `tail` returns the empty list.

```
? tail (#(4, 5, 6))
#(5, 6)
? tail (#())
#()
```

head-setter *object pair* ⇒ *object* [Function]

This function sets the head of *pair* to contain *object* and returns *object*.

```
? define variable x = list (4, 5, 6)
#(4, 5, 6)
? head (x) := 9
9
? x
#(9, 5, 6)
```

tail-setter *object pair* ⇒ *object* [Function]

This function sets the tail of *pair* to contain *object* and returns *object*.

```
? define variable x = list (4, 5, 6)
#(4, 5, 6)
? tail (x) := #(9, 8, 7)
#(9, 8, 7)
? x
#(4, 9, 8, 7)
```

add! *list element* ⇒ *pair* [G.F. Method]

Equivalent to `(pair` *element list*`)`. That is, the result shares structure with *list* but will not be == to *list*.

remove! *list element* `#key` *test count* ⇒ *list* [G.F. Method]

`remove!` destructively modifies *list*. The result is not necessarily == to *list*.

size *list* ⇒ *{integer or #f}* [G.F. Method]

For circular lists, `size` is guaranteed to terminate and return `#f`. For non-circular lists, `size` returns an integer size value.

$list_1$ **=** $list_2 \Rightarrow$ *boolean* [G.F. Method]

For lists, = returns #f unless the two lists are the same size, corresponding elements of $list_1$ and $list_2$ are = and the final tails are =.

list **=** *sequence* ⇒ *boolean* [G.F. Method]

sequence **=** *list* ⇒ *boolean* [G.F. Method]

For mixed lists and sequences, = returns #f unless the *list* is not a dotted list, both have the same size, and elements with = keys are =.

Operations on Ranges

range #key *from to above below by size* ⇒ *range* [Function]

Creates a range. *from* (default 0) is the first value in the range. *by* (default 1) is the step between consecutive elements of the range. *to*, *above* and *below*, constrain the range. *to* is an inclusive bound, *above* is an exclusive lower bound and *below* is an exclusive upper bound. *above* and *below* constrain the range independent of the sign of *by*. See <range>

member? *val range* #key *test* ⇒ *boolean* [G.F. Method]

If *range* is unbounded, this method is guaranteed to terminate if *test* is == (the default).

size *range* ⇒ *size* [G.F. Method]

For unbounded ranges, size always terminates and returns #f. For finite ranges, size returns an integer.

copy-sequence *range* #key *start end*⇒ *new-range* [G.F. Method]

When applied to a range, copy-sequence returns another range, even though the class-for-copy of a range is the <list> class.

range1 **=***range2* ⇒ *boolean* [G.F. Method]

When called with two ranges, = always terminates, even if one or both ranges are unbounded in size.

`reverse` *range* ⇒ *new-range* [G.F. Method]

Reversing a range produces another range. An unbounded range cannot be reversed.

`reverse!` *range* ⇒ *range* [G.F. Method]

The result of `reverse!` on a range is == to the *range* argument. An unbounded range cannot be reversed.

`intersection` *range1 range2* `#key` *test* ⇒ *range* [G.F. Method]

`intersection` applied to two ranges and a test of == (the default) will produce another range as its result, even though the `class-for-copy` of a range is not `<range>`. If either *range1* or *range2* is unbounded, this method is guaranteed to terminate only if the *test* is ==.

Operations on Stretchy Vectors

`add!` *stretchy-vector new-element* ⇒ *stretchy-vector* [G.F. Method]

`add!` adds *new-element* at the end of *stretchy-vector*. The result is == to *stretchy-vector*, and *stretchy-vector* is destructively modified by this operation.

`remove!` *stretchy-vector element* `#key` *test count* ⇒ *stretchy-vector* [G.F. Method]

The result of `remove!` on a stretchy vector is == to *stretchy-vector*, and *stretchy-vector* is destructively modified by this operation.

Operations on Strings

Strings support lexicographic ordering through a shared implementation of <:

string1 **<** *string2* ⇒ *boolean* [G.F. Method]

When both arguments are strings, < compares strings lexicographically, using < on corresponding elements. If one string is a strict prefix of the other, the shorter string is considered the “smaller” one.

For variations on string comparison (such as comparisons that ignore case), different comparison operators must be used.

`as-lowercase` *string* ⇒ *new-string* [G.F. Method]

This method is equivalent to `map (as-lowercase,` *string*`)`.

```
? define variable x = "Van Gogh"
x
? as-lowercase (x)
"van gogh"
```

`as-lowercase!` *string* ⇒ *string* [G.F. Method]

This method is equivalent to `map-into (`*string*`, as-lowercase,` *string*`)`.

```
? define variable x = "Van Gogh"
x
? as-lowercase! (x)
"van gogh"
```

`as-uppercase` *string* ⇒ *new-string* [G.F. Method]

This method is equivalent to `map (as-uppercase,` *string*`)`.

```
? define variable x = "Van Gogh"
x
? as-uppercase (x)
"VAN GOGH"
```

`as-uppercase!` *string* ⇒ *string* [G.F. Method]

This method is equivalent to `map-into (`*string*`, as-uppercase,` *string*`)`.

```
? define variable x = "Van Gogh"
x
? as-uppercase (x)
"VAN GOGH"
```

element *unicode-string index* #key *default* ⇒ *character* [G. F. Method]

The class `<unicode-string>` provides a constant time implementation for the `element` function.

element-setter *character unicode-string index* ⇒ *character* [G. F. Method]

The class `<unicode-string>` provides a constant time implementation for the `element-setter` function.

element *byte-string index* #key *default* ⇒ *character* [G. F. Method]

The class `<byte-string>` provides a constant time implementation for the `element` function.

element-setter *character byte-string index* ⇒ *character* [G. F. Method]

The class `<byte-string>` provides a constant time implementation for the `element-setter` function.

Operations on Tables

Tables are "stretchy," in that they allow the addition and removal of keys, but they are not stretchy sequences (because they are not sequences). `<table>` and its subclasses are the only predefined classes that are stretchy but are not stretchy sequences.

`<table>` provides an implementation of the Iteration Protocol, using the function `table-protocol`. Every concrete subclass of `<table>` must provide or inherit a methodfor `table-protocol`. See the Table Protocol section for details.

remove-key! *table key* ⇒ *table* [Generic Function]

`remove-key!` modifies *table*so that it no longer has a key equal to *key*. Equality is determined by the table's test function.

element-setter *new-value table key* [G. F. Method]

Even if no element with the given key exists, `element-setter` for a table will add key and new-value to the table.

Operations on Vectors

vector `#rest` *args* ⇒ *vector* [Function]

This function returns a vector of the *args*, in order.

dimensions *vector* ⇒ *sequence* [G. F. Method]

Returns a sequence whose single element is the size of the vector.

element *simple-object-vector index* #key *default* ⇒ *element* [G. F. Method]

The class <simple-object-vector> provides a constant time implementation for the `element` function.

element-setter *new-element simple-object-vector index* [G. F. Method]
⇒ *new-element*

The class `<simple-object-vector>` provides a constant time implementation for the `element-setter` function.

The Iteration Protocol

All collections implement an **iteration protocol** that allows iteration to be specified abstractly. Many higher level operations on collections can be defined in terms of only the iteration protocol.

The iteration protocol centers on the notion of a “state” object for an iteration. Each collection class chooses its own most appropriate representation for an iteration state, and only the functions of the iteration protocol are affected by this choice.

Use of the iteration protocol is based on the assumption that the collection over which iteration occurs remains static for the duration of the iteration. That is, arbitrary changes to a mutable collection while an iteration is in progress may cause the iteration to produce incorrect results.

In general, two or more iterations over the same collection are not guaranteed to produce the same values in the same order, even if the collection is unaltered (but see the next section on Iteration Stability).

The iteration protocol consists of the following two functions:

`forward-iteration-protocol` *collection* [Generic Function]

```
⇒  initial-state             :: <object>
   limit                     :: <object>
   next-state                :: <function>
   finished-state?           :: <function>
   current-key               :: <function>
   current-element           :: <function>
   current-element-setter    :: <function>
   copy-state                :: <function>
```

The eight values returned by this function are used to implement iteration over the *collection* argument. Only the *collection* argument this function was called with may be used as the *collection* argument to functions returned by this function. Only the `initial-state` object and state objects returned by the `next-state` and `copy-state` functions may be used as the*state* argument to functions returned by this function. Only the `limit` object may be used as the *limit* argument to the `finished-state?` function.

`initial-state`
The initial iteration state object.

`limit`
A value that may be used by the `finished-state?` function to determine whether the iteration has been completed.

`next-state` *collection state* => *new-state*
This function “steps” the iteration by producing a new state from the associated *collection* and *state.* The `next-state` function may or may not modify the *state* argument; it is an error to use a state value after it has been passed to the associated `next-state` function. The `copy-state` function provides a mechanism for saving a particular state in an iteration for later resumption.

`finished-state?` *collection state limit*=> *boolean*
This function returns `#t` if the iteration of the collection has been completed, i.e., there are no other elements of the collection to consider. It returns `#f`

otherwise. It is an error to use a finished state in a call to the associated `next-state`, `current-element`, `current-key` or `current-element-setter` functions.

`current-key` *collection state* => *key*
This function returns the unique key associated with *state* in the *collection*. If the `current-key` function were called once with each *state* value produced during an iteration over a collection, the resulting sequence of values would contain every key from the collection exactly once.

`current-element` *collection state* => *element*
This function returns the element of *collection* "currently indicated" by *state*.

`current-element-setter` *value collection state* => *value*
This function sets the element of *collection* indicated by *state* to *value* and returns *value*. If the collection is not mutable, i.e. is not a generalized instance of `<mutable-collection>`, this function instead signals an error.

`copy-state` *collection state* => *new-state*
This function returns a state which represents the same point in the iteration over *collection* as is represented by *state.*

`backward-iteration-protocol` *collection* [Generic Function]

```
⇒  final-state             :: <object>
   limit                   :: <object>
   previous-state          :: <function>
   finished-state?         :: <function>
   current-key             :: <function>
   current-element         :: <function>
   current-element-setter  :: <function>
   copy-state              :: <function>
```

Some collection classes that are stable under iteration additionally support the ability to iterate in the reverse of the natural order, by providing a method on the generic function `backward-iteration-protocol`. The eight values returned by this function are analogous to the corresponding values returned by `forward-iteration-protocol`, except that the `final-state` object indicates the last element in the collection and the `previous-state` returns a state for the preceding element.

An example of the use of the iteration protocol is the following definition of a single-argument version of the `do` function:

```
define method do1 (f :: <function>, c :: <collection>)
  let (init, limit, next, end?, key, elt) =
                     forward-iteration-protocol(c);
  for (state = init then next(c, state),
                    until end?(c, state, limit))
    f(elt(c, state));
  end for;
end method do1;
```

The Table Protocol

Tables, also known as hashtables, map arbitrary keys to elements. Table keys may be any object, including complex objects such as strings. Each table has an associated **equivalence predicate** which is used to compare keys. The table maps keys that are equivalent under the predicate to the same table element.

