Harlequin Dylan

Common Dylan and Harlequin Extensions Reference

Library Reference

Version 2.0 Beta

harlequin

Copyright and Trademarks

Harlequin Dylan: Library Reference: Common Dylan and Harlequin Extensions

Version 2.0 Beta

May 1999

Part number: 3DPDTZ-15MC

http://www.harlequin.com/

Europe:

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
UK

telephone +44 1223 873 800
fax +44 1223 873 873

North America:

Harlequin Incorporated
One Cambridge Center
Cambridge, MA 02142
USA

telephone +1 617 374 2400
fax +1 617 252 6505

Asia Pacific:

Harlequin Australia Pty. Ltd.
Level 12
12 Moore Street
Canberra, ACT 2601
Australia

telephone +61 2 6206 5522
fax +61 2 6206 5525

Contents

1

Dylan Language Extensions

1.1 Introduction

The Dylan language is described in *The Dylan Reference Manual* by Andrew Shalit (Addison-Wesley, 1996). We call this book "the DRM" hereafter.

Harlequin Dylan provides an implementation of the Dylan language described by the DRM, with a few exceptions that are documented in Section 1.4 on page 7 of this chapter.

Harlequin Dylan provides the Dylan language in the **`dylan`** module of the **`dylan`** library.

As of the 2.0 release, Harlequin Dylan has joined forces with the other major implementor of Dylan, the Gwydion Dylan Development Cooperative, to provide a set of common extensions to the Dylan language (as defined by the DRM). These extensions are portable across implementations and are either built in to the **`dylan`** library or are available in a separate library, **`common-extensions`**. This chapter is an introduction to the language extensions.

1.2 Using Harlequin Dylan's language extensions

There are a number of ways to use Harlequin Dylan's language extensions in your applications.

A few extensions have become part of the **dylan** library. You can read about these extensions in Section 1.3 on page 2.

The majority of the extensions are in the **common-extensions** module of the **common-extensions** library. That library also exports a number of smaller modules that contain other basic facilities such as simplified formatting (**simple-format**), pseudo-random integer generation (**simple-random**), and object finalization (**finalization**).

Harlequin Dylan provides a convenience library, **common-dylan**, that combines the **dylan** and **common-extensions** libraries to provide a convenient "dialect" of Dylan, exported from the module **common-dylan**:

```
define library common-dylan
  use dylan, export: all;
  use common-extensions, export: all;
  export common-dylan;
end module;

define module common-dylan
  use dylan, export: all;
  use common-extensions, export: all;
end module;
```

1.3 The core of the common extensions

This section describes the common language extensions, that is, extensions made to the Dylan library as it is defined in DRM. These extensions are available to applications in the **dylan** library's **dylan** module.

All the other language extensions are described in Chapter 2, "The Common Dylan Library".

1.3.1 DEFINE FUNCTION

The **define function** definition macro provides a convenient way to define functions that have no generic properties and hence are not suitable for definition with **define generic** or **define method**. This extension has been accepted as part of the language since the DRM was published.

The **define function** macro provides a way of defining a function that says clearly to other programmers that the function is not part of any generic operation; furthermore, the function will not be extended as a generic function, and calling it need not involve any generic dispatch. Without this macro, programmers who wanted to do so would have to turn to **`define constant`**. With **`define function`**, programmer intent is more explicit and it relays more information to future maintainers of a piece of code.

The language definition of **`define function`** explicitly *does not* specify what it expands into, so that Dylan implementations have latitude to support this definer in the best way suited to the implementation.

define function *Definition macro*

Summary

Defines a constant binding in the current module and initializes it to a new function.

Macro call

```
define {adjective}* function name parameter-list
  [ body ]
end [ function ] [ name ]
```

Arguments

adjective	A Dylan unreserved-name$_{\text{bnf}}$.
name	A Dylan variable-name$_{\text{bnf}}$.
parameter-list	A Dylan parameter-list$_{\text{bnf}}$.
body	A Dylan body$_{\text{bnf}}$.

Description

Creates a constant module binding with the name *name*, and initializes it to a new function described by *parameter-list*, *options*, and any adjectives.

The adjectives permitted depend on the implementation.

The *parameter-list* describes the number and types of the function's arguments and return values. It is an error to supply **`#next`** in the parameter list, and there is no implicit **`#next`** parameter.

Operations

The following functions return the same values as they would if the function had been defined as a bare method with the same signature:

```
function-specializers
function-arguments
function-return-values
```

Calling some of the following reflective operations on a function defined with **`define function`** may be an error:

```
generic-function-methods
add-method
generic-function-mandatory-keywords
sorted-applicable-methods
find-method
remove-method
applicable-method?
```

1.3.2 Extensions to the FOR iteration construct

We have also made two extensions to the **`for`** iteration construct: a **`keyed-by`** clause and **`in`** … **`using`** clauses.

The **`keyed-by`** clause allows iteration over table elements:

```
for (my-element keyed-by my-key in my-table)
  …
end;
```

The **`in`** … **`using`** clause allows you to specify a iteration protocol other than the default (**`forward-iteration-protocol`**):

```
for (element in my-sequence using backward-iteration-protocol)
  …
end;
```

1.3.3 Weak tables

We have extended **`define table`** to incorporate *weak references* through keys and values.

A weak reference is an reference that the garbage collector treats as irrelevant to establishing whether the object referred to is live. If an object has only weak references to it, the garbage collector can delete the reference and recycle the object's memory. We call a normal reference a *strong reference*.

Weak references are a useful tool for building data structures where you do not want the garbage collector to preserve objects in the structure on account of certain references merely used to build up the structure.

Typically, this level of control is not required in a language like Dylan, which does not expose memory references to programs. But without the ability to tell the garbage collector to disregard certain kinds of reference, data structures such as tables could be bloated unnecessarily by the garbage collector preserving entries (a key / value pair) solely because the table object itself has a reference to the entry's key or value.

Common Dylan provides weakness options for instances of **`<table>`**. A table can have *weak keys* or *weak values*:

```
make(<table>, weak: #"key");     // makes a weak-key table

make(<table>, weak: #"value");   // makes a weak-value table
```

In a weak-keyed table, if a key is no longer referenced from anywhere else in the program (apart from weak references, including from the same table), then the entry (key and value) can be deleted from the table. After that, the key object will be recycled. The value will also be recycled unless it has strong references from elsewhere in the program.

Weak-valued tables are much the same, except that the focus is values and not keys. In a weak-valued table, if a value is no longer referenced from anywhere else in the program (apart from weak references, including from the same table), then the entry (value and key) can be deleted from the table. After that, the value object will be recycled. The key will also be recycled unless it has strong references from elsewhere in the program.

Weak tables are useful for implementing many sorts of cache, where the cached data is recomputable and yet both expensive to compute and also expensive to keep for a long time. For example, consider something like a font cache for an X Window System server, or a printer. Fonts might be looked up by name, so the strings would be the keys of the table. The values would be the bitmaps for the font. While the X server is using a font, the cache will be kept alive — so any further requests to select the font will find the data already present. However, if the font is not used then you would eventually expect the garbage collector to clean it out. Any future request would then have to re-load all the bitmaps.

1.3.4 Inlining adjectives for methods, constants, functions, and slots

To *inline* a value is to replace, at compile time, a reference to a variable with the value of that variable. Such inlining often allows compile-time evaluation ("constant folding") or partial evaluation.

The Harlequin Dylan compiler can perform inlining on generic function methods, constants, class slots, and functions (created with **`define function`**—see Section 1.3.1 on page 2). We have extended the Dylan language specification of **`define method`**, **`define constant`**, and class slots with inlining definition adjectives and have included those same adjectives in our language extension **`define function`**. The adjectives are:

`not-inline`	Never inline this item.
`default-inline` (default)	Inline this item within a library, at the compiler's discretion. Never inline a cross-library reference.
`may-inline`	Inline this item within or between libraries, at the compiler's discretion.
`inline`	Inline this item wherever the compiler can do so.

In addition, **`define constant`** and **`define function`** permit the adjective **`inline-only`**, which forces every reference to the constant or function to be inlined.

Note: If you export from a library any variables created with **`may-inline`**, **`inline`**, or **`inline-only`**, and then change the values of the variables, client libraries may need to be recompiled.

1.4 Language differences

1.4.1 Tables

For efficiency, Common Dylan adopts a slightly different table protocol to that described by the DRM. Hashing functions take an additional hash-state argument and merge it into the hash-state result. The function **`merge-hash-codes`** is replaced by **`merge-hash-ids`** because hash-states are merged as part of the hashing process. The constant **`$permanent-hash-state`** is no longer required; the same effect can be achieved by returning the argument **`hash-state`** unchanged as the result **`hash-state`**. Finally, **`object-hash`** has been altered to use the new protocol.

This section describes the items that have been changed. We also provide a Table-extensions module, which you can read about in Chapter 3.

table-protocol *Open generic function*

Summary	Returns functions used to implement the iteration protocol for tables.	
Signature	**`table-protocol`** ***table*** **`=>`** ***test-function hash-function***	
Arguments	*table*	An instance of **`<table>`**.
Values	*test-function*	An instance of **`<function>`**.
	hash-function	An instance of **`<function>`**.
Library	**`dylan`**	
Module	**`dylan`**	
Description	Returns the functions used to iterate over tables. These functions are in turn used to implement the other collection operations on **`<table>`**.	

The *test-function* argument is for the table test function, which is used to compare table keys. It returns true if, according to the table's equivalence predicate, the keys are members of the same equivalence class. Its signature must be:

`test-function` ***key1 key2*** **`=>`** ***boolean***

The *hash-function* argument is for the table hash function, which computes the hash code of a key. Its signature must be:

`hash-function` ***key initial-state*** **`=>`** ***id result-state***

In this signature, *initial-state* is an instance of **`<hash-state>`**. The hash function computes the hash code of *key*, using the hash function that is associated with the table's equivalence predicate. The hash code is returned as two values: an integer *id* and a hash-state *result-state*. This *result-state* is obtained by merging the *initial-state* with the hash-state that results from hashing *key*. The *result-state* may or may not be == to *initial-state*. The *initial-state* could be modified by this operation.

merge-hash-ids *Function*

Summary	Returns a hash ID created by merging two hash IDs.	
Signature	**`merge-hash-ids`** ***id1 id2*** **`#key`** ***ordered*** **`=>`** ***merged-id***	
Arguments	*id1*	An instance of **`<integer>`**.
	id2	An instance of **`<integer>`**.
	ordered	An instance of **`<boolean>`**. Default value: **`#f`**.
Values	*merged-id*	An instance of **`<integer>`**.
Description	Computes a new hash ID by merging the argument hash IDs in some implementation-dependent way. This can be used, for example, to generate a hash ID for an object by combining hash IDs of some of its parts.	

The *id1*, *id2* arguments and the return value *merged-id* are all integers.

The *ordered* argument is a boolean, and determines whether the algorithm used to the merge the IDs is permitted to be order-dependent. If false (the default), the merged result must be independent of the order in which the arguments are provided. If true, the order of the arguments matters because the algorithm used need not be either commutative or associative. It is best to provide a true value for *ordered* when possible, as this may result in a better distribution of hash IDs. However, *ordered* must only be true if that will not cause the hash function to violate the second constraint on hash functions, described on page 123 of the DRM.

object-hash *Function*

Summary

The hash function for the equivalence predicate ==.

Signature

`object-hash` ***object initial-state*** **`=>`** ***hash-id result-state***

Arguments

object	An instance of **`<integer>`**.
initial-state	An instance of **`<hash-state>`**.

Values

hash-id	An instance of **`<integer>`**.
result-state	An instance of **`<hash-state>`**.

Description

Returns a hash code for *object* that corresponds to the equivalence predicate ==.

This function is a useful 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 the result of merging the initial state with the associated hash state for the object, computed in some implementation-dependent manner.

2

The Common Dylan Library

2.1 Introduction

The Common Dylan library contains the Common Extensions library and the Dylan library. It provides a number of features that were either omitted from the Dylan language described in the DRM, or that Harlequin's Dylan developers have found useful in a broad range of situations.

The Common Dylan library exports the following modules:

`common-extensions`	Miscellaneous extensions to the Dylan language.
`harlequin-extensions`	Provided for backward compatibility.
`simple-format`	Simple formatting facilities. For more flexible formatting and printing, consider the separate Format and Print libraries.
`simple-random`	A simple facility for generating pseudo-random integers.
`finalization`	An object finalization interface.

`transcendentals`
A set of open generic functions for ANSI C-like behavior over real numbers.

`machine-words`
A set of functions for representing a limited range of integral values.

2.2 General language extensions

The **`common-extensions`** module contains a variety of useful basic extensions to the Dylan language.

The Common Dylan extensions are:

- Collection model extensions: **`<stretchy-sequence>`**, **`<string-table>`**, **`difference`**, **`fill-table!`**, **`find-element`**, **`position`**, **`remove-all-keys!`**, and **`define table`**.
- Condition system extensions: **`<format-string-condition>`**, **`<simple-condition>`**, and **`condition-to-string`**.
- Program constructs: **`iterate`** and **`when`**.
- Application development conveniences: **`iterate`**, **`debug-message`**, **`ignore`**, **`ignorable`**, **`profiling`**, **`timing`**, **`$unsupplied`**, **`unsupplied?`**, **`unsupplied`**, **`when`**, **`$unfound`**, **`one-of`**, **`unfound?`**, and **`found?`**.
- Type conversion functions: **`integer-to-string`**, **`string-to-integer`**, and **`float-to-string`**.

See “The COMMON-EXTENSIONS module” on page 19 for reference descriptions of these items.

2.3 Simple formatting and printing

Common Dylan provides several libraries relevant to formatting and printing strings, or otherwise using strings for output. These libraries include Format, Format-out, Print, and Standard-IO. The facilities provided by these libraries will be excess to many users’ requirements, who may prefer to use the **`simple-format`** module that the **`common-extensions`** library exports.

The **`format-out`** function converts its arguments into a Dylan *format string* and then sends that string to the standard output. The **`format-to-string`** function converts its arguments into a format string and then returns that format string.

See “The SIMPLE-FORMAT module” on page 49 for reference descriptions of **`format-out`** and **`format-string`**.

2.4 Simple random number generation

Common Dylan provides a simple facility for generating sequences of pseudo-random integers via the **`simple-random`** module exported from the **`common-extensions`** library.

Instances of the sealed class **`<random>`** generate pseudo-random integers. Given an instance of **`<random>`**, the function **`random`** will return a pseudo-random integer. See “The SIMPLE-RANDOM module” on page 50 for reference descriptions of **`random`** and **`<random>`**.

2.5 Finalization

Common Dylan provides a finalization interface in the **`finalization`** module of **`common-extensions`**. This section explains finalization, the finalization interface provided, and how to use the interface in applications. See “The FINALIZATION module” on page 51 for reference descriptions of the interface.

2.5.1 What is finalization?

Common Dylan’s Memory Management Reference defines finalization as follows:

> In garbage-collected languages, it is often necessary to perform actions on some objects after they are no longer in use and before their memory can be recycled. These actions are known as finalization or termination.
>
> A common use of finalization is to release a resource when the corresponding “proxy” object dies. For example, an open file might be represented by a stream object. When the stream object has no references and

> can be collected, it is certain that the file is no longer in use by the [application] and can be closed.

(See **`<URL:http://www.harlequin.com/mm/>`** for the entire Reference.)

Finalization is also commonly required when interfacing Dylan code with foreign code that does not have automatic memory management. If an interface involves a Dylan object that references a foreign object, it may be necessary to free the memory resources of the foreign object when the Dylan object is reclaimed.

2.5.2 How the finalization interface works

The following sections give a broad overview of how finalization works and how to use the interface.

2.5.2.1 Registering objects for finalization

Finalization works through cooperation with the garbage collector. Objects that are no longer referenced by the application that created them will eventually be discovered by Dylan's garbage collector and are then available to be reclaimed.

By default, the garbage collector reclaims such objects without notifying your application. If it is necessary to finalize an object before it is reclaimed, your application must inform the garbage collector.

The garbage collector maintains a register of objects requiring finalization before being reclaimed. To add an object to the register, call the function **`finalize-when-unreachable`** on the object. Objects on the register are said to be *finalizable.*

If the garbage collector discovers that a finalizable object is no longer referenced by the application, it does not reclaim it immediately. Instead, it takes the object off its finalization register, and adds it to the *finalization queue.*

The finalization queue contains all the objects awaiting finalization. The garbage collector will not reclaim the objects until they have been finalized.

2.5.2.2 Draining the finalization queue

Objects in the finalization queue wait there until the application drains it by calling the function **`drain-finalization-queue`**. This function finalizes every object in the queue.

The finalization queue is not normally drained automatically. See Section 2.5.4.1 on page 19 for details of how you can set up a thread to do so.

Note: The order in which objects in the finalization queue are finalized is not defined. Applications should not make any assumptions about finalization ordering.

2.5.2.3 Finalizers

The **`drain-finalization-queue`** function finalizes each object in the finalization queue by calling the generic function **`finalize`** on it. You should define methods for **`finalize`** on those classes whose instances may require finalization. These methods are called *finalizers*.

The recommended interface to finalization is through **`finalize-when-unreachable`** and **`drain-finalization-queue`**, but calling **`finalize`** on an object directly is also permitted. If you are certain you are finished with an object, it may be desirable to do so. For example, you might want to finalize an object created in a local binding before it goes out of scope.

Note: Finalizable objects are only removed from the register if the garbage collector discovers that they are unreachable and moves them into the finalization queue. Calling **`finalize`** on an object directly does not affect its registration status.

The **`drain-finalization-queue`** function makes each call to **`finalize`** inside whatever dynamic handler environment is present when **`drain-finalization-queue`** is called. If the call to **`drain-finalization-queue`** is aborted via a non-local exit during a call to **`finalize`**, the finalization queue retains all the objects that had been added to it but which had not been passed to **`finalize`**.

There is a default method for **`finalize`** on **`<object>`**. The method does nothing. It is available so that it is safe for all finalizers to call **`next-method`**, a practice that we strongly encourage. See Section 2.5.3.

2.5.2.4 After finalization

Once an object in the finalization queue has been finalized, it typically becomes available for reclamation by the garbage collector. Because it has been taken off the garbage collector's finalization register, it will not be queued up for finalization again.

Note: There are exceptions to this rule; see Section 2.5.2.6 on page 16 and Section 2.5.2.7 on page 17.

2.5.2.5 Upon application exit

There are no guarantees that objects which are registered for finalization will actually be finalized before the application exits. This is not a problem on many operating systems, which free any resources held by a process when it exits.

Where it is necessary to guarantee an action at the time the application exits, you should use a more explicit mechanism.

2.5.2.6 The effects of multiple registrations

Sometimes objects are registered for finalization more than once. The effects of multiple registration are defined as follows:

> Calling **`finalize-when-unreachable`** on an object *n* times causes that object to be added to the finalization queue up to *n* times, where *n* is greater than or equal to zero. There is no guarantee that the object will be added exactly *n* times.

Note that this definition so general that it does not guarantee that any object will ever be added to be finalization queue. In practice, Common Dylan's implementation guarantees that an object is added to the queue at least once whenever an object has ben determined to be unreachable by the garbage collector.

To remain robust under multiple registration, finalizers should be idempotent: that is, the effect of multiple **`finalize`** calls on an object should is the same as the effect of a single call.

2.5.2.7 The effects of resurrecting objects

If a finalizer makes an object reachable again, by storing a reference to the object in a variable, slot, or collection, we say it has *resurrected* it. An object may also be resurrected if it becomes reachable again when some other object is resurrected (because it is directly or indirectly referenced by that other object).

Resurrecting objects has pitfalls, and must be done with great care. Since finalizers typically destructively modify objects when freeing their resources, it is common for finalization to render objects unusable. We do not recommend resurrection if there is any possibility of the object being left in an unusable state, or if the object references any other objects whose transitive closure might include an object left in such a state by another call to **`finalize`**.

If you do resurrect objects, note that they will not be finalized again unless you re-register them.

2.5.2.8 The effects of finalizing objects directly

Any object that has been finalized directly, through the application itself calling **`finalize`** on it, may not yet be unreachable. Like any normal object it only becomes eligible for reclamation when it is unreachable. If such an object was also registered for finalization using **`finalize-when-unreachable`**, it can end up being finalized again via the queue mechanism.

2.5.2.9 Finalization and weak tables

If an object is both registered for finalization and is weakly referred to from a weak table, finalization occurs *first,* with weak references being removed afterwards. That is, reachability is defined in terms of strong references only, as far as finalization is concerned. Weak references die only when an object's storage is finally reclaimed.

For more on weak tables, see Section 1.3.3 on page 4.

2.5.3 Writing finalizers

Because the default **`finalize`** method, on **`<object>`**, does nothing, you must define your own **`finalize`** methods to get results from the finalization interface. This section contains useful information about writing finalizers.

2.5.3.1 Class-based finalization

If your application defines a class for which all instances require finalization, call **`finalize-when-unreachable`** in its **`initialize`** method.

2.5.3.2 Parallels with INITIALIZE methods

The default method on **`<object>`** is provided to make it safe to call **`next-method`** in all finalizers. This situation is parallel to that for class **`initialize`** methods, which call **`next-method`** before performing their own initializations. By doing so, **`initialize`** methods guarantee that the most specific initializations occur last.

By contrast, finalizers should call **`next-method`** last, in case they depend on the superclass finalizer not being run.

2.5.3.3 Simplicity and robustness

Write finalizers that are simple and robust. They might be called in any context, including within other threads; with careful design, your finalizers will work in most or all possible situations.

A finalizer might be called on the same object more than once. This could occur if the object was registered for finalization more than once, or if your application registered the object for finalization and also called **`finalize`** on it directly. To account for this, write finalizers that are idempotent: that is, the effect of multiple calls is the same as the effect of a single call. See Section 2.5.2.6 for more on the effects of multiple registrations.

Remember that the order in which the finalization queue is processed is not defined. Finalizers cannot make assumptions about ordering.

This is particularly important to note when writing finalizers for classes that are typically used to form circular or otherwise interestingly connected graphs of objects. If guarantees about finalization in graphs of objects are important, we suggest registering a root object for finalization and making its finalizer traverse the graph (in some graph-specific well-ordered fashion) and call the **`finalize`** method for each object in the graph requiring finalization.

2.5.3.4 Singleton finalizers

Do not write singleton methods on **`finalize`**. The singleton method itself would refer to the object, and hence prevent it from becoming unreachable.

2.5.4 Using finalization in applications

This section answers questions about using finalization in an application.

2.5.4.1 How can my application drain the finalization queue automatically?

If you would prefer the queue to be drained asynchronously, use the automatic finalization interface. For more details, see **`automatic-finalization-enabled?`**, page 51 and **`automatic-finalization-enabled?-setter`**, page 52.

Libraries that do not wish to depend on automatic finalization should not use those functions. They should call **`drain-finalization-queue`** synchronously at useful times, such as whenever they call **`finalize-when-unreachable`**.

Libraries that are not written to depend on automatic finalization should always behave correctly if they are used in an application that does use it.

2.5.4.2 When should my application drain the finalization queue?

If you do not use automatic finalization, drain the queue synchronously at useful points in your application, such as whenever you call **`finalize-when-unreachable`** on an object.

2.6 The COMMON-EXTENSIONS module

This section contains a reference entry for each item exported from the Common Extensions library's **`common-extensions`** module.

assert *Statement macro*

Summary	Signals an error if the expression passed to it evaluates to false.
Macro call (1)	**`assert`** ***expression format-string*** **`[`*****format-arg*****`]* =>`** ***false***
Macro call (2)	**`assert`** ***expression*** **`=>`** ***false***

Arguments

expression	A Dylan expression$_{\text{bnf}}$.
format-string	A Dylan expression$_{\text{bnf}}$.
format-arg	A Dylan expression$_{\text{bnf}}$.

