Gwydion Dylan[1] *1.6*

Mindy and Common Libraries

by the Gwydion Project
Copyright 1994, 1995, 1996, 1997 Carnegie Mellon University
All rights reserved

1.Dylan is a trademark of Apple Computer.

Copyright (c) 1994, 1995, 1996, 1997 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*.

The d2c Dylan-to-C compiler 4

- d2c Overview 4
- Online documentation 4

The Mindy Compiler and Interpreter 5

- Introduction 5
- Hello, World 5
- The Main Routine 6
- Multiple Files 6
- Return Codes 7
- Command Line Switches and Environment Variables 7
- Errors and Warnings 7
  - Syntax Errors 7
  - Runtime Errors 8
  - Warnings 8
  - Stack Overflow 8
  - Internal Lossage 8
- Dylan vs. Mindy Language Issues 8
- Built-in Libraries and Modules 9
- Using Libraries and Modules 10
- The Extensions Module 11
  - Generally Useful Definitions 11
  - Exiting Applications 12
  - Weak Pointers 12
  - Debugger Customizations 12
- The System Module 12
- The Introspection Module 13
  - Classes and Instances 13
- The File-descriptor Module 15
- The Cheap-io Module 18
- The Threads Module 19
  - Classes and Functions 19
  - Examples 23
- Miscellaneous Implementation Choices 25
- Copyright and Terms of Use 25

The Mindy Debugger 26

- Introduction 26
- Stack Manipulation Commands 26
- Examining Variables 27
- Libraries and Modules 28
- Evaluating Expressions 29
- Debugger Variables 30
- Restarts and Returning 31
- Interrupting and Single Stepping 32
- Breakpoints 33
- Threads 36
- Inspect and Xinspect 37
- Miscellaneous Commands 37
- Copyright and Terms Of Use 38

The Mindy Object Inspector 39

- Introduction 39
- Inspector-Base 39
- Text-Inspector 39
  - Exported Names 39
  - Text-Inspector Commands 40
  - The History Command 40
  - The Up Command 40
  - The Print Command 40
  - The Store Command 41
  - The View Command 41
  - The Help Command 41
  - The Quit and Exit Command 41
- X-Inspector 41

Gwydion Extensions 42

- Introduction 42
- Dylan Language Issues 42
- Conditional Compilation 43
- Modules of the Dylan Library 44
- The Extensions Module 44
  - Generally Useful Definitions 44
  - Debugger Customizations 45
  - Exiting Applications 46
  - Collections 46
  - Integers 47
  - Ratios 47
- The System Module 48
- The Introspection Module 49
  - Functions 49
  - Classes 49
  - Types 50
  - Miscellaneous 51
- The Cheap-io Module 51
- Extensions to the Streams Library 52
- Copyright and Terms of Use 52

The Streams Library 53

- About this document 53
  - Acknowledgments 53
  - Discussing error conditions 53
- Goals of the library 54
- Concepts 55
  - Streams, growing sequences, and object identity 57
- Stream classes 59
  - Creating streams 61
  - Closing streams 66
- Reading and writing from streams 67
  - Reading from streams 67
  - Convenience functions for reading from streams 70
  - Writing to streams 71
  - Reading and writing by lines 73
  - Querying streams 75

Positionable stream protocol 76
Using file streams 80
Locking streams 81
Using buffered streams 83
Overview 83
Buffer access protocol 83
Copying to and from buffers 91
Stream extension protocol 93
Conditions 95
Wrapper streams 96
Wrapper streams and delegation 98
Wrapper stream protocol 100

The Standard-IO Library 103

The Print Library 104
Print Functions 104
Print Request Inspection Functions 105
Pretty Printing 106

The Format Library 109
Functions 109
Control Strings 111

The TK Library 112
Introduction and Warning 112
General Principles 112
Parameter and Return Values 113
The "Standard" Options 114
Windows 115
Packing 116
Event bindings 117
Active Variables 118
The widget types 118
<button>: ***118***
<checkbutton>: ***118***
<menubutton> ***119***
<radiobutton>: ***119***
<canvas>: ***120***
<entry>: ***121***
<frame>: ***121***
<label>: ***121***
<listbox>: ***122***
<menu>: ***122***
<message>: ***123***
<scale>: ***123***
<scrollbar>: ***123***
<text>: ***124***
<toplevel>: ***124***
Text indices 125
Text marks 125
Text tags 125
Canvas items and tags 126
Window Information commands 126
Requested Enhancements 128
The Extension Protocol 128

The Collection-extensions Library 130
Introduction 130
The Self-Organizing-List Module 130
The Subsequence Module 130
The Vector-Search Module 131

The Table-Extensions Library 132
Introduction 132
The Table-Extensions Module 132
Extensions 134

The String-extensions Library 136
Introduction 136
The String-Conversions Module 136
The Character-type Module 136
The String-hacking Module 137
The Substring-search Module 138

The Regular-Expressions Library 140
Introduction 140
Exported Names 140
Known bugs 143

The Transcendental Library 145
Introduction 145
Exported Names 145
Unimplemented Functions 146

The Time Library 148
Introduction 148
The Time Module 148
The Time-IO Module 149
Parse-Time Directives 150

The Random Library 151

The Matrix Library 152
Initialization 152
Functions 152
Future Revisions 153
References 153

The Melange Interface Generator 154

Introduction 154
A Concrete Example 155
Basic Use 157
    Loading and Finding Objects 158
Importing Header Files 159
Specifying Object Names 160
    Mapping functions 160
    Prefixes 161
    Explicit Renaming 162
    Anonymous types 162
Type Definitions 163
    Implicit class definitions 163
    Specifying class inheritance 164
Translating Object Representations 165
    Specifying low level transformations 165
    Specifying high level transformations 166
Other File Options 167
Function Clauses 167
Struct and Union Clauses 169
Pointer Clauses 170
Constant Clauses 170
Variable Clauses 170
Low level support facilities 172
    Predefined types 172
    Locating native C objects 173
    Pointer manipulation operations 174
Static linking mechanisms 176
Differences from Creole 177
Known limitations 178
Proposed modifications 179
    Enumeration clauses 179
    Inheritance of “map” and “equate” options 179
    Remerging of the “equate:” and “map:” options 179

The d2c Dylan-to-C compiler

1. d2c Overview

d2c is an optimizing compiler for Dylan. It emits C code when is then compiled into stand-alone executables. The resulting code typically runs 20 to 100 times faster than Mindy's interpreted code. Much of this speedup comes not so much from compilation per-se, but from extensive Dylan-specific optimizations such as compile time method selection.

2. Online documentation

Nearly all of the libraries described in this document are also available using d2c. Documentation specific to d2c and of d2c's limitations is online at:

```
http://eldritch.gwydion.cs.cmu.edu/gwydion/dylan/d2c.html
```

The Mindy Compiler and Interpreter

Copyright (c) 1994, 1995, 1996, 1997 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, 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, Alain Rogister, Mark Chu-Carroll, Enrico Colombini, Dave Dyer, Jonathan Bachrach, Michael Binz, Jonathan Sobel, Eric Kidd, John Shen, Carl Gay, Patrick Premont, Eric Gouriou, Kim Barrett

Special thanks for major efforts to Roger Critchlow, Patrick Beard, Gary Palter, and Jim Studt for enhancements to Mindy.

1. Introduction

Mindy is an implementation of a language that is very much like the language described in the *Dylan™ Reference Manual* (DRM). *Mindy* was named for "Mindy Is Not Dylan Yet", and it still is and will probably always remain a Dylan subset, primarily because mindycomp doesn't do enough semantic analysis to implement macros or to implement the precise semantics of Dylan naming. Now that the d2c Dylan-to-C compiler is available (see web documentation), debugged Dylan applications will normally be compiled for better performance. Mindy remains useful because:

- Mindy provides a better debugging environment when its performance can be tolerated, since Mindy has a source-level debugger and vastly faster compile times.
- Mindy is more portable, so can be used on machines where d2c is not supported.

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.

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. Instructions for obtaining the different versions of Mindy can be found in the file README at the top level of the Mindy release.

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-exports.dylan`:

```
Module: dylan-user

define library hello-world
  use Dylan;
end library;

define module hello-world
```

```
    use dylan;
    use cheap-io;
    use extensions;
  end module;
```

And put this in `hw.dylan`:

```
  module: Hello-World

  // This is the canonical ‘‘hello, world’’ demo.
  define method main (argv0 :: <byte-string>, #rest noise)
    puts("Hello, World.\n");
  end;
```

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

```
    % $INSTALL/bin/mindycomp -lhello-world hw-exports.dylan
    % $INSTALL/bin/mindycomp -lhello-world hw.dylan
```

This produces files named `hw.dbc` and `hw-exports.dbc`. The `.dbc` stands for “Dylan Byte Code”. To run the program, say:

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

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

Note that even the minimal Dylan program has at least two files. This is because in order to do anything much at all, you must access additional libraries, and the only way to set up your namespace is to define your own library and module. But the only way to place code in a module is to use the `Module:` file header, so two files are required.

Note also that a blank line is required after the `Module:` file header.

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

No matter how many files there are, 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 *Using 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
```

Under Microsoft Windows, many implementations of `cat` will corrupt `.dbc` files because they perform newline translation. For this reason, under MS-Windows, we recommend using `dbclink`:

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

5. Return Codes

If Mindy encounters an unrecoverable error, or if Mindy is exited via the debugger `quit` command, Mindy exits with a return code of 1. If the Dylan program ends with a call to exit() and a return code is specified (see section *Exiting Applications*), Mindy exits with that return code. Otherwise, the return code is 0.

6. Command Line Switches and Environment Variables

Mindycomp recognizes the following command line switches:

- `-D`*feature* : This tells the conditional compilation system that *feature* is present (ie, "define"). Notice that there is no space between `-D` and *feature*. (See `libraries/extensions.txt` for more details)
- `-U`*feature* : This tells the conditional compilation system that *feature* is not present (ie, "undefine"). Notice that there is no space between `-U` and *feature*. (See `libraries/extensions.txt` for more details)

Mindy recognizes the following command line switches:

- `-f` *filename* : This tells Mindy to load file *filename*. See section *Multiple Files*.
- `-x` *filename* : This tells Mindy to load file *filename* like `-f` does, but the rest of the command line is then left uninterpretted by Mindy. This can be useful if your Dylan program also has a `-f` option.

Mindy recognizes the following environment variables:

- `BYTES_CONSED_BETWEEN_GCS` : This is the number of bytes Mindy will allocate before invoking a garbage collection. For maximum speed, this should be set to the largest value possible that won't cause thrashing. See also **collect-garbage** and ***print-GC-messages*** in section *The System Module*.
- `DYLANPATH`: This controls where Mindy searches for Dylan libraries. See section *Using Libraries and Modules*.
- DYLANDIR: This provides a way to inform a precompiled Gwydion Dylan binary distribution where it has been installed. All Mindy uses this variable for is to determine the default for DYLANPATH.

7. Errors and Warnings

7.1. 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.

7.2. 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.

7.3. Warnings

Mindycomp issues warnings at compile time when:

- `While` or `until` is used inside a `for` loop instead of `while:` or `until:`
- The obsolete "`keyword: (default)`" syntax is used instead of the "`keyword = default`" syntax.
- Function return values have names but not types
- `Next-method` is referenced inside a method that does not declare `#next` in the method header

Mindy issues warnings at runtime when:

- A variable is undefined (this becomes an error if the code referencing the undefined variable is executed)
- A method's return types do not match the generic function's return types because return types were not specified for the method.

7.4. Stack Overflow

Mindy does not gracefully handle stack overflows. If Mindy gets a stack overflow, it will die with some kind of signal, like `SIGSEG` (segmentation fault). If this happens, you probably have an infinite recursion in your program.

7.5. 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 Reference Manual*, as well as some features we would like to see in Dylan. In addition to those extensions described in the document *Gwydion Extensions*, Mindy

diverges from the DRM as described below:

Additions:

- Mindy supports multiple value binding in the `=/then` clauses of `for` statements. The format of such a clause is
  ```
  (var1, var2, ...) = expr1 THEN expr2
  ```
- Keyword parameters may have type information, and you can specify default values with either DRM syntax (which uses "`= ...`") or with the obsolete syntax (which uses "`(...)`"). Using the second syntax will generate a compiler warning. 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 allows you to place library and module definitions in the same file as the actual code. See section *Using Libraries and Modules* for details.

Deficiencies:

- Mindy does not implement macros. The DRM built-in macros (such as `if` and `method-definer`) can not be manipulated via the module system.
- Mindy does not have limited collections. (It does have limited integers, though)
- `Sealed/open` and `primary` keywords are parsed where allowed, but Mindy ignores this information about your program.
- Mindy parses the `seal generic` and `define sealed domain` forms, but does not enforce them.
- Define method does not automatically insert `#next next-method` in parameter lists. You have to explicitly add it yourself.
- Many of the DRM built-in macros allow the bodies to be empty. Mindy does not. For example, in Mindy the following is not legal:
  ```
  if (foo)
  end if;
  ```
- `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.

Mindy comes bundled with several other libraries. Documentation for these libraries can be found in `$INSTALL/doc/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 `DYLANPATH` environment variable. If `DYLANPATH` is undefined, then Mindy uses the path:

```
.:$DYLANDIR/lib/dylan
```

If `DYLANDIR` is also undefined, then the fallback default is `/usr/local/lib/dylan` on Unix and `c:\dylan\lib\dylan` on Windows. This default can be changed by recompiling Mindy, see the build instructions.

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 `DYLANPATH`.

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 `DYLANPATH`.

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. 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. If a file contains no file headers, it still must contain a leading blank line to conform to Dylan syntax.

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. Extensions which are likely to be supported in future Gwydion compilers are documented in the document *Gwydion Extensions*; here we document those extensions whose future is not so certain.

11.1. Generally Useful Definitions

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

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.

load [Function]

Arguments

name :: <byte-string>

Values

none

Description

This function takes the name of a `.dbc` file and loads the code in the file into Mindy as if the file had been specified on the command line to Mindy. There is one exception: you can only load code that defines new variables or adds methods. You cannot redefine existing definitions. Advanced Mindy users can call this function from the debugger to add code to a running process. (See the document *The Mindy Debugger*)

load-library [Function]

Arguments

name :: <symbol>

Values

none

Description

This function takes a library name and loads the code for the library into Mindy as if the library was used by another library. Advanced Mindy users can call this function from the debugger to add code to a running process. (See the document *The Mindy Debugger*)

11.2. Exiting Applications

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

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.

11.3. 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`.

11.4. Debugger Customizations

The `Extensions` module exports the following debugger customization:

debug-output [Variable]

The debugger uses the value of ***debug-output*** when performing output; its use is similar to ***warning-output*** (See "Debugger Customizations" on page 45).

Using the `Debugger-format` module in the Format library will set both ***debug-output*** and ***warning-output*** to ***standard-output***.

12. The System Module

In addition to those extensions described in the document *Gwydion Extensions*, the `System` module exports the following:

print-GC-messages [Variable]

Default value
#f
Description
This variable controls whether Mindy prints garbage collection information whenever the garbage collector runs.

collect-garbage [Function]

Arguments
#key purify :: <boolean> = #f
Values
meaningless :: singleton(#f)
Description
If *purify* is true, collect-garbage does a purifying garbage collection. Otherwise, it does a normal garbage collection. A purifying collection is just like a normal collection except that everything left over after the collection is rendered permanent and is never again considered for collection. This can be quite useful in reducing the memory demands of your programs, as the total heap you need is:

```
overhead + permanent + newspace + oldspace
```

with overhead being about five megabytes. Without *purify*, `permanent` will be 0, `newspace` is the amount of live data, and `oldspace` is the amount of live data plus `BYTES_CONSED_BETWEEN_GCS`. Note that the live data is being counted twice: once in `newspace` and once in `oldspace`. But if you purify, that live data gets moved over to `permanent`, and then only gets counted once.

getcwd [Function]

Arguments
none
Values
current-directory :: <string>
Description
Returns the current working directory.

13. The Introspection Module

The Mindy `Introspection` module exports reflective operations for examining classes, functions, and types. (See also the document *Gwydion Extensions*)

13.1. Classes and Instances

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

abstract? [Function]

Arguments
class :: <class>
Values
result :: <boolean>
Description
Returns `#t` if *class* is an abstract class; otherwise returns `#f`.

<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", #"each-subclass", #"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*. Note, this function does not go through generic function dispatch, and it calls no user methods; this function uses an internal primitive to set the slot's value.

init-keyword [Function]

Arguments
slot :: <slot-descriptor>
Values
keyword :: false-or(<symbol>)
Description
Returns the init-keyword associated with *slot*, or `#f` if there is none.

keyword-required? [Function]

Arguments
slot :: <slot-descriptor>
Values
answer :: <boolean>
Description
Returns `#t` if there is a required-init-keyword for *slot*, otherwise returns `#f`.

14. The File-descriptor Module

A cleaner interface to most of these functions is available from the `Streams` library (see the document *The Streams Library*). You probably do not need to use the `File-descriptor` module, unless you 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`.

Note: This function has been deprecated in favor of **piped-exec**, which is exported from the `Piped-exec` module of the `Streams` library. See *section Extensions to the Streams Library* in the document *Gwydion Extensions*.

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`.

SEEK_SET [Constant]
SEEK_CUR [Constant]
SEEK_END [Constant]
O_RDONLY [Constant]
O_WRONLY [Constant]
O_RDWR [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]
ENOSPC [Constant]
EROFS [Constant]
ENAMETOOLONG [Constant]
EBADF [Constant]
EINTR [Constant]
EPIPE [Constant]
EFBIG [Constant]

These constants are mostly the same constants from the standard C libraries, `file.h` and `errno.h`, but a few names have been changed. Those names that have changed should be obvious. 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 Streams Library*, *The Print Library*, and *The Format Library*, respectively). It is an error to use both `Cheap-IO` functions and `Streams/Print/Format` functions on a single stream. (For example, if you are using the `Streams` library ***standard-input***, do not also use the `Cheap-io` input functions) For this reason, if any library that you load into Mindy uses the `Debugger-format` library, the debugger will use **format** from the `Format` library. Also, look at the document *Gwydion Extensions* for `Cheap-IO` functionality also supported in other Gwydion compilers.

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.

current-thread [Function]

Arguments

none

Values

thread :: <thread>

Description

Returns the thread handle of the current thread.

<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. 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 the document *The Format Library* 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>**.

18. Copyright and Terms of Use

Copyright (c) 1994, 1995, 1996, 1997 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`.

The Mindy Debugger

Copyright (c) 1994, 1995, 1996, 1997 Carnegie Mellon University All rights reserved. Refer to the end of this document for precise terms of use.

1. Introduction

When something goes wrong with your program, Mindy drops into the debugger. From the debugger, you can examine the stack, print out variables, evaluate expressions, and do various other things that can be helpful in figuring out what went wrong.

For example, if you did not define a method for main, after starting Mindy you would see something like the following:

```
No applicable methods for main with arguments #["mindy"]

thread [0] D   main
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
mindy>
```

The first line is the error message. The second line tells you about the thread that encountered the error. For more information about threads see Section *Threads*. The third line tells you about the current stack frame for the thread; in this example, the last function called, which is at the top of the stack, is the **invoke-debugger** function. It was called with one argument, a **<simple-error>**.

The following sections discuss the various commands provided by the debugger. As a general rule, you can invoke a command by typing at least a unique prefix of its name. There are three commonly used commands for which a single letter suffices, regardless of all other command names:

- (d)own
- (l)ocals
- (c)ontinue

Throughout this document, some examples build on previous examples, even when those previous examples come from previous sections of the document. If there is a reference to the "previous example", then please look to the previous section's text.

This documentation uses the term *built-in* for definitions created in C code, within Mindy's implementation. Opposed to built-in definitions are definitions written in Dylan code. Parts of Mindy are built-in and parts are defined in Dylan code. Of course, all user code is written in Dylan. Some debugger commands behave differently depending on whether the object being manipulated was defined in Dylan code or built-in.

2. Stack Manipulation Commands

The Mindy debugger offers a few commands for moving up and down the stack. The two most common commands are `up` and `down`. Mindy considers the most recently called function to be at the top of the stack and the least recently called function to be at the bottom of the stack. Hence, moving down the stack moves you from a callee to its caller. For example, if you were to type `down` after the previous example, you would see something like the following:

```
mindy> down
fp 0x10034078: error({<simple-error> 0x1023fa91}, #[], #())
/afs/cs.cmu.edu/project/gwydion/mindy/src/runtime/cond.dylan
132     signal(cond);
mindy>
```

The first line tells you about the new current frame, which is a call to the **error** function. For a function written in Dylan, as opposed to a built-in function, the debugger tries to show the line of source code associated with the current frame. If the debugger could not find the source file, it still prints the line number from the source file.

While moving down the stack, you might have expected to see a call to the **signal** function before seeing a call to the **error** function. This does not happen because **signal** tail calls **invoke-debugger**. When a function tail calls another function, the callee reuses the current stack frame of the caller.

In addition to the `up` and `down` commands, you can move to a specified stack frame using the `frame` command. The debugger numbers stack frames starting at zero at the top of the stack. Currently, the debugger does not print frame numbers when it prints frame information, so moving with the `frame` command is only useful as a rough thumb bar. The following is an example of using this command to go to the top of the stack:

```
mindy> frame 0
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
mindy>
```

If you use the `frame` command without supplying a frame number, the command prints the current frame's information. This is useful if the description of the current frame has scrolled off the screen, and you want to see it again.

You can view the entire stack by using the `backtrace` command. The current frame stays the same, but the `backtrace` command always shows the entire stack from the top to the bottom. The following is example output from this command:

```
mindy> backtrace
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
fp 0x10034078: error({<simple-error> 0x1023fa91}, #[], #()) \
[/afs/cs.cmu.edu/project/gwydion/mindy/src/runtime/cond.dylan, line 132]
fp 0x10034058: main()
mindy>
```

3. Examining Variables

The `locals` command prints the value for every local variable in the function associated with the current frame. If you were at the frame for the **error** call in the previous example, using the `locals` command would look like the following:

```
mindy> locals
noise: #[]
cond: {<simple-error> 0x1023fa91}
mindy>
```

You can use the `print` command to print a specific local variable. The following is an example of printing the **cond** variable shown in the previous sample output:

```
mindy> print cond
$0={<simple-error> 0x1023fa91}
mindy>
```

The `print` command can also print the value of global variables:

```
mindy> print size
$1={<generic-function> size}
mindy>
```

For information on the labels the debugger assigns to values (that is, the `$N` identifications), see Section *Debugger Variables*.

If the debugger does not find a local variable with the name you supplied, the debugger looks for a global variable by that name in the current library and module. For more information about libraries and modules, see Section *Libraries and*

Modules.

You can refer to global variables in libraries and modules that are not the current library and module by prefixing variable names with module and/or library names. We support a colon notation that only works in the Mindy debugger and has nothing to do with legal Dylan syntax. For example, `foo:bar:baz` refers to the `baz` variable in the `bar` module of the `foo` library. There must not be any whitespace anywhere in the *identifier*. To refer to a variable in the current library but in a module that is not the current module, use the form `quux:splat`, which refers to the `splat` variable in the `quux` module.

For more information about the `print` command, see Section *Evaluating Expressions*.

4. Libraries and Modules

When evaluating expressions, the debugger uses the *current library* and *current module*. When the debugger starts up, it guesses at what library and module to make current. If you want to access a global variable from another module or library, you first make another module or library be the current one with the `library` or `module` command. If you invoke the `library` command without an argument, it lists the available libraries and tells you which one is the current one. If you invoke the `library` command with an argument, the debugger makes that library be the current library. In the same way, the `module` command either lists the modules of the current library, or it selects another module be the current module. The following are examples of using the `library` command:

```
mindy> library
Dylan-User
Dylan

Current library is Dylan
mindy> library dylan-user
mindy>
```

The following is an example of using the `module` command after having just switched to the `Dylan-user` library:

```
mindy> module
   Dylan-User
 i File-Descriptors
 i Threads
 i Extensions
 i System
 i Dylan

The current module is Dylan-User
mindy>
```

The `i` in the second column indicates that those modules are being imported into the `Dylan-user` library as opposed to being defined there. The `module` command also indicates which modules are exported from the current library. For example, if you were to switch to the `Dylan` library, the `module` command would produce the following output:

```
mindy> library dylan
mindy> module
   Dylan-User
 x  File-Descriptors
 x  Threads
    Builtin-Stuff
 x  Extensions
```

```
 x  System
 x  Dylan

The current module is Dylan-User
mindy>
```

The `x` in the first column indicates that those modules are exported. There were no `x`s in the listing of modules in the `Dylan-user` library because no modules are exported from the `Dylan-user` library. There were no `x`s in the listing of modules for the `Dylan` library because the `Dylan` library does not import any modules.

Whenever you change libraries with the `library` command, the debugger resets the current module to the `Dylan-user` module. This is because the debugger needs to make a module current in the new library, and every library has a `Dylan-user` module.

5. Evaluating Expressions

The `print` command can evaluate simple expressions and print their results. The following is an example:

```
mindy> print list(1, 2, 3)
$2=#(1, 2, 3)
mindy> print vector(4, 5, 6)
$3=#[4, 5, 6]
mindy>
```

The `print` command evaluates the variable `list` and then invokes that function with the arguments `1`, `2`, and `3`. The debugger labels values printed with a dollar sign and a number, and you can use these labels in later expressions. For more information on these, see Section *Debugger Variables*.

The expressions that the debugger accepts are limited. An expression can be one of the following:

- One of the following literals:
  - decimal number (`47`)
  - keyword (`foo:`)
  - string (`"foo"`)
  - `#t`
  - `#f`
- A variable name.
- A debugger variable (for example, `$5`).
- A function call (for example, `foo(a, b)` and `bar(c, quux: 3)`). Note that dot notation (`object.slot`) and infix operators (`x + y`) are not supported.
- The address, in hexidecimal (C format, not Dylan), of a valid dylan object (for example, `0x102050b1`). Note: use this feature with care, as a mistyped address can cause Mindy to dump core.

If the expression results in multiple values, all the values are printed on a single line:

```
mindy> print values(1, 2, 3)
$4=1, $5=2, $6=3
```

If an error occurs while the debugger is evaluating the expression, it prints the error message, aborts the `print` command, and returns to the debugger prompt. The following is an example of this situation:

```
mindy> print error("oops")
invocation failed:
  oops
mindy>
```

The `call` command is like the `print` command, but the `call` command does not handle errors by aborting. When you use the `call` command, and the expression causes an error, the debugger returns to its prompt, but any stack frames that were created due to the `call` command are now visible for inspection. The following is an example of using the `call` command:

```
mindy> call error("oops")

oops

thread [0] D   main
fp 0x100341f4: invoke-debugger({<simple-error> 0x102456b1})
mindy>
```

The `print` and `call` commands can also evaluate multiple, comma-separated expressions:

```
mindy> print 1, 2, 3
$7=1
$8=2
$9=3
mindy>
```

6. Debugger Variables

The `print` or `call` commands label every value printed, and these labels identify *debugger variables*. You can use these identifers in later expressions to refer to previously computed values. The following is an example:

```
mindy> p list(1, 2, 3)
$4=#(1, 2, 3)
mindy> p second($4)
$5=2
mindy>
```

The notation `$-N` provides a dynamic alternative to identifying debugger variables. This notation refers to previously printed values by using `N` as a count from the most recently printed value to the least recently printed. The counting begins at one.

```
mindy> print a:, b:, c:, d:
$12=a
$13=b
$14=c
$15=d
mindy> print $-1, $-2, $-3, $-4
$16=d
$17=c
$18=b
$19=a
mindy>
```

You can use `$` as a shorthand for `$-1`, and `$$` for `$-2`:

```
mindy> p 2
$20=2
mindy> p list($, 4)
$21=#(2, 4)
mindy> p list($$, 6)
$22=#(2, 6)
mindy>
```

Mindy keeps references to all debugger variables to prevent them from being garbage collected. If you no longer care about previously printed values, you might want to use the `flush` command to get rid of them:

```
mindy> flush
Flushed all debugger variables.
mindy> p $0
invocation failed:
No debug variable $0
mindy> p list(a:, b:, c:)
$0=#(a, b, c)
mindy>
```

You can use `$aN` notation to refer to the arguments passed to the function call associated with the current stack frame. `N` is the argument number, counting from zero. The following is an example:

```
mindy> frame
fp 0x10034078: error({<simple-error> 0x1023fa91}, #[], #())
/afs/cs.cmu.edu/project/gwydion/mindy/src/runtime/cond.dylan
132     signal(cond);
mindy> p $a0
$1={<simple-error> 0x1023fa91}
mindy> p $a1
$2=#[]
mindy> p $a2
$3=#()
mindy>
```

The `$aN` notation does not identify a debugger variable, and the debugger does not have to create storage for these values because they are already stored on the call stack. The `flush` command has no effect on argument values.

7. Restarts and Returning

This section discusses invoking Dylan restart handlers and returning values for conditions whose recovery protocols allow returning. If you do not know what these are, see the *Dylan Reference Manual*.

The debugger has commands that allow you to try to continue executing your program. The most common way to continue execution is to invoke a Dylan restart. To either list the available restarts or invoke a restart, you use the `restart` command:

```
mindy> call cerror("go on", "oops")

oops

thread [0] D   main
fp 0x1003428c: invoke-debugger({<simple-error> 0x10245361})
mindy> restart
0 [{class <simple-restart>}]: go on
```

```
1 [{class <abort>}]: Blow off call
mindy> restart 0
$0=#f
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
mindy>
```

In this example, the `restart` command lists two restarts. The **cerror** function establishes the `"go on"` restart (numbered 0). The `call` command establishes the `"Blow off call"` restart (numberd 1). The `restart 0` command caused **cerror** to return `#f`, which the `call` command printed.

The `abort` command invokes the first restart that handles **<abort>** restarts. The following is an example of this command:

```
mindy> call error("oops")

oops

thread [0] D   main
fp 0x100341fc: invoke-debugger({<simple-error> 0x10241d49})
mindy> abort
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
mindy>
```

If Mindy entered the debugger due to a condition that allows returning as part of its recovery protocol, then you can use the `return` command. For example, consider an **<ignorable-error>** condition that is a subclass of **<error>** and that allows returning as part of its recovery protocol. The following example shows returning from the signalling of this condition:

```
mindy> call signal(make(<ignorable-error>))

{<ignorable-error> 0x10247759}

thread [0] D   main
fp 0x100341d4: invoke-debugger({<ignorable-error> 0x10247759})
mindy> restart
0 [{class <abort>}]: Blow off call

Returning is allowed:
  ignore it.
mindy> return
$0=#f
fp 0x10034090: invoke-debugger({<simple-error> 0x10244831})
mindy>
```

8. Interrupting and Single Stepping

Sometimes it is useful to interrupt your program to see where it is currently executing. Consider the following program as an example:

```
module: Dylan-User

define method main (#rest noise)
  foo(#t);
end;

define method foo (x)
  if (x)
```

```
    foo(#f);
  else
    foo(#t);
  end;
end;
```

If you were to run this program and then interrupt it, you would see output similar to the following:

```
^C
Interrupted
thread [0] R   main
fp 0x10034060: foo(#f, #())
foo.dylan
8       if (x)
mindy>
```

After interrupting the program you have the full debugger at your disposal, as if an error had occurred. Additionally, you can use the `continue` command to resume execution:

```
mindy> continue
```

You can also use the `step` command to advance line by line through your program. When stepping, if the debugger encounters a function call, it descends into that function and steps line by line. The following is an example:

```
^C
Interrupted
thread [0] R   main
fp 0x10034060: foo(#f, #())
foo.dylan
8       if (x)
mindy> step
foo.dylan
11      foo(#t)
mindy> step
foo.dylan
8       if (x)
mindy> step
foo.dylan
9       foo(#f)
mindy> step
foo.dylan
8       if (x)
mindy>
```

The `next` command is the same as the `step` command, but the next command tries to avoid descending into function calls.

9. Breakpoints

The easiest way to insert a breakpoint into a Mindy program is to insert a call to **break** into the program text, recompile the program, and rerun it.**break** is exported from the `Extensions` module of the `Dylan` library.

The debugger also has a primitive facility for setting breakpoints in methods written in Dylan, as opposed to built-in methods. The `breakpoint` command takes two arguments, a reference to a method in which to install the breakpoint, and the line number at which to install the breakpoint. For example, consider the following program:

```
module: dylan-user

define constant foo =
  method ()
    puts("this is a test\n");
    puts("of breakpoints.\n");
    #f;
  end;
```

If you were to put a breakpoint at line 6 (the second `puts`), Mindy would produce output similar to the following:

```
mindy> break foo, 6
breakpoint 1 installed in {anonymous <byte-method> 0x10243d31\
#()} at line 6 (pc 47)
mindy> call foo()
this is a test
Breakpoint
thread [0] R   main
fp 0x100341dc: {anonymous <byte-method> 0x10243d31 #()}(#())
foo.dylan
6       puts("of breakpoints.\n");
mindy>
```

The `continue` and `step` commands can be used to continue execution (see Section *Interrupting and Single Stepping*):

```
mindy> step
of breakpoints.
foo.dylan
7       #f;
mindy> c
$0=#f
fp 0x10034090: invoke-debugger({<simple-error> 0x10243e49})
mindy>
```

The `breakpoint` command evaluates its first argument, so you can use an arbitrary expression for the function. For example, you could use **find-method** to extract a specific method from a generic function and insert a breakpoint in that method:

```
mindy> br find-method(size, list(<table>)), 886
breakpoint 1 installed in {<byte-method> size #({class <table>})}\
at line 886 (pc 35)
mindy>
```

The `breakpoint` command with no arguments lists the currently installed breakpoints:

```
mindy> breakpoint
id  where
 1  pc 47 in {<component> 0x10204ea9}
mindy>
```

The `delete N` command removes a breakpoint, where `N` is the breakpoint ID reported in the `breakpoint` listing.

Sometimes the Mindy compiler has to split a single top level form into multiple methods. When this happens, the debugger cannot always figure out where to insert your breakpoint. Consider the following program:

```
module: dylan-user

define constant foo =
  method ()
    block (exit)
      puts("this is a test\n");
      puts("of breakpoints.\n");
      #f;
    end;
  end;
```

When this program is compiled, the compiler has to put the contents of the block in a seperate method. Because of this, if you were to try to insert a breakpoint at line 7 it would not work:

```
mindy> break foo, 7
{anonymous <byte-method> 0x10243f59 #()} does not span line number 7
mindy>
```

To insert a breakpoint into this method, you need to use the `disassemble` command. It disassembles a method and all Mindy-generated methods that might be associated with that method. For example:

```
mindy> disassemble foo
anonymous component, from "foo.dylan"
5           block (exit)
    47: b0              push    function catch
    48: 21              push    const(1)        {<method-info> 0x10205149}
    49: b2              push    function list
    50: a3              push    value <object>
    51: 91              call    nargs = 1, for single
    52: 0e              push    #()
    53: 10              push    #t
    54: 06              make-method
    55: 71              call    nargs = 1, tail
{<method-info> 0x10205149}, anonymous component, from "foo.dylan"
5           block (exit)
    51: 31              push    arg(1)
    52: 20              push    const(0)        {<method-info> 0x102050b1}
    53: b1              push    function list
    54: 90              call    nargs = 0, for single
    55: 0e              push    #()
    56: 10              push    #t
    57: 06              make-method
    58: 60              pop     local(0)
6             puts("this is a test\n");
    59: b2              push    function puts
    60: 23              push    const(3)        "this is a test\n"
    61: 81 00           call    nargs = 1, for 0
7             puts("of breakpoints.\n");
    63: b2              push    function puts
    64: 24              push    const(4)        "of breakpoints.\n"
    65: 81 00           call    nargs = 1, for 0
8             #f;
    67: 11              push    #f
    68: 02              return single

