$\sum$
A Library for ANSI Common Lisp

Christopher Mark Gore
http://www.cgore.com
cgore@cgore.com

INCOMPLETE DRAFT
Monday, April 5th, AD 2021

Contents

Copyright

22 Forest Glade Court, Saint Charles, Missouri 63304 USA.
Web: `http://www.cgore.com`
Email: cgore@cgore.com

Introduction

The Σ library is a generic library of mostly random useful code for ANSI Common Lisp. It is currently only really focused on SBCL, but patches to add support for other systems are more than welcome.

This library started out as a single file, `utilities.lisp`, that I personally used for shared generic code for all of my Lisp code. Most lispers have a similar file of some name, `utilities.lisp`, `misc.lisp`, `shared.lisp`, or even `stuff.lisp`, that is just a random collection of useful little generic macros and functions. Mine has grown over the years, and in 2012 I decided that I should try to make it useful to people other than myself.

You can download the library from GitHub at:
`https://github.com/cgore/sigma`
and I have some other information on it at my own website at:
`http://cgore.com/programming/lisp/sigma/`

Getting Lisp

Before using this library you need a working Lisp. I use and recommend SBCL, Steel Bank Common Lisp, which is available at:
`http://www.sbcl.org`
This is derived from CMUCL, Carnegie Mellon University Common Lisp, which is still under active development and is: available at:
`http://www.cons.org/cmucl/`

SBCL has information on getting started at:
`http://www.sbcl.org/getting.html`
If you are using Debian or a similar Linux distribution (including Ubuntu), you can just run as root:

```
apt-get install sbcl sbcl-doc sbcl-source
```

Getting EMACS and SLIME

After installing, the best way to interact with any Common Lisp is via SLIME, the Superior Lisp Interaction Mode for EMACS, which is available at:

```
http://common-lisp.net/project/slime/
```

This can be installed on Debian by:

```
apt-get install slime emacs emacs-goodies-el
```

Using the Library

First we need to clone the utilities.

```
mkdir -p  /programming/lisp
cd  /programming/lisp
git clone git@github.com:cgore/sigma.git
```

Now we need to make a directory for our project and symlink to the ASDF definition. There are other ways to load ASDF libraries, especially if you want to have them available globally; I strongly recommend you read the documentation to ASDF.

```
mkdir our-new-project
cd our-new-project
ln -s  /programming/lisp/sigma/sigma.asd
```

Now we need to start up our Lisp REPL. The best way to do this for perfonal use is SLIME from within Emacs, but I will demonstrate using the shell itself here.

```
sbcl
```

Now we are in SBCL. Let's calculate something.

$$\sum_{i=0}^{100} i$$

```
(require :asdf)} ; Require ASDF
(require :sigma) ; Require the Sigma system via ASDF.
(sigma:use-all-sigma) ; This will pollute COMMON-LISP-USER.
(sum (loop for i from 1 to 100
          collect i)) ; Returns 5050, and makes Euler sad.
```

Have fun!

Chapter 1

The `sigma/behave` Package

The `sigma/behave` package contains some useful code for confirming behavior of code, supporting a very basic form of *behavior-driven development*, BDD. The basic flow is to define the *behavior* of something, with multiple *specs* specified within that behavior specification, each consisting of various assertions, such as `should=`, `should-equal`, `should-not-equal`, and many others. If the behavior of the thing doesn't match the specified behavior, then there is some error.

1.1 Macros

1.1.1 The `behavior` Macro

The `behavior` macro is used to specify a block of expected behavior for a `thing`. It specifies an example group, loosely similar to the `describe` blocks in Ruby's RSpec. It takes a single argument, the `thing` we are trying to describe, and then a body of code to evaluate that is evaluated in an implicit `progn`. It is to be used around a set of examples, or around a set of assertions directly.

Syntax

```
(behavior thing &body body)
```

Arguments and Values

`thing` This is what we are describing the behavior of.

`body` This is an implicit proc to contain the behavior.