Values

false	**`#f`**.

Description

Signals an error if *expression* evaluates to **`#f`**.

An assertion or “assert” is a simple tool for testing that conditions hold in program code.

The *format-string* is a format string as defined on page 112 of the DRM. If *format-string* is supplied, the error is formatted accordingly, along with any instances of *format-arg*.

If *expression* is not **`#f`**, **`assert`** does not evaluate *format-string* or any instances of *format-arg*.

See also **`debug-assert`**, page 22

<byte-character> *Sealed class*

Summary	The class of 8-bit characters that instances of **`<byte-string>`** can contain.
Superclasses	**`<character>`**
Init-keywords	None.

Description	The class of 8-bit characters that instances of **`<byte-string>`** can contain.

concatenate! *Open generic function*

Summary: A destructive version of the Dylan language's **`concatenate`**; that is, one that might modify its first argument.

Signature: **`concatenate!`** ***sequence*** **`#rest`** ***more-sequences*** **`=>`** ***result-sequence***

Arguments:

sequence	An instance of **`<sequence>`**.
more-sequences	Instances of **`<sequence>`**.

Values:

result-sequence	An instance of **`<sequence>`**.

Description: A destructive version of the Dylan language's **`concatenate`**; that is, one that might modify its first argument.

It returns the concatenation of one or more sequences, in a sequence that may or may not be freshly allocated. If *result-sequence* is freshly allocated, then, as for **`concatenate`**, it is of the type returned by **`type-for-copy`** of *sequence*.

Example

```
> define variable *x* = "great-";
"great-"

> define variable *y* = "abs";
"abs"

> concatenate! (*x*, *y*);
"great-abs"

> *x*;
"great-abs"
>
```

condition-to-string *Open generic function*

Summary
Returns a string representation of a condition object.

Signature
`condition-to-string` ***condition*** **`=>`** ***string***

Arguments

condition	An instance of **`<condition>`**.

Values

string	An instance of **`<string>`**.

Description
Returns a string representation of a general instance of **`<condition>`**. There is a method on **`<format-string-condition>`** and method on **`<type-error>`**.

debug-assert *Statement macro*

Summary
Signals an error if the expression passed to it evaluates to false — but only when the code is compiled in interactive development mode.

Macro call (1)
`debug-assert` ***expression format-string*** **`[`** ***format-arg*** **`]* =>`** ***false***

Macro call (2)
`debug-assert` ***expression*** **`=>`** ***false***

Arguments

expression	A Dylan expression$_{bnf}$.
format-string	A Dylan expression$_{bnf}$.
format-arg	A Dylan expression$_{bnf}$.

Values

false	**`#f`**.

Description
Signals an error if *expression* evaluates to false — but only when the code is compiled in debugging mode.

An assertion or “assert” is a simple and popular development tool for testing conditions in program code.

This macro is identical to **`assert`**, except that the assert is defined to take place only while debugging.

The Harlequin compiler removes debug-assertions when it compiles code in “production” mode as opposed to “debugging” mode.

The *format-string* is a format string as defined on page 112 of the DRM.

debug-message *Function*

Summary Formats a string and outputs it to the debugger.

Signature **`debug-message`** ***format-string*** **`#rest`** ***format-args*** **`=> ()`**

Arguments

	format-string	An instance of **`<string>`**.
	format-args	Instances of **`<object>`**.

Values None.

Description Formats a string and outputs it to the debugger.

The *format-string* is a format string as defined on page 112 of the DRM.

default-handler *G.f. method*

Summary Prints the message of a warning instance to the Harlequin Dylan debugger window’s messages pane.

Syntax **`default-handler`** ***warning*** **`=>`** ***false***

Arguments	*warning*	An instance of **`<warning>`**.
Values	*false*	**`#f`**.

Description

Prints the message of a warning instance to the Harlequin Dylan debugger window's messages pane. It uses **debug-message**, page 23, to do so.

This method is a required, predefined method in the Dylan language, described on page 361 of the DRM as printing the warning's message in an implementation-defined way. We document this method here because our implementation of it uses the function **debug-message**, which is defined in the Harlequin-Extensions library. Thus to use this **default-handler** method on **<warning>**, your library needs to use the Harlequin-Extensions library or a library that uses it (such as Harlequin-Dylan), rather than simply using the Dylan library.

Example

In the following code, the signalled messages appear in the Harlequin Dylan debugger window.

```
define class <my-warning> (<warning>)
end class;

define method say-hello()
  format-out("hello there!\n");
  signal("help!");
  signal(make(<my-warning>));
  format-out("goodbye\n");
end method say-hello;

say-hello();
```

The following messages appear in the debugger messages pane:

```
Application Dylan message: Warning: help!
Application Dylan message: Warning: {<my-warning>}
```

Where **{<my-warning>}** means an instance of **<my-warning>**.

See also

debug-message, page 23.

default-handler, page 361 of the DRM.

default-last-handler *Function*

Summary

Formats and outputs a Dylan condition using **`format-out`** and passes control on to the next handler.

Syntax

`default-last-handler` ***serious-condition next-handler*** **`=> ()`**

Arguments

serious-condition	A object of class **`<serious-condition>`**.
next-handler	A function.

Values

None.

Description

A handler utility function defined on objects of class **`<serious-condition>`** that can be by bound dynamically around a computation via **`let handler`** or installed globally via **`last-handler-definer`**.

This function formats and outputs the Dylan condition *serious-condition* using **`format-out`** from the Format-Out library, and passes control on to the next handler.

This function is automatically installed as the last handler if your library uses the Harlequin-Extensions library.

Example

The following form defines a dynamic handler around some body:

```
let handler <serious-condition> = default-last-handler;
```

while the following form installs a globally visible last-handler:

```
define last-handler <serious-condition>
  = default-last-handler;
```

See also

`last-handler-definer`, page 34

win32-last-handler in the *C FFI and Win32* library reference, under library **win32-user** and module **win32-default-handler**.

define table *Definition macro*

Summary
: Defines a constant binding in the current module and initializes it to a new table object.

Macro call
: **define table *name* [:: *type*] = { [*key* => *element*]* }**

Arguments

name	A Dylan name_{bnf}.
type	A Dylan $\text{operand}_{\text{bnf}}$. Default value: **<table>**.
key	A Dylan $\text{expression}_{\text{bnf}}$.
element	A Dylan $\text{expression}_{\text{bnf}}$.

Description
: Defines a constant binding *name* in the current module, and initializes it to a new table object, filled in with the keys and elements specified.

If the argument *type* is supplied, the new table created is an instance of that type. Therefore *type* must be **<table>** or a subclass thereof. If *type* is not supplied, the new table created is an instance of a concrete subclass of **<table>**.

Example

```
define table $colors :: <object-table>
  = { #"red"    => $red,
      #"green"  => $green,
      #"blue"   => $blue };
```

difference *Open generic function*

Summary
: Returns a sequence containing the elements of one sequence that are not members of a second.

Signature **difference** ***sequence***$_1$ ***sequence***$_2$ **#key** ***test*** **=>** ***result-sequence***

Arguments

sequence$_1$	An instance of **<sequence>**.
sequence$_2$	An instance of **<sequence>**.
test	An instance of **<function>**. Default value: **\==**.

Values

result-sequence	An instance of **<sequence>**.

Description Returns a sequence containing the elements of *sequence*$_1$ that are not members of *sequence*$_2$. You can supply a membership test function as *test*.

Example

```
> difference(#(1,2,3), #(2,3,4));
#(1)
>
```

false-or *Function*

Summary Returns a union type comprised of **singleton(#f)** and one or more types.

Signature **false-or** ***type*** **#rest** ***more-types*** **=>** ***result-type***

Arguments

type	An instance of **<type>**.
more-types	Instances of **<type>**.

Values

result-type	An instance of **<type>**.

Description Returns a union type comprised of **singleton(#f)**, *type*, any other types passed as *more-types*.

This function is useful for specifying slot types and function return values.

The expression

```
false-or(t1, t2, ..)
```

is type-equivalent to

```
type-union(singleton(#f), t1, t2, ..)
```

fill-table! *Function*

Summary
: Fills a table with the keys and elements supplied.

Signature
: **`fill-table!`** ***table keys-and-elements*** **`=>`** ***table***

Arguments

table	An instance of **`<table>`**.
keys-and-elements	An instance of **`<sequence>`**.

Values

table	An instance of **`<table>`**.

Description
: Modifies table so that it contains the keys and elements supplied in the sequence *keys-and-elements*.

This function interprets *keys-and-elements* as key-element pairs, that is, it treats the first element as a table key, the second as the table element corresponding to that key, and so on. The keys and elements should be suitable for *table*.

Because *keys-and-elements* is treated as a sequence of paired key-element values, it should contain an even number of elements; if it contains an odd number of elements, *fill-table!* ignores the last element (which would have been treated as a key).

find-element *Open generic function*

Summary
: Returns an element from a collection such that the element satisfies a predicate.

Signature	**`find-element`** ***collection function*** **`#key`** ***skip failure*** **`=>`** ***element***	
Arguments	*collection*	An instance of **`<collection>`**.
	predicate	An instance of **`<function>`**.
	skip	An instance of **`<integer>`**. Default value: 0.
	failure	An instance of **`<object>`**. Default value: **`#f`**.
Values	*element*	An instance of **`<object>`**.

Description

Returns a collection element that satisfies *predicate*.

This function is identical to Dylan's **`find-key`**, but it returns the element that satisfies *predicate* rather than the key that corresponds to the element.

float-to-string *Function*

Summary: Formats a floating-point number to a string.

Signature: **`float-to-string`** ***float*** **`=>`** ***string***

Arguments	*float*	An instance of **`<float>`**.
Values	*string*	An instance of **`<string>`**.

Description

Formats a floating-point number to a string. It uses scientific notation where necessary.

<format-string-condition> *Sealed instantiable class*

Summary: The class of conditions that take a format string.

Superclasses: **`<condition>`**

Init-keywords	None.
Description	The class of conditions that take a format string, as defined by the DRM. It is the superclass of Dylan's **`<simple-condition>`**.
See also	The Format library.

found? *Function*

Summary	Returns true if *object* is not equal to **`$unfound`**, and false otherwise.	
Signature	**`found?`** ***object*** **`=>`** ***boolean***	
Arguments	*object*	An instance of **`<object>`**.
Values	*boolean*	An instance of **`<boolean>`**.
Description	Returns true if *object* is not equal to **`$unfound`**, and false otherwise. It uses **`\=`** as the equivalence predicate.	

ignore *Function*

Summary	A compiler directive that tells the compiler it must not issue a warning if its argument is bound but not referenced.	
Signature	**`ignore`** ***variable*** **`=> ()`**	
Arguments	*variable*	A Dylan variable-name$_{\text{bnf}}$.
Values	None.	

Description

When the compiler encounters a variable that is bound but not referenced, it normally issues a warning. The **`ignore`** function is a compiler directive that tells the compiler it *must not* issue this warning if *variable* is bound but not referenced. The **`ignore`** function has no run-time cost.

The **`ignore`** function is useful for ignoring arguments passed to, or values returned by, a function, method, or macro. The function has the same extent as a **`let`**; that is, it applies to the smallest enclosing implicit body.

Use **`ignore`** if you never intend to reference *variable* within the extent of the **`ignore`**. The compiler will issue a warning to tell you if your program violates the **`ignore`**. If you are not concerned about the **`ignore`** being violated, and do not wish to be warned if violation occurs, use **`ignorable`** instead.

Example

This function ignores some of its arguments:

```
define method foo (x ::<integer>, #rest args)
  ignore(args);
  …
end
```

Here, we use **`ignore`** to ignore one of the values returned by **`fn`**:

```
let (x,y,z) = fn();
ignore(y);
```

See also

`ignorable`, page 31

ignorable *Function*

Summary

A compiler directive that tells the compiler it *need not* issue a warning if its argument is bound but not referenced.

Signature

`ignorable` ***variable*** **`=> ()`**

Arguments

variable	A Dylan variable-name$_{bnf}$.

Values None.

Description When the compiler encounters a variable that is bound but not referenced, it normally issues a warning. The **ignorable** function is a compiler directive that tells the compiler it *need not* issue this warning if *variable* is bound but not referenced. The **ignorable** function has no run-time cost.

The **ignorable** function is useful for ignoring arguments passed to, or values returned by, a function, method, or macro. The function has the same extent as a **let**; that is, it applies to the smallest enclosing implicit body.

The **ignorable** function is similar to **ignore**. However, unlike **ignore**, it does not issue a warning if you subsequently reference *variable* within the extent of the **ignorable** declaration. You might prefer **ignorable** to **ignore** if you are not concerned about such violations and do not wish to be warned about them.

Example This function ignores some of its arguments:

```
define method foo (x ::<integer>, #rest args)
  ignorable(args);
  …
end
```

Here, we use **ignorable** to ignore one of the values returned by **fn**:

```
let (x,y,z) = fn();
ignorable(y);
```

See also **ignore**, page 30

integer-to-string *Function*

Summary Returns a string representation of an integer.

Signature **`integer-to-string`** ***integer*** **`#key`** ***base size fill*** **`=>`** ***string***

Arguments

integer	An instance of **`<integer>`**.
base	An instance of **`<integer>`**. Default value: 10.
size	An instance of **`<integer>`** or **`#f`**. Default value: **`#f`**.
fill	An instance of **`<character>`**. Default value: 0.

Values

string	An instance of **`<byte-string>`**.

Description Returns a string representation of *integer* in the given *base*, which must be between 2 and 36. The size of the string is right-aligned to *size* if *size* is not **`#f`**, and it is filled with the *fill* character. If the string is already larger than *size* then it is not truncated.

iterate *Statement macro*

Summary Iterates over a body.

Macro call

```
iterate name ({argument [ = init-value ]}*)
  [ body ]
end [ iterate ]
```

Arguments

name	A Dylan variable-name$_{\text{bnf}}$.
argument	A Dylan variable-name$_{\text{bnf}}$.
init-value	A Dylan expression$_{\text{bnf}}$.
body	A Dylan body$_{\text{bnf}}$.

Values Zero or more instances of **`<object>`**.

Description
Defines a function that can be used to iterate over a body. It is similar to **`for`**, but allows you to control when iteration will occur.

It creates a function called *name* which will perform a single step of the iteration at a time; *body* can call *name* whenever it wants to iterate another step. The form evaluates by calling the new function with the initial values specified.

last-handler-definer *Definition macro*

Summary
Defines a "last-handler" to be used after any dynamic handlers and before calling **`default-handler`**.

Definition

```
define last-handler (condition, #key test, init-args)
  = handler;

define last-handler condition = handler;

define last-handler;
```

Arguments

condition	A Dylan expression$_{\text{bnf}}$. The class of condition for which the handler should be invoked.
test	A Dylan expression$_{\text{bnf}}$. A function of one argument called on the condition to test applicability of the handler.
init-args	A Dylan expression$_{\text{bnf}}$. A sequence of initialization arguments used to make an instance of the handler's condition class.
handler	A Dylan expression$_{\text{bnf}}$. A function of two arguments, *condition* and *next-handler*, that is called on a condition which matches the handler's condition class and test function.

Values
None.

Description

A last-handler is a global form of the dynamic handler introduced via **`let handler`**, and is defined using an identical syntax. The last handler is treated as a globally visible dynamic handler. During signalling if a last-handler has been installed then it is the last handler tested for applicability before **`default-handler`** is invoked. If a last-handler has been installed then it is also the last handler iterated over in a call to **`do-handlers`**.

The first two defining forms are equivalent to the two alternate forms of let handler. If more than one of these first defining forms is executed then the last one executed determines the installed handler. The current last-handler can be uninstalled by using the degenerate third case of the defining form, that has no condition description or handler function.

The intention is that libraries will install last handlers to provide basic runtime error handling, taking recovery actions such as quitting the application, trying to abort the current application operation, or entering a connected debugger.

Example

The following form defines a last-handler function called **`default-last-handler`** that is invoked on conditions of class **`<serious-condition>`**:

```
define last-handler <serious-condition>
  = default-last-handler;
```

See also

`one-of`, page 35

`win32-last-handler` in the *C FFI and Win32* library reference, under library **`win32-user`** and module **`win32-default-handler`**.

one-of *Function*

Summary

Returns a union type comprised of singletons formed from its arguments.

Signature **`one-of`** ***object*** **`#rest`** ***more-objects*** **`=>`** ***type***

Arguments

	object	An instance of **`<object>`**.
	more-objects	Instances of **`<object>`**.

Values

	type	An instance of **`<type>`**.

Description

Returns a union type comprised of **`singleton(`***object***`)`** and the singletons of any other objects passed with *more-object*.

`one-of(`***x*****`, `*****y*****`, `*****z*****`)`**

Is a type expression that is equivalent to

`type-union(singleton(`***x*****`), singleton(`*****y*****`), singleton(`*****z*****`))`**

position *Open generic function*

Summary

Returns the key at which a particular value occurs in a sequence.

Signature **`position`** ***sequence value*** **`#key`** ***predicate skip*** **`=>`** ***key***

Arguments

	sequence	An instance of **`<sequence>`**.
	value	An instance of **`<object>`**.
	predicate	An instance of **`<function>`**. Default value: **`\==`**.
	skip	An instance of **`<integer>`**. Default value: 0.

Values

	key	An instance of **`<object>`**.

Description

Returns the key at which *value* occurs in *sequence*.

If *predicate* is supplied, **`position`** uses it as an equivalence predicate for comparing *sequence*'s elements to *value*. It should take two objects and return a boolean. The default predicate used is **`\==`**.

The *skip* argument is interpreted as it is by Dylan's **`find-key`** function: **`position`** ignores the first *skip* elements that match *value*, and if *skip* or fewer elements satisfy *predicate*, it returns **`#f`**.

remove-all-keys! *Open generic function*

Summary
: Removes all keys in a mutable collection, leaving it empty.

Signature
: **`remove-all-keys!`** ***mutable-collection*** **`=> ()`**

Arguments
: *mutable-collection*
  An instance of **`<mutable-collection>`**.

Values
: None.

Description
: Modifies *mutable-collection* by removing all its keys and leaving it empty. There is a predefined method on **`<table>`**.

<simple-condition> *Sealed instantiable class*

Summary
: The class of simple conditions.

Superclasses
: **`<format-string-condition>`**

Init-keywords
: None.

Description
: The class of simple conditions. It is the superclass of **`<simple-error>`**, **`<simple-warning>`**, and **`<simple-restart>`**.

Operations
: **`condition-format-string`**

  `condition-format-args`

Example

<stretchy-sequence> *Open abstract class*

Summary	The class of stretchy sequences.
Superclasses	**`<sequence> <stretchy-collection>`**
Init-keywords	None.
Description	The class of stretchy sequences.

<string-table> *Sealed instantiable class*

Summary The class of tables that use strings for keys.

Superclasses **`<table>`**

Init-keywords None.

Description The class of tables that use instances of **`<string>`** for their keys. It is an error to use a key that is not an instance of **`<string>`**.

Keys are compared with the equivalence predicate **`\=`**.

The elements of the table are instances of **`<object>`**.

It is an error to modify a key once it has been used to add an element to a **`<string-table>`**. The effects of modification are not defined.

Note: This class is also exported from the **`table-extensions`** module of the **`table-extensions`** library.

string-to-integer *Function*

Summary Returns the integer represented by its string argument, or by a substring of that argument, in a number base between 2 and 36.

Signature **`string-to-integer`** ***string*** **`#key`** ***base start end default*** **`=>`** ***integer next-key***

Arguments

string	An instance of **`<byte-string>`**.
base	An instance of **`<integer>`**. Default value: 10.
start	An instance of **`<integer>`**. Default value: 0.
end	An instance of **`<integer>`**. Default value: **`sizeof(`***string***`)`**.
default	An instance of **`<integer>`**. Default value: **`$unsupplied`**.

Values

integer	An instance of **`<integer>`**.
next-key	An instance of **`<integer>`**.

Description Returns the integer represented by the characters of *string* in the number base *base*, where *base* is between 2 and 36. You can constrain the search to a substring of *string* by giving values for *start* and *end*.

This function returns the next key beyond the last character it examines.

If there is no integer contained in the specified region of the string, this function returns *default*, if specified. If you do not give a value for *default*, this function signals an error.

This function is similar to C's **`strtod`** function.

subclass *Function*

Summary	Returns a type representing a class and its subclasses.	
Signature	**subclass** ***class*** **=>** ***subclass-type***	
Arguments	*class*	An instance of **<class>**.
Values	*subclass-type*	An instance of **<type>**.

Description

Returns a type that describes all the objects representing subclasses of the given class. We term such a type a *subclass type*.

The **subclass** function is allowed to return an existing type if that type is type equivalent to the subclass type requested.

Without **subclass**, methods on generic functions (such as Dylan's standard **make** and **as**) that take types as arguments are impossible to reuse without resorting to ad hoc techniques. In the language defined by the DRM, the only mechanism available for specializing such methods is to use singleton types. A singleton type specializer used in this way, by definition, gives a method applicable to exactly one type. In particular, such methods are not applicable to subtypes of the type in question. In order to define reusable methods on generic functions like this, we need a type which allows us to express applicability to a type and all its subtypes.

For an object *O* and class *Y*, the following **instance?** relationship applies:

INSTANCE-1: instance?(*O*, subclass(*Y*))

True if and only if *O* is a class and *O* is a subclass of *Y*.

For classes *X* and *Y* the following **subtype?** relationships hold (note that a rule applies only when no preceding rule matches):

SUBTYPE-1: subtype?(subclass(*X*), subclass(*Y*))

True if and only if *X* is a subclass of *Y*.

`SUBTYPE-2: subtype?(singleton(`*`X`*`), subclass(`*`Y`*`))`

True if and only if *X* is a class and *X* is a subclass of *Y*.

`SUBTYPE-3: subtype?(subclass(`*`X`*`), singleton(`*`Y`*`))`

Always false.

`SUBTYPE-4: subtype?(subclass(`*`X`*`), `*`Y`*`)`

where *Y* is not a subclass type. True if *Y* is **`<class>`** or any proper superclass of **`<class>`** (including **`<object>`**, any implementation-defined supertypes, and unions involving any of these). There may be other implementation-defined combinations of types *X* and *Y* for which this is also true.

`SUBTYPE-5: subtype?(`*`X`*`, subclass(`*`Y`*`))`

where *X* is not a subclass type. True if *Y* is **`<object>`** or any proper supertype of **`<object>`** and *X* is a subclass of **`<class>`**.

Note that by subclass relationships **`SUBTYPE-4`** and **`SUBTYPE-5`**, we get this correspondence: **`<class>`** and **`subclass(<object>)`** are type equivalent.

Where the **`subtype?`** test has not been sufficient to determine an ordering for a method's argument position, the following further method-ordering rules apply to cases involving subclass types (note that a rule applies only when no preceding rule matches):

`SPECIFICITY+1`. **`subclass(`*`X`*`)`** precedes **`subclass(`*`Y`*`)`** when the argument is a class *C* and *X* precedes *Y* in the class precedence list of *C*.

`SPECIFICITY+2`. **`subclass(`*`X`*`)`** always precedes *Y*, *Y* not a subclass type. That is, applicable subclass types precede any other applicable class-describing specializer.

The constraints implied by sealing come by direct application of sealing rules 1–3 (see page 136of the DRM) and the following disjointness criteria for subclass types (note that a rule applies only when no preceding rule matches):

DISJOINTNESS+1. A subclass type **subclass(***X***)** and a type *Y* are disjoint if *Y* is disjoint from **<class>**, or if *Y* is a subclass of **<class>** without instance classes that are also subclasses of *X*.

DISJOINTNESS+2. Two subclass types **subclass(***X***)** and **subclass(***Y***)** are disjoint if the classes *X* and *Y* are disjoint.