{<method-info> 0x102050b1}, exit component, from "foo.dylan"
```

```
5           block (exit)
    39: b0              push    function apply
    40: a1              push    value throw
    41: 30              push    arg(0)
    42: 32              push    arg(2)
    43: 73              call    nargs = 3, tail
    mindy>
```

As you can see, the function **foo** has been split into three methods. The first one corresponds to the part of **foo** that is outside the block. The second method corresponds to the code inside the block. And the third one corresponds to the **exit** function established by the block. Look for the second method which spans line 7. The following shows how to install the breakpoint:

```
mindy> br 0x10205149, 7
breakpoint 1 installed in {<method-info> 0x10205149} at line 7 (pc 63)
mindy> call foo()
this is a test
Breakpoint
thread [0] R   main
fp 0x100341f8: {anonymous <byte-method> 0x10245f41 #({class <object>})}\
({<catch> 0x10245f81}, #())
foo.dylan
7       puts("of breakpoints.\n");
mindy> c
of breakpoints.
$0=#f
fp 0x10034090: invoke-debugger({<simple-error> 0x10244071})
mindy>
```

10. Threads

Normally, there is only one thread of execution, in which case you won't need any of the commands in this section. When you debug a multi-threaded program, these commands become very useful. The `thread` command either lists the available threads or switches between them, depending on how you invoke it. For example:

```
mindy> p spawn-thread(foo:, curry(break, "Thread foo"))
$0={<thread> 0x10243f49}
mindy> p spawn-thread(bar:, curry(break, "Thread bar"))
$1={<thread> 0x10246f19}
mindy> thread
c [0] D   main
  [1] R   foo
  [2] R   bar
mindy>
```

In this example, the `thread` command lists three threads: the `main` (or original) thread and the two threads you just created. The `c` in the first column indicates which thread the debugger is currently examining. The `[N]` indicates the thread ID for each thread. The `D` and `R` designations indicate the status of each thread. The `main`, `foo`, and `bar` labels are the debug-names passed as the first argument to **spawn-thread**.

The different thread status codes are as follows:

```
STATUS     MEANING
   D       current thread the debugger is examining
   R       running/runable
   S       suspended
   B       blocked on a lock
   W       waiting for an event
```

Giving the `thread` command an argument causes the debugger to examine another thread. You can designate threads with either its numeric ID or the debug-name passed to **spawn-thread**:

```
mindy> thread foo
thread [1] R   foo
fp 0x102550bc: {anonymous <byte-method> 0x102443d9 #({class <\
object>})}({<catch> 0x10244421}, #(), {<value-cell> 0x1024436\
9}, {<breakpoint> 0x102441e1})
/afs/cs.cmu.edu/project/gwydion/mindy/src/runtime/cond.dylan
212     init-arguments: list(format-string: "Continue from break"))
mindy> thread 0
thread [0] D   main
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
mindy>
```

Sometimes it is useful to temporarily disable some threads while debugging other threads. The `disable <thread-id-or-name>` command disables (suspends) the indicated thread, and the `enable` command allows a thread to run again:

```
mindy> disable foo
[1] S 1 foo
mindy> enable foo
[1] R   foo
mindy>
```

In this example, the status of the `foo` thread changes from `R` (runnable) to `S` (suspended) when it is disabled.

If you repeatedly use the `disable` command on the same thread, then the `enable` command must be used the same number of times to before the thread's status changes to `R`. The `1` after the `S` above is the number of times the thread `foo` has been disabled.

When a thread is suspended, the `continue` and `step` commands do not advance the thread's execution. The `disable` and `enable` commands can help you find thread synchronization problems by allowing you to explicitly control when each thread runs.

Invoking the `disable` or `enable` command with no argument affects the current thread the debugger is examining.

The `kill <thread-id-or-name>` command kills the indicated thread.

11. Inspect and Xinspect

The `inspect` and `xinspect` commands can be used to interactively examine an object and related objects. The `inspect` command is a text-based interface to the Inspector facilities, while `xinspect` is a graphical interface. See the document *The Mindy Object Inspector* for information on using the Inspector.

12. Miscellaneous Commands

The `help` command prints a one line summary of all the debugger commands.

The `quit` command causes Mindy to exit without executing any of the **on-exit** hooks. If you want the **on-exit** hooks to run,

you should invoke the **exit** function with the `print` command:

```
mindy> print exit()
```

The `tron` command turns on an internal trace facility that prints the arguments and results for every function call. The `troff` command turns this off.

The `error` command repeats the error message for the condition that caused this thread to drop into the debugger.

The `gc` command invokes the garbage collector.

The `describe` command takes an expression as an argument and evaluates it. If the result is an instance of a class defined in Dylan, as opposed to a built-in class, then the debugger identifies the class, prints the slot names, and prints the slot values. If the result of the expression is an instance of a built-in class, then the debugger prints the value and its class. The `describe` command does not create or assign to debugger variables, but you can use debugger variables in the expression given to the command. The following examples show the `describe` command:

```
mindy> describe make
{<generic-function> make} is an instance of {class <generic-function>}

mindy> describe "Testing"
"Testing" is an instance of {class <byte-string>}

mindy> describe make(<table>)
{<object-table> 0x10245d79} is an instance of {class <object-table>}
and has the following slots:
merged-hash-state-slot: {permanent hash state}
shrink-to-slot: 100
shrink-when-slot: 10
expand-to-slot: 300
expand-when-slot: 200
bucket-states-slot: #[{permanent hash state}, {permanent hash state}, \
{permanent hash state}, {permanent hash state}, {permanent hash state}]
bucket-count-slot: 5
bucket-array-slot: #[#(), #(), #(), #(), #()]
item-count-slot: 0
```

13. Copyright and Terms Of Use

Copyright (c) 1994, 1995, 1996, 1997 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`.

The Mindy Object Inspector

1. Introduction

This document describes the use of the Inspector libraries for Mindy. The Inspector libraries are used to supplement the current Mindy debugger, and provide programmers with a tool for interactively inspecting their data structures. The Inspector has two separate interfaces, a text-based interface and a grpahical interface. The text interface is similar to that of Mindy debugger. The graphical interface offers a pleasant if uninspiring way of inspecting objects; it also offers a rudimentary class diagram viewer.

The Inspector is divided into three libraries, `Inspector-Base`, `Text-Inspector`, and `X-Inspector`. The first of these provides facilities that are common to the other two libraries. The `Text-Inspector` library implements a text interface to the Inspector and the function **inspect**. The `X-Inspector` library implements a graphical interface to the Inspector and the function **xinspect**. Use of the X-Inspector requires Dylan/TK, which may not available on all systems.

The Inspector is most often invoked via the `inspect` and `xinspect` debugger commands. These debugger commands evaluate their arguments and invoke the function **inspect** or **xinspect** with the appropriate arguments. The debugger commands will automatically load the appropriate libraries if they have not already been loaded.

2. Inspector-Base

The `Inspector-base` library exists solely to support the `Text-Inspector` and `X-Inspector` libraries. Normal users will not find it useful.

3. Text-Inspector

3.1. Exported Names

inspect [Function]

Arguments

#rest objects

Values

none

Description

The **inspect** function launches an interactive Inspection session, starting with the object passed in. If no objects were supplied, the Inspector will begin by inspecting **$all-libraries** (see below). It is an error to pass more than one object to **inspect**; the #rest args is really just a hack to get "zero or one" semantics. See below for a description of Inspector commands.

$all-libraries [Constant]

This constant magically represents all the libraries currently loaded in Mindy. Inspecting this constant will allow you to peruse libraries, modules, and even named bindings (ie, constants and variables). **$all-libraries** isn't really useful for anything other than passing to **inspect** and **xinspect**.

display-object-info [Function]

Arguments

the-object :: <object>

Values

none

Description

Writes a description of the object to ***debug-output*** (see mindy.txt for information on ***debug-output***). The output is almost the same as that of **inspect**, except that it is not interactive.

3.2. Text-Inspector Commands

When the text inspector starts, it will display the object you have passed to it, along with all of the information you requested using the keyword flags. Each item will be numbered, and to inspect a sub-object you simply type the number corresponding to the desired choice (in this case, even a superclass is considered a sub-object, because it was inspected via the main object). This lets you move around the object heterarchy, but it is not very convienent. There are additional commands that are designed to allow easier use of the inspector:

`1`, `2`, `3`, ...
: Inspect the subobject corresponding to that number in the current object.

`history`
: Shows the stack of inspected objects

`up`
: Moves up the inspected object stack (i.e. inspects the previous object)

`print`
: Prints the current object using the print function, and the length and level that were passed to inspect.

`store`
: Stores the current object in the next unused debugger variable.

`view`
: Redisplays the current object (useful if it has gone offscreen)

`?`, `help`
: Displays a help page much like this

`quit`, `exit`
: Leaves the inspector and continues program execution

With the exception of `history`, all commands may be abbreviated by their first letter.

3.3. The History Command

The `history` command lists all of the previously visited objects in the current session. A sample output may look like this:

```
Instance of <my-class>          <--- Initial object, the one given in the argument
Instance of <sequence>
Instance of <fixed-integer>
Class <fixed-integer>
Class <integer>                 <--- Last object seen, the one that "up" goes to
Class <rational>                <--- The current object
```

3.4. The Up Command

The `up` command moves you "up" the history, and lets you see the previously visited object. In the above example, if you moved up twice, the history would look like this:

```
Instance of <my-class>
Instance of <sequence>
Instance of <fixed-integer>
Class <fixed-integer>
```

And you would be inspecting the class **<fixed-integer>**.

3.5. The Print Command

The `print` command pretty-prints the current object (using the Gwydion Print library), but does not number the sub-objects. It is useful only for displaying the objects in a different, possible more informative manner.

3.6. The Store Command

The `store` command stores the current object in the next unused debugger variable. See `mindy/debug.txt` for information about debugger variables.

3.7. The View Command

The view command will allow you to redisplay the object that you are currently inspecting. This is mainly useful if previous commands such as history or view have moved the object off of the screen.

3.8. The Help Command

The help command will display a short help page listing all of the commands and a brief one line description of how they work.

3.9. The Quit and Exit Command

This command (both words mean the same thing) will allow you to leave the inspector and continue with the execution of your program. The next time inspect is called, it will not have any memory of the past session. This means that the history will initially show only the object that you called inspector with. If **inspect** was invoked from the debugger, you will return to the debugger.

4. X-Inspector

The X-Inspector's graphical interface is pretty self explanatory. In addition to offering all the features of the text interface, it also offers a rudimentary class diagram viewer. (The class diagrams are non-interactive--you can click all you want on them, but they won't do anything) The `Close` button will close a single window; the `Quit` button will close all the X-Inspector windows. When there are no longer any open X-Inspector windows, you will be returned to the Mindy debugger prompt.

The name X-Inspector is something of a misnomer, since it does not require X-Windows to work.

The X-Inspector library exports the following objects:

xinspect [Function]

Arguments

#rest objects

Values

none

Description

The **xinspect** function launches an interactive Inspection session, starting with the objects passed in. If no objects were supplied, the Inspector will begin by inspecting **$all-libraries** (see below). Unlike the **inspect** function, it is ok to pass multiple objects to **xinspect**; they will be displayed in separate windows.

$all-libraries [Constant]

This is the same constant as exported from the Text-inspector library.

Caveat: Both Inspectors differentiate between a binding (like a variable or a constant) and the object bound to that binding. If the binding is read-only, inspecting the binding is almost identical to inspecting the object it refers to. It is not exactly the same, because if the binding refers to a class, the X-Inspector will not offer you the option of a class hierarchy diagram.

Gwydion Extensions

Copyright (c) 1994, 1995, 1996, 1997 Carnegie Mellon University All rights reserved.

1. Introduction

In the process of working with Dylan, the Gwydion Project has come up with numerous extensions to the Dylan language. Some of them form entire libraries, like the Collection-Extensions and String-Extensions libraries. Others have been added to the Dylan library, in such modules as Extensions and System.

We continue to make no claims about future support for our extensions. However, some extensions are more likely than others to make it into our future compilers. This file documents those extensions which we think will be included in our compiler's Dylan library. Extensions which go in separate libraries are documented in their own files; extensions which are part of the Mindy Dylan library but which have a less certain future are documented in the Mindy documentation.

For the remainder of this document, we shall refer to "Gwydion compilers" as a shorthand for "Mindy and other Dylan compilers that the Gwydion Project may write." It is not meant as a guarentee that all future Gwydion releases will support these extensions.

Specific Gwydion compilers may support extensions not listed here; see their documentation for details.

2. Dylan Language Issues

Whenever possible, we have tried to keep the Dylan module pristine and unextended, prefering to add our extensions to separate modules or libraries. However, this is not always possible, particularly when it involves extending the behavior of a function or macro that is exported from the Dylan module. Currently, Gwydion compilers support these extensions to the Dylan module as described below:

- Gwydion compilers support *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.
- Gwydion compilers supports *using* clauses in for statements. The format of such a clause is
  var `IN` *collection* `USING` *protocol*

  protocol will be used instead of `forward-iteration-protocol`. *protocol* must be a variable name, not an expression. Using clauses may be used together with `keyed-by`:
  var `KEYED-BY` *key* `IN` *collection* `USING` *protocol*
- Gwydion compilers have an additional type of top level definition, `define function`, which creates a constant binding in the current module and initializes it to a new function. `Define function`'s usage is similar to `define method`. The following is an example:

```
define function cube (x)
  x * x * x;
end function cube;
```

  A similar result might be had by writing

```
define constant cube = method (x)
                         x * x * x;
                       end method;
```

  or

```
define method cube (x)
  x * x * x;
end method cube;
```

- Gwydion compilers 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;
```

3. Conditional Compilation

Gwydion compilers supports conditional compilation. The syntax is

```
#if (feature-expression)
   dylan-code
#elseif (feature-expression)
   dylan-code
#else
   dylan-code
 #endif
```

(The `#elseif` and `#else` clauses are optional) A *feature-expression* is composed of features and the `~`, `&`, and `|` operators, and may be parenthesiszed as usual. Features are not case sensitive. Mindy and d2c use the `-D` and `-U` command line switches to define new features. Some features currently supported include:

mindy
: Defined when compiling with mindycomp.

compiled-for-hppa
: Defined when compiling for the HP PA-RISC architecture.

compiled-for-x86
: Defined when compiling for the Intel 80x86 architecture.

compiled-for-sparc
: Defined when compiling for the SPARC architecture.

compiled-for-hpux
: Defined when compiling for an HP/UX system.

compiled-for-win32
: Defined when compiling for a Microsoft Win32 operating system (ie, Windows '95 or Windows/NT).

compiled-for-linux
: Defined when compiling for the Linux operating system.

compiled-for-solaris
: Defined when compiling for the Solaris operating system.

compiled-for-unix
: Defined when compiling for a Unix-based OS.

newlines-are-CRLF
: Defined when compiling for a system whose newline representation is two bytes, a carriage return ('\r') followed by a linefeed ('\f'). Examples of such systems include MS-DOS, Windows '95, Windows/NT, and OS/2. It should be kept in mind that not every file on a newlines-are-CRLF system will adhere to this convention; some will adhere to the Unix convention of using a single linefeed as the newline character.

For this reason, programs should consider the carriage return an optional part of the newline sequence.

4. Modules of the Dylan Library

In addition to containing the `Dylan` module, the `Dylan` library contains a variety of modules which provide extensions. Gwydion compilers export the following modules from the `Dylan` library:

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.

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

5. 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.

5.1. Generally Useful Definitions

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

<byte-vector> [Class]
: This class is a subclass of **<vector>** that can only hold integers between 0 and 255 inclusively.

<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 Gwydion compilers the **<character>** class is equivalent to Unicode characters.

assert [Function]
: Arguments
  : value :: <object>
: Values
  : none
: Description
  : This function signals an error if *value* is `#f`. Otherwise, it does nothing. In future Gwydion compilers, **assert** may be changed to a macro, which may or may not evaluate its argument exactly once.

one-of [Function]
: Arguments
  : #rest objects :: <object>
: Values
  : type :: <type>
: Description
  : This function takes any number of objects, and returns the type that is the type-union of the singletons of those

objects. For example, the expression

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

is equivalent to

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

false-or [Function]

Arguments

type:: <type>

Values

bigger-type :: <type>

Description

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

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

5.2. Debugger Customizations

(A note on terminology: We use the term "debugger" here in the loose, Dylan sense of anything that handles an uncaught error. In Mindy, this debugger is indeed a full fledged debugger, but in other Gwydion compilers it may not be)

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. If any library that is used itself uses the `Debugger-format` library, then the debugger uses **format** from the `Format` library, which is shipped with Gwydion compilers. You can extend how the debugger 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

none

Description

This is the function that is used 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 use **condition-format** to format output. Using **condition-format** makes your module more flexible for users of your module. If you call `Cheap-IO`'s **format**, you'll be forced to write to only one destination, 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 **condition-format**.

condition-format [Generic Function]

Arguments

stream :: <object>
control-string :: <string>
#rest arguments

Values

none

Description

This function serves as a firewall between the condition system and the `Streams` and `Format` libraries. Methods on **report-condition** should use **condition-format** to do their formatting. Users will generally use ***debug-output*** or ***warning-output*** for the *stream* argument, but this is not required.

Mindy supplies a method for when *stream* is `#"Cheap-IO"`. The Gwydion `Format` library supplies a method for when *stream* is a subclass of **<stream>**. If you are implementing your own streams or format libraries, you will need to define a method on **condition-format** for your type of stream.

condition-force-output [Generic Function]

Arguments

stream :: <object>

Values

none

Description

Condition-force-output forces any pending output from *stream*'s buffer to *stream*'s destination. This function is invoked by the debugger after a condition has been reported and before it pauses for user input. Unless you are writing a debugger, you do not need to call **condition-force-output** yourself.

Mindy supplies a method for when *stream* is `#"Cheap-IO"`. The Gwydion `Format` library supplies a method for when *stream* is a subclass of **<stream>**. If you are implementing your own streams or format libraries, you will need to define a method on **condition-force-output** for your type of stream.

warning-output [Variable]

Default-handler for **<warning>** uses ***warning-output*** to print warning messages. This variable must be either a **<stream>** from the `Streams` library, or `#"Cheap-IO"` (the default). When this variable `is#"Cheap-IO"`, the output goes to stderr.

5.3. 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 with return code *exit-code*.

5.4. Collections

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

key-exists? [Generic Function]

Arguments
collection :: <collection>
key :: <object>
Values
win? :: <boolean>
elt :: <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`.

5.5. Integers

Gwydion compilers have an abstract class **<general-integer>** which has two concrete subclasses, **<integer>** and **<extended-integer>**. **<integer>**s have a limited range of values, and **<integer>** arithmetic uses the computer's underlying integer facilities. **<extended-integer>**s can take on any value, and are similar to Common Lisp "big-nums." Expressions involving **<extended-integer>**s produce **<extended-integer>** results because **<extended-integer>**s are contagious. If an expression involving only **<integer>** values would produce a result that does not fit in an **<integer>**, then the Gwydion compiler will signal an overflow error. You can use the **as** function to convert back and forth between **<integer>**s and **<extended-integer>**s. **As** signals an error when converting an **<extended-integer>** to a **<integer>**, and the value does not fit in a **<integer>**.

The Extension module exports the following integer functionality:

<general-integer> [Abstract Class]
<extended-integer> [Class]
<general-integer> is a subclass of **<rational>**, and is the superclass of **<integer>** and **<extended-integer>**.

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

5.6. Ratios

The `Extensions` module exports the following:

<ratio> [Class]
This class is a subclass of <rational>. The ratio is normalized so that it has a positive denominator, and the greatest common divisor of the numerator and the denominator is one. Ratios are never automatically converted to integers. For example, `ratio(4,2)` would return `2/1`.

A numeric operation involving two ratios produces a normalized ratio result. A numeric operation involving a ratio and an integer produced a normalized ratio result. A numeric operation involving a ratio and a float produces a float result.

ratio [Function]

Arguments
numerator :: <general-integer>
denominator :: <general-integer>
Values
ratio :: <ratio>
Description
This function makes a ratio from the two integers.

numerator [Function]

Arguments
ratio :: <ratio>
Values
the-numerator :: <general-integer>
Description
This function returns the numerator part of *ratio*.

denominator [Function]

Arguments
ratio :: <ratio>
Values
the-numerator :: <general-integer>
Description
This function returns the denominator part of *ratio*.

6. The System Module

The `System` module exports the following:

<buffer> [Class]

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

copy-bytes [Function]

Arguments
dst :: type-union(<buffer>, <byte-vector>, <byte-string>)
dst-offset :: <integer>
src :: type-union(<buffer>, <byte-vector>, <byte-string>)
src-offset :: <integer>
count :: <integer>)
Values
dst :: type-union(<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.

getenv [Function]

Arguments
environment-variable-name :: <string>
Values
environment-variable-value :: false-or(<string>)
Description
Returns the value of the environment variable *environment-variable-name*. If *environment-variable-name* is undefined, **getenv** returns #f.

get-time-of-day [Function]

Arguments
none
Values
time-in-seconds :: <general-integer>
Description
Returns the number of seconds since midnight, January 1, 1970.

system [Function]

Arguments

command-line :: <string>

Values

return-code :: <integer>

Description

System causes *command-line* to be given to your shell as input as if the string had been typed as a command. If environment variable `SHELL` is found, its value will be used as the command interpreter (shell); otherwise `sh`(1) is used.

The parent process will wait until the command terminates. Upon termination of the sub-process, **system** will return a negative value if the command couldn't be executed, or the command's return code if it was executed.

7. The Introspection Module

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

7.1. Functions

Dylan provides some reflective operations for functions, such as **function-specializers** and **instance?**. With the latter, you can determine if a function is a **<generic-function>** or **<method>**, but neither Dylan nor Gwydion compilers provide exported 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 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.

7.2. Classes

The `Introspection` module exports the following for class 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`.

7.3. 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 *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-base-class**, **limited-integer-minimum**, and **limited-integer-maximum**.

limited-integer-base-class [Function]

Arguments

specializer :: <limited-integer>

Values

class :: one-of(<integer>, <extended-integer>)

Description

Returns the class specified for the limited-integer specializer, either **<integer>** or **<extended-integer>**.

limited-integer-minimum [Function]
limited-integer-maximum [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 (ie, what **type-union** returns). 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.

7.4. 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.

8. The Cheap-io Module

The `Cheap-io` module exports some basic, unextendable I/O functionality. Gwydion compilers use the `Cheap-io` functions internally. The Gwydion Project also provides the `Streams`, `Print`, and `Format` libraries (see the `$INSTALL/doc/libraries/` directory for documentation). It is an error to use both `Cheap-IO` functions and `Streams`/`Print`/`Format` functions on a single stream. (For example, if you are using the `Streams` library ***standard-input***, do not also use the `Cheap-io` input functions) For this reason, if any library that you load into Mindy uses the `Debugger-format` library, the debugger will use **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. Some Gwydion compilers 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 DRM.

print [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.

9. Extensions to the Streams Library

The `Piped-exec` module is a Gwydion extension to the `Streams` library. It exports the following function:

piped-exec [Function]

Arguments

command :: <string>

Values

writeable-stream :: <stream>

readable-stream :: <stream>

Description

piped-exec spawns a child process whose standard input and standard output are obtained from the parent process. Specifically, the parent's *writeable-stream* is attached to the child's standard input, and the parent's *readable-stream* is attached to the child's standard output. The *readable-stream* will be closed when the child closes its standard output, or when the child process terminates. The child process will be spawned by executing the command line *command*. Unlike the **system** command, the command line is not passed to a shell, so no wildcard expansion will be done. Like **system**, however, if the command name is not a fully qualified path, the executable file will be searched for.

Caveat: On Microsoft Windows machines, the command may not be a batch file.

10. Copyright and Terms of Use

Copyright (c) 1994, 1995, 1996, 1997 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`.

The Streams Library

Authors: Scott McKay, Bill Chiles, Marc Ferguson, and Eliot Miranda.

Note: This document was written by Harlequin, Inc., and not by the Gwydion Project.

Editor: Andrew Shires.

Date: 06 Feb 1997.

0.1 About this document

0.1.1 Acknowledgments

We'd like to thank the other people who have been instrumental in the production of this proposal: Jonathan Bachrach, Dave Berry, John Dunning, Chris Fry, Paul Haahr, William Lott, Rob Maclachlan, Tim McNerney, Tony Mann, Keith Playford, Robert Stockton, and Tucker Withington.

0.1.2 Discussing error conditions

This document uses two special terms in discussions of error conditions.

When we say that something *is an error*, we mean that the result is undefined. In particular, we do *not* mean that a Streams implementation must signal an error condition; that is the implementor's choice. So, for instance, the following text,

from page 79, means only that the result of using **unread-element** in the case described is undefined:

> It is an error to apply **unread-element** to an element that is not the element most recently read from the stream.

Only when we specifically mention signaling do we mean that a Streams implementation must signal an error condition. Note that we may not, in such a case, say exactly which error condition must be signaled; if we do not say so, the choice is again up to the implementor. In this text from the description of **stream-position-setter** on page 77, for instance, we state that an implementation must signal an error, but we do not say what error must be signaled:

> When *position* is a **<stream-position>**, if it is invalid for some reason, this function signals an error.

By contrast, the following text from the description of **read-element** on page 67 says exactly which error must be signaled:

> If the end of the stream is encountered and no value was supplied for *on-end-of-stream*, **read-element** signals an **<end-of-stream-error>** condition.

0.2 Goals of the library

The Dylan Streams library aims to provide:

- A generic, easy-to-use interface for streaming over sequences and files. The same high-level interface for consuming or producing is available irrespective of the type of stream, or the types of the elements being streamed over.
- Efficiency, especially for the common case of file I/O.
- Access to an underlying buffer management protocol.
- An extensible framework. Other areas of functionality that require a stream interface should be easy to integrate with the library.

The proposal presents the design of a Streams library that meets these goals using Dylan's built-in sequences and a buffered disk file interface.

The proposal does not address a number of related issues, including:

- A standard object-printing package such as Smalltalk's **`printOn:`** or Lisp's **`print-object`**, or a formatted printing facility such as Lisp's **`format`**. Additional libraries are expected to provide these facilities.
- General object dumping and loading.
- A comprehensive range of I/O facilities for using memory-mapped files, network connections, and so on. Such facilities should be easy to add to the Streams library because of its extensible framework.
- An interface for naming files.
- An interface to operating system functionality, such as file renaming or deleting operations.

0.3 Concepts

A *stream* provides sequential access to an aggregate of data, such as a Dylan sequence or a disk file. Streams grant this access according to a metaphor of *reading* and *writing*: elements can be read from streams or written to them.

Streams are represented as Dylan objects, and all are general instances of the class **`<stream>`**, which the Streams library defines.

We say that a stream is established *over* the data aggregate. Hence, a stream providing access to the string **`"hello world"`** is said to be a stream over the string **`"hello world"`**.

Streams permitting reading operations are called *input* streams. Input streams allow elements from the underlying data aggregate to be consumed. Conversely, streams permitting writing operations are called *output* streams. Output streams allow elements to be written to the underlying data aggregate. Streams permitting both kinds of operations are called *input-output* streams.

The library provides a set of functions for reading elements from an input stream. These functions hide the details of indexing, buffering, and so on. For instance, the function **`read-element`** reads a single data element from an input stream.

The following expression binds **`stream`** to an input stream over the string **`"hello world"`**:

```
let stream = make(<string-stream>, contents: "hello world");
```

The first invocation of **read-element** on **stream** returns the character 'h', the next invocation 'e', and so on. Once a stream has been used to consume all the elements of the data, the stream is said to be at its end. This condition can be tested with the function **stream-at-end?**. The following code fragment applies **function** to all elements of the sequence:

```
let stream = make(<sequence-stream>, contents: seq);
while (~stream-at-end?(stream))
  function(read-element(stream));
end;
```

When all elements of a stream have been read, further calls to **read-element** result in the **<end-of-stream-error>** condition being signalled. An alternative end-of-stream behavior is to have a distinguished end-of-stream value returned. You can supply such an end-of-stream value as a keyword argument to the various read functions; the value can be any object. Supplying an end-of-stream value to a read function is more efficient than asking whether a stream is at its end on every iteration of a loop.

The library also provides a set of functions for writing data elements to an output stream. Like the functions that operate upon input streams, these functions hide the details of indexing, growing an underlying sequence, buffering for a file, and so on. For instance, the function **write-element** writes a single data element to an output stream.

The following forms bind **stream** to an output stream over an empty string and create the string "I see!", using the function **stream-contents** to access all of the stream's elements.

```
let stream = make(<byte-string-stream>, direction: #"output");
write-element(stream, 'I');
write-element(stream, ' ');
write(stream, "see");
write-element(stream, '!');
stream-contents(stream);
```

Calling **write** on a sequence has the same effect as calling **write-element** on all the elements of the sequence. However, it is not required that **write** be implemented directly in terms of **write-element**; it might be implemented more efficiently, especially for buffered streams.

Some streams are *positionable*; that is, they permit random access to their elements. Postionable streams allow you to set the position at which the stream

will be accessed by the next operation. The following example uses positioning to return the character 'w' from a stream over the string **"hello world"**:

```
let stream = make(<string-stream>, contents: "hello world");
stream-position(stream) := 6;
read-element(stream);
```

The following example returns a string, but the contents of the first ten characters are undefined:

```
let stream = make(<string-stream>, direction: #"output");
adjust-stream-position(stream, 10);
write(stream, "whoa!");
stream-contents(stream);
```

You can request a sequence containing all of the elements of a positionable stream by calling **stream-contents** on it. The sequence returned never shares structure with any underlying sequence that might be used in future by the stream. For instance, the string returned by calling **stream-contents** on an output **<string-stream>** will not be the same string as that being used to represent the string stream.

When making an input **<string-stream>**, you can cause the stream to produce elements from any subsequence of the supplied string. For example:

```
read-to-end(make(<string-stream>,
                 contents: "hello there, world",
                 start: 6,
                 end: 11));
```

This example evaluates to **"there"**. The interval (*start*, *end*) includes the index *start* but excludes the index *end*. This is consistent with standard Dylan functions over sequences, such as **copy-sequence**. The **read-to-end** function is one of a number of convenient utility functions for operating on streams and returns all the elements up to the end of the stream from the stream's current position.

0.3.1 Streams, growing sequences, and object identity

When writing to output streams over sequences, Dylan may from time to time need to grow the underlying sequence that it is using to represent the stream data.

Consider the example of an output stream instantiated over an empty string. As soon as a write operation is performed on the stream, it is necessary to replace the string object used in the representation of the string stream. As well as incurring the cost of creating a new string, the replacement operation can affect the integrity of other references to the string within the program.

To guarantee that alias references to a sequence used in an output **<sequence-stream>** will have access to any elements written to the sequence via the stream, supply a **<stretchy-vector>** to **make**. A stream over a stretchy vector will use the same stretchy vector throughout the stream's existence.

For example:

```
let sv = make(<stretchy-vector>);

let stream = make(<sequence-stream>,
                  contents: sv,
                  direction: #"output");

write(stream,#(1, 2, 3, 4, 5, 6, 7, 8, 9));

write(stream,"ABCDEF");

values(sv, stream-contents(stream));
```

The example returns two values. Each value is the same (**\==**) stretchy vector:

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 'A', 'B', 'C', 'D', 'E', 'F')
```

If a stretchy vector is not supplied, the result is different:

```
let v = make(<vector>, size: 5);

let stream = make(<sequence-stream>,
                  contents: v,
                  direction: #"output");

write(stream,#(1, 2, 3, 4, 5, 6, 7, 8, 9));

write(stream,"ABCDEF");

values(v, stream-contents(stream));
```

This example returns as its first value the original vector, whose contents are undefined, but the second value is a new vector:

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 'A', 'B', 'C', 'D', 'E', 'F')
```

This difference arises because the output stream in the second example does not use a stretchy vector to hold the stream data. A vector of at least 15 elements is necessary to accommodate the elements written to the stream, but the vector supplied, **v**, can hold only 5. Since the stream cannot change **v**'s size, it must allocate a new vector each time it grows.

0.4 Stream classes

The exported streams class heterarchy is as follows:

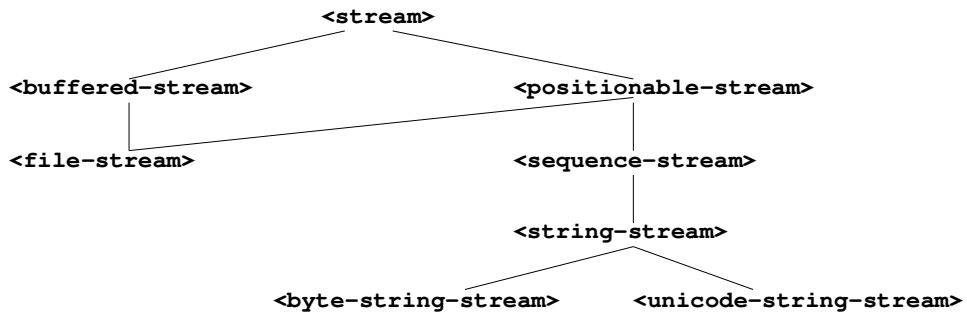


Figure 1 Streams library classes.

Except for the classes **<stream>**, **<buffered-stream>**, and **<positionable-stream>**, these are instantiable classes.

<stream> *Open abstract class*

The superclass of all stream classes and a direct subclass of **<object>**. It is not instantiable.

<positionable-stream> *Open abstract class*

A subclass of **<stream>** supporting the Positionable Stream Protocol. It is not instantiable.

<buffered-stream>

Open abstract class

A subclass of **<stream>** supporting the Stream Extension Protocol and the Buffer Access Protocol. It is not instantiable.

Streams of this class support the **buffer-size:** init-keyword, which can be used to suggest the size of the stream's buffer. However, the instantiated stream might not use this value: it is taken purely as a suggested value. For example, a stream that uses a specific device's hardware buffer might use a fixed buffer size regardless of the value passed with the **buffer-size:** init-keyword.

<file-stream>

Open abstract class

The class of streams over disk files. It is a subclass of **<positionable-stream>** and **<buffered-stream>**.

When you instantiate this class, an implementation-dependent, indirect instance of it is created. The file being streamed over is opened immediately upon creating the stream.

The class supports several init-keywords: **locator:**, **direction:**, **if-exists:**, and **if-does-not-exist:**.

<sequence-stream>

Open instantiable class

The class of streams over sequences. It is a subclass of **<positionable-stream>**.

The **<sequence-stream>** class can be used for streaming over all sequences, but there are also subclasses **<string-stream>**, **<byte-string-stream>**, and **<unicode-string-stream>**, which are specialized for streaming over strings.

The class supports several init-keywords: **contents:**, **direction:**, **start:**, and **end:**.

<string-stream>

Open instantiable class

The class of streams over strings. It is a subclass of **<sequence-stream>**.

The class supports the same init-keywords as **<sequence-stream>**.

