The Mindy Compiler and Interpreter

Copyright (c) 1994 Carnegie Mellon University All rights reserved. Refer to the end of this document for precise terms of use.

The Gwydion Project would like to thank those on the net that have contributed code patches and bug reports for Mindy:

Adam Alpern, Patrick Beard, Steve Strassman, Scott Collins, Ed Gamble, Bruno Haible, John Shen, Galen Hunt, Richard Lynch, Dan Ratner, Court Demas, Miles Bader, Kelly Murray, Nick Thompson, Brent Benson, Brian Rogoff

Special thanks for a major effort to Roger Critchlow for enhancements to Mindy.

1. Introduction

Mindy is an implementation of a language that is very much like the language described in the *Dylan™ Interim Reference Manual* (DIRM). The name *Mindy* is derived from "Mindy Is Not Dylan Yet", and as the name implies, Mindy is incomplete. Mindy is incomplete for the following reasons:

- We do not implement everything in the DIRM.
- The DIRM does not specify all that Apple™ intends Dylan to be.
- There's no way to validate what a Dylan implementation is, even if we had a full specification.

However, Mindy does implement most of what we believe Dylan will be.

Mindy was developed by the Gwydion Project at Carnegie Mellon University for our own internal use as a development tool while we work on our *real* high-performance Dylan implementation. We have decided to make Mindy available for other people who want to learn about Dylan. However, the amount of effort that we can put into maintaining Mindy is strictly limited.

Mindy will never be an industrial-strength implementation, and nobody should depend on it for real work. We will make future releases from time to time as we add new features and fix bugs, but this is strictly a sideshow for us. We would appreciate receiving bug reports (especially those accompanied by code patches) and suggestions for improvements, but we may not fix every bug reported in a timely manner, or fix it at all. Our work on development of the *real* Gwydion/Dylan must take precedence.

We hope that nobody will draw any conclusions about the performance of our future Gwydion/Dylan compiler or the performance attainable in Dylan from experience using Mindy. It was not designed to be fast.

Mindy comprises two C programs, a compiler that produces byte-codes and a byte-code interpreter. Instructions for compiling and installing Mindy can be found in the file `INSTALL` at the top level of the Mindy release. We have built and tested Mindy under MACH on the DECstation and HP-UX on HP 700's. We have built and run Mindy, but not tested it extensively, under OSF1 on the Alpha and Irix on the SGI.

2. Hello, World

Well, the first program anyone should endeavor to write in a new language is, of course, *Hello World*. Type this into a file called `hw.dylan`:

```
module: dylan-user

define method main (invocaton-name, #rest ignore)
  puts("Hello, World.\n");
end;
```

To compile your program invoke `$INSTALL/bin/mindycomp`, for example:

```
% $INSTALL/bin/mindycomp hw.dylan
```

This produces a file named `hw.dbc`. The `.dbc` stands for "Dylan Byte Code". To run the program, say:

```
% $INSTALL/bin/mindy -f hw.dbc
```

It should print "`Hello, World.`" to standard output and then exit.

3. The Main Routine

After loading your program, Mindy invokes the generic function **main** from the `Extensions` module of the `Dylan` library. Your program must define a method for **main**, or Mindy will signal a *no applicable methods* error and put you in the debugger. For more information on the **main** function, see Section *The Extensions Module*.

It can be useful to load code into Mindy with no main method. Once you land in the debugger, you can call any function manually. This provides a way to test any library.

4. Multiple Files

When working with a larger program, you will probably have more than one `.dylan` file. In which case, you just compile them each independently, and then run Mindy with multiple `-f` switches:

```
% mindy -f foo.dbc -f bar.dbc -f baz.dbc
```

Mindy loads the files specified with the `-f` switches in the order you specify the files on the command line. This becomes important when you define your own modules (see Section *Libraries and Modules*).

If you typically load several `.dbc` files as part of a single program, you can combine them into one file for convenience. The mechanism for combining `.dbc` files is the Unix™ cat utility:

```
% cat foo.dbc bar.dbc baz.dbc > big.dbc
% mindy -f big.dbc
```

5. Syntax Errors

If there are any syntax errors in your program, mindycomp will report them to stderr while compiling. For example, if you had left off the closing parenthesis in the call to **puts** in the above example, mindycomp would have reported:

```
hw.dylan:4: parse error at or before `;'
```

Because the line introduction, `hw.dylan:4:`,has the same format that the C compiler uses, gnu-emacs's compile package can parse the error messages from mindycomp.

Mindycomp's error recovery is not the best in the world. Often, it has to completely punt, telling you only about the first few errors it found. You have to fix what it reports and try again.

A hint to getting slightly tighter error recovery is to end all method and class definitions with "`end method;`" or "`end class;`". For example, if you forget an `end` token for a statement inside a method definition, the mindycomp parser goes all the way to the end of the file and then reports a syntax error at the EOF position. You do not get any more clues. If you use "`end method;`", then the parser can recover at the end of the method containing the bad syntax and reports the syntax error there. This gives you a lot tighter recovery and more information in this situation.

6. Runtime Errors

Much more common than syntax errors are runtime errors. And given the simplistic model of compilation mindycomp uses, most semantic errors are not detected until runtime. When Mindy hits a runtime error that is not handled via the condition system, it drops you into a debugger. From this debugger you can look at variables, examine the stack, and invoke functions. For example, if you had assumed that **puts** would be named something more reasonable, like **put-string**, you would have gotten the following when you tried to run your *Hello World* program:

```
% mindy -f hw.dbc

Warning: the following variables are undefined:
  in library Dylan-user:
    in module Dylan-user:
      put-string[hw.dylan, line 9]

thread [0] D   main
fp 0x1003009c: invoke-debugger({<simple-error> 0x101a24c9})
mindy>
```

Typing `help` at the `mindy>` prompt will list the various commands you can use. See the document `debug.ps` for more information.