Examples

```
(behavior 'float
          (spec "is␣an␣Abelian␣group"
                (let ((a (random 10.0))
                      (b (random 10.0))
                      (c (random 10.0))
                      (e 1.0))
                  (spec "closure"
                        (should-be-a 'float (* a b)))
                  (spec "associativity"
                        (should= (* (* a b) c)
                                 (* a (* b c))))
                  (spec "identity␣element"
                        (should= a (* e a)))
                  (spec "inverse␣element"
                        (let ((1/a (/ 1 a)))
                          (should= (* 1/a a)
                                   (* a 1/a)
                                   1.0)))
                  (spec "commutitativity"
                        (should= (* a b) (* b a))))))
```

1.1.2 The spec Macro

The `spec` macro is used to indicate a specification for a desired behavior. It will normally serve as a grouping for assertions or nested `spec`s.

Syntax

(spec *description* &body *body*)

Arguments and Values

description This is a string to describe the specification.

body This is an implicit proc to contain the specification.

Examples

```
(spec "should␣pass␣some␣tests"
      (should= 12 (foo 3.5))
      (should= 14 (foo 4.22)))
```

1.1.3 The `should` Macro

The `should` macro is the basic building block for most of the behavior checking. It asserts that `test` returns truthfully for the arguments. Typically you will want to use one of the macros defined on top of `should` instead of using it directly, such as `should=`.

Syntax

(should *test* &rest *arguments*)

Arguments and Values

test This is the test predicate to evaluate.

arguments These are the arguments to the test predicate.

Examples

```
(should #'= 12 (* 3 4)) ; Passes
(should #'< 4 (* 2 3))  ; Passes
(should #'< 4 5 6 7)    ; Passes
```

1.1.4 The `should-not` Macro

The `should-not` macro is identical to the `should` macro, except that it inverts the result of the call with `not`.

Syntax

(should-not *test* &rest *arguments*)

Arguments and Values

test This is the test predicate to evaluate.

arguments These are the arguments to the test predicate.

Examples

```
(should-not #'< 12 4)  ; Passes
(should-not #'= 12 44) ; Passes
```

1.1.5 The `should-be-null` Macro

The `should-be-null` macro is a short-hand method for `(should #'null ...)`.

Syntax

(should-be-null &rest *arguments*)

Arguments and Values

arguments These are the arguments to `null`.

Examples

```
(should-be-null ())            ; Passes
(should-be-null nil)           ; Passes
(should-be-null (not 12))      ; Passes
(should-be-null (and t t nil)) ; Passes
```

1.1.6 The `should-be-true` Macro

The `should-be-true` macro is a short-hand method for `(should #'identity ...)`.

Syntax

(should-be-true &rest *arguments*)

Arguments and Values

arguments These are the arguments to `identity`.

Examples

```
(should-be-true t)               ; Passes
(should-be-true (not nil))       ; Passes
(should-be-true (or nil nil 12)) ; Passes
```

1.1.7 The `should-be-false` Macro

The `should-be-false` macro is a short-hand method for `(should #'not ...)`.

Syntax

(should-be-false &rest *arguments*)

Arguments and Values

arguments These are the arguments to `not`.

Examples

```
(should-be-false nil)
(should-be-false (not t))
(should-be-false (< 44 2))
```

1.1.8 The `should-be-a` Macro

The `should-be-a` macro specifies that one or more `things` should be of the type specified by `type`.

Syntax

(should-be-a *type* &rest *things*)

Arguments and Values

type This is the type to compare with via `typep`.

things These are the things to confirm the type of.

```
(should-be-a 'integer 1)                 ; Passes
(should-be-a 'float 1)                   ; Passes
(should-be-a 'integer 1 2 3 4 5 6 7 8 9) ; Passes
(should-be-a 'integer 1.0)               ; Fails
```

1.1.9 The `should=` Macro

The `should=` macro is a short-hand method for `(should #'= ...)`.

Syntax

(should= &rest *arguments*)

Arguments and Values

arguments These are the arguments to `=`.