<byte-string-stream> *Open instantiable class*

The class of streams over byte strings. It is a subclass of **<string-stream>**.

The class supports the same init-keywords as **<sequence-stream>**.

<unicode-string-stream> *Open instantiable class*

The class of streams over Unicode strings. It is a subclass of **<string-stream>**.

The class supports the same init-keywords as **<sequence-stream>**.

0.4.1 Creating streams

The following functions are used to create streams.

0.4.1.1 File streams

File streams are intended only for accessing the contents of files. They are not intended to provide a general file handling facility of renaming, deleting, moving, parsing directory names and so on.

make *file-stream-class* *G.f method*

make *file-stream-class* **#key** *locator direction if-exists if-does-not-exist buffer-size element-type encoding*
=> *file-stream-instance*

Creates and opens a stream over a file.

Returns a new instance of a concrete subclass of **<file-stream>** that streams over the contents of the file referenced by *locator*. To determine the concrete subclass to be instantiated, this method calls the generic function **type-for-file-stream** (see page 62.)

The *file-stream-class* argument is the class **<file-stream>**.

The *locator* init-keyword should be a string naming a file. If the Locators library is in use, *locator* should be an instance of **<locator>** or a string that can be coerced to one.

The *direction* init-keyword specifies the direction of the stream.

The *if-exists* and *if-does-not-exist* init-keywords specify actions to take if the file named by *locator* does or does not already exist when the stream is created. These init-keywords are discussed in more detail in Section 0.4.1.2 on page 62.

The *buffer-size* init-keyword is explained in **<buffered-stream>**, page 60.

The *element-type* init-keyword specifies the type of the elements in the file named by *locator*. This allows file elements to be represented abstractly; for instance, contiguous elements could be treated as a single database record. This init-keyword defaults to something useful, potentially based on the properties of the file; **<byte-character>** and **<unicode-character>** are likely choices. See Section 0.4.1.2 on page 62.

type-for-file-stream *Open generic function*

type-for-file-stream *locator element-type* **#rest** *all-keys* **=>** *file-stream-type*

Returns the kind of file-stream class to instantiate for a given file. The method for **make** on **<file-stream>** calls this function to determine the class of which it should create an instance.

0.4.1.2 Options when creating file streams

When creating file streams, you can can supply the following init-keywords to **make** in addition to the usual direction:

if-exists	An action to take if the file already exists.
if-does-not-exist	An action to take if the file does not already exist.
element-type	How the elements of the underlying file are accessed.

The *if-exists* init-keyword allows you to specify an action to take if the file named by *locator* already exists. The options are:

#f No action. This is the default when the stream's direction is **#"input"** or **#"input-output"**.

#"new-version"
If the underlying file system supports file versioning, a new version of the file is created. This is the default when the stream's direction is **#"output"**.

If the file system does not support file versioning, the implementation should substitute one of the other *if-exists* behaviors; the **#"replace"** behavior is a good choice.

#"overwrite" Set the stream's position to the beginning of the file, but preserve the current contents of the file. This is useful when the direction is **#"input-output"** and you want to overwrite an existing file.

#"replace" Delete or rename the existing file and create a new file.

#"append" Set the stream's initial position to the end of the existing file so that all new output occurs at the end of the file.

#"truncate" If the file exists, it is truncated, setting the size of the file to 0. If the file does not exist, create a new file.

#"signal" Signal a **<file-exists-error>** condition.

The *if-does-not-exist* init-keyword allows you to specify an action to take if the file named by *locator* does not exist. The options are:

#f No action.

#"signal" Signal a **<file-does-not-exist-error>** condition. This is the default when the stream's direction is **#"input"**.

#"create" Create a new zero-length file. This is the default when the stream's direction is **#"output"** or **#"input-output"**.

Because creating a file stream *always* involves an attempt to open the underlying file, the aforementioned error conditions will occur during file stream instance initialization.

If an implementation checks file permissions when creating and opening file streams, and it detects an attempt to read a file for which the user has no read permission, or to write a file for which the user has no write permission, then an **<invalid-file-permissions-error>** condition is signalled at the time the file stream is created.

The *element-type* init-keyword controls how the elements of the underlying file are accessed.

Implementation Note: Ideally, *element-type* could be any valid Dylan type such as **limited(<integer>, min: 0, max: 255)** or **<unicode-character>**. This approach may make it possible to implement a potentially inefficient but general set of file streams. Unfortunately the current language definition does not include adequate support for this approach, so we specify instead an interim minimum set of three element types. The element types are for the time being exported from the **streams** module of the Streams library.

The three possible element types are:

<byte-character>
: The file is accessed as a sequence of 8-bit characters.

<unicode-character>
: The file is accessed as a sequence of 16-bit Unicode characters.

<byte>
: The file is accessed as a sequence of unsigned 8-bit integers.

Portability Note: Portable code can count on the existence of these three element types, but implementations are free to provide more.

0.4.1.3 Sequence streams

There are **make** methods on **<sequence-stream>**, **<string-stream>**, **<byte-string-stream>** and **<unicode-string-stream>**. The **make** methods on **<sequence-stream>** and **<string-stream>** might not create direct instances of those classes, but instead an instance of a subclass determined by **type-for-sequence-stream**.

make *sequence-stream-class* *G.f. method*

make *sequence-stream-class* **#key** *contents direction start end*
=> *sequence-stream-instance*

Creates and opens a stream over a sequence.

This method returns a general instance of **<sequence-stream>**. To determine the concrete subclass to be instantiated, this method calls the generic function **type-for-sequence-stream** (see page 65).

The *sequence-stream-class* argument is the class **<sequence-stream>**.

The *contents* init-keyword is a general instance of **<sequence>** which is used as the input for input streams, and as the initial storage for an output stream. If *contents* is a stretchy vector, then it is the only storage used by the stream.

The *direction* init-keyword specifies the direction of the stream. It must be one of **#"input"**, **#"output"**, or **#"input-output"**; the default is **#"input"**.

The *start* and *end* init-keywords are only valid when *direction* is **#"input"**. They specify the portion of the sequence to create the stream over: *start* is inclusive and *end* is exclusive. The default is to stream over the entire sequence: *start* is by default 0, and *end* is *contents*`.size`.

type-for-sequence-stream *Open generic function*

type-for-sequence-stream *sequence* **=>** *sequence-stream-type*

Returns the sequence-stream class to instantiate over a given sequence object. The method for **make** on **<sequence-stream>** calls this function to determine the concrete subclass of **<sequence-stream>** that it should instantiate.

There are **type-for-sequence-stream** methods for each of the string object classes. These methods return a stream class object that the particular Streams implementation considers appropriate.

make *string-stream-class* *G.f. method*

make *string-stream-class* **#key** *contents direction start end*
=> *string-stream-instance*

Creates and opens a stream over a string.

This method returns an instance of **<string-stream>**. If supplied, *contents* must be an instance of **<string>**. The *string-stream-class* argument is the class **<string-stream>**. The *direction*, *start*, and *end* init-keywords are as for **make** on **<sequence-stream>**.

make *byte-string-stream-class* *G.f. method*

make *byte-string-stream-class* **#key** *contents direction start end*
=> *byte-string-stream-instance*

Creates and opens a stream over a byte string.

This method returns a new instance of **<byte-string-stream>**. If supplied, *contents* must be an instance of **<byte-string>**.

The *byte-string-stream-class* argument is the class **<byte-string-stream>**. The *direction*, *start*, and *end* init-keywords are as for **make** on **<sequence-stream>**.

make *unicode-string-stream-class* *G.f. method*

make *unicode-string-stream-class* **#key** *contents direction start end*
=> *unicode-string-stream-instance*

Creates and opens a stream over a Unicode string.

This method returns a new instance of **<unicode-string-stream>**. If supplied, *contents* must be an instance of **<unicode-string>**.

The *unicode-string-stream-class* argument is the class **<unicode-string-stream>**. The *direction*, *start*, and *end* init-keywords are as for **make** on **<sequence-stream>**.

0.4.2 Closing streams

When creating new stream classes it may be necessary to add a method to the **close** function, even though it is not part of the Stream Extension Protocol.

close *Open generic function*

close *stream* **#key #all-keys => ()**

Closes *stream*, an instance of **<stream>**.

0.5 Reading and writing from streams

It is an error to call any of these functions on a buffered stream while its buffer is held.

0.5.1 Reading from streams

The following are the basic functions for reading from streams. To implement a new input stream that is not a **<buffered-stream>**, you must provide methods for **read-element**, **stream-input-available?**, **peek**, **read**, **read-into!**, and **discard-input**. If you implement a new stream that is a **<positionable-stream>**, you might need to supply a new method for **unread-element**.

read-element *Open generic function*

read-element *input-stream* **#key** *on-end-of-stream* **=>** *element-or-eof*

Returns the next element in the stream. If the stream is not at its end, the stream is advanced so that the next call to **read-element** returns the next element along in *input-stream*.

The *on-end-of-stream* keyword allows you to specify a value to be returned if the stream is at its end. If the stream is at its end and no value was supplied for *on-end-of-stream*, **read-element** signals an **<end-of-stream-error>** condition.

If no input is available and the stream is not at its end, **read-element** blocks until input becomes available.

See also **unread-element**, page 79.

peek *Open generic function*

peek *input-stream* **#key** *on-end-of-stream* **=>** *element-or-eof*

Behaves as **read-element** does, but the stream position is not advanced.

read *Open generic function*

read *input-stream n* **#key** *on-end-of-stream* **=>** *sequence-or-eof*

Returns a sequence of the next *n* elements from *input-stream*.

The type of the sequence returned depends on the type of the stream's underlying aggregate. For instances of **<sequence-stream>**, the type of the result is given by **type-for-copy** of the underlying aggregate. For instances of **<file-stream>**, the result is a vector that can contain elements of the type returned by calling **stream-element-type** on the stream.

The stream position is advanced so that subsequent reads start after the *n* elements.

If the end of the stream is reached before all *n* elements have been read, the behavior is as follows.

If the *on-end-of-stream* argument is supplied, it is returned as the value of **read**.

If the *on-end-of-stream* argument was not supplied, and at least one element was read from the stream, then an **<incomplete-read-error>** condition is signalled. When signalling this condition, **read** supplies two values: a sequence of the elements that were read successfully, and *n*.

If the *on-end-of-stream* argument was not supplied, and no elements were read from the stream, an **<end-of-stream-error>** condition is signalled.

If the stream is not at its end, **read** blocks until input becomes available.

Implementation Note: Buffered streams are intended to provide a very efficient implementation of **read**, particularly when the result is an instance of **<byte-string>**, **<unicode-string>**, or **<byte-vector>**.

read-into! *Open generic function*

read-into! *input-stream n sequence* **#key** *start on-end-of-stream* **=>** *count-or-eof*

Reads the next *n* elements from *input-stream*, and inserts them into a mutable sequence starting at the position *start*. Returns the number of elements

actually inserted into *sequence* unless the end of the stream is reached, in which case the *on-end-of-stream* behavior is as for **read**.

If the sum of *start* and *n* is greater than the size of *sequence*, **read-into!** reads only enough elements to fill sequence up to the end. If *sequence* is a stretchy vector, no attempt is made to grow it.

If the stream is not at its end, **read-into!** blocks until input becomes available.

Implementation Note: Buffered streams are intended to provide a very efficient implementation of **read**, particularly when the result is an instance of **<byte-string>**, **<unicode-string>**, or **<byte-vector>**.

discard-input

Open generic function

discard-input *input-stream* **=> ()**

Discards any pending input from *input-stream*, both buffered input and, if possible, any input that might be at the stream's source.

This operation is principally useful for "interactive" streams, such as TTY streams, to discard unwanted input after an error condition arises. There is a default method on **<stream>** so that applications can call this function on any kind of stream. The default method does nothing.

stream-input-available?

Open generic function

stream-input-available? *input-stream* **=>** *available?*

Returns **#t** if *input-stream* would not block on **read-element**, otherwise it returns **#f**.

This function differs from **stream-at-end?**. When **stream-input-available?** returns **#t**, **read-element** will not block, but it may detect that it is at the end of the stream's source, and consequently inspect the **on-end-of-stream** argument to determine how to handle the end of stream.

0.5.2 Convenience functions for reading from streams

The following is a small set of convenient reading functions that search for particular elements in a stream. These functions behave as though they were implemented in terms of the more primitive functions described in Section 0.5.1.

read-to *Function*

`read-to` *input-stream element* **`#key`** *on-end-of-stream test* **`=>`** *sequence-or-eof found?*

Returns a new sequence containing the elements of *input-stream* from the stream's current position to the first occurrence of *element*. The result does not contain *element*.

The second return value is **`#t`** if the read terminated with *element*, or **`#f`** if the read terminated by reaching the end of the stream's source. The "boundary" element is consumed, that is, the stream is left positioned after *element*.

The **`read-to`** function determines whether the element occurred by calling the test function *test*, which defaults to **`\==`**. The test function must accept two arguments. The order of the arguments is the element retrieved from the stream first and *element* second.

The type of the sequence returned is the same that returned by **`read`**. The end-of-stream behavior is the same as that of **`read-line`**.

read-through *Function*

`read-through` *input-stream element* **`#key`** *on-end-of-stream test* **`=>`** *sequence-or-eof found?*

This function is the same as **`read-to`**, except that *element* is included in the resulting sequence.

If the *element* is not found, the result does not contain it. The stream is left positioned after *element*.

read-to-end *Function*

`read-to-end` *input-stream* **`=>`** *sequence*

Returns a sequence of all the elements up to, and including, the last element of *input-stream*, starting from the stream's current position.

The type of the result sequence is as described for **read**. There is no special end-of-stream behavior; if the stream is already at its end, an empty collection is returned.

skip-through *Function*

skip-through *input-stream element* **#key** *test* **=>** *found?*

Positions *input-stream* after the first occurrence of *element*, starting from the stream's current position. Returns **#t** if the element was found, or **#f** if the end of the stream was encountered. When **skip-through** does not find the *element*, it leaves *input-stream* positioned at the end.

The **skip-through** function determines whether the element occurred by calling the test function *test*, which defaults to **\==**. The test function must accept two arguments. The order of the arguments is the element retrieved from the stream first and element second.

0.5.3 Writing to streams

The following are the basic functions for writing to streams.

To implement a new output stream that is not a **<buffered-stream>**, you must provide methods for **write-element**, **write**, **force-output**, and **discard-output**.

write-element *Open generic function*

write-element *output-stream element* **=> ()**

Writes *element* to *output-stream* at the stream's current position. It is an error if the type of *element* is inappropriate for the strream's underlying aggregate.

If the stream is positionable, and it is not positioned at its end, **write-element** overwrites the element at the current position and then advance the stream position.

write *Open generic function*

write *output-stream sequence* **#key** *start end* **=> ()**

Writes the elements of *sequence* to *output-stream*, starting at the stream's current position.

The elements in *sequence* are accessed in the order defined by the forward iteration protocol on **<sequence>**. This is effectively the same as the following:

```
do (method (elt) write-element(stream, elt) end, sequence);
sequence;
```

If supplied, *start* and *end* delimit the portion of *sequence* to write to the stream. The value of *start* is inclusive and that of *end* is exclusive. They default to 0 and **sequence.size**, respectively.

If the stream is positionable, and it is not positioned at its end, **write** overwrites elements in the stream and then advance the stream's position to be beyond the last element written.

Implementation Note: Buffered streams are intended to provide a very efficient implementation of **write**, particularly when *sequence* is an instance of **<byte-string>**, **<unicode-string>**, **<byte-vector>**, or **<buffer>**, and the stream's element type is the same as the element type of *sequence*.

force-output *Open generic function*

force-output *output-stream* **=> ()**

Forces any pending output from *output-stream*'s buffers to its destination.

When creating new stream classes it may be necessary to add a method to the **force-output** function, even though it is not part of the Stream Extension Protocol.

synchronize-output *Open generic function*

synchronize-output *output-stream* **=> ()**

Forces any pending output from *output-stream*'s buffers to its destination. Before returning to its caller, **synchronize-output** also attempts to ensure that the output reaches the stream's destination before, thereby synchronizing the output destination with the application state.

When creating new stream classes it may be necessary to add a method to the **synchronize-output** function, even though it is not part of the Stream Extension Protocol.

discard-output *Open generic function*

discard-output *output-stream* **=> ()**

Attempts to abort any pending output for *output-stream*.

A default method on **<stream>** is defined, so that applications can call this function on any sort of stream. The default method does nothing.

0.5.4 Reading and writing by lines

The following functions facilitate line-based input and output operations.

The newline sequence for string streams is a sequence comprising the single newline character **\n**. For character file streams, the newline sequence is whatever sequence of characters the underlying platform uses to represent a newline. For example, on MSDOS platforms, the sequence might comprise two characters: a carriage return followed by a linefeed.

Implementation Note: The functions described in this section are potentially an interim solution to one aspect of the more general problem of encoding and data translation. At some point, these functions may be moved into, or subsumed by, another higher level library that deals with the encoding problems in a better way. Note that no other functions in the Streams library do anything to manage the encoding of newlines; calling **write-element** on the character **\n** does not cause the **\n** character to be written as the native newline sequence, except by coincidence.

read-line *Open generic function*

read-line *input-stream* **#key** *on-end-of-stream* **=>** *string-or-eof newline?*

Returns a new string containing all the input in *input-stream* up to the next newline sequence.

The resulting string does not contain the newline sequence. The second value returned is **#t** if the read terminated with a newline or **#f** if the read terminated because it came to the end of the stream.

The type of the result string is chosen so that the string can contain characters of *input-stream*'s element type. For example, if the element type is **<byte-character>**, the string will be a **<byte-string>**.

If *input-stream* is at its end immediately upon calling **read-line** (that is, the end of stream appears to be at the end of an empty line), then the end-of-stream behavior and the interpretation of *on-end-of-stream* is as for **read-element**.

read-line-into! *Open generic function*

read-line-into! *input-stream string* **#key** *start on-end-of-stream grow?* **=>** *string-or-eof newline?*

Fills *string* with all the input from *input-stream* up to the next newline sequence. The *string* must be a general instance of **<string>** that can hold elements of the stream's element type.

The input is written into *string* starting at the position *start*, which is an integer defaulting to 0.

The second return value is **#t** if the read terminated with a newline, or **#f** if the read completed by getting to the end of the input stream.

If *grow?* is **#t**, and *string* is not large enough to hold all of the input, this function takes one of the two following actions.

If *string* is stretchy, **read-line-into!** grows it enough to hold the input.

If *string* is not stretchy, **read-line-into!** creates a new string which it writes to and returns instread. The resulting string holds all the original elements of *string*, except where read-line-into! overwrites them with input from *input-stream*.

In a manner consistent with the intended semantics of *grow?*, when *grow?* is true and *start* is greater than or equal to *string* **.size**, **read-line-into!** grows *string* to accomodate the *start* index and the new input.

If *grow?* is **#f** (the default) and *string* is not large enough to hold the input, the function signals an error.

The end-of-stream behavior and the interpretation of *on-end-of-stream* is the same as that of **read-line**.

write-line *Open generic function*

write-line *output-stream string* **#key** *start end* **=> ()**

Writes *string* followed by a newline sequence to *output-stream*.

The default method behaves as though it calls **write** on *string* and then calls **new-line**, with *output-stream* locked across both calls.

If supplied, *start* and *end* delimit the portion of *string* to write to the stream. They default to 0 and *string* **.size** respectively.

new-line *Open generic function*

new-line *output-stream* **=> ()**

Writes a newline sequence to *output-stream*.

A method for **new-line** is defined on **<string-stream>** that writes the character **\n** to the string stream.

0.5.5 Querying streams

The following functions can be used to determine various properties of a stream.

To implement a new stream you must provide methods for **stream-open?**, **stream-at-end?**, and **stream-element-type**.

stream-open? *Open generic function*

stream-open? *stream* **=>** *open?*

Returns **#t** if *stream* is open and **#f** if it is not.

stream-element-type *Open generic function*

stream-element-type *stream* **=>** *element-type*

Returns the element type of *stream* as a Dylan **<type>**.

stream-at-end? *Open generic function*

stream-at-end? *stream* **=>** *boolean*

Returns **#t** if the stream is at its end and **#f** if it is not. For input streams, it returns **#t** if a call to **read-element** with no supplied keyword arguments would signal an **<end-of-stream-error>**.

This function differs from **stream-input-available?**. See the description of that function on page 69.

For output-only streams, this function always returns **#f**.

0.5.6 Positionable stream protocol

The following comprises the protocol for positionable streams.

To implement a new positionable stream you must provide methods for **stream-position**, **stream-position-setter**, **adjust-stream-position**, **stream-size**, **stream-contents**, and **unread-element**.

A stream position can be thought of as a natural number that indicates how many elements into the stream the stream's current location is. However, it is not always the case that a single integer contains enough information to reposition a stream. Consider the case of an "uncompressing" file stream that requires additional state beyond simply the file position to be able to get the next input character from the compressed file.

The Streams library addresses this problem by introducing the class **<stream-position>**, which is subclassed by various kinds of stream implementations that need to maintain additional state. A stream can be repositioned as efficiently as possible when **stream-position-setter** is given a value previously returned by **stream-position** on that stream.

It is also legal to set the position of a stream to an integer position. However, for some types of streams, to do so might be slow, perhaps requiring the entire contents of the stream up to that point to be read.

<stream-position> *Abstract class*

A direct subclass of **<object>**. It is used to represent positions within streams of a certain kind: those for which a natural number is not sufficient to fully describe the position in the stream. For example, a stream that supports compression will have some state associated with each position in the stream that a single integer is not sufficient to represent.

The **<stream-position>** class is disjoint from the class **<integer>**.

stream-position *Open generic function*

stream-position *positionable-stream* **=>** *position*

Returns the current position of *positionable-stream* for reading or writing.

The value returned can be either an instance of **<stream-position>** or an integer. When the value is an integer, it is an offset from position zero, and is in terms of the stream's element type. For instance, in a Unicode stream, a position of 4 means that 4 Unicode characters have been read.

stream-position-setter *Open generic function*

stream-position-setter *position positionable-stream* **=>** *new-position*

Changes the stream's position for reading or writing to *position*.

The position can be either an integer or a **<stream-position>**.

When it is an integer, if it is less than zero or greater than *positionable-stream* **.stream-size** this function signals an error.

When *position* is a **<stream-position>**, if it is invalid for some reason, this function signals an error. Streams are permitted to restrict the *position* to being a member of the set of values previously returned by calls to **stream-position** on the same stream.

The *position* may also be **#"start"**, meaning that the stream should be positioned at its start, or **#"end"**, meaning that the stream should be positioned at its end.

adjust-stream-position

Open generic function

adjust-stream-position *positionable-stream delta* **#key** *from* **=>** *new-position*

Moves the position of *positionable-stream* to be offset *delta* elements from the position indicated by *from*. The new position is returned.

The value of *from* can be one of the symbols **#"current"**, **#"start"**, and **#"end"**. The default value is **#"current"**. When *from* is **#"start"**, the stream is positioned relative to the beginning of the stream. When *from* is **#"end"**, the stream is positioned relative to its end.

Using **adjust-stream-position** to set the position of a stream to be beyond its current last element causes the underlying aggregate to be grown to a new size. When extending the underlying aggregate for a stream, it is undefined what the unwritten elements of the aggregate will contain.

as

G.f. method

as *integer-class stream-position* **=>** *integer*

Coerces a **<stream-position>** to an integer. The *integer-class* argument is the class **<integer>**.

stream-size

Open generic function

stream-size *positionable-stream* **=>** *size*

Returns the number of elements in *positionable-stream*.

For input streams, this number is the number of elements that were available when the stream was created. It is unaffected by any read operations that might have been performed on the stream.

For output and input-output streams, this number is the number of elements that were available when the stream was created (just as with input

streams), added to the number of elements written past the end of the stream (regardless of any repositioning operations).

It is assumed that there is no more than one stream open on the same source or destination at a time; that there are no shared references to files by other processes; that there are no alias references to underlying sequences, or any other such situations. In such situations, the behavior of **stream-size** is undefined.

stream-contents

Open generic function

stream-contents *positionable-stream* **#key** *clear-contents?* **=>** *sequence*

Returns a sequence that contains all of *positionable-stream*'s elements from its start to its end, regardless of its current position. The type of the returned sequence is as for **read**. See page 68.

Note: For streams whose direction is **#"input"**, **stream-contents** signals an error. Use **read-to-end** instead on such streams. See page 70.

If *clear-contents?* is **#t** (the default), this function sets the size of the stream to zero, and the position to the stream's start. Thus the next call to **stream-contents** will return only the elements written after the previous call to **stream-contents**.

unread-element

Open generic function

unread-element *positionable-stream element* **=>** *element*

"Unreads" the last element from *positionable-stream*. That is, it returns *element* to the stream so that the next call to **read-element** will return *element*. The stream must be a **<positionable-stream>**.

It is an error (**i**) to apply **unread-element** to an element that is not the element most recently read from the stream; (**ii**) to call **unread-element** twice in succession; (**iii**) to unread an element if the stream is at its initial positionl; and (**iv**) to unread an element after explicitly setting the stream's position.

0.5.7 Using file streams

The following are operations that pertain to file streams.

close *G.f. method*

close *file-stream* **#key** *abort wait?*

Closes a file stream. This method frees whatever it can of any underlying system resources held on behalf of the stream.

If *abort* is false (the default), any pending data is forced out and synchronized with the file's destination. If *abort* is true, then any errors caused by closing the file are ignored. Furthermore, if *abort* is true, the file should be restored to its initial state if possible (for example, on a versioned file system, the previous version of the file should be restored as the latest version).

with-open-file *Macro*

with-open-file (*stream-var* **=** *locator*, **#rest** *keys***)** *body* **end**

Provides a safe mechanism for working with file streams. The macro creates a file stream and binds it to *stream-var*, evaluates a *body* of code within the context of this binding, and then closes the stream. The macro calls **close** upon exiting *body*.

The *locator* argument should evaluate to a valid argument to

as(<locator>, *locator***)**

The values of the last expression in *body* are returned.

The *keys* are passed to the **make** method on **<file-stream>**.

For example, the following expression yields the contents of file **foo.text** as a **<byte-vector>**:

```
with-open-file (fs = ("foo.text", element-type: <byte>))
  read-to-end(fs)
end;
```

It is roughly equivalent to:

```
begin
  let hidden-fs = #f;     // In case the user bashes fs variable
  block ()
    hidden-fs := make(<file-stream>,
                      locator: "foo.text", element-type: <byte>);
    let fs = hidden-fs;
    read-to-end(fs);
  cleanup
    if (hidden-fs) close(hidden-fs) end;
  end block;
end;
```

0.6 Locking streams

Stream locks have multilocking semantics. That is, a single thread can lock a stream more than once, but the thread must unlock the stream for each time it locked the stream. Furthermore, threads waiting for a stream lock are expected to do so by blocking, not by "spinning".

This allows a high-level printing routine to lock a stream across several calls to output functions, ensuring all the output is contiguous at the stream's destination. For example, the **write-line** function locks its stream argument and then calls the **write** and **new-line** functions. The **write** function locks its stream argument by calling **get-output-buffer**, but because of the multilocking semantics, the call to **write** within **write-line** does not block waiting for a lock. The same thing happens with **new-line**. Before returning, **write-line** unlocks the stream so that other routines may call output functions on the stream or get the stream's buffer for direct manipulation.

The Locking Protocol isolates access to a stream so that only one thread may use the stream at any time. In a single-threaded Dylan implementation, these functions do nothing.

The Buffer Access Protocol functions that get a buffer first lock the stream, and those functions that release the buffer unlock the stream. Thus, getting a buffer both isolates access to the stream for a single thread and ensures that the single thread does not try to get the stream's buffer multiple times while already holding the buffer.

The Buffer Access Protocol isolates access to a buffer within a single thread to prevent reentrancy problems and programming mistakes. Essentially, the lightweight buffer locking ensures that applications do not call output functions that

directly manipulate a stream's buffer from within routines that are already directly manipulating the stream's buffer. This situation must be forbidden because the inner call to get the buffer cannot reliably return the state of the stream's buffer while the application already holds the buffer.

stream-locked? *Function*

stream-locked? *stream* **=>** *boolean*

Returns **#t** if *stream* is locked and **#f** if it is not.

lock-stream *Function*

lock-stream *stream* **=> ()**

Locks *stream*.

If *stream* is already locked by another thread, this function waits until it gets the lock.

In a single-threaded Dylan implementation, this function does nothing.

unlock-stream *Function*

unlock-stream *stream* **=> ()**

Releases *stream*'s lock.

In a multi-threaded Dylan implementation, stream locks can be taken more than once by a thread, so they must be unlocked for each time they are locked. This function signals an error if it is invoked in order to release a stream's lock and the current thread does not actually hold that lock.

In a single-threaded Dylan implementation, this function does nothing.

with-stream-locked *Macro*

with-stream-locked (*stream***)** *body* **end**

Evalutes *body* in a context in which *stream* is locked.

Upon exiting the body, the code resulting from the macro unlocks the stream once, as a **block** cleanup.

0.7 Using buffered streams

A goal of the streams library is to provide efficient support for general use of buffered I/O. At the same time, programmers using the library should not need to be concerned with buffering in most cases. For most uses of buffered streams, the buffering is transparent, but programs requiring more control can access buffering functionality when appropriate. This section describes the available buffering functionality.

0.7.1 Overview

A buffered stream maintains some sort of buffer. All buffered streams use the sealed class **<buffer>** for their buffers. You can suggest a buffer size when creating buffered streams, but normally you do not need to do so. Streams implementations should choose buffer sizes that are appropriate for the stream's source or destination.

Instances of the class **<buffer>** also contain some state information. This state information includes an index where reading or writing should begin, and an index that is the end of input to be read, or the end of space available for writing.

Buffered streams also maintain a *held* state, indicating whether the application has taken the buffer for a stream and has not released it yet. When a thread already holds the buffer for a stream, it is an error to get the buffer again (or any other buffer for the same stream); this property holds for the one thread of a single-threaded Dylan implementation as well as for multi-threaded implementations.

0.7.2 Buffer access protocol

This section describes the functions that users of buffered streams invoke to manipulate stream buffers directly. These functions use corresponding functions from the Stream Extension Protocol to do their work. For example, **get-input-buffer** calls **do-get-input-buffer**. All Stream Extension Protocol func-

tions are named with a **do-** prefix on the corresponding Buffer Access Protocol function. Users should never call the Stream Extension Protocol directly.

Threaded Dylan implementations should place system-dependent mutual exclusion calls in the Buffer Access Protocol functions. All streams implementations are encouraged to check the buffer-held state in the Buffer Access Protocol functions, and these functions should signal an error if the buffer is already held. The separation of the Buffer Access Protocol and the Stream Extension Protocol allows users to more portably extend the buffered stream protocol to new stream types. Programmers avoid the following design and maintenance hassles:

- Deciding whether to write for a threaded or non-threaded Dylan implementation.
- Rewriting mutual exclusion code for each Dylan implementation.
- Having to duplicate mutual exclusion code wherever the application uses stream buffers directly.

Programs that manipulate buffers are entirely responsible for tracking the buffer's state and informing the stream of any changes to the buffer's state. The technique for using a buffer is to get the input or output buffer and its bounds, consume or produce data, update the buffer's state, and then release the buffer. The new bounds are indicated by calling **buffer-next-setter** and **buffer-end-setter**. Using buffers directly interoperates with using higher-level functions such as **read-element**, **read**, **write-element**, **write**, and so on. This is because higher-level stream operations are implemented in terms of the Buffer Access Protocol for instances of **<buffered-stream>**.

0.7.2.1 Useful types when using buffers

<byte> *Type*

A type representing limited integers in the range 0 to 255 inclusive.

<byte-character> *Type*

A type representing 8-bit characters that instances of **<byte-string>** can contain.

<unicode-character> *Type*

A type representing Unicode characters that instances of **<unicode-string>** can contain.

<byte-vector> *Type*

A subtype of **<vector>** whose element-type is **<byte>**.

<buffer> *Sealed instantiable class*

A subclass of **<vector>** whose **element-type** is **<byte>**.

Instances of **<buffer>** contain a data vector and two indices: the inclusive start and the exclusive end of valid data in the buffer. The accessors for these indexes are called **buffer-next** and **buffer-end**. This class supports three init-keywords: **size:**, **next:** (defaults to zero), and **end:** (defaults to zero).

Note that **size:** is not taken as a suggestion of the size the user would like, as with the value passed with **buffer-size:** to **make** on **<buffered-stream>**; if you supply a value with the **size:** init-keyword, that size is allocated, or, if that is not possible, an error is signalled, as with making any vector.

<buffer-index> *Type*

A subtype of **<integer>**.

All buffer-index return values and parameters are of this type. See also Harlequin and CMU's proposal for **<integer>**.

0.7.2.2 Using buffers for input

get-input-buffer *Function*

get-input-buffer *buffered-stream*, **#key** *wait?*, *bytes* **=>** *buffer-or-false*

Calls **do-get-input-buffer** to return a buffer for the stream. You should never call **do-get-input-buffer** directly.

The stream extension function signals an error if *buffered-stream* is an output-only stream, or if it cannot return a buffer for any reason.

The *wait?* keyword is a **<boolean>** and defaults to **#t**. It indicates whether **get-input-buffer** should ensure that there is valid data in the input buffer, and block for input if necessary.

When *wait?* is **#f**, this function returns the stream's current buffer in whatever state it is in.

When *wait?* is **#t**, **do-get-input-buffer** waits until there is some valid data in the buffer. If **do-get-input-buffer** needs to wait for valid data, and it encounters the end of the stream before any valid data arrives, then **do-get-input-buffer** returns **#f**.

When *bytes* is supplied, *wait?* has no effect. The *bytes* keyword is an **<integer>** or **#f**, and is the minimum number of bytes that must be present in the buffer when it is returned. If **do-get-input-buffer** cannot return a buffer with the specified minimum number of bytes, it signals an **<incomplete-read-error>**.

Values supplied for *bytes* should be small relative to a buffer's size. If you need to read large numbers of bytes at once, specify a large buffer size when making the stream and do not specify any values for the *bytes* keyword parameter. When *bytes* is not **#f**, **get-input-buffer** may block for input if necessary.

If an application thread calls **get-input-buffer**, and another thread already holds any buffer the stream owns, then this function might block. Multi-threaded implementations should eventually return. In multi-threaded implementations this function calls lock-stream before calling do-get-input-buffer, and it does not release the lock.

If an application's thread calls this function, and that same thread already holds the buffer, this situation is a re-entrancy programmer error. The stream cannot return the buffer and guarantee that its state is up to date with input consumption that occurred between the first call to this function and the current call. Implementations should detect and signal this error.

release-input-buffer *Function*

release-input-buffer *buffered-stream* **=> ()**

Call this function to announce that you have finished using the stream's current buffer. There may still be valid input in the buffer even though you have finished with it.

This function calls **do-release-input-buffer** so that the stream can perform any record-keeping or management tasks it has. You should never call **do-release-input-buffer** directly.

After consuming input from the buffer, you must update the buffer's state by assigning to **buffer-next** before releasing the buffer. When a user of the Buffer Access Protocol sets **buffer-end**, the results are undefined.

If the calling application does not hold the buffer, this function signals an error.