DISJOINTNESS+3. A subclass type **subclass(***X***)** and a singleton type **singleton(***O***)** are disjoint unless *O* is a class and *O* is a subclass of *X*.

The guiding principle behind the semantics is that, as far as possible, methods on classes called with an instance should behave isomorphically to corresponding methods on corresponding subclass types called with the class of that instance. So, for example, given the heterarchy:

```
<object>
   |
  <A>
  / \
 <B> <C>
   \ /
   <D>
```

and methods:

```
method foo (<A>)
method foo (<B>)
method foo (<C>)
method foo (<D>)

method foo-using-type (subclass(<A>))
method foo-using-type (subclass(<B>))
method foo-using-type (subclass(<C>))
method foo-using-type (subclass(<D>))
```

that for a direct instance D_1 of **<D>**:

```
foo-using-type(<D>)
```

should behave analogously to:

```
foo(D1)
```

with respect to method selection.

Example

```
define class <A> (<object>) end;
define class <B> (<A>) end;
define class <C> (<A>) end;
define class <D> (<B>, <C>) end;

define method make (class :: subclass(<A>), #key)
  print("Making an <A>");
  next-method();
end method;

define method make (class :: subclass(<B>), #key)
  print("Making a <B>");
  next-method();
end method;

define method make (class :: subclass(<C>), #key)
  print("Making a <C>");
  next-method();
end method;

define method make (class :: subclass(<D>), #key)
  print("Making a <D>");
  next-method();
end method;

? make(<D>);
Making a <D>
Making a <B>
Making a <C>
Making an <A>
{instance of <D>}
```

supplied? *Function*

Summary

Returns true if its argument is not equal to the unique "unsupplied" value, **$unsupplied**, and false if it is.

Signature

supplied? ***object*** **=>** ***supplied?***

Arguments

object	An instance of **<object>**.

Values	*supplied?*	
		An instance of **`<boolean>`**.

Description

Returns true if *object* is not equal to the unique “unsupplied” value, **`$unsupplied`**, and false if it is. It uses **`\=`** as the equivalence predicate.

See also

`$unsupplied`, page 46

`unsupplied`, page 47

`$unsupplied`, page 46

timing *Statement macro*

Summary

Returns the time, in seconds and microseconds, spent executing the body of code it is wrapped around.

Macro call

`timing () [` ***body*** **`] end [ timing ]`**

Arguments	*body*	A Dylan body$_{\text{bnf}}$
Values	*seconds*	An instance of **`<integer>`**.
	microseconds	An instance of **`<integer>`**.

Description

Returns the time, in seconds and microseconds, spent executing the body of code it is wrapped around.

The first value returned is the number of whole seconds spent in *body*. The second value returned is the number of microseconds spent in *body* in addition to *seconds*.

Example

An example:

```
timing ()
  for (i from 0 to 200)
    format-to-string("%d %d", i, i + 1)
  end
end;
```

```
=> 1 671000
```

$unfound *Constant*

Summary
: A unique value that can be used to indicate that a search operation failed.

Type
: `<list>`

Value
: A unique value.

Description
: A unique value that can be used to indicate that a search operation failed.

See also
: **`found?`**, page 30

  `unfound?`, page 46

  `unfound`, page 45

unfound *Function*

Summary
: Returns the unique "unfound" value, **`$unfound`**.

Signature
: **`unfound () =>`** ***unfound-marker***

Arguments
: None.

Values
: *unfound-marker* The value **`$unfound`**.

Description
: Returns the unique "unfound" value, **`$unfound`**.

See also	**found?**, page 30
	unfound?, page 46
	$unfound, page 45

unfound? *Function*

Summary	Returns true if its argument is equal to the unique "unfound" value, **$unfound**, and false if it is not.
Signature	**unfound?** ***object*** **=>** ***unfound?***

Arguments	*object*	An instance of **<object>**.
Values	*unfound?*	An instance of **<boolean>**.

Description	Returns true if *object* is equal to the unique "unfound" value, **$unfound**, and false if it is not. It uses **\=** as the equivalence predicate.
See also	**found?**, page 30
	$unfound, page 45
	unfound, page 45

$unsupplied *Constant*

Summary	A unique value that can be used to indicate that a keyword was not supplied.
Type	**<list>**
Value	A unique value.

Description A unique value that can be used to indicate that a keyword was not supplied.

See also **`supplied?`**, page 43

`unsupplied`, page 47

`unsupplied?`, page 47

unsupplied *Function*

Summary Returns the unique “unsupplied” value, **`$unsupplied`**.

Signature **`unsupplied () =>`** ***unsupplied-marker***

Arguments None.

Values *unsupplied-marker*

The value **`$unsupplied`**.

Description Returns the unique “unsupplied” value, **`$unsupplied`**.

See also **`supplied?`**, page 43

`$unsupplied`, page 46

`unsupplied?`, page 47

unsupplied? *Function*

Summary Returns true if its argument is equal to the unique “unsupplied” value, **`$unsupplied`**, and false if it is not.

Signature **`unsupplied?`** ***value*** **`=>`** ***boolean***

Arguments *value* An instance of **`<object>`**.

Values	*boolean*	An instance of **`<boolean>`**.

Description

Returns true if its argument is equal to the unique "unsupplied" value, **`$unsupplied`**, and false if it is not. It uses **`\=`** as the equivalence predicate.

See also

`supplied?`, page 43

`$unsupplied`, page 46

`unsupplied`, page 47

when *Statement macro*

Summary

Executes an implicit body if a test expression is true, and does nothing if the test is false.

Macro call

`when (`*test*`) [ `*consequent*` ] end [ when ]`

Arguments	*test*	A Dylan expression$_{\text{bnf}}$.
	consequent	A Dylan body$_{\text{bnf}}$.

Values

Zero or more instances of **`<object>`**.

Description

Executes *consequent* if *test* is true, and does nothing if *test* is false.

This macro behaves identically to Dylan's standard **`if`** statement macro, except that there is no alternative flow of execution when the test is false.

Example

```
when (x < 0)
  ~ x;
end;
```

2.7 The SIMPLE-FORMAT module

This section contains a reference entry for each item exported from the Harlequin-extensions library's **`simple-format`** module.

format-out *Function*

Summary
Formats its arguments to the standard output.

Signature
`format-out` ***format-string*** **`#rest`** ***format-arguments*** **`=> ()`**

Arguments
format-string An instance of **`<byte-string>`**.

format-arguments
Instances of **`<object>`**.

Values
None.

Description
Formats its arguments to the standard output.

This function does not use the **`*standard-output*`** stream defined by the Standard-IO library.

format-to-string *Function*

Summary
Returns a formatted string constructed from its arguments.

Signature
`format-to-string` ***format-string*** **`#rest`** ***format-arguments*** **`=>`** ***string***

Arguments
format-string An instance of **`<byte-string>`**.

format-arguments
Instances of **`<object>`**.

Values
result-string An instance of **`<byte-string>`**.

Exceptions — This function signals an error if any of the format directives in *format-string* are invalid.

Description — Returns a formatted string constructed from its arguments, which include a *format-string* of formatting directives and a series of *format-arguments* to be formatted according to those directives.

The *format-string* must be a Dylan format string as described on pages 112–114 of the DRM.

2.8 The SIMPLE-RANDOM module

This section contains a reference entry for each item exported from the Harlequin-extensions library's **`simple-random`** module.

<random> — *Sealed instantiable class*

Summary — The class of random number generators.

Superclasses — **`<object>`**

Init-keywords		
	seed	An instance of **`<integer>`**. Default value: computed to be random.

Description — The class of random number generators.

The seed value from which to start the sequence of integers. Default value: computed to be random.

Example

random — *Function*

Summary — Returns a pseudorandomly generated number greater than or equal to zero and less than a specified value.

Signature	**`random`** ***upperbound*** **`#key`** ***random*** **`=>`** ***random-integer***
Arguments	*range* An instance of **`<integer>`**.
	random An instance of **`<random>`**.
Values	*random-integer* An instance of **`<integer>`**.
Description	Returns a pseudorandomly generated number greater than or equal to zero and less than *range*.

2.9 The FINALIZATION module

This section contains a reference description for each item in the finalization interface. These items are exported from the **`common-extensions`** library in a module called **`finalization`**.

automatic-finalization-enabled? *Function*

Summary	Returns true if automatic finalization is enabled, and false otherwise.
Signature	**`automatic-finalization-enabled? () =>`** ***enabled?***
Arguments	None.
Values	*enabled?* An instance of **`<boolean>`**. Default value: **`#f`**.
Description	Returns true if automatic finalization is enabled, and false otherwise.
See also	**`automatic-finalization-enabled?-setter`**, page 52
	`drain-finalization-queue`, page 52
	`finalize-when-unreachable`, page 53

finalize, page 54

automatic-finalization-enabled?-setter *Function*

Summary — Sets the automatic finalization system state.

Signature — **automatic-finalization-enabled?-setter *newval* => ()**

Arguments — *newval* — An instance of **<boolean>**.

Values — None.

Description — Sets the automatic finalization system state to *newval*.

The initial state is **#f**. If the state changes from **#f** to **#t**, a new thread is created which regularly calls **drain-finalization-queue** inside an empty dynamic environment (that is, no dynamic condition handlers). If the state changes from **#t** to **#f**, the thread exits.

See also — **automatic-finalization-enabled?**, page 51

drain-finalization-queue, page 52

finalize-when-unreachable, page 53

finalize, page 54

drain-finalization-queue *Function*

Summary — Calls **finalize** on every object in the finalization queue.

Signature — **drain-finalization-queue () => ()**

Arguments — None.

Values — None.

Description

Calls **`finalize`** on each object that is awaiting finalization.

Each call to **`finalize`** is made inside whatever dynamic handler environment is present when **`drain-finalization-queue`** is called. If the call to **`drain-finalization-queue`** is aborted via a non-local exit during a call to **`finalize`**, the finalization queue retains all the objects that had been added to it but which had not been passed to **`finalize`**.

The order in which objects in the finalization queue will be finalized is not defined. Applications should not make any assumptions about finalization ordering.

See also

`finalize-when-unreachable`, page 53

`finalize`, page 54

`automatic-finalization-enabled?`, page 51

`automatic-finalization-enabled?-setter`, page 52

finalize-when-unreachable *Function*

Summary

Registers an object for finalization.

Signature

`finalize-when-unreachable` ***object*** **`=>`** ***object***

Arguments

object	An instance of **`<object>`**.

Values

object	An instance of **`<object>`**.

Description

Registers *object* for finalization. If *object* becomes unreachable, it is added to the finalization queue rather than being immediately reclaimed.

Object waits in the finalization queue until the application calls **`drain-finalization-queue`**, which processes each object in the queue by calling the generic function **`finalize`** on it.

The function returns its argument.

See also

`finalize`, page 54

`drain-finalization-queue`, page 52

`automatic-finalization-enabled?`, page 51

`automatic-finalization-enabled?-setter`, page 52

finalize *Open generic function*

Summary

Finalizes an object.

Signature

`finalize` ***object*** **`=> ()`**

Arguments

object	An instance of **`<object>`**.

Values

None.

Description

Finalizes *object*.

You can define methods on **`finalize`** to perform class-specific finalization procedures. These methods are called *finalizers*.

A default **`finalize`** method on **`<object>`** is provided.

The main interface to finalization is the function **`drain-finalization-queue`**, which calls **`finalize`** on each object awaiting finalization. Objects join the finalization queue if they become unreachable after being registered for finalization with **`finalize-when-unreachable`**. However, you can call **`finalize`** directly if you wish.

Once finalized, *object* is available for reclamation by the garbage collector, unless finalization made it reachable again. (This is called *resurrection*; see Section 2.5.2.7 on page 17.) Because the object has been taken off the garbage collector's finalization register, it will not be added to the finalization queue again, unless it is resurrected. However, it might still appear in the queue if it was registered more than once.

Do not write singleton methods on **`finalize`**. A singleton method would itself reference the object, and hence prevent it from becoming unreachable.

See also

`finalize`, page 55.

`finalize-when-unreachable`, page 53

`drain-finalization-queue`, page 52

`automatic-finalization-enabled?`, page 51

`automatic-finalization-enabled?-setter`, page 52

finalize *G.f. method*

Summary — Finalizes an object.

Signature — **`finalize`** ***object*** **`=> ()`**

Arguments

object	An instance of **`<object>`**.

Values — None.

Description — This method is a default finalizer for all objects. It does nothing, and is provided only to make **`next-method`** calls safe for all methods on **`finalize`**.

See also

`finalize-when-unreachable`, page 53

`finalize`, page 54

`drain-finalization-queue`, page 52

`automatic-finalization-enabled?`, page 51

`automatic-finalization-enabled?-setter`, page 52

3 The Collections Library

3.1 The Collections library

The Collections library's Table Extensions module extends the Dylan language's standard table features. It is available to applications as the **`table-extensions`** module.

Note: Common Dylan provides a slightly different table implementation from that described by the DRM. See Section 1.4.1 on page 7 for details of these differences.

3.1.1 Basics

The **`table-extensions`** module exports the class **`<string-table>`**; the type **`<hash-state>`**; the generic function **`remove-all-keys!`** and two methods thereon; and the functions **`collection-hash`**, **`sequence-hash`**, **`string-hash`**, **`values-hash`**, **`case-insensitive-string-hash`**, and **`case-insensitive-equal`**.

The **`<string-table>`** class is a class of tables that use strings for keys.

The **`<hash-state>`** type implements *hash states*. A hash state is defined by the DRM, page 123, as "an implementation-dependent type that is associated with a hash id and can be used by the implementation to determine whether the hash id has been invalidated." See pages 122–123 of the DRM for more details.

The various hash functions and the **`case-insensitive-equal`** equivalence predicate are convenient building blocks for creating new table classes and hash functions.

3.1.2 Hash functions

Different hash functions are not required to return the same hash code for equal or even identical objects. For instance,

```
collection-hash(#(), object-hash, object-hash);
```

is not guaranteed to return the same values as

```
sequence-hash(#(), object-hash);
```

Furthermore, **`collection-hash`** with **`ordered: #t`** is not guaranteed to return the same hash code as **`collection-hash`** with **`ordered: #f`**. Such a requirement would render the **`ordered:`** keyword useless.

3.1.3 Weak tables

Common Dylan allows all general instances of the built-in class **`<table>`** to be *weak*. See "Weak tables" on page 4 of this volume for information about weakness.

You can create weak tables with the **`<table>`** class's **`weak:`** init-keyword. The legal values for this keyword are:

`#"key"`	Creates a table with weak keys. When there are no longer any strong references to a key, the table entry of which it is part becomes eligible for garbage collection.
`#"value"`	Creates a table with weak values. When there are no longer any strong references to a value, the table entry of which it is a part becomes eligible for garbage collection.
`#f`	Creates a table with strong keys and values. This is the default value.

3.2 The TABLE-EXTENSIONS module

This section contains a reference description for each item exported from the module **`table-extensions`**.

<string-table> *Sealed class*

Summary A table class that uses strings for keys.

Superclasses **`<table>`**

Init-keywords See Superclasses.

Description The **`<string-table>`** class is the class of tables that use instances of **`<string>`** for their keys. It is an error to use a key that is not an instance of **`<string>`**.

Keys are compared with the equivalence predicate **`\=`**.

The elements of the table are instances of **`<object>`**.

It is an error to modify a key once it has been used to add an element to a **`<string-table>`**. The effects of modification are not defined.

<hash-state> *Type*

Summary A hash state.

Supertypes

`<object>`

Init-keywords

None.

Description Anything that the Dylan Reference Manual describes as a *hash state* is an instance of this type.

Examples of hash states include the second argument and second return value of **`object-hash`**.

collection-hash *Function*

Summary Hashes the elements of a collection.

Signature **`collection-hash`** ***key-hash-function elt-hash-function collection initial-state*** **`#key`** ***ordered*** **`=>`** ***hash-id hash-state***

Arguments

key-hash-function	An instance of **`<function>`**.
elt-hash-function	An instance of **`<function>`**.
collection	An instance of **`<collection>`**.
initial-state	An instance of **`<hash-state>`**.
ordered	An instance of **`<boolean>`**. Default value: **`#f`**.

Values

hash-id	An instance of **`<integer>`**.
result-state	An instance of **`<hash-state>`**.

Description Hashes every element of *collection* using *key-hash-function* on the keys and *elt-hash-function* on the elements, and merges the resulting hash codes in order.

The *ordered* keyword is passed on to **`merge-hash-ids`**.

The functions *key-hash-function* and *elt-hash-function* must be suitable for use as hash functions. See page 123 of the DRM.

sequence-hash *Function*

Summary Hashes the elements of a sequence.

Signature **`sequence-hash`** ***elt-hash-function sequence initial-state*** **`#key`** ***ordered*** **`=>`** ***hash-id result-state***

Arguments		
	elt-hash-function	An instance of **`<function>`**.
	sequence	An instance of **`<sequence>`**.
	initial-state	An instance of **`<hash-state>`**.

Values		
	hash-id	An instance of **`<integer>`**.
	result-state	An instance of **`<hash-state>`**.

Description

Hashes every element of *sequence* using *elt-hash-function*, and merges the resulting hash codes in order.

The function *elt-hash-function* must be suitable for use as a hash function. See page 123 of the Dylan Reference Manual.

The *ordered* keyword is passed on to **`merge-hash-ids`**.

values-hash *Function*

Summary

Hashes the values passed to it.

Signature

`values-hash` ***elt-hash-function initial-state*** **`#rest`** ***arguments*** **`=>`**
hash-id result-state

Arguments		
	elt-hash-function	An instance of **`<function>`**.
	hash-state	An instance of **`<hash-state>`**.
	arguments	Instances of **`<object>`**.
	initial-state	An instance of **`<hash-state>`**.

Values		
	hash-id	An instance of **`<integer>`**.
	result-state	An instance of **`<hash-state>`**.

Description

Hashes every object in *arguments* using *elt-hash-function*, and merges the resulting hash codes in order.

The function *elt-hash-function* must be suitable for use as a hash function. See page 123 of the Dylan Reference Manual.

The *ordered* keyword is passed on to **`merge-hash-ids`**.

string-hash *Function*

Summary	Hashes a string.
Signature	**`string-hash`** ***string initial-state*** **`=>`** ***hash-id result-state***

Arguments	*string*	An instance of **`<string>`**.
	initial-state	An instance of **`<hash-state>`**.
Values	*hash-id*	An instance of **`<integer>`**.
	result-state	An instance of **`<hash-state>`**.

Description

Produces a hash code for a string, using the equivalence predicate **`\=`**.

case-insensitive-string-hash *Function*

Summary	Hashes a string, without considering case information.
Signature	**`case-insensitive-string-hash`** ***string initial-state*** **`=>`** ***hash-id result-state***

Arguments	*string*	An instance of **`<string>`**.
	initial-state	An instance of **`<hash-state>`**.
Values	*hash-id*	An instance of **`<integer>`**.
	result-state	An instance of **`<hash-state>`**.

Description

Produces a hash code for a string using the equivalence predicate **`case-insensitive-equal`**, which does not consider the case of the characters in the strings it compares.

See also **`case-insensitive-equal`**, page 63

case-insensitive-equal *Function*

Summary
Compares two strings for equality, ignoring case differences between them.

Signature
`case-insensitive-equal` ***string1 string2*** **`=>`** ***boolean***

Arguments

string1	An instance of **`<string>`**.
string2	An instance of **`<string>`**.

Values

boolean	An instance of **`<boolean>`**.

Description
Compares *string1* and *string2* for equality, ignoring any case differences between them. Returns true if they are equal and false otherwise.

The function has the same behavior as Dylan's standard method on **=** for sequences, except that when comparing alphabetical characters, it ignores any case differences.

This function is used as an equivalence predicate by **`case-insensitive-string-hash`**.

This function uses **`as-uppercase`** or **`as-lowercase`** to convert the characters in its string arguments.

Example
The **`case-insensitive-equal`** function returns true if passed the following strings:

```
"The Cat SAT ON the Mat"
```

```
"The cat sat on the Mat"
```

Conversely, the standard method on **=** returns false when passed those strings.

See also **`case-insensitive-string-hash`**, page 62

remove-all-keys! *Open generic function*

Summary	Removes all keys from a collection and leaves it empty.	
Signature	**`remove-all-keys!`** ***collection*** **`=>`** ***collection***	
Arguments	*collection*	An instance of **`<mutable-explicit-key-collection>`**.
Values	*collection*	An instance of **`<mutable-explicit-key-collection>`**.

Description

Modifies *collection* by removing all its keys and elements, and leaves it empty.

Note: To empty collections that are not instances of **`<mutable-explicit-key-collection>`**, use **`size-setter`**.

remove-all-keys! *G.f. method*

Summary	Removes all keys from a collection and leaves it empty.	
Signature	**`remove-all-keys!`** ***collection*** **`=>`** ***collection***	
Arguments	*collection*	An instance of **`<mutable-explicit-key-collection>`**.
Values	*collection*	An instance of **`<mutable-explicit-key-collection>`**.

Description

Modifies *collection* by removing all its keys and elements, and leaves it empty. This method implements the generic function by making repeated calls to **`remove-key!`**.

Note: To empty collections that are not instances of **`<mutable-explicit-key-collection>`**, use **`size-setter`**.

remove-all-keys! *Sealed g.f. method*

Summary	Removes all keys from a table and leaves it empty.	
Signature	**`remove-all-keys!`** ***table*** **`=>`** ***table***	
Arguments	*table*	An instance of **`<table>`**.
Values	*table*	An instance of **`<table>`**.

Description

Modifies *table* by removing all its keys and elements, and leaves it empty.

This method does not use **`remove-key!`**.

Note: To empty collections that are not instances of **`<mutable-explicit-key-collection>`**, use **`size-setter`**.

4 The Threads Library

4.1 Introduction

The Threads library provides a portable threads interface for Dylan. The Threads library is designed to map easily and efficiently onto the threads facilities provided by all common operating systems.

The Threads library is called **`threads`**. All documented bindings are exported from the module **`threads`**.

4.2 Multi-thread semantics

The Threads library provides multiple threads of control within a single space of objects and module variables. Each thread runs in its own independent stack. The mechanism by which the threads are scheduled is not specified, and it is not possible to determine how the execution of instructions by different threads will be interleaved. No mechanism is provided to call a function on an existing thread other than the current thread. Neither is there a mechanism to signal an exception on a thread other than the current thread.

4.2.1 Atomicity

In general, the Threads library guarantees that assignments to slots and variables are atomic. That is, after an assignment, but before synchronization, another thread will see either the old value or the new value of the location. There is no possibility of seeing a half-way state.

In some circumstances, when a slot or a variable is specialized to be of a particularly constrained type, the Threads library does not guarantee atomicity of assignments. Such a type may include a subtype of **`<double-float>`** or a subtype of **`<extended-float>`**. It may not include any other type that is either defined in the current specification of the Dylan language, or that could be created from standard facilities provided by the current specification of the language. This restriction of the atomicity guarantee is intended to permit implementations to represent the values of such slots or variables in a form which uses more space than a normal Dylan value, for optimal efficiency.

For those cases where the implementation does not provide the atomicity guarantee, the results of accessing a normal variable are undefined if:

- The read could proceed in parallel with some write of the same location
- Two writes of the same location could have proceeded in parallel since the last non-parallel write

Two memory references *proceed in parallel* if they are not explicitly sequentialized, either by being in a single thread, or by explicit inter-thread synchronization.

Programmers should guard against the possibility of undefined values by using explicit inter-thread synchronization.

4.2.2 Ordering

The ordering of visibility of side effects performed in other threads is undefined, unless explicit synchronization is used. Implementations of the library may guarantee that the visibility of side-effects performed by another thread is ordered according to the control flow of the other thread (*strong ordering*), but multi-processor implementations might not be strongly ordered. Portable code should not assume strong ordering, and should use explicit synchronization where the order of side effects is important. There is currently no library introspection facility to determine if the implementation is strongly or weakly ordered.