7. Internal Lossage

Sometimes mindycomp or Mindy will get an internal error. When this happens, it will print a message to stderr and then abort. This results in the process dying due to some kind of signal. On the pmax, this signal is `SIGILL`, or Illegal Instruction. When this happens, send gwydion-bugs@cs.cmu.edu a piece of mail containing the error message and information on what it was you did that triggered the problem.

8. Dylan vs. Mindy Language Issues

The Dylan language is still changing slightly. Mindy implements most of the *Dylan Interim Reference Manual*, some features that have been accepted but not described in the DIRM, and some features we would like to see accepted before the official language specification is published. Currently, the Mindy diverges from the DIRM as described below:

Additions:

- Mindy supports multiple value binding in the =/then clauses of for statements. This official status of this feature is uncertain at this time. The format of such a clause is
  ```
  (var1, var2, ...) = expr1 THEN expr2
  ```
- Mindy supports *keyed-by* clauses in for statements. The format of such a clause is
  ```
  var KEYED-BY key IN collection
  ```
  `Var` is bound to each element in `collection`, and `key` is bound to the element's key value.
- Keyword parameters may have type information, and you can specify default values with either Harlequin's proposed syntax (which uses "`= ...`") or with Apple's syntax (which uses "`(...)`"). Mindy does not enforce any congruence rules for keyword parameter types, so effectively, keyword type information in generic function declarations serves as documentation only.
- Mindy supports subclass specializers via the **limited** function. A subclass specializer causes a method to be invoked whenever the generic function was called on a value that is the specified class or any subclass of the specified class. The method is never invoked on a value that is an instance (direct or indirect) of the specified class, only when the value is a subclass of the specified class. The following is an example:

```
define method make
    (result-class :: limited(<class>, subclass-of(<my-class>)));
  let x = next-method();
  do-special-logging-or-something(x);
  x;
end method;
```

Deficiencies:

- Sealed/open, abstract/concrete, and primary keywords are parsed where allowed, but Mindy ignores this information about your program.
- Mindy does not parse the `seal generic` form.
- Library `use` chains cannot be circular; that is, there can be no cycles in the graph representing library `use` relationships.
- Define method does not automatically insert `#next next-method` in parameter lists. You have to explicitly add it yourself.
- `Case` and `select` bodies cannot be empty.
- `Make(<class>, ...)` is unsupported.

9. Built-in Libraries and Modules

Mindy has full support for modules and libraries. Mindy provides two built-in libraries, `Dylan` and `Dylan-user`. The `Dylan` library contains the Dylan language implementation and the following exported modules:

`Dylan`
: This module contains the Dylan language implementation and exports all the built-in Dylan definitions.

`Extensions`
: This module exports useful extensions to the Dylan language (see Section *The Extensions Module*). Ultimately, there will be several, more logically separate libraries that extend Dylan or provide an application framework for users. For now, we put any commonly used utilities in the `Extensions` module.

`System`
: This module exports an interface to operating system calls and special, low-level functionality (see Section *The System Module*).

`Introspection`
: This module exports reflective operations for examining classes, functions, and so on.

`File-descriptors`
: This module exports an interface to most standard C system calls that operate on file descriptors.

`Cheap-io`
: This module exports some basic, unextendable input and output functionality.

`Threads`
: This module exports an interface to threads, locks, and objects that behave similarly to `cthreads.h` condition variables.

The `Dylan-user` library is the default library in which mindycomp compiles user code. Mindy provides this library for user convenience when whipping up play code or small applications for which the programmer does not want to bother to create a library. You cannot redefine the `Dylan-user` library. This library contains one module, `Dylan-user`, and you cannot redefine this module.

The Dylan language requires every library to contain a `Dylan-user` module, and this module must use the `Dylan` module from the `Dylan` library regardless of any user specifications. This module provides a starting point in every library where users can begin to define modules; without an initial module in the library, you would be unable to write any code,

including module definitions. Each `Dylan-user` module in Mindy also automatically uses the modules from the `Dylan` library described above. You cannot redefine the `Dylan-user` module, so if your code requires module other than those described above, then you must define your own library and module.

Other libraries are available to Mindy users. Later sections of this document describe these libraries.

10. Using Libraries and Modules

To compile code into a particular library use the `-l` switch to mindycomp:

```
% mindycomp -lmy-lib foo.dylan
```

If there is no `-l` switch, then mindycomp compiles the code into the `Dylan-user` library. When loading a `.dbc` file into Mindy that was compiled into a particular library, one of the following conditions must be satisfied to avoid errors:

- The library must be the `Dylan-user` library. Technically, you could put code in the `Dylan` library, but do not do this.
- You must have defined the library in a file previously loaded (see Section *Multiple Files* for information on loading multiple files).
- The first piece of code in the source file that produced the `.dbc` file must be the library definition.

While loading a file, if Mindy processes a library definition that uses an undefined library, then Mindy stops loading the current file, searches for the undefined library, and loads it. After loading the undefined library, Mindy continues loading the current file and processing the original library definition. Mindy searches for the undefined library in the directories listed in the `MINDYPATH` environment variable. If `MINDYPATH` is undefined, then Mindy uses the pathname `$INSTALL/lib`. In each directory, Mindy first looks for the file `<library>-lib.dbc`, where `<library>` is the name of the undefined library, and if this file does not exist, then Mindy looks for `<library>.dbc`.

Mindy loads the `Dylan` library when it first sees a reference to it. A reference to the `Dylan` library occurs when loading a file compiled to be in the `Dylan` library, or when loading a file with a library definition that uses the `Dylan` library. Mindy loads the `Dylan` library by looking for the file `dylan.dbc` on `MINDYPATH`.

To make a single compiled file for a library which has multiple source files, compile all the files that constitute the library with the `-l` switch set to the library's name. Then `cat` all the resulting `.dbc` files together (see Section *Multiple Files*), making sure the file that defines the library is first. Then install the combined `.dbc` file in one of the directories in your `MINDYPATH`.

To compile code into a particular module, use the `module:` file header. Whenever a source file lacks a `module:` file header, mindycomp issues a compiler warning and compiles the code into the `Dylan-user` module. Note, this is the `Dylan-user` module of the library specified with the `-l` switch, and if there was no `-l` switch, it is the `Dylan-user` module of the `Dylan-user` library.