An equivalence predicate is a boolean function of two arguments that returns true if and only if the arguments are "the same" according to some specified criteria. For a function to be used as an equivalence predicate, it must be reflexive, commutative, and transitive. That is, for a function F and any arguments X, Y, and Z in the domain of F, it must be true that (1) F(X,X) is true, (2) F(X,Y) is true if and only if F(Y,X) is true, and (3) if both F(X,Y) and F(Y,Z) are true then F(X,Z) is true.

An **equivalence class** (for an equivalence predicate) is a set of objects, or potential objects, that are all the same under the specified equivalence predicate and different from all objects not in the class. (This “class” is not meant to refer to Dylan classes as in `define class`.)

An object is said to be **visibly modified** with respect to an equivalence predicate if the modification changes the equivalence class of the object. The modifications that are visible to an equivalence predicate are determined by the definition of the predicate.

If an object X is a key in a table T and is visibly modified with regard to the test function of T, then the consequences are unspecified if any of the following objects are used as a key in any subsequent operations on T:

- The original object, X.
- Some object Y that is in the same equivalence class (as determined by the test function) as X prior to the modification of X.

- Some object Z that is in the same equivalence class (as determined by the test function) as X after the modification of X.

Each table also has an associated **hash function**, which is a function that relates table keys and table elements by computing **hash codes**. A hash code is a conceptual object consisting of a **hash id** and its associated **hash state**. (It is not an actual Dylan object.) A hash id is an integer encoding of an object. A hash state is an object of implementation-dependent type which is associated with a particular hash id and can be used by the implementation to determine whether the hash id has been invalidated. All hash functions have one argument, a key, and return two values, a hash id and a hash state, which together represent the hash code.

Each hash function is associated with a specific equivalence predicate, and must obey the following constraints:

1. The domain of the hash function must include the domain of valid arguments to the corresponding equivalence predicate. A hash function need not be defined for all Dylan objects, only those which are acceptable as arguments to the equivalence predicate.

2. All objects in a given equivalence class have the same (=) valid hash id, where validity is determined from the associated hash state.

In addition, a hash function should have the property that the hash ids computed by it are well distributed over the range of possible values. That is, it should be unlikely that two randomly chosen equivalence classes have the same valid hash id.

`table-protocol` *table* [Generic Function]

⇒ *test-function hash-function*

The class `<table>` provides an implementation of the Iteration Protocol, using the function `table-protocol`. Every concrete subclass of `<table>` must provide or inherit a method for `table-protocol`. `table-protocol` returns the following functions:

`test-function` *key1 key2* ⇒ *boolean*

The `test-function` is used to compare keys. It returns true if the keys are members of the same equivalence class according to the table's equivalence predicate.

`hash-function` *key* ⇒ *id state*

The `hash-function` computes the hash code of the *key*, using the hash function associated with the table's equivalence predicate. The hash code is returned as two values, *id* (an integer) and *state* (a hash state).

Users can choose either to define a new subclass of `<table>` for every test function they use, or to define a generic subclass of `<table>` which holds its test function and hash function in slots, according to their particular needs.

`table-protocol` *object-table* [G. F. Method]

⇒ *test-function hash-function*

The method for `<object-table>` returns `==` as the *test-function* and `object-hash` (described below) as the *hash-function*. It could be written as

```
define method table-protocol (table :: <object-table>)
      => test-function :: <function>, hash-function :: <function>;
  values(\==, object-hash);
end method table-protocol;
```

`merge-hash-codes` *id1 state1 id2 state2* `#key` *ordered* [Function]

⇒ *merged-id merged-state*

This function computes a new hash code by merging the argument hash codes in some implementation dependent way. *id1*, *id2*, and *merged-id* are all integers. *state1*, *state2*, and *merged-state* are all hash states. *ordered* is a boolean and determines whether the algorithm used to perform the merge is permitted to be order dependent. If `#f`, which is the default, then the merged result must be independent of the order in which the argument pairs are provided. If true, then the order of the argument pairs matters because the algorithm used need not be either commutative or associative in this case. Providing a true value for *ordered* is recommended when doing so will not cause the hash function to violate the second constraint on hash functions, because it may result in a better distribution of hash ids.

`$permanent-hash-state` [Constant]

An implementation-dependent hash state that indicates that the associated hash id is always valid, and does not depend on any mutable property of the object that can be changed without a visible modification to the object.

`object-hash` *object* ⇒ *id state* [Function]

This is the hash function for the equivalence predicate ==. It is made available as a tool for writing hash functions in which the object identity of some component of a key is to be used in computing the hash code. It returns a hash id (an integer) and associated hash state for the object, computed in some implementation dependent manner. The values returned by `object-hash` when called repeatedly on the same object might not be the same for each call. If the *id* value changes then the *state* value will also change.

Iteration Stability and Natural Order

A collection is **stable under iteration** if any two iterations over the collection are guaranteed to produce the same values in the same order (unless, of course, the collection has been modified). A collection that is not stable under iteration is said to be **unstable under iteration**.

Collections in general are not required to be stable under iteration, although several important subclasses are so required. In particular, sequences are required to be stable under iteration.

The order in which elements (and keys) are enumerated by the iteration protocol for a particular iteration is known as the "natural order" for that iteration over the collection. If a collection is stable under iteration, then every iteration over that collection will have the same natural order, and we may speak of the natural order of the collection itself. Most of the higher order operations are required to operate in "natural order," usually for the purpose of understanding interactions among side effects.

Collection Keys All collections in Dylan are "keyed" collections, i.e., all collections can be viewed abstractly as partial functions that take keys to elements. (This choice precludes pure sets from being considered collections, although it is straightforward simply to ignore the keys for a collection and consider it simply as a set of elements.) The `element` function implements this partial mapping of keys to elements. In addition, it is possible to obtain all the keys for a given collection, formed into a sequence collection.

`element` *collection key* `#key` *default* ⇒ *element* [Generic Function]

`element` returns the element associated with *key* in *collection*. If no element is associated with *key*, then the behavior of `element` depends on whether it was called with a *default* argument: if a *default* argument was passed, its value is returned; otherwise, an error is signaled.

All collections are required to implement `element`.

`key-sequence` *collection* ⇒ *keys* [Generic Function]

`key-sequence` returns a sequence containing the keys of *collection*. Although elements may be duplicated in a collection, keys, by their nature, must be unique; two different elements in a collection may not share a common key, even though distinct keys may yield identical elements.

The order in which the keys from *collection* appear in the key sequence is unspecified if *collection* is unstable under iteration. In particular, different calls to `key-sequence` with the same argument may yield differently ordered key sequences. If *collection* is stable under iteration, however, the resulting sequence of keys will be in the natural order for *collection*.

Iterating through a collection often involves examining keys as well as elements. To allow some freedom of implementation, the protocol recognizes two strategies for associating keys with states of iteration: explicitly or implicitly. The distinction between these strategies gives rise to two covering subclasses of `<collection>`. Every concrete subclass of `<collection>` must also be a subclass of either `<explicit-key-collection>` or `<sequence>` (or both):

`<explicit-key-collection>` [Abstract Class]

`<explicit-key-collection>` is the class of all collections that are not sequences.

`<sequence>` [Abstract Class]

`<sequence>` is a subclass of `<collection>` whose keys are consecutive integers ranging from zero up to (but not including) the size of the sequence.

Sequences must be stable under iteration, and the iteration order must match the order of keys. Thus, the key associated with a sequence's iteration state can be determined by keeping a counter in parallel with the iteration state, as in the next example.

Since every collection must be a sequence or an explicit-key-collection, it is always possible to keep track of keys during an iteration by defining appropriate methods on <sequence> and <explicit-key-collection>.[24] For example, consider these two methods for a function that applies a function f to each key-and-element pair in a collection. The method on <explicit-key-collection> uses the current-key function returned by forward-iteration-protocol; the method on <sequence> keeps a counter "alongside" the state during the iteration:

```
define method do-with-keys (f :: <function>,
                            c :: <explicit-key-collection>)
  let (init, limit, next, end?, key, elt) =
               forward-iteration-protocol(c);
  for (state = init then next(c, state),
                    until end?(c, state, limit))
     f(key(c, state), elt(c, state));
  end for;
end method do-with-keys;

define method do-with-keys (f :: <function>, c :: <sequence>)
  let (init, limit, next, end?, key, elt) =
                forward-iteration-protocol(c);
  for (key :: <integer> from 0,
                    state = init then next(c, state),
                    until end?(c, state, limit))
                 f(key, elt(c, state));
  end for;
end method do-with-keys;
```

Mutability

Some collections can be modified after they have been created; others cannot. To allow methods to distinguish between mutable and immutable collections, the <mutable-collection> and <stretchy-collection> mixin classes are provided:

<mutable-collection> [Abstract Class]

This abstract subclass of <collection> is used to indicate collections that can be modified. Every mutable collection is required to allow modification by implementing element-setter.

[24]That is, no more than two implementations are required for a function that operates on both keys and elements.

`<stretchy-collection>` [Abstract Class]

This abstract subclass of `<collection>` is used to indicate collections that may grow or shrink to accomodate adding or removing elements.

`element-setter` *new-value mutable-collection key* [Generic Function]

⇒ *new-value*

This function alters *mutable-collection* so that the value associated with *key* will subsequently be *new-value*.

An error is signaled if a program calls `element-setter` with a key that is not already a key to *collection*, unless the collection supports the dynamic addition of keys and adds the *key* and *new-value*.

Stretchy collections allow `element-setter` to be called with a key that is not present in the collection, expanding the collection as necessary to add a new element in that case. Each concrete subclass of `<stretchy-collection>` must provide or inherit a method for `element-setter` that behaves as follows when there is not already an element present for the indicated key:

- If the class is a subclass of `<explicit-key-collection>`, adds a new element to the collection with the indicated key.
- If the class is a subclass of `<sequence>`, first calls `size-setter` on the key + 1 and the collection to expand the sequence. The key must be a non-negative integer.

`<mutable-collection>` is mixed in with `<sequence>` and `<explicit-key-collection>`, respectively, to form the classes `<mutable-sequence>` and `<mutable-explicit-key-collection>`:

`<mutable-sequence>` [Abstract Class]

This class inherits from `<sequence>` and from `<mutable-collection>`.

`<mutable-explicit-key-collection>` [Abstract Class]

This class inherits from `<explicit-key-collection>` and `<mutable-collection>`.

Collection Alignment

Some operations on collections, such as mapping, are defined to allow the use of more than a single collection. The presence of potentially “unstable” collections, i.e., collections for which two iterations may produce values in different orders even though the collection remains unchanged, can create problems for multi-collection operations unless special care is taken. That is, if iteration is effectively performed in random order in the general case, then naively performing parallel iterations over two different collections would randomly pair values from the two collections, which would presumably have no meaning.

To prevent such random pairing, when operating on more than one collection, `map`, `do`, etc. must, in general, “align” the collections, by first computing the intersection of the collections’ key sequences and then using the random-access operations (`element` and `element-setter`) to operate on the collections themselves.

