Gwydion Format for Dylan

Designed by the Gwydion Project

Version 1.0 : 14 Dec 94

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 and from the Gwydion Project.

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 [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 [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 [Method]

Arguments

object :: 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.