In multi-threaded implementations, **release-input-buffer** releases the lock obtained in **get-input-buffer**.

with-input-buffer *Macro*

with-input-buffer (*buffer-var* **=** *exp*, **#key** *wait?*, *bytes***)** *body* **end;**

Calls **get-input-buffer** on *exp*, *wait?*, and *bytes*, binding *buffer-var* to the result, then executes *body* within the scope of the buffer binding.

After the *body* has executed, **with-input-buffer** calls **release-input-buffer** and returns any values returned by *body*.

next-input-buffer *Function*

next-input-buffer *buffered-stream* **#key** *wait?*, *bytes* **=>** *buffer-or-false*

Call this function to get more input when you are already holding *buffered-stream*'s buffer. If the application does not hold the stream's buffer, this function signals an error.

This function calls **do-next-input-buffer** to get more input. You should never call **do-next-input-buffer** directly.

You must update the buffer's state by assigning to **buffer-next** before getting the next buffer; otherwise, it might seem that the current buffer still has valid input in it. When **buffer-next** = **buffer-end** there is no valid input remaining in the current buffer.

If a user of the Buffer Access Protocol sets **buffer-end** while holding the stream's buffer, the results are undefined.

The stream extension function may return a new **<buffer>** object or the current one with **buffer-next** and **buffer-end** updated to reflect the location of the new input data. You should not make any assumptions about which option a stream's implementation uses.

The **do-next-input-buffer** function signals an error when the current buffer is not held for input.

The *wait?* and *bytes* arguments are the same as for **get-input-buffer**.

input-available-at-source? *Function*

input-available-at-source? *buffered-stream* **=>** *available?*

Returns **#t** when *buffered-stream*'s source has any available input or when the stream is at the end of its source. If this function returns **#t**, the next call to **next-input-buffer** will not block.

This function calls **do-input-available-at-source?** to get more input. You should never call **do-input-available-at-source?** directly.

Call this function while you are holding a stream's buffer. If the application does not hold the stream's buffer, this function signals an error.

The stream extension function signals an error when the current buffer is not held for input.

0.7.2.3 Using buffers for output

get-output-buffer *Function*

get-output-buffer *buffered-stream*, **#key** *bytes* **=>** *buffer*

Calls **do-get-output-buffer** to return a buffer for *buffered-stream*. You should never call **do-get-output-buffer** directly.

The stream extension function signals an error if the stream is an input only stream, or if it cannot return a buffer for any reason. The resulting buffer is never completely full of pending output.

The *bytes* keyword is an **<integer>**, and is the minimum number of bytes that must be available for writing in the buffer when it is returned. It defaults to 1. If **do-get-output-buffer** cannot return a buffer with the specified minimum number of bytes, it signals an error.

Values supplied for *bytes* should be small relative to a buffer's size. If you need to write large numbers of bytes at once, specify a large buffer size when making the stream and do not specify any values for the *bytes* keyword parameter.

If an application's thread calls **get-output-buffer** and another thread already holds any buffer for the stream, the function might block. Multi-threaded implementations should eventually return. In multi-threaded implementations this function calls **lock-stream** before calling **do-get-input-buffer**, and it does not release the lock.

If an application's thread calls this function, and that same thread already holds the buffer, this situation is a re-entrancy programmer error. The stream cannot return the buffer and guarantee that its state is up to date with output written between the first call to this function and the current call. Implementations should detect and signal this error.

release-output-buffer *Function*

release-output-buffer *buffered-stream* **=> ()**

Call this function to announce that you have finished using the stream's current buffer. When you have written output to the buffer, you must update the buffer's state by assigning to **buffer-next** before releasing the buffer. When a user of the Buffer Access Protocol sets **buffer-end**, the results are undefined.

If the application does not hold the buffer, this function signals an error.

This function calls **do-release-output-buffer** so that the stream can perform any record-keeping or management tasks it has. You should never call **do-release-output-buffer** directly.

In multi-threaded implementations, **release-output-buffer** releases the lock obtained in **get-output-buffer**.

with-output-buffer *Macro*

with-output-buffer (*buffer-var* **=** *exp*, **#key** *bytes***)** *body* **end;**

Calls **get-output-buffer** on *exp* and *bytes*, binding *buffer-var* to the result, then executes *body* within the scope of the buffer binding.

After the *body* has executed, **with-output-buffer** calls **release-output-buffer** and returns any values returned by *body*.

next-output-buffer *Macro*

next-output-buffer *buffered-stream* **#key** *bytes* **=>** *buffer*

Call this function to get the next buffer for writing output when you are already holding a stream's buffer. If the application does not hold the stream's buffer, this function signals an error.

This function calls **do-next-output-buffer** to get the next output buffer. You should never call **do-next-output-buffer** directly.

You must update the buffer's state by assigning to **buffer-next** before getting the next buffer; otherwise, it might seem that the current buffer is empty (or does not contain the output you just stored in it). When **buffer-next** = **buffer-end** there is no pending output in the current buffer.

When a user of the Buffer Access Protocol sets **buffer-end**, the results are undefined.

The stream extension function may return a new **<buffer>** object or the current one with **buffer-next** updated to indicate the next available element to be written for output. You should not make any assumptions about which option a stream's implementation uses.

The **do-next-output-buffer** function signals an error when the current buffer is not held for output.

The **do-next-output-buffer** function may force output to a stream's destination, but it is not required to do so. It might empty the current buffer into secondary buffers or get a new buffer from a queue, while not forcing any output.

The *bytes* argument is the same as for **get-output-buffer**, but **do-next-output-buffer** signals any error associated with this parameter.

0.7.3 Copying to and from buffers

All generic sequence operations work on buffers. The Streams library provides some additional functions to compensate for lacking functionality in Dylan's sequence operations. There are also a few functions for updating a buffer's state.

buffer-next *Function*

buffer-next *buffer* **=>** *buffer-index*

For buffers held for input, this function returns the location of the next valid byte of input to be read. When the location equals **buffer-end**, there is no more valid input data in the buffer.

For buffers held for output, this function returns the location of the next free byte that should be written in the buffer, and when the location equals **buffer-end**, the buffer is full of pending output.

buffer-next-setter *Function*

buffer-next-setter *new-value buffer* **=>** *new-value*

Sets *buffer*'s next index.

buffer-end *Function*

buffer-end *buffer* **=>** *buffer-index*

For buffers held for input, this function returns the index of the exclusive end of data to be read. For buffers held for output, this function returns the size of the buffer.

See also **buffer-end-setter**, below.

buffer-end-setter *Function*

buffer-end-setter *new-value buffer* **=>** *new-value*

Sets *buffer*'s end index.

Note: Users of the Buffer Access Protocol functions should not set this accessor because the results are undefined; however, stream implementors may need to set this accessor inside Stream Extension Protocol methods, which execute while users are holding a stream buffer.

buffer-subsequence *Open generic function*

buffer-subsequence *buffer result-class start end* **=>** *sequence*

Returns an instance of the class *result-class*, filled with the elements from the buffer from the index *start* to the index *end*.

The *start* and *end* arguments are both instances of **<buffer-index>**.

Implementation Note: This function should be implemented as efficiently as possible. There should be sealed methods for when *result-class* is an instance of **<byte-string>**, **<byte-vector>**, and **<buffer>**.

copy-into-buffer! *Open generic function*

copy-into-buffer! *buffer buffer-start sequence* **#key** *start end* **=> ()**

Copies the portion of *sequence* given by the indices *start* and *end* into *buffer*.

The buffer is filled starting at *buffer-start*.

The *start* index defaults to 0, while *end* defaults to *sequence*`.size`.

If *buffer* is too small to hold the data, an error is signalled.

Implementation Note: This function should be implemented as efficiently as possible. There should be sealed methods for when *sequence* is an instance of **<byte-string>**, **<byte-vector>**, or **<buffer>**.

copy-from-buffer! *Open generic function*

copy-from-buffer! *buffer buffer-start sequence* **#key** *start end* **=> ()**

Copies a portion of *buffer* starting at *buffer-start* into *sequence*. The number of elements copied is determined by *start* and *end*, which are indices into *sequence*. The sequence is filled starting at *start*, which defaults to 0, and ending at *end*, which defaults to *sequence*.**size**. If *sequence* is too small to hold the data, an error is signalled.

Implementation Note: This function should be implemented as efficiently as possible. There should be sealed methods for when *sequence* is an instance of **<byte-string>**, **<byte-vector>**, or **<buffer>**.

0.7.4 Stream extension protocol

These are the generic functions to which implementors of streams add methods when extending the stream protocol to new subclasses of **<buffered-stream>**. Although the **close** function is not described here, you might need to add a method to it when implementing a new stream. See **close**, page 67.

0.7.4.1 Creating new input or input-output streams

do-get-input-buffer *Open generic function*

do-get-input-buffer *buffered-stream* **#key** *wait? bytes* **=> buffer-or-f**

The methods of this generic function implement **get-input-buffer** for new streams. These methods can assume that the stream is locked, so no other thread can access it. Methods must signal an error if the stream is an output-only stream, or if they cannot return a buffer for any reason. If this function's methods cannot return a buffer with the specified minimum number of bytes, they signal an **<incomplete-read-error>**.

do-release-input-buffer *Open generic function*

do-release-input-buffer *buffered-stream* **=> ()**

The methods of this generic function implement **release-input-buffer** for new streams. These methods can assume that the stream is locked, so no other thread can access it.

do-next-input-buffer *Open generic function*

do-next-input-buffer *buffered-stream* **#key** *wait?*, *bytes* **=>** *buffer-or-false*

The methods of this generic function implement **next-input-buffer** for new streams. These methods can assume that the stream is locked, so no other thread can access it. Methods must signal an error if the current buffer is not held for input, or they for any reason cannot return a buffer, or a buffer with the minimum number of bytes specified by *bytes* (an **<incomplete-read-error>**).

do-input-available-at-source? *Open generic function*

do-input-available-at-source? *buffered-stream* **=>** *available?*

The methods of this generic function implement **input-available-at-source?** for new streams. These methods can assume that the stream is locked, so no other thread can access it. Methods must signal an error when the current buffer is not held for input.

0.7.4.2 Creating new output or input-output streams

do-get-output-buffer *Open generic function*

do-get-output-buffer *buffered-stream* **#key** *bytes* **=>** *buffer*

The methods of this generic function implement **get-output-buffer** for new streams. These methods can assume that the stream is locked, so no other thread can access it. Methods must signal an error if the stream is an input only stream, or if they cannot return a buffer for any reason.

The resulting *buffer* is never completely full of pending output. If this function's methods cannot return a buffer with the minimum number of bytes specified by *bytes*, they must signal an error.

do-release-output-buffer *Open generic function*

do-release-output-buffer *buffered-stream* **=> ()**

The methods of this generic function implement **release-output-buffer** for new streams. These methods can assume that the stream is locked, so no other thread can access it.

do-next-output-buffer *Open generic function*

do-next-output-buffer *buffered-stream* **#key** *bytes* **=>** *buffer*

The methods of this generic function implement **next-output-buffer** for new streams. These methods must signal an error when they cannot return a buffer for any reason, or the current buffer is not held for output. If this function's methods cannot return a buffer with the minimum number of bytes specified by *bytes*, they must signal an error.

Methods may force output to *buffered-stream*'s destination, but they are not required to do so. They might empty the current buffer into secondary buffers or get a new buffer from a queue, while not forcing any output.

0.8 Conditions

The class definitions for the error conditions are as follows. There is no recovery protocol defined for any of these errors. Every condition described in this section that takes an init-keyword has a slot accessor for the value supplied, and the name of the accessor function is the name of the condition class (without the angle brackets) to which a hyphen and the name of the init-keyword is appended.

<end-of-stream-error> *Error*

Signalled when one of the read functions reaches the end of an input stream. It is a subclass of **<error>**. It takes one init-keyword, **stream:**.

<incomplete-read-error> *Error*

Signalled when input functions are reading a required number of elements but they read the end of the stream before completing the required read. It is a subclass of **<end-of-stream-error>**. It takes two additional init-keywords, **sequence:** and **count:**. The sequence is whatever input was read before reaching the end of the stream. The count is the number of elements that were requested to be read.

<file-error> *Error*

The base class for all errors related to file I/O. It is a subclass of **<error>**. It takes one init keyword, **locator:**.

<file-exists-error> *Error*

Signalled when an output file stream creation function tries to create a file that already exists. It is a subclass of **<file-error>**.

<file-does-not-exist-error> *Error*

Signalled when an input file stream creation function tries to read a file that does not exist. It is a subclass of **<file-error>**.

<invalid-file-permissions-error> *Error*

Signalled when one of the file stream creation functions tries to access a file in a manner for which the user does not have permission. It is a subclass of **<file-error>**.

0.9 Wrapper streams

Sometimes stream data requires conversion before an application can use it: you might have a stream over a file of EBCDIC characters which you would prefer to handle as their ASCII equivalents, or you might need to encrypt or decrypt file data.

Wrapper streams provide a mechanism for working with streams which require such conversion. Wrapper streams hold on to an underlying stream, delegating to it most of the operations that implement streaming. The wrapper stream carries out appropriate processing in its own implementations of the streaming protocol.

The Dylan Streams Library includes a base class called **<wrapper-stream>** upon which other wrapping streams can be implemented.

```
define class <wrapper-stream> (<stream>)
  slot inner-stream :: <stream>,
       required-init-keyword: inner-stream:;
end class;
```

A subclass of **<wrapper-stream>** can "pass on" functions such as **read-element** and **write-element** by simply delegating these operations to the inner stream:

```
define method read-element (ws :: <io-wrapper-stream>);
  read-element(ws.inner-stream)
end method;

define method write-element (ws :: <io-wrapper-stream>, element);
  write-element(ws.inner-stream,element)
end method;
```

Assuming that **<io-wrapper-stream>** delegates all other operations to its inner stream, the following would suffice to implement a 16-bit Unicode character stream wrapping an 8-bit character stream.

```
define class <unicode-stream> (<io-wrapper-stream>) end class;

define method read-element (s :: <unicode-stream>)
  => ch :: <unicode-character>;
  with-stream-locked (s)
    let first-char = read-element(s.inner-stream);
    let second-char = read-element(s.inner-stream)
  end;
  convert-byte-pair-to-unicode(first-char, second-char)
end method;
```

```
define method write-element(s :: <unicode-stream>,
                            c :: <character>);
  let (first-char, second-char) = convert-unicode-to-byte-pair(c);
  with-stream-locked (s)
    write-element(s.inner-stream, first-char);
    write-element(s.inner-stream, second-char)
  end;
  c
end method;

define method stream-position (s :: <unicode-stream>)
  =>  p :: <integer>;
  truncate/(stream-position(s.inner-stream), 2)
end method;

define method stream-position-setter(p :: <integer>,
                                     s :: <unicode-stream>);
  stream-position(s.inner-stream) := p * 2
end method;
```

0.9.1 Wrapper streams and delegation

One problem with wrapper streams is the need for a wrapper stream to intercept methods invoked by its inner stream. For example, consider two hypothetical streams, **<interactive-stream>** and **<dialog-stream>**, the latter a subclass of **<wrapper-stream>**. Both of these classes have a method called **prompt**. The **<interactive-stream>** class specializes **read** thus:

```
define method read (s :: <interactive-stream>,
                    n :: <integer>,
                    #key on-end-of-stream);
  prompt(s);
  next-method()
end method;
```

If a **<dialog-stream>** is used to wrap an **<interactive-stream>** then an invocation of **read** on the **<dialog-stream>** will call **prompt** on the inner **<interactive-stream>**, not on the **<dialog-stream>**, as desired. The problem is that the **<dialog-stream>** wants to delegate some tasks to its inner stream, but handle some other tasks itself.

Some languages, notably Self, support such delegation in the language. Dylan, in keeping with other generic-function-based languages, does not, but experience with streams packages in a number of other generic-function-based lan-

guages has shown that it is *very* useful to provide delegation for wrapper streams. So we provide an explicit implementation of delegation in the streams system.

Delegation is implemented by the use of the **outer-stream** slot, which is defined in the base class **<stream>**:

```
define abstract class <stream> (<object>)
  slot outer-stream :: <stream>, init-keyword: outer-stream:;
end class;
```

outer-stream is used instead of the stream itself whenever a stream invokes one of its other protocol methods.

Note: A stream must not use **inner-stream** if performing recursion, since this would cause an infinite regress.

A correct implementation of the **read** method in the example above would be as follows:

```
define method read (stream :: <interactive-stream>,
                    n :: <integer>
                    #key on-end-of-stream)
  prompt(s.outer-stream);
  next-method()
end method;
```

The **initialize** method on **<stream>** is defined to set the **outer-stream** slot to be the stream itself. The **initialize** method on **<wrapper-stream>** is specialized to set the **outer-stream** slot to be the "parent" stream:

```
define method initialize (stream :: <wrapper-stream>,
                  #key on, #rest all-keys);
  an-inner-stream.outer-stream := stream;
  next-method()
end method;
```

Implementation Note: One disadvantage of this scheme is that it may prevent useful optimizations, such as inlining. If this is an important performance issue in circumstances where wrapper streams are not necessary, then it is relatively simple to provide an implementation of the **streams** module that omits the delegation mechanism.

0.10 Wrapper stream protocol

<wrapper-stream> *Open instantiable class*

The class that implements the basic wrapper-stream functionality.

It takes a required init-keyword **inner-stream:**, which is used to specify the wrapped stream.

The **<wrapper-stream>** class implements default methods for all of the stream protocol functions described in this document. Each default method on **<wrapper-stream>** simply "trampolines" to its inner stream.

inner-stream *Open generic function*

inner-stream *wrapper-stream* **=>** *wrapped-stream*

Returns the stream wrapped by *wrapper-stream*.

inner-stream-setter *Open generic function*

inner-stream-setter *stream wrapper-stream* **=>** *stream*

Wraps *stream* with *wrapper-stream*. It does so by setting the **inner-stream** slot of *wrapper-stream* to *stream*, and the **outer-stream** slot of *stream* to *wrapper-stream*.

Note: Applications should not set **inner-stream** and **outer-stream** slots directly. The **inner-stream-setter** function is for use only when implementing stream classes.

outer-stream *Open generic function*

outer-stream *stream* **=>** *wrapping-stream*

Returns the stream that is wrapping *stream*.

outer-stream-setter *Open generic function*

outer-stream-setter *wrapper-stream stream* **=>** *wrapper-stream*

Sets the **`outer-stream`** slot of *stream* to *wrapper-stream*.

Note: Applications should not set **`inner-stream`** and **`outer-stream`** slots directly. The **`inner-stream-setter`** function is for use only when implementing stream classes.

The Standard-IO Library

The `Standard-IO` module of the Standard-IO library exports the following:

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

These **<stream>**s represent the standard input stream, the standard output stream, and the standard error stream, respectively. Usually standard input gets its input from the terminal, and standard output and error write their output to the terminal. However, most operating systems allow these to be "redirected" on the command line.

The Print Library

Designed by the Gwydion Project

Version 1.0 : 22 May 97

This document describes the `Print` library designed by the Gwydion Project at Carnegie Mellon University. This library provides an interface that outputs an object in Dylan literal syntax if the object can be represented as a Dylan literal, and otherwise, outputs the object in an implementation-dependent manner. There are essentially two functions, **print** and **print-object**. The **print** function accepts keyword arguments that form a print request, controlling features such as circular printing, how deep within a data structure to print, how many elements in long sequences to print before using an ellipsis notation, whether pretty printing is desired, and so on. Users extend **print**'s ability to print various objects by adding methods to the **print-object** function. The **print** function handles most of the overhead to satisfy special print requests, outputting any special notations required, and it only calls **print-object** when it is necessary to print objects. Users should always call the **print** function to output objects, especially recursively from within **print-object** methods to output an object's components. Users should never call **print-object** directly.

The `Print` library exports two modules, `Print` and `Pprint`. The Gwydion implementation of this library requires the Gwydion `Streams` library.

1. Print Functions

The `Print` module offers two functions for users to call to print objects, **print** and **print-to-string**.

default-level [Variable]
default-length [Variable]
default-circle? [Variable]
default-pretty? [Variable]

These variables provide default values for calls to the **print** function. Their values are implementation-dependent.

print [Function]

Arguments

object :: <object>
stream :: <stream>
#key level :: false-or(<fixed-integer>) = *default-level*,
length :: false-or(<fixed-integer>) = *default-level*,
circle? :: <boolean> = *default-circle?*,
pretty? :: <boolean> = *default-pretty?*

Values

none

Description

Prints *object* to *stream* according to the print request formed by the keyed arguments. A *first* call to **print** creates a printing stream to represent the print request, and recursive calls to **print** on this printing stream process the keyed arguments differently (see below). There are inspection functions for querying the print request (see Section *Print Request Inspection Functions*). When **print** actually prints an object, it calls **print-object**. Though the inspection functions for querying the print request allow you to inspect any parameter of the print request, **print-object** methods should only need to call **print-length**. All other aspects of the print request are handled by **print**. There is one exception which is described in Section *Pretty Printing*.

Level controls how deep into a nested data structure to print. The value `#f` indicates that there is no limit. The default, ***default-level***, has no effect on recursive calls to **print**. Recursive calls to **print** may change the value of **print-level** explicitly, but **print** always uses a value to ensure the print request formed by the first call to **print** is never exceeded. For example, if a first call to **print** set the level to 5, and while at a depth of 3, a recursive call

specified a level of 4, the recursive call would only descend 2 more levels, not 4.

Length controls how many elements of a sequence to print before printing ellipsis notation (`...`). The value `#f` indicates that there is no limit. The **print-length** control can be interpreted loosely by some **print-object** methods to control how many *elements* of any kind of object to print; for example, the default **<object>** method might regard **print-length** to determine how many slot-name/value pairs to print. The default, ***default-length***, has no effect on recursive calls to **print**. Recursive calls to **print** may change the value of **print-length** explicitly, but they may only decrease the value, never increase it.

Circle? indicates whether printing should check all subcomponent references to make sure the printing process does not infinitely recurse through a data structure. Circular printing also tags objects that occur more than once when they are first printed, and later occurrences are printed as a reference to the previously emitted tag. The default, ***default-circle?***, has no effect on recursive calls to **print**. If **print-circle?** is already `#t`, then it remains `#t` throughout all recursive calls. If **print-circle?** is `#f`, then recursive calls to **print** can change the value to `#t`; however, when printing exits the dynamic scope of the call that changed the value to `#t`, the value reverts back to `#f`. If the original call to **print** specifies *circle?* as `#f`, and dynamically distinct recursive calls turn circular printing on and off, all output generated while circular printing was on shares the same tagging space; that is, if `#1#` is printed twice, once from each of two distinct recursive calls to print, then each `#1#` is guaranteed to signify the same `\==` object.

Pretty? indicates whether printing should attempt to insert line breaks and indentation to *format* objects according to how programmers tend to find it easier to *read* data. The default, ***default-pretty?***, has no effect on recursive calls to **print**. If **print-pretty?** is already `#t`, then it remains `#t` throughout all recursive calls. If **print-pretty?** is `#f`, then recursive calls to **print** can change the value to `#t`; however, when printing exits the dynamic scope of the call that changed the value to `#t`, the value reverts back to `#f`.

print-to-string [Function]

Arguments

object :: <object>
#key level :: false-or(<fixed-integer>) = *default-level*,
length :: false-or(<fixed-integer>) = *default-level*,
circle? :: <boolean> = *default-circle?*,
pretty? :: <boolean> = *default-pretty?*

Values

result :: <byte-string>

Description

Calls **print** to produce output according to the print request formed by the keyed arguments and returns the output as a string.

print-object [Generic Function]

Arguments

object :: <object>
stream :: <stream>

Values

none

Description

Users extend **print**'s ability to print various objects by adding methods to the **print-object** function. When **print** actually prints an object, it calls **print-object**. Users should never call **print-object** directly.

2. Print Request Inspection Functions

The `Print` module exports the following functions for querying the print request and current print state:

print-length [Function]

Arguments
stream :: <stream>
Values
length :: false-or(<fixed-integer>)
Description
Returns the current value for the print request. See the **print** function for details.

print-level [Function]

Arguments
stream :: <stream>
Values
level :: false-or(<fixed-integer>)
Description
Returns the current value for the print request. See the **print** function for details. Users should have little use for this function because **print** takes care to call **print-object** only when the print level has not been exhausted.

print-depth [Function]

Arguments
stream :: <stream>
Values
depth :: false-or(<fixed-integer>)
Description
Returns the current depth to which printing has descended into the object on which **print** was originally called. Users should have little use for this function because **print** takes care to call **print-object** only when the print level has not been exhausted.

print-circle? [Function]

Arguments
stream :: <stream>
Values
circle? :: <boolean>
Description
Returns whether circular printing is on. Users should have little use for this function because **print** takes care to detect circularities, tag multiply referenced objects, and emit tags rather than descending into objects to repeatedly print them.

print-pretty? [Function]

Arguments
stream :: <stream>
Values
pretty? :: <boolean>
Description
Returns whether pretty printing is on. Users should have little use for this function (see Section *Pretty Printing*).

3. Pretty Printing

The `Print` library implements most of the pretty printing technology described by Richard C. Waters in *Common Lisp The Language*, second edition. The interface is slightly different because Mindy does not have macros. This section only summarizes the pretty printing functionality to provide a quick reference for users of the `Print` library, and readers should refer to the Common Lisp manual for more details.

When writing **print-object** methods, users can ignore whether pretty printing is in effect. If you write your **print-object** method using pretty printing functions, then when pretty printing is in effect, the output will be pretty printed. When pretty

printing is not in effect, your method will produce output as though you had not written it to use pretty printing. All **print-object** methods that are written to do pretty printing must call the pretty printing functions within the dynamic scope of a call to **pprint-logical-block**; otherwise, the pretty printing functions are no-ops.

The `Pprint` module exports the following variables:

default-line-length [Variable]

This is the line length used by the pretty printer to determine how much output will fit on a single line. The value must be an **<integer>**, and it defaults to 80.

print-miser-width [Variable]

This variable controls *miser mode*. Whenever a logical block (see **pprint-logical-block**) begins in a column of output that is greater than `*default-line-length* - *print-miser-width*`, then pretty printing is in miser mode. The value must be an integer or `#f` (the default). `#f` indicates that the pretty printer should never enter miser mode.

pprint-logical-block [Function]

Arguments

stream :: <stream>
#key prefix :: false-or(<byte-string>)
per-line-prefix :: false-or(<byte-string>)
body :: <function>
suffix :: false-or(<byte-string>)
column :: limited(<fixed-integer>, min: 0)

Values

none

Description

This function groups printing into a logical block. The logical block provides boundaries for new levels of indentation, affects `#"linear"` newlines, and so on. *Prefix* is a string to print at the beginning of the logical block. The blocks indentation is automatically set to be one character position greater than the column in which *prefix* ends. Alternatively, *per-line-prefix* is a string to print on every line of the logical block. This function signals an error if it is called with both *prefix* and *per-line-prefix* supplied as non-`#f`. *Suffix* is a string to print at the end of the logical block. *Column* advises the pretty printer as to the current column of the output stream (defaults to zero). The *column* argument may be ignored entirely by some methods, and it may be ignored in some cases by methods that can better determine the stream's current output column.

Body must be a function that can take one argument, and this argument is a stream. The *body* function should use the stream argument passed to it; the *body* function should not close over the stream argument to **pprint-logical-block**. **Pprint-logical-block** wraps *stream* with a pretty printing stream when *stream* is any other kind of stream. If *stream* is already a pretty printing stream, then the *body* function is called on *stream*.

All **print-object** methods that are written to do pretty printing must call the other pretty printing functions within the dynamic scope of a call to **pprint-logical-block**; otherwise, the pretty printing functions are no-ops.

pprint-newline [Function]

Arguments

kind :: one-of(#"fill", #"linear", #"miser", #"mandatory")
stream :: <stream>

Values

none

Description

This function announces a conditional newline to the pretty printer. The pretty printer emits a newline depending on the *kind* and the state of the pretty printer's current line buffer. The *kind* argument has roughly the following meanings:

`#"fill"`
: Emit a newline if the current *section* of output does not fit on one line.

`#"linear"`
: Emit a newline if any `#"linear"` newline in the current *section* needs to be emitted. That is, if a current *section* of output cannot fit on one line, and any one of the `#"linear"` newlines in the section needs to be emitted, then emit them all.

`#"miser"`
: Emit a newline as if it were a `#"linear"` newline, but only when *miser mode* is in effect. Miser style is in effect when a logical block starts past a particular column of output.

`#"mandatory"`
: Emit a newline always. Establish that any containing *sections* cannot be printed on a single line so that `#"linear"` and `#"miser"` newlines will be emitted as appropriate.

pprint-indent [Function]

Arguments
: relative-to :: one-of(#"block", #"current")
: n :: <fixed-integer>
: stream :: <stream>

Values
: *none*

Description
: This function specifies the indentation to use within the current logical block. When *relative-to* is `#"block"`, then **pprint-indent** sets the indentation to the column of the first character of the logical block plus *n*. When *relative-to* is `#"current"`, then **pprint-indent** sets the indentation to the current column plus *n*.

pprint-tab [Function]

Arguments
: kind :: one-of(#"line", #"line-relative", #"section", #"section-relative")
: colnum :: <fixed-integer>
: colinc :: <fixed-integer>
: stream :: <stream>

Values
: *none*

Description
: This function announces a tab to the pretty printer. *Colnum* and *colinc* have meaning based on the value of *kind*:

`#"line"`
: Tab to output column *colnum*. If the output is already at or beyond *colnum*, then add *colinc* to *colnum* until printing can continue at a column beyond the end of the output already on the line.

`#"line-relative"`
: Output *colnum* spaces. Then output enough spaces to tab to a column that is a multiple of *colinc* from the beginning of the line.

`#"section"`
: This is similar to `#"line"`, but column counting is relative to the beginning of the current *section* rather than the beginning of the line.

`#"section-relative"`
: This is similar to `#"line-relative"`, but column counting is relative to the beginning of the current *section* rather than the beginning of the line.

The Format Library

Designed by the Gwydion Project

Version 1.0 : 022 May 97

This document describes the `Format` library designed by the Gwydion Project at Carnegie Mellon University. This library extends the functionality of the format strings described in Dylan's condition system and provides two new functions for processing the extended format strings. The `Format` library is a small library, but it requires the `Print` library and some of the `Streams` library from the Gwydion Project. The `Format` library exports one module, `Format`, which exports all the idnetifiers described in this document.

1. Functions

The `Format` library offers two functions, **format** and **format-to-string**.

format [Generic Function]

Arguments
- stream :: <stream>
- control-string :: <string>
- #rest arguments

Values
- *none*

Description
- Performs output to *stream* according to the directives in *control-string*. Each directive consumes one argument from *arguments*. The *control-string* contents that are not part of any directive are output directly to *stream*, as if by the `Stream` library's **write** function.

format [Method]

Arguments
- stream :: <stream>
- control-string :: <byte-string>
- #rest arguments

Values
- *none*

Description
- There is one method for **format**, and it is specialized to **<byte-string>**s.

format-to-string [Generic Function]

Arguments
- control-string :: <string>
- #rest arguments

Values
- result :: <string>

Description
- Calls **format** to produce output according to *control-string* and returns the output as a string.

format-to-string [Method]

Arguments

control-string :: <byte-string>
#rest arguments

Values

result :: <byte-string>

Description

There is one method for **format-to-string**. The *control-string* argument must be a **<byte-string>**. *Result* is a **<byte-string>**.

print-message [Generic Function]

Arguments

object :: <object>
stream :: <stream>

Values

none

Description

Prints *object* to *stream*. Methods for this function should print objects *as a message*, as opposed to printing objects in any form intending to represent Dylan data, literal syntax, and so on. For example, printing a condition object with this function presents the condition as an error message, but printing the condition object with the **print** function from the Gwydion `Print` library prints the condition in some form such as `{Simple-error}`.

See the individual methods for the details of how this function prints various objects.

This function exists to define the behavior of the `%S` format directive and to allow users the ability to extend the `%S` directive. Users should have little need to call this function directly.

print-message [Sealed Method]

Arguments

condition :: <condition>
stream :: <stream>

Values

none

Description

Prints *condition* as an error message, as described for the Dylan `%S` format directive. In Gwydion Dylan, this method actually calls the **report-condition** function from the `Dylan` library's `Extensions` module. Users should not specialize the **print-message** protocol for subclasses of **<condition>**, but they should instead extend the **print-message** protocol to new condition objects by specializing methods on **report-condition**.

print-message [Sealed Method]

Arguments

symbol :: <symbol>
stream :: <stream>

Values

none

Description

Prints *symbol* to *stream* by converting it to a string with the **as** function and then writing the string with the **write** function from the `Streams` library.

print-message [Sealed Method]

Arguments

object :: type-union(<string>, <character>)

stream :: <stream>

Values

none

Description

Prints *object* to *stream* by calling the **write** function from the `Streams` library.

2. Control Strings

The `Format` library's format strings, or control strings, offer the same directives as Dylan's format strings offer, but this library provides a few more directives and permits a single argument to all format directives. The argument is an integer that must appear contiguously between the dispatch character (%) and the format directive, and it indicates a printing field in which to justify the output of the directive. A positive integer indicates that the output should be flushright within the field, and a negative integer indicates the output should be flushleft within the field. If the output length is greater than the field's width, then output occurs as if there were no field specification. The following are examples of valid format directives:

```
%S
%s
%15D
%-10=
```

The following describes the directives:

`%S`

Prints the next format argument *as a message* by calling the function **print-message** on the format argument and the stream. This directive is the same as Dylan's `%S` format-string directive except for two features: This library's `%S` directive outputs character objects, and users can extend the `%S` functionality by adding methods to **print-message**.

`%=`

Prints the next format argument by calling the **print** function from the `Print` library on the format argument and the stream. Users can extend the `%=` functionality by adding methods to the **print-object** function from the `Print` library.

`%C`

Print the next format argument, which must be a character, according to Dylan's `%S` format-string directive. This library's `%C` directive is the same as this library's `%S` directive.

`%D`

Prints a decimal representation of the next format argument, which must be an integer.

`%B`

Prints a binary representation of the next format argument, which must be an integer.

`%O`

Prints an octal representation of the next format argument, which must be an integer.

`%X`

Prints a hexadecimal representation of the next format argument, which must be an integer.

`%M`

Invokes the next format argument, which must be a function, on the stream passed to **format**.

`%%`

Outputs a single % character.

The TK Library

1. Introduction and Warning

Dylan/TK is an experimental TK binding for the Mindy Dylan interpreter. Although it is definitely usable, it is not intended to be an "industrial strength" product. The implementation strategy virtually guarantees that there will be some rough edges.

This binding operates by starting a `wish` interpreter as a slave process and passing both Tcl routines and data between the Mindy interpreter and the slave process. However, Dylan/TK provides an abstraction layer that should vastly reduce or even eliminate the need for users to be familiar with the Tcl extension language. For example, the following Dylan code:

```
define constant text-frame
  = make(<frame>, height: 500, fill: "both", side: "bottom", expand: #t);
define constant text-window
  = make(<text>, in: text-frame, relief: "sunken", font: normal-font,
         fill: "both", side: "right", expand: #t);
define constant text-scroll
  = scroll(text-window, in: text-frame, fill: "y");
```

could be used to produce the effect as the following Tcl/Tk code:

```
frame .text-frame -height 500
pack .text-frame -fill both -side bottom -expand 1
text .text-frame.text -relief sunken -font normal-font
pack .text-frame.text -in .text-frame -fill both -side right -expand 1
scrollbar .text-frame.scroll -command {.text-frame.text yview }
.text-frame.text configure -yscrollcommand {.text-frame.scroll set }
pack .text-frame.scroll -in .text-frame -fill y
```

while being considerably clearer to the average Dylan programmer.