When loading a `.dbc` file into Mindy that was compiled into a particular module, one of the following conditions must be satisfied to avoid errors:

- The module must be the `Dylan-user` module.
- You must have defined the module in a file previously loaded (see Section *Multiple Files* for information on loading multiple files).
- The first code in the source file that produced the `.dbc` file must be library and module definitions, and one of the module definitions must be the module in question.

11. The Extensions Module

Ultimately, there will be several, more logically separate libraries that extend Dylan or provide an application framework for users. For now, we put any commonly used utilities in the `Extensions` module.

11.1. Generally Useful Definitions

The `Extensions` module exports the following generally useful functionality:

<boolean> [Class]

This class is a subclass of **<object>**. There are exactly two instances of this class, `#t` and `#f`.

<byte-vector> [Class]

This class is a subclass of **<vector>** that can only hold integers between 0 and 255 inclusively. This class is a temporary addition to Mindy to support the requirement that the `Streams` library export a **<byte-vector>** definition. When Mindy supports limited collections, this may be defined within the `Streams` library.

<byte-character> [Class]

This class is a subclass of **<character>**. Characters of this type represent the ASCII character set (or extensions to ASCII). Note, in Mindy the **<character>** class is equivalent to unicode characters.

main [Generic Function]

Arguments

invocation-name :: <byte-string>
#rest arguments

Description

Has no methods, but is called by Mindy when it starts up. To make a standalone program, you define a method on main that does whatever you want it to do. *Invocation-name* is the first token on the command line that invoked Mindy. *Arguments* is a sequence of strings. There is a string in *arguments* for every argument on the command line that invoked Mindy, except all `-f` switches and the argument following each `-f` switch (that is, the file to load) is missing. Remember that any module that adds a method to main must use the `Extensions` module from the `Dylan` library.

one-of [Constant]

This function is useful in type expressions. Because the union function can only take two arguments, any type that is an enumeration of three or more **singleton** values requires cascading calls to **union**. Combine that with having to wrap each value in a call to **singleton**, and using **union** starts to create a lot of parameter list bloat. For example, the expression

```
one-of(#"foo", #"bar", #"baz")
```

is equivalent to

```
union(singleton(#"foo"), union(singleton(#"bar"), singleton("baz")))
```

type-or [Constant]

This function is useful in type expressions. Because the **union** function can only take two arguments, any type that is the union of three or more types requires cascading calls to **union**. Using **type-or** can be more convenient and more clear to read. For example, the expression

```
type-or(<foo>, <bar>, <baz>, <quux>)
```

is equivalent to

```
union(<foo>, union(<bar>, union(<baz>, <quux>)))
```

false-or [Constant]

This function is useful in type expressions. It captures the common idiom of returning an instance of a particular type or the value `#f`. The expression

```
false-or(<integer>)
```

is equivalent to the expression

```
union(<integer>, singleton(#f))
```

11.2. Debugger Customizations

The debugger uses the function **report-condition** to print conditions as error messages to users; for example, this is the function that implements the `%S` format-string directive for conditions. The debugger also uses the **format** function exported from the `Cheap-io` module to process format strings, and it prints directly to the Unix stdout passed to the Mindy process. If any library that you load into Mindy uses the `Debugger-format` library, then the debugger uses **format** from the `Format` library. You can extend how Mindy prints conditions, change what formatting function it uses, and direct where debugger output goes with the following:

report-condition [Generic Function]

Arguments

condition :: **<condition>**

stream :: **<stream>**

Values

meaningless :: singleton(#f)

Description

This is the function Mindy uses to print condition variables as error messages to users. The internal **format** function used by Mindy uses **report-condition** for condition arguments to the `%S` format directive. The `Format` library's **print-message** method for conditions calls **report-condition**.

If you are writing a module that does no output but still provides **report-condition** methods, you should consider using the value of ***format-function*** to format output. Using ***format-function*** makes your module more flexible for users of your module. If you call Mindy's internal **format**, you'll be forced to write to only one destination, Mindy's stdout, ignoring the *stream* argument. If you call the `Format` library's **format** function, then your module will require the `Format`, `Print`, and `Streams` libraries; therefore, users of your module may ultimately load these other libraries needlessly. Of course, if you want to make use of the extended functionality of the `Format` library's format control strings, then you only have one choice anyway, and there's no reason to use ***format-function***.

format-function [Variable]

This variable holds a function that takes a stream, format string, and format arguments. This function should force output when it is done. The default value of this variable is a function that ignores the stream argument and uses the **format** function from the `Cheap-io` module.

The `Debugger-format` library replaces the value of ***format-function*** with a function that calls **format** from the `Format` library on the stream argument, and then calls the `Stream` library's **force-output** function on the stream argument.

debug-output [Variable]

The debugger uses the value of this variable when performing output. The value must be a **<stream>** from the `Streams` library, or `#f` (the default). When this variable is `#f`, the debugger outputs to stderr.

11.3. Tables

The `Extensions` module exports the following **<table>** subclasses:

<equal-table> [Class]

This class is a subclass of **<table>** that uses the **\=** function to compare keys and the **equal-hash** function to generate hash codes. If you define your own classes and **\=** methods specialized on those classes, then you

should define a method for the **equal-hash** function specialized to your classes (see the **equal-hash** function description).