Examples

```
(should= 12 12)   ; Passes
(should= 12 12.0) ; Passes
```

1.1.10 The should-not= Macro

The should-not= macro is a short-hand method for (should-not #'= ...).

Syntax

(should-not= &rest *arguments*)

Arguments and Values

arguments These are the arguments to =.

Examples

```
(should-not= 12 12)   ; Fails
(should-not= 12 12.0) ; Fails
(should-not= 12 14)   ; Passes
```

1.1.11 The should/= Macro

The should/= macro is a short-hand method for (should #'/= ...).

Syntax

(should/= &rest *arguments*)

Arguments and Values

arguments These are the arguments to /=.

Examples

```
(should/= 12 13)   ; Passes
(should/= 12 12)   ; Fails
(should/= 12 12.0) ; Fails
```

1.1.12 The should-not/= Macro

The should-not/= macro is a short-hand method for (should-not #'/= ...).

Syntax

(should-not/= &rest *arguments*)

Arguments and Values

arguments These are the arguments to `/=`.

Examples

```
(should-not/= 12 13)   ; Fails
(should-not/= 12 12)   ; Passes
(should-not/= 12 12.0) ; Passes
```

1.1.13 The `should<` Macro

The `should<` macro is a short-hand method for `(should #'< ...)`.

Syntax

(should< &rest *arguments*)

Arguments and Values

arguments These are the arguments to `<`.

Examples

```
(should< 12 13) ; Passes
(should< 13 12) ; Fails
(should< 12 12) ; Fails
```

1.1.14 The `should-not<` Macro

The `should-not<` macro is a short-hand method for `(should-not #'< ...)`.

Syntax

(should-not< &rest *arguments*)

Arguments and Values

arguments These are the arguments to `<`.

Examples

```
(should-not< 12 13) ; Passes
(should-not< 13 12) ; Fails
(should-not< 12 12) ; Fails
```

1.1.15 The `should>` Macro

The `should<` macro is a short-hand method for `(should #'> ...)`.

Syntax

(should> &rest *arguments*)

Arguments and Values

arguments These are the arguments to >.

Examples

```
(should> 12 13) ; Fails
(should> 13 12) ; Passes
(should> 12 12) ; Fails
```

1.1.16 The `should-not>` Macro

The `should-not>` macro is a short-hand method for `(should-not #'> ...)`.

Syntax

(should-not> &rest *arguments*)

Arguments and Values

arguments These are the arguments to >.

Examples

```
(should-not> 12 13) ; Passes
(should-not> 13 12) ; Fails
(should-not> 12 12) ; Passes
```

1.1.17 The `should<=` Macro

The `should<=` macro is a short-hand method for `(should #'<= ...)`.

Syntax

`(should<= &rest` *`arguments`*`)`

Arguments and Values

`arguments` These are the arguments to <=.

Examples

```
(should<= 12 13) ; Passes
(should<= 13 12) ; Fails
(should<= 12 12) ; Passes
```

1.1.18 The `should-not<=` Macro

The `should-not<=` macro is a short-hand method for `(should-not #'<= ...)`.

Syntax

`(should-not<= &rest` *`arguments`*`)`

Arguments and Values

`arguments` These are the arguments to <=.

Examples

```
(should-not<= 12 13) ; Fails
(should-not<= 13 12) ; Passes
(should-not<= 12 12) ; Fails
```

1.1.19 The `should>=` Macro

The `should>=` macro is a short-hand method for `(should #'>= ...)`.

Syntax

`(should>= &rest` *`arguments`*`)`

Arguments and Values

arguments These are the arguments to >=.

Examples

```
(should>= 12 13) ; Fails
(should>= 13 12) ; Passes
(should>= 12 12) ; Passes
```

1.1.20 The should-not>= Macro

The `should-not>=` macro is a short-hand method for `(should-not #'>= ...)`.

Syntax

```
(should-not>= &rest arguments)
```

Arguments and Values

arguments These are the arguments to >=.