Because of the possibility of weak ordering, the compiler is free to assume that the effects of other threads may be ignored between explicit synchronization points, and it may perform any optimizations which preserve the semantics of a single-thread model regardless of their effects on other threads — for example, common sub-expression elimination, or changing the order of evaluation.

4.2.3 Explicit synchronization

The Threads library provides low-level synchronization functions which control the ordering of operations with respect to other threads, and control when the side effects that have been performed within one thread become visible within other threads.

At a higher level, the Threads library provides a variety of synchronization facilities, described below. These facilities include mutual-exclusion locks, semaphores and notifications. Each facility guarantees that when synchronization has been achieved, all the side effects of another thread are visible, at least up to the point where that other thread last released the synchronization facility.

An appropriate synchronization must be used to guard side-effects on state if there is any possibility of those side-effects either being corrupted by another thread or corrupting another thread. For example, a function which assigns to two slots of an object may require the use of a lock to guarantee that other threads never observe the object in a partly updated state.

It is up to library designers to document when synchronization is not performed internally, and when synchronization protocols must be used by clients. The implications for the Dylan library, and some other low-level libraries, are discussed in Section 4.3 on page 72.

4.2.4 Conditional update

In addition to the synchronization primitives, the library provides a conditional update mechanism which is not synchronized, but which tests whether the value in a variable or slot has changed and atomically updates it if not.

By using conditional updates, a thread can confirm (or deny) that there has been no interference from other threads, without any need for a blocking operation. This is more efficient for those circumstances where interference is not disastrous and it is possible to recompute the update.

For example, a function which increments the value of a variable might use a conditional update to store the new value into place, in order to guarantee a numeric sequence for the variable. In this example, the function might loop until the conditional update has succeeded.

It is possible to achieve synchronization by looping until a conditional update is successful, and then synchronizing side effects. This is not recommended, because the busy-waiting state during the loop may disallow other threads from running. Normally, conditional update should be used only when it is expected to succeed. If it is likely that the conditional update might fail multiple times around the loop, then either the number of times around the loop should be limited, or a blocking function from the Threads library should be used within the loop.

4.2.5 The dynamic environment

Dylan has an implicit notion of a *dynamic environment*, corresponding to language constructs with *dynamic extent*. For example, the **`block`** construct can introduce *cleanup-clauses*, and the *body* of the block is executed in a dynamic environment in which those cleanup-clauses are active. *Handlers* and *exit procedures* are other examples of language features related to the dynamic environment.

The dynamic environment is defined to be thread-local. When a new thread is created, it starts with a fresh dynamic environment. It is an error to attempt to use a handler or a non-local exit function belonging to another thread. It is impossible to use an unwind-protect cleanup from another thread.

Although the binding of condition handlers only affects the dynamic environment of the current thread, unhandled conditions are passed to the global generic function **`default-handler`**. This function might *call the debugger*. The Threads library does not define what calling the debugger means.

Note that in Dylan, unlike in C and C++, *lexical* variables (that is local, or **`let`**-bound variables) have indefinite extent — that is, have a lifetime independent of the function or block in which they were created — and are not bound in the dynamic environment. Because those variables are in general potentially global, you may need to explicitly synchronize accesses to them.

4.2.6 Thread variables

The Threads library provides a new type of variable: a *thread* variable, also known as a *thread-local* variable. These variables are similar to normal module variables in the sense that they are visible according to the same scoping rules and have the same semantics in a single-threaded program. However, in contrast to a normal variable, assignments to a thread variable in one thread are not visible when evaluating the variable in another thread.

Whenever a thread is created, the value of each thread variable is initialized to a thread-independent value resulting from a once-only evaluation of the initialization expression of the thread variable definition.

See page 103 for details of the **`thread`** adjective to **`define variable`**.

4.2.7 Dynamic binding

The Threads library exports a macro for dynamic binding. A *binding* is a mapping between a variable and a *value-cell* which holds the variable's value. A *dynamic* binding is a binding which has dynamic extent, and shadows any outermost bindings. Dynamic bindings can be considered to be a property of the dynamic environment.

Thread variables can have new dynamic bindings created for them with the macro **`dynamic-bind`**, page 104. Thread variables inherently have thread-local bindings, so it is possible to re-bind a thread variable dynamically using the Dylan construct **`block`** … **`cleanup`**. The **`dynamic-bind`** macro can be implemented in this way.

The thread-local nature of dynamically bindable variables may not be optimal for all problem domains. For instance a shared, global, outermost binding may be desirable, or alternatively, a thread may want to inherit current bindings from the parent thread at creation time, giving a "fork"-type model of state inheritance. These alternatives are not pursued in this library, but they might be an interesting area for future research.

4.3 Thread safety in client libraries

If an application uses multiple threads, then there may be thread safety requirements for any library that can be called simultaneously by multiple threads, even if the called library does not use the Threads library directly.

This section is about thread safety in any library that is designed to be used in a multi-threaded application.

4.3.1 General requirements

A library's designer is responsible for documenting which features of the library offer built-in synchronization and which do not. While there is no definitive rule that can assist designers in this documentation, the following guidelines may be useful.

If a client of the library forgets to use a synchronization feature when one is necessary, the library designer should ensure that the effect of the lack of synchronization is limited to a small unit — probably a single object. In cases where the designer cannot guarantee that the effect will be limited, the library should either implement the synchronization internally, or provide a macro for clients to use instead.

Library implementors must ensure that the library provides implicit synchronization for any hidden global state which is maintained by the library. Library designers may choose whether the library should offer implicit synchronization of the state of objects managed by the library. The interface is more convenient if the synchronization is implicit, but it may be more efficient to rely on explicit synchronization by the client. Library designers should always document the choice they make.

4.3.2 Effects on the Dylan library

The definition of the Dylan library is not changed with the addition of the Threads library. The implementation ensures that all hidden global state (such as the symbol table and any generic function caches) is implicitly synchronized. Those functions in the Dylan library which are defined to modify the state of objects are not defined to provide implicit synchronization. However, implementations are expected to ensure that synchronization bugs in Dylan programs will not cause obscure errors that cannot be explained in terms of the semantics of Dylan language constructs.

The library guarantees that **`element`** and **`element-setter`** will be atomic for all of Dylan's non-stretchy built-in collection classes, and for **`<table>`**, except for subclasses of **`<string>`**, and limited collections where the elements are constrained to be either of a type for which slots and variables do not guarantee atomicity (see Section 4.2.1 on page 67) or a subtype of **`<character>`**, or of a proper subtype of **`<integer>`**. This design is intended to permit implementations to use efficient representations for element values, which use either more or less space than a normal Dylan value. It is undefined whether any of the other standard Dylan functions are atomic. Where atomicity is not guaranteed, clients should guard against unexpected behavior by using explicit synchronization, as appropriate.

4.4 The Threads class hierarchy

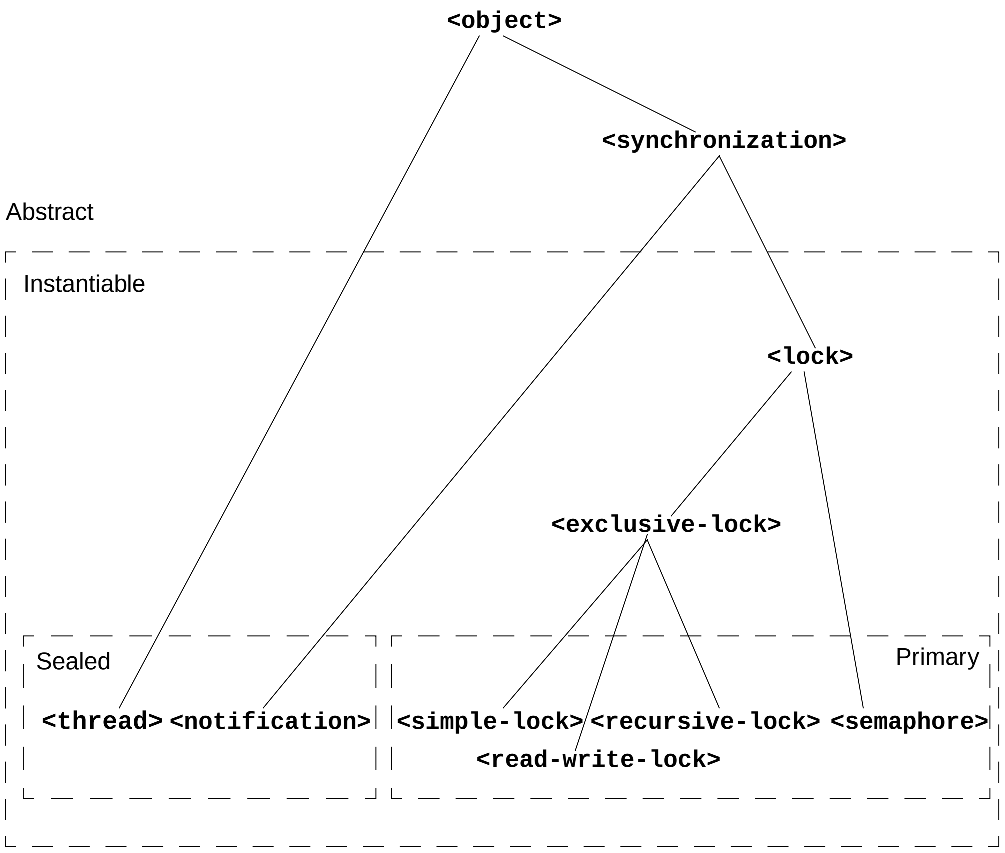

Figure 4.1 Threads class hierarchy.

4.5 Basic features

This section documents basic features of the Threads library: operations on threads and low-level synchronization.

4.5.1 Low-level synchronization

sequence-point *Function*

Summary
: Tells the compiler that it must consider the possibility of visible side effects from other threads at the point of the call.

Signature
: **`sequence-point () => ()`**

Arguments
: None.

Values
: None.

Description
: Tells the compiler that it must consider the possibility of visible side effects from other threads at the point of the call.

Normally, the compiler is not obliged to consider this possibility, and is free to rearrange program order provided that the reordering cannot be detected within a thread.

Calling this function effectively prohibits the compiler from rearranging the order of reads or writes from or to global data, relative to the call. This function may disallow compiler optimizations, leading to less efficient code — even for strongly ordered machines.

synchronize-side-effects *Function*

Summary
: As **`sequence-point`**, with the addition that all side effects that have been performed within the calling thread are made visible within all other threads.

Signature
: **`synchronize-side-effects () => ()`**

Arguments
: None.

Values

None.

Description

A call to this function implies all the constraints to the compiler of a call to **`sequence-point`**. In addition it ensures that all side effects that have been performed within the calling thread are made visible within all other threads. Hence, no side effect performed after the call can be visible to other threads before side effects performed before the call. On a strongly ordered machine, this function might legitimately be performed as a null operation.

Some of the standard synchronization functions in the Threads library also ensure the visibility of side effects and act as sequence points, as if by a call to this function. This is defined to happen as follows:

- Immediately before a thread exits and becomes available for joining with **`join-thread`**
- Before **`thread-yield`** yields control
- After **`wait-for`** achieves synchronization (for all methods provided by the Threads library)
- Upon entry to **`release`** (for all methods provided by the Threads library)
- Upon entry to **`release-all`**

Example

This example uses low-level synchronization to implement a class for performing lazy evaluation in a thread-safe manner, without the need for locks.

The class guarantees that the value will not be computed until it is needed, although it does not guarantee that it will not be computed more than once concurrently. This might be useful for memorization purposes.

The class uses 3 slots: one for a function which may be used to compute the value, one for a boolean indicating whether the value is already known, and one for the value itself, if known.

It is essential that no instance can ever be observed in a state where the boolean indicates a known value before the value is present. The low-level synchronization functions ensure this cannot happen.

```
define class <lazy-value> (<object>)
  slot thunk :: <function>,
       required-init-keyword: thunk:;
  slot internal-guard :: <boolean> = #t;
  slot computed-value;
end class;

define method lazy-value (lv :: <lazy-value>)
    => (value)
  if (lv.internal-guard)

      // Don’t yet have a value -- so compute it now;
      let value = lv.thunk();

      // Store the value in place
      lv.computed-value := value;

      // Before droppping the guard, synchronize side
      // effects to ensure there is no possibility that
      // other threads might see the lowered guard
      // before seeing the value

      synchronize-side-effects();

      // Now we can drop the guard to permit other
      // threads to use this value

      lv.internal-guard := #f;

      // Finally, return the computed value

      value

  else  // The value has already been computed and
        // stored, so use it

    // First, need a sequence-point to force the
    // compiler not to move the read of the
    // computed-value so that it is performed BEFORE
    // the read of the guard.

    sequence-point();

    lv.computed-value;
  end if;
end method;
```

4.5.2 Operations on threads

<thread> *Sealed instantiable class*

Summary — The class of threads.

Superclasses — **`<object>`**

Init-keywords — *function* An in

stance of **`<function>`**. Required.

priority	A signed integer.
name	An instance of **`<string>`**.

Description — The class representing a thread of control executing *function*.

The *function* is called with no arguments in the empty dynamic environment of the new thread. The thread terminates when the function returns.

The function is executable immediately. You can suspend a new thread (almost) immediately on creation by arranging for it to synchronize on an unavailable resource upon entry to the function.

The optional *priority* keyword provides a scheduling priority for the thread. The higher the value, the greater the priority. The default value is zero, which is also the value of the constant **`$normal-priority`**, one of several constants that correspond to useful priority levels. The library offers no way to change the priority of a thread dynamically.

The following constants, listed in order of increasing value, may be useful as values for the optional *priority* keyword.

```
$low-priority
$background-priority
$normal-priority
$interactive-priority
$high-priority
```

The *name* keyword is a string that is used as the function's name for convenience purposes, such as debugging.

Operations

The class **`<thread>`** provides the following operations:

`thread-name`	Returns the name of a thread, or **`#f`** if no name was supplied.
`join-thread`	Blocks until one of the specified threads has terminated, and returns the values of its function.

thread-name *Function*

Summary

Returns the name of a thread.

Signature

`thread-name` ***thread*** **`=>`** ***name-or-false***

Arguments

thread	An instance of **`<thread>`**.

Values

name-or-false	An instance of **`type-union(<string>, singleton(#f))`**.

Description

Returns the name of *thread* as a string. If *thread* does not have a name, this function returns **`#f`**.

join-thread *Function*

Summary

Waits for another, existing, thread to terminate, and then returns the values of its function.

Signature	**join-thread** ***thread*** **#rest** ***threads*** **=>** ***thread-joined*** **#rest** ***results***	
Arguments	*thread*	An instance of **<thread>**. A thread to join.
	threads	Instances of **<thread>**. More threads to join.
Values	*thread-joined*	An instance of **<thread>**. The thread that was joined.
	results	Zero or more instances of **<object>**. The values returned from the thread that was joined.

Exceptions

An implementation of **join-thread** is permitted to signal the following condition:

<duplicate-join-error>

A condition of this class (a subclass of **<error>**) may be signalled when a thread is passed to **join-thread**, if that thread has already been joined by an earlier call to **join-thread**, or if that thread is currently active in another call to **join-thread**.

Description

Waits for another, existing, thread to terminate, by blocking if necessary, and then returns the values of its function. The function returns the thread object that was joined, along with any values its function returns.

If more than one thread is passed to **join-thread**, the current thread blocks until the first of those threads terminates. The values returned are those of the first thread to terminate.

If one or more of the multiple threads has already terminated at the time of the call, then one of those terminated threads is joined. When more than one thread has already terminated, it is undefined which of those threads the implementation will join.

It is an error to pass a thread to **`join-thread`** if it has already been joined in a previous call to **`join-thread`**. It is an error to pass a thread to **`join-thread`** if that thread is also being processed by another simultaneous call to **`join-thread`** from another thread.

thread-yield *Function*

Summary
: Force the current thread to yield control to the part of the implementation responsible for scheduling threads.

Signature
: **`thread-yield () => ()`**

Description
: Forces the current thread to yield control to the part of the implementation responsible for scheduling threads. Doing so may have the effect of allowing other threads to run, and may be essential to avoid deadlock in a co-operative scheduling environment.

current-thread *Function*

Summary
: Returns the current thread.

Signature
: **`current-thread () =>`** ***thread***

Arguments
: None.

Values
: *thread* An instance of **`<thread>`**.

Description
: Returns the current thread.

4.6 Synchronization protocol

4.6.1 Basic features

<synchronization> *Open abstract class*

Summary
: The class of objects that are used for inter-thread synchronization.

Superclasses
: **`<object>`**

Init-keywords

`name:`	An instance of **`<string>`**.

Description
: The class of objects that are used for inter-thread synchronization.

There is no explicit mechanism in the library to block on a number of synchronization objects simultaneously, until synchronization can be achieved with one of them. This mechanism can be implemented by creating a new thread to wait for each synchronization object, and arranging for each thread to release a notification once synchronization has been achieved.

The *name* keyword is a string that is used as the synchronization object's name for convenience purposes, such as debugging.

Operations
: The class **`<synchronization>`** provides the following operations:

`wait-for`	Block until synchronization can be achieved.
`release`	Release the object to make it available for synchronization.

`synchronization-name`	Returns the name of the synchronization object.

wait-for *Open generic function*

Summary
: Blocks until a synchronization object is available.

Signature
: **`wait-for`** ***object*** **`#key`** ***timeout*** **`=>`** ***success***

Arguments

object	An instance of **`<synchronization>`**.
timeout	Time-out interval. If the value is **`#f`** (the default), the time-out interval never elapses. Otherwise the value should be a **`<real>`**, corresponding to the desired interval in seconds.

Values

success	An instance of **`<boolean>`**.

Description
: Blocks until a synchronization object is available.

This function is the basic blocking primitive of the Threads library. It blocks until *object* is available and synchronization can be achieved, or the *timeout* interval has expired. A non-blocking synchronization may be attempted by specifying a *timeout* of zero. Individual methods may adjust the state of the synchronization object on synchronization. The function returns **`#t`** if synchronization is achieved before the timeout interval elapses; otherwise it returns **`#f.`**

release *Open generic function*

Summary
: Releases a synchronization object.

Signature
: **`release`** ***object*** **`#key => ()`**

Arguments	*object*	An instance of **`<synchronization>`**.
Values	None.	

Description: Releases the supplied synchronization object, *object*, potentially making it available to other threads. Individual methods describe what this means for each class of synchronization. This function does not block for any of the subclasses of **`<synchronization>`** provided by the library.

synchronization-name *Open generic function*

Summary: Returns the name of a synchronization object.

Signature: **`synchronization-name`** ***object*** **`=>`** ***name-or-false***

Arguments	*object*	An instance of **`<synchronization>`**.
Values	*name-or-false*	An instance of **`type-union(<string>, singleton(#f))`**.

Description: Returns the name of the synchronization object, *object*, if it was created with the *name* init-keyword. Otherwise **`#f`** is returned.

4.6.2 Locks

<lock> *Open abstract instantiable class*

Summary: The class of locks.

Superclasses: **`<synchronization>`**

Description	Locks are synchronization objects which change state when they are *claimed* (using **`wait-for`**), and revert state when *released* (using **`release`**).

It is normally necessary for programs to ensure that locks are released, otherwise there is the possibility of *deadlock*. Locks may be used to restrict the access of other threads to shared resources between the synchronization and the release. It is common for a protected operation to be performed by a body of code which is evaluated in a single thread between synchronization and release. A macro **`with-lock`** is provided for this purpose. When a thread uses a lock for *mutual-exclusion* in this way, the thread is said to *own the lock*.

`<lock>` has no direct instances; calling **`make`** on **`<lock>`** returns an instance of **`<simple-lock>`**.

Operations

The class **`<lock>`** provides the following operations:

`with-lock`	Execute a body of code between **`wait-for`** and **`release`** operations.

with-lock *Statement macro*

Summary

Holds a lock while executing a body of code.

Macro call

```
with-lock (lock, #key keys)
  body
[failure failure-expr]
end
```

Arguments

lock	An instance of **`<lock>`**.
keys	Zero or more of the keywords provided by **`wait-for`**, page 83.
body	A body of Dylan code.

Values

values	Zero or more instances of **`<object>`**.

Exceptions

`with-lock` may signal a condition of the following class (a subclass of **`<serious-condition>`**):

`<timeout-expired>`

This is signalled when **`with-lock`** did not succeed in claiming the lock within the timeout period.

Description

Execute the *body* with *lock* held. If a **`failure`** clause is supplied, then it will be evaluated and its values returned from **`with-lock`** if the lock cannot be claimed (because a timeout occurred). The default, if no **`failure`** clause is supplied, is to signal an exception of class **`<timeout-expired>`**. If there is no failure, **`with-lock`** returns the results of evaluating the body.

Example

If no **`failure`** clause is supplied, the macro expands into code equivalent to the following:

```
let the-lock = lock;

if (wait-for(the-lock, keys ...))
  block ()
    body ...
  cleanup
    release(the-lock)
  end block
else
  signal(make(<timeout-expired>,
              synchronization: the-lock)
end if
```

4.6.3 Semaphores

<semaphore> *Open instantiable primary class*

Summary

The class of traditional counting semaphores.

Superclasses

`<lock>`

Description

The **`<semaphore>`** class is a class representing a traditional counting semaphore. An instance of **`<semaphore>`** contains a counter in its internal state. Calling **`release`** on a semaphore increments the internal count. Calling **`wait-for`** on a semaphore decrements the internal count, unless it is zero, in which case the thread blocks until another thread releases the semaphore.

Semaphores are less efficient than exclusive locks, but they have asynchronous properties which may be useful (for example for managing queues or pools of shared resources). Semaphores may be released by any thread, so there is no built-in concept of a thread owning a semaphore. It is not necessary for a thread to release a semaphore after waiting for it — although semaphores may be used as locks if they do.

Init-keywords

initial-count	A non-negative integer, corresponding to the initial state of the internal counter. The default value is 0.
maximum-count	A non-negative integer corresponding to the maximum permitted value of the internal counter. The default value is the largest value supported by the implementation, which is the value of the constant **`$semaphore-maximum-count-limit`**. This constant will not be smaller than 10000.

wait-for

Sealed method

Summary

Claims a semaphore object.

Signature

`wait-for` ***object*** **`#key`** ***timeout*** **`=>`** ***success***

Arguments

object	An instance of **`<semaphore>`**. The semaphore object to wait for.

	timeout	Time-out interval. If the value is **`#f`** (the default), the time-out interval never elapses. Otherwise the value should be a **`<real>`**, corresponding to the desired interval in seconds.
Values	*success*	An instance of **`<boolean>`**.

Description — Decrements the internal count of the semaphore object, blocking if the count is zero.

See also — **`wait-for`**, page 83.

release *Sealed method*

Summary — Releases a semaphore object.

Signature — **`release`** ***object*** **`#key => ()`**

Arguments	*object*	An instance of **`<semaphore>`**.

Values — None.

Exceptions — An implementation of this **`release`** method is permitted to signal a condition of the following class, which is a subclass of **`<error>`**:

`<count-exceeded-error>`

This may be signalled when an attempt is made to release a **`<semaphore>`** when the internal counter is already at its maximum count.

Description — Releases a semaphore object, by incrementing its internal count.

See also **`release`**, page 83.

4.6.4 Exclusive locks

<exclusive-lock> *Open abstract instantiable class*

Summary The class of locks which prohibit unlocking by threads that do not own the lock.

Superclasses **`<lock>`**

Description The class of locks which prohibit unlocking by threads that do not own the lock.

The notion of ownership is directly supported by the class, and a thread can test whether an **`<exclusive-lock>`** is currently owned. An instance of **`<exclusive-lock>`** can only be owned by one thread at a time, by calling **`wait-for`** on the lock.

Once owned, any attempt by any other thread to wait for the lock will cause that thread to block. It is an error for a thread to release an **`<exclusive-lock>`** if another thread owns it.