Note: In order for this implementation to work you must have a copy of `wish` available on your system. This implementation was developed upon version 4.0 of Tcl/Tk but should work any version Tk implementation after 4.0. Since version 4 introduces incompatible changes, you should not (at present) expect all features to work with earlier versions.

2. General Principles

In order to use Dylan/Tk, you should import module `Tk` from library Tk. This defines the **<window>** class, a variety of subclasses representing most of the Tk widgets, and a variety of support routines. All of these will be explained in some detail below. However, the user will sometimes find that he can gain more information by going to the book Tcl and the Tk Toolkit by John Osterhout, which explains the widgets' behavior in more depth than is practical for this document.

Any Dylan program which uses the `Tk` library will automatically create a top level window (which is accessible via the ***root-window*** variable) and a separate thread to process events upon that window. You build an interface by making widgets and "packing" them into windows -- either the root window, newly created top-level windows, or into subwindows of these windows.

An simple application might be the following:

```
define method main (program-name :: <string>, #rest args);
make(<message>, text: "Hello, world!", aspect: 500, side: "top");
make(<button>, text: "Okay", relief: "raised",
     command: curry(destroy-window, *root-window*), expand: #t);
map-window(*root-window*);
end method main;
```

This creates a text message and a pushbutton, packs them into the root window, and then exits, and makes the root window visible. It then exits the main thread, leaving just the event processing thread active. When the button is pressed, it will destroy the root window (and all its subwindows). Since there are no windows left, the event thread exits, and the program terminates. We could just as easily specified "`command: exit`", which would force the Dylan program to exit, killing off all threads, and thus destroying the windows. Go ahead and try it, if you like.

Widgets, like any other Dylan object, are typically created via **make**. There are three different varieties of keywords which can be specified:

Widget Specific Options:
: These include the `aspect:` and `command:` keywords in the above example. Their precise effects depend upon the definition of the widget.

Standard Options:
: These include the `relief:` keyword in the above example. They are accepted by most (if not all) widgets, and have pretty much the same effect for all of them. (Some keywords, like `text:`, are not accepted by all widgets, but are common and predictable enough that users may be left to infer their effects.)

Packing Options:
: These include the `side:` and `expand:` keywords in the above example. They are used to control the positioning and resizing of the new widgets within their "parent" windows. They are, for the most part, passed on directly to the **pack** function. In very rare cases, you will wish to create a window without (yet) packing it into another window. In this case, you can specify `pack: #f` to defer calling **pack** until you are ready to do so.

3. Parameter and Return Values

The values which are passed into **make** or into many other Tcl/Tk functions are passed into to Tcl/Tk as strings, and thus it is sometimes most convenient to write them as strings in Dylan. Most Dylan/Tk functions therefore accept strings as parameter values. However, you will usually find it easier to work with a richer set of types within Dylan's type system. Parameters values can commonly include instances of the following native Dylan classes:

- **<boolean>**
- **<integer>**
- **<function>**
- **<string>**
- **<sequence>**
- **<class>**

and will occasionally include others on a case-by-case basis. In addition, the following new classes are also frequently accepted as parameter values:

- **<window>**
- **<tk-variable>**
- **<text-index>**
- **<text-mark>**

Those Dylan/Tk functions which return values typically cast the results back into meaningful Dylan types before returning them. However, there are some occasions where this proves impractical, and result values remain as **<string>**s which must be explicitly cast into appropriate types. For example, the **configuration** function returns the values of all the configuration options for a widget. Since it can't guess what the types of these values ought to be, they simply remain as **<string>**s.

In some cases you must specify callback functions for Dylan/Tk widgets. These are typically specified via the `command:` keyword to make. On the few occasions where these functions take parameters, they will be passed in as **<string>**, and may include "\" quotation characters. You may need to call **tk-unquote** to strip out backslashes or to call **tk-as** to convert this to another type. For example: `tk-as(<boolean>, "1")` returns `#t`.

4. The "Standard" Options

The following options seem to apply universally to all widgets. They are mostly concerned with presentation details of the widget's window, as opposed to the widget's function.

`background:`
: This specifies the background color to be used within the widget. Typically it is a color name, like `Black` or `LightBrown`, or as hexadecimal integers, like `#x3a7` or `#x3000a0007000`.

`borderwidth:`
: This specifies the size of the border to be placed around the window. Typically this will be an integer which specifies the number of pixels, but you may also use string values like `"1c"`, `"1m"`, `"1i"`, `"1p"`, which specify centimeters, millimeters, inches, and points respectively. This keyword is typically combined with the `relief:` keyword.

`cursor:`
: This specifies a bitmap which will be displayed when the user's cursor is in this widget. The most useful forms will be `name` to specify a name from the standard cursor font, or `@Sourcefile fgColor` to load the cursor from a file.

`foreground:`
: This specifies the background color to be used within the widget. Typically it is a color name, like `Black` or `LightBrown`, or as hexadecimal integers, like `#x3a7` or `#x3000a0007000`.

`highlightbackground:`
: This specifies the color of the highlight area (a thin border surronding the widget) when it is disabled. This is normally a color name, like `Black` or `LightBrown`, or as hexadecimal integers, like `#x3a7` or `#x3000a0007000`.

`highlightforeground:`
: This specifies the color of the highlight area (a thin border surronding the widget) when it is enabled. This is normally a color name, like `Black` or `LightBrown`, or as a hexadecimal integer, like `#x3a7` or `#x3000a0007000`.

`highlightthickness:`
: This specifies the thickness of the highlight area. Typically this will be an integer which specifies the number of pixels, but you may also use string values like "`1c`", "`1m`", "`1i`", "`1p`", which specify centimeters, millimeters, inches, and points respectively. A value of zero will remove the highlight ring.

`relief:`
: This specifies what sort of 3D border should be displayed around the widget. This must be one of: "flat" (default), "groove", "raised", "ridge", or "sunken".

Other options are by no means universal, but tend to mean the same thing whenever they occur:

`bitmap:`
: This specifies a bitmap which will be displayed in the given widget. The most useful forms will be `name` to specify a name from the standard cursor font, or `@Sourcefile fgColor` to load the cursor from a file.

`command:`
: This specifies an action to be taken when a particular widget is invoked or when values associated with the widget change. Typically this will be a Dylan function, but it may also be a bit of Tcl code which can be executed directly by the interpreter. The number of parameters which might be passed to the function vary depending on the widget, but they will all be passed in as **<string>**s.

`font:`
: This is a string which specifies the font to be used for any text which is displayed within the widget.

`justify:`
: This is a string which specifies how the text within the widget is to be justified. Possible valuse are

"left", "center", and "right".

`state:`
: This specifies whether the given widget is in an active state. Typically, any widget which accepts a `command:` keyword will accept this keyword as well. Acceptable values are "normal", "disabled", and sometimes "active" (for buttons only).

`text:`
: This is a string which will be displayed in the widget in an implementation dependent way. A few widgets also take a `label:` keyword which acts much the same way.

`textvariable:`
: This specifies a "variable" containing a value which will be displayed in the widget. If the variables value changes, the widget is updated. If the widget allows editing, the variable will be updated after each change. Dylan/Tk programs will typically supply a **<tk-variable>** object for this keyword. (These will be explained in a later section.)

`underline:`
: This specifies a charater in the text string of a widget to underline.

`variable:`
: This functions much like the `textvariable:` keyword, but tends to be displayed somewhat differently.

5. Windows

The abstract type **<window>** represents any one of a wide variety of widgets, although it doesn't include pseudo-widgets like menu entries, text tags, or canvas items (which will be discussed later). They are almost always created via **make**, which accepts all of the standard options described above, as well as the packer options `name:`, `in:`, `after:`, `before:`, `side:`, `expand:`, `padx:`, `pady:`, and `pack:`. (Packing will be described briefly in the next section.) All windows except **<toplevel>**s must specify their parents by some means -- either directly via `in:` or indirectly via `after:` or `before:`.

You also have the right to specify a debug name for the window via the `name:` option. If you refuse this right, one will be created for you automatically.

One window is automatically created as a result of importing the `tk` library. This is the top level window ***root-window***.

There are a small set of methods which operate upon all windows:

configure (*widget* :: **<window>**, *#all-keys*) [Function]
: Configure accepts pretty much the same options as initialize, and uses them to change the state of the object.

configuration (*widget* :: **<window>**) => (*result* :: **<sequence>**) [Function]
: Returns a complete list of options for the given widget. Each option consists of a sequence of the switch name, rdb name, rdb class, default, and value. The "value" will be a string, which may be converted to a more pleasing form via either **tk-as** or **tk-unquote**.

map-window (*window* :: **<window>**) => () [Function]
: Make sure that a window is displayed on the screen. ***root-window*** is unmapped initially, so nothing will appear until you call `map-window(*root-window*)`.

unmap-window (*window* :: **<window>**) => () [Function]
: Remove a window from the display. This is handy to keep from showing intermediate states of newly defined windows.

destroy-window (*window* :: **<window>**) => () [Function]
: Completely obliterate a window, removing it from the screen and all knowledge of mankind.

pack (*window* :: **<window>**, *#all-keys*) => (*window* :: **<window>**) [Function]
This is explained in the next section. You should never need to call pack unless you either specified "`pack: #f`" when making the window or you used the **unpack** function to remove it from the display.

unpack (*window* :: **<window>**) => (*window* :: **<window>**) [Function]
This hides a window which has previously been packed into another window. The window still exists, and may be updated or repacked into its parent. You may not pack the window into a different parent -- this will result in errors.

\= (*window* :: **<window>**, *window* :: **<window>**) => *equal?* :: **<boolean>** [Method]
\= (*window* :: **<window>**, *path-name* :: **<string>**) => *equal?* :: **<boolean>** [Method]
\= (*path-name* :: **<string>**, *window* :: **<window>**) => *equal?* :: **<boolean>** [Method]
The three **\=** methods are useful for comparing two windows to see if they are, in fact, the same widget. Most function which should return windows will return them as **<window>** objects, but if you are dealing with lower level Tk, it may be useful to compare a Tk window path-name to a Dylan **<window>** object.

6. Packing

Unless they are some sort of top level window (i.e. ***root-window*** or instances of **<toplevel>** or **<menu>**), windows must be "packed" into some containing window before they will appear on the screen. Normally, all Dylan/Tk windows are automatically packed when they are created, using the procedure and options described below. The exception to this is if the `pack:` option is given a value of `#f` or is the window was **unpack**ed; in these cases, the **pack** function will need to be called explicity to display the window. The Dylan/Tk **pack** function closely follows the Tcl/Tk function described in "Tcl and the Tk Toolkit", and you should look there for more complete information. However, we will provide a brief overview here.

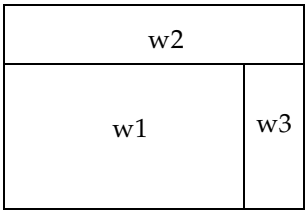
Each "slave" (i.e. non-top-level) window must be packed into some other window. You specify this window either directly, via "`in: parent`" or indirectly via "`before: sibling`" or "`after: sibling`". (The latter options specify that the window should have the same parent as the specified "sibling", and that they should be placed just before or after the sibling in the parent's "packing order".) If no parent is specified explicitly, Dylan/Tk implicitly assumes "`in: *root-window*`".

The `side:` option specifies where the widget should be placed relative to the ones which follow it in the "packing order". If you specify "`side: "left"`", then all widgets which are packed into the parent after this one (either because they were created later or because of a `before:` or `after:` option) will be placed somewhere to the right of this widget. Note that it will not necessarily be placed on the parent's absolute left side, because all windows which precede this one in the packing order have "first dibs" on prime real estate. Possible values for this option are "left", "right", "top", and "bottom" -- the default is "left".

If you were to execute:

```
let w1 = make(<listbox>, in: *root-window*, side: "left");
let w2 = make(<text>, before: w1, side: "top");
let w3 = make(<scrollbar>, after: w1);
```

you would end up with something like:



Window W2 comes first in the packing order (because of the `before:` option) and preempts the top of the root window. W1 comes second, and grabs the left side of the remaining space, leaving a small chunk on the right for W3 (and any

windows which might be packed after W3).

The other options available are `anchor:`, `expand:`, `fill:`, `padx:`, and `pady:`, which are explained in the book. `Expand:` typically takes a boolean value, while `fill:` takes one of "x", "y", "both", or "none".

The **unpack** function will remove a window from the control of the "packer" and, as a side effect, remove it from the display. The window will still exist and can be updated or repacked via the **pack** function. You will, however, get errors if you try to specify a different parent when you repack it.

7. Event bindings

Each Dylan/Tk widget has a default set of "bindings" which determine how they react to X events like button or key presses. These may be changed or extended via the "bind" function. Windows, "tag" strings, and subclasses of **<window>** can have bindings, and more than one binding may apply to an event. All of the applicable bindings will get called, unless explicitly avoided, and the call order can be changed.

bind (*window-tag-or-class*, *event* :: **<string>**, *command*) => (*window*)

This function specifies an action to be performed when an X event matching "event" occurs. Typically this is a nil-adic Dylan function, but it may also be an arbitrary line of Tcl code. A typical binding might be the following:

```
bind(l1, "<Double-Button-1>",  method ()
       do(print-config, current-selection(l1))
       end method);
```

This would cause Dylan/Tk to call **print-config** on each of the current selections in the **<listbox>** "l1" whenever the leftmost mouse button is double-clicked.

The first argument to this function may be a **<window>** object, in which case it acts on only that window, a "tag" string, which acts on all windows with that tag, or a subclass of **<window>**, which acts on all objects of that type.

There are five other utility functions which allow you to determine and manipulate the bindings for windows:

get-binding (*window-tag-or-class*, *event*) => (*result* :: **<string>**) [Function]

Returns the window, tag, or class's binding for a single event. Unless you have specified an explicit binding, this will be an empty string.

get-bindings (*window-tag-or-class*) => (*result* :: **<sequence>**) [Function]

Returns a complete list of explicit bindings for a window, tag string, or class. Each element of the sequence will be a **<pair>** of event-name and binding.

binding-call-order (*window* :: **<window>**) => (*result* :: **<sequence>**) [Function]

Returns a list of bindings that apply to a window, in the order that they are called. Each element is a Tk text string which could represent a window, a tag, or a class.

binding-call-order-setter (*bindings* :: **<sequence>**, *window* :: **<window>**) => (*bindings* :: **<sequence>**) [Function]

Sets the calling order for bindings. The elements of the sequence may be **<window>** objects, tag strings, or subclasses of **<window>**

tk-break () [Function]

If you don't want to call all bindings which apply to an event, use the **tk-break** command to stop the execution of bindings. This command must be executed on the top level of the binding, and not in any functions that the command calls.

8. Active Variables

Some widgets have the ability to take an obsessive interest in the value of some "variable". They may update their contents or appearance when the variable's value changes, or they might update the variable in response to some user action. Because this level of monitoring is not supported by ordinary Dylan variables, Dylan/Tk users must instead use instances of **<active-variable>**.

Make on **<active-variable>** requires a `value:` keyword and accepts an optional `class:` keyword. If specified, the class defines the logical type of the variable's value. The **value** slot will always contain a value of that type, although **value-setter** will accept arbitrary types (especially **<string>**) and attempt to convert them to the given type.

You may also specify a Dylan function to be executed when an active variable's value changes (i.e. becomes `\~=` to the old value). This method is specified via the `command:` keyword to **make**. It will be invoked with both the new and the old value at some point after the value is changed. Note that the value is updated immediately (i.e. asynchronously), while the calls to the given command are executed sequentially along with event callbacks.

Active variables will typically be passed as values of `variable:` or `text-variable:` keywords.

9. The widget types

Dylan/Tk provides types corresponding to all of the standard Tcl/Tk widgets:

<button>:

Options:

```
#"activebackground"
#"activeforeground"
#"bitmap"
#"command"
#"disabledforeground"
#"font"
#"height"
#"justify"
#"state"
#"text"
#"textvariable"
#"underline"
#"width"
#"wraplength"
```

Functions:

activate (*button*) => *button*
deactivate (*button*) => *button*
flash (*button*) => *button*
invoke (*button*) => *button*

<checkbutton>:

Options:

```
#"activebackground"
#"activeforeground"
#"bitmap"
#"command"
#"disabledforeground"
#"font"
#"height"
#"offvalue"
```

#"onvalue"
#"selector"
#"state"
#"text"
#"textvariable"
#"variable"
#"width".

Functions:

select-value (*button*) => *button*
deselect-value (*button*) => *button*
toggle-value (*button*) => *button*

<menubutton>

Options:

#"activebackground"
#"activeforeground"
#"bitmap"
#"disabledforeground"
#"font"
#"height"
#"indicatoron"
#"justify"
#"menu"
#"state"
#"text"
#"textvariable"
#"underline"
#"width"
#"wraplength"

Functions:

activate (*button*) => *button*
deactivate (*button*) => *button*

<radiobutton>:

Options:

#"activebackground"
#"activeforeground"
#"bitmap"
#"command"
#"disabledforeground"
#"font"
#"height"
#"indicatoron"
#"justify"
#"state"
#"selectcolor"
#"text"
#"textvariable"
#"underline"
#"value"
#"variable"
#"width"
#"wraplength".

Functions:

select-value (*button*) => *button*
deselect-value (*button*) => *button*

<canvas>:

Options:

```
#"closeenough"
#"confine"
#"height"
#"insertbackground"
#"insertborderwidth"
#"insertofftime"
#"insertontime"
#"insertwidth"
#"scrollregion"
#"selectbackground"
#"selectborderwidth"
#"selectforeground"
#"width"
#"xscrollcommand"
#"xscrollincrement"
#"yscrollcommand"
#"yscrollincrement"
```

Functions:

xview (*canvas* :: **<canvas>**, *index*) => *canvas*
yview (*canvas* :: **<canvas>**, *index*) => *canvas*
find-items (*canvas* :: **<canvas>**, *spec*, *spec-args*) => *found-items*
focus (*canvas* :: **<canvas>**) => *result* :: **<canvas-item>**
focus-setter (*value* :: **<canvas-item>**, *canvas* :: **<canvas>**) => ()
scan-mark (*window* :: **<canvas>**, #*rest coords*) => *window*
scan-dragto (*window* :: **<canvas>**, #*rest coords*) => *window*
select-item (*window* :: **<canvas>**, *index*) => *window*
canvas-x (*canvas* :: **<canvas>**, *screen-x*, #*key grid-spacing* = 1) => *x*
canvas-y (*canvas* :: **<canvas>**, *screen-y*, #*key grid-spacing* = 1) => *y*
create-arc (*canvas*, *x1*, *y1*, *x2*, *y2*, #*rest opts*) => *item*
Options include: #"extent", #"fill", #"outline", #"outlinestipple:", #"start", #"stipple", #"style", #"tags",, #"width"
create-bitmap (*canvas*, *x1*, *y1*, #*rest opts*) => *item*
Options include: #"anchor", #"bitmap", #"background", #"foreground", #"tags"
create-line (*canvas*, *points* :: **<sequence>**, #*rest opts*) => *item*
Options include: #"arrow", #"arrowshape", #"capstyle", #"fill", #"joinstyle", #"smooth", #"splinesteps", #"stipple", #"tags", #"width"
create-oval (*canvas*, *x1*, *y1*, *x2*, *y2*, #*rest opts*) => *item*
Options include: #"fill", #"outline", #"stipple", #"tags", #"width"
create-polygon (*canvas*, *pnts* :: **<sequence>**, #*rest opts*) => *item*
Options include: #"fill", #"outline", #"smooth", #"splinesteps", #"stipple", #"tags", #"width"
create-rectangle (*canvas*, *x1*, *y1*, *x2*, *y2*, #*rest opts*) => *item*
Options include: #"fill", #"outline", #"stipple", #"tags", #"width"
create-text (*canvas*, *x1*, *y*, #*rest opts*) => *item*
Options include: #"anchor", #"fill", #"font", #"justify", #"stipple", #"tags", #"text", #"width"
create-window (*canvas*, *x1*, *y1*, #*rest opts*) => *item*
Options include: #"anchor", #"height", #"tags", #"width", #"window"
postscript (*canvas*, #*rest opts*) => *result* :: **<string>**
Options include: #"colormap", #"colormode", #"fontmap", #"height", #"pageanchor", #"pageheight",

#"pagewidth", #"pagex", #"pagey", #"rotate", #"width", #"x", #"y".

<entry>:

Options:

```
#"exportselection"
#"font"
#"insertbackground"
#"insertborderwidth"
#"insertofftime"
#"insertontime"
#"insertwidth"
#"justify"
#"selectbackground"
#"selectborderwidth"
#"selectforeground"
#"show"
#"state"
#"textvariable"
#"width"
#"xscrollcommand"
```

Functions:

icursor (*entry*, *index*) => *entry*
xview (*entry*, *index*) => *entry*
delete (*entry*, *index*, *#key end: last*) => *entry*
insert (*entry*, *index*, *#rest elements*) => *entry*
get-all (*entry*) => *result* :: **<string>**
get-elements (*entry*, *index*, *#key end*) => *result* :: **<string>**
scan-mark (*entry*, *#rest coords*) => *entry*
scan-dragto (*entry*, *#rest coords*) => *entry*
select-adjust (*entry*, *index*) => *entry*
select-clear (*entry*) => *entry*
select-from (*entry*, *index*) => *entry*
select-to (*entry*, *index*) => *entry*

<frame>:

Options:

```
#"colormap"
#"height"
#"width"
#"visual"
```

Supports no operations.

<label>:

Options:

```
#"bitmap"
#"font"
#"height"
#"justify"
#"text"
#"textvariable"
#"underline"
#"width"
#"wraplength"
```

Supports no operations.

<listbox>:

Options:

```
#"exportselection"
#"font"
#"height"
#"selectbackground"
#"selectborderwidth"
#"selectforeground"
#"selectmode"
#"setgrid"
#"width"
#"xscrollcommand"
#"yscrollcommand
```

Functions:

clear-selection (*listbox*, *first* :: **<integer>**,*#rest last*) => ()
current-selection(*listbox*, *#rest rest*) => *indices* :: **<sequence>**
nearest(*listbox*, *y-coord*) => *index* :: **<integer>**
size(*listbox*) => *result* :: **<integer>**
xview(*listbox*, *index*) => *listbox*
yview(*listbox*, *index*) => *listbox*
delete (*listbox*, *index*, *#key end*) => *listbox*
insert (*listbox*, *index*, *#rest elements*) => *listbox*
get-all (*listbox*) => *result* :: **<string>**
get-elements (*listbox*, *index*, *#key end*) => *result* :: **<string>**
scan-mark (*listbox*, *#rest coords*) => *listbox*
scan-dragto (*listbox*, *#rest coords*) => *listbox*
see (*listbox*, *index*) => ()
select-adjust (*listbox*, *index*) => *listbox*
select-clear (*listbox*) => *listbox*
select-from (*listbox*, *index*) => *listbox*
select-to (*listbox*, *index*) => *listbox*
selection-anchor-setter (*listbox*, *index*) => *listbox*
selection-includes? (*listbox*, *index*) => *in-selection?* :: **<boolean>**
set-selection (*listbox*, *first* :: **<integer>**, *#rest last*) => ()

<menu>:

Options:

```
#"activebackground"
#"activeborderwidth"
#"activeforeground"
#"disabledforeground"
#"font"
#"postcommand"
#"selectcolor"
#"tearoff"
```

Functions:

activate-entry (*menu*, *index*) => *menu*
delete (*menu*, *index*, *#key end*) => *menu*
configure-entry (*menu*, *index*, *#rest options*) => *menu*
entry-configuration (*menu*, *index*) => *result* :: **<sequence>**
invoke-entry (*menu*, *index*) => *result*
post (*menu*, *x*, *y*) => ()
post-cascade (*menu*, *index*) => ()
unpost (*menu*) => ()

yposition-entry (*menu*, *index*) => *result* :: **<integer>**
add-command (*menu*, *#key state*, *command*, *label*) => ()
add-checkbutton (*menu*, *#key variable*, *label*) => ()
add-radiobutton (*menu*, *#key variable*, *value*, *label*) => ()
add-cascade (*menu*, *#key menu*, *label*) => ()
add-separator (*menu*) => ()

Note that if you wish to assocaiate a menu with a menubutton, you must create it `in:` that button.

<message>:

Options:

```
#"aspect"
#"font"
#"justify"
#"text"
#"textvariable"
#"width"
```

Supports no operations.

<scale>:

Options:

```
#"activebackground"
#"bigincrement"
#"command"
#"digits"
#"font"
#"from"
#"label"
#"length"
#"orient"
#"repeatdelay"
#"repeatinterval"
#"resolution"
#"showvalue"
#"sliderforeground"
#"sliderlength"
#"state"
#"tickinterval"
#"to"
#"troughcolor"
#"variable"
#"width"
```

Functions:

get-value (*scale*) => *result* :: **<integer>**
set-value (*scale*, *value*) => *scale*;

<scrollbar>:

Options:

```
#"activebackground"
#"activerelief"
#"command"
#"elementborderwidth"
#"jump"
#"orient"
#"repeatdelay"
```

```
#"repeatinterval"
#"troughcolor"
#"width"
```

Functions:

scroll (*widget*, *#key*, *#all-keys*) => *scrollbar*
get-units (*scrollbar*) => (*#rest units* :: **<integer>**)
set-units (*scrollbar*, *#rest Units*) => *scrollbar*

The **scroll** function exists to reduce the inefficiency inherent in "standard" method of connecting widgets to scrollbars. This function creates a new scrollbar with the given orientation (i.e. "vertical" or "horizontal") and connects it via lower level mechanisms to the given widget. All of the applicable creation and packing options for scrollbars are accepted.

<text>:

Options:

```
#"exportselection"
#"font"
#"height"
#"insertbackground"
#"insertborderwidth"
#"insertofftime"
#"insertontime"
#"insertwidth"
#"selectbackground"
#"selectborderwidth"
#"selectforeground"
#"setgrid"
#"spacing1"
#"spacing2"
#"spacing3"
#"state"
#"tabs"
#"textvariable"
#"width"
#"wrap"
#"yscrollcommand".
```

Functions:

yview (*text*, *index*) => *text*
delete (*text*, *index*, *#key end*) => *text*
insert (*text*, *index*, *#rest elements*) => *text*
get-all (*text*) => *result* :: **<string>**
get-elements (*text*, *index*, *#key end*) => *result* :: **<string>**
scan-mark (*text*, *#rest coords*) => *text*
scan-dragto (*text*, *#rest coords*) => *text*
select-adjust (*text*, *index*) => *text*
select-clear (*text*) => *text*
select-from (*text*, *index*) => *text*
select-to (*text*, *index*) => *text*
tags (*text*) => *result* :: **<sequence>**

<toplevel>:

Options:

```
#"colormap"
#"height"
#"screen"
```

```
#"visual"
#"width"
```

Functions:

tk-dialog (*toplevel*, *title*, *text*, *bitmap*, *default*, *#rest buttons*) => *button* :: **<integer>**

Tk-dialog automatically creates a dialog box with the given title, text, bitmap and buttons. The default argument is an integer that corresponds to the button that should be made the default (beginning at zero for the left-most button)

These operations will hopefully be described at greater length in future editions of this document, but for now you are encouraged to check the Tcl/Tk manual for further information.

10. Text indices

The types **<text-index>**, **<text-mark>**, and **<text-tag>** all represent references to the contents of a **<text>** window. Functions are provided to let you create and manipulate instances of these types.

<Text-index>s represent an "absolute" index within a **<text>** object. They have two slots -- **line** and **character**. **Make** on **<text-index>** requires a `line:` keyword and accepts an optional `character:` keyword (which defaults to 0). Lines numbers start at 1, while characters start at 0, so you could specify the first character of a **<text>** by calling `make(<text-index>, line: 1, character: 0)`

The following functions create and manipulate **<text-index>**s:

text-at (*line*, *character*) => *result* :: **<text-index>**
line-end (*line*) => *result* :: **<string>**
as (*cls* == <text-index>, *value* :: **<string>**) => *string*
as (*cls* == <string>, *value* :: **<text-index>**) => *text-index*

The **text-at** function provides a convenient shorthand for making **<text-index>**s. You might, for example replace the above **make** call with `text-at(1, 0);`

Line-end creates a reference to the last character of the named line.

11. Text marks

<Text-mark> objects represent more abstract character positions. It will continue to refer to the same character even if other text is added or deleted. They have two slots -- **value** and **name**, and are associated with a particular **<text>** widget via the required **make** keyword `in:`.

If you make a **<text-mark>** with the name of a "standard" Tcl/Tk mark, such as "end" or "insert", then the new object will pick up the values already associated with the Tcl/Tk mark.

The following functions create or manipulate **<text-mark>**s:

as (*cls* == **<string>**, *object* :: **<text-mark>**) => *string*
as (*cls* == **<text-index>**, *mark* :: **<text-mark>**) => *text-index*
marks (*text* :: **<text>**) => *result* :: **<sequence>**

The **marks** function returns a sequence containing all of the marks within a given **<text>**. This will include both the "standard" Tcl/Tk marks and any that you have created. These will not necessarily be **==** to the original objects.

12. Text tags

<Text-tag>s represent attributes which may be associated with zero or more sequences of characters in a specific **<text>**. You may specify different display properties or event bindings for the characters that have been associated with a particular **<text-tag>**.

<Text-tag>s are created within a particular **<text>** via **make**, which requires an `in:` keyword. Optional keywords include `name:`, `bgstipple:`, `fgstipple:`, `font:`, `justify:`, `lmargin1:`, `lmargin2:`, `offset:`, `overstrike:`, `relief:`, `rmargin:`, `spacing1:`, `spacing2:`, `spacing3:`, `tabs:`, `wrap:`, and `underline:`. Although tags are not widgets, they support the **configure**, **configuration**, and **bind** functions. You may add ranges of characters to **<text-tag>**s via **add-tag**. A complete list of functions upon **<text-tag>**s follows:

configure (*tag*, *#rest options*) => *tag*
configuration (*tag*, *index*) => *result* :: **<sequence>**
bind (*tag*, *event* :: **<string>**, *command*) => *tag*
as (*cls* == **<string>**, *object* :: **<text-tag>**) => *string*
add-tag (*tag*, *#key start*, *end*: *last*) => *tag*
remove-tag (*tag*, *#key start*, *end: last*) => *tag*
tags (*text* :: **<text>**) => *result* :: **<sequence>**
delete-tag (*tag*) => ()
raise-tag (*tag*, *#key past* :: **<text-tag>**) => ()
lower-tag (*tag*, *#key past* :: **<text-tag>**) => ()
next-range (*tag*, *#key start*, *end*) => *result* :: **false-or**(**<pair>**)

13. Canvas items and tags

<canvas-item>s represent different graphical entity within a **<canvas>**. Each **create-shape** function returns a **<canvas-item>** corresponding to the newly created shape. These may be used to delete, resize, move, or respecify the shape. Although canvas items are not widgets, they support the **configure**, **configuration** and **bind** functions. The configuration options which are accepted for a given item depend upon its type. You should consult the list of **create-...** functions for a list of options.

<canvas-tag>s are a convienient way of grouping various canvas items. If more than one item has a given tag, then any operation performed on the tag will be applied to all items with that tag. Tags may be created using the make method with required keywords `#"in"`, the canvas that contains that tag, and `#"name"`, a string identifier. Tags may be assigned to **<canvas-item>**s when they are created, by giving a single **<canvas-tag>** or a sequence of **<canvas-tag>**s in the `#"tags"` keyword.

The following functions operate upon canvas items and canvas tags:

configure (*item*, *#rest options*) => *item*
configuration (*item*, *index*) => *result* :: **<sequence>**
bind (*item*, *event* :: **<string>**, *command*) => *item*
delete-item (*item*) => ()
raise-item (*item*, *#key past* :: **<canvas-item>**) => ()
lower-item (*item*, *#key past* :: **<canvas-item>**) => ()
move-item (*item*, *x* :: **<object>**, *y* :: **<object>**) => ()
scale-item (*item*, *x-origin*, *y-origin*, *x-scale*, *y-scale*) => ()
item-coords (*item*) => *result* :: **<sequence>**
item-coords-setter (*value* :: **<sequence>**, *item*) => ()
item-type (*item*) => *result* :: **<string>**
add-canvas-tag (*item*, *spec*, *spec-args*) => ()
get-canvas-tags (*item*) => *tags* :: **<sequence>**
delete-canvas-tag (*item*, *tag-to-delete*) => ()

14. Window Information commands

There are a variety of window information commands that return various data about the state of the widgets, most of these corresponding to Tcl/TK "winfo" commands. There are also two new objects that are returned by some of these functions, **<point>** and **<rgb-color>**. **<point>** has slots **x** and **y**, both integers, and **<rgb-color>** has slots **red**, **green**, and **blue**, all integers. Functions that return these classes come in two forms for flexibility in programming. One form has the class name

appended to the function name , which returns the named object (e.g. **mouse-point** returns a **<point>**). The second form lists the components as part of the function name, and returns each component individually, as multiple values (e.g. **mouse-x-y** returns the values **x** and **y**). A complete list of the window information functions follows.

colormap-cells (*window*) => *c-m-cells* :: **<integer>**
colormap-full? (*window*) => *c-m-full?* :: **<boolean>**
children (*window*) => *child-seq* :: **<sequence>**
window-containing-x-y (*x*, *y*, *#key on-display-of* = #f) => *window*
window-containing-point (*point* :: **<point>**, *#key on-display-of* = #f) => *window*
depth (*window*)=> *colors* :: **<integer>**
exists? (*window*) => *exist?* :: **<boolean>**
distance-to-float-pixels (*dist*, *#rest units*) => *pixels*
geometry (*window*) => *geom-string*
height (*window*) => *h*
X-id (*window*) => *id* :: **<integer>**
mapped? (*window*) => *map?* :: **<boolean>**
geometry-manager (*window*) => *manager* :: **<string>**
window-with-X-id (*id* :: **<integer>**, *#key on-display-of* = #f) => *window*
distance-to-pixels (*dist*, *#rest units*) => *pixels* :: **<integer>**
mouse-x-y (*window*) => (*x*, *y*)
mouse-point (*window*) => *point* :: **<point>**
requested-height (*window*) => *h*
requested-width (*window*) => *w*
width (*window*) => *w*
color-to-r-g-b (*window*, *color* :: **<string>**) => (*red*, *green*, *blue*)
color-to-rgb-color (*window*, *color* :: **<string>**) => *rgb-color* :: **<rgb-color>**
abs-position-x-y (*window*) => (*x*, *y*)
abs-position-point (*window*) => *point* :: **<point>**
screen-name (*window*) => *name*
screen-colormap-cells (*window*) => *num-cells* :: **<integer>**
screen-depth (*window*) => *bit-depth* :: **<integer>**
screen-height (*window*) => *h*
screen-width (*window*) => *w*
server (*window*) => *srv* :: **<integer>**
toplevel (*window*) => *top* :: **<window>**
viewable? (*window*) => *view?*
visual-class (*window*) => *vis-class* :: **<string>**
available-visuals (*window*) => *avail-vis* :: **<sequence>**
virtual-root-height (*window*) => *h*
virtual-root-width (*window*) => *w*
virtual-root-position-x-y (*window*) => (*x*, *y*)
virtual-root-position-point (*window*) => *point* :: **<point>**
x-y-in-parent (*window*) => (*x*, *y*)
point-in-parent (*window*) => *point* :: **<point>**

Notes on the functions:

Distance-to-pixels and **distance-to-float-pixels** take a units paramater that could be one of `#"centimeters"`, `#"inches"`, `#"millimeters"`, or `#"points"`.