Examples

```
(should-not>= 12 13) ; Passes
(should-not>= 13 12) ; Fails
(should-not>= 12 12) ; Fails
```

1.1.21 The should-eq Macro

The `should-eq` macro is a short-hand method for `(should #'eq ...)`.

Syntax

```
(should-eq &rest arguments)
```

Arguments and Values

arguments These are the arguments to `eq`.

Examples

```
(should-eq 12 12)       ; Probably passes
(should-eq 13 12)       ; Fails
(should-eq "foo" "foo") ; May pass, may fail.
```

1.1.22 The should-not-eq Macro

The `should-not-eq` macro is a short-hand method for `(should-not #'eq ...)`.

Syntax

(should-not-eq &rest *arguments*)

Arguments and Values

arguments These are the arguments to `eq`.

Examples

```
(should-not-eq 12 12)       ; Probably fails
(should-not-eq 13 12)       ; Passes
(should-not-eq "foo" "foo") ; May pass, may fail.
```

1.1.23 The should-eql Macro

The `should-eql` macro is a short-hand method for `(should #'eql ...)`.

Syntax

(should-eql &rest *arguments*)

Arguments and Values

arguments These are the arguments to `eql`.

Examples

```
(should-eql 12 12)       ; Passes
(should-eql 13 12)       ; Fails
(should-eql "foo" "foo") ; May pass, may fail.
```

1.1.24 The should-not-eql Macro

The should-not-eql macro is a short-hand method for (should-not #'eql ...).

Syntax

(should-not-eql &rest *arguments*)

Arguments and Values

arguments These are the arguments to eql.

Examples

```
(should-not-eql 12 12)       ; Fails
(should-not-eql 13 12)       ; Passes
(should-not-eql "foo" "foo") ; May pass, may fail.
```

1.1.25 The should-equal Macro

The should-equal macro is a short-hand method for (should #'equal ...).

Syntax

(should-equal &rest *arguments*)

Arguments and Values

arguments These are the arguments to equal.

Examples

```
(should-equal 12 12)       ; Passes
(should-equal 13 12)       ; Fails
(should-equal "foo" "foo") ; Passes
(should-equal "FOO" "foo") ; Fails
```

1.1.26 The should-not-equal Macro

The should-not-equal macro is a short-hand method for (should-not #'equal ...).

Syntax

(should-not-equal &rest *arguments*)

Arguments and Values

arguments These are the arguments to `equal`.

Examples

```
(should-not-equal 12 12)        ; Passes
(should-not-equal 13 12)        ; Fails
(should-not-equal "foo" "foo") ; Fails
(should-not-equal "FOO" "foo") ; Passes
```

1.1.27 The `should-equalp` Macro

The `should-equalp` macro is a short-hand method for `(should #'equalp ...)`.

Syntax

(should-equalp &rest *arguments*)

Arguments and Values

arguments These are the arguments to `equalp`.

Examples

```
(should-equalp 12 12)        ; Passes
(should-equalp 13 12)        ; Fails
(should-equalp "foo" "foo") ; Passes
(should-equalp "FOO" "foo") ; Passes
```

1.1.28 The `should-not-equalp` Macro

The `should-not-equalp` macro is a short-hand method for `(should-not #'equalp ...)`.

Syntax

(should-not-equalp &rest *arguments*)

Arguments and Values

arguments These are the arguments to `equalp`.

Examples

```
(should-not-equalp 12 12)       ; Passes
(should-not-equalp 13 12)       ; Fails
(should-not-equalp "foo" "foo") ; Passes
(should-not-equalp "FOO" "foo") ; Fails
```

1.1.29 The `should-string=` Macro

The `should-string=` macro is a short-hand method for `(should #'string= ...)`.

Syntax

(should-string= &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string=`.

Examples

```
(should-string= "foo" "foo") ; Passes
(should-string= "FOO" "foo") ; Fails
```

1.1.30 The `should-not-string=` Macro

The `should-not-string=` macro is a short-hand method for `(should-not #'string= ...)`.