<value-table> [Abstract Class]

This class is a subclass of **<table>**. Users can define subclasses of this class and provide a method for **table-protocol** that is specialized to their new subclass. Any subclass of **<value-table>** must use a hash function that never use an object's identity (that is, its location in the heap) as a means of computing a hash ID. These tables are specifically designed to save overhead in testing hash states and whether the table needs to be rehashed after garbage collections. The second value of the hash function should always be **$permanent-hash-state**. For example:

```
define class <my-table> (<value-table>)
end class;

define method table-protocol (table :: <my-table>)
  values(\=, string-hash);
end method;
```

The `Extensions` module exports the following functions to make it easier for users to use **<equal-table>**s and **<value-table>**s:

equal-hash [Generic Function]

Arguments

key :: <object>

Values

hash-id :: <integer>

hash-state :: <object>

Description

This function returns a hash ID and hash state for use with **<equal-table>**s. If you define your own classes and **\=** methods specialized on those classes, then you should define a method for the **equal-hash** function specialized to your classes. Specialized methods exist for **<number>**, **<character>**, **<function>**, **<symbol>**, and **<collection>**. The method for **<object>** returns the integer 42 and **$permanent-hash-state**. This function may use an object's identity (that is, its location in the heap) to produce a hash ID.

collection-hash [Function]

Arguments

collection :: <collection>

key-hash-function :: <function>

elt-hash-function :: <function>)

Values

hash-id :: <integer>

hash-state :: <object>

Description

This function hashes every element of *collection* using *key-hash-function* on the keys and *element-hash-function* on the elements. Note, though two sequences may be equal according to the **\=** function, **sequence-hash** and **collection-hash** may return different hash codes for the sequences.

sequence-hash [Function]

Arguments

sequence :: <sequence>
elt-hash-function :: <function>

Values

hash-id :: <integer>
hash-state :: <object>

Description

This function hashes every element of *sequence* using *elt-hash-function*, merging the resulting hash codes in order. Note, though two sequences may be equal according to the **\=** function, s**equence-hash** and **collection-hash** may return different hash codes for the sequences.

string-hash [Function]

Arguments

string :: <string>

Values

hash-id :: <integer>
hash-state :: <object>

Description

This function calls produces hash codes for strings without using the strings' identities. This function is suitable for use with **<value-table>**s.

value-hash [Generic Function]

Arguments

object :: <object>

Values

hash-id :: <integer>
hash-state :: <object>

Description

This function produces hash codes for objects without using the objects' identities. This function is suitable for use with **<value-table>**s. Mindy provides methods specialized for the following types: **<string>**, **<integer>**, **<float>**, **<character>**, **<symbol>**, `singleton(#t)`, and `singleton(#f)`.

11.4. Exiting Applications

The `Extensions` module exports the following functionality for controlling the exiting of applications:

exit [Function]

Arguments

#key exit-code :: <integer> = 0

Description

Causes the process to exit. Mindy calls this function when there is no code left to execute.

on-exit [Function]

Arguments

function :: <function>

Values

meaningless :: singleton(#f)

Description

Arranges for the **exit** function to call the argument *function*. The argument *function* must take no required arguments. Users may call **on-exit** multiple times to install more than one function for **exit** to call, but the order in which **exit** invokes the functions is undefined. Calling **on-exit** on the same function repeatedly, installs that function multiple times.

11.5. Weak Pointers

The `Extensions` module exports the following weak-pointer functionality:

<weak-pointer> [Class]

This class is a subclass of **<object>**. The **make** method for this class takes the keyword parameter *object:*. Instances of **<weak-pointer>** refer to the object passed to the **make** method as long as some other reference to the object exists. Whenever an instance of **<weak-pointer>** is the only reference to an object, and a garbage collection occurs, then Mindy considers the object to be garbage. When Mindy garbage collects an object referred to by a weak pointer, then Mindy marks the weak pointer as being *broken* (see the **weak-pointer-object** function).

weak-pointer-object [Function]

Arguments

wp :: <weak-pointer>

Values

object :: <object>

broken? :: <boolean>

Description

Returns the object referred to by the weak pointer and whether the weak pointer is *broken*. A weak pointer is broken when it contains the only reference to an object, and in this situation, **weak-pointer-object** returns the values `#f` and `#t`.

11.6. Collections

The `Extensions` module exports the following **<collection>** functionality:

key-exists? [Generic Function]

Arguments

collection :: <collection>

key :: <object>

Values

win? :: <boolean>

ele :: <object>

Description

Return whether *key* is in *collection*. If the key is in the collection, then the second value is the element associated with *key*; otherwise, the second return value is `#f`.

12. The System Module

The `System` module exports the following:

<buffer> [Class]

This class is a subclass of **<vector>**. It is the built-in class in Mindy that the `Streams` module supports.

copy-bytes [Function]

Arguments

dst :: type_or(<buffer>, <byte-vector>, <byte-string>)
dst-offset :: <integer>
src :: type_or(<buffer>, <byte-vector>, <byte-string>)
src-offset :: <integer>
count :: <integer>)

Values

dst :: type_or(<buffer, <byte-vector>, <byte-string>)

Description

Copies *count* bytes from *src* to *dst*, starting at *src-offset* and *dst-offset*, respectively. This function returns *dst*. This function does no bounds checking. *Dst* and *src* may be the same (\==) object; this function ensures that it copies bytes from to the destination portion correctly, regardless of overlap.

13. The Introspection Module

The `Introspection` module exports reflective operations for examining classes, functions, and types.

13.1. Functions

Dylan provides some reflective operations for functions, **function-specializers** and **instance?**. With the latter, you can determine if a function is a **<generic-function>** or **<method>**, but neither Dylan nor Mindy provides exports class identifiers for other types of functions (such as **block** exit functions). The Subsection *Types* describes definitions that are also useful when inspecting methods because you can get detailed information about method specializer types.

The `Introspection` module exports the following for functions:

function-name [Function]

Arguments

function :: <function>

Values

result :: false-or(<symbol>)

Description

Returns the name of *function* as a **<symbol>** if *function* has a name; otherwise **function-name** returns `#f`. All functions defined with `define generic` or `define method` have names, and some other functions have names.

13.2. Classes and Instances

The `Introspection` module exports the following for class objects, slot descriptions, and fetching and modifying the slot values of general objects:

class-name [Function]

Arguments

class :: <class>

Values

result :: false-or(<symbol>)

Description

Returns the name of *class* as a **<symbol>** if class has a name; otherwise, this function returns `#f`. Mindy can always determine the name of classes defined with `define class`.