Collection alignment is only possible for collections whose key-test is identical, i.e. `key-test(c1) == key-test(c2)`. Otherwise, an error is signalled. As a concrete example, here is the two-collection case for `do`:

```
define method do2 (f :: <function>, c1 :: <collection>,
                   c2 :: <collection>)
  let keys = intersection(key-sequence(c1), key-sequence(c2));
  let (init, limit, next, end?, key, elt)
              = forward-iteration-protocol(keys);
  for (state = init then next(keys, state),
       until end?(keys, state, limit))
    let key = elt(keys, state);
    f(c1[key], c2[key]);
  end for;
end method do2;
```

Note that this definition has the potential for extreme inefficiency, because of its dependence on `element` and the potential loops implied by the calls to `key-sequence`.

An important special case of this problem is that of iterating over multiple sequences. In this case, the intersection of key sequences is clearly the non-negative integers up to the length of the shortest sequence. Further, unlike collections in general, sequences are required to exhibit stability, so no explicit computation of key sequences need be made. Instead, it is correct (and much more efficient) simply to iterate until one or more of the sequences is exhausted. Here is a concrete example for `do2`:

```
define method do2 (f :: <function>, c1 :: <sequence>, c2 ::
<sequence>)
  let (init1, limit1, next1, end1?, key1, elt1)
                                = forward-iteration-protocol(c1);
  let (init2, limit2, next2, end2?, key2, elt2)
                                = forward-iteration-protocol(c2);
  for (state1 = init1 then next1(c1, state1),
       state2 = init2 then next2(c2, state2),
       until (end1?(c1, state1, limit1) |
              end2?(c2, state2, limit2)))
    f(elt1(c1, state1), elt2(c2, state2));
  end for;
end method do2;
```

Most cases of iteration over more than a single collection will be iterations over sequences, rather than over arbitrary collections. Since this case *is* efficient, the requirement for alignment is probably not burdensome in practice.

Any iteration operations that modify a collection must also include the key sequence of the target collection during “alignment” of the collection arguments. For example, consider these definitions for a simplified `map-into` function:

```
define method map-into1 (target :: <mutable-collection>,
                         f :: <function>,
                         source :: <collection>)
  let keys = intersection(key-sequence(target),
                          key-sequence(source));
  let (init, limit, next, end?, key, elt)
              = forward-iteration-protocol(keys);
  for (state = init then next(keys, state),
       until end?(keys, state, limit))
    let key = elt(keys, state);
    target[key] := f(source[key]);
  end for;
end method map-into1;

define method map-into1 (target :: <mutable-sequence>,
                         f :: <function>,
                         source :: <sequence>)
  let (init1, limit1, next1, end1?, key1, elt1, setter)
              = forward-iteration-protocol(target);
  let (init2, limit2, next2, end2?, key2, elt2)
              = forward-iteration-protocol(source);
  for (state1 = init1 then next1(target, state1),
       state2 = init2 then next2(source, state2),
       until (end1?(target, state1, limit1) |
              end2?(source, state2, limit2)))
    setter(target, state1, f(elt2(source, state2)));
  end for;
end method map-into1;
```

Defining a New Collection Class: A Summary

Every collection class must provide an implementation of the iteration protocol (`forward-iteration-protocol`; backward is optional).

Every collection must provide or inherit methods for `element` and `key-test`. If the collection is also a `<mutable-collection>`, it must provide or inherit a method for `element-setter`. A collection that is not a `<mutable-collection>` must provide an implementation of `class-for-copy`.

Individual collection classes may impose further requirements on their subclasses. For example, concrete subclasses of `<table>` must provide or inherit a method for `table-protocol`.

For efficiency, it may be desirable to provide specialized implementations for certain generic functions. Collections that can implement functions such as `size` or `member?` more efficiently should do so. Sequences that can reuse storage to implement functions such as `reverse!` and `sort!` should do so.

10. Characters and Symbols

Characters

See also the description of comparison operations, which discuss the comparison of characters.

`<character>` [Instantiable Class]

`<character>` is a subclass of `<object>`.

`as-uppercase` *character* ⇒ *uppercase-character* [G.F. Method]

The `<character>` method on `as-uppercase` returns the upper-case equivalent for *character*. If *character* already is uppercase or does not exist in two cases, it is returned unchanged.

`as-lowercase` *character* ⇒ *lowercase-character* [G.F. Method]

The `<character>` method on `as-lowercase` returns the lower-case equivalent for *character*. If *character* already is lowercase or does not exist in two cases, it is returned unchanged.

`as` `<integer>` *character* ⇒ *integer* [G.F. Method]

This method on `as` returns a numeric equivalent for *character*. The integer returned is implementation dependent.

`as` `<character>` *integer* ⇒ *character* [G.F. Method]

This method on `as` returns the character equivalent to *integer*. The meaning of *integer* is implementation dependent.

Symbols

The <symbol> class provides a built-in, non-case-sensitive dictionary that associates a string with a unique immutable object that can be compared with == (which should be faster than calling a string-comparison routine). This dictionary is accessed through the `as` function: `as(<symbol>, `*string*`)` and `as(<string>, `*symbol*`)`. Any string can be used, even ones that aren't allowed as variable names.

For symbols that use the same restricted character set that are allowed in variable names, the "keyword syntax" `foo:` (the symbol followed by "`:`") may be used everywhere that keywords are allowed in the grammar. The trailing colon is not part of the symbol.

```
? example: == #"Example"
#t
```

The case of symbols is remembered from the first time the symbol is entered. `as(<string>, `*symbol*`)` returns the original case.

<symbol> [Class]

<symbol> is a subclass of <object>.

as <symbol> *string* ⇒ *symbol* [G.F. Method]

This method on `as` returns the symbol that has the name *string*. If the symbol does not yet exist, it is created. This method on `as` will always return the same symbol for strings of the same characters, without regard to alphabetic case.

```
? as (<symbol>, "foo")
#"foo"
? #"FOO" == as (<symbol>, "Foo")
#t
? #"Foo"
#"foo"
```

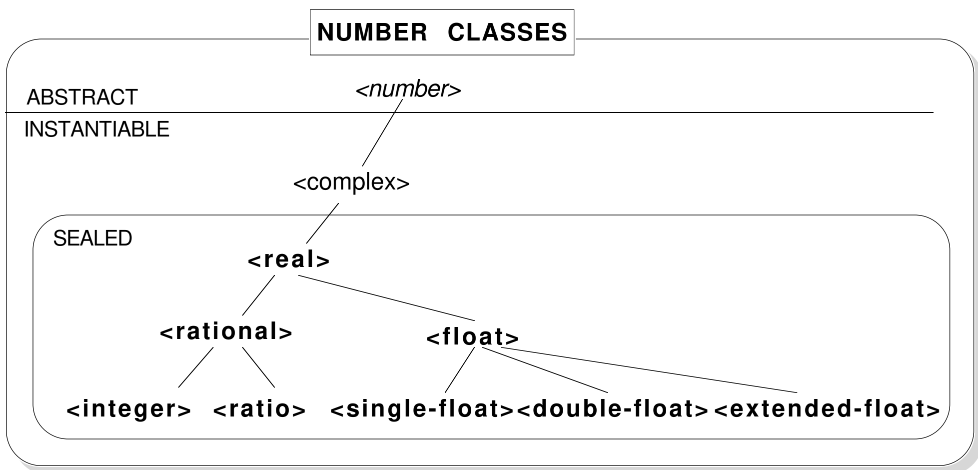
as <string> *symbol* ⇒ *string* [G.F. Method]

This method on `as` returns the name of the symbol, which will be a string.

```
? as (<string>, #"Foo")
"Foo"
```

11. Numbers

Dylan supports several kinds of numerical representations. The classes for these representations are placed in a hierarchy of abstract classes corresponding to mathematical number types. The abstract classes have no direct instances but are useful for specialization. The complete class hierarchy is shown below. Abstract classes are shown in italics. Sealed classes are shown in bold.



The `<single-float>`, `<double-float>`, and `<extended-float>` classes implement the IEEE standard floating-point formats[25].

Automatic Type Conversion

The Dylan rules for automatic type conversion are the same as in Common Lisp (X3J13): floating-point contagion (with rational contagion for comparisons) and rational canonicalization. Argument coercions are implemented by the individual methods on the arithmetic functions.

[25]With one exception: comparison operations may not behave in IEEE fashion when performed on NaNs.

Numeric Classes

`<number>` [Abstract Class]

`<real>` [Sealed Class]

`<float>` [Sealed Class]

`<single-float>` [Sealed Class]

`<double-float>` [Sealed Class]

`<extended-float>` [Sealed Class]

`<rational>` [Sealed Class]

`<ratio>` [Sealed Class]

`<integer>` [Sealed Class]

`<complex>` [Sealed Class]

General Arithmetic Functions

Properties

`odd?` *integer* ⇒ *boolean* [Generic Function]

`even?` *integer* ⇒ *boolean* [Generic Function]

`zero?` *number* ⇒ *boolean* [Generic Function]

`positive?` *real* ⇒ *boolean* [Generic Function]

`negative?` *real* ⇒ *boolean* [Generic Function]

`integral?` *number* ⇒ *boolean* [Generic Function]

These functions test a number for the given property and return a Boolean result.

Arithmetic Operations

+ *number1 number2* ⇒ *number* [Generic Function]

***** *number1 number2* ⇒ *number* [Generic Function]

- *number1 number2* ⇒ *number* [Generic Function]

/ *number1 number2* ⇒ *number* [Generic Function]

These functions return the sum, product, difference, and quotient of their arguments, respectively. Division by zero signals an error.