X-y-in-parent and **point-in-parent** return the position of the upper-left corner of the window in it's parent window.

Abs-position-x-y and **abs-position-point** return the position of the upper-left corner of the window relative to the root window.

Mouse-x-y and **mouse-point** return the position of the pointer relative to the argument window.

Anything that returns a **<window>** or a sequence of **<window>**s will return a new object which refers to the same Tk widget. A **\=** method is provided for comparing these to the previously existing ones.

15. Requested Enhancements

Allow event bindings to access the window, mouse coords, character, and other "%" substitutions.

Add better support for scale and scrollbar widgets.

Provide a "focus" command.

Revise the interface to provide a more uniform object-oriented integration of **<window>**s, **<text-tag>**s, and **<canvas-item>**s. (This would probably be an incompatible change.)

16. The Extension Protocol

Because early versions of Dylan/Tk don't support the entire functionality of Tcl/Tk, users may wish to interface directly with the underlying Tcl/Tk interpreter. This capability is provided by the Tk-extension module. (Note: if any user decides to use these capabilities to provide a firm interface to a Tcl/Tk subsystem (e.g. the "wm" command), we would willingly incorporate such code into future versions of this library.)

The following functions are exported:

tk-as (*cls* :: **<class>**, *value*) => *result* :: **<object>** [Generic Function]

Performs conversions comparable to **as**, but adds some which do not generalize outside Tk -- for example, `tk-as(<boolean>, "1") == #t`.

You may wish to add methods to this function for any new Dylan types you add.

put-tk-line (*#rest pieces*) => () [Function]

Converts all of the input arguments into strings (using **tk-as**), concatentates them into a line of tk input, and sends then to the running tk process. This routine expects no results and therefore does not wait around for them.

You should be careful to include spaces between distinct tokens in the input, since **put-tk-line** will cheerfully combine several strings into a single Tk token.

call-tk-function (*#rest pieces*) => *result* :: **<string>** [Function]

Concatenates the input into a tk command, executes it, and returns the result as a string. Pauses execution of the current thread until the tk function returns. You may need to call **tk-as**, **tk-unquote**, or **parse-tk-list** on the result depending upon the expected result type.

join-tk-args (*#rest object*) => *result* :: **<string>** [Function]

Creates a string containing the tk representation (via **tk-as**) of each argument, separated by spaces.

make-option (*option*, *value*) => *option* :: **<string>** [Function]

Creates a tk "option" switch out of option name and an arbitrary value. The value may be an **<integer>**, **<string>**, **<symbol>**, or **<true>**. If the option is #f, then an empty string (i.e. no option) will be returned.

parse-tk-list (*string*, *#key depth*, *unquote*, *start*, *end*) => *sequence* [Function]

Takes a string returned by a tk function and converts it into a sequence of values. Values will either be 'words' or subsequences, depending upon whether the string contains list grouping (i.e. '{' and '}') characters. If depth is not #f, all groupings below the given depth will be returned as simple strings. If *unquote* is true, then result strings will be filtered through **tk-unquote**.

tk-quote (*object* :: **<object>**) => *result* :: **<string>** [Function]

Performs whatever quotation is necessary to get the data to WISH intact. For most objects, this is the same as `tk-as(<string>, ...)`, but it does extra quotation on strings to make sure brackets and such like come out OK.

tk-unquote (*string* :: **<string>**) => *result* :: **<string>** [Function]

Strips away all quotation from a string. This will undo the various escapes required to imbed brackets and such like inside a tk list. This should be called for any user level string, just as **tk-as** might be called for any other data type.

anonymous-name (*#key prefix* = "w") => *result* :: **<string>** [Function]

Generates a unique new (but unexciting) name for a window.

The Collection-extensions Library

Designed by the Gwydion Project

Copyright (c) 1994, 1995, 1996 Carnegie Mellon University All rights reserved.

1. Introduction

The various modules in this library contain a few new types and operations which are compatible with the collection types specified in the Dylan Reference Manual, but which are not part of that specification.

It is to be expected that more collection types will appear in time, and they will likely be added to this library. This may also result in reorganizations which could force incompatible changes to the existing modules. We hope to minimize such imcompatibilities and, when forced to them, will include sufficient information to facilitate conversion of existing code.

`Collection-extensions` exports these modules:

`self-organizing-list`
: Provides "self-organizing lists". These explicit key collections provide roughly the semantics of hash tables, but use a probabilistic implementation which provides O(n) worst case performance but can provide very fast constant time access in the best case.

`subseq`
: Provides "subsequences", which represent an aliased reference to some part of an existing sequence. These are analogous to slices (in Ada or Perl) or displaced arrays (in Common Lisp). Subsequences are themselves subclasses of **<sequence>**, and can therefore be passed any **<collection>** or **<sequence>** operation.

`vector-search`
: Provides a small assortment of specialized operations for searching and modifying **<vector>**s. These operations are analogous to existing collection operations but provide keywords and efficiency improvements which are meaningful only within the more limited domain.

2. The Self-Organizing-List Module

The "Self Organizing List" is a poor man's hash table. More precisely, **<self-organizing-list>** is a subclass of **<mutable-explicit-key-collection>** and **<stretchy-collection>** for which addition and retrieval are both linear in the worst case, but which use a probabilistic strategy which yields nearly constant time in the best case.

Because they have a very low overhead, self-organizing lists may provide better peformance than hash tables in cases where references have a high degree of temporal locality. They may also be useful in situations where it is difficult to create a proper hash function.

Instantiate **<self-organizing-list>**s with

```
make(<self-organizing-list>, test: test)
```

Test is expected to be an equality function. In particular, it is expected to satisfy the identity and transitivity requirements as described in the DRM. If not specified, *test* defaults to `\==`.

3. The Subsequence Module

<Subsequence> is a new subclass of **<sequence>**. A subsequence represents an aliased reference to some part of an existing sequence. Although they may be created by **make** (with required keywords `source:`, `start:` and `end:`) on one of the instantiable subclasses, they are more often created by calls of the form

```
subsequence(sequence, start: 0, end: 3)
```

where `start:` and `end:` are optional keywords which default to the beginning and end, respectively, of the source sequence. No other new operations are defined for subsequences, since all necessary operations are inherited from **<sequence>**.

Because subsequences are aliased references into other sequences, several properties must be remembered:

- The contents of a subsequence are undefined after any destructive operation upon the source sequence.
- Destructive operations upon subsequences may be reflected in the source. The results of **reverse!** and **sort!** should be expected to affect the source sequence for vector subsequences.

If the source sequences are instances of **<vector>** or **<string>**, then the implementation will use subclasses of **<subsequence>** which are also subclasses of **<vector>** or **<string>**.

Efficiency notes:

- The implementation tries to insure that subsequences of subsequences can be accessed as efficiently as the original subsequence. (For example, the result of
  ```
  subsequence(subsequence(source, start: 1), start: 2)
  ```
  would produce a subsequence identical to the one produced by
  ```
  subsequence(source, start: 3)
  ```
- Vector subsequences, like all other vectors, implement constant time element access.
- Non-vector subsequences may take non-constant time to create, but will provide constant-time access to the first element. This should produce the best performance provided some element of the subsequence is accessed at least once.

4. The Vector-Search Module

The vector-search module provides basic search and replace capabilities upon restricted subsets of **<sequence>** -- primarily **<vector>**. Exploiting the known properties of these types yields substantially better performance than can be achieved for sequences in general.

The following functions are supplied:

find-first-key(*vector*, *predicate?*, #key *start*, *end*, *failure*) => *key* [Function]
Find the index of first element (after *start* but before *end*) of a vector which satisfies *predicate?*. If no matching element is found, return *failure*. The defaults for *start*, *end* and *failure* are, respectively, `0`, `size(vector)`, and `#f`. This function is like **find-key**, but accepts `start:` and `end:` rather than `skip:`.

find-last-key(*vector*, *predicate?*, #key *start*, *end*, *failure*) => *key* [Function]
This is like **find-first-key**, but goes backward from *end*.

The Table-Extensions Library

1. Introduction

The `Table-Extensions` library contains extensions to Dylan **<table>**s. It exports one module named `Table-Extensions`.

2. The Table-Extensions Module

<string-table> [Instantiable Sealed Class]

This class is a subclass of **<table>**. It is a table that has **<string>**s for keys (compared with \=) and **<object>**s for elements. It is an error to use a key that is not a **<string>**. It is an error to modify a key once it has been used to add an element to a **<string-table>**.

<case-insensitive-string-table> [Instantiable Sealed Class]

This class is a subclass of **<table>**. It is a table that has **<string>**s for keys (compared with **case-insensitive-equal**) and **<object>**s for elements. It is an error to use a key that is not a **<string>**. It is an error to modify a key once it has been used to add an element to a **<case-insensitive-string-table>**.

<hash-state> [Type]

Anything the DRM describes as a hash state is an instance of this type. This includes the second return value of **merge-hash-codes** and of all hash functions, and the second and fourth parameters to **merge-hash-codes**.

collection-hash [Function]

Arguments

key-hash-function :: <function>
elt-hash-function :: <function>

Values

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

Description

This function hashes every element of collection using *key-hash-function* on the keys and *elt-hash-function* on the elements. This function is equivalent to the following reference implementation:

sequence-hash [Function]

Arguments

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

Values

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

Description

This function hashes every element of sequence using *elt-hash-function*, merging the resulting hash codes in order.

values-hash [Function]

Arguments
elt-hash-function :: <function>
#*rest* values
Values
hash-id :: <integer>
hash-state :: <hash-state>
Description
This function hashes each object in the #*rest* arguments using *elt-hash-function*, merging the resulting hash codes in order.

string-hash [Function]

Arguments
string :: <string>
Values
hash-id :: <integer>
hash-state :: <hash-state>
Description
This function produces hash codes for strings using the equality test \=.

case-insensitive-string-hash [Function]

Arguments
string :: <string>
Values
hash-id :: <integer>
hash-state :: <hash-state>
Description
This function produces hash codes for strings using the equality test **case-insensitive-equal**.

case-insensitive-equal [Function]

Arguments
string1 :: <string>
string2 :: <string>
Values
answer :: <boolean>
Description
Performes a case insensitive comparison of the strings.

remove-all-keys! [Open Generic Function]

Arguments
coll :: <mutable-explicit-key-collection>
Values
coll :: <mutable-explicit-key-collection>
Description
Modifies collection so that the collection no longer contains any keys or elements (i.e., is empty).

remove-all-keys! [G. F. Method]

Arguments
coll :: <mutable-explicit-key-collection>
Values
coll :: <mutable-explicit-key-collection>
Description
This method implements **remove-all-keys!** by repeated calls to **remove-key!**.

remove-all-keys! [Sealed G. F. Method]

Arguments

table :: **<table>**

Values

table :: **<table>**

Description

There is a predefined method on **<table>** which does not necessarily use **remove-key**.

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. Extensions

The Gwydion Project hopes that other Dylan implementations will also implement our Table-Extensions library. However, there are certain things that we realize other implementors are not so likely to implement; we call these things "extensions" to the Table-Extensions proposal.

The following definitions are part of the Gwydion Table-Extensions library, but are not part of the current Table-Extensions proposal.

<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 function *equal-hash [Generic Function]*).

<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 uses an object's identity (that is, its location in the heap) as a means of computing a hash ID. In practice, this means hash IDs must be constructed using only **value-hash** and **merge-hash-codes**. 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.

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. `Table-Extensions` provides methods specialized for the following types: **<string>**, **<integer>**, **<float>**, **<character>**, **<symbol>**, and **<boolean>.**

Caveat: This generic function is not related to the similarly named **values-hash**.

The String-extensions Library

Designed by the Gwydion Project

1. Introduction

`String-extensions` is a library of routines for working with characters and strings. `String-extensions` exports these modules:

`String-Conversions`
: This module consists of various useful conversions involving strings.

`Character-type`
: This module is a Dylanized version of the C library `ctype.h`

`String-hacking`
: This module exports miscellanous functions and data structures that are useful when working with strings and characters.

`Substring-search`
: This module contains methods for searching for fixed substrings rather than general regular expressions.

2. The String-Conversions Module

The `String-Conversions` module consists of various useful conversions involving strings. They are:

string-to-integer(*string*, #key *base*) => *integer* [Function]
integer-to-string(*integer*, #key *base*) => *string* [Function]
digit-to-integer(*character*) => *integer* [Function]
integer-to-digit(*integer*) => *character* [Function]

Base defaults to 10, and is the radix for the number system to convert from/to. Bases below 2 are errors, as are bases above 36. When converting from a string, the string must exactly describe a number, with no excess characters. **Digit-to-integer** will signal an error if the digit is non-alphanumeric. Errors will be signalled for all invalid input.

as(<string>, *character*) [G.F. Method]

Turns a character into the appropriate string of length one.

3. The Character-type Module

`Character-type` is a Dylanized version of the C library `ctype.h` It contains the following functions:

Function and Argument Type	Returns `#t` for these characters
alpha?(*character*)	a-zA-Z
alphabetic?(*character*)	a-zA-Z *(same as* **alpha?***)*
digit?(*character*)	0-9
alphanumeric?(*character*)	a-zA-Z0-9
whitespace?(*character*)	*Space, tab, newline, formfeed, carriage return*
uppercase?(*character*)	A-Z
lowercase?(*character*)	a-z
hex-digit?(*character*)	0-9a-f
punctuation?(*character*)	`,./<>?;\:"|'[]{}!@#$%^&*()-=_+`~`
graphic?(*character*)	*alphanumeric or punctuation*
printable?(*character*)	*graphic or whitespace*
control?(*character*)	*not printable*

4. The String-hacking Module

The `String-hacking` module exports miscellanous functions and data structures that are useful when working with strings and characters.

add-last(*stretchy-sequence*, *object*) => *stretchy-sequence* [Generic Function]
add-last(*string*, *character*) => *string* [G.F. Method]
Like **add** except it's guarenteed to add the character to the end of the string.

predecessor(*character*) => *character* [Function]
Get the character before this character. Equivalent to

```
as(<character>, -1 + as(<integer>, character))
```

successor(*character*) => *character* [Function]
Get the character after this character. Equivalent to

```
as(<character>, 1 + as(<integer>, character))
```

case-insensitive-equal(*object1*, *object2*) [Generic Function]
case-insensitive-equal(*string1*, *string2*) [G.F. Method]
case-insensitive-equal(*character1*, *character2*) [G.F. Method]
Does a case insensitive equality test. Methods are provided only for strings and characters, not general collections.

<character-set> [Sealed Abstract Class]
<case-sensitive-character-set> [Class]
<case-insensitive-character-set> [Class]

A **<character-set>** is a non-mutable subclass of **<collection>**, and is conceptually an unordered set of characters. Dylan collection elements always have keys, so to fit sets into Dylan, the key of an element of a character set is the element itself. There are two instantiable subclasses of **<character-set>**, **<case-sensitive-character-set>** and **<case-insensitive-character-set>**. **<character-set>** is not instantiable; one must always specify one of the instantiable subclasses when creating a character set.

There are two ways of making a character set. The first is a method for make using the *description:* keyword. The value that follows the *description:* keyword is a string that describes the set using a notation like a regular expression character set, except without the '`[`' and '`]`' delimiters. For example,

```
make(<case-sensitive-character-set>, description: "a-z")
```

would be the set of all lowercase alphabetic characters.

A second way to create character sets is to use an **as** method. The **as** method basically takes a collection of characters and discards the keys of these characters. Example:

```
as(<case-insensitive-character-set>,
   "abcdefghijklmnopqrstuvwxyz")
```

is again the set of all lowercase alphabetic characters. It is important to realize that the **as** method does *not* take a description:

```
as(<case-sensitive-character-set>, "a-z")
```

returns the set of '`a`', '`-`', and '`z`', not the set of all lowercase alphabetic characters.

The most useful operation on character sets is **member?**, which does what one would expect. Another useful operation is the **forward-iteration-protocol**. This basically calls **member?** on every possible character until it finds a character that is a member of the set. This means that in a **<case-insensitive-character-set>**, both '`a`' and '`A`' will come up.

<byte-character-table> [Class]

A byte-character-table is a vector that uses byte characters as indices instead of integers. The following are equivalent:

```
regular-vector[as(<integer>, character)]

byte-character-table[character]
```

<byte-character-table> has absolutely no relation to **<table>**. It is simply a **<mutable-explicit-key-collection>**.

5. The Substring-search Module

`Substring-search` contains methods for searching for fixed substrings. It is as similar to the regular-expression module as we could make it. (See the document *The Regular-Expressions Library* for details about regular expressions, and about the "make-fooer" convention. However, note that while the "make-fooer" convention is obsolete for regular expression functions, it is not obsolete for substring searching.) Substring functions work only on byte strings, and are always case sensitive.

substring-position [Generic Function]

(big-string, search-for-string, #key start, end)
=> position-or-false;

Returns the position of the *search-for-string* in the *big-string* (or that portion of the *big-string* specified by *start:* and *end:*). This search is always case sensitive.

This function uses the Boyer-Moore algorithm for long strings, and a simple dumb search for short strings. It should yield good performance under all circumstances.

make-substring-positioner [Generic Function]

(search-for-string) => *an anonymous positioner*
method (big-string, #key start, end) => *position-or-false*

Does the obvious.

substring-replace [Generic Function]

(big-string, search-for-string, replace-with-string, #key count, start, end)
=> *replaced-string*

Replaces the substring, or the first *count* instances of it if *count:* is specified. Note this function does not support *start:* or *end:*.

make-substring-replacer [Generic Function]

(search-for :: <*byte-string*>*, #key replace-with)*
=> *an anonymous function replacer that is either*
method (big-string, #key count, start, end) => *new-string*
or
method (big-string, replace-with-string, #key count, start, end)

Does the obvious.

The Regular-Expressions Library

Designed by the Gwydion Project

1. Introduction

The `Regular-expressions` library exports the `Regular-expressions` module, which contains various functions that deal with regular expressions (*regexps*). The module is based on Perl (version 4), and has the same semantics unless otherwise noted. The syntax for Perl-style regular expressions can be found on page 103 of *Programming Perl* by Larry Wall and Randal L. Schwartz. There are some differences in the way `String-extensions` handles regular expressions. The biggest difference is that regular expressions in Dylan are case insensitive by default. Also, when given an unparsable regexp, `String-extensions` will produce undefined behavior while Perl would give an error message.

A regular expression that is grammatically correct may still be illegal if it contains an infinitely quantified sub-regexp that may match the empty string. That is, if `R` is a regexp that can match the empty string, then any regexp containing `R*`, `R+`, and `R{n,}` is illegal. In this case, the `Regular-expressions` library will signal an **<illegal-regexp>** error when the regexp is parsed. Note: Perl also has this restriction, although it isn't mentioned in *Programming Perl*.

In previous versions of the regular-expressions library, each basic function had a companion function that would pre-compute some information needed to use the regular expression. By using the companion function, one could avoid recomputing the same information. In the present version, the regular-expressions library caches this information, so the companion functions are no longer necessary and should be considered obsolete. However, they have been kept for backwards compatibility.

Companion functions differ in details, but they all essentially return curried versions of their corresponding basic function. For example, the following two pieces of code yield the same result:

```
regexp-position("This is a string", "is");

let is-finder = make-regexp-positioner("is");
is-finder("This is a string");
```

Both pieces of code should have roughly the same performance, even if the code is inside a loop.

2. Exported Names

The following names are exported by the `Regular-Expressions` module of the `Regular-Expressions` library:

regexp-position [Function]
(big-string, regexp, #key start, end, case-sensitive)
=> variable-number-of-marks-or-#f

This function returns the index of the start of the regular expression in the *big-string*, or `#f` if the regular expression is not found. As a second value, it returns the index of the end of the regular expression in the *big-string* (assuming it was found; otherwise there is no second value). These values are called marks, and they come in pairs, a start-mark and an end-mark. If there are groups in the regular expression, **regexp-position** will return an additional pair of marks (a start and an end) for each group. If the group is matched, these marks will be integers; if the group is not matched, the marks will be `#f`. So

```
regexp-position("This is a string", "is");
```

returns `values(2, 4)`, and

```
regexp-position("This is a string", "(is)(.*)ing");
```

returns `values(2, 16, 2, 4, 4, 13)`, while

```
regexp-position("This is a string", "(not found)(.*)ing");
```

returns `#f`. Marks are always given relative to the start of *big-string*, not relative to the *start:* keyword.

Start: and *end:* specify what part of *big-string* to look at, and they default to the beginning and end of the string, respectively. *Case-sensitive* defaults to false.

make-regexp-positioner [Function]

(regexp, #key byte-characters-only, need-marks, maximum-compile, case-sensitive)
=> an anonymous positioner
method (big-string, #key start, end)

Make-regexp-positioner can return several different types of positioners, and it is up to the user to specify what kind of positioner the user wants. It returns a positioner that works like **regexp-position**. *Needs-marks*, *byte-characters-only*, and *maximum-compile* are ignored, and are included only for backwards compatibility.

Note: This function is obsolete.

regexp-replace [Function]

(big-string, search-for-regexp, replace-with-string, #key count, case-sensitive, start, end)
=> new-string

This replaces all occurrences of *regexp* in *big-string* with replace-string. If *count:* is specified, it replaces only the first count occurrences of *regexp*. (This is different from Perl, which replaces only the first occurrence unless `/g` is specified) *Replace-string* can contain backreferences to the regexp. For instance,

```
regexp-replace("The rain in Spain and some other text",
               "the (.*) in (\\w*\\b)", "\\2 has its \\1")
```

returns `"spain has its rain and some other text"`. If the subgroup referred to by the backreference was not matched, the reference is interpreted as the null string. For instance,

```
regexp-replace("Hi there", "Hi there(, Bert)?",
               "What do you think\\1?")
```

returns `"What do you think?"` because `", Bert"` wasn't found.

make-regexp-replacer [Function]

(regexp, #key replace-with, case-sensitive)
=> an anonymous replacer function that is either
method (big-string, #key count, start, end)
or
method (big-string, replace-string, #key count, start, end)

The first form is returned if the *replace-with:* keyword isn't supplied, otherwise the second form is returned.

Note: This function is obsolete.

translate [Generic Function]

(big-string, from-string, to-string, #key delete, start, end)
=> new-string

This is equivalent to Perl's `tr///` construct. *From-string* is a string specification of a character set, and *to-string* is another character set. **Translate** converts *big-string* character by character, according to the sets. For instance,

```
translate("any string", "a-z", "A-Z")
```

will convert `"any string"` to all uppercase: `"ANY STRING"`.

Like Perl, character ranges are not allowed to be "backwards". The following is not legal:

```
translate("any string", "a-z", "z-a")
```

(This restriction may be removed in future releases) Unlike Perl's `tr///`, **translate** doesn't return the number of characters translated.

If *delete:* is true, any characters in the *from-string* that don't have matching characters in the *to-string* are deleted. The following will remove all vowels from a string and convert periods to commas:

```
translate("any string", ".aeiou", ",", delete: #t)
```

Delete: is false by default. If delete: is false and there aren't enough characters in the *to-string*, the last character in the *to-string* is reused as many times as necessary. The following converts several punctuation characters into spaces:

```
translate("any string", ",./:;[]{}()", " ");
```

Start: and *end:* indicate which part of *big-string* to translate. They default to the entire string.

Caveats: **Translate** is always case sensitive.

translate [G.F. Method]

(big-byte-string, from-byte-string, to-byte-string, #key delete, start, end)
=> new-string

The only predefined method of **translate** operates only on byte strings.

make-translator [Generic Function]

(from-string, to-string, #key delete)
=> an anonymous translator
method (big-string, #key start, end) => new-string

Does what you'd expect it to.

make-translator [G.F. Method]

(from-byte-string, to-byte-string, #key delete)
=> an anonymous translator
method (big-string, #key start, end) => new-byte-string

Again, the existing method on make-translator only handles byte strings.

split [Function]

(regexp, big-string, #key count, remove-empty-items, case-sensitive, start, end)
=> a variable number of strings

This is like Perl's **split** function. It searches *big-string* from occurrences of *regexp*, and returns substrings that were delimited by that regexp. For instance,

```
split("-", "long-dylan-identifier")
```

returns `values("long", "dylan", "identifier")`. Note that what matched the regexp is left out.

Remove-empty-items, which defaults to true, magically skips over empty items, so that

```
split("-", "long--with--multiple-dashes)
```

returns `values("long", "with", "multiple", "dashes")`. *Count* is the maximum number of strings to return. If there are *n* strings and *count* is specified, the first *count* - 1 strings are returned as usual, and the *count*'th string is the remainder, unsplit. So

```
split("-", "really-long-dylan-identifier", count: 3)
```

returns `values("really", "long", "dylan-identifier")`. If *remove-empty-items* is true, empty items aren't counted.

Case sensitive determines if the regexp for the delimiter should be considered case sensitive or not; it defaults to case-insensitive. *Start:* and *end:* indicate what part of the big string should be looked at for delimiters. They default to the entire string. For instance,

```
split("-", "really-long-dylan-identifier", start: 8)
```

returns `values("really-long", "dylan", "identifier")`.

Caveat: Unlike Perl, empty regular expressions are never legal regular expressions, so there is no way to split a string into a bunch of single character strings. Of course, in Dylan this is not a useful thing to do, so this is not really a problem.

make-splitter [Function]

(pattern :: <string>, #key case-sensitive)
=> an anonymous splitter
method (big-string, #key count, remove-empty-items, start, end) =>
buncha-strings

Does what you would expect.

Note: This function is obsolete.

join [Function]

(delimiter :: <string>, #rest strings) => big-string

Does the opposite of a **split**.

```
join(":", word1, word2, word3)
```

is equivalent to

```
concatenate(word1, ":", word2, ":", word3)
```

(and no more efficient) Note that there is no **make-joiner**.

<illegal-regexp> [Class]

When an illegal regular expression is parsed, an error of this type will be signalled.

3. Known bugs

The regular expression parser does a very poor job with syntactically invalid regular expressions. Depending on the expression, the parser may signal an error, improperly parse it, or simply crash.

A regular expression that matches a large enough substring can produce a stack overflow. This can happen much more

easily under d2c than under Mindy -- as few as two dozen lines of 80 column text under d2c for Windows.

The Transcendental Library

4. Introduction

The `Transcendental` library implements some common mathematical functions and constants, such as sine and cosine. All functions in the Transcendental library signal errors when given invalid arguments, and Dylan floating point precision contagion rules are obeyed. Precise contagion rules are given for each function below.

Note: The `Transcendental` library is not available on all platforms. At present, it will work only under HP/UX and Microsoft Windows.

5. Exported Names

The following names are exported from the `Transcendental` module of the `Transcendental` library:

$single-pi [Constant]
The value of pi as a **<single-float>**.

$double-pi [Constant]
The value of pi as a **<double-float>**.

sin (x :: <real>) => (y :: <float>) [Function]
cos (x :: <real>) => (y :: <float>) [Function]
tan (x :: <real>) => (y :: <float>) [Function]
Returns the sine, cosine, or tangent of x, respectively. x is given in radians.

The floating point precision of the result is given by the precision of x. The result will be a **<single-float>** if x is an integer.

asin (y :: <real>) => (x :: <float>), -1 <= y <= +1 [Function]
acos (y :: <real>) => (x :: <float>), -1 <= y <= +1 [Function]
Returns the arc sine or arc cosine of y, in radians. If y is not in the range [-1, +1], an error is signalled.

The floating point precision of the result is given by the precision of y. The result will be a **<single-float>** if y is an integer.

atan (y :: <real>) => (x :: <float>) [Function]
Returns the arc tangent of y, in radians.

The floating point precision of the result is given by the precision of y. The result will be a **<single-float>** if y is an integer.

atan2 (y :: <real>, x :: <real>) => (z :: <float>) [Function]
Returns the arc tangent of y/x, in radians. x may be zero if y is not zero. The signs of x and y are used to derived what quadrant the angle falls in.

The floating point precision of the result is given by the precision of y/x. The result will be a **<single-float>** if y/x is an integer.

sinh (x :: <real>) => (y :: <float>) [Function]
cosh (x :: <real>) => (y :: <float>) [Function]
tanh (x :: <real>) => (y :: <float>) [Function]
Returns the hyperbolic sine, hyperbolic cosine, or hyperbolic tangent of x, respectively.

The floating point precision of the result is given by the precision of *x*. The result will be a **<single-float>** if *x* is an integer.

$single-e [Constant]
The value of *e* as a **<single-float>**.

$double-e [Constant]
The value of *e* as a **<double-float>**.

log (*x* :: <real>, #key base) => (*y* :: <float>), *x* > 0, base > 1 [Function]
Returns the natural logarithm of *x* in base *base*. *base* defaults to the mathematical value of *e*. If *x* <= 0 or *base* <= 1, an error is signalled.

The floating point precision of the result is given by the precision of *x*. The result will be a **<single-float>** if *x* is an integer.

exp (*x* :: <real>) => (*y* :: <float>) [Function]
Returns *e* raised to the power *x*.

The floating point precision of the result is given by the precision of *x*.

\^ (*b* :: <real>, *x* :: <real>) => (*y* :: <float>) [G.F. method]
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.

The floating point precision of the result is given by the precision of *b*. The result will be a **<single-float>** if *b* is an integer.

\^ (*b* :: <integer>, *x* :: <integer>) => (*y* :: <integer>) [G.F. method]
Returns an integer result giving *b* raised to the power *x*. If *b* is 0 and x is not positive, an error is signalled. If *x* is negative, an error is signalled.

sqrt (*x* :: <real>) => (*y* :: <float>), *x* >= 0 [Function]
Returns the square root of *x*. If *x* < 0, an error is signalled.

The floating point precision of the result is given by the precision of *x*. The result will be a **<single-float>** if *x* is an integer.

isqrt (*x* :: <integer>) => (*y* :: <integer>), *x* >= 0 [Function]
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.

6. Unimplemented Functions

We intend to someday implement the following functions, but haven't done so yet:

asinh (*y* :: <real>) => (*x* :: <float>) [Function]
acosh (*y* :: <real>) => (*x* :: <float>) [Function]
atanh (*y* :: <real>) => (*x* :: <float>) [Function]
Returns the hyperbolic arc sine, hyperbolic arc cosine, or hyperbolic arc tangent of *y*, respectively.

The floating point precision of the result is given by the precision of *y*. The result will be a **<single-float>** if *y* is an integer.

The Time Library

Designed by the Gwydion Project

1. Introduction

`Time` is a library of routines for working with time and dates. `Time` exports two modules:

`Time`
: This module contains the basic time classes, functions to convert between them, and functions to get the current time.

`Time-IO`
: This module contains functions for basic time input and output.

Note: The `Time` library is not available on all platforms. At present, it will work only under HP/UX and Microsoft Windows.

2. The Time Module

<universal-time> [Class]

A **<universal-time>** is a simple **<general-integer>** representation of the number of seconds since 00:00:00 1 January 1970, UTC.

<decoded-time> [Sealed Immutable Class]

Decoded time is for representing absolute times in a nice human readable way. The **<decoded-time>** class contains the following constant slots: *seconds, minutes, hours, day-of-week* (0 represents Monday, 1 represents Tuesday, etc.), *day-of-month, month, year, daylight-savings-time?*, and *timezone* (in seconds west of UTC). Keywords of the same names are used to initialize these slots. A **<decoded-time>** may be incompletely specified; any of its slots may be `#f`. An additional init-keyword is `default-from:`. If `default-from:` is specified, any slot that isn't specified via an init-keyword will take its value of the *default-from* object. If `default-from:` is not specified, any slot not specified via an init-keyword will have a value of `#f`.

Note: If the timezone is specified in a **<decoded-time>,** the functions in the time library assume that it is the correct timezone and ignore the *daylight-savings-time?* flag. Therefore if you encode a **<decoded-time>** with the timezone representing EST and the *daylight-savings-time?* flag set to **#t,** it will convert the time assuming the local timezone is EST, *not* EDT.

<seconds> [Type]
<minutes> [Type]
<hours> [Type]
<day-of-week> [Type]
<day-of-month> [Type]
<month> [Type]
<year> [Type]
<timezone> [Type]

These are all limited integers that represent the acceptable ranges for the respective slots in the **<decoded-time>** class.

get-universal-time () => *universal-time* [Function]

Returns the current time as a **<universal-time>.**

get-decoded-time (#*key timezone*) => *decoded-time* [Function]

Returns the current time as a completely specified **<decoded-time>.** The time returned is relative to the timezone indicated by the *timezone* keyword parameter. The *timezone* parameter is the number of seconds west of UTC,

which defaults to the local timezone.

Known Bug: Mindy for MS-Windows will give an incorrect timezone. We have no idea why. (d2c under Windows gives the correct timezone)

decode-time (*universal-time, #key timezone*) => *decoded-time* [Function]
Returns the supplied **<universal-time>** as a completely specified **<decoded-time>.** The time returned is relative to the timezone indicated by the *timezone* keyword keyword paramter (in seconds west of UTC). The timezone defaults to the local timezone.

as(<decoded-time>, *universal-time*) => *decoded-time* [G.F. Method]
Equivalent to `decode-time(universal-time)`.

encode-time(*decoded-time*) => *universal-time* [Function]
Returns the supplied **<decoded-time>** as a **<universal-time>.** Only completely specified decoded-times can be encoded (as determined by **encodable-time?**. An error is returned if the decoded-time is not encodable.

encodable-time?(*decoded-time*) => *boolean* [Function]
Returns **#t** if the time can be encoded. A **<decoded-time>** is encodable if all its slots are specified, except possibly day-of-week.

3. The Time-IO Module

format-time [Function]
(stream, format-string, decoded-time)
=> *()*

This is a wrapper for the C function `strftime`. *Format-string* is a string describing the format for the output written to *stream*.

parse-time [Function]
(input-stream, control-string)
=> *decoded-time*

This function is similar to the C function `strptime`, but contains considerably fewer directives. (See below for a complete list of directives) **Parse-time** works by trying to match things in *input-stream* up against things in the *control-string*. A whitespace character inside *control-string* will match zero or more whitespace characters in *input-stream*, a *control-string* directive will match as described below, and all other characters match themselves (case sensitive). If a directive expects a number, a leading zero is optional but not required. Two consecutive directives that expect numbers must be separated by something non-numeric (such as whitespace or characters like `':'`, `'-'`, and `'/'`), otherwise **parse-time** can't tell where the first directive should stop parsing and the second one should begin. Similarly, two consecutive directives that expect words must be separated by something non-alphabetic.

If *input-stream* does not match *control-string*, a **<time-parsing-error>** will be signaled, and additional input will not be read from *input-stream*. An unknown *control-string* directive is an error, but is not a **<time-parsing-error>**.