<slot-descriptor> [Class]

This class is a subclass of **<object>**. The **slot-descriptors** function returns instances of this class to describe the slots of a class object.

slot-descriptors [Function]

Arguments
class :: <class>
Values
descriptors :: <list>
Description
Returns a list of **<slot-descriptor>**s for *class*. The result may be the empty list.

slot-name [Function]

Arguments
slot :: <slot-descriptor>
Values
name :: <symbol>
Description
Returns the name of *slot* as a **<symbol>**.

slot-allocation [Function]

Arguments
slot :: <slot-descriptor>
Values
allocation :: one-of(#"instance", #"class", #"subclass", #"constant", #"virtual")
Description
Returns the allocation type for *slot* as a **<symbol>**.

slot-type [Function]

Arguments
slot :: <slot-descriptor>
Values
type :: <type>
Description
Returns the type of values permitted for *slot*.

slot-getter [Function]

Arguments
slot :: <slot-descriptor>
Values
gf :: <generic-function>
Description
Returns the generic function that accesses *slot*.

slot-setter [Function]

Arguments
slot :: <slot-descriptor>
Values
gf :: <generic-function>
Description
Returns the generic function that stores into *slot*.

slot-value [Function]

Arguments
slot :: <slot-descriptor>
object :: <object>
Values
value :: <object>
initialized? :: <boolean>
Description
Returns the value for *slot* in *object* and `#t`. If the slot in the object is uninitialized, then this function returns `#f` and `#f`. Note, this function does not go through generic function dispatch, and it calls no user methods; this function uses an internal primitive to fetch the slot's value.

slot-value-setter [Function]

Arguments
value :: <object>
slot :: <slot-descriptor>
object :: <object>
Values
value :: <object>
Description
Stores *value* into *slot* of *object* and returns *value*. This function performs whatever type checking is necessary to ensure *value* is safe for *slot*.

13.3. Types

The `Introspection` module exports the following for inspecting types (and therefore, method specializers):

singleton-object [Function]

Arguments
specializer :: <singleton>
Values
object :: <object>
Description
This function returns the object of the singleton value type.

<subclass> [Class]

This class is a subclass of **<type>**. Instances of this class represent subclass specializers. A subclass specializer causes a method to be invoked whenever the generic function was called on a value that is the specified class or any subclass of the specified class (see Section *Mindy vs. Dylan Language Issues* for more information). The function **subclass-of** returns the class specified for the subclass specializer.

subclass-of [Function]

Arguments
specializer :: <subclass>
Values
class :: <class>
Description
Returns the class specified for the subclass specializer.

<limited-integer> [Class]

This class is a subclass of **<type>**. Instances of this class represent limited integer types. See the functions **limited-integer-class**, **limited-integer-min**, and **limited-integer-max**.

limited-integer-class [Function]
Arguments
specializer :: <limited-integer>
Values
class :: one-of(<fixed-integer>, <extended-integer>)
Description
Returns the class specified for the limited-integer specializer, either **<fixed-integer>** or **<extended-integer>**.

limited-integer-min [Function]
limited-integer-max [Function]
Arguments
specializer :: <limited-integer>
Values
class :: false-or(<integer>)
Description
Return the inclusive bounds of the limited-integer specializer. If the minimum or maximum is unbounded, then the appropriate function returns `#f`.

<union> [Class]
This class is a subclass of **<type>**. Instances of this class represent union types. The function **union-members** returns a list of the member types in the union.

union-members [Function]
Arguments
specializer :: <union>
Values
types :: <list>
Description
Returns the member types of the union type. The result may contain more than two elements. This function collapses nested union types to a flat list.

13.4. Miscellaneous

The `Introspection` module exports the following miscellaneous functionality:

object-address [Function]
Arguments
object :: <object>
Values
address :: <integer>
Description
Returns an integer for *object*. If the object is represented internally represented as immediate data, then the integer returned is only unique to the value of the object. If the object is represented on the dynamic heap, then the integer uniquely identifies the object from all other objects.

14. The File-descriptor Module

A cleaner interface to most of these functions is available from the `Streams` library (see the document `$INSTALL/doc/streams.{ps,txt}`). You probably do not need to use the `File-descriptor` module, unless you are using **fd-exec** or need an obscure file mode.

The `File-descriptor` module exports the following functions and constants:

fd-exec [Function]

Arguments
command-line :: <string>
Values
in-fd :: false-or(<integer>)
out-fd :: false-or(<integer>)
Description
This function provides a facility for running programs and scripts from within Mindy. The *command-line* argument should contain the name of the program and all of the command line arguments for that program. This function returns the file descriptors for the new process's standard input and output. If **fd-exec** is unable to start the process, then it returns `#f` and `#f`.

fd-open [Function]

Arguments
path :: <byte-string>
flags :: <integer>
Values
fd :: false-or(<integer>)
errno :: false-or(<integer>)
Description
This function calls the C **open** system call and returns the file descriptor and `#f`, if successful. If the first value is `#f`, then the second value is the error number. You can convert the error number to a string using the **fd-error-string** function.

fd-close [Function]

Arguments
fd :: <integer>
Values
win? :: <boolean>
errno :: false-or(<integer>)
Description
This function calls the C **close** system call and returns `#t` and `#f`, if successful. If the first value is `#f`, then the second value is the error number. You can convert the error number to a string using the **fd-error-string** function.

fd-read [Function]

Arguments
fd :: <integer>
buffer :: <buffer>
offset :: <integer>
count :: <integer>
Values
count :: false-or(<integer>)
errno :: false-or(<integer>)
Description
This function calls the C **read** system call and returns the number of bytes read and `#f`, if successful. *Offset* is an index into *buffer*, and it the index at which **fd-read** should start writing into the buffer. All other arguments are the same as those described by the Unix man page.

If the first value is `#f`, then the second value is the error number. You can convert the error number to a string using the **fd-error-string** function.

This function does no bounds checking.

fd-write [Function]

Arguments
- fd :: <integer>
- buffer :: <buffer>
- offset :: <integer>
- count :: <integer>

Values
- count :: false-or(<integer>)
- errno :: false-or(<integer>)

Description

This function calls the C **write** system call and returns the number of bytes written and `#f`, if successful. *Offset* is an index into *buffer*, and it is the index at which **fd-write** should start reading from the buffer. All other arguments are the same as those described by the Unix man page.