Use the name of the function (`+`, `*`, `-`, or `/`) when you use the function in an infix expression:

```
5 + 6 * 4
```

Use the name of the function preceded by a backslash (`\+`, `\*`, `\-`, or `\/`) when you are using the function in any other way, such as adding new methods to it or passing it as a functional argument:

```
define class <my-number> (<number>) end class;

define method \+ (a :: <my-number>, b :: <my-number>)
  my-personal-addition-method(a, b);
end method;
```

negative *number* ⇒ *number* [Generic Function]

This function returns the additive inverse of its argument. The unary minus operator is defined to call `negative`.

floor *real* ⇒ *integer real* [Generic Function]

ceiling *real* ⇒ *integer real* [Generic Function]

round *real* ⇒ *integer real* [Generic Function]

truncate *real* ⇒ *integer real* [Generic Function]

These functions are equivalent to the one-argument forms of the like-named Common Lisp (X3J13) functions.

floor/ *real1 real2* ⇒ *integer real* [Generic Function]

ceiling/ *real1 real2* ⇒ *integer real* [Generic Function]

round/ *real1 real2* ⇒ *integer real* [Generic Function]

truncate/ *real1 real2* ⇒ *integer real* [Generic Function]

These functions are equivalent to the two-argument forms of `floor`, `ceiling`, `round`, and `truncate` in Common Lisp (X3J13). Division by zero signals an error.

modulo *real1 real2* ⇒ *real* [Generic Function]

`modulo` returns the second value of `floor/ (` *real1* `,` *real2*`).`

remainder *real1 real2* ⇒ *real* [Generic Function]

`remainder` returns the second value of `truncate/ (` *real1* `,` *real2*`).`

number1 **^** *integer2* ⇒ *number* [Generic Function]

Returns *number1* raised to the power *integer2*.

abs *number* ⇒ *number* [Generic Function]

logior #rest *integers* ⇒ *integer* [Generic Function]

logxor #rest *integers* ⇒ *integer* [Generic Function]

logand #rest *integers* ⇒ *integer* [Generic Function]

lognot *integer* ⇒ *integer* [Generic Function]

logbit? *index integer* ⇒ *boolean* [Generic Function]

ash *integer count* ⇒ *integer* [Generic Function]

The generic functions abs, logior, logxor, logand, lognot, ash are as defined in Common Lisp. logbit? is equivalent to Common Lisp's logbitp.

rationalize *number* ⇒ *number* [Generic Function]

numerator *number* ⇒ *number* [Generic Function]

denominator *number* ⇒ *number* [Generic Function]

The generic functions rationalize, numerator, anddenominator are as defined in *Revised^4 Report on Scheme*.

lcm *integer1 integer2* ⇒ *integer* [Generic Function]

gcd *integer1 integer2* ⇒ *integer* [Generic Function]

These functions return the least common multiple and greatest common divisor of *integer1* and *integer2*, respectively

min *real* #rest *more-reals* ⇒ *real* [Function]

max *real* #rest *more-reals* ⇒ *real* [Function]

min returns the argument that is least (closest to negative infinity). max returns the argument that is greatest (closest to positive infinity). The methods operate by calling <.

12. Other Operations

Functional Operations

The following operations are used to create new functions from other functions or objects. Often the Dylan compiler will have special knowledge of these operations, to allow for efficient in-line compilation.

compose *function1* `#rest` *more-functions* ⇒ *function* [Function]

When called with just a single argument, `compose` returns that argument.

When called with two arguments, `compose` returns a function that applies the second function to its arguments and then applies the first function to the (single) result value.

With three or more arguments, `compose` composes pairs of argument functions, until a single composite function is obtained. (It doesn't matter if the pairings are done from the left or from the right, as long as the order of application is preserved.)

```
? define method square (x :: <number>) x * x end;
square
? define method square-all (coords :: <sequence>)
      map (square, coords);
  end;
square-all
? define method sum (numbers :: <sequence>)
      reduce1 (\+, numbers);
  end;
sum
? define constant distance = compose(sqrt, sum, square-all);
distance
? distance ( #(3, 4, 5) );
7.0710678118654755
```

complement *predicate* ⇒ *function* [Function]

`complement` returns a function that applies *predicate* to its arguments. If the *predicate* returns `#f`, the complement returns `#t`; otherwise, the complement returns `#f`. For example, `odd?` could be defined as `complement (even?)`.

```
? map (female?, list (michelle, arnold, roseanne));
#(#t, #f, #t)
? map (complement (female?), list (michelle, arnold, roseanne));
#(#f, #t, #f)
```

disjoin *predicate1* `#rest` *more-predicates* ⇒ *function* [Function]

`disjoin` returns a single function, termed the disjunction of its argument functions. The disjunction accepts any number of arguments and operates by applying the predicates, in order, to the arguments. If any of the predicates returns true, the remaining predicates (if any) are not applied, and the true result is returned. Otherwise, all the predicates will be applied, and `#f` returned.

A disjunction is similar to an | expression of calls to the predicates.

conjoin *predicate1* `#rest` *more-predicates* ⇒ *function* [Function]

`conjoin` returns a single function, termed the conjunction of its argument functions. The conjunction accepts any number of arguments and operates by applying the predicates, in order, to the arguments. If any of the predicates returns `#f`, the remaining predicates (if any) are not applied and `#f` is immediately returned. Otherwise, all the predicates will be applied, and the result of the last application is returned.

A conjunction is similar to an `&` expression of calls to the predicates.

curry *function* `#rest` *curried-args* ⇒ *new-function* [Function]

`curry` returns a function that applies *function* to *curried-args* plus its own arguments, in that order. For example `curry (\=, "x")` is a predicate that tests for equality with the string `"x"`; `curry (\+, 1)` is an incrementing function; `curry (\>, 6)` is a predicate that returns true for values less than 6; `curry (concatenate, "set-")` is a function that concatenates the string `"set-"` to any additional sequences it is passed.

```
? map (curry (\+, 1), #(3, 4, 5))
#(4, 5, 6)
```

rcurry *function* `#rest` *curried-args* ⇒ *new-function* [Function]

`rcurry` ("right" curry) operates just like `curry`, except it allows the rightmost arguments of *function* to be specified in advance, rather than the leftmost arguments. For example, `rcurry (\>, 6)` is a predicate that returns true for values greater than 6.

```
? define constant yuppify = rcurry (concatenate, ", ayup");
yuppify
? yuppify ("I'm from New Hampsha");
"I'm from New Hampsha, ayup"
```

always *object* ⇒ *function* [Function]

`always` returns a function that can be called with any number of arguments. The function ignores its arguments and always returns *object*.

```
? always (1) ("x", "y", "z")
1
? always (#t) (#f, #f)
#t
```

Function application

apply *function* #rest *args* ⇒ *values* [Function]

apply calls *function* with *args* as the arguments. The last *arg* must be a sequence. The elements of the sequence are spread out and taken as individual arguments to *function*.

```
? apply(\+, #(1, 2))
3
? 1 + 2
3
? define constant math-functions = list (\+, \*, \/, \-))
? math-functions
#({generic +}, {generic *}, {generic /}, {generic -})
? first (math-functions)
{generic +}
? apply (first (math-functions), #(1, 2))
3
```

Identity function

identity *object* ⇒ *object* [Function]

identity simply returns *object*.

13. Conditions

Background

A long-standing problem of software engineering is the need to develop an organized way to deal with exceptions, situations that must be handled gracefully but that are not conceptually part of the normal operation of the program. A number of programming languages contain linguistic features for exceptions, among them Common Lisp, C++, PL/I, and Ada.

Of course it is possible to program exception handling without using special linguistic features. For example, all functions could return an extra result that indicates whether they succeeded or failed, functions could take an extra argument that they consult if an exception occurs, or a designated exception-handling function could be called whenever a problem arises. All of these approaches have been used in one real-life system or another, but they are deficient in two ways. First, they are too informal and don't provide enough structure to allow an organized, systematic approach to exception handling. Second, and more importantly, the first two approaches do not provide textual separation between "normal code" and "code for dealing with exceptions"; exception-related code is sprinkled throughout the program. This leads to two problems: one is the well-known mistake of forgetting to test error codes and thus failing to detect an exception, perhaps one that "could never happen;" the other is that program clarity is lost because it isn't easy to think about the main flow of the program while temporarily ignoring exceptions.

Thus, the most important requirements of a linguistic exception-handling facility are to provide overall structure, to eliminate the possibility of failing to notice an exception, and to provide a clean separation between "normal code" and "code for dealing with exceptions."

All exception systems involve the concept of "signal" (sometimes with a different name, such as "raise" or "throw") and the concept of "handle" (sometimes with a different name such as "on-unit" or "catch"). All exception systems considered here dynamically match signalers with handlers, first invoking the most recently established matching handler still active, and then, if that matching handler declines to handle the exception, invoking the next most recent matching handler, and so on. Thus, exception systems really are a way of establishing a run-time connection between a signaler and a handler, as distinct from the usual fixed connection between a caller and a callee through function-name matching.

The major design choices in exception systems—name-based versus object-based, calling versus terminating, and formal versus ad-hoc recovery—are discussed in the following paragraphs.

PL/I and Ada have name-based exception systems. A program signals a name, and a handler matches if it handles the same name or "any." The name is a constant in the source text of the program, not the result of an expression.

C++ and Common Lisp have object-based exception systems. A program signals an object, and a handler matches if it handles a type that object belongs to. Object-based exceptions are more powerful, because the object can communicate additional information from the signaler to the handler, because the object to be signaled can be chosen at run-time rather than signaling a fixed name, and because type inheritance in the handler matching adds abstraction and provides an organizing framework.

The C++ object-based exception system has complicated inheritance rules and allows any object whatsoever to be used as an exception. Common Lisp, in contrast, allows only objects that inherit from a certain built-in class to be used as exceptions, thus limiting the inheritance rules to class inheritance and providing a class on which to hang any protocols specific to exceptions.

Ada and C++ have terminating exception systems: before a handler receives control, all dynamic state between the handler and the signaler is unwound, as if signaling were a non-local goto from the signaler to the handler. PL/I has a calling exception system: when a handler receives control, the signaler is still active and control can be returned to it, as if signaling were a function call from the signaler to the handler. Common Lisp has both, with the calling semantics regarded as fundamental and the terminating semantics explained in terms of it. Note that calling semantics need not imply that the signal operation can return the way a function call returns; Common Lisp provides a "restart" mechanism that signalers use to grant handlers permission to return. The Common Lisp model shows that terminating versus calling is a property of handlers, not of exception systems as a whole.

Terminating exception systems are acceptable for errors. However, they do not work for an exception that is not an error and doesn't require termination, either because there is a default way to handle it and recover or because it can safely be ignored by applications that don't care about it. Non-error exceptions are quite common in networked environments, in computers with gradually expiring resources (such as batteries), in complex user interfaces, and as one approach for reflecting hardware exceptions such as page protection violations or floating-point overflow to the application.

Most languages have not formalized how to recover from exceptions, leaving programmers to invent ad hoc mechanisms. Common Lisp provides "restarts," which is an object-based way of establishing a run-time connection from a handler back to the signaler or to any other part of the dynamically active program. A handler can choose to return control to any available restart, or else can mimic terminating semantics, using the language's normal non-local exit mechanisms. Restarts can also be a good way for programs to indicate to an interactive debugger what the end user's choices are for recovery from an exception that lands in the debugger. In this sense, the debugger can be understood as just a complicated handler. Unfortunately some details of restarts are somewhat poorly designed in Common Lisp.

It is necessary to have a way to clean up when execution of a function is terminated by a non-local exit initiated either by the function itself or by something it explicitly or implicitly called. In C++ this includes invoking destructors for automatic objects, as well as application-specific cleanup. Ada, C++, and the Multics dialect of PL/I associate this action with exception handling. Common Lisp realizes it is related but different and calls it `unwind-protect`. Dylan uses the Common Lisp approach, except that Common Lisp's `unwind-protect` becomes a cleanup clause in Dylan's`block` statement.

For detailed information on exceptions in Common Lisp, refer to chapter 29 of *Common Lisp: the Language*, Second Edition, by Guy L. Steele Jr. For detailed information on exceptions in C++ (not yet implemented in many C++ implementations but adopted into the draft proposed ANSI C++ standard), refer to chapter 15 of *The Annotated C++ Reference Manual*, by Margaret A. Ellis and Bjarne Stroustrup. Although it is not necessary to read either document to understand the Dylan condition system, they may be helpful as sources of background information.