Syntax

(should-not-string= &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string=`.

Examples

```
(should-not-string= "foo" "foo") ; Fails
(should-not-string= "FOO" "foo") ; Passes
```

1.1.31 The `should-string/=` Macro

The `should-string/=` macro is a short-hand method for `(should #'string/= ...)`.

Syntax

(should-string/= &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string/=`.

Examples

```
(should-string/= "foo" "foo") ; Fails
(should-string/= "FOO" "foo") ; Passes
```

1.1.32 The `should-not-string/=` Macro

The `should-not-string/=` macro is a short-hand method for `(should-not #'string/= ...)`.

Syntax

(should-not-string/= &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string/=`.

Examples

```
(should-not-string/= "foo" "foo") ; Passes
(should-not-string/= "FOO" "foo") ; Fails
```

1.1.33 The `should-string<` Macro

The `should-string<` macro is a short-hand method for `(should #'string< ...)`.

Syntax

(should-string< &rest *arguments*)

Arguments and Values

arguments These are the arguments to string<.

Examples

```
(should-string< "foo" "f")      ; Fails
(should-string< "foo" "foo")    ; Fails
(should-string< "foo" "FOOBAR") ; Fails
(should-string< "foo" "foobar") ; Passes
```

1.1.34 The should-not-string< Macro

The should-not-string< macro is a short-hand method for (should-not #'string< ...).

Syntax

(should-not-string< &rest *arguments*)

Arguments and Values

arguments These are the arguments to string<.

Examples

```
(should-not-string< "foo" "f")      ; Passes
(should-not-string< "foo" "foo")    ; Passes
(should-not-string< "foo" "foobar") ; Fails
```

1.1.35 The should-string> Macro

The should-string> macro is a short-hand method for (should #'string> ...).

Syntax

(should-string> &rest *arguments*)

Arguments and Values

arguments These are the arguments to string>.

Examples

```
(should-string> "foo" "f")      ; Passes
(should-string> "foo" "foo")    ; Fails
(should-string> "foo" "FOO")    ; Passes
(should-string> "foo" "foobar") ; Fails
```

1.1.36 The `should-not-string>` Macro

The `should-not-string>` macro is a short-hand method for `(should-not #'string> ...)`.

Syntax

(should-not-string> &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string>`.

Examples

```
(should-not-string> "foo" "f")      ; Fails
(should-not-string> "foo" "foo")    ; Passes
(should-not-string> "foo" "foobar") ; Passes
```

1.1.37 The `should-string<=` Macro

The `should-string<=` macro is a short-hand method for `(should #'string<= ...)`.

Syntax

(should-string<= &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string<=`.

Examples

```
(should-string<= "foo" "f")      ; Fails
(should-string<= "foo" "foo")    ; Passes
(should-string<= "foo" "foobar") ; Passes
```

1.1.38 The should-not-string<= Macro

The should-not-string<= macro is a short-hand method for (should-not #’string<= ..

Syntax

(should-not-string<= &rest *arguments*)

Arguments and Values

arguments These are the arguments to string<=.

Examples

```
(should-not-string<= "foo" "f")      ; Passes
(should-not-string<= "foo" "foo")    ; Fails
(should-not-string<= "foo" "foobar") ; Fails
```

1.1.39 The should-string>= Macro

The should-string>= macro is a short-hand method for (should #’string>= ...).

Syntax

(should-string>= &rest *arguments*)

Arguments and Values

arguments These are the arguments to string>=.

Examples

```
(should-string>= "foo" "f")      ; Passes
(should-string>= "foo" "foo")    ; Passes
(should-string>= "foo" "foobar") ; Fails
```

1.1.40 The should-not-string>= Macro

The should-not-string>= macro is a short-hand method for (should-not #’string>= ...).

Syntax

(should-not-string>= &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string>=`.