If the first value is `#f`, then the second value is the error number. You can convert the error number to a string using the **fd-error-string** function.

This function does no bounds checking.

fd-input-available? [Function]

Arguments
- fd :: <integer>

Values
- input? :: <boolean>
- errno :: false-or(<integer>)

Description

This function returns whether there is any input available on the file descriptor. The second return value is `#f` if **fd-input-available?** could determine whether input was available. If there is an error, the second return value is the error number. You can convert the error number to a string using the **fd-error-string** function.

fd-sync-output [Function]

Arguments
- fd :: <integer>

Values
- win? :: <boolean>
- errno :: false-or(<integer>)

Description

This function calls the C **fsync** system call and returns `#t` and `#f`, if successful. If the first value is `#f`, then the second value is the error number. You can convert the error number to a string using the **fd-error-string** function.

fd-seek [Function]

Arguments
- fd :: <integer>
- offset :: <integer>
- whence :: <integer>

Values
- new-pos :: false-or(<integer>)
- errno :: false-or(<integer>)

Description

This function calls the C **lseek** system call and returns the new absolute position in the file and `#f`, if successful. If the first value is `#f,` then the second value is the error number. You can convert the error number to a string using the **fd-error-string** function.

fd-error-string [Function]

Arguments

errno :: <integer>

Values

msg :: false-or(<byte-string>)

Description

This function calls the C **strerror** system call and returns the string that describes the given error number. If the error number is unknown, then **fd-error-string** return `#f`.

L_SET [Constant]
L_INCR [Constant]
L_XTND [Constant]
FNDELAY [Constant]
FAPPEND [Constant]
FCREAT [Constant]
FTRUNC [Constant]
FEXCL [Constant]
O_RDONLY [Constant]
O_WRONLY [Constant]
O_RDWR [Constant]
O_NDELAY [Constant]
O_APPEND [Constant]
O_CREAT [Constant]
O_TRUNC [Constant]
O_EXCL [Constant]
ENOENT [Constant]
EIO [Constant]
ENXIO [Constant]
EACCES [Constant]
EFAULT [Constant]
EEXIST [Constant]
ENOTDIR [Constant]
EISDIR [Constant]
EINVAL [Constant]
ENFILE [Constant]
EMFILE [Constant]
ETXTBSY [Constant]
ENOSPC [Constant]
EROFS [Constant]
EOPNOTSUPP [Constant]
ELOOP [Constant]
ENAMETOOLONG [Constant]
EDQUOT [Constant]
EBADF [Constant]
EINTR [Constant]
EWOULDBLOCK [Constant]
EPIPE [Constant]
EFBIG [Constant]

These constants are the same constants from the standard C libraries, `file.h` and `errno.h`. The `File-descriptors` module exports all the constants users need to call the functions in the module, or test the functions' return values.

15. The Cheap-io Module

The `Cheap-io` module exports some basic, unextendable I/O functionality. Mindy uses the `Cheap-io` functions internally. The Gwydion Project also provides the `Streams`, `Print`, and `Format` libraries (see the `$INSTALL/doc/` directory for documentation). If any library that you load into Mindy uses the `Debugger-format` library, then the debugger uses **format** from the `Format` library.

format [Function]

Arguments
control-string :: <byte-string>
#rest arguments
Values
meaningless :: singleton(#f)
Description
This **format** adheres to the format strings described in the *Dylan Interim Reference Manual* with one exception. Mindy incorrectly prints instances of **<condition>** supplied to the `%S` directive. The `Format` library provides a correct **format** function that supports an upward-compatible extension to the format control strings described in the DIRM.

print [Function]
prin1 [Function]

Arguments
object :: <object>
Values
meaningless :: singleton(#f)
Description
Prints thing to stdout. **Print** follows thing with a newline. You cannot extend or specialize how objects are printed because these function's are written in C code, within Mindy's implementation.

puts [Function]

Arguments
string :: <byte-string>
Values
meaningless :: singleton(#f)
Description
Prints the contents of *string*. to stdout.

putc [Function]

Arguments
char :: <byte-character>
Values
meaningless :: singleton(#f)
Description
Prints *char* to stdout.

getc [Function]

Arguments
none
Values
char :: <byte-character>
Description
Read and return the next character from stdin. Returns `#f` at `EOF`.

fflush [Function]

Arguments

none

Values

meaningless :: singleton(#f)

Description

Forces out any pending output generated by **format**, **print**, **prin1**, **puts**, and **putc**.

16. The Threads Module

This module is in the `Dylan` library and exports an interface to **<thread>**s, **<lock>**s, and **<event>**s (objects on which threads can wait until a signalling thread indicates the events have occurred).

16.1. Classes and Functions

The `Threads` module exports the following classes and functions:

<thread> [Class]

This class is a subclass of **<object>**. Instances of this class are the handles by which programs manipulate threads.

spawn-thread [Function]

Arguments

debug-name :: <byte-string>

init-function :: <function>

Values

thread :: <thread>

Description

Spawns a concurrent asynchronous thread and invokes *init-function* in that thread. The dynamic context of the thread is the same as if it were the main thread of a program at the beginning of the program's execution.

kill-thread [Function]

Arguments

thread :: <thread>

Values

thread :: <thread>

Description

Kills thread immediately. After calling this function, the argument thread never executes again.

<lock> [Abstract Class]

This class is a subclass of **<object>**. Instances of this class provide logical locks. A lock is locked when a thread successfully *grabs* a lock, and we say the thread *holds* the lock. Holding a lock in no way prohibits access to a resource. It is purely the convention of various threads to access a shared resource only after successfully grabbing a lock. If **<lock>** is passed to **make**, **make** returns a **<spinlock>**.

<spinlock> [Sealed Class]

This class is a subclass of **<lock>**. Instances of this class provide a single-locking model. Whenever a **<spinlock>** is locked, any thread that tries to grab it will block. Whenever a **<spinlock>** is locked, any thread may release it. Whenever a **<spinlock>** is unlocked, any thread may grab it.