`<exclusive-lock>` has no direct instances; calling **`make`** on **`<exclusive-lock>`** returns an instance of **`<simple-lock>`**.

Operations The class **`<exclusive-lock>`** provides the following operations:

owned?	Tests to see if the lock has been claimed by the current thread.

release *Protocol*

Summary Releases an exclusive lock.

Signature	**`release`** ***object*** **`#key => ()`**
Arguments	*object* An instance of **`<exclusive-lock>`**.
Values	None.

Exceptions

Implementations of **`release`** methods for subclasses of **`<exclusive-lock>`** are permitted to signal a condition of the following class, which is a subclass of **`<error>`**:

`<not-owned-error>`

This may be signalled when an attempt is made to release an **`<exclusive-lock>`** when the lock is not owned by the current thread.

Description

Releases a lock that is owned by the calling thread. It is an error if the lock is not owned.

The Threads library does not provide a method on **`release`** for **`<exclusive-lock>`**, which is an open abstract class. Each concrete subclass will have an applicable method which may signal errors according to the protocol described above.

owned? *Open generic function*

Summary

Tests whether an exclusive lock has been claimed by the current thread.

Signature	**`owned?`** ***object*** **`=>`** ***owned?***	
Arguments	*object*	An instance of **`<exclusive-lock>`**.
Values	*owned?*	An instance of **`<boolean>`**.

Description

Tests whether the exclusive lock has been claimed by the current thread.

4.6.5 Recursive locks

<recursive-lock> *Open instantiable primary class*

Summary	The class of locks that can be locked recursively.
Superclasses	**`<exclusive-lock>`**
Description	A thread can lock a **`<recursive-lock>`** multiple times, recursively, but the lock must later be released the same number of times. The lock will be freed on the last of these releases.

wait-for *Sealed method*

Summary: Claims a recursive lock.

Signature: **`wait-for`** ***object*** **`#key`** ***timeout*** **`=>`** ***success***

Arguments	*object*	An instance of **`<recursive-lock>`**.
	timeout	Time-out interval. If the value is **`#f`** (the default), the time-out interval never elapses. Otherwise the value should be a **`<real>`**, corresponding to the desired interval in seconds.
Values	*success*	An instance of **`<boolean>`**.

Description: Claims a recursive lock, blocking if it is owned by another thread.

See also: **`wait-for`**, page 83.

release *Sealed method*

Summary	Releases a recursive lock.	
Signature	**`release`** ***object*** **`#key => ()`**	
Arguments	*object*	An instance of **`<recursive-lock>`**.
Values	None.	
Description	Releases a recursive lock, and makes it available if it has been released as many times as it was claimed with **`wait-for`**.	

owned? *Sealed method*

Summary	Tests whether a recursive lock has been claimed by the current thread.	
Signature	**`owned?`** ***object*** **`=>`** ***owned?***	
Arguments	*object*	An instance of **`<recursive-lock>`**.
Values	*owned?*	An instance of **`<boolean>`**.
Description	Tests whether a recursive lock has been claimed by the current thread.	

4.6.6 Simple locks

<simple-lock> *Open instantiable primary class*

Summary	A simple and efficient lock.
Superclasses	**`<exclusive-lock>`**

Description — The **`<simple-lock>`** class represents the most simple and efficient mutual exclusion synchronization primitive. It is an error to lock a **`<simple-lock>`** recursively. An attempt to do so might result in an error being signalled, or deadlock occurring.

wait-for *Sealed method*

Summary — Claims a simple lock.

Signature — **`wait-for`** ***object*** **`#key`** ***timeout*** **`=>`** ***success***

Arguments

object	An instance of **`<simple-lock>`**.
timeout	Time-out interval. If the value is **`#f`** (the default), the time-out interval never elapses. Otherwise the value should be a **`<real>`**, corresponding to the desired interval in seconds.

Values

success	An instance of **`<boolean>`**.

Description — Claims a simple lock, blocking if it is owned by another thread.

See also — **`wait-for`**, page 83.

release *Sealed method*

Summary — Releases a simple lock.

Signature — **`release`** ***object*** **`#key => ()`**

Arguments

object	An instance of **`<simple-lock>`**.

Values	None.	
Description	Releases a simple lock.	
See also	**`release`**, page 83.	

owned? *Sealed method*

Summary	Tests whether a simple lock has been claimed by the current thread.	
Signature	**`owned?`** ***object*** **`=>`** ***owned?***	
Arguments	*object*	An instance of **`<simple-lock>`**.
Values	*owned?*	An instance of **`<boolean>`**.
Description	Tests whether a simple lock has been claimed by the current thread.	

4.6.7 Multiple reader / single writer locks

<read-write-lock> *Open instantiable primary class*

Summary

The class of locks that can have multiple readers but only one writer.

Superclasses

`<exclusive-lock>`

Description

The class of locks that can have multiple readers but only one writer.

The **`<read-write-lock>`** class can be locked in either of two modes, *read* and *write*. A write lock is exclusive, and implies ownership of the lock. However, a read lock is non-exclusive, and an instance can be locked multiple times in read mode, whether by multiple threads, recursively by a single thread, or a combination of both.

A **`<read-write-lock>`** can only be locked in write mode if the lock is free, and the operation will block if necessary. It can only be freed by the thread that owns it.

A **`<read-write-lock>`** can be locked in read mode provided that it is not owned with a write lock. The operation will block while the lock is owned. Each time it is locked in read mode, an internal counter is incremented. This counter is decremented each time a read-mode lock is released. The lock is freed when the counter becomes zero.

The **`<read-write-lock>`** class is less efficient than the other lock classes defined in the Threads library. However, it provides an efficient and convenient means to protect data that is frequently read and may occasionally be written by multiple concurrent threads.

wait-for

Sealed method

Summary

Claims a read-write lock.

Signature

`wait-for` ***object*** **`#key`** ***timeout mode***

Arguments

object	An instance of **`<read-write-lock>`**.
timeout	Time-out interval. If the value is **`#f`** (the default), the time-out interval never elapses. Otherwise the value should be a **`<real>`**, corresponding to the desired interval in seconds.
mode	The mode of the lock to wait for. Valid values are **`#"read"`** (the default) and **`#"write"`**, which wait for locks in read mode and write mode respectively.

Values	*success*	An instance of **`<boolean>`**.

Description

Claims a read-write lock, blocking if necessary. The behavior depends on the value of *mode*:

`#"read"`	If there is a write lock, blocks until the lock becomes free. Then claims the lock by incrementing its internal read-lock counter.
`#"write"`	First waits until the lock becomes free, by blocking if necessary. Then claims exclusive ownership of the lock in write mode.

If the claim is successful, this method returns true; otherwise it returns false.

release *Sealed method*

Summary

Releases a read-write-lock.

Signature

`release object #key => ()`

Arguments	*object*	An instance of **`<read-write-lock>`**.

Values

None.

Description

Releases a read-write lock.

If the lock is owned by the calling thread, it is freed. If the lock is locked in read mode, the count of the number of locks held is decremented; the lock is freed if the count becomes zero. Otherwise it is an error to release the lock, and an implementation is permitted to signal a **`<not-owned-error>`** condition.

owned? — *Sealed method*

Summary	Tests whether a read-write lock is owned — that is, has been locked in write mode — by the current thread.
Signature	**`owned?`** ***object*** **`=>`** ***owned?***

Arguments	*object*	An instance of **`<read-write-lock>`**.
Values	*owned?*	An instance of **`<boolean>`**.

Description: Tests whether a read-write lock is owned — that is, has been locked in write mode — by the current thread.

4.6.8 Notifications

<notification> — *Sealed instantiable class*

Summary	The class of objects that can be used to notify threads of a change of state elsewhere in the program.
Superclasses	**`<synchronization>`**

Init-keywords	**`lock:`**	An instance of **`<simple-lock>`**. Required.

Description: The class of objects that can be used to notify threads of a change of state elsewhere in the program. Notifications are used in association with locks, and are sometimes called *condition variables*. They may be used to support the sharing of data between threads using *monitors*. Each **`<notification>`** is permanently associated with a **`<simple-lock>`**, although the same lock may be associated with many notifications.

The required *lock* is associated with the notification, and it is only possible to wait for, or release, the notification if the lock is owned.

Threads wait for the change of state to be notified by calling **`wait-for`**. Threads notify other threads of the change of state by calling **`release`**.

Operations

The class **`<notification>`** provides the following operations:

`associated-lock`	Returns the lock associated with the notification object.
`wait-for`	Wait for the notification of the change in state. The associated lock must be owned, and is atomically released before synchronization, and reclaimed after.
`release`	Notify the change of state to a single waiting thread. This has no effect on the associated lock, which must be owned.
`release-all`	Notify the change of state to all waiting threads. This has no effect on the associated lock, which must be owned.

Example

This example shows how to use a notification and an associated lock to implement a queue. The variable **`*queue*`** is the actual queue object (a **`<deque>`**). Queue access is performed by interlocking pushes and pops on the **`<deque>`**. The **`*queue*`** variable can be a constant, since it is the **`<deque>`** which is mutated and not the value of **`*queue*`**.

```
define constant *queue* = make(<deque>);
```

The variable **`*lock*`** is used to isolate access to the queue

```
define constant *lock* = make(<lock>);
```

The variable **`*something-queued*`** is a notification which is used to notify other threads that an object is being put onto an empty queue.

```
define constant *something-queued* =
  make(<notification>, lock: *lock*);
```

The function **`put-on-queue`** pushes an object onto the queue. If the queue was initially empty, then all threads which are waiting for the queue to fill are notified that there is a new entry.

```
define method put-on-queue (object) => ()
  with-lock (*lock*)
    if (*queue*.empty?)
      release-all(*something-queued*)
    end;
    push(*queue*, object)
  end with-lock
end method;
```

The **`get-from-queue`** function returns an object from the queue. If no object is immediately available, then it blocks until it receives a notification that the queue is no longer empty. After receiving the notification it tests again to see if an object is present, in case it was popped by another thread.

```
define method get-from-queue () => (object)
  with-lock (*lock*)
    while (*queue*.empty?)
      wait-for(*something-queued*)
    end;
    pop(*queue*)
  end with-lock
end method;
```

associated-lock *Function*

Summary
: Returns the lock associated with the notification object supplied.

Signature
: **`associated-lock`** ***notification*** **`=>`** ***lock***

Arguments	*notification*	An instance of **`<notification>`**.
Values	*lock*	An instance of **`<simple-lock>`**.

Description: Returns the lock associated with the notification object *notification*.

wait-for *Sealed method*

Summary: Wait for another thread to release a notification.

Signature: **`wait-for`** ***notification*** **`#key`** ***timeout*** **`=>`** ***success***

Arguments	*notification*	An instance of **`<notification>`**, page 97.
	timeout	Time-out interval. If the value is **`#f`** (the default), the time-out interval never elapses. Otherwise the value should be a **`<real>`**, corresponding to the desired interval in seconds.
Values	*success*	An instance of **`<boolean>`**.

Description: Wait for another thread to release *notification*. The lock associated with the notification must be owned. Atomically, the lock is released and the current thread starts blocking, waiting for another thread to release the notification. The current thread reclaims the lock once it has received the notification.

Note that the state should be tested again once **`wait-for`** has returned, because there may have been a delay between the **`release`**, page 101 of the notification and the claiming of the lock, and the state may have been changed during that time. If a timeout is supplied, then this is used for waiting for the release of the notification only. The **`wait-for`** function always waits for the lock with no timeout, and it is guaranteed that the lock will be owned on return. The **`wait-for`** function returns **`#f`** if the notification wait times out.

Exceptions

Implementations of this **`wait-for`** method are permitted to signal a condition of the following class, which is a subclass of **`<error>`**:

`<not-owned-error>`

Implementations can signal this error if the application attempts to wait for a notification when the associated lock is not owned by the current thread.

release *Sealed method*

Summary

Releases a notification to one of the threads that are blocked and waiting for it.

Signature

`release` ***notification*** **`#key => ()`**

Arguments

notification	An instance of **`<notification>`**.

Values

None.

Exceptions

Implementations of this **`release`** method are permitted to signal a condition of the following class, which is a subclass of **`<error>`**:

`<not-owned-error>`

Implementations can signal this error if the application attempts to release a notification when the associated lock is not owned by the current thread.

Description

Releases *notification*, announcing the change of state to one of the threads which are blocked and waiting for it. The choice of which thread receives the notification is undefined. The receiving thread may not be unblocked immediately, because it must first claim ownership of the notification's associated lock.

release-all *Function*

Summary
: Release a notification to all the threads that are blocked and waiting for it.

Signature
: **`release-all`** ***notification*** **`=> ()`**

Arguments
: *notification* An instance of **`<notification>`**.

Exceptions
: Implementations of the **`release-all`** function are permitted to signal a condition of the following class, which is a subclass of **`<error>`**:

`<not-owned-error>`

This may be signalled when an attempt is made to release a notification when the associated lock is not owned by the current thread.

Description
: Releases *notification*, announcing the change of state to all threads which are blocked and waiting for it. Those threads will then necessarily have to compete for the lock associated with the notification.

4.7 Timers

sleep *Function*

Summary
: Blocks the current thread for a specified number of seconds.

Signature	**sleep** ***interval*** **=> ()**	
Arguments	*interval*	An instance of **<real>**.
Values	None.	
Description	Blocks the current thread for the number of seconds specified in *interval*.	

4.8 Thread variables

thread *Variable definition adjective*

Summary
: An adjective to **define variable** for defining thread variables.

Macro call
: **define thread variable** ***bindings*** **=** ***init*****;**

Description
: An adjective to **define variable**. The construct **define thread variable** defines module variables in the current module which have thread-local bindings. The initialization expression is evaluated once, and is used to provide the initial values for the variables in each thread. The value of a thread variable binding may be changed with the normal assignment operator **:=**. This assignment is not visible in other threads.

Example

```
define thread variable *standard-output*
  = make(<standard-output-stream>);
```

4.9 Dynamic binding

dynamic-bind *Statement macro*

Summary

Executes a body of code in a context in which variables are dynamically rebound.

Macro call

dynamic-bind (*place1* = *init1*, *place2* = *init2*, ...) *body* end;

Description

Executes *body* with the specified *places* rebound in the dynamic environment, each place being initialized to the results of evaluating the initialization expressions. In other words, the places are initialized to new values on entry to the body but restored to their old values once the body has finished executing, whether because it finishes normally, or because of a non-local transfer of control. Typically, each *place* is a thread variable.

If the *place* is a *name*, it must be the name of a thread variable in the module scope.

Example

The following example shows the dynamic binding of a single variable.

```
dynamic-bind (*standard-output* = new-val())
  top-level-loop ()
end;
```

This expands into code equivalent to the following:

```
begin
  let old-value = *standard-output*;
  block ()
    *standard-output* := new-val();
    top-level-loop()
  cleanup
    *standard-output* := old-value
  end
end
```

4.9.1 An extended form of dynamic-bind

Some implementations of the Threads library may provide an extended form of **`dynamic-bind`** for binding places other than variables. The implementation of this extended form requires the use of non-standard features in the Dylan macro system, and hence cannot be written as a portable macro. These non-standard extensions are subject to discussion amongst the Dylan language designers, and may eventually become standard features. Until such time as standardization occurs, implementations are not mandated to implement the extended form of **`dynamic-bind`**, and portable code should not depend upon this feature.

The extended form is described below.

dynamic-bind *Statement macro*

Summary

Executes a body of code in a context in which variables or other places are dynamically rebound.

Macro call

`dynamic-bind (`*`place1`*` = `*`init1`*`, `*`place2`*` = `*`init2`*`, ...) body end;`

(This is the same as the simple form.)

Description

If *place* is not a name, then it may have the syntax of a call to a function. This permits an extended form for **`dynamic-bind`**, by analogy with the extended form for **`:=`**. In this case, if the place appears syntactically as **`name(`***arg1*, ... *argn***`)`**, then the macro expands into a call to the function

`name-dynamic-binder(`*`init`*`, `*`body-method`*`, `*`arg1`*`, ... `*`argn`*`)`

where *init* is the initial value for the binding, and *body-method* is function with no parameters whose body is the *body of* the **`dynamic-bind`**. The extended form also permits the other "**`.`**" and "**`[]`**"syntaxes for function calls.

There are no features in the current version of the Threads library which make use of the extended form of **`dynamic-bind`**.

Example

The following example shows the extended form of **dynamic-bind**.

```
dynamic-bind (object.a-slot = new-slot-val())
  inner-body(object)
end;
```

This expands into code equivalent to the following:

```
a-slot-dynamic-binder(new-slot-val(),
            method () inner-body(object) end,
            object)
```

4.10 Locked variables

locked *Variable definition adjective*

Summary

Defines a locked variable.

Macro call

define locked variable ***bindings*** **=** ***init*****;**

Description

An adjective to **define variable**. The construct **define locked variable** defines module variables in the current module that can be tested and updated with **conditional-update!**, **atomic-increment!**, or **atomic-decrement!**.

Other threads are prevented from modifying the locked variable during the conditional update operation by means of a low-level locking mechanism, which is expected to be extremely efficient.

Operations

conditional-update!

Atomically compare and conditionally assign to the variable.

atomic-increment!

Atomically increment the variable.

`atomic-decrement!`

Atomically decrement the variable.

Example

```
define locked variable *number-detected* = 0;
```

4.11 Conditional update

conditional-update! *Statement macro*

Summary
Performs an atomic test-and-set operation.

Macro call

```
conditional-update!(local-name = place)
  body
  [success success-expr]
  [failure failure-expr]
end
```

Arguments

local-name	A Dylan variable-name$_{bnf}$.
place	A Dylan variable-name$_{\mathrm{bnf}}$, If the implementation provides the extended form of **`conditional-update!`**, *place* can also be a function call.
body	A Dylan body$_{bnf}$.

Values
See Description.

Description
Performs an atomic test-and-set operation. Where appropriate, it should be implemented using dedicated processor instructions, and is expected to be extremely efficient on most platforms.

The value of the *place* is evaluated once to determine the initial value, which is then bound to the *local-name* as a lexical variable. The *body* is then evaluated to determine the new value for the place. The place is then conditionally updated

— which means that the following steps are performed atomically:

1. The place is evaluated again, and a test is made to see if it has been updated since the initial evaluation. This may involve a comparison with the old value using **`==`**, though implementations might use a more direct test for there having been an assignment to the place. It is undefined whether the test will succeed or fail in the case where the place was updated with a value that is identical to the old value when compared using **`\==`**.
2. If the value was found not to have been updated since the initial evaluation, the new value is stored by assignment. Otherwise the conditional update fails.

If the update was successful, then **`conditional-update!`** returns the result of the **`success`** expression, or returns the new value of the place if no **`success`** clause was supplied.

If the update failed, then **`conditional-update!`** signals a condition, unless a **`failure`** clause was given, in which case the value is returned.

If the *place* is a *name*, it must be the name of a **`locked variable`** in the current module scope. See Section 4.10 on page 106.

Exceptions

`conditional-update!` may signal a condition of the following class (which is a subclass of **`<error>`**), unless a **`failure`** clause is supplied.

`<conditional-update-error>`

Example

The following example does an atomic increment of **`*number-detected*`**.

```
until (conditional-update!
        (current-val = *number-detected*)
          current-val + 1
        failure #f
       end conditional-update!)
end until
```

atomic-increment! *Function macro*

Summary	Atomically increments a place containing a numeric value.	
Macro call	**atomic-increment!(*place*);**	
	atomic-increment!(*place*, *by*);	
Arguments	*place*	A Dylan variable-name$_{bnf}$. If the implementation provides the extended form of **conditional-update!**, *place* can also be a function call.
	by	An instance of **<object>**. Default value: 1.
Values	*new-value*	An instance of **<object>**.

Description

Atomically increments a place containing a numeric value.

The value of the *place* is evaluated one or more times to determine the initial value. A new value is computed from this value and *by*, by applying **+** from the Dylan module. The new value is atomically stored back into *place*.

The macro returns the new value of *place*.

The *place* must be a suitable place for **conditional-update!**.

Implementations of **atomic-increment!** are permitted to use **conditional-update!** (as in the described example), and hence can involve a loop and can cause *place* to be evaluated more than once. However, an atomic increment of a locked variable might be implemented by a more efficient non-looping mechanism on some platforms.

Example

The following example atomically increments **`*number-detected*`** by 2, and returns the incremented value.

```
atomic-increment!(*number-detected*, 2);
```

atomic-decrement! *Function macro*

Summary

Atomically decrements a place containing a numeric value.

Macro call

`atomic-decrement!(`*place***`)`**

`atomic-decrement!(`*place, by***`)`**

Arguments

place	A Dylan variable-name$_{bnf}$. If the implementation provides the extended form of **`conditional-update!`**, *place* can also be a function call.
by	An instance of **`<object>`**. Default value: 1.

Values

new-value	An instance of **`<object>`**.

Description

Atomically decrements a place containing a numeric value. It has the same semantics as **`atomic-increment!`** with the exception that the *place* is decremented.

4.11.1 An extended form of conditional-update!

Some implementations of the Threads library may provide an extended form of **`conditional-update!`** for updating places other than locked variables. The implementation of this extended form requires the use of non-standard features in the Dylan macro system, and hence cannot be written as a portable macro. These non-standard extensions are subject to discussion amongst the Dylan language designers, and may eventually become features. Until such time as standardization occurs, implementations are not mandated to implement the extended form of **`conditional-update!`**, and portable code should not depend upon the feature.

conditional-update! *Statement macro*

Summary
Performs an atomic test-and-set operation.

Macro call

```
conditional-update!(local-name = place)
  body
  [success success-expr]
  [failure failure-expr]
end
```

Arguments

local-name	A Dylan variable-name$_{bnf}$.
place	A Dylan variable-name$_{bnf}$ or a function call.
body	A Dylan body$_{bnf}$.

Values
See Description.

Description
This extended form of **`conditional-update!`** additionally accepts a *place* that has the syntax of a call to a function. This extended form for **`conditional-update!`** is analogous to that for **`:=`**. In this case, if the *place* appears syntactically as

***name*(*arg*$_1$, … *arg*$_n$)**

The macro expands into this call:

***name*-conditional-updater(*new-value*, *local-name*, *arg*$_1$, … *arg*$_n$)**

If the result of this function call is **`#f`**, the conditional update is deemed to have failed.

5

Integers

5.1 Introduction

This chapter describes the Common Dylan implementation of arithmetic functions, especially integer arithmetic. It describes a number of extensions to the Dylan language, which are available from the Dylan library. It also describes a generic arithmetic facility that, through the use of other libraries, allows you to extend arithmetic to special number types, such as "big" (64-bit) integers.

Throughout this chapter, arguments are instances of the class specified by the argument name (ignoring any numeric suffixes), unless otherwise noted. Thus, the arguments *integer, integer1,* and *integer2* would all be instances of the class **`<integer>`**.

The goals of the extensions to the Dylan language described in this chapter are as follows:

- Provide arithmetic operations that are closed over small integers.

  This allows type inference to propagate small integer declarations more widely, because there is no possibility of automatic coercion into some more general format.

- Make the arithmetic operations that are closed over small integers easily accessible to programmers.

- Allow the Dylan library to be described in such a way that only small integers are present by default, moving support for infinite precision integer arithmetic to the Big-Integers library, which must be explicitly used.
- Support infinite precision integer arithmetic through the Big-Integers library.

  Note: Using that library in another library does not have a negative effect on the correctness or performance of other libraries in the same application that do not use it.
- Maintain compatibility with the DRM specification.

  In particular, the extensions support the production of efficient code for programs written to be portable with respect to the DRM specification. Use of implementation-specific types or operations in order to get reasonable efficiency is not required. This precludes relegating the **`<integer>`** class and **`limited-<integer>`** types to inefficient implementations.

