Gwydion Print for Dylan

Designed by the Gwydion Project

Version 1.0 : 14 Dec 94

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 Dylan does not currently 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.