<spinlock>s are designed to be held for a very short period of time, several machine instructions at most. Threads should only hold a **<spinlock>** for a very short period of time because other threads that are waiting for the lock are blocked and could be wasting CPU cycles by busy looping; that is, waiting for a **<spinlock>** does not necessarily use anything as heavy weight as a system call to sleep the thread waiting for the lock. If only a

couple threads are sharing a resource, it may be more efficient to actually hold a **<spinlock>** for a moderate amount of time while performing a high-level operation, rather than use a lock to build a more heavy-weight mutual exclusion mechanism (such as a semaphore) to isolate access to the shared resource.

Unlocking a **<spinlock>** when it is already unlocked signals an error.

<multilock> [Sealed Class]

This class is a subclass of **<lock>**. Instances of this class provide a multilocking model. Whenever a **<multilock>** is unlocked, any thread may grab it. A thread that holds a **<multilock>** may grab the lock repeatedly without releasing the lock. Each grab effectively increments a counter, and each release effectively decrements a counter. A **<multilock>** is available to be grabbed by any thread when the counter returns to zero; therefore, a thread must release the lock for each grabbing of the lock. This behavior is useful for implementing a high-level operation that needs to isolate access to a resource while calling a few lower-level operations that lock the resource; in this way, the high-level operation effectively calls all the lower-level operations atomically with no other threads affecting the state of the resource between the calls.

Whenever a **<multilock>** is locked, only the thread that holds the lock may release it.

<multilock>s are designed to be held for as long as a thread requires. When other threads call the **grab-lock** function and block because a **<multilock>** is locked, the other threads are guaranteed to sleep until the lock is available.

Unlocking a **<multilock>** when it is already unlocked signals an error.

<semaphore> [Sealed Class]

This class is a subclass of **<lock>**. Instances of this class provide a single-locking model.

Whenever a **<semaphore>** is unlocked, any thread may grab it. Whenever a **<semaphore>** is locked, any thread that tries to grab it will block. Whenever a d is locked, any thread may release it.

<semaphore>s are designed to be held for as long as a thread requires. When other threads call the **grab-lock** function and block because a **<semaphore>** is locked, the other threads are guaranteed to sleep until the lock is available.

Unlocking a **<semaphore>** when it is already unlocked signals an error.

locked? [Function]

Arguments
lock :: <lock>
Values
locked? :: <boolean>
Description
Returns whether the lock is held by any thread.

grab-lock [Generic Function]

Arguments
lock :: <lock>
Values
meaningless :: singleton(#f)
Description
Returns after successfully grabbing the lock. If the lock is not immediately available, this function waits for the lock to become available.

grab-lock [G.F. Method]

Arguments
lock :: <spinlock>
Values
meaningless :: singleton(#f)
Description
Returns after successfully grabbing the lock. This method can only grab *lock* when it is unlocked. When the lock is held, this method may busy-loop until the lock is unlocked.

grab-lock [G.F. Method]

Arguments
lock :: <semaphore>
Values
meaningless :: singleton(#f)
Description
Returns after successfully grabbing the lock. This method can only grab *lock* when it is unlocked. When the lock is held, this method puts the calling thread to sleep until the lock is available.

grab-lock [G.F. Method]

Arguments
lock :: <multilock>
Values
meaningless :: singleton(#f)
Description
Returns after successfully grabbing the lock. A single thread may successfully call this method repeatedly, but the thread must call **release-lock** once for each call to **grab-lock**. If the thread calls **release-lock** fewer times than **grab-lock**, the lock remains locked, and any threads waiting for the lock will continue to wait. When a thread that does not hold the lock calls this method, the method puts the calling thread to sleep until the lock is available.

release-lock [Generic Function]

Arguments
lock :: <lock>
Values
meaningless :: singleton(#f)
Description
Releases the lock. If *lock* is unlocked, this function signals an error.

release-lock [G.F. Method]

Arguments
lock :: union(<spinlock>, <semaphore>)
Values
meaningless :: singleton(#f)
Description
Releases the lock. If *lock* is unlocked, this function signals an error. Any thread may unlock a **<spinlock>** or **<semaphore>**, regardless of whether it is the thread that successfully grabbed the lock.

release-lock [G.F. Method]

Arguments
lock :: <multilock>
Values
meaningless :: singleton(#f)
Description
Releases the lock. If *lock* is unlocked, this function signals an error. Only the thread that holds *lock* may call this

function, and if another thread tries to release the lock, this method signals an error. When this function returns, *lock* may still be locked. A thread that has repeatedly grabbed a **<multilock>** must call **release-lock** once for each call to **grab-lock**.

<event> [Class]

This class is a subclass of **<object>**. Threads use events to block without busy looping and to communicate to other threads that they should wake up.

wait-for-event [Generic Function]

Arguments

event :: <event>
lock :: <lock>

Values

meaningless :: singleton(#f)

Description

Releases the lock and puts the calling thread to sleep until some other thread signals *event*. After this function returns, the lock is unheld, and the calling thread must try to grab the lock before accessing any shared resources. Due to implementation details, this function may return even when the lock is unavailable, or the event has not truly occurred; because of this, programs need to loop over **wait-for-event** and **grab-lock**, testing that the event actually occurred. Methods exist for both **<spinlock>**s and **<semaphore>**s.

signal-event [Function]

Arguments

event :: <event>

Values

meaningless :: singleton(#f)

Description

Signals that the event occurred, indicating that Mindy should wake up a thread that is waiting on this event.

broadcast-event [Function]

Arguments

<event>

Values

meaningless :: singleton(#f)

Description

Signals that the event occurred and causes Mindy to wake up every thread that is waiting on this event.