  Note: When there are several distinct interfaces with the same name but in different modules, the notation *interface#module* is used in this chapter to remove ambiguity.
- Specify that the class **`<integer>`** has a finite, implementation-dependent range, bounded by the constants **`$minimum-integer`** and **`$maximum-integer`**.

  The representation for integers must be at least 28 bits, including the sign. That is, the minimum conforming value for **`$maximum-integer`** is 2^{27}-1 and the maximum conforming value for **`$minimum-integer`** is -2^{27}.

  Rationale: Restricting **`<integer>`** in this way allows the programmer to stay in the efficient range without requiring exact knowledge of what that range might be. The full generality of extended precision integers is provided by the Big-Integers library, for programmers who actually need that functionality.
- Define the type **`<machine-number>`** to be the type union of **`<float>`** and **`<integer>`**.

The Dylan library provides implementations of the generic functions and functions described in this chapter. If the result of one of these operations is specified to be an instance of **`<integer>`** and the mathematically correct result cannot be represented as an **`<integer>`** then an error is signaled. This removes fully generic arithmetic from the Dylan library. In particular, it removes extended integers, ratios, and rectangular complex numbers.

5.2 Extensions to the Dylan library

This section describes the extensions to the Dylan library that provide the arithmetic operations available as standard to your applications. You do not have to explicitly use any additional libraries to have access to any of the functionality described in this section. Note that this section only describes extensions to the Dylan library; for complete descriptions, you should also refer to the *Dylan Reference Manual*.

Note that the Common-Dylan library also has these extensions because it uses the Dylan library.

5.2.1 Ranges

The initialization arguments for **`<range>`** must all be instances of **`<machine-number>`** rather than **`<real>`**.

5.2.2 Specific constructors

The following specific constructors are available for use with the class **`<integer>`**.

limited *G.f. method*

Summary

Defines a new type that represents a subset of the class **`<integer>`**.

Arguments

`singleton(<integer>)`

`min:`	The lower bound of the range. The default is **`$minimum-integer`**.

max:	The upper bound of the range. The default is **$maximum-integer**

Signature **limited** ***integer-class*** **#key** ***min max*** **=>** ***limited-type***

Description The *integer-class* argument is the class **<integer>**, and all other arguments are instances of **<integer>**. The range of **<integer>** is bounded by default.

range *Function*

Summary This function is used to specify ranges of numbers.

Arguments

Signature **range (#key from:, to:, above:, below:, by:, size:) => <range>**

Description All of the supplied arguments must be instances of **<machine-number>**.

5.2.3 Equality comparisons

The **=** function compares two objects and returns **#t** if the values of the two objects are equal to each other, that is of the same magnitude.

= *Generic function, Sealed domain, G.f. method*

Summary Tests its arguments to see if they are of the same magnitude.

Signature

> **=** ***object1 object2*** **=>** ***boolean*** **(*Generic function*)**
> **=** ***complex1 complex2*** **=>** ***boolean*** **(*Sealed domain*)**
> **=** ***machine-number1 machine-number2*** **=>** ***boolean*** **(*G.f. method*)**

Value `<boolean>`

Other available methods are described in the *Dylan Reference Manual*.

5.2.4 Magnitude comparisons

The Dylan library provides the following interfaces for testing the magnitude of two numbers:

< *Generic function, Sealed domain, G.f. method*

Summary Returns #t if its first argument is less than its second argument.

Signature

> **<** ***object1 object2*** **=>** ***boolean*** **(*Generic function*)**
> **<** ***complex1 complex2*** **(*Sealed domain*)**
> **<** ***machine-number1 machine-number2*** **=>** ***boolean*** **(*G.f. method*)**

Other available methods are described in the *Dylan Reference Manual*.

5.2.5 Properties of numbers

Various number properties can be tested using the following predicates in the Dylan library:

odd? *Open generic function, Sealed domain, G.f. method*

Summary Tests whether the argument supplied represents an odd value.

Signature

odd? ***object*** **=>** ***boolean*** **(*Open generic function*)**
odd? ***complex*** **=>** ***boolean*** **(*Sealed domain*)**
odd? ***integer*** **=>** ***boolean*** **(*G.f. method*)**

even? *Open generic function, Sealed domain, G.f. method*

Summary

Tests whether the argument supplied represents an even value

Signature

even? ***object*** **=>** ***boolean*** **(*Open generic function*)**
even? ***complex*** **=>** ***boolean*** **(*Sealed domain*)**
even? ***integer*** **=>** ***boolean*** **(*G.f. method*)**

zero? *Open generic function*

zero? ***object*** **=>** ***boolean***

zero? *Sealed domain*

zero? ***complex***

zero? *G.f. method*

zero? ***machine-number*** **=>** ***boolean***

Tests whether the argument supplied represents a zero value.

positive? *Open generic function*

`positive?` ***object*** **`=>`** ***boolean***

positive? *Sealed domain*

`positive?` ***complex***

positive? *G.f. method*

`positive?` ***machine-number*** **`=>`** ***boolean***

Tests whether the argument supplied represents a positive value.

negative? *Open generic function*

`negative?` ***object*** **`=>`** ***boolean***

negative? *Sealed domain*

`negative?` ***complex***

negative? *G.f. method*

`negative?` ***machine-number*** **`=>`** ***boolean***

Tests whether the argument supplied represents a negative value.

integral? *Open generic function*

`integral?` ***object*** **`=>`** ***boolean***

integral? *Sealed domain*

`integral?` ***complex***

integral? *G.f. method*

`integral?` ***machine-number*** **`=>`** ***boolean***

Tests whether the argument supplied represents an integral value.

5.2.6 Arithmetic operations

The following arithmetic operations are available in the Dylan library:

+ *Open generic function*

+ ***object1 object2*** **=> #rest** ***object***

+ *Sealed domain*

+ ***complex1 complex2***

+ *G.f. method*

+ ***integer1 integer2*** **=>** ***integer***

+ *G.f. method*

+ ***machine-number1 machine-number2*** **=>** ***machine-number***

Returns the sum of the two supplied arguments. The actual type of the value is determined by the contagion rules when applied to the arguments.

- *Open generic function*

- ***object1 object2*** **=> #rest** ***object***

- *Sealed domain*

- ***complex1 complex2***

- *G.f. method*

- ***integer1 integer2*** **=>** ***integer***

- *G.f. method*

- ***machine-number1 machine-number2*** **=>** ***machine-number***

Returns the result of subtracting the second argument from the first. The actual type of the value is determined by the contagion rules when applied to the arguments.

***** *Open generic function*

***** ***object1 object2*** **=> #rest** ***object***

***** *Sealed domain*

***** ***complex1 complex2***

***** *G.f. method*

***** ***integer1 integer2*** **=>** ***integer***

***** *G.f. method*

***** ***machine-number1 machine-number2*** **=>** ***machine-number***

Returns the result of multiplying the two arguments. The actual type of the value is determined by the contagion rules when applied to the arguments.

/ *Open generic function*

`/` *object1 object2* `=> #rest` *object*

/ *Sealed domain*

`/` *complex1 complex2*

/ *G.f. method*

`/` *float1 float2* `=>` *float*

Returns the result of dividing the first argument by the second. The actual type of the value is determined by the contagion rules when applied to the arguments.

negative *Open generic function*

`negative` *object* `=> #rest` *negative-object*

negative *Sealed domain*

`negative` *complex*

negative *G.f. method*

`negative` *integer* `=>` *negative-integer*

negative *G.f. method*

`negative` *float* `=>` *negative-float*

Negates the supplied argument. The returned value is of the same float format as the supplied argument.

floor *Function*

`floor` ***machine-number*** **`=>`** ***integer machine-number***
`floor` ***integer*** **`=>`** ***integer integer***
`floor` ***float*** **`=>`** ***integer float***

Truncates a number toward negative infinity. The integer part is returned as *integer*, the remainder is of the same float format as the argument.

ceiling *Function*

`ceiling` ***machine-number*** **`=>`** ***integer machine-number***
`ceiling` ***integer*** **`=>`** ***integer integer***
`ceiling` ***float*** **`=>`** ***integer float***

Truncates a number toward positive infinity. The integer part is returned as *integer*, the remainder is of the same float format as the argument.

round *Function*

`round` ***machine-number*** **`=>`** ***integer machine-number***
`round` ***integer*** **`=>`** ***integer integer***
`round` ***float*** **`=>`** ***integer float***

Rounds a number toward the nearest mathematical integer. The integer part is returned as *integer*, the remainder is of the same float format as the argument. If the argument is exactly between two integers, then the result *integer* will be a multiple of two.

truncate *Function*

`truncate` ***machine-number*** **`=>`** ***integer machine-number***
`truncate` ***integer*** **`=>`** ***integer integer***
`truncate` ***float*** **`=>`** ***integer float***

Truncates a number toward zero. The integer part is returned as *integer*, the remainder is of the same float format as the argument.

floor/ *Function*

`floor/` ***machine-number1 machine-number2*** **`=>`** ***integer machine-number***
`floor/` ***integer1 integer2*** **`=>`** ***integer integer***
`floor/` ***machine-number1 machine-number2*** **`=>`** ***integer machine-number***

Divides the first argument into the second and truncates the result toward negative infinity. The integer part is returned as *integer*, the type of the remainder is determined by the contagion rules when applied to the arguments.

ceiling/ *Function*

`ceiling/` ***machine-number1 machine-number2*** **`=>`** ***integer machine-number***
`ceiling/` ***integer1 integer2*** **`=>`** ***integer integer***
`ceiling/` ***machine-number1 machine-number2*** **`=>`** ***integer machine-number***

Divides the first argument into the second and truncates the result toward positive infinity. The integer part is returned as *integer*, the type of the remainder is determined by the contagion rules when applied to the arguments.

round/ *Function*

`round/` ***machine-number1 machine-number2*** **`=>`** ***integer machine-number***
`round/` ***integer1 integer2*** **`=>`** ***integer integer***
`round/` ***machine-number1 machine-number2*** **`=>`** ***integer machine-number***

Divides the first argument into the second and rounds the result toward the nearest mathematical integer. The integer part is returned as *integer*, the type of the remainder is determined by the contagion rules when applied to the arguments.

truncate/ *Function*

`truncate/` ***machine-number1 machine-number2*** **`=>`** ***integer machine-number***
`truncate/` ***integer1 integer2*** **`=>`** ***integer integer***
`truncate/` ***machine-number1 machine-number2*** **`=>`** ***integer machine-number***

Divides the first argument into the second and truncates the result toward zero. The integer part is returned as *integer*, the type of the remainder is determined by the contagion rules when applied to the arguments.

modulo *Function*

`modulo` ***machine-number1 machine-number2*** **`=>`** ***machine-number***
`modulo` ***integer1 integer2*** **`=>`** ***integer***
`modulo` ***machine-number1 machine-number2*** **`=>`** ***machine-number***

Returns the second value of **`floor/ (`***arg1***`,`** *arg2***`)`**. The actual type of the second value is determined by the contagion rules when applied to the arguments.

remainder *Function*

`remainder` ***machine-number1 machine-number2*** **`=>`** ***machine-number***
`remainder` ***integer1 integer2*** **`=>`** ***integer***
`remainder` ***machine-number1 machine-number2*** **`=>`** ***machine-number***

Returns the second value of **`truncate/ (`***arg1***`,`** *arg2***`)`**.The actual type of the second value is determined by the contagion rules when applied to the arguments.

^ *Open generic function*

^ ***object1 object2*** **=> #rest** ***object***

^ *Sealed domain*

^ ***complex1 complex*2**

^ *G.f. method*

^ ***integer1 integer2*** **=>** ***integer***

^ *G.f. method*

^ ***float1 integer2*** **=>** ***float***

Returns the first argument raised to the power of the second argument. The value is of the same float format as the first argument. An error is signalled if both arguments are 0.

abs *Open generic function*

abs ***object*** **=> #rest** ***object***

abs *Sealed domain*

abs ***complex***

abs *G.f. method*

abs ***integer*** **=>** ***integer***

abs *G.f. method*

abs ***float*** **=>** ***float***

Returns the absolute value of the argument. The value is of the same float format as the argument.

logior *Function*

`logior #rest` ***integers*** **`=>`** ***integer***

Returns the bitwise inclusive **`OR`** of its integer arguments.

logxor *Function*

`logxor #rest` ***integers*** **`=>`** ***integer***

Returns the bitwise exclusive **`OR`** of its integer arguments.

logand *Function*

`logand #rest` ***integers*** **`=>`** ***integer***

Returns the bitwise **`AND`** of its integer arguments.

lognot *Function*

`lognot` ***integer1*** **`=>`** ***integer2***

Returns the bitwise **`NOT`** of its integer arguments.

logbit? *Function*

`logbit?` ***index integer*** **`=>`** ***boolean***

Tests the value of a particular bit in its integer argument. The *index* argument is an instance of **`<integer>`**.

ash *Function*

`ash` ***integer1 count*** **`=>`** ***integer***

Performs an arithmetic shift on its first argument.

lcm *Function*

`lcm` ***integer1 integer2*** **`=>`** ***integer***

Returns the least common multiple of its two arguments.

gcd *Function*

`gcd` ***integer1 integer2*** **`=>`** ***integer***

Returns the greatest common divisor of its two arguments.

5.2.7 Collections

The keys for sequences are always instances of **`<integer>`**. This means that certain kinds of collections cannot be sequences; very large (or unbounded) sparse arrays are an example.

5.2.8 The table protocol

The following functions in the Dylan library are extended. Note that the hash IDs for tables are always instances of **`<integer>`**.

merge-hash-codes *Function*

`merge-hash-codes` ***id1 state1 id2 state2*** **`#key`** ***ordered?***
`=>` ***merged-id merged-state***

Returns a hash code created from the merging of two argument hash codes. The *id* arguments are hash IDs, and the *state* arguments are hash states (instances of **`<object>`**). The *ordered?* argument is an instance of **`<boolean>`**. The returned merged values are instances of **`<integer>`** and **`<object>`**, as determined by the name of each argument.

object-hash *Function*

`object-hash` ***object*** **`=>`** ***hash-id hash-state***

The hash function for the equivalence predicate **`==`**. The return values are of the same types as the return values of **`merge-hash-codes`**.

5.2.9 Iteration constructs

for *Statement macro*

The *start*, *bound*, and *increment* expressions in a numeric clause must evaluate to instances of **`<machine-number>`** for this macro.

5.3 The Generic-Arithmetic library

The Generic-Arithmetic library exports the functions described in this section from an exported module called **`generic-arithmetic`**.

The Generic-Arithmetic library provides a fully extensible version of all arithmetic operations. If an application only uses Generic-Arithmetic, these versions of the operators reduce themselves to be equivalent to those in the Dylan library. But when you use additional implementation libraries, the arithmetic operators are extended.

The Big-Integers library is one such implementation library. It provides a 64-bit implementation of **`<integer>`**.

The standard integer implementation in the Dylan library is actually part of the following class hierarchy:

```
<abstract-integer>
  <integer>
  <big-integer>
    <double-integer>
```

(The classes **`<big-integer>`** and **`<double-integer>`** are implementation classes. You do not need to use them.)

The modules in the Generic-Arithmetic library export **`<abstract-integer>`** with the name **`<integer>`**. They also export a full set of arithmetic operators that use instances of **`<abstract-integer>`** rather than instances of **`<integer>`** (in the Dylan library naming scheme). However, those operators just fall back to the Dylan library operators until you include an implementation library, such as Big-Integers, in your application.

When you use the Big-Integers library, the arithmetic operators exported by Generic-Arithmetic are enhanced to extend their results to 64-bit integers. If a result is small enough to fit in a Dylan library **`<integer>`**, it will be fitted into one.

Note that the Generic-Arithmetic library uses the same naming conventions for arithmetic operators as used by the Dylan library. This means that some renaming is required in modules that require access to both the basic Dylan interfaces and the interfaces supplied by the Generic-Arithmetic library. As described earlier, the notation *interface#module* is used to denote different interfaces of the same name, where *interface* is the name of the interface, and *module* is the name of the module it is exported from.

See "Using special arithmetic features" on page 133 for an example of how to use an implementation library with Generic-Arithmetic.

5.3.1 Ranges

The Generic-Arithmetic library defines the class **`<range>`**, which is in most respects functionally equivalent to **`<range>#Dylan`**, but uses generic arithmetic operations in its implementation so that the initialization arguments can be instances of **`<real>`**, rather than being restricted to **`<machine-number>`**.

5.3.2 Classes

The class **`<abstract-integer>`** is imported and re-exported under the name **`<integer>#generic-arithmetic`**.

5.3.3 Specific constructors

range *Function*

`range #key` ***from to above below by size*** **`=>`** ***range***

This function is identical to the function **`range#Dylan`**, except that all of the supplied arguments must be instances of **`<real>`**.

5.3.4 Arithmetic operations

The following functions all apply *function***`#Dylan`** to the arguments and return the results, where *function* is the appropriate function name. See Section 5.2.6

on page 120 for descriptions of each function as implemented in the Dylan library.

> `+` *object1 object2* `=>` `#rest` *object*
> `-` *object1 object2* `=>` `#rest` *object*
> `*` *object1 object2* `=>` `#rest` *object*
> `/` *object1 object2* `=>` `#rest` *object*
> `negative` *object* `=>` `#rest` *negative-object*
> `floor` *real1* `=>` *abstract-integer real*
> `ceiling` *real1* `=>` *abstract-integer real*
> `round` *real1* `=>` *abstract-integer real*
> `truncate` *real1* `=>` *abstract-integer real*
> `floor/` *real1 real2* `=>` *abstract-integer real*
> `ceiling/` *real1 real2* `=>` *abstract-integer real*
> `round/` *real1 real2* `=>` *abstract-integer real*
> `truncate/` *real1 real2* `=>` *abstract-integer real*
> `modulo` *real1 real2* `=>` *real*
> `remainder` *real1 real2* `=>` *real*
> `^` *object1 object2* `=>` `#rest` *object*
> `abs` *object1* `=>` `#rest` *object*
> `logior` `#rest` *abstract-integer1* `=>` *abstract-integer*
> `logxor` `#rest` *abstract-integer1* `=>` *abstract-integer*
> `logand` `#rest` *abstract-integer1* `=>` *abstract-integer*
> `lognot` *abstract-integer1* `=>` *abstract-integer*
> `logbit?` *integer abstract-integer* `=>` *boolean*
> `ash` *abstract-integer1 integer* `=>` *abstract-integer*
> `lcm` *abstract-integer1 abstract-integer2* `=>` *abstract-integer*
> `gcd` *abstract-integer1 abstract-integer2* `=>` *abstract-integer*

5.3.5 Iteration constructs

While a programmer could make use of generic arithmetic in a **`for`** loop by using explicit-step clauses, this approach leads to a loss of clarity. The definition of the **`for`** macro is complex, so a version that uses generic arithmetic in numeric clauses is provided, rather than requiring programmers who want that feature to reconstruct it.

for *Statement macro*

The *start*, *bound*, and *increment* expressions in a numeric clause must evaluate to instances of **`<machine-number>`** for this macro. Otherwise, this macro is similar to **`for#Dylan`**.

5.3.6 Exported modules from the Generic-Arithmetic library

The Generic-Arithmetic library exports several modules that are provided for the convenience of programmers who wish to create additional modules based on the **dylan** module plus various combinations of the arithmetic models.

5.3.6.1 The Dylan-Excluding-Arithmetic module

The Dylan-Excluding-Arithmetic module imports and re-exports all of the interfaces exported by the **dylan** module from the Dylan library, except for the following excluded interfaces:

```
<integer>
range
+ - * /
negative
floorceilingroundtruncate
floor/ceiling/round/truncate/
moduloremainder
^
abs
logiorlogxorlogandlognot
logbit?
ash
lcmgcd
for
```

5.3.6.2 The Dylan-Arithmetic module

The Dylan-Arithmetic module imports and re-exports all of the interfaces exported by the **dylan** module from the Dylan library which are excluded by the **dylan-excluding-arithmetic** module.

5.3.6.3 The Generic-Arithmetic-Dylan module

The Generic-Arithmetic-Dylan module imports and reexports all of the interfaces exported by the **dylan-excluding-arithmetic** module and the **generic-arithmetic** module.

The **dylan-excluding-arithmetic**, **dylan-arithmetic**, and **generic-arithmetic** modules provide convenient building blocks for programmers to build the particular set of global name bindings they wish to work with. The purpose of the **`generic-arithmetic-dylan`** module is to provide a standard environment in which generic arithmetic is the norm, for those programmers who might want that.

5.4 Using special arithmetic features

As noted in Section 5.3, the Generic-Arithmetic library provides an extensible protocol for adding specialized arithmetic functionality to your applications. By using the Generic-Arithmetic library alongside a special implementation library, you can make the standard arithmetic operations support number types such as big (64-bit) integers, or complex numbers.

This section provides an example of extending the basic Dylan arithmetic features using the Generic-Arithmetic library and the Big-Integers implementation library.

To use special arithmetic features, an a library's **`define library`** declaration must use at least the following libraries:

```
common-dylan
generic-arithmetic
special-arithmetic-implementation-library
```

So for Big-Integers you would write:

```
define library foo
  use common-dylan;
  use generic-arithmetic;
  use big-integers;
  …
end library foo;
```

Next you have to declare a module. There are three ways of using big-integer arithmetic that we can arrange with a suitable module declaration:

1. Replace all integer arithmetic with the big-integer arithmetic
2. Use both, with normal arithmetic remaining the default
3. Use both, with the big-integer arithmetic becoming the default

To get one of the three different effects described above, you need to arrange the **`define module`** declaration accordingly. To replace all integer arithmetic with big-integer arithmetic, include the following in your **`define module`** declaration:

```
use generic-arithmetic-common-dylan;
```

(Note that the module definition should not use the Big-Integers module. The Big-Integers library is used as a side-effects library only, that is, it is referenced in the library definition so that it will be loaded. Its definitions extend the Generic-Arithmetic library.)

If you replace all integer arithmetic with big-integer arithmetic in this way, there will be performance hits. For instance, loop indices will have to be checked at run-time to see whether a normal or big integer representation is being used, and a choice must be made about the representation for an incremented value.

You can take a different approach that reduces the cost of big-integer arithmetic. Under this approach you leave normal integer arithmetic unchanged, and get access to big-integer arithmetic when you need it. To do this, use the same libraries but instead of using the **`common-dylan-generic-arithmetic`** module, include the following in your **`define module`** declaration:

```
use common-dylan;
use generic-arithmetic, prefix: "ga/"; // use any prefix you like
```

This imports the big-integer arithmetic binding names, but gives them a prefix **`ga/`**, using the standard renaming mechanism available in module declarations. Thus you gain access to big arithmetic using renamed classes and operations like:

```
ga/<integer>
ga/+
ga/-
ga/*
…
```

The operations take either instances of **`<integer>`** or **`ga/<integer>`** (a subclass of **`<integer>`**) and return instances of **`ga/<integer>`**.

Note that having imported the big-integer operations under new names, you have to use prefix rather than infix syntax when calling them. For example:

```
ga/+ (5, 4);
```

not:

```
5 ga/+ 4;
```

The existing functions like **+** and **-** will only accept **<integer>** instances and **ga/<integer>** instances small enough to be represented as **<integer>** instances.

Under this renaming scheme, reduced performance will be confined to the **ga/** operations. Other operations, such as loop index increments and decrements, will retain their efficiency.

Finally, you can make big-integer arithmetic the default but keep normal arithmetic around for when you need it. Your **define module** declaration should contain:

```
use generic-arithmetic-common-dylan;
use dylan-arithmetic, prefix: "dylan/"; //use any prefix you like
```

5.5 The Big-Integers library

The Big-Integers library exports a module called **big-integers**, which imports and re-exports all of the interfaces exported by the **generic-arithmetic** module of the Generic-Arithmetic library.

The Big-Integers library modifies the behavior of functions provided by the Dylan library as described in this section.