Caveat: The C specification numbers days of the week differently from the Dylan Time library (C has Sunday as day 0, while Dylan uses Monday for day 0). The **parse-time** and **format-time** functions correctly convert between the two conventions, but this can have unexpected consequences:

```
make(<decoded-time>, day-of-week: 0)
```

is not the same as

```
let stream = make(<string-stream>, string: "0");
  parse-time(stream, "%w")
```

4. Parse-Time Directives

Control-strings consist of directives, much like format-strings from the `Format` library do. Unlike the `Format` library, **parse-time** directives are case sensitive. (This is mostly because "minute" and "month" start with the same letter) **Parse-time** supports the following directives:

`%a`
Day of the week, as a name. Either the full name (ie, `"Monday"`), or the three letter abbreviation (`"Mon"`) may be used; case does not matter.

`%b`
Month of the year, as a name. Either the full name (ie, `"January"`), or the three letter abbreviation (`"Jan"`) may be used; case does not matter.

`%d`
Day of the month.

`%D`
Date as either `"%m/%d/%y"` or `"%m-%d-%y"`.

`%H`
Hour on a 24 hour clock (ie, 0-23).

`%I`
Hour on a 12 hour clock (ie, 1-12).

`%m`
Month number (1-12).

`%M`
Minute (0-59).

`%p`
Either `"AM"` or `"PM"`. Case does not matter, but the `"AM"` or `"PM"` may not contain periods (`'.'`).

`%r`
Equivalent to "%I:M:S %p". In other words, 12-hour time with AM/PM.

`%S`
Seconds (0-59). Unlike C's `strptime`, leap-seconds and double-leap-seconds are not allowed.

`%T`
Equivalent to `"%H:%M:%S"`

`%w`
Weekday number, with 0 as Monday.

`%y`
Year. If less than 100, assume an implicit "19" in front of it.

`%Y`
Year. Do not assume any implicit "19".

`%%`
A single % character.

The Random Library

Designed by the Gwydion Project

The Random library provides random number support in several random number distributions. The Random module of the Random library exports the following names:

random [Function]
(upper-bound :: <integer>, #key state) => uniform-random-number :: <integer>;

Provides a random non-negative integer less than *upper-bound*. The distribution of the random numbers returned is uniform. The *state* argument is described below.

random-bits [Function]
(#key state) => uniform-random-number :: <fixed-integer>;

Returns a fixed integer that has **$random-bits-count** random bits. This function is the fastest way to get a random series of bits. The *state* argument is described below.

$**random-bits-count** [Constant]
The number of random bits that the **random-bits** function returns.

random-float [Function]
(upper-bound :: <number>, #key state) => uniform-random-number :: <float>;

Returns a random non-negative float less than *upper-bound*. The distribution of the random numbers returned is uniform. The *state* argument is described below.

random-gaussian [Function]
(#key mean = 0, standard-deviation = 1, state)

Returns a random float. The distribution of the random numbers returned is Gaussian (also known as "normal"), with a mean of *mean* and a standard deviation of *standard-deviation*. The *state* argument is described below.

random-exponential [Function]
(#key lambda = 1, state)

Returns a random non-negative float. The distribution of the random numbers returned is exponential, with a lambda of *lambda*. The *state* argument is described below.

<random-state> [Sealed Class]
<random-state> is the class of objects which serve as states for the random number generators. `Make()` of **<random-state>** accepts the *seed* keyword, which is an integer used as a seed value for the random-state. If *seed* is not provided, a seed based on the current time of day will be used. **<random-state>**s are not threadsafe; see below.

All of the random functions optionally take a **state:** keyword, which specifies the **<random-state>** to use. If a state is not specified, a default state will be used. In a multi-threaded environment, using the default state is guaranteed to be threadsafe. Implementations may use a unique default state object per thread, or a single default state object shared by all threads but accessed in a safe way. Random-states other than the default state objects are not threadsafe.

The Matrix Library

This library implements some basic matrix operations. These functions are not guarenteed to be stable or numerically sound, nor are they very optimized, but for smaller applications they should be suitable.

The **<matrix>** class is a subclass of **<array>**, so all operations on **<array>**s and collections in general should work as expected. One important limitation of matrices is that they are limited to two dimensions, and trying to use any other size results in an error.

1. Initialization

The generic function **make** will create matrices, with two keywords:

`dimensions:` a sequence of two elements which represent the rows and columns of the matrix respectively

`fill:` the objects that each element of matrix should be initialized to. This defaults to 0

matrix(#*rest row-vectors*) => **<matrix>** [Function]
Matrix will return a new matrix containing the given vectors as the rows of the matrix.

2. Functions

+ (*x* :: **<matrix>**, *y* :: **<matrix>**) => **<matrix>** [Method]
- (*x* :: **<matrix>**, *y* :: **<matrix>**) => **<matrix>** [Method]
***** (*x* :: **<matrix>**, *y* :: **<matrix>**) => **<matrix>** [Method]
***** (*x* :: **<number>**, *y* :: **<matrix>**) => **<matrix>** [Method]
***** (*x* :: **<matrix>**, *y* :: **<number>**) => <matrix> [Method]
Basic arithmetic operations. The * method on two matrices uses the algorithm from Sedgewick [1].

identity-matrix (#*key dimensions* :: **<sequence>**) => **<matrix>** [Function]
This returns a matrix filled with zeros, and having ones down the major diagonal. The following is an example:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

augment-matrix (*x* :: **<matrix>**, *y* :: **<matrix>**) => **<matrix>** [Function]
This takes two matrices and returns a matrix that places the two arguments side by side in a single matrix. The following is an example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \text{ augmented with } \begin{bmatrix} 9 & 8 \\ 7 & 6 \\ 5 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 2 & 9 & 8 \\ 3 & 4 & 7 & 6 \\ 5 & 6 & 5 & 4 \end{bmatrix}$$

gauss-jordan (*x* :: **<matrix>**) => **<matrix>** [Function]
This will take a matrix of dimension N by N+1, and return an N by 1 matrix. The result matrix is the solution to the system of equations represented by the argument matrix. For example, if your equations were

$$\begin{aligned} (3x) + (4y) &= 10 \\ (2x) - (3y) &= 1 \end{aligned}$$

then the argument matrix would be

$$\begin{bmatrix} 3 & 4 & 10 \\ 2 & -3 & 1 \end{bmatrix}$$

and gauss-jordan would return

$$\begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

representing x = 2 and y = 1. The algorithm for Gauss-Jordan was taken from Sedgewick [1].

inverse (*x* :: **<matrix>**) => **<matrix>** [Function]
Returns the inverse of the given matrix. For inverse to work, the matrix must be square, and this function signals an error if the argument matrix is not square. This function uses a modified Gauss-Jordon algorithm from Sedgewick [1] with quick error checking from Strang [2].

det (*x* :: **<matrix>**) => **<matrix>** [Function]
Returns the determinant of the given matrix. This function uses an algorithm from Strang [2].

transpose (*x* :: **<matrix>**) => **<matrix>** [Function]
Transposes the given matrix. Transpose takes each element, [i,j], and effectively stores it in location [j,i].

3. Future Revisions

Write functions for row-operations, eigen-value, and eigen-vector.

4. References

[1]: Sedgwick, Robert. Algorithms. 1993. Addison-Wesley Publishing Co.

[2]: Strang, Gilbert. Introduction to Linear Algebra. 1993. Wellesley-Cambridge Press.

The Melange Interface Generator

Robert Stockton

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

30 May 1997

Abstract

The Melange interface generator provides a mechanism for providing access to native C code. It is modeled upon Apple Computer's Creole, and shares Creole's goals of automatically providing full support for a foreign interface based upon existing interface descriptions. It also, like Creole, provides mechanisms for explicitly adapting these interfaces to provide a greater match between C and Dylan data models.

Melange, however, differs from Creole in a number of significant ways. This document, therefore, provides a gentle introduction to Melange without attempting to build upon any existing descriptions of Creole.

1. Introduction

Melange is an automatic interface generator which provides transparent access to both functions and data defined or generated by existing C libraries. It allows users to import "interfaces"[1] from existing C header files, controlled by the contents of a "define interface" top-level form which may be included in the same file as arbitrary Dylan code. The user may use the functions and data specified by this interface as if they were native Dylan objects, and may export them to other modules.

Melange provides reasonable interpretations for the various sorts of C declarations which may appear in a header file, as well as mechanisms for explicitly modifying the default interpretations when necessary. For example, users may:

- specify rules for the translation of foreign names
- explicitly specify new names for specific objects or routines
- specify parameter passing conventions or mutability of foreign objects
- specify mappings or equivalences between "foreign" data and native equivalents
- choose to import only a subset of the declarations in the header file

All of these customizations, as well as the name of the C header file, are specified by a "define interface" clause. See the next section for an example.

The basic model for interface importation is based upon that used within Apple Computer's "Creole" interface generator. There are, however, significant differences in some of the details. (For instance, the "equate", "map", and "object-file" directives used in the above example are unique to Melange. Likewise, Creole's "type" directive would

1. In fact, a C header file may contain arbitrary C code which Melange is unprepared to handle. By convention, however, ".h" files contain only "interface declarations" -- type declarations, function prototypes, global variable declarations, and "preprocessor constants". Since Melange can meaningfully process all of these, it is capable of handling the vast majority of header files which will be encountered in practice.

not be accepted by Melange) You should, therefore, not expect Creole interface declarations to work within Melange without some modification.

2. A Concrete Example

In order to get a feel for using Melange, it is probably best to start with a concrete example. This section contains a complete program which will use native C libraries to list the contents of some directories.For now, you should simply skim this example to get a general overview of Melange's capabilities. These will be described in more detail in later sections.

We will first begin with an "interface file" which contains a mixture of basic Dylan code and "define interface" forms which will be processed by Melange. We will name this file "dirent.intr".

```
module: Junk
synopsis: A poor imitation of “ls”

define library junk
  use dylan;
  use streams;
end library junk;

define module junk
  use dylan;
  use extensions;
  use extern;
  use streams;
  use standard-io;
end module junk;

define interface
  // This clause is more complex than it needs to be, but it does
  // demonstrate a lot of Melange’s features.
  #include “/usr/include/sys/dirent.h”,
    mindy-include-file: “dirent.inc”,
    equate: {“char /* Any C declaration is legal */ *” => <c-string>},
    map: {“char *” => <byte-string>},
    // The two functions require callbacks, which we don’t support.
    exclude: {“scandir”, “alphasort”, “struct _dirdesc”},
    seal-functions: open,
    read-only: #t,
    name-mapper: minimal-name-mapping;
  function “opendir”, map-argument: {#x1 => <string>};
  function “telldir” => tell, map-result: <integer>;
  struct “struct dirent”,
    prefix: “dt-”,
    exclude: {“d_namlen”, “d_reclen”};
end interface;

define method main (program, #rest args)
  for (arg in args)
    let dir = opendir(arg);
    for (entry = readdir(dir) then readdir(dir),
         until entry = null-pointer)
      write-line(entry.dt-d-name, *standard-output*);
    end for;
    closedir(dir);
  end for;
end method main;
```

We will then process this file through Melange to produce a file of pure Dylan code. If Melange is contained in the file “`melange.dbc`”, we would use the following command line:

```
mindy -f melange.dbc dirent.intr dirent.dylan
```

This command will process “melange.intr” and write a file named “dirent.dylan”. In this case, it will also silently write a file named “dirent.inc”, whose use will be explained later.

You can compile “dirent.dylan” normally, via mindycomp, but in order to execute it, you must make sure that the Mindy interpreter will be able to load the appropriate routines from the library containing the “dirent” routines. Ideally, you would simply let Mindy load the appropriate code dynamically. However, this is presently only available for a few machines. Therefore, we will follow a messier approach and build a new version of the interpreter which is aware of the desired functions.

Move to the build directory for the Mindy interpreter, and edit the Makefile so the “EXTERN-INCLUDES” line mentions “your_dir_path/dirent.inc” and then run “make mindy”. In this case, this is all that is required to build a new interpreter which is aware of the dirent routines.

You can now put it all together, invoking the new interpreter on the compiled program, with:

```
mindy -f dirent.dbc .
```

This should print a list of all files in the current directory.

Because of the difficulty of relinking the interpreter for each new library, it is expected that administrators will build a set of “standard” library interfaces which are prelinked into the interpreter and exported as general Dylan library interfaces. In the future, as Melange (and the Gwydion environment) are extended to support better linking and loading capabilities, it should become easier to incorporate C libraries on an “as-needed” basis.

3. Basic Use

Although the “define interface” form provides a fairly rich sublanguage for specifying interfaces, it is often sufficient to use just the “minimal” form. For example, if “gc.h” contained the following code:

```
typedef char bool;
typedef struct obj obj_t;
typedef char *str;

extern obj_t alloc(obj_t class, int bytes);
extern void scavenge(obj_t *addr);
extern obj_t transport(obj_t obj, int bytes);
extern void shrink(obj_t obj, int bytes);

extern void collect_garbage(void);

extern bool TimeToGC;

#define ForwardingMarker ((obj_t)(0xDEADBEEF))
```

then you could import it by creating a file named “class.intr” which includes arbitrary Dylan code and the following:

```
define interface
   #include “gc.h”;
end interface;
```

You would then run “`melange class.intr`”[1], which would produce a file of Dylan code which contains appropriate definitions for the classes “`<bool>`”, “`<obj>`”, “`<obj_t>`”, and “`<str>`”; the variable “`TimeToGC`”; and the functions “`alloc`”, “`scavenge`”, “`transport`”, “`shrink`”, and “`collect_garbage`”. (The constant

1. Or possibly “`mindy -f melange.dbc class.intr`”, depending upon the installation on your particular machine.

"ForwardingMarker" will be excluded because it is not a simple literal.)

If you were running under the HPUX operating system and the named functions were already linked into Mindy, then this might be all that you would need. After compiling the resulting Dylan file, you could call the functions as if they had been written in native Dylan code, and you can access the "TimeToGC" variable by calling the function with that name. For example you might run:

```
if (TimeToGC() ~= 0)
   collect_garbage();
end if;
```

This code fragment points out some of the hazards of "simple" imports. Melange has no way of knowing that "bool" should correspond to Mindy's <boolean> class, so you are stuck with a simple integer. Likewise, the system wouldn't be able to guess that "char *" shoudl correspond to the Mindy class "<c-string>". We will explain in later sections how "map:" or "equate:" options may be used to provide this information to Melange.

3.1. Loading and Finding Objects

As mentioned above, the include directive in the previous example will only work for files which have been previously linked into Mindy. There are extra facilities available to handle other situations.

If your machine is one for which we support dynamic loading[1], and you wish to load some declared objects from a shared library, you can add one or more "object-file:" options to the "#include" clause, as in the following:

```
define interface
   #include "gc.h",
      object-file: "/usr/lib/mindy/gc.sl";
end interface;
```

This would cause the code from "gc.sl" to be loaded into Mindy at run-time and make its functions and objects available just as they were in the previous example.

If you are running on a non-HPUX machine, you will have to statically link Mindy with the appropriate library and a list of mappings from names to addresses. This can be accomplished most easily by following these steps:

1. Add a "mindy-include-file:" option to your interface definition. This specifies the name of an "interface description file" which will be written by Melange, and which can later be linked into Mindy along with the appropriate library.
2. Run Melange on the source file in the normal manner. You may wish to move the newly created interface description file into your Mindy build directory.
3. Change the Makefile in the Mindy build directory, by adding the imported library to LIBS and the interface description file to EXTERN-INCLUDES.
4. Run "make" to rebuild Mindy with the new library information.
5. Compile and run the generated Dylan code as normal.

A typical interface definition for this approach might be:

1. Currently support is primarily for HPUX machines, but some work has been done on Macintoshes and ELF systems. Contact us for more details.

```
define interface
   #include “gc.h”,
      mindy-include-file: “/usr/local/mindy-build/gc.inc”
end interface;
```

4. Importing Header Files

You import C definitions into Dylan by specifying one or more header files in an “#include” clause. This may take one of two different forms:

```
define interface
   #include “file1.h”;
end interface;
```

or

```
define interaface
   #include {“file1.h”, “file2.h”};
end interface;
```

Melange will parse all of the named files in the specified order, and produce Dylan equivalents for (i.e. “import”) some fraction of declarations in these files. By default, Melange will import all of the declarations from the named files, and any declarations in recursively included files (i.e. those specified via “`#include`” directives in the “`.h`” file) which are referenced by imported definitions. It will not, however, import every declaration in recursively included files. This insures that you will see a complete, usable, set of declarations without having to closely control the importation process. If you wish to exert more control over the set of objects to be imported, you may do so via the “`import`”, “`exclude`”, and “`exclude-file`” options..

If you only need a small set of definitions from a set of imported files, you can explicitly specify the complete list of declarations to be imported by using the “`import:`” option. You could, for example, say:

```
define interface
   #include “gc.h”,
      import: {“scavenge”, “transport” => move};
end interface;
```

This would result in Dylan definitions for “`scavenge`”, “`move`”, “`<obj_t>`”, and “`<obj>`”. The latter types would be dragged in because they are referenced by the two imported functions. Again, if you equated “`obj_t`” to `<object>` then neither of the types would be imported. The second import in the above example performs a renaming at the same time as it specifies the object to be imported. Other forms specify global behaviors. “`Import: all`” willcause Melange to import every “top level” definition which is not explicitly excluded. “`Import: all-recursive`” causes it to import definitions from recursively included files as well. “`Import: none`” restricts importation to those declarations which are explicitly imported or are used by an imported declaration.

You may also use the “`import:`” option to specify importation behavior on a per-file basis. The options

```
import: “file.h” => {“import1”, ...}
import: “file.h” => all
import: “file.h” => none
```

work like the options described above, except that they only apply to the symbols in a single imported file.

The “exclude:” and “exclude-file:” options may be used to keep one or more unwanted definitions from being imported. For example, you could use:

```
define interface
   #include “gc.h”,
      exclude: {“scavenge”, “transport”},
      exclude-file: “gc1.h”;
end interface;
```

This would prevent the two named functions and everything in the named file from being imported, while still including all of the other definitions from “gc.h”. Note that these options should be used with care, as they can easily result in “incomplete” interfaces in which some declarations refer to types which are not defined. This could result in errors in the generated Dylan code. (The “import: file => none” option described above is a safer way of achieving an effect similar to “exclude-file:”

You may also prevent some type declarations from being imported by using the “equate:” option (described in a later section). If, for example, you equated “obj_t” to <object>, then Melange would ignore the definition for “obj_t” and simply assume that the existing definition for <object> was sufficient.

You may have any number of “import:”, “exclude:”, and “exclude-file:” options, and may name the same declarations in multiple clauses. “Exclude:” options take priority over “import:”s. If no “import:” options are specified, the system will import all non-excluded symbols, just as if you had said “import: all”.

5. Specifying Object Names

Because naming conventions differ between C and Dylan, Melange attempts to translate the names specified in C declarations into a form more appropriate to Dylan. This involves

- Adding angle brackets around type names.
- Adding dollar signs at the beginning of constant names.
- Translating (non-initial) underlines into hyphens.
- Adding “struct-name$” prefixes to slot accessors.

In many cases, this default behavior will be precisely what you want. However, Melange provides mechanisms for specifying different translations for some or all of the declarations.

5.1. Mapping functions

The translations described above are provided by calls to a built-in “name mapping function” named “minimal-name-mapping-with-structure-prefix”. You may specify other mapping functions via a “name-mapper:” option. Our example interface might then look like this:

```
define interface
   #include “gc.h”,
      object-file: “/usr/lib/mindy/gc.o”,
      name-mapper: c-to-dylan;
end interface;
```

Table 1 describes the four standard mapping functions that are provided by Melange.

Table 1:

minimal-name-mapping-with-structure-prefix	Provides the translations described above.
minimal-name-mapping	Same as above, but excludes the “struct-name$” prefixes.
c-to-dylan	Like miinimal-name-mapping, but: Adds hyphens to reinforce “CaseBased” word separation. Adds “get-” prefixes to slot accessors.
identity-name-mapping	Does no translation.

Users may link new mapping functions into Melange. In the Mindy implementation, this is done as follows:

- Create a new module which imports module “`name-mappers`” from library “`c-parse`”.
- Define methods on the “`map-name`” generic function which accepts the following parameters:
  - **mapper** -- a `<symbol>` which is typically specialized by a singleton to select a specific name mapper method.
  - **category** -- a `<symbol>` which will always be one of: `#”type”`, `#“constant”`, `#”variable”`, or `#”function”`.
  - **prefix** -- a `<string>` which is typically prepended to the result string.
  - **name** -- a `<string>` which supplies the original C name.
  - **sequence-of-classes** -- a sequence of simple names for the classes which logically “contain” the given object. For example, if we were processing the declaration “`struct str {int size; char *chars`”, one of the calls to the mapping function would have with `name`-bound to “`size`” and `classes` bound to `#[“str”].`

  and returns a `<string>` which will be used as the Dylan name for the declaration.
- Compile this module and “link” it into Melange by concatenating it to the end of the `melange.dbc`.

Mapping functions may call “`hyphenate-case-breaks`” which performs the same “CaseBased separation” as is done by “`c-to-dylan`”. The trivial “`identity-name-mapping`” described above might be implemented by:

```
define method map-name
   (mapper == #”identity-name-mapping”, category, prefix, name, classes)
=> (result :: <string>)
   name;
end method map-name;
```

You may specify different name mappers to be applied to the slots of “container types”. This capability is described in a later section.

5.2. Prefixes

As noted above, the name mapping function is passed a “prefix” argument. By default, it is an empty string, but users may specify a different value by adding a “`prefix:`” option to the interface definition. For example, we might expand the previous example to:

```
define interface
   #include “gc.h”,
      object-file: “/usr/lib/mindy/gc.o”,
      name-mapper: c-to-dylan,
      prefix: “gc-”;
end interface;
```

This would cause Melange to tack “`gc-`” onto the beginning of every translated symbol. Because the system knows about the “standard” Dylan naming conventions, it can do this intelligently. You would, therefore, get names like “`<gc-bool>`”, “`gc-time-to-gc`”, and “`gc-scavenge`”.

Note that the interpretation of the “prefix” is entirely up to the name mapping routine. `Identity-name-mapping`, for example, completely ignores the prefix. All of the other standard mapping functions prepend it to the name before adding brackets or dollar signs, but after performing all other transformations.

Facilities for adding “localized” prefixes to slot accessors, enumeration literals, etc. will be described in later sections.

5.3. Explicit Renaming

Although the automatic name mapping described above is sufficient for most objects named within a header file, there are cases in which you might wish to explicitly control the name of one or more specific objects. You can do this through a “`rename:`” option. This options specifies a list of translations between raw C names and Dylan identifiers. For example, we might have:

```
define interface
   #include “gc.h”,
      object-file: “/usr/lib/mindy/gc.o”,
      name-mapper: c-to-dylan,
      prefix: “gc-”
      rename: {“struct obj” => <C-Object>, “collect_garbage” => GC};
end interface;
```

Note that the “target” of the renaming is an ordinary Dylan variable and is therefore case-insensitive. However, the source is an “alien name”, which is (like all C code) case sensitive. Alien names should refer to an object, function, or type in exactly the same way you would refer to them in C. We therefore say “`struct obj`” instead of simply “`obj`”, and might also say “`enum foo`” or “`union bar`”. Alien names are actually parsed according to the standard lexical conventions of C, so you may use arbitrary spacing and even include comments if you really wish.

Note that “rename:” options supply names for new objects (and types) that are being imported into Dylan. You cannot, therefore, simply rename “`bool`” to “`<Boolean>`” to make it equivalent to the existing type -- this would simply result in a name conflict. For these purposes, you would instead use the “equate” and “map” operations, which will be described later. (In fact, if the C declaration had defined a type name “`boolean`”, you might have to explicitly rename it to something else in order to avoid name conflicts with the existing type. Of course, in the above example, the “`gc-`” prefix would be sufficient to make the name unique.)

5.4. Anonymous types

The alien names described above can also be used to refer to C’s so-called “anonymous types”. You can therefore refer to “`char *`”, “`int [23]`”, or even “`int (*) (char *foo)`” (i.e. a pointer to function which takes a string and returns an integer)[1]. The ability to refer to anonymous types is useful because it allows you to use the

"rename" option to provide explicit names for such types. Normally Melange would simply generate a an arbitrary "anonymous" identifier for the type. Without knowing the name of this type, you could not define new operations upon it. However, by saying, for example, "`rename: {"char * => <char-ptr>"}`", you can provide a convenient handle to use in defining new operations.

6. Type Definitions

When Melange encounters a "type definition"[1] within a header file, it will typically create a new Dylan class which corresponds to that C type. Usually, this will be a subclass of `<statically-typed-pointer>`, which encapsulates the raw C pointer value (i.e an object address). Each statically typed pointer class will have exactly the same structure (i.e. a single address), but the class itself can be used to determine what operations are supported on the data. This could include slot accessors for "`struct`"s and "`union`"s, dereference operations for "pointer" types, or general information about the object's size, etc.

There are times when you will find that some of the types defined in a header file are not really "new". It might be that they are completely identical to some type defined in another interface definition, or they might be "isomorphic" to some existing type which has more complete support. Melange provides support for both of these cases. The first case is handled by "equating" the two types, while the second is handled by "mapping" (i.e. transforming) one type into the other.

For example, many header files contain definitions use the types "`char *`" and "`boolean`". The declarations of these types don't provide any semantic interpretations -- "`char *`" is simply the address of a character, and boolean is nothing but a one-byte integer. However, by equating "`char *`" to the predefined `<c-string>` type, we can tell Melange that it is actually a `<string>` and should inherit all of the operations defined upon `<string>`s. Likewise, we can map the integral "`boolean`" values into "`#t`" and "`#f`" to get a `<boolean>`. These integral values will be automatically translated into `<boolean>`s when they are returned by a C function, and `<boolean>`s will be translated back into integers when passed as arguments to C functions.

6.1. Implicit class definitions

Unless otherwise specified, new classes will be created for each type defined in a C header file. When the header file provides meaningful names for these types, then Melange will pass those names to the mapping functions to generate names for the Dylan classes. Otherwise, an anonymous name will be generated, limiting your ability to refer to the new type. For example, "`struct foo`" would typically generate the class "`<foo>`", while "`struct foo ***`" might generate the class "`<anonymous-107>`". In either case, you can explicitly specify the name for the new class by using the "`rename:`" option described above.

Different sorts of C declarations will yield different sorts of Dylan classes as well as different sets of operations defined upon them. Therefore, we will consider each variety separately:

- **Primitive types** -- The types "`int`", "`char`", "`long`", "`short`" and their unsigned counterparts are simply translated into `<integer>`, while "`float`" and "`double`" are translated into `<float>`. However, Melange knows the sizes of each of these types so that pointers and native C "vectors" of them (described below) will work properly. No new types are created for these types.

- **Pointer types**-- Declarations like "`int *`" or "`struct foo ***`" generate new subclasses of "`<statically-typed-pointer>`". Note that "`struct foo *`" is actually treated as a synonym for "`struct`

1. At present, function types are not fully supported. You should not depend upon them to work as expected.

1. The definition may be implicit, as in "char ** int" or "struct foo *bar". Simply by being present these code fragments supply implicit definitions for "char *", "char **" and "struct foo".

`foo`", and does not get a distinct class, although any extra levels of indirection (i.e. "`struct foo **`") will generate new pointer classes. Three operations are supported upon pointer classes:

`pointer-value (pointer, #key index) => (value)`
: This function "dereferences" the pointer and returns the value. If index is supplied, then "`pointer`" is treated as a vector of values and the appropriate element is returned.

`content-size (cls) => integer`
: Returns the size of the value referenced by instances of "`cls`". If the size is not known, this is `0`.

Note that these types are *not* automatically treated as vectors. You may, however, make them so by using a "`superclasses:`" option to make them `<c-vector>`s.

- **Vector types** -- Declarations like "`char [256]`" are treated almost identically to pointer types, but they are automatically defined as subclasses of `<c-vector>`, so that all vector operations will be defined on them. However, because many systems depend upon the lack of bounds checking in C, vector types have a default size of "`#f`". You may explicitly define "`size`" functions to provide a more accurate size.

- **Structure types** -- Declarations like "`struct bar {int a; char *b;}`" also generate new subclasses of "`<statically-typed-pointer>`". Melange will define all of the operations defined for pointer values (described above), as well as accessors for each of the structure slots. Structure objects are always accessed through "pointers" to them. Therefore, unless a non-zero index is specified, "`pointer-value`" will simply return the object passed to it. (The operation is still defined because non-zero indices can be used for vector access.)

- **Union types** -- Declarations like "`union bar {int a, char *b;}`" are treated the same as struct declarations, except that the slot accessors all refer to the same areas in memory.

- **Enumeration types** -- Declarations like "`enum foo {one, two, three};`" are simply aliased to `<integer>`. However, constants are defined for each of the enumeration literals.

- **Typedefs** -- Declarations like "`typedef struct foo bar`" simply define new names for existing types.

6.2. Specifying class inheritance

When Melange creates new "`<statically-typed-pointer>`" classes, it typically creates them as simple subclasses of "`<statically-typed-pointer>`", with no other superclasses. However, you might sometimes need more control over the class hierarchy. For example, you might wish to specify that a C type should be considered a subtype of the abstract class "`<sequence>`". You could accomplish this via the following declarations:

```
define interface
   #include "sequence.h";
   struct "struct cons_cell" => <c-list>,
      superclasses: {<sequence>};
   function "c_list_size" => size;
end interface;

define method forward-iteration-protocol (seq :: <c-list>)
....
```

Note that the type "`<c-list>`" will still be a subclass of "`<statically-typed-pointer>`" -- we have simply added "`<sequence>`" to the list of superclasses. If "`<statically-typed-pointer>`" is not explicitly

included in the "superclasses:" option, then it will be added at the end of the superclass list.

As demonstrated in the above example, you are still responsible for specifying whatever functions are required to satisfy the contract for the declared superclasses. "<C-list>" will be declared as a sequence, but you must specify a forward iteration protocol before any of the standard sequence operations will work properly.

The "superclasses:" option may currently be used within "struct", "union", and "pointer" clauses.

7. Translating Object Representations

Whenever a native C object is returned from a function or a Dylan object is passed into a C function, it is necessary to translate between the object representations used by the two languages. From Melange's standpoint, native C objects consist of an arbitrary bit pattern which can be translated to or from a small number of "low level" Dylan types -- namely <integer>, <float>, or any subclass of <statically-typed-pointer>. This translation is handled automatically, although the user may explicitly specify which of the possible Dylan types should be chosen for any given C object type. In some cases, a further translation may take place, converting the "low level" Dylan value to or from some arbitrary "high level" Dylan type. (For example, an <integer> might be translated into a <boolean> or a <character>, and a <c-string> might be translated into a <byte-string>.) These "high level" translations are automatically invoked at the appropriate times, but both the "target" types and the methods for performing the translation must be specified by the user.

7.1. Specifying low level transformations

The target Dylan type for "low level" translations is typically chosen automatically by Melange. Integer and enumeration types are translated into <integer>; floating point types are translated to <float>; and all other types are translated into newly created subclasses of <statically-typed-pointer>. However, you may explicitly declare the target Dylan type for any C type by means of an "equate:" option:

```
define interface
   #include "gc.h",
      equate: {"char *" => <c-string>};
end interface;
```

This declaration makes the very strong statement that any values declared in C as "char *" are identical in form to the predefined type "<c-string>" (which is described in Appendix I). The system will therefore not define a distinct type for "char *" and will ignore any structural information provided in the header file. You migh also use an "equate:" option to equate a type mentioned in one interface definition with an identically named type which was defined in an earlier interface definition.

You should use caution when equating two types. Since Melange has no way of knowing when two types are equivalent, it must trust your declarations. No type checking can or will be done, so if you incorrectly equate two types, the results will be unpredictable. In some cases, you may wish to go with the less efficient but slightly safer technique of letting Melange create a new type and then "mapping" that new type into the desired type. (This is described in detail below.)

Note also that two types with identical purposes will not necessarily have identical representations. For example, C's boolean types are simple integers and are not equivalent to Dylan's <boolean>. Again, explicit "mapping" may be used to transform between these two representations.

In the current implementation, an "equate:" option only applies within a single interface definition. Other interface definitions will not automatically inherit the effects of the declaration. In future versions, we may add the ability to

“use” other interface definitions (just as you would “use” another module withing a module definition) and thus pick up the effects of the “equate: (and “map:”) options within those interfaces.

7.2. Specifying high level transformations

Sometimes you may wish to use instances of some C type as if they were instances of some existing Dylan class, even though they have different representations. In this case, you can specify a secondary translation phase which semi-automatically translates between a “low level” and a “high level” Dylan representation. In order to do this, you must provide a “map:” option:

```
define interface
   #include “gc.h”,
      equate: {“char *” => <c-string>},
      map: {“bool” => <boolean>};
end interface;
```

This clause will cause any functions defined within the interface to call transformation functions wherever the original C functions accept or return values of type “bool”. Two different functions may be called:

- `import-value (high-level-class :: <class>, low-level-value :: <object>)`
  This function is called to transform result values returned by C functions into a “high level” Dylan class. It should always return an instance of “high-level-class”.
- `export-value (low-level-class :: <class>, high-level-value :: <object>)`
  This function is called to transform “high level” argument values passed to C functions into the “low level” representations which will be meaningful to native C code. It should always return an instance of “low-level-class”.

Default methods, which simply call “as”, are provided for each of these functions. This will be sufficient to transform C’s integral “char”s into <character>s, <c-string>s into other <string>s, or one “pointer” type into another. There is also a predefined method which will transform <integer>s into <boolean>s. However, if you wish to perform arbitrary transformations upon the values, you may need to define additional methods for either or both of these functions. For example, the default methods for transforming to and from <boolean> are:

```
define method export-value (cls == <integer>, value :: <boolean>)
 => (result :: <integer>);
   if (value) 1 else 0 end if;
end method export-value;

define method import-value (cls == <boolean>, value :: <integer>)
 => (result :: <boolean>);
   value ~= 0;
end method import-value;
```

It is important to note that, unlike “equate:” options, “map:” options don’t prevent Melange from creating new types. You may, in fact, both equate and map the same type. This will cause low level values to be created as instances of the “equated” type and then transformed into instances of the “target” type of the mapping. For example, you might take advantage of the defined transformations between string types by declaring:

```
define interface
   #include “/usr/include/sys/dirent.h”,
      equate: {“char *” => <c-string>},
      map: {“char *” => <byte-string>};
end interface;
```

This causes the system to automatically translate "`char *`" pointers into `<c-string>`s (i.e. a particular variety of statically typed pointer) and then to call "`import-value`" ot translate the `<c-string>` into a `<byte-string>`. If we did not provide the "`equate:`" option, then we would have to explicitly provide a function to transform "pointers to characters" into `<byte-string>`s. The "`equate:`" option lets us take advantage of all of the predefined functions for `<string>`s, which includes transformation into other string types.

8. Other File Options

There are a few other options that may be specified within an "`#include`" clause, but which do not fit into any of the above categories. These options are "`define:`", "`undefine:`", "`seal-functions:`" and "`read-only:`".

The "`define:`" and "`undefine:`" options control the C preprocessor definitioins which will be implicitly defined during parsing of the header files. If you specify neither of these options, Melange will use a default set of definitions which correspond to those used by a typical C compiler for the machine you are running on.[1] The define options takes a string containing a single C token and an optional string or integer literal, which will be used as the expansion. (If no literal is specified, the token will be expanded to "`1`".) The "`undefine:`" removes one or more of the default definitions. You might, for example, use:

```
define interface
   #include "gc.h",
      define: {"PMAX", "BSD_VERSION" => "4.3"},
      undefine: {"HPUX"};
end interface;
```

The "`seal-functions:`" option controls whether the various imported functions and slot accessors will be sealed or open. By default, functions are sealed, but you may explicitly specify this by using "`seal-functions: sealed`" or reverse it by using "`seal-functions: open`". Melange does not support the Creole's "`inline`" sealing option.