16.2. Examples

The following code shows how to use locks and events to isolate access to a queue:

```
// This example shows two routines, get-queue and release-queue.  Code
// that accesses the queue should call get-queue before doing so and call
// release-queue when done.  Any code failing to isolate access to the
// queue in this way has undefined behavior and is incorrectly written.
//

// This variable is #t if and only if the queue is generally available.
//
define variable queue-available? = #t;

// This constant holds an event object used to signal when the queue
// becomes generally available again.
//
define constant queue-available = make(<event>);
```

```
// This constant holds a lock object used to isolate access to
// queue-available? for testing and setting purposes.
//
define constant queue-lock = make(<lock>);

// When this function returns, the caller has exclusive access to the
// queue.  If necessary, this function waits for the queue to become
// available, but it does not busy loop.  This function returns #f as
// a meaningless return value.
//
define method get-queue ()
  grab-lock(queue-lock);
  while (~ queue-available?)
    wait-for-event(queue-available, queue-lock);
    grab-lock(queue-lock);
  end;
  queue-available? := #f;
  lock-release(queue-lock);
  #f;
end;

// This function releases the queue and signals that it is released so
// that someone waiting on the queue will be woken up.  This function
// returns #f as a meaningless return value.
//
define method release-queue ()
  grab-lock(queue-lock);
  queue-available? := #t;
  release-lock(queue-lock);
  signal-event(queue-available);
  #f;
end;
```

The following example shows how to use a lock to isolate queue access in a different way than the previous example:

```
// This constant holds an event object used to signal when an element
// exists in the queue.
//
define constant something-available = make(<event>);

// This constant holds a lock that is held whenever a thread is accessing
// queue.
//
define constant lock = make(<lock>);

// This constant holds a queue object.
//
define constant queue = make(<deque>);

// This function returns an element from queue.  If no element is
// immediately available, then this function blocks until it can return
// an element.  This function assumes only one or two other threads are
// ever waiting for the queue, and it assumes pop is a fast high-level
// operation.
//
define method get-something()
```

```
  grab-lock(lock);
  while (empty?(queue))
    wait-for-event(something-available, lock);
    grab-lock(lock);
  end;
  let result = pop(queue);
  lock-release(lock);
  result;
end;

// This function adds thing to queue.  It assumes only one or two other
// threads are ever waiting for the queue, and it assumes push is a fast
// high-level operation.
//
define method put-something(thing)
  grab-lock(lock);
  push(queue, thing);
  release-lock(lock);
  signal-event(something-available);
end;
```

17. Streams Library

There is a `Streams` library that adheres to the Gwydion streams specification. For documentation on the stream specification, see the file `$INSTALL/doc/streams.{ps,txt}`.

The `Streams` library exports two modules, `Streams` and `Standard-io`. The `Streams` module exports all identifiers from the streams specification. The `Streams` module also exports **<fd-stream>**:

<fd-stream> [Class]

This class is a subclass of **<stream>**. These streams are based on C file descriptors, and they do not adhere to the Random Access Protocol described in the Gwydion streams specification. The **make** method accepts the following keywords:

direction:
: This keyword is optional and defaults to `#"input"`. When supplied, it must be either `#"input"` or `#"output"`.

fd:
: This keyword is required and should be an open file-descriptor.

size:
: This keyword is optional and is the size of the buffer. See the Streams specification for details.

The `Standard-io` module exports the following:

standard-input [Constant]
standard-output [Constant]
standard-error [Constant]

These have the following values respectively:

```
make(<fd-stream>, fd: 0)
make(<fd-stream>, fd: 1, direction: #"output")
make(<fd-stream>, fd: 2, direction: #"output")
```

18. Integers

Mindy's **<integer>** class is abstract. Mindy provides two concrete classes, **<fixed-integer>** and **<extended-integer>**. Both concrete classes are direct subclasses of **<integer>**. Expressions involving **<extended-integer>**s produce **<extended-integer>** results because **<extended-integer>**s are contagious. If an expression involving only **<fixed-integer>** values would produce a result that does not fit in a **<fixed-integer>**, then Mindy signals an overflow error. You can use the **as** function to convert back and forth between **<fixed-integer>**s and **<extended-integer>**s. **As** signals an error when converting an **<extended-integer>** to a **<fixed-integer>**, and the value does not fit in a **<fixed-integer>**.

Mindycomp parses all integer literals as **<fixed-integer>**s. If a literal does not fit in a **<fixed-integer>**, then mindycomp issues a compiler error. Though the compiler supports no literal syntax for **<extended-integer>**s, the Mindy debugger prints them in a `#eDDD...` format where each `D` is a decimal digit.

The `Extension` module of the `Dylan` library exports the following constants:

$maximum-fixed-integer [Constant]
$minimum-fixed-integer [Constant]
These constants hold the largest positive **<fixed-integer>** and the largest negative **<fixed-integer>**.

19. Miscellaneous Implementation Choices

The **error** method specialized on **<byte-string>** applies the **format** function to the arguments passed to **error**. See Section *The Extensions Module* for the details of **format** from the `Cheap-io` module of the `Dylan` library. See `$INSTALL/doc/format.{ps,txt}` for the details of **format** from the `Format` library.

Rest arguments in Mindy are **<sequence>**s. You cannot use any functions on the rest argument that assumes the collection is an instance of any class more specific than **<sequence>**; for example, you cannot use the **head** or **tail** functions because they operate on instances of **<pair>**.

Mindy's **<character>** implementation is equivalent to unicode characters. The **<byte-character>** class exported from the `Extensions` module of the `Dylan` library is a subclass of **<character>**.

20. Copyright and Terms of Use

Copyright (c) 1994 Carnegie Mellon University All rights reserved.

Use and copying of this software and preparation of derivative works based on this software are permitted, including commercial use, provided that the following conditions are observed:

- This copyright notice must be retained in full on any copies and on appropriate parts of any derivative works.
- Documentation (paper or online) accompanying any system that incorporates this software, or any part of it, must acknowledge the contribution of the Gwydion Project at Carnegie Mellon University.

This software is made available *as is*. Neither the authors nor Carnegie Mellon University make any warranty about the software, its performance, or its conformity to any specification.

Bug reports, questions, comments, and suggestions should be sent by E-mail to the Internet address `gwydion-bugs@cs.cmu.edu`.