Examples

```
(should-not-string>= "foo" "f")      ; Fails
(should-not-string>= "foo" "foo")    ; Fails
(should-not-string>= "foo" "foobar") ; Passes
```

1.1.41 The `should-string-equal` Macro

The `should-string-equal` macro is a short-hand method for `(should #'string-equal ...)`.

Syntax

```
(should-string-equal &rest arguments)
```

Arguments and Values

arguments These are the arguments to `string-equal`.

Examples

```
(should-string-equal "foo" "foo")    ; Passes
(should-string-equal "FOO" "foo")    ; Passes
(should-string-equal "foo" "foobar") ; Fails
```

1.1.42 The `should-not-string-equal` Macro

The `should-not-string-equal` macro is a short-hand method for `(should-not #'string-equal ...)`.

Syntax

```
(should-not-string-equal &rest arguments)
```

Arguments and Values

arguments These are the arguments to `string-equal`.

Examples

```
(should-not-string-equal "foo" "foo")    ; Fails
(should-not-string-equal "FOO" "foo")    ; Fails
(should-not-string-equal "foo" "foobar") ; Passes
```

1.1.43 The `should-string-not-equal` Macro

The `should-string-not-equal` macro is a short-hand method for `(should #'string-not-equal ...)`.

Syntax

(should-string-not-equal &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string-not-equal`.

Examples

```
(should-string-not-equal "foo" "foo")    ; Fails
(should-string-not-equal "FOO" "foo")    ; Fails
(should-string-not-equal "foo" "foobar") ; Passes
```

1.1.44 The `should-not-string-not-equal` Macro

The `should-not-string-not-equal` macro is a short-hand method for `(should-not #'string-not-equal ...)`.

Syntax

(should-not-string-not-equal &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string-not-equal`.

Examples

```
(should-not-string-not-equal "foo" "foo")    ; Passes
(should-not-string-not-equal "FOO" "foo")    ; Passes
(should-not-string-not-equal "foo" "foobar") ; Fails
```

1.1.45 The `should-string-lessp` Macro

The `should-string-lessp` macro is a short-hand method for `(should #'string-lessp ...)`.

Syntax

(should-string-lessp &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string-lessp`.

Examples

```
(should-string-lessp "foo" "f")      ; Fails
(should-string-lessp "foo" "foo")    ; Fails
(should-string-lessp "foo" "FOOBAR") ; Passes
(should-string-lessp "foo" "foobar") ; Passes
```

1.1.46 The `should-not-string-lessp` Macro

The `should-not-string-lessp` macro is a short-hand method for `(should-not #'string-lessp ...)`.

Syntax

(should-not-string-lessp &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string-lessp`.

Examples

```
(should-not-string-lessp "foo" "f")      ; Passes
(should-not-string-lessp "foo" "foo")    ; Passes
(should-not-string-lessp "foo" "FOOBAR") ; Fails
(should-not-string-lessp "foo" "foobar") ; Fails
```

1.1.47 The `should-string-greaterp` Macro

The `should-string-greaterp` macro is a short-hand method for `(should #'string-greaterp ...)`.

Syntax

(should-string-greaterp &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string-greaterp`.

Examples

```
(should-string-greaterp "foo" "f")      ; Passes
(should-string-greaterp "foo" "foo")    ; Fails
(should-string-greaterp "foo" "FOO")    ; Fails
(should-string-greaterp "foo" "foobar") ; Fails
```

1.1.48 The `should-not-string-greaterp` Macro

The `should-not-string-greaterp` macro is a short-hand method for `(should-not #'string-greaterp ...)`.

Syntax

(should-not-string-greaterp &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string-greaterp`.

Examples

```
(should-not-string-greaterp "foo" "f")      ; Fails
(should-not-string-greaterp "foo" "foo")    ; Passes
(should-not-string-greaterp "foo" "FOO")    ; Passes
(should-not-string-greaterp "foo" "foobar") ; Passes
```

1.1.49 The `should-string-not-greaterp` Macro

The `should-string-not-greaterp` macro is a short-hand method for `(should #'string-not-greaterp ...)`.