Overview

The Dylan exception facility is object-based. It uses calling semantics but also provides terminating handlers. It provides formal recovery.

Conditions are a dynamically bound, type-based mechanism for finding and invoking functions. When an exceptional situation occurs, it is indicated to the calling code by signaling a condition. The condition facility locates an applicable handler and calls it. An applicable handler is one that matches the signaled condition by type and by (an optional) test function associated with the handler. The condition system is simply a way for a signaler and a handler to be introduced to each other. Once they have met, they communicate through an ordinary function call.

Like any function, the called handler either returns some values or takes a non-local exit. Either way, the handler has handled the condition, and the act of signaling is completed. A dynamic handler also has the option of declining to handle the condition, passing the buck to the next applicable handler, by calling a next-handler function passed to the dynamic handler as an argument. A default handler cannot decline because it is always the last applicable handler. Every signaled condition is handled, because the system ensures that there is always an applicable default handler.

The following terms describe portions of the dynamic state of execution. These terms come from the stack model of function calling.

outside stack The state existing just before the handler was established

signaling unit The conceptual program component that includes the form that signaled the condition and does not include the form that established the handler. This informal concept provides a notion of where the interface boundary between the signaler and the handler lies.

middle stack The state existing just before the signaling unit was called, minus the outside stack

inside stack The state existing just before signaling occurred, minus the middle stack and outside stack

The simplest way to handle a condition is to terminate the signaling unit by taking a non-local exit established in the outside stack. The convenient `exception` clause of the `block` statement is provided for this purpose.

A less common handling style is to terminate the signaling unit by taking a non-local exit established in the middle stack, leaving the handler in force.

Instead of terminating, a handler can recover, returning control to the signaling unit. This can be done either by returning values that the signaling unit will understand or by taking a non-local exit established in the inside stack. Returning or exiting can be done directly, or a more formal recovery mechanism called restarting can be used. The advantage of the formal mechanism is the advantage of any formal interface: it provides more assurance that the handler and the signaling unit agree on the meaning of what they are doing and provides some isolation of the handler from names and data representations internal to the signaling unit.

Returning values or non-locally exiting directly, rather than restarting, is a less formal mechanism with an increased likelihood of accidentally violating the protocol or contract between a signaler and a handler. Nonetheless it is allowed, in the spirit of freedom and because it is the primitive upon which restarting is built. When the direct mechanism is used, it should be used with due care.

A handler restarts by signaling a restart. This is the formal recovery mechanism. Any values needed for recovery are passed in the restart (that is, in initialization arguments that the restart remembers either in slots or in any other way it chooses). The restart is handled by a restart handler which either returns or takes a non-local exit. If the restart handler returns some values, `signal` returns those values and the handler that called `signal` also returns them. The call to `signal` from the signaling unit that signaled the original condition returns the same values, and the signaling unit recovers as directed by those values. If the restart handler was established by the signaling unit, then it was established in the inside stack and typically takes a non-local exit in the inside stack. It is just as valid to invoke a restart handler established in the outside or middle stack by a program component other than the signaling unit.

For every condition class there should be a "recovery protocol" that defines the meaning of handling by returning, the meaning of the values returned, and which restart handlers are supposed to be established by the signaling unit. The recovery protocol tells the handler what to expect from the signaler. For many condition classes, this is the empty protocol: handling by returning isn't allowed, and no particular restart handlers are provided. In this case only handling by terminating is possible. (Terminating might be accomplished by signaling a restart whose handler was established in the outside or middle stack, or by an ordinary non-local exit.) The recovery protocol for a subclass should be compatible with the recovery protocol of a superclass. That is, a handler that applies the superclass's recovery protocol, when the condition actually signaled was a direct instance of the subclass, should operate correctly.

An example recovery protocol for a hypothetical `<unbound-slot>` condition could be: returning a value uses that value as if it had been the contents of the slot; a restart handler for `<new-value>` is available; `<new-value>` has initialization arguments `value:`, the value to use, and `permanent:`, `#t` to store the value into the slot or `#f` (the default) to leave the slot unbound.

At present, no formal mechanism is provided for describing recovery protocols; they are left to the documentation of a condition class. Introspective functions are provided for discovering which recovery facilities are actually available, but this is different from (and sometimes is a superset of) the recovery facilities guaranteed by a recovery protocol always to be available.

The debugger is the condition handler of last resort which receives control if no program-provided handler handles a serious condition. (This is true even if the "debugger" provided cannot analyze or intervene in the execution of programs but can only abort or restart them. The debugger might be merely a "core dumper," a "bomb box," or something similar.) An interactive debugger ought to offer the user the ability to signal any restart for which a restart handler is applicable and to return if the condition's recovery protocol allows it. This could, for example, be done with a menu titled "Recovery."

Specification

The following sections outline the classes and operations on conditions available in Dylan.

Format Strings

A "format string" is a string template into which values can be inserted to construct a message. The two-character sequences `%d`, `%b`, `%o`, `%x`, `%c`, `%s`, and `%=` are replaced by the next element of the associated sequence of "format arguments." Upper and lower case letters are equivalent in these format directives. The inserted value is formatted according to the following table:

directive	argument type	textual format
`%d`	`<integer>`	decimal number
`%b`	`<integer>`	binary number
`%o`	`<integer>`	octal number
`%x`	`<integer>`	hexadecimal number
`%c`	`<character>`	character (with no quotes)
`%s`	`<string>`	string (with no quotes)
`%s`	`<condition>`	error message (with no quotes)
`%=`	`<object>`	unspecified but works for any object
`%%`	*none*	literal `%`

The text printed by the `%=` format directive for any given object is implementation-defined. The behavior when a format argument is not of the type specified in the table above is implementation-defined. The behavior when too many or too few format arguments are supplied is implementation-defined.

The two-character sequence `%%` does not consume a format argument, but inserts a `%` character.

All other uses of the `%` character in a format string are implementation-defined. Extensions in the style of Common Lisp `format` or C `printf` might exist in some implementations but are not portable. This is not attempting to be a general text-formatting package, just a simple least common denominator for casual construction of error messages.

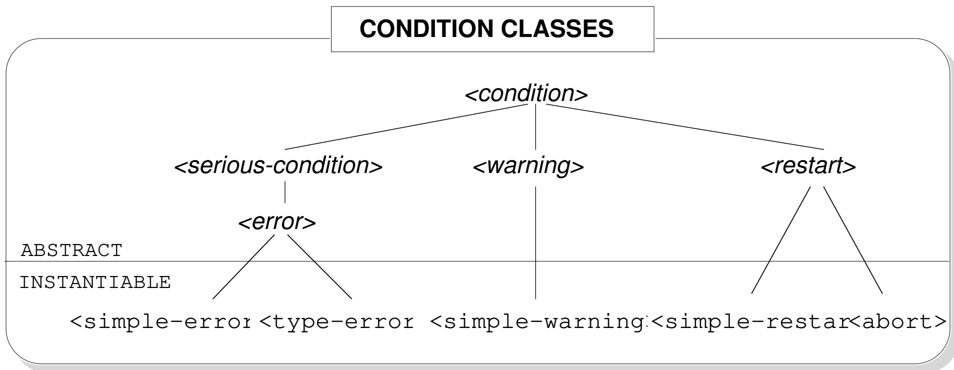
There is no standard way to get the "error message" from a condition, although it can be inserted into another message by using the `%s` directive in a format string. Debuggers get the error message using implementation-dependent mechanisms. When a streams library is present, it might include a function to get the "error message" from a condition.

There is no standard way for a user-defined condition class that does not inherit from `<simple-error>` `<simple-restart>`, or `<simple-warning>` to supply an "error message." Individual platforms, application frameworks, or user interfaces should specify a mechanism for this that is appropriate to their needs.

Classes

The classes described in this section are defined by Dylan. In addition, numerous condition classes specific to particular Dylan built-in functions or user functions will no doubt exist. Note that there can be other direct subclasses of `<condition>` besides the three specified ones. Also note that the subclass relationships shown below are not necessarily direct subclass relationships (i.e. implementation-specific classes can be inserted in the class hierarchy).

Abstract classes are shown in italics.



<condition> [Abstract Class]

A condition is a general instance of the abstract class `<condition>`. It is an object that represents the occurrence of an exception. Conditions have indefinite extent and no special restrictions. A program that does not encounter any exceptional situations will not implicitly allocate any conditions.

There is a default handler for `<condition>` that returns `#f`.

<serious-condition> [Abstract Class]

`<serious-condition>` is the abstract class of all conditions that cannot safely be ignored. There is a default handler for `<serious-condition>` that calls the debugger, using an unspecified mechanism.

<error> [Abstract Class]

`<error>` is the abstract class of all conditions that represent something invalid about the program (as opposed to environmental conditions such as running out of memory or battery power, or inability to establish a network connection, for example). `<error>` is a subclass of `<serious-condition>`. `<error>` is distinct from `<serious-condition>` so one can establish a handler for errors that does not also trap unpredictable environmental exceptions such as network problems.

<simple-error> [Instantiable Class]

`<simple-error>` is the concrete class of error conditions that consist of just an error message constructed from a format string and arguments; it is a subclass of `<error>`. The recovery protocol is empty. The initialization arguments `format-string:` and `format-arguments:` are accepted. The accessor functions `condition-format-string` and `condition-format-arguments` are applicable.

<type-error> [Instantiable Class]

`<type-error>` is a concrete subclass of `<error>` used by `check-type`. The recovery protocol is empty. The initialization arguments `value:` and `type:` are accepted. The accessor functions `type-error-value` and `type-error-expected-type` are applicable.

<sealed-object-error> [Instantiable Class]

`<sealed-object-error>` is a subclass of `<error>`

<warning> [Abstract Class]

`<warning>` is an abstract subclass of `<condition>`. There is a default handler for `<warning>` that displays the warning in a user-interface dependent way and then returns `#f`. The recovery protocol is that any value can be returned and will be ignored.

<simple-warning> [Instantiable Class]

`<simple-warning>` is a concrete subclass of `<warning>` used by `signal`. The recovery protocol is the same as for `<warning>`. The initialization arguments and accessor functions are the same as for `<simple-error>`.

<restart> [Abstract Class]

A restart is a general instance of the abstract class `<restart>`. `<restart>` is a subclass of `<condition>`. There is a default handler for `<restart>` that signals an error reporting an attempt to use a restart for which no restart handler was established. The recovery protocol concept is not applicable to restarts. The initialization argument `condition:` is accepted and ignored by `<restart>`; some subclasses save the value of this initialization argument and use it to associate a restart with a particular condition from which the restart can recover. At present, none of these subclasses are defined as part of the language. Other restarts do not care; they can recover from any condition.

<simple-restart> [Instantiable Class]

`<simple-restart>` is a concrete subclass of `<restart>` that is used by `cerror`. The initialization arguments `format-string:` and `format-arguments:` are accepted. The accessor functions `condition-format-string` and `condition-format-arguments` are applicable. Typical implementations will use the format string and format arguments to produce a description of the restart.