5.5.1 Specific constructors

The Big-Integers library extends the functionality of specific constructors in the Dylan library as follows:

limited *G.f. method*

limited ***abstract-integer-class*** **#key** ***min max*** **=>** ***limited-type***

Returns a limited integer type, which is a subtype of **<abstract-integer>**, whose instances are integers greater than or equal to *min* (if specified) and less than or equal to *max* (if specified). If no keyword arguments are specified, the result type is equivalent to **<abstract-**

integer>. The argument *abstract-integer-class* is the class **<abstract-integer>**.

If both *min* and *max* are supplied, and both are instances of **<integer>**, then the result type is equivalent to calling **limited** on **<integer>** with those same bounds.

The Limited Integer Type Protocol is extended to account for limited **<abstract-integer>** types.

Table 5.1 Instances and subtypes in the Big-Integers library

This is true if and only if …	… all these clauses are true

`instance?`
`  (x,`
`   limited(<abstract-integer>,`
`   min: y, max: z))`
	`instance?(x, <abstract-integer>)`
	`(y <= x)`
	`(x <= z)`

`instance?`
`  (x,`
`   limited(<abstract-integer>,`
`   min: y))`
	`instance?(x, <abstract-integer>)`
	`(y <= x)`

`instance?`
`  (x,`
`   limited(<abstract-integer>,`
`   max: z))`
	`instance?(x, <abstract-integer>)`
	`(x <= z)`

`subtype?`
`  (limited(<abstract-integer>,`
`   min: w, max: x),`
`   limited(<abstract-integer>,`
`   min: y, max: z))`
	`(w >= y)`
	`(x <= z)`

`subtype?`
`  (limited(<abstract-integer>,`
`   min: w ...),`
`   limited(<abstract-integer>,`
`   min: y))`
	`(w >= y)`

Table 5.1 Instances and subtypes in the Big-Integers library

This is true if and only if …	… all these clauses are true

`subtype?`
`(limited(<abstract-integer>,`
`max: x ...),`
`limited(<abstract-integer>,`
`max: z))`
	`(x <= z)`

Table 5.2 Type-equivalence in the Big-Integers library

This is type equivalent to …	… this, if and only if …	… this is true

`limited`
`(<abstract-integer>,`
`min: y, max: z)`
	`limited`
	`(<integer>,`
	`min: y, max: z)`
		`y` and **`z`** are both instances of **`<integer>`**.

`limited`
`(<abstract-integer>,`
`min: y,`
`max: $maximum-integer)`
	`limited`
	`(<integer>, min: y)`
		`y` is an instance of **`<integer>`**.

`limited`
`(<abstract-integer>,`
`min: $minimum-integer,`
`max: z)`
	`limited`
	`(<integer>, max: z)`
		`z` is an instance of **`<integer>`**.

Type disjointness is modified as follows to account for limited **`<abstract-integer>`** types.

> A limited integer type is disjoint from a class if their base types are disjoint or the class is **`<integer>`** and the range of the limited integer type is disjoint from the range of **`<integer>`** (that is, from **`$minimum-integer`** to **`$maximum-integer`**).

5.5.2 Equality comparisons

The behavior of equality comparisons in the Dylan library is modified by the Big-Integers library as follows:

> **=** ***abstract-integer1 abstract-integer2*** **=>** ***boolean***
> **=** ***abstract-integer float*** **=>** ***boolean***
> **=** ***float abstract-integer*** **=>** ***boolean***

5.5.3 Magnitude comparisons

The behavior of magnitude comparisons in the Dylan library is modified by the Big-Integers library as follows:

> **<** ***abstract-integer1 abstract-integer2*** **=>** ***boolean***
> **<** ***abstract-integer float*** **=>** ***boolean***
> **<** ***float abstract-integer*** **=>** ***boolean***

5.5.4 Properties of numbers

The behavior of number property tests in the Dylan library is modified by the Big-Integers library as follows:

> **`odd?`** ***abstract-integer*** **=>** ***boolean***
> **`even?`** ***abstract-integer*** **=>** ***boolean***
> **`zero?`** ***abstract-integer*** **=>** ***boolean***
> **`positive?`** ***abstract-integer*** **=>** ***boolean***
> **`negative?`** ***abstract-integer*** **=>** ***boolean***
> **`integral?`** ***abstract-integer*** **=>** ***boolean***

5.5.5 Arithmetic operations

The Big-Integers library modifies the behavior of the functions provided by the Generic-Arithmetic library as described below.

The actual type of the return value for all the following interfaces is determined by the contagion rules when applied to the arguments.

+ ***abstract-integer1 abstract-integer2*** **=>** ***abstract-integer***
+ ***abstract-integer float1*** **=>** ***float***
+ ***float1 abstract-integer*** **=>** ***float***

- ***abstract-integer1 abstract-integer2*** **=>** ***abstract-integer***
- ***abstract-integer float1*** **=>** ***float***
- ***float1 abstract-integer*** **=>** ***float***

***** ***abstract-integer1 abstract-integer2*** **=>** ***abstract-integer***
***** ***abstract-integer float1*** **=>** ***float***
***** ***float1 abstract-integer*** **=>** ***float***

The return value of the following interface is of the same float format as the argument.

negative ***abstract-integer*** **=>** ***negative-abstract-integer***

The second return value of all the following interfaces is of the same float format as the argument.

floor ***abstract-integer*** **=>** ***abstract-integer1 abstract-integer2***
floor ***float1*** **=>** ***abstract-integer float***

ceiling ***abstract-integer*** **=>** ***abstract-integer1 abstract-integer2***
ceiling ***float1*** **=>** ***abstract-integer float***

round ***abstract-integer*** **=>** ***abstract-integer1 abstract-integer2***
round ***float1*** **=>** ***abstract-integer float***

truncate ***abstract-integer*** **=>** ***abstract-integer1 abstract-integer2***
truncate ***float1*** **=>** ***abstract-integer float***

The second return value of all the following interfaces is of the same float format as the first argument.

floor/ ***abstract-integer1 abstract-integer2*** **=>** ***abstract-integer3 abstract-integer4***
floor/ ***float1 abstract-integer1*** **=>** ***abstract-integer2 float2***

ceiling/ ***abstract-integer1 abstract-integer2***
=> ***abstract-integer3 abstract-integer4***
ceiling/ ***float1 abstract-integer1*** **=>** ***abstract-integer2 float2***

round/ ***abstract-integer1 abstract-integer2*** **=>** ***abstract-integer3 abstract-integer4***
round/ ***float1 abstract-integer1*** **=>** ***abstract-integer2 float2***

truncate/ ***abstract-integer1 abstract-integer2***

=> ***abstract-integer3 abstract-integer4***
truncate/ ***float1 abstract-integer1*** **=>** ***abstract-integer2 float2***

The second return value of the following interfaces is of the same float format as the second argument.

floor/ ***abstract-integer1 float1*** **=>** ***abstract-integer2 float2***

ceiling/ ***abstract-integer1 float1*** **=>** ***abstract-integer2 float2***

round/ ***abstract-integer1 float1*** **=>** ***abstract-integer2 float2***

truncate/ ***abstract-integer1 float1*** **=>** ***abstract-integer2 float2***

The return value of the following interfaces is of the same float format as the first argument.

modulo ***float1 abstract-integer*** **=>** ***float***

remainder ***float1 abstract-integer*** **=>** ***float***

The return value of the following interfaces is of the same float format as the second argument.

modulo ***abstract-integer1 abstract-integer2*** **=>** ***abstract-integer***
modulo ***abstract-integer float1*** **=>** ***float***

remainder ***abstract-integer1 abstract-integer2*** **=>** ***abstract-integer***
remainder ***abstract-integer float1*** **=>** ***float***

The behavior of the following miscellaneous interfaces is also modified by the Big-Integers library.

^ ***abstract-integer1 integer*** **=>** ***abstract-integer***
abs ***abstract-integer1*** **=>** ***abstract-integer***
logior #rest ***abstract-integer1*** **=>** ***abstract-integer***
logxor #rest ***abstract-integer1*** **=>** ***abstract-integer***
logand #rest ***abstract-integer1*** **=>** ***abstract-integer***
lognot ***abstract-integer1*** **=>** ***abstract-integer***
logbit? ***integer abstract-integer*** **=>** ***boolean***
ash ***abstract-integer1 integer*** **=>** ***abstract-integer***
lcm ***abstract-integer1 abstract-integer2*** **=>** ***abstract-integer***
gcd ***abstract-integer1 abstract-integer2*** **=>** ***abstract-integer***

6

The Machine Words Module

6.1 Introduction

This chapter describes the Harlequin Dylan implementation of machine words. It describes a number of extensions to the Dylan language, which are available from the Dylan library.

Throughout this chapter, arguments are instances of the class specified by the argument name, unless otherwise noted. Thus, the arguments ***machine-word*** and ***integer*** are instances of **`<machine-word>`** and **`<integer>`**, respectively.

The class **`<machine-word>`** is a sealed subclass of **`<object>`**, defined in the Dylan library. The class **`<machine-word>`** represents a limited range of integral values. The representation used has the natural size suggested by the implementation architecture. (On the PC, a **`<machine-word>`** is 32 bits wide.) The class **`<machine-word>`** is disjoint from all other classes specified by the Dylan language.

The **`\==`** function compares instances of **`<machine-word>`** by value.

6.2 Useful functions from the Dylan module

This section describes additional methods defined in the Dylan module that pertain to **`<machine-word>`**. Note that this section only describes extensions to the Dylan library; for complete descriptions, you should also refer to the *Dylan Reference Manual*.

Note that the Common Dylan library also has these extensions because it uses the Dylan library.

odd? *Sealed Method*

`odd? (m :: `*machine-word*`) => _ :: `*boolean*

even? *Sealed Method*

`even? (m :: `*machine-word*`) => _ :: `*boolean*

zero? *Sealed Method*

`zero? (m :: `*machine-word*`) => _ :: `*boolean*

Cannot be used as the name of a result. It is not a valid Dylan name.

positive? *Sealed Method*

`positive? (m :: `*machine-word*`) => _ :: `*boolean*

negative? *Sealed Method*

`negative? (m :: `*machine-word*`) => _ :: `*boolean*

These functions return a result based on interpreting `m` as a signed integer value.

= *Sealed Method*

`= (m1 :: `*machine-word*`, m2 :: `*machine-word*`) => _ :: `*boolean*

= *Sealed Method*

= (i1 :: ***abstract-integer*, m2 ::** ***machine-word*) => _ ::** ***boolean***

= *Sealed Method*

= (m1 :: ***machine-word*, i2 ::** ***abstract-integer*) => _ ::** ***boolean***

The comparison is performed with the ***machine-word*** arguments interpreted as signed integer values.

< *Sealed Method*

< (m1 :: ***machine-word*, m2 ::** ***machine-word*) => _ ::** ***boolean***

< *Sealed Method*

< (i1 :: ***abstract-integer*, m2 ::** ***machine-word*) => _ ::** ***boolean***

< *Sealed Method*

< (m1 :: ***machine-word*, i2 ::** ***abstract-integer*) => _ ::** ***boolean***

The comparison is performed with the ***machine-word*** arguments interpreted as signed integer values.

as *Sealed Method*

as(t == <*integer*>, m :: ***machine-word*) => _ ::** ***integer***

The result is an ***integer*** with the same value as `m` when interpreted as a signed integer value. An error is signaled if the value of `m` cannot be represented as an instance of ***integer***.

as *Sealed Method*

as(t == <*abstract-integer*>, m :: ***machine-word*) => _ ::** ***abstract-integer***

The result is an ***abstract-integer*** with the same value as `m` when interpreted as a signed integer value.

(The uses for an instance of ***abstract-integer*** that is not also an instance of ***integer*** are rather limited without the Generic-Arithmetic library.)

as *Sealed Method*

```
as(t == <machine-word>, i :: abstract-integer) => _ :: machine-word
```

If the value of **`i`** is outside the machine word range, then the result consists of the low **`$machine-word-size`** bits of the twos-complement representation of **`i`**. If any of the discarded bits differ from the sign of **`i`**, then an error is signaled.

limited *Sealed Method*

```
limited(t == <machine-word>,
  #key signed? :: boolean,
       min :: machine-word, max :: machine-word)
=> _ :: type
```

If the *signed?* argument is true (the default) then the *min* and *max* arguments are interpreted as signed values. When *signed?* is false, the *min* and *max* arguments are interpreted as unsigned values. The default value for each of min and max depends on the value of *signed?*. The defaults are taken from the corresponding minimum and maximum machine word values (see **`$maximum-signed-machine-word`** and related constants below).

For convenience, the values of *min* and / or *max* may also be instances of **`<abstract-integer>`**, in which case they are coerced to instances of **`<machine-word>`** as if by using **`as`**.

6.3 The MACHINE-WORDS module

This section contains a reference entry for each item exported from the Machine-Words module, which is exported by the Common Dylan library.

<machine-word> *Sealed Class*

Summary	The class of objects that can represent a limited range of integral values.
Superclasses	**`<object>`**
Init-keywords	None.
Library	**`dylan`**
Module	**`machine-word`**
Description	The class **`<machine-word>`** represents a limited range of integral values. The representation used has the natural size suggested by the implementation architecture. The class **`<machine-word>`** is disjoint from all other classes specified by the Dylan language.
Operations	The **`<machine-words>`** class provides the operations described below and in Section 6.2 on page 142.

6.3.1 Variables

The following variables are exported from the Machine-Words module.

$machine-word-size *Constant*

`$machine-word-size ::` ***integer***

The number of bits in the representation of a **`<machine-word>`**.

$maximum-signed-machine-word *Constant*

`$maximum-signed-machine-word ::` ***machine-word***

The largest machine word, when interpreted as a signed integer value.

$minimum-signed-machine-word *Constant*

`$minimum-signed-machine-word ::` ***machine-word***

The smallest machine word, when interpreted as a signed integer value.

$maximum-unsigned-machine-word *Constant*

`$maximum-unsigned-machine-word ::` ***machine-word***

The largest machine word, when interpreted as an unsigned integer value.

$minimum-unsigned-machine-word *Constant*

`$minimum-unsigned-machine-word ::` ***machine-word***

The smallest machine word, when interpreted as an unsigned integer value.

as-unsigned *Function*

`as-unsigned (t ::` ***type*****`, m ::`** ***machine-word*****`) =>`** ***result***

The value of **`m`** is interpreted as an unsigned value and converted to an instance of **`<abstract-integer>`**, then the result of that conversion is converted to type **`t`** using **`as`**.

6.3.2 Basic and signed single word operations

For all of the following functions, all arguments that are specified as being specialized to **`<machine-word>`** accept an instance of **`<abstract-integer>`**, which is then coerced to a **`<machine-word>`** before performing the operation.

%logior *Function*

%logior (*#rest machine-words*) => (r :: *machine-word*)

%logxor *Function*

%logxor (*#rest machine-words*) => (r :: *machine-word*)

%logand *Function*

%logand (*#rest machine-words*) => (r :: *machine-word*)

%lognot *Function*

%lognot (m :: *machine-word*) => (r :: *machine-word*)

These four functions have the same semantics as **logior**, **logxor**, **logand**, and **lognot** in the Dylan library, but they operate on **<machine-word>**s instead of **<integer>**s.

%logbit? *Function*

%logbit? (index :: *integer*, m :: *machine-word*) => (set? :: *boolean*)

Returns true iff the indexed bit (zero based, counting from the least significant bit) of **m** is set. An error is signaled unless **0 <= index < $machine-word-size**.

%count-low-zeros *Function*

%count-low-zeros (m :: *machine-word*) => (c :: *integer*)

Returns the number of consecutive zero bits in **m** counting from the least significant bit.

Note: This is the position of the least significant non-zero bit in **m**. So if **i** is the result, then **%logbit?(i, m)** is true, and for all values of **j** such that **0 <= j < i**, **%logbit?(j, m)** is false.

%count-high-zeros *Function*

`%count-high-zeros (m ::` ***machine-word*****`) => (c ::`** ***integer*****`)`**

Returns the number of consecutive zero bits in **`m`** counting from the most significant bit.

Note: The position of the most significant non-zero bit in **`m`** can be computed by subtracting this result from **`$machine-word-size - 1`**. So if **`i`** is the result and **`p = ($machine-word-size - i - 1)`**, then **`%logbit?(p, m)`** is true, and for all values of **`j`** such that **`p < j < $machine-word-size`**, **`%logbit?(j, m)`** is false.

%+ *Function*

`%+ (m1 ::` ***machine-word*****`, m2 ::`** ***machine-word*****`) => (sum ::`** ***machine-word*****`, overflow? ::`** ***boolean*****`)`**

Signed addition.

%- *Function*

`%- (m1 ::` ***machine-word*****`, m2 ::`** ***machine-word*****`) => (difference ::`** ***machine-word*****`, overflow? ::`** ***boolean*****`)`**

Signed subtraction.

%* *Function*

`%* (m1 ::` ***machine-word*****`, m2 ::`** ***machine-word*****`) => (low ::`** ***machine-word*****`, high ::`** ***machine-word*****`, overflow? ::`** ***boolean*****`)`**

Signed multiplication. The value of **`overflow?`** is false iff the **`high`** word result is a sign extension of the **`low`** word result.

%floor/ *Function*

`%floor/ (dividend ::` ***machine-word*****`, divisor ::`** ***machine-word*****`) =>`**
`(quotient ::` ***machine-word*****`, remainder ::`** ***machine-word*****`)`**

%ceiling/ *Function*

`%ceiling/ (dividend ::` ***machine-word*****`, divisor ::`** ***machine-word*****`) =>`**
`quotient ::` ***machine-word*****`, remainder ::`** ***machine-word***

%round/ *Function*

`%round/ (dividend ::` ***machine-word*****`, divisor ::`** ***machine-word*****`)=>`**
`(quotient ::` ***machine-word*****`, remainder ::`** ***machine-word*****`)`**

%truncate/ *Function*

`%truncate/ (dividend ::` ***machine-word*****`, divisor ::`** ***machine-word*****`) =>`**
`(quotient ::` ***machine-word*****`, remainder ::`** ***machine-word*****`)`**

%divide *Function*

`%divide (dividend ::` ***machine-word*****`, divisor ::`** ***machine-word*****`) =>`**
`(quotient ::` ***machine-word*****`, remainder ::`** ***machine-word*****`)`**

The functions **`%divide`**, **`%floor/`**, **`%ceiling/`**, **`%round/`**, and **`%truncate/`** all perform signed division of the dividend by the divisor, returning a quotient and remainder such that

```
(quotient * divisor + remainder = dividend)
```

When the division is inexact (in other words, when the remainder is not zero), the kind of rounding depends on the operation:

`%floor/`	The quotient is rounded toward negative infinity.
`%ceiling/`	The quotient is rounded toward positive infinity.
`%round/`	The quotient is rounded toward the nearest integer. If the mathematical quotient is exactly halfway between two integers, then the resulting quotient is rounded to the nearest even integer.

`%truncate/`	The quotient is rounded toward zero.
`%divide`	If both operands are non-negative, then the quotient is rounded toward zero. If either operand is negative, then the direction of rounding is unspecified, as is the sign of the remainder.

For all of these functions, an error is signaled if the value of the divisor is zero or if the correct value for the quotient exceeds the machine word range.

%negative *Function*

`%negative (m ::` ***machine-word*****`) => (r ::`** ***machine-word*****`, overflow? ::`** ***boolean*****`)`**

%abs *Function*

`%abs (m ::` ***machine-word*****`) => (r ::`** ***machine-word*****`, overflow? ::`** ***boolean*****`)`**

%shift-left *Function*

`%shift-left (m ::` ***machine-word*****`, count ::`** ***integer*****`) => (low ::`** ***machine-word*****`, high ::`** ***machine-word*****`, overflow? ::`** ***boolean*****`)`**

Arithmetic left shift of `m` by count. An error is signaled unless **`0 <= count < $machine-word-size`**. The value of **`overflow?`** is false iff the high word result is a sign extension of the low word result.

%shift-right *Function*

`%shift-right (m ::` ***machine-word*****`, count ::`** ***integer*****`) => (r ::`** ***machine-word*****`)`**

Arithmetic right shift of `m` by **`count`**. An error is signaled unless **`0 <= count < $machine-word-size`**.

6.3.3 Overflow signalling operations

For all of the following functions, all arguments that are specified as being specialized to **`<machine-word>`** accept an instance of **`<abstract-integer>`**, which is then coerced to a **`<machine-word>`** before performing the operation.

so%+ *Function*

so%+ (m1 :: *machine-word*, m2 :: *machine-word*) => (sum :: *machine-word*)

Signed addition. An error is signaled on overflow.

so%- *Function*

so%- (m1 :: *machine-word*, m2 :: *machine-word*) => (difference :: *machine-word*)

Signed subtraction. An error is signaled on overflow.

so%* *Function*

so%* (m1 :: *machine-word*, m2 :: *machine-word*) => (product :: *machine-word*)

Signed multiplication. An error is signaled on overflow.

so%negative *Function*

so%negative (m :: *machine-word*) => (r :: *machine-word*)

Negation. An error is signaled on overflow.

so%abs *Function*

so%abs (m :: *machine-word*) => (r :: *machine-word*)

Absolute value. An error is signaled on overflow.

so%shift-left *Function*

so%shift-left (m :: *machine-word*, count :: *integer*) => (r :: *machine-word*)

Arithmetic left shift of **m** by **count**. An error is signaled unless **0 <= count < $machine-word-size**. An error is signaled on overflow.

6.3.4 Signed double word operations

For all of the following functions, all arguments that are specified as being specialized to **`<machine-word>`** accept an instance of **`<abstract-integer>`**, which is then coerced to a **`<machine-word>`** before performing the operation.

d%floor/ *Function*

`d%floor/ (dividend-low ::` ***machine-word*****`, dividend-high ::`** ***machine-word*****`, divisor ::`** ***machine-word*****`) => (quotient ::`** ***machine-word*****`, remainder ::`** ***machine-word*****`)`**

d%ceiling/ *Function*

`d%ceiling/ (dividend-low ::` ***machine-word*****`, dividend-high ::`** ***machine-word*****`, divisor ::`** ***machine-word*****`) => (quotient ::`** ***machine-word*****`, remainder ::`** ***machine-word*****`)`**

d%round/ *Function*

`d%round/ (dividend-low ::` ***machine-word*****`, dividend-high ::`** ***machine-word*****`, divisor ::`** ***machine-word*****`) => (quotient ::`** ***machine-word*****`, remainder ::`** ***machine-word*****`)`**

d%truncate/ *Function*

`d%truncate/ (dividend-low ::` ***machine-word*****`, dividend-high ::`** ***machine-word*****`, divisor ::`** ***machine-word*****`) => (quotient ::`** ***machine-word*****`, remainder ::`** ***machine-word*****`)`**

d%divide *Function*

`d%divide (dividend-low ::` ***machine-word*****`, dividend-high ::`** ***machine-word*****`, divisor ::`** ***machine-word*****`) => (quotient ::`** ***machine-word*****`, remainder ::`** ***machine-word*****`)`**

The functions **`d%divide`**, **`d%floor/`**, **`d%ceiling/`**, **`d%round/`**, and **`d%truncate/`** all perform signed division of the double word dividend by the divisor, returning a quotient and remainder such that

```
(quotient * divisor + remainder = dividend)
```

When the division is inexact (in other words, when the remainder is not zero), the kind of rounding depends on the operation:

`d%floor/`	The quotient is rounded toward negative infinity.
`d%ceiling/`	The quotient is rounded toward positive infinity.
`d%round/`	The quotient is rounded toward the nearest integer. If the mathematical quotient is exactly halfway between two integers then the resulting quotient is rounded to the nearest even integer.
`d%truncate/`	The quotient is rounded toward zero.
`d%divide`	If both operands are non-negative, then the quotient is rounded toward zero. If either operand is negative, then the direction of rounding is unspecified, as is the sign of the remainder.