The "`read-only:`" option specifies whether setter functions should be defined for slot and object accessors. They will be defined by default, but if you specify "`read-only: #t`", no setters will be defined.

The effects of the "`seal-functions:`" and "`read-only:`" options can be modified for particular container types. We will explain how to do this in a later sections.

9. Function Clauses

Imported functions can be easily invoked, in almost every case, without any additional declarations. However, by exerting explicit control over argument handling, the interfaces to some functions may be made cleaner. This control is exerted via function clauses. The primary purpose of these clauses is to specify additional type information for specific parameters or to specify alternative argument passing conventions. For example, if we had two alternate "read-integers" functions with the following declarations:

```
int ReadInts1(int **VectorPtr); /* result is a count of integers */
int *ReadInts2(int *Count); /* result is a vector of integers */
```

we might use the following interface definition:

1. At present, the only set of definitions provided will be those appropriate for the HPUX OS. However, it is straightforward to add different sets of definitions to Melange.

```
define interface
   #include “readints.h”,
      rename: {“int *” => int-vector};
   function “ReadInts1”
      output-argument: 1;
   function “ReadInts2” => Read-Integers-Vector,
   output-argument: Count;
end interface;
```

This would produce two functions, both of which take 0 arguments but return two values. The first would return an `<integer>` following by an “`<int-vector>`”, while the second would return the `<int-vector>` first and the `<integer>` second.

```
let (count :: <integer>, values :: <int-vector>)
   = Read-Ints1();
let (values :: <int-vector>, count :: <integer>)
   = Read-Integers-Vector();
```

The function clause consists of a function name (which is a string), an optional renaming (as illustrated above), and an optional sequence of “options”. The options include the following:

- **seal:** -- specifies whether the resulting method should be sealed. Possible values are `sealed` or `open`, and the default is taken from the value specified in the initial file clause. (The “default default” is sealed.)
- **equate-result:** -- overrides the default interpretation of the result type. The named type is assumed to be fully defined.
- **map-result:** -- specifies that “import-value” should be called to map the result value to the named type.
- **ignore-result:** -- specifies that the functions result value should be ignored, just as if the function had been declared “void”. Although you may specify any boolean literal, the only meaningful value is #t.
- **equate-argument:** -- overrides the default interpretation of some argument’s type. The argument may be specified by name or by position.
- **map-argument:** -- specifies that “export-value” should be called to map the given argument into the named type. Again, the argument may be specified by position or by name.
- **input-argument:** -- indicates that the specified argument should be passed by value. This is the default.
- **output-argument:** -- indicates that the specified argument should be be treated as a return value rather than a “parameter”. The effect is to declare that the C parameter will be passed by reference and that the reference variable need not be initialized to any object. This option assumes that the C parameter will have been declared as a “pointer” type, and will strip one “*” off of the argument type. Thus, if the parameter declaration specifies “`int **`”, the actual value returned will have the Dylan type corresponding to “`int *`”.
- **input-output-argument:** -- indicates that the specified argument should be considered both an input argument and that its (potentially modified) value should be returned as an additional result value. The effect is similar to that of “output-argument” except that the reference variable will be initialized with the argument value.

The following (nonsensical) example demonstrates all of the options, as they might be applied to the functions:

```
extern struct object *bar(int first, int *second,struct object **third);
extern baz(char first, struct object *second);
```

```
define interface
   #include "demo.h";
   function "bar",
      seal: open,
      equate-result: <object>,
      map-result: <bar-object>,
      input-argument: first,// passed normally
      output-argument: 2,  // nothing passed in, second result value
                           // will be <integer>
      input-output-argument: third;// passed in as second argument,
                           // returned as third result
   function "baz" => arbitrary-function-name,
      seal: sealed,        // default
      ignore-result: #t,
      equate-argument: {second => <object>},
      map-argument: {2 => <baz-object>};
end interface;
```

10. Struct and Union Clauses

"Struct clauses" and "union clauses" (referred to collectively as "container clauses") are used to specify naming in inclusion of class slots in exactly the same way that the options in the file clause control the handling of global definitions. Like the function clauses described above, they consist of the reserved word "struct" or "union", a string which gives the full C name of the container declaration, an optional renaming, and a list of options. If we have a structure defined by

```
typedef struct cons {
   int index;
   struct object *head;
   struct cons *tail;
} cons_cell;
```

we could use the following interface definition:

```
define interface
   #include "cons.h";
   struct "struct cons" => <c-list>,
      superclasses: {<sequence>},
      prefix: "c-list-",
      name-mapper: identity-name-mapping,
      exclude: {"index"};
end interface;
```

Valid options for container clauses include: `import:`, `prefix:`, `exclude:`, `rename:`, `seal-functions:`, `read-only:`, `equate:`, and `map:`. These options act like the equivalent options which may be specified in a file clause, but they apply to the slots of a single "class" rather than to globally defined objects. Options specified within a container clause override any global defaults that might have been specified in the "`#include`" clause.

Container clauses also permit the "superclasses:" option described in section 6.2.

Although the recommended method for specifying a container type is to use the full C name (i.e. "`struct foo`"), you may also use an alias defined by a typedef. Thus, in the above example, you could have specified either

"`struct cons`" or "`cons_cell`", with identical results.

11. Pointer Clauses

"Pointer clauses" modify the definitions of pointer declarations such as "int *" or "struct foo ***", or vector declarations such as "char [256]". Like all such clauses, they may be used to specify renamings for the classes. This is particularly useful for pointer types since they are not automatically assigned user-meaningful names. It also allows specification of the "superclasses:" option described in 6.2. A typical use might be:

```
define interface
   #include "vec.h";
   pointer "int *" => <int-vector>,
      superclasses: {<c-vector>};
   pointer "struct person **" => <people>,
      superclasses: {<c-vector>};
   pointer "char [256]" => <fixed-string>;
end interface;
```

This clause is particularly useful for declaring pointer types to be subclasses of `<c-vector>` so that they can be indexed via "`element`". (Note that this is not necessary for vector declarations, since they are automatically declared to be `<c-vectors>`.)

12. Constant Clauses

Constant clauses are used to override the values of constants specified in header files (i.e. "`#define MAXINT 27`"). The "`value:`" option, which is the only one supported, specifies a Dylan literal which will be taken as the value of the named constant. A typical use might be:

```
define interface
   #include "const.h";
   constant "MAXINT" => $maximum-fixed-integer,
      value: 9999999;
end interface;
```

13. Variable Clauses

Global variables declared within C header files are translated into "getter" functions which retrieve the value of the C variables and optional "setter" functions to modify those values. In effect, they are treated as slots of a "null object" --the getter function takes no arguments and returns the value of the variable, while the setter function takes a single value which will be the new value of the variable. Type mapping takes place for the arguments and results of these functions, just as it would for slot accessors.

Variable clauses support the following options:

- **getter:** -- specifies a Dylan variable name which will be used to hold the getter function.
- **setter:** -- specifies either a Dylan variable name which will be used to hold the setter function, or `#f` to indicate that there should be no setter function.
- **read-only:** -- specifies whether the variable should be settable. "`read-only: #t`" is equivalent to "`setter: #f`".

- **seal:** -- specifies whether the getter and setter functions should be sealed. Possible values are "`sealed`" or "`open`", and the default is taken from the "`seal-functions:`" option in the "`#include`" clause (or "`sealed`" if not specified there).
- **map:** -- specifies the high-level type to which the variable should be mapped.
- **equate:** -- specifies the low-level type to which the raw C value should be implicitly converted.

Appendix I -- Low level support facilities

The high level functions for calling C routines or for accessing global variables are all built upon a relatively small number of built-in primitives which perform specific low-level tasks. You should seldom have any need to deal with these primitives directly, but they are nonetheless available should you need to make use of them.

To use these types and operations, you should "use" the module "extern" from the "Dylan" library.

I.i. Predefined types

`<statically-typed-pointer>` [class]

Unless otherwise specified, C pointers are implicitly "equated" to newly created subclasses of `<statically-typed-pointer>`. This class is contains a single implicit slot which contains the raw C pointer. Because of implementation limitations in Mindy, you may not add any extra slots to subclasses of `<statically-typed-pointer>`, nor can such a subclass inherit slots from other classes. You may, however, create classes which are subclasses of both `<statically-typed-pointer>` and other (presumably abstract) classes which have no slots.

The "`make`" method for takes three keywords. The "`pointer:`" keyword tells it to initialize the new variable with the given value, which must be a `<statically-typed-pointer>` or an `<integer>`. If the no pointer value is specified, space will be allocated based upon the content-size of the specific type and upon the "`extra-bytes:`" and "`element-count:`" keywords. These keywords, which default to "0" and "1" respectively, tell how many objects are going to be stored in the memory and how many bytes of extra memory (beyond that specified by "`content-size`") should be allocated for each element.

`<c-vector>` [class]

`<C-vector>` is a subclass of `<statically-typed-pointer>` which inherits operations from <vector>. Because systems often depend upon C's lack of bounds checking, the default size for `<c-vector>`s is "`#f`". However, subclasses of `<c-vector>` may provide a concrete size if desired. Types corresponding to declarations such as "`char [256]`" are automatically declared as subclasses of `<c-vector>`, but pointer declarations such as "`char *`" are not.

`<c-string>` [class]

`<C-string>` is a subclass of `<statically-typed-pointer>` which also inherits operations from `<string>`. It is implemented as a C pointer to a null-terminated vector of characters, and provides a method on `forward-iteration-protocol` which understands this implementation. This class may, therefore, be used for manipulating C's native format for "string"s (i.e. "`char *`"). Note that the "null" string is considered to be a valid empty string. This is somewhat contrary to the semantics of many C operations, but provides a safer model for Dylan code.

The "make" method for `<c-string>`s accepts the "`size:`" and "`fill:`" keywords.

There are a few surprising properties of `<c-strings>` which may users should be aware of, both of which result from the "null-terminated" implementation. Firstly, the "`size`" of the string is computed by counting from the beginning of the string, and is therefore not nearly as efficient as you might expect. Secondly, you should expect odd results if you try to store "`as(<character>, 0)`" into such a string. Finally, the "`element`" and "`element-setter`" methods must scan the string in order to do bounds checking, and may therefore be fairly slow. If you wish to (unsafely) bypass this checking, you must use "`pointer-value`" instead.

`<c-function>` `[class]`

`<c-function>`s, like `<statically-typed-pointer>`s, encapsulate a raw "C" pointer. However, `<c-function>`s also encode information about the calling conventions of the function which is (presumably) located at the given address. They may, therefore, be directly invoked just like any other function.

`<foreign-file>` `[class]`

The <foreign-file> class is used to store information about the contents of a particular object file. It is created by "load-object-file", and may be passed as an option to "find-c-function" and "find-c-pointer". (All of these functions are described below.)

I.ii. Locating native C objects

There are several functions provided which search for C functions or variables and return Dylan objects which refer to them. Note that Mindy does not have sufficient information to determine whether any given C object is a function, and therefore it depends upon the user (or, more often, Melange) to provide it with correct information.

`load-object-file(files :: <list>, #key symbols)` `[function]`

This function (which is presently only works on HPUX machines) attempts to dynamically load a given object file (i.e. ".o" or ".a") into the current Mindy process and load it's symbol table to allow its contents to be located by "`find-c-pointer`" or "`find-c-function`". If it successfully loads the file, it will return a `<foreign-file>` encapsulating the symbol table information. Otherwise, it will return `#f`.

If you are not running on an HPUX machine, you will have to statically link object files into Mindy, as described in Appendix II.

`find-c-pointer(name :: string, #key file :: <foreign-file)` `[function]`

This function searches through the symbol table for the object file corresponding to the specified file (or for Mindy itself) and attempts to locate a symbol with the given name. If it finds such a symbol, it converts the corresponding address to a `<statically-typed-pointer>` and returns it. Otherwise, it returns `#f`.

```
find-c-function (name :: <string>, #key file)                  [function]
constrain-c-function (fun :: <c-function>,                     [function]
                      specializer :: <list>, rest? :: <boolean>,
                      results :: <list>)
```

The function "`find-c-function`" is like "`find-c-pointer`", except that the result is a `<c-function>` (or `#f`). The resulting function is specialized to "`fun(#rest args) :: <object>`". However, it may be constrained to a different set of specializers via "`constrain-c-function`". This function accepts lists of types for the arguments and for the return values, and a boolean value which states whether optional arguments are accepted. The result declarations are particularly important, since they are used to coerce the raw C result value into an appropriate low level Dylan type. The possible types are `<boolean>`, `<integer>`, or any subclass of `<statically-typed-pointer>`. Note that although a list of result types is accepted, only the first can be meaningful since C does not support multiple return values.

Note: The functions in this section are likely to change drastically in the near future.

I.iii. Pointer manipulation operations

Each <statically-typed-pointer> encapsulates a pointer to some area of memory (i.e. a raw machine address). In itself, this does little good, except as an arbitrary token. However, Mindy provides a number of primitive operations which manipulate the contents of these addresses, or do basic comparisons and arithmetic upon the addresses themselves.

```
signed-byte-at (ptr :: <statically-typed-pointer>, #key offset)[function]
unsigned-byte-at (ptr :: <statically-typed-pointer>, #key offset)[function]
signed-short-at (ptr :: <statically-typed-pointer>, #key offset)[function]
unsigned-short-at( ptr :: <statically-typed-pointer>, #key offset)[function]
signed-long-at (ptr :: <statically-typed-pointer>, #key offset)[function]
unsigned-long-at (ptr :: <statically-typed-pointer>, #key offset)[function]
pointer-at (ptr :: <statically-typed-pointer>,                [function]
            #key offset, class)
```

These operations return an object which represents the value stored at the address corresponding to “ptr”. The first six operations all return <integer>s -- the different versions are required because the same number may be represented in a variety of formats (differing in length and interpretation of the high-order bit) and because Mindy has no way of determining which might be used in a given situation. The final operation, “pointer-at”, returns a new <statically-typed-pointer> encapsulating the address referenced by the origninal pointer. You may use the “class:” keyword to specify that the new object should be an instance of some particular subclass of <statically-typed-pointer>. (Thus, for example “pointer-at(foo, class: <bar>)” would be roughly equivalent to “as(<bar>, pointer-at(foo))”.)

The offset parameter (if provided) is added to the integer corresponding to the machine address before the pointer is dereferenced. This is useful, for example, in loading an object from within a C “struct”.

Setter functions are provided corresponding to each of the above functions. You can therefore, say

```
signed-short-at(ptr) := 32767;
pointer-at(ptr1) := ptr2;
```

```
as(cls == <integer>, ptr :: <statically-typed-pointer>)        [G.F. Method]
as(cls == <statically-typed-pointer>, ptr :: <statically-typed-pointer>)
                                                               [G.F. Method]
as(cls == <statically-typed-pointer>, int:: <integer>)         [G.F. Method]
```

Method upon “as” are provided for converting from <integer> to any statically typed pointer class and from any statically typed pointer class to <integer> or to another statically typed pointer class.

```
\+ (ptr :: <statically-typed-pointer>, int :: <integer>)       [G.F. Method]
\- (ptr1 :: <statically-typed-pointer, ptr2 :: <statically-typed-pointer>)
                                                               [G.F. Method]
\= (ptr1 :: <statically-typed-pointer, ptr2 :: <statically-typed-pointer>)
                                                               [G.F. Method]
```

These functions do arithmetic upon the integers corresponding to the given pointers. The following code fragment

```
let new-ptr = ptr1 + 3;
let difference = ptr2 + ptr3;
let same? = (ptr2 = ptr3)
```

is equivalent to

```
let new-ptr = as(ptr1.object-class, as(<integer>, ptr1) + 3);
let difference = as(<integer>, ptr2) - as(<integer>, ptr3);
let same = (as(<integer>, ptr2) = as(<integer>, ptr3));
```

Appendix II -- Static linking mechanisms

Because object file formats vary widely by architecture, Mindy does not support dynamic loading of object files or automatic symbol table look up for all architectures. In the general case, it is necessary to depend upon a less elegant technique for explicitly making certain C objects available.

Simple instructions for using this mechanism from within Melange are given in section 3.1. This appendix simply provides more information on the underlying mechanism.

In order to make sure that the desired symbols can be located, it is necessary to build an explicit table which maps between the symbol's name and its address. This table is automatically created by running the "`make-init.pl`" script[1] upon a list of "interface definition files". This will create two files "`,extern1.def`" and "`,extern2.def`", which should then be renamed to "`extern1.def`" and "`extern2.def`" respectively. These files are automatically included by "`ext-init.c`" so that the table will be created after Mindy is rebuilt.

The interface definition files consist of zero or more lines of text, each of which should contain the name of one object. If the object is a function, it should be immediately followed by a set of parentheses. For example, the file which defines the memory allocation routines used by Melange's support code contains the following four lines:

```
free()
malloc()
strcmp()
strlen()
```

The only other step required to make the objects available is simply to ensure that the library which contains them is linked into Mindy. The easiest way to accomplish all of this is to simply modify the `Makefile` in Mindy's source directory. If you add the names of the required libraries to `LIBS` and the names of the interface definition files to `EXTERN-INCLUDES`, `make` will do the necessary work for you. You should be sure to leave "`../compat/lib-compat.a`" or "`-lm`" in `LIBS` and "`malloc.inc`" in `EXTERN-INCLUDES`.

1. This requires you to have PERL installed on your system.

Appendix III -- Differences from Creole

It would be difficult to produce an exhaustive list of the differences between Creole and Melange. We can, however, include a brief examination of the most important incompatibilities between the two systems.

1. Creole's "`type:`" options have been replaced by Melange's "`equate:`" and "`map:`" options.
2. Creole's "access path" options have been replaced by "`object-file:`" and "`mindy-include-file:`".
3. The interface to "`import-value`" and "`export-value`" differ between the two systems.
4. Melange does not inherit type mappings from other "define interface" forms.
5. Creole does not import definitions from "recursively included" header files, even if they are referenced by definitions which *are* imported.
6. Creole does not support C vectors or "sub-structures" as first class objects.
7. Melange does not presently support callbacks, "`export-temporary-value`", "`<pascal-string>`", "`with-stack-structure`", "`with-stack-block`", or "`alien-method`".
8. Creole will never consider instances of two distinct statically typed pointer classes to be "`=`", even if they refer to the same address.

Appendix IV -- Known limitations

Although mostly complete, the current implementation of Melange is missing a few elements which might be required for some applications. The following capabilites probably *should* be present, but are not yet supported:

1. Floating point numbers.
2. Callbacks.
3. Function types. (It is, however, possible to import a function as a simple `<statically-typed-pointer>` and then manipulate it like any other object.)

Appendix V -- Proposed modifications

Although Melange seems to be fairly useful in its present form, we are currently considering a number of ways in which it may be made more useful. This section contains a brief discussion of several potential changes which may be implemented in the future.

V.i. Enumeration clauses

At present, there is no way to modify the default handling of a C enumeration declaration. It is clear that you might wish a mechanism to specify several different explicit options: prefixes for the enumeration constants; respecification of constant values; and, of course, explicit "`import:`" and "`exclude:`" options.

V.ii. Inheritance of "map" and "equate" options

There are some cases in which a set of types imported within one interface definition might be used extensively within another. In the present implementation, the two interface definitions would be handled independently and equivalences between types would not be recognized in the abscence of explicit "`equate:`" options.

One proposed solution would involve the ability to explicitly "`use`" one interface definition within another. This would result in all identically named types being implicitly equated and all top-level "`map:`" options being inherited. The "`use`" clause could support roughly the same syntax as the "`use`" clauses in library and module definitions.In order to make this work, it would be necessary to assign arbitrary names to interface definitions. This would have the added benefit of making them more consistent with other standard Dylan definition forms.

If this change were implemented, a typical interface definition might look something like the following:

```
define interface date
   #include "date.h";
   use time, import: {"struct time"};
end interface date;
```

A less ambitious version might remain compatible with the current syntax by replacing the interface name with an "interface-name" option, which would default to the root of the file name. Thus,

```
define interface
   #include "date.h",
      interface-name: "date";
end interface;
```

would yield the same effect as the previous example.

V.iii. Remerging of the "equate:" and "map:" options

It has been pointed out that the current method of specifying low-level and high-level mappings, while sufficiently expressive, is somewhat verbose and confusing. It would therefore be good to find an alternative notation.

It has been suggested that definitions like:

```
define interface
   #include “dirent.h”,
      equate: {“char *” => <c-string>},
      map: {“char *” => <byte-string>};
end interface;
```

might be replaced by something like:

```
define interface
   #include “dirent.h”,
      equate-and-map: {“char *” => <c-string> => <byte-string>};
end interface;
```

or

```
define interface
   #include “dirent.h”;
   transform “char *”,
      low-level: <c-string>,
      high-level: <byte-string>;
end interface;
```

Symbols

- 152
$ 30
$$ 30
$aN 31
$double-e 146
$-N 30
$N 30
$single-e 147
%% 111, 150
%= 111
%a 150
%B 111
%b 150
%C 111
%D 111, 150
%d 150
%H 150
%I 150
%M 111, 150
%m 150
%O 111
%p 150
%r 150
%S 111, 150
%T 150
%w 150
%X 111
%Y 150
%y 150
* 152
+ 152
^ 146

A

abort 32
abs-position-point 127
abs-position-x-y 127
abstract? 13
acos 145
acosh 146
activate 118, 119
activate-entry 122
<active-variable> 118
add-canvas-tag 126
add-cascade 123
add-checkbutton 123
add-command 123
add-last 137
add-radiobutton 123
add-separator 123
add-tag 126
adjust-stream-position 78
Algorithms 153
$all-libraries 39, 41
alpha? 137
alphabetic? 137
alphanumeric? 137
anonymous-name 129
as 78, 149
asin 145
asinh 146
assert 44
atan 145
atan2 145
atanh 146
augment-matrix 152
available-visuals 127

B

background 114
backtrace 27
bind 117, 126
binding 41
binding-call-order 117
binding-call-order-setter 117
bitmap 114
borderwidth 114
Boyer-Moore 139
breakpoint 33
broadcast-event 23
<buffer> 48, 85
Buffer access protocol 83
<buffered-stream> 60
buffer-end 91
buffer-end-setter 92
<buffer-index> 85
buffer-next 91
buffer-next-setter 91
buffer-subsequence 92
<button> 118
<byte> 84

<byte-character> 44, 84
<byte-character-table> 138
BYTES_CONSED_BETWEEN_GCS 7
<byte-string-stream> 61
<byte-vector> 44, 85

C

call 30
call-tk-function 128
<canvas> 120
<canvas-item> 126
<canvas-tag> 126
canvas-x 120
canvas-y 120
<case-insensitive-character-set> 138
case-insensitive-equal 133, 137
case-insensitive-string-hash 133
<case-insensitive-string-table> 132
<case-sensitive-character-set> 138
<character-set> 138
Character-type Module 136
Cheap-io Module 18, 51
<checkbutton> 118
children 127
class-name 13, 49
clear-selection 122
close 67, 80
collect-garbage 13
Collection-extensions Library 130
collection-hash 132, 135
collections, limited 9
colon notation 28
colormap-cells 127
colormap-full? 127
color-to-r-g-b 127
color-to-rgb-color 127
command 114
Command Line Switches 7
compiled-for-hppa 43
compiled-for-hpux 43
compiled-for-linux 43
compiled-for-solaris 43
compiled-for-sparc 43
compiled-for-unix 43
compiled-for-win32 43
compiled-for-x86 43
compilers 42
Conditional Compilation 43
condition-force-output 46
condition-format 46
configuration 115, 126
configure 115, 126
configure-entry 122
contagion 145
continue 33
Control Strings 111
control? 137
copy-bytes 48
copy-from-buffer! 93
copy-into-buffer! 92
cos 145
cosh 145
create-arc 120
create-bitmap 120
create-line 120
create-oval 120
create-polygon 120
create-rectangle 120
create-text 120
create-window 120
ctype.h 136
current-selection 122
current-thread 20
cursor 114

D

daylight-savings-time? 148
<day-of-month> 148
day-of-month 148
<day-of-week> 148
day-of-week 148
deactivate 118, 119
Debugger Customizations 12, 45
Debugger Variables 30
Debugger-format 45
Debugger-format module 12
debug-output 12
<decoded-time> 148
decode-time 149
default-circle? 104
default-length 104
default-level 104

default-line-length 107
default-pretty? 104
define function 42
delete 34, 121, 122, 124
delete-canvas-tag 126
delete-item 126
delete-tag 126
denominator 48
depth 127
describe 38
deselect-value 119, 120
destroy-window 115
det 153
determinant 153
digit? 137
digit-to-integer 136
Directives 150
disable 37
disassemble 35
discard-input 69
discard-output 73
display-object-info 39
distance-to-float-pixels 127
distance-to-pixels 127
do-get-input-buffer 93
do-get-output-buffer 94
do-input-available-at-source? 94
do-next-input-buffer 94
do-next-output-buffer 95
do-release-input-buffer 94
do-release-output-buffer 95
$double-e 146
$double-pi 145
down 26
Dylan vs. Mindy 8, 42
Dylan/TK 39, 112
Dylan/Tk Extension Protocol 128
Dylan/Tk standard options 114

E

e 146
EACCES 18
EBADF 18
EEXIST 18
EFAULT 18
EFBIG 18
EINTR 18
EINVAL 18
EIO 18
EISDIR 18
EMFILE 18
enable 37
ENAMETOOLONG 18
encodable-time? 149
encode-time 149
<end-of-stream-error> 95
ENFILE 18
ENOENT 18
ENOSPC 18
ENOTDIR 18
<entry> 121
entry-configuration 122
Environment Variables 7
environment variables 48
ENXIO 18
EPIPE 18
equal-hash 135
<equal-table> 134
EROFS 18
errno.h 18
error 38
Evaluating Expressions 29
<event> 22
Examining Variables 27
exists? 127
exit 41, 46
exp 146
Exponentiation 146
<extended-integer> 47
Extensions Module 11, 44

F

false-or 45
fd-close 16
fd-error-string 18
fd-exec 16
fd-input-available? 17
fd-open 16
fd-read 16
fd-seek 17
fd-sync-output 17
fd-write 17

fflush 19
file.h 18
File-descriptor Module 15
<file-does-not-exist-error> 96
<file-error> 96
<file-exists-error> 96
<file-stream> 60
find-first-key 131
find-items 120
find-last-key 131
flash 118
flush 31
focus 120
focus-setter 120
font 114
for loop extensions 9
force-output 72
foreground 114
format 51, 109
Format library 109
format-time 149
format-to-string 109
<frame> 121
frame 27
function-definer 42
function-name 49
future Gwydion releases 42

G

gauss-jordan 152
gc 38
<general-integer> 47
geometry 127
geometry-manager 127
get-all 121, 122, 124
get-binding 117
get-bindings 117
getc 19
get-canvas-tags 126
getcwd 13
get-decoded-time 148
get-elements 121, 122, 124
getenv 48
get-input-buffer 85
get-output-buffer 88
get-time-of-day 48
get-units 124
get-universal-time 148
get-value 123
grab-lock 21
graphic? 137
Gwydion compilers 42

H

<hash-state> 132
height 127
Hello, World 5
help 37, 41
hex-digit? 137
highlightbackground 114
highlightforeground 114
highlightthickness 114
history 40
<hours> 148
hours 148

I

icursor 121
identity-matrix 152
<illegal-regexp> 143
<incomplete-read-error> 96
init-keyword 15
inner-stream 100
inner-stream-setter 100
input-available-at-source? 88
insert 121, 122, 124
inspect 37, 39
Inspector-Base 39
Integers 47
integer-to-digit 136
integer-to-string 136
Interrupting 32
Introduction to Linear Algebra 153
Introspection Module 13, 49
<invalid-file-permissions-error> 96
inverse 153
invoke 118
invoke-debugger 26
invoke-entry 122
isqrt 146
item-coords 126
item-coords-setter 126

item-type 126

J

join 143
join-tk-args 128
justify 114

K

keyed-by 42
key-exists? 47
keyword default syntax 9
keyword-required? 15
kill-thread 20

L

<label> 121
library 28
limited collections 9
<limited-integer> 50
limited-integer-base-class 50
limited-integer-maximum 50
limited-integer-minimum 50
line-end 125
<listbox> 122
load 11, 12, 45
locals 27
<lock> 20
locked? 21
Locking streams 81
lock-stream 82
log 146
lowercase? 137
lower-item 126
lower-tag 126

M

macros 9
main 11
make-fooer 138
make-option 128
make-regexp-positioner 141
make-regexp-replacer 141
make-splitter 143
make-substring-positioner 139
make-substring-replacer 139
make-translator 142
mapped? 127
map-window 115
marks 125
<matrix> 152
matrix 152
matrix arithmetic 152
$maximum-integer 47
<menu> 122
<menubutton> 119
<message> 123
mindy 43
Mindy Debugger 26
Mindy vs. Dylan 8, 42
MINDYPATH 10
$minimum-integer 47
<minutes> 148
minutes 148
module 28
<month> 148
month 148
mouse-point 127
mouse-x-y 127
move-item 126
<multilock> 20

N

nearest 122
new-line 75
newlines-are-CRLF 43
#next 9
next-input-buffer 87
next-method 9
next-output-buffer 90
next-range 126
numerator 48

O

O_APPEND 18
O_CREAT 18
O_EXCL 18
O_RDONLY 18
O_RDWR 18
O_TRUNC 18
O_WRONLY 18
object-address 51
one-of 11, 44

on-exit 12
open 9
Osterhout, John 112
outer-stream 100
outer-stream-setter 100

P

pack 116
Packing 116
parse-time 149
Parse-Time Directives 150
parse-tk-list 128
peek 67
Perl 140
pi 145
piped-exec 52
<point> 126
point-in-parent 127
Positionable stream protocol 76
<positionable-stream> 59
post 122
post-cascade 122
postscript 120
Power 146
pprint-indent 108
pprint-logical-block 107
pprint-newline 107
pprint-tab 108
predecessor 137
Pretty Printing 106
primary 9
prin1 19, 51
print 27, 40, 51, 104
Print module 104
printable? 137
print-circle? 106
print-depth 106
print-GC-messages 13
print-length 106
print-level 106
print-message 110
print-miser-width 107
print-object 105
print-pretty? 106
print-to-string 105
punctuation? 137
purify 13
putc 19
puts 19
put-tk-line 128

Q

quit 37, 41

R

<radiobutton> 119
raise-item 126
raise-tag 126
random 151
Random Library 151
Random module 151
random-bits 151
$random-bits-count 151
random-exponential 151
random-float 151
random-gaussian 151
<random-state> 151
<ratio> 47
ratio 47
read 68
read-element 67
read-into! 68
read-line 73
read-line-into! 74
read-through 70
read-to 70
read-to-end 70
regexp-position 140
regexp-replace 141
regexps 140
Regular-Expressions Library 140
release-input-buffer 87
release-lock 22
release-output-buffer 89
relief 114
remove-all-keys! 133
remove-tag 126
report-condition 45
requested-height 127
requested-width 127
restart 31
Restarts and Returning 31

Return Codes 7
<rgb-color> 126
root-window 115
Runtime Errors 8

S

<scale> 123
scale-item 126
scan-dragto 120, 121, 122, 124
scan-mark 120, 121, 122, 124
screen-colormap-cells 127
screen-depth 127
screen-height 127
screen-name 127
screen-width 127
scroll 124
<scrollbar> 123
sealing 9
<seconds> 148
seconds 148
Section The Extensions Module 6
Sedgwick, Robert 153
see 122
SEEK_CUR 18
SEEK_END 18
SEEK_SET 18
select-adjust 121, 122, 124
select-clear 121, 122, 124
select-from 121, 122, 124
selection-anchor-setter 122
selection-includes? 122
select-item 120
select-to 121, 122, 124
select-value 119, 120
<self-organizing-list> 130
Self-Organizing-List Module 130
<semaphore> 21
sequence-hash 132
<sequence-stream> 60
server 127
set-selection 122
set-units 124
set-value 123
signal-event 23
sin 145
$single-e 146
$single-pi 145
singleton-object 50
sinh 145
size 122
skip-through 71
slot-allocation 14
<slot-descriptor> 13
slot-descriptors 14
slot-getter 14
slot-name 14
slot-setter 14
slot-type 14
slot-value 15
slot-value-setter 15
spawn-thread 20
<spinlock> 20
split 142
sqrt 146
Stack Manipulation Commands 26
state 115
step 33
Stepping 32
Store 41
Strang, Gilbert 153
<stream> 59
Stream extension protocol 93
stream-at-end? 76
stream-contents 79
stream-element-type 76
stream-input-available? 69
stream-locked? 82
stream-open? 75
<stream-position> 77
stream-position 77
stream-position-setter 77
stream-size 78
strftime 149
String-Conversions Module 136
String-extensions Library 136
String-hacking Module 137
string-hash 133
<string-stream> 60
<string-table> 132
string-to-integer 136
strptime 149
<subclass> 50

subclass specializers 43
subclass-of 50
<subsequence> 130
subsequence 130
substring-position 138
substring-replace 139
Substring-search Module 138
successor 137
synchronize-output 72
Syntax Errors 7
system 49
System Module 12, 48

T

Table-Extensions Library 132
Table-Extensions Module 132
Tables 46
tags 124, 126
tan 145
tanh 145
Tcl 112
Tcl and the Tk Toolkit 112
<text> 124
text 115
text-at 125
<text-index> 125
Text-Inspector 39
Text-Inspector Commands 40
<text-mark> 125
<text-tag> 125
textvariable 115
The Subsequence Module 130
<thread> 19
thread 36
Threads Module 19
Time Library 148
Time Module 148
<timezone> 148
timezone 148
tk-as 128
tk-break 117
tk-dialog 125
Tk-extension 128
tk-quote 129
tk-unquote 129
toggle-value 119
<toplevel> 124
toplevel 127
Transcendental Library 145
Transcendental module 145
translate 141
transpose 153
troff 38
tron 38
type-for-file-stream 62
type-for-sequence-stream 65

U

underline 115
<unicode-character> 85
<unicode-string-stream> 61
<union> 50
union-members 51
<universal-time> 148
unlock-stream 82
unmap-window 115
unpack 116
unpost 122
unread-element 79
up 26, 40
uppercase? 137
using 42

V

value-hash 135
values-hash 133
<value-table> 134
variable 115
Vector-Search Module 131
view 41
viewable? 127
virtual-root-height 127
virtual-root-position-point 127
virtual-root-position-x-y 127
virtual-root-width 127
visual-class 127

W

wait-for-event 23
warning-output 46
Warnings 8
Waters, Richard C. 106

Weak Pointers 12
<weak-pointer> 12
weak-pointer-object 12
whitespace? 137
width 127
<window> 115
window-containing-point 127
window-containing-x-y 127
window-with-X-id 127
winfo 126
wish 112
with-input-buffer 87
with-open-file 80
with-output-buffer 90
with-stream-locked 82
Wrapper stream protocol 100
<wrapper-stream> 100
write 72
write-element 71
write-line 75

X

X-id 127
xinspect 37, 41
X-Inspector 39
xview 120, 121, 122
x-y-in-parent 127

Y

<year> 148
year 148
yposition-entry 123
yview 120, 122, 124