Syntax

(should-string-not-greaterp &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string-not-greaterp`.

Examples

```
(should-string-not-greaterp "foo" "f")      ; Fails
(should-string-not-greaterp "foo" "foo")    ; Passes
(should-string-not-greaterp "foo" "FOO")    ; Passes
(should-string-not-greaterp "foo" "foobar") ; Passes
```

1.1.50 The `should-not-string-not-greaterp` Macro

The `should-not-string-not-greaterp` macro is a short-hand method for `(should-not #'string-not-greaterp ...)`.

Syntax

(should-not-string-not-greaterp &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string-not-greaterp`.

Examples

```
(should-not-string-not-greaterp "foo" "f")      ; Passes
(should-not-string-not-greaterp "foo" "foo")    ; Fails
(should-not-string-not-greaterp "foo" "FOO")    ; Fails
(should-not-string-not-greaterp "foo" "foobar") ; Fails
```

1.1.51 The `should-string-not-lessp` Macro

The `should-string-not-lessp` macro is a short-hand method for `(should #'string-not-lessp ...)`.

Syntax

(should-string-not-lessp &rest *arguments*)

Arguments and Values

arguments These are the arguments to `string-not-lessp`.

Examples

```
(should-string-not-lessp "foo" "f")      ; Passes
(should-string-not-lessp "foo" "foo")    ; Passes
(should-string-not-lessp "foo" "FOOBAR") ; Fails
(should-string-not-lessp "foo" "foobar") ; Fails
```

1.1.52 The should-not-string-not-lessp Macro

The should-not-string-not-lessp macro is a short-hand method for (should-not #'string-not-lessp ...).

Syntax

(should-not-string-not-lessp &rest *arguments*)

Arguments and Values

arguments These are the arguments to string-not-lessp.

Examples

```
(should-not-string-not-lessp "foo" "f")      ; Fails
(should-not-string-not-lessp "foo" "foo")    ; Fails
(should-not-string-not-lessp "foo" "FOOBAR") ; Passes
(should-not-string-not-lessp "foo" "foobar") ; Passes
```

Chapter 2

The sigma/control Package

The sigma/control package contains code for basic program control systems. These are mostly basic macros to add more complicated looping, conditionals, or similar. These are typically extensions to Common Lisp that are inspired by other programming languages. Thanks to the power of Common Lisp and its macro system, we can typically implement most features of any other language with little trouble.

2.1 Macros

2.1.1 The aif Macro

The aif macro is an anaphoric variation of the built-in if control structure. This is based on [1, p. 190]. The basic idea is to provide an anaphor (such as pronouns in English) for the conditional so that it can easily be referred to within the body of the conditional expression. The most natural pronoun in the English language for a thing is "it", so that is what is used. If you need or want to use a different anaphor, use a?if. The most common use of aif is for when you want to do some additional computation with some time-consuming calculation, but only if it returned successfully.

Syntax

(aif *conditional t-action* &optional *nil-action*)

Arguments and Values

conditional The boolean conditional to select between the *t-action* and the *nil-action*.

t-action The action to evaluate if the *conditional* evaluate as true.

nil-action The action to evaluate if the *conditional* evaluates as nil.

Examples

```
(aif (big-long-calculation)
     (foo it)
     (format t "The␣big-long-calculation␣failed!~%"))
```

This is similar to the following, but with less typing:

```
(let ((it (big-long-calculation)))
  (if it
      (foo it)
\      (format t "The␣big-long-calculation␣failed!~%")))
```

Or say you need to get a user name from a database call, which might be slow.

```
(aif (get-user-name)
     (format -t "Hello,␣~A!~%" it)
     (format -t "You␣aren't␣logged␣in,␣go␣away!~%"))
```

2.1.2 The `a?if` Macro

The `a?if` macro is a variation of `aif` that allows for the specification of the anaphor to use, instead of being restricted to just `it`, the default with `aif`. This is most often useful when you need to nest calls to anaphoric macros.