<abort> [Instantiable Class]

`<abort>` is a concrete subclass of `<restart>` whose handlers are expected to terminate execution of the current application command, or similar unit of execution, and return control to something like an application command loop. This is comparable to command-period on the Macintosh. The exact details of this feature depend on the particular environment, of course, but signaling an instance of `<abort>` is a uniform way to "get out."

Basic Operator for Signaling

signal *condition* ⇒ *values* [Function]

Signals the *condition*, trying each active dynamic handler, the most recent first. If all dynamic handlers decline, `signal` calls `default-handler (`*condition*`)`. If a handler returns, all the values that it returned are returned from `signal`. If `signal` returns when *condition*'s recovery protocol does not allow returning, some handler has violated protocol; `signal` does not check for this error. If *condition* is a restart, the caller of `signal` should always assume that it might return.

Additional operators for signaling, defined for convenience, are described in a later section.

Basic Operator for Handling Conditions

let handler *condition* **=** *handler* [Local Declaration]

`let handler` establishes a condition handler during the dynamic extent of the evaluation of the rest of the enclosing body. It is scoped as in `let`.

condition should be of the form

type

or `(` *type* [`, test:` *test*] [`,init-arguments:` *init-arguments*] `)`

type, *handler*, *test*, and *init-arguments* are evaluated before evaluation of the rest of the enclosing body begins.

type	The handler applies to conditions that are general instances of *type*.
test	A function called with a condition that is a general instance of *type* and that returns true if this handler applies to this condition. The default is a function that always returns true. An example usage is a restart handler for restarting only from a particular condition object, for example, restarting from an unbound-slot error by setting the slot and retrying the invocation of the accessor. The `<set-and-continue>` restart condition will have the signaled `<unbound-slot>` condition in a slot, and the handler's test will check for it. (These class names are invented for this example and are not part of the specification.)
handler	A function called to handle a condition that matches *type* and passes *test*. The values of the arguments are the condition being signaled and the next handler function. The function handles the condition by taking a non-local exit, returning values according to the condition's recovery protocol, or tail-recursively calling `signal` of a restart. The function can decline to handle the condition by tail-recursively calling *next-handler* with no arguments.[26]
init-arguments	A sequence of alternating keywords and objects which can be used as initialization arguments to construct an instance of *`type`*. It defaults to an empty sequence. This is probably used only in restart handlers.

[26]The two calling possibilities are described as tail-recursive to ensure that all values returned by the call are returned by the handler. Not returning all the values could interfere with the condition's recovery protocol. A handler that *really* knows what it is doing could use a non-tail-recursive call, but anything that knows what it's doing in this situation is probably unmodular. Note that a handler might not know the full recovery protocol, because the condition's class might be a subclass of the handler's expected type.

test and *handler* are distinct so that handler applicability can be tested without actually handling (which might take a non-local exit). One use for this is constructing a list of available restart handlers.

There is no "condition wall," i.e., when executing *handler* the set of available handlers is not reset to the handlers that were in effect when the `let handler` was entered.[27]

The expansion of a `let handler` form is implementation-dependent. Implementations are encouraged to use an expansion that optimizes establishing a handler for both speed and space, even if that increases the cost of signaling. The assumption is that most of the time a handler will never be used, because the exception it is looking for will never occur.

Miscellaneous Condition Operations

Some condition classes define accessor functions. The accessor functions for the standardized classes are specified briefly with the class above. None of these are settable.

Full Set of Operators for Signaling

The first operator listed here is a repeat of the basic operator. The others all have straightforward implementations in terms of other primitives but are provided to enable concise code in simple, common cases.

`signal` *condition* ⇒ *values* [Function]

`signal` *string* $argument_1$ $argument_2$ … ⇒ *values*

Signals the *condition*, trying each active dynamic handler, the most recent first. If all dynamic handlers decline, `signal` calls `default-handler(`*condition*`)`. If a handler returns, all the values that it returned are returned from `signal`. If `signal` returns when the condition's recovery protocol does not allow returning, some handler has violated protocol; `signal` does not check for this error. If the condition is a restart, the caller of `signal` should always assume that it might return.

The second form signals a condition of type `<simple-warning>`.

[27]This is necessary to keep restart handlers available to other condition handlers.

`error` *condition* ⇒ *{will never return}* [Function]

`error` *string* $argument_1$ $argument_2$ … ⇒ *{will never return}*

`error` is similar to `signal` but never returns; if a handler returns, `error` bombs directly to the debugger. `error` is used to make it clear that a program does not expect to receive control again after signaling a condition and might enable the compiler to generate slightly more compact code.

The second form signals a condition of type `<simple-error>`.

`cerror` *restart-description condition* ⇒ `#f` [Function]

`cerror` *restart-description string* $argument_1$ $argument_2$ … ⇒ `#f`

`cerror` is the same as `error` but first establishes a handler for `<simple-restart>`, with `init-arguments:` specifying `format-string:` *`restart-description`* and `format-arguments:` the sequence *`argument`*$_1$, *`argument`*$_2$, … .

If the restart handler is invoked, `cerror` returns `#f`; otherwise, `cerror` never returns. If `cerror` returns, the program should take the corrective actions promised in the restart-description. `cerror` is the standard way to signal correctable errors when no special class of restart condition is required.

`break` [*condition*] ⇒ `#f` [Function]

`break` [*string* $argument_1$ $argument_2$…] ⇒ `#f`

`break` obtains a condition in the same way as `signal` but then invokes the debugger immediately without signaling first. `break` establishes a `<simple-restart>` so the debugger can continue execution. This is useful for breakpoints. `break` always returns `#f`. With no arguments, a default message string is used.

`check-type` *value type* ⇒ *value* [Function]

`check-type` signals a `<type-error>` if `instance?(`*value* , *type*`)` is false.

`abort` [Function]

Performs `error(make (<abort>))`. This function is provided as a convenient shortcut.

The call is to `error`, rather than to `signal`, to guarantee that `abort` will never return.

Additional Operators for Handling

The optional `exception` clauses in Dylan's `block` statement establish exception handlers during the dynamic extent of the evaluation of the *body* of the `block` statement. Any number of *cleanup-clauses* (described under `block` in the chapter on Control Constructs) may be provided, and they may be arbitrarily interleaved with *exception-clauses* .The syntax of an exception clause is as follows:

`exception (` *type* | *name* `::` *type* #key *test init-arguments* `)`
exception-body [;]

If one of these handlers is invoked, it never declines, but immediately takes a non-local exit, executes the expressions in the *exception-body* , and returns the values of the last expression in the *exception-body*, or `#f` if the *exception-body* contains no expressions. If the exception handlers are never invoked, `block` returns the values of the last expression in the `block` *body* (or the the values of the last expression in the last *cleanup-clause*, if there are *cleanup-clauses*). Note that when the expressions in an *exception-body* are executed, the handler established by that `exception` clause is no longer active.

type, *test*, and *init-arguments* are as for `let handler`. *name*, if present, is not evaluated but is the name of a variable that is bound to the condition around the expressions in the `block` *body*.

The exception clauses are checked in the order in which they appear. That is, the first handler will take precedence over the second, etc.

A very trivial use of an exception clause might look like:

```
block (return)
  open-files();
  ...
  result
exception (<error>)
  return(#f);
cleanup
  close-files();
end block
```

default-handler *condition* ⇒ *values* [Generic Function]

`default-handler` is called if no dynamic handler handles a condition. There are predefined methods for `<condition>`, which returns an empty sequence, `<serious-condition>`, which calls the debugger,`<warning>`, which prints the message and returns `#f`, and `<restart>`, which signals an error.

Operators for Interactive Handling

restart-query *restart* [Generic Function]

`restart-query` engages the interactive user in a dialog and stores the results in slots of restart. This function is designed to be called from a handler, after making a restart and before signaling it. The debugger uses `restart-query`, for example. There is a default method for `<restart>` which does nothing.

return-query *condition* [Generic Function]

If the recovery protocol of *condition* allows returning values, this engages the interactive user in a dialog and returns the results as any number of values, which the handler should return. `return-query` should not be called if `return-allowed?` returns `#f`. If you define your own condition class whose recovery protocol allows returning values, you need to define a method for `return-query`, unless the inherited method is suitable.

Operators for Introspection

do-handlers *funarg* [Function]

`do-handlers` applies *funarg* to all dynamically active handlers, the most recently established first. *funarg* receives four arguments: *type*, *test*, *function*, and *init-arguments*. The arguments describe a dynamically active handler. All arguments have dynamic extent and must not be modified. *test* is defaulted to a function that always returns `#t`. *init-arguments* is `#f` if unsupplied by the handler.

return-allowed? *condition* ⇒ *boolean* [Generic Function]

This predicate returns `#t` if the recovery protocol of *condition* allows returning values, or `#f` if it does not. There is a default method for `<condition>` that returns `#f`. If you define your own condition class, you need to define a method for `return-allowed?` unless the inherited method is suitable.

`return-description` *condition* ⇒ *description* [Generic Function]

If the recovery protocol of this condition allows returning values, `return-description` returns a description of the meaning of returning values. This *description* can be a restart, a string, or `#f`. `return-description` should not be called if `return-allowed?` returns`#f`. If you define your own condition class whose recovery protocol allows returning values, you need to define a method for `return-description` unless the inherited method is suitable.

14. Modules

Modules are used for creating large-scale variable namespaces. Modules let you build programs and program segments that have private variables. Only the variables you export are visible from outside the module. **Module variables** are similar to global variables of other languages.

Some languages have module systems with distinct support for exporting variables, functions, types, and classes. Dylan modules operate only on variables. Because functions and classes are stored in variables, you can control access to them by controlling access to the variables that contain them. If you export the variable containing a class or function, you have effectively exported the class or function. If you do not export the variable, then you have effectively kept the class or function private.[28]

Overview

A module establishes a mapping from variable names to variables (memory locations containing values). It does this in one of two ways: a variable can be **owned** by the module, or the module may **import** variables exported by another module by **using** the other module. Modules **export** variables to make them accessible to other modules. Only exported variables can be imported by other modules.

Within a given module, a variable name refers to at most one variable. It is an error to create or import two or more different variables with the same name in a single module. If a name does refer to a variable, the variable is said to be **accessible** from the module. Each variable is owned by exactly one module, but it can be accessible from many modules.

Owned variables are created by a `create` clause in a `define module` and in some cases by definitions associated with a module.

---

[28]In the general case, reflective operations can be used to defeat module encapsulation. For example, a programmer can trace from an instance to its class by calling `object-class` on the instance, even if the implementor of the class did not export the variable containing the class. This problem can sometimes be solved by the proper use of sealing, which blocks many reflective operations.

Dylan includes two kinds of definitions. Explicit definitions are created by `define constant`, `define variable`, `define generic`, and the class name in `define class`. Implicit definitions are created by `define method` and the slot specifications of `define class`.

Definitions are used both to create variables and to provide values for variables. An explicit definition performed on a variable which was not imported creates an owned variable and provides a value for it. An explicit definition performed on a variable which was imported just provides a value for the variable. An implicit definition has the same behavior, but only if there is no explicit definition for the variable. (If there is an explicit definition for the variable, then the implicit definition does not create the variable, nor does it provide the value for it.)