For all of these functions, an error is signaled if the value of the divisor is zero or if the correct value for the quotient exceeds the machine word range.

6.3.5 Unsigned single word operations

For all of the following functions, all arguments that are specified as being specialized to **`<machine-word>`** accept an instance of **`<abstract-integer>`**, which is then coerced to a **`<machine-word>`** before performing the operation.

u%+ *Function*

`u%+ (m1 ::` ***machine-word*****`, m2 ::`** ***machine-word*****`) => (sum ::`** ***machine-word*****`, carry ::`** ***machine-word*****`)`**

Unsigned addition. The value represented by **`carry`** is either 0 or 1.

u%- *Function*

`u%- (m1 ::` ***machine-word*****`, m2 ::`** ***machine-word*****`) => (difference ::`** ***machine-word*****`, borrow ::`** ***machine-word*****`)`**

Unsigned subtraction. The value represented by **`borrow`** is either 0 or 1.

u%* *Function*

u%* (m1 :: ***machine-word*****, m2 ::** ***machine-word*****) => (low ::** ***machine-word*****, high ::** ***machine-word*****)**

Unsigned multiplication.

u%divide *Function*

u%divide (dividend :: ***machine-word*****, divisor ::** ***machine-word*****) => (quotient ::** ***machine-word*****, remainder ::** ***machine-word*****)**

Performs unsigned division of the dividend by the divisor, returning a quotient and remainder such that

(quotient * divisor + remainder = dividend)

An error is signaled if the value of the **divisor** is zero.

u%rotate-left *Function*

u%rotate-left (m :: ***machine-word*****, count ::** ***integer*****) => (r ::** ***machine-word*****)**

Logical left rotation of **m** by **count**. An error is signaled unless **0 <= count < $machine-word-size**.

u%rotate-right *Function*

u%rotate-right (m :: ***machine-word*****, count ::** ***integer*****) => (r ::** ***machine-word*****)**

Logical right rotation of **m** by **count**. An error is signaled unless **0 <= count < $machine-word-size**.

u%shift-left *Function*

u%shift-left (m :: ***machine-word*****, count ::** ***integer*****) => (r ::** ***machine-word*****)**

Logical left shift of **m** by **count**. An error is signaled unless **0 <= count < $machine-word-size**.

u%shift-right *Function*

u%shift-right (m **::** ***machine-word*****, count ::** ***integer*****) => (r ::** ***machine-word*****)**

Logical right shift of **m** by **count**. An error is signaled unless **0 <= count < $machine-word-size**.

u%< *Function*

u%< (m1 :: ***machine-word*****, m2 ::** ***machine-word*****) => (smaller? ::** ***boolean*****)**

Unsigned comparison.

6.3.6 Unsigned double word operations

For all of the following functions, all arguments that are specified as being specialized to **<machine-word>** accept an instance of **<abstract-integer>**, which is then coerced to a **<machine-word>** before performing the operation.

ud%divide *Function*

ud%divide (dividend-low :: ***machine-word*****, dividend-high ::** ***machine-word*****, divisor ::** ***machine-word*****) => (quotient ::** ***machine-word*****, remainder ::** ***machine-word*****)**

Performs unsigned division of the double word dividend by the **divisor**, returning a **quotient** and **remainder** such that

(quotient * divisor + remainder = dividend)

An error is signaled if the value of the **divisor** is zero or if the correct value for the **quotient** exceeds the machine word range.

ud%shift-left *Function*

ud%shift-left (low :: ***machine-word*****, high ::** ***machine-word*****, count ::** ***integer*****) => (low ::** ***machine-word*****, high ::** ***machine-word*****)**

Logical left shift by **count** of the double word value represented by **low** and **high**. An error is signaled unless **0 <= count < $machine-word-size**.

ud%shift-right *Function*

> **ud%shift-right (low ::** ***machine-word*****, high ::** ***machine-word*****, count ::** ***integer*****) => (low ::** ***machine-word*****, high ::** ***machine-word*****)**

Logical right shift by **`count`** of the double word value represented by **`low`** and **`high`**. An error is signaled unless **`0 <= count < $machine-word-size`**.

7

The Transcendentals Module

7.1 Introduction

The Transcendentals module, exported from the Common Extensions library, provides a set of open generic functions for ANSI C-like behavior over real numbers. The restrictions and error cases described in this chapter are intended to be the same as they are in ANSI C.

The single module, **`transcendentals`**, exports these generic functions and methods.

Because implementation of these functions might be by a standard library for transcendentals accessed by a foreign function interface, the exact precision and algorithms (and hence, the exact results) for all of these functions is explicitly unspecified. Note, however, that a program expects the following, even in libraries that are implemented by calling foreign libraries:

- Domain and range errors should be signalled as Dylan errors.
- Floating point precision *contagion* must obey Dylan rules (that is, functions called on single precision values return single precision results, and functions on double precision values return double precision results)

7.2 The Transcendentals module

This section contains a reference entry for each item exported from the Common Extensions library's **`transcendentals`** module.

^ *G.f. method*

Summary	Returns its first argument, raised to the power indicated by its second argument.	
Signature	`^ b x => y`	
Arguments	*b*	An instance of type `<real>`.
	x	An instance of type `<real>`.
Values	*y*	An instance of type `<float>`.

Description

Returns *b* raised to the power *x*. If *b* is `0` and *x* is not positive, an error is signalled. If *b* is negative and *x* is not an integer, an error is signalled.

If *b* and *x* are both integers, the result is an integer. If *x* is negative, an error is signalled.

The floating point precision is given by the precision of *b*. The result is a single-float if *b* is an integer.

See also

See the function **`exp`**, page 164.

acos *G.f. method*

Summary	Returns the arc cosine of its argument.	
Signature	`acos x => y`	
Arguments	*x*	An instance of type `<real>`. The angle, in radians. If *x* is not in the range `[-1,+1]`, an error is signalled.
Values	*y*	An instance of type `<float>`.

Description — Returns the arc cosine of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

See also — See the functions **`asin`**, page 159 and **`atan`**, page 160.

acosh *G.f. method*

Summary — Returns the hyperbolic arc cosine of its argument.

Signature — **`acosh`** ***`x`*** **`=>`** ***`y`***

Arguments	*x*	An instance of type **`<real>`**. The angle, in radians.
Values	*y*	An instance of type **`<float>`**.

Description — Returns the hyperbolic arc cosine of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

See also — See the functions **`asinh`**, page 160 and **`atanh`**, page 161.

asin *G.f. method*

Summary — Returns the arc sine of its argument.

Signature — **`asin`** ***`x`*** **`=>`** ***`y`***

Arguments	*x*	An instance of type **`<real>`**. The angle, in radians. If *x* is not in the range **`[-1,+1]`**, an error is signalled.
Values	*y*	An instance of type **`<float>`**.

Description	Returns the arc sine of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.
See also	See the functions **`acos`**, page 158 and **`atan`**, page 160.

asinh *G.f. method*

Summary	Returns the hyperbolic arc sine of its argument.	
Signature	**`asinh`** ***`x`*** **`=>`** ***`y`***	
Arguments	*x*	An instance of type **`<real>`**. The angle, in radians.
Values	*y*	An instance of type **`<float>`**.
Description	Returns the hyperbolic arc sine of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.	
See also	See the functions **`acosh`**, page 159 and **`atanh`**, page 161.	

atan *G.f. method*

Summary	Returns the arc tangent of its argument.	
Signature	**`atan`** ***`x`*** **`=>`** ***`y`***	
Arguments	*x*	An instance of type **`<real>`**. The angle, in radians. If *x* is not in the range **`[-1,+1]`**, an error is signalled.
Values	*y*	An instance of type **`<float>`**.

Description — Returns the arc tangent of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

See also — See the functions **`acos`**, page 158 and **`asin`**, page 159.

atan2 *G.f. method*

Summary — Returns the arc tangent of one angle divided by another.

Signature — **`atan2`** ***`x y`*** **`=>`** ***`z`***

Arguments	*x*	An instance of type **`<real>`**. The first angle, in radians.
	y	An instance of type **`<real>`**. The second angle, in radians.
Values	*z*	An instance of type **`<float>`**.

Description — Returns the arc tangent of *x* divided by *y*. x may be zero if y is not zero. The signs of x and y are used to derive what quadrant the angle falls in.

The floating point precision of the result is given by the precision of *x* / y. The result is a single-float if *x/y* is an integer.

atanh *G.f. method*

Summary — Returns the hyperbolic arc tangent of its argument.

Signature — **`atanh`** ***`x`*** **`=>`** ***`y`***

Arguments	*x*	An instance of type **`<real>`**. The angle, in radians.

Values	*y*	An instance of type **`<float>`**.

Description — Returns the hyperbolic arc tangent of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

See also — See the functions **`acosh`**, page 159 and **`asinh`**, page 160.

cos *G.f. method*

Summary — Returns the cosine of its argument.

Signature — **`cos`** ***x*** **`=>`** ***y***

Arguments	*x*	An instance of type **`<real>`**. The angle, in radians.
Values	*y*	An instance of type **`<float>`**.

Description — Returns the cosine of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

See also — See the functions **`sin`**, page 165 and **`tan`**, page 168.

cosh *G.f. method*

Summary — Returns the hyperbolic cosine of its argument.

Signature — **`cosh`** ***x*** **`=>`** ***y***

Arguments	*x*	An instance of type **`<real>`**. The angle, in radians.
Values	*y*	An instance of type **`<float>`**.

Description Returns the hyperbolic cosine of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

See also See the functions **`sinh`**, page 167 and **`tanh`**, page 168.

$double-e *Constant*

Summary The value of **`e`**, the base of natural logarithms, as a double precision floating point number.

Type **`<double-float>`**

Superclass **`<float>`**

Description The value of **`e`**, the base of natural logarithms, as a double precision floating point number.

See also See the constant **`$single-e`**, page 166.

$double-pi *Constant*

Summary The value of π as a double precision floating point number.

Type **`<double-float>`**

Superclass **`<float>`**

Description The value of π as a double precision floating point number.

See also See the constant **`$single-pi`**, page 166.

exp *G.f. method*

Summary	Returns **`e`**, the base of natural logarithms, raised to the power indicated by its argument.	
Signature	**`exp`** ***`x`*** **`=>`** ***`y`***	
Arguments	*x*	An instance of type **`<real>`**.
Values	*y*	An instance of type **`<float>`**.
Description	Returns **`e`**, the base of natural logarithms, raised to the power *x*. The floating point precision is given by the precision of *x*.	
See also	See the functions **`^`**, page 158 and **`log`**, page 164.	

isqrt *G.f. method*

Summary	Returns the integer square root of its argument.	
Signature	**`isqrt`** ***`x`*** **`=>`** ***`y`***	
Arguments	*x*	An instance of type **`<integer>`**.
Values	*y*	An instance of type **`<integer>`**.
Description	Returns the integer square root of *x*, that is the greatest integer less than or equal to the exact positive square root of *x*. If *x* < **`0`**, an error is signalled.	
See also	See the function **`sqrt`**, page 167.	

log *G.f. method*

Summary	Returns the natural logarithm of its argument.

Signature **`log`** *x* **`=>`** *y*

Arguments	*x*	An instance of type **`<real>.`**
Values	*y*	An instance of type **`<float>`**.

Description Returns the natural logarithm of *x* to the base e. If x <= 0 <= 1, an error is signalled. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

See also See also **`exp`**, page 164, and **`logn`**, page 165.

logn *G.f. method*

Summary Returns the logarithm of its argument to the given base.

Signature logn number, base

Arguments	number	
	base	A number greater than **`1`**.

Description Returns the logarithm of *number* to the base *base*. If x <= 0 <= 1, an error is signalled. The floating point precision of the result is given by the precision of *number*. The result is a single-float if *number* is an integer.

See also See also **`log`**, page 164, and **`exp`**, page 164.

sin *G.f. method*

Summary Returns the sine of its argument.

Signature **`sin`** *x* **`=>`** *y*

Arguments	*x*	An instance of type **`<real>`**. The angle, in radians.
Values	*y*	An instance of type **`<float>`**.

Description — Returns the sine of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

See also — See the functions **`cos`**, page 162 and **`tan`**, page 168.

$single-e *Constant*

Summary — The value of **`e`**, the base of natural logarithms, as a single precision floating point number.

Type — **`<single-float>`**

Superclass — **`<float>`**

Description — The value of **`e`**, the base of natural logarithms, as a single precision floating point number.

See also — See the constant **`$double-e`**, page 163.

$single-pi *Constant*

Summary — The value of π as a single precision floating point number.

Type — **`<single-float>`**

Superclass — **`<float>`**

Description — The value of π as a single precision floating point number.

See also See the constant **`$double-pi`**, page 163.

sinh *G.f. method*

Summary Returns the hyperbolic sine of its argument.

Signature **`sinh`** ***x*** **`=>`** ***y***

Arguments *x* An instance of type **`<real>`**. The angle, in radians.

Values *y* An instance of type **`<float>`**.

Description Returns the hyperbolic sine of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

See also See the functions **`cosh`**, page 162 and **`tanh`**, page 168.

sqrt *G.f. method*

Summary Returns the square root of its argument.

Signature **`sqrt`** ***x*** **`=>`** ***y***

Arguments *x* An instance of type **`<real>`**. The angle, in radians.

Values *y* An instance of type **`<float>`**.

Description Returns the square root of x. If x is less than zero an error is signalled. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

See also — See the function **`isqrt`**, page 164.

tan *G.f. method*

Summary — Returns the tangent of its argument.

Signature — **`tan`** ***x*** **`=>`** ***y***

Arguments	*x*	An instance of type **`<real>`**. The angle, in radians.
Values	*y*	An instance of type **`<float>`**.

Description — Returns the tangent of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

tanh *G.f. method*

Summary — Returns the hyperbolic tangent of its argument.

Signature — **`tanh`** ***x*** **`=>`** ***y***

Arguments	*x*	An instance of type **`<real>`**. The angle, in radians.
Values	*y*	An instance of type **`<float>`**.

Description — Returns the hyperbolic tangent of its argument. The floating point precision of the result is given by the precision of *x*. The result is a single-float if *x* is an integer.

See also — See the functions **`cosh`**, page 162 and **`sinh`**, page 167.

8

Library Interchange

8.1 Introduction

This chapter is about the library interchange definition format, *LID*.

The DRM defines an interchange format for Dylan source files (DRM, page 21), but does not define an interchange format for Dylan libraries. Without such an agreed format, different Dylan development environments would find it difficult to import and build libraries developed using another Dylan vendor's environment. It would also be impossible to automate the process of importing a library into another environment.

LID solves this problem. It allows you to describe Dylan library sources in a form that any Dylan environment should be able to understand. Harlequin and other Dylan vendors have adopted LID to make it easier to port applications from one environment to another.

Note: LID is a convention, and not an extension to the Dylan language.

Note: The Harlequin Dylan environment can convert LID files to its own internal project file format, the **.hdp** file. It can also save project files as LID files with the **File > Save As** command in the project window.

8.2 LID files

LID works by supplementing each set of library sources with a LID file. A *LID file* describes a Dylan library using a set of keyword statements. Together, these statements provide enough information for specifying and locating the information necessary to build a library from its source. This means all Dylan libraries designed for interchange consist of at least two files: a LID file, and one or more files containing the library source code.

Thus a LID file performs a similar function to the **`makefile`** used in some C and C++ development environments.

LID files have the file extension **`.lid`**.

Every file referred to by a LID file must reside in the same folder (directory) as the LID file.

8.3 LID keyword statements

A LID file consists of a series of keyword / value statements, just like the Dylan source file interchange format described in DRM (page 21.) In this section, we describe the standard LID keywords.

Library:

LID file keyword

`Library:` ***library-name***

Names the Dylan library being described. The *library-name* must be the name of the Dylan library that the LID file describes. This keyword is required in every LID file, and it may appear only once per LID file.

Files:

LID file keyword

`Files:` ***file-designators***

Associates a set of files with the library named by the **`Library:`** keyword. This keyword can appear one or more times per LID file. Every file specified is considered to be associated with the library.

A file designator is something that can be mapped to a concrete file name. Only one file designator can appear per line of the LID file. See

Section 8.5 on page 175 for a description of the format of file designators and how they are mapped to concrete file names.

The order in which the designated source files are specified with the `Files:` keyword in the LID file determines the initialization order across the files within the defined library.

All the files specified must reside in the same folder (directory) as the LID file.

Synopsis: *LID file keyword*

`Synopsis:` ***arbitrary text***

A concise description of the library.

Keywords: *LID file keyword*

`Keywords:` ***comma-separated phrases***

Any number of phrases, separated by commas, that are relevant to the description or use of the library.

Author: *LID file keyword*

`Author:` ***arbitrary text***

The name of the library's author.

Version: *LID file keyword*

`Version:` ***arbitrary text***

The current version number of the library.

Description: *LID file keyword*

`Description:` ***arbitrary text***

A description of the library. The intention of this keyword is to provide a fuller, less concise description than that given by the **`Synopsis:`** keyword.

Comment: *LID file keyword*

`Comment:` ***arbitrary text***

Any additional comments about the library.

8.4 Common Dylan's LID extensions

This section contains extensions to LID that Common Dylan supports.

8.4.1 Specifying foreign files and resource files

The following keywords allow you to specify that files of foreign source code and resource files are a part of the library.

C-Source-Files: *LID file keyword*

`C-Source-Files:` ***c-source-files***

Identifies one or more C source files which are to be included as part of the library. Dylan environments copy these files to their build area and ensure that they are compiled by the appropriate batch file. The filenames specified must include the **`.c`** suffix.

C-Header-Files: *LID file keyword*

`C-Header-Files:` ***c-header-files***

Identifies one or more C header files included as part of the library. Dylan environments copy these files to their build area and ensure that they are compiled by the appropriate batch file. Any files specified using the **`C-Source-Files:`** or **`RC-Files:`** keywords depend on these header files in order to decide when they need to be recompiled. The file names given here must include the **`.h`** suffix.

C-Object-Files: *LID file keyword*

`C-Object-Files:` ***c-object-files***

Identifies one or more C object files included as part of the library. Dylan environments copy these files to their build area and ensure that they are compiled by the appropriate batch file and included in the final output as **`.DLL`** or **`.EXE`** files. The file names given here must include the **`.obj`** suffix.

RC-Files: *LID file keyword*

`RC-Files:` ***resource-files***

Identifies one or more resource files to be included as part of the library. Dylan environments copy these files to their build area and ensure that they are compiled by the appropriate batch file. The resulting resource object files are included in the **`.DLL`** or **`.EXE`** built for the library. The file names given here must include the **`.rc`** suffix.

C-Libraries: *LID file keyword*

`C-Libraries:` ***c-lib-files***

Identifies one or more C libraries to be included in the link phase when building the **`.DLL`** or **`.EXE`** for the library. You can use this keyword to specify arbitrary linker options as well as libraries.

Unlike the other keywords described in this section, the **`C-Libraries:`** keyword propagates to dependent libraries. For example, suppose library A uses library B, and the LID file or library B specifies

```
C-Libraries: foo.lib
```

In this case, both library A and library B are linked against **`foo.lib`**.

8.4.2 Specifying compilation details

The following keywords control aspects of compilation for the library.

Executable: *LID keyword*

`Executable:` ***name***

Specifies the name of the executable (that is, **`.DLL`** or **`.EXE`**) file to be generated for this library.

The suffix (**`.DLL`**, **`.EXE`**) should not be included in the *name* as the appropriate suffix will be added automatically.

If this keyword is not specified, the compiler generates a default name for the executable from the name of the library. With some library names, particularly when you are building a DLL, you may need to specify this keyword to override the default name and avoid conflicts with other DLLs from a third party.

Base-Address: *LID keyword*

`Base-Address:` ***address***

Specifies the base address of the DLL built from this Dylan library. The *address* must be a hexadecimal value. For convenience, you can use either Dylan (**`#xNNNNNNNN`**) or C (**`0xNNNNNNNN`**) notations when specifying the address.

This base address is ignored when building a **`.EXE`** file.

If this keyword is not specified, the compiler will compute a default base address for the library. However, it is possible for more than one library to end up with the same default base address. If an application uses any of these libraries, all but one of them will have to be relocated when the application starts. This process is automatic, but cuts down on the amount of sharing, increases your application's memory footprint, and slows down load time. In such circumstances, you may want to give one or more libraries an explicit base address using this keyword.

Linker-Options: *LID keyword*

`Linker-Options:` ***options***

Specifies additional options and libraries to be passed to the linker when building this DLL or EXE. Unlike the C-Libraries: keyword, the options and libraries specified here apply only to this Dylan library; they are not propagated to any libraries which use this library.

8.5 File naming conventions

In practice, importing a source distribution into a Dylan program involves unpacking the source distribution into its own subtree and then informing the environment of the location of the tree root. The environment then walks the entire subtree locating LID files, which describe the libraries in the distribution by giving a name to, and designating the source files associated with, each library.

Importing a Dylan program into the environment in this way requires two things:

1. That the LID files in the distribution can be identified.
2. That the file designators supplied to the **`Files:`** keyword in LID files can be mapped to the corresponding source filenames on disk.

If you are importing files from a platform that does not insist on, or conventionally use, standard filename suffixes to identify the filetype (such as MacOS), then you must rename your source files as follows:

- LID files must be given filenames with the suffix **`.lid`**.
- Dylan source files must be given filenames with the suffix **`.dylan`** or **`.dyl`**.

The file designators that appear in LID files may be a string of characters of any length, constructed from the set of hyphen, underscore, and the mixed-case alphanumeric characters. Note that you do not have to specify the source filename suffix as part of the filename designator. This ensures that the LID files themselves do not need to be edited when importing source code from a platform, such as MacOS, that does not insist on particular filename suffixes to specify the file type.

The name of a LID file is not significant, and in particular need not be the same as the library name. Hierarchical directory structure can be used to organize multi-library systems as long as the files directly associated with each library are in a single directory.

8.6 Application example

This section contains an example of a complete Dylan application that uses a generic factorial calculation routine to return the value of the factorial of 100. Two libraries are defined: the **`factorial`** library provides an implementation of the generic factorial routine, and the **`factorial-application`** library provides a method that calls the generic routine and returns the appropriate result.

File: **`fact.lid`**. LID file describing the components of the **`factorial`** library.

```
Library:  factorial
Synopsis: Provides a naive implementation of the factorial
          function
Keywords: factorial, integer, simple, recursive
Files:    fact-def
          fact
```

File: **`fact-def.dyl`**. Defines the **`factorial`** library and its one module.

```
Module: dylan-user

define library factorial
  use dylan;
  export factorial;
end;

define module factorial
  export fact;
end;
```

File: **fact.dyl**. Defines the method for calculating a factorial.

```
Module: factorial

define generic fact(n);

define method fact(n == 0)
  1;
end;

define method fact(n)
  n * fact(n - 1);
end;
```

File: **app.lid**. LID file describing the components of the **factorial-application** library.

```
Library:        factorial-application
Synopsis:       Computes factorial 100
Files:          appdef
                app
Start-Module:   factorial-application
Start-Function: main
```

File: **appdef.dyl**. Defines the **factorial-application** library and its one module.

```
Module: dylan-user

define library factorial-application
  use dylan;
  use factorial;
end library;

define module factorial-application
  use dylan;
  use factorial;
end module;
```

File: **app.dyl**. Defines a routine that calls the factorial routine.

```
Module: factorial-application

define method main (#rest ignore)
  fact(100);
end method;
```

The following example demonstrates how files of foreign source code and resource files can be integrated into a Dylan library:

```
Library:         app-with-foreign-code
Synopsis:        Uses some C code and resources
Files:           dylan-code
C-Source-Files: first.c
                 second.c
C-Header-Files: headers.h
RC-Files:        extra-resources.rc
```

Index