There must be exactly one explicit definition for each module variable, with the exception that the explicit definition can be left out if there are one or more implicit definitions. Any module variable whose value is a generic function can have any number of implicit definitions.

Programs, module declarations, and expressions

A Dylan program is composed of expressions. Each expression is associated with a module. Within an expression, variables are referenced by variable names. The module associated with the expression provides the mapping from variable name to variable used within the expression. It is an error to reference a variable name for the purpose of getting or setting its value if the variable name does not designate either a variable locally bound with a scope that includes the reference or a variable accessible in the associated module.

Module declarations are expressions. Like other “defining forms,” module declarations are only allowed at top level or inside of `begin`. The variable names in module declarations are relative either to the module being declared or to a module that it uses, as specified in part b, and thus are not affected by the module declaration’s associated module. A module declaration can be associated with any module where `define module` has its normal meaning.

Before an expression can be compiled, the module declaration for the module associated with the expression must be compiled and then made available to the development environment in an implementation-defined way.

Module declarations

Modules are defined by `define module` forms.

`define module` *module-name* [Definition]

`[` *module-clauses* `]`

`end [ module ] [` *module-name* `]`

where each module-clause is either a **use clause**, a **create clause**, or an **export clause**. (see below)

This form defines the module with the given name. It describes which modules are used by the module being defined, which variables are imported from the used modules, and which variables are exported by the module being defined.

module-name has the same syntax as a variable name. Module names are scoped to a library. The namespace of module names is distinct from that of variables. No variable with the name *module-name* is created.

Used modules

There is one or more *use-clause* for each module used by the module being defined. Each *use-clause* has the form:

`use` *module-name* `[ ,` *module-use-options* `] ;`

module-name is the name of a module to be used. By default all exported variables from the used module are imported by the module being defined, under the same name they had in the used module.

When there are multiple use clauses using the same module, the set of imported variables is the union of those specified by all the use clauses. Some variables may be imported under more than one variable name.

Circular use relationships among modules are not allowed. The graph of the module-uses-module relation must be a directed acyclic graph.

The module-use-options are used to prevent some variables from being imported, to give some or all variables different names in the new module, and to re-export variables which were imported from a used module. Each of these options applies within the scope of the particular use clause, and does not affect the behavior of other use clauses (even if the other use clauses indicate the same module). The various options in a module-use-option may each appear no more than once.

```
import-option ::=
  import: all
  import: import-set

import-set ::=
  { imports_opt }

imports ::=
  import
  import , imports

import ::=
  variable-name
  variable-name => variable-name
```

Indicates which variables are to be imported from the module being used. The default is `all`, meaning that all the variables exported by the used module should be imported. When => appears in an import-option, it specifies both an import and a rename. In other words, "`import: {foo => bar}`" is simply an abbreviation for "`import: {foo}, rename: {foo => bar}`" and means exactly the same thing.

```
exclude-option ::=
  exclude: variable-name-set

variable-name-set ::=
  { variable-names_opt }

variable-names ::=
```

```
  variable-name
  variable-name , variable-names
```

Indicates variables which should not be imported. This keyword can only be used if `import:` is `all`. The default for the `exclude:` keyword is `{})`.

```
prefix-option ::=
  prefix: string
```

Prepends *string* to the names of variables as they are imported from the used module. This option can be overridden for a particular variable by using the `rename:` keyword for that variable. The default value for the `prefix:` keyword is `""` (the empty string).

```
rename-option ::=
  rename: { rename-specs_opt }

rename-specs ::=
  rename-spec
  rename-spec , rename-specs

rename-spec ::=
  variable-name => variable-name
```

Used to override the `import:`, `exclude:`, and `prefix:` keywords. The variables named are imported, regardless of whether the `import:` and `export:` keywords indicate they should be imported. *old-name* indicates the name of the variable in the module being used; *new-name* indicates the name to be given to the variable in the module being defined. The `prefix:` keyword of the use clause is ignored for variables specified by the `rename:` keyword. The default value for the `rename:` keyword is `{}`.

```
export-option ::=
  export: all
  export: variable-name-set
```

Specifies variables which should be exported from the module being defined. Each of these variables must have been imported by this use clause. *variable-name* is the name of the variable in the module being defined. It is also the name under which the variable will be exported. It is allowed for the same *variable-name* to appear more than once, as this is sometimes useful for documentation purposes. `all` indicates that all the variables imported by this use clause should be exported. The default value for the `export:` keyword is `{}`.

Exporting owned variables

A **module export clause** has the following syntax:

`export` *variable-names*

This option specifies that the named variables are to be exported from the module being defined. Each *variable-name* is the name of a variable to export. These variables must be defined by a defining form in the module being defined. It is an error if any of the variables were imported from other modules. It is allowed for the same name to appear more than once, since this is sometimes useful for documentation purposes.

A **module create clause** has the following syntax:

`create` *variable-names*

This option specifies that the named variables are to be created in and exported from the module being defined. A *variable-name* is the name of a variable to create and export. These variables must not be defined by a defining form in the module being defined, and they must be defined by a module which uses the module being defined. It is an error if any of the variables were imported from other modules. It is allowed for the same name to appear more than once, since this is sometimes useful for documentation purposes.

Examples

```
define module graphics
  use dylan;
  create draw-line,
         erase-line,
         invert-line,
         skew-line
         frame-rect,
         fill-rect,
         erase-rect,
         invert-rect;
end module graphics;

define module lines
```

```
  use dylan;
  use graphics,
      import: {draw-line,
               erase-line,
               invert-line,
               skew-line};
end module lines;

define module rectangles
  use dylan;
  use graphics,
      prefix: "graphics$",
      exclude: {skew-line};
end module rectangles;

define module dylan-gx
  use dylan, export: all;
  use graphics,
      rename: {skew-line => warp-line},
      export: all;
end module dylan-gx;
```

The modules created by these module declarations would have access to variables with the following names:

`graphics`	`draw-line`
	`erase-line`
	`invert-line`
	`skew-line`
	`frame-rect`
	`fill-rect`
	`erase-rect`
	`invert-rect`
	plus all the variables in the `Dylan` *module*

`lines`	`draw-line`
	`erase-line`
	`invert-line`
	`skew-line`
	plus all the variables in the `Dylan` *module*

`rectangles`	`graphics$draw-line`
	`graphics$erase-line`
	`graphics$invert-line`
	`graphics$frame-rect`

```
            graphics$fill-rect
            graphics$erase-rect
            graphics$invert-rect
```
plus all the variables in the `Dylan` *module*

```
dylan-gx    draw-line
            erase-line
            invert-line
            warp-line
            frame-rect
            fill-rect
            erase-rect
            invert-rect
```
plus all the variables in the `Dylan` *module*

The `lines` and `rectangles` modules do not export any variables. They are presumably used to provide definitions for the variables created and exported by the `graphics` modules. The difference between the `graphics` module and the `dylan-gx` module is that one variable is renamed, and the `dylan-gx` module exports the variables which it imports from the `dylan` module, while the `graphics` module does not.

15. Libraries

A library description consists of the following parts:

- A single library declaration form. It specifies a name for the library, a set of modules which are exported by the library for use by other libraries, and a set of modules that are imported from other libraries for use by the library being defined. Library declaration forms are described below.
- The association of source code with the library. The mechanism by which this association is made is implementation-defined.
- The association of executable code with the library. The mechanism by which this association is made is implementation-defined. The mechanism by which the compiler is invoked to produce the executable code is implementation-defined.
- The association of library export information with the library. The mechanism by which this association is made is implementation-defined. The mechanism by which the compiler is invoked to produce the library export information is implementation-defined. The contents of library export information is implementation-dependent, but it comprises the information required to process the source code of some other library that imports the library.

The library export information is the only part of a Dylan library that is needed to allow some other library to import it. A library that exports some modules does not have any additional declarations whose purpose is to provide information to the compiler when it is processing the code of a library that imports those modules. Rather, any such information that might be needed is obtained in some implementation defined way while processing the source expressions of the exporting library and is retained in the library export information of the exporting library.

The syntax for library declarations matches the syntax for module definitions exactly, except of course that the word `module` is replaced with the word `library`, and that library declarations do not have create clauses.

Exporting a module from a library makes all of the variables exported by the module available for import by modules in other libraries. There are two ways in which a module may be exported from a library.

(1) If the module is defined in the library, it is exported by including a *library-export-clause* that names the module in the library declaration. Each *module-name* in a *library-export-clause* is the name of a module exported by the library.

(2) If the module is imported from some other library, it is exported by including an *export-option* that specifies the module (either by name or implicitly by using "export: all") in a *library-use-option* that imported the module.

Importing a module into a library allows the module to be used by modules defined within the library. Importing a module into a library does not allow expressions in the library to be associated with the module. A module is imported into a library by including a *library-use-clause* in the library declaration that includes the module in the set of modules imported from the used library. By default, all exported modules from the used library are imported by the library being defined, under the same name they had in the used library.

When there are multiple use clauses using the same library, the set of imported modules is the union of those specified by all the *use-clauses.* Some modules may be imported under more than one module name.

The options in a *use-clause* are used to prevent some modules from being imported, to give some or all the imported modules different names in the new library, and to re-export modules which were imported from a used library. Each of these options applies within the scope of the particular *use-clause,* and does not affect the behavior of other *use-clauses.*

The meanings of the various options in a *library-use-clause* are similar to the corresponding options in a *module-use-clause,* except that it is module names in libraries that are being specified, rather than variable names in modules.

The mechanism by which the *library-name* in a *library-use-clause* is associated with the corresponding library description is implementation-defined.

Each implementation must provide a library named `dylan` which exports a module named `dylan`. That module must export exactly those variables documented as being part of the Dylan language, and the values of those variables must be as specified by the Dylan language. The `dylan` library is permitted to export additional implementation dependent modules.

Library declarations are expressions. Like other "defining forms," library declarations are only allowed at top level or inside of `begin`. The names in a library declaration are not module variable names, and thus are not affected by the library declaration's associated module. A library declaration can be associated with any module where "`define library`" has its normal meaning.

Macros can expand into library declarations. This happens during compilation of the library declaration.

Each library contains an implicitly defined module whose name is `dylan-user`. Within this module, all the variables specified by the Dylan language are accessible using their specified names. Additional implementation-dependent variables may also be accessible from this module.

Appendix A: How to get more information

Internet anonymous ftp site:

```
cambridge.apple.com:/pub/dylan/
```

This contains the latest version of this manual, the Dylan FAQ, design notes (language spec updates), public Dylan implementations, code, and papers about Dylan.

Mosaic/World-Wide Web site:

```
http://legend.gwydion.cs.cmu.edu:8001/dylan
```

Internet newsgroup:

```
comp.lang.dylan
```

The Dylan newsgroup is also available as an email list or daily digest. To subscribe, send email to `majordomo@cambridge.apple.com` containing one of these requests in the message body:

`subscribe info-dylan` *your name*

or

`subscribe info-dylan-digest` *your name*

or just send a message saying “`help`” for more information.

Dylan information on Applelink:

```
Developer Support:Developer Services:Development
Platforms:Dylan Related:
```

For inquiries about Apple’s Dylan products, send email to

```
DYLAN@applelink.apple.com
```

If you have questions or comments about the Dylan language design, or if you’re considering building your own Dylan implementation, please let us know! Email to:

```
dylan-comments@cambridge.apple.com
```

Appendix B: Dylan Syntax BNF

The grammar uses some special notation to make it more readable.

- The $_{opt}$ suffix means that the preceding item is optional.
- A trailing ellipsis (...) is used in two different ways to signal possible repetition.
    - If there is only one item on the line preceding the ellipsis, the item may appear one or more times.
    - If more than one item precedes the ellipsis, the last of these items is designated a separator; the rest may appear one or more times, with the separator appearing after each occurrence but the last. (When only one item appers, the separator does not appear.)

Lexical grammar

Lexical notes

In the lexical grammar, the various elements that come together to form a single token on the right-hand sides of rules must *not* be separated by white-space, so that the end result will be a single token. This is in contrast to the phrase grammar, where each element is already a complete token or a series of complete tokens.

Arbitrary white-space is permitted between tokens, but it is required only as necessary to separate tokens that might otherwise blend together.

Case is not significant except within character and string literals. The grammars do not reflect this, using one case or the other, but it is still true.

Comments

comment:

// …the rest of the line
/* …everything even across lines… ***/**

Tokens

token:

SYMBOL
KEYWORD
LITERAL
STRING
UNARY-OPERATOR
BINARY-OPERATOR
reserved-word
punctuation
#-word

LITERAL:

number
character-literal

punctuation:

one of **( ) , . ; [ ] { } :: - = == => #(#[? ?? ...**

#-word:

one of **#t #f #next #rest #key #all-keys**

Reserved words

reserved-word:

core-word
begin-word
intermediate-word
DEFINE-WORD

begin-word:

SIMPLE-BEGIN-WORD
EXPR-BEGIN-WORD
BEGIN-WORD

intermediate-word:

SIMPLE-INTERMEDIATE-WORD
EXPR-INTERMEDIATE-WORD
INTERMEDIATE-WORD

core-word:

one of **define end generic handler let**
one of **local method macro otherwise**

Symbols and keywords

SYMBOL*:*
  any *symbol* that's not also a *reserved-word*
  **** *operator-symbol*

KEYWORD*:*
  symbol **:**
  # STRING

symbol:
  leading-alphabetic
  leading-numeric alphabetic-character leading-alphabetic
  leading-graphic leading-alphabetic

leading-alphabetic:
  alphabetic-character
  leading-alphabetic any-character

leading-numeric:
  numeric-character
  leading-numeric any-character

leading-graphic:
  graphic-character
  leading-graphic any-character

any-character:
  alphabetic-character
  numeric-character
  graphic-character
  special-character

alphabetic-character:
  one of **a b c d e f g h i j k l m n o p q r s t u v w x y z**

numeric-character:
  one of **0 1 2 3 4 5 6 7 8 9**

graphic-character:
  one of **! & * < = > | ^ $ % @ _**

special-character:
  one of **- + ~ ? /**

Operators

UNARY-OPERATOR*:*
    unary-operator

BINARY-OPERATOR*:*
    binary-operator
    special-operator

operator-symbol:
    unary-operator
    binary-operator

unary-operator:
    one of **- ~**

binary-operator:
    one of **+ - * / ^ = == ~= < <= > >=**

special-operator:
    one of **& | :=**

Character and string literals

character-literal:
    ' *character* **'**

character:
    any printing character (including space) except for **'** or ****
    **** *escape-character*
    \ '

STRING*:*
    " *more-string*

more-string:
    string-character more-string
    "

string-character:
    any printing character (including space) except for **"** or ****
    **** *escape-character*
    \ "

escape-character:
    one of **\ a b e f n r t 0**

Numbers

number:
    integer
    ratio
    floating-point

integer:
    binary-integer
    octal-integer
    sign$_{opt}$ *decimal-integer*
    hex-integer

binary-integer:
    `#b` *binary-digit*
    binary-integer binary-digit

octal-integer:
    `#o` *octal-digit*
    octal-integer octal-digit

decimal-integer:
    decimal-digit
    decimal-integer decimal-digit

hex-integer:
    `#x` *hex-digit*
    hex-integer hex-digit

binary-digit:
    one of **`0 1`**

octal-digit:
    one of **`0 1 2 3 4 5 6 7`**

decimal-digit:
    one of **`0 1 2 3 4 5 6 7 8 9`**

hex-digit:
    one of **`0 1 2 3 4 5 6 7 8 9 A B C D E F`**

ratio:
    sign$_{opt}$ *decimal-integer* **`/`** *decimal-integer*

floating-point:
    sign$_{opt}$ *decimal-integer*$_{opt}$ **`.`** *decimal-integer exponent*$_{opt}$
    sign$_{opt}$ *decimal-integer* **`.`** *decimal-integer*$_{opt}$ *exponent*$_{opt}$
    sign$_{opt}$ *decimal-integer exponent*

exponent:
    `E` *sign*$_{opt}$ *decimal-integer*

sign:
    one of **`+ -`**

Phrase Grammar

Program structure

dylan-program:
    body$_{opt}$

body:
    constituents **;**$_{opt}$

constituents:
    constituent **;** …

constituent:
    defining-form
    local-declaration
    expression

Property lists

property-list:
    property …

property:
    , KEYWORD *value*

value:
    expression
    { *property-set*$_{opt}$ **}**

property-set:
    property-set-member **,** …

property-set-member:
    property-set-item
    property-set-item **=>** *property-set-item*

property-set-item:
    SYMBOL

Defining Forms

defining-form:

`define` *modifiers*$_{opt}$ **`method`** *method-definition*
`define` *modifiers*$_{opt}$ **`generic`** *generic-function-definition*
`define` *modifiers*$_{opt}$ DEFINE-WORD *definition*
`define` *modifiers*$_{opt}$ DEFINE-WORD *bindings*
macro-definition

modifiers:

SYMBOL …

method-definition:

SYMBOL *method-body* **`end method`**$_{opt}$ SYMBOL$_{opt}$

generic-function-definition:

SYMBOL *generic-function-body* *property-list*$_{opt}$

definition:

SYMBOL *detail-info*$_{opt}$ *item-list*$_{opt}$ **`end`** DEFINE-WORD$_{opt}$ SYMBOL$_{opt}$

item-list:

items `;`$_{opt}$

items:

item `;` …

item:

item-modifiers$_{opt}$ *item-word* *item-contents* *property-list*$_{opt}$

item-modifiers:

item-modifier …

item-modifier:

SYMBOL
DEFINE-WORD

item-word:

SYMBOL

item-contents:

variable
KEYWORD
SYMBOL `,` …

Local declarations

local-declaration:
    `let` *bindings*
    `let handler` *condition* **`=`** *handler*
    `local` *local-methods*

condition:
    type
    `(` *type property-list*$_{opt}$ **`)`**

handler:
    expression

local-methods:
    `method`$_{opt}$ *method-definition* **`,`** …

bindings:
    variable **`=`** *expression*
    `(` *variable-list* **`)`** **`=`** *expression*

variable-list:
    variables
    variables **`,`** **`#rest`** SYMBOL
    `#rest` SYMBOL

variables:
    variable **`,`** …

variable:
    SYMBOL
    SYMBOL **`::`** *type*

type:
    operand

Expressions

expressions:
expression **,** …

expression:
binary-operand BINARY-OPERATOR …

binary-operand:
KEYWORD
UNARY-OPERATOR$_{opt}$ *operand*

operand:
operand **(** *arguments*$_{opt}$ **)**
operand **[** *arguments* **]**
operand **.** SYMBOL
leaf

arguments:
KEYWORD$_{opt}$ *expression* **,** …

leaf:
literal
SYMBOL
(*expression* **)**
method *method-body* **end method**$_{opt}$
statement

literal:
LITERAL
STRING …
#t
#f
#(*constants* **.** *constant* **)**
#(*constants*$_{opt}$ **)**
#[*constants*$_{opt}$ **]**

constants:
constant **,** …

constant:
literal
KEYWORD

Statements

statement:
    begin-clause body$_{opt}$ *intermediate-clauses*$_{opt}$ *end-clause*
    begin-clause case-body **;**$_{opt}$ *end-clause*

begin-clause:
    BEGIN-WORD *detail-info*
    EXPR-BEGIN-WORD **(** *expression* **)**
    SIMPLE-BEGIN-WORD

intermediate-clauses:
    intermediate-clause body …

intermediate-clause:
    INTERMEDIATE-WORD *detail-info*
    EXPR-INTERMEDIATE-WORD **(** *expression* **)**
    SIMPLE-INTERMEDIATE-WORD

end-clause
    end *begin-word*$_{opt}$

begin-word:
    BEGIN-WORD
    EXPR-BEGIN-WORD
    SIMPLE-BEGIN-WORD

case-body:
    case-label constituents$_{opt}$ **;** *…*

case-label:
    expressions **=>**
    (*expressions* **)** **=>**
    otherwise =>$_{opt}$

detail-info:
    (*detail-list*$_{opt}$ **)**

detail-list:
    expression detail-clauses$_{opt}$ *property-list*$_{opt}$
    details property-list$_{opt}$
    details **,** EXPR-BEGIN-WORD *expression*

details:
    detail **,** *…*

detail:
    variable detail-clauses$_{opt}$
    variable **=** *expression detail-clauses*$_{opt}$

detail-clauses:
    detail-clause …

detail-clause:
    SYMBOL *expression*

Methods and generic functions

method-body:

(*parameter-list*$_{opt}$ **)** **;**$_{opt}$ *body*$_{opt}$
(*parameter-list*$_{opt}$ **)** **=>** *variable* **;** *body*$_{opt}$
(*parameter-list*$_{opt}$ **)** **=>** **(** *variable-list*$_{opt}$ **)** **;**$_{opt}$ *body*$_{opt}$

generic-function-body:

(*parameter-list*$_{opt}$ **)**
(*parameter-list*$_{opt}$ **)** **=>** *variable*
(*parameter-list*$_{opt}$ **)** **=>** **(** *variable-list*$_{opt}$ **)**

parameter-list:

parameters
parameters **,** *next-rest-key-parameter-list*
next-rest-key-parameter-list

next-rest-key-parameter-list:

#next SYMBOL
#next SYMBOL **,** *rest-key-parameter-list*
rest-key-parameter-list

rest-key-parameter-list:

#rest SYMBOL
#rest SYMBOL **,** *key-parameter-list*
key-parameter-list

key-parameter-list:

#key *keyword-parameters*$_{opt}$
#key *keyword-parameters*$_{opt}$ **,** **#all-keys**
#all-keys

parameters:

parameter **,** …

parameter:

variable
SYMBOL **==** *expression*

keyword-parameters:

keyword-parameter **,** …

keyword-parameter:

KEYWORD$_{opt}$ SYMBOL *default*$_{opt}$

default:

(*expression* **)**

Index