Syntax

(a?if *anaphor conditional t-action* &optional *nil-action*)

Arguments and Values

anaphor The result of the *conditional* will be stored in the variable specified as the anaphor.

conditional The boolean conditional to select between the *t-action* and the *nil-action*.

t-action The action to evaluate if the *conditional* evaluate as true.

nil-action The action to evaluate if the *conditional* evaluates as nil.

Examples

```
(a?if foo 'outer
  (a?if bar 'inner
    `(,foo ,bar))) ; Returns '(outer inner)
```

2.1.3 The aand Macro

The aand macro is an anaphoric variation of the built-in and. This is based on [1, p. 191]. It works in a similar manner to aif, defining it as the current argument for use in the next argument, reassigning it with each argument.

Syntax

```
(aand &rest arguments)
```

Examples

```
(aand 2         ; Sets 'it' to 2.
      (* 3 it)  ; Sets 'it' to 6.
      (* 4 it)) ; Returns 24.
```

2.1.4 The a?and Macro

The a?and macro is a variant of aand that allows for the specification of the anaphor to use, instead of being restricted to just it, the default with aand. This is most often useful when you need to nest calls to anaphoric macros.

Examples

```
(a?and foo 12 (* 2 foo) (* 3 foo)) ; Returns 72.

(a?and foo 1 2 3 'outer
  (a?and bar 4 5 6 'inner `(,foo ,bar))) ; Returns '(outer inner)
```

2.1.5 The alambda Macro

The alambda macro is an anaphoric variant of the built-in lambda. This is based on [1, p. 193]. It works in a similar manner to aif and aand, except it defines self instead of it as the default anaphor. This is useful so that you can write recursive lambdas.

```
(funcall (alambda (x) ; Simple recursive factorial example.
           (if (<= x 0)
               1
               (* x (self (1- x)))))
         10))) ; Calculates 10!, inefficently.
```

2.1.6 The a?lambda Macro

The a?lambda macro is an variant of alambda that allows you to specify the anaphor to use, instead of just the default of it.

```
(funcall (a?lambda ! (x) ; Simple recursive factorial example.
           (if (<= x 0)
               1
               (* x (! (1- x)))))
         10))) ; Calculates 10!, inefficently.
```

2.1.7 The ablock Macro

The ablock macro is an anaphoric variant of the built-in block. This is based on [1, p. 193]. It works in a similar manner to aand, defining the anaphor it for each argument to the block.

Examples

```
(let (w x y z)
   (ablock b
           (setf w 7)
           (setf x (* 2 it)) ; Twice w, 14.
           (setf y (* 3 it)) ; Thrice x, 42.
           (return-from b)   ; Leave the block.
           (setf z 123))     ; Never happens.
   (list w x y z))           ; Returns '(7 14 42 nil)
```

2.1.8 The a?block Macro

The a?block macro is an anaphoric variant of ablock that allows you to specify the anaphor to use, instead of just the default of it.

Examples

```
(let (w x y z)
   (a?block b foo
            (setf w 7)
            (setf x (* 2 foo)) ; Twice w, 14.
            (setf y (* 3 foo)) ; Thrice x, 42.
            (return-from b)    ; Leave the block.
            (setf z 123))      ; Never happens.
   (list w x y z))             ; Returns '(7 14 42 nil)
```

2.1.9 The acond Macro

The acond macro is an anaphoric variant of the built-in cond. This is based on [1, p. 191]. It works in a similar manner to aand, defining the anaphor it for each argument to the conditional.

Examples

```
(let (a b (c 3))
  (acond (a it)         ; No.
         (b it)         ; No.
         (c (* 4 it)))) ; Yes, returns 12 = 4*3, the value of c.
```

2.1.10 The a?cond Macro

The a?cond macro is an anaphoric variant of acond that allows you to specify the anaphor to use, instead of just the default of it.

Examples

```
(let (a b (c 3))
  (a?cond foo
          (a foo)         ; No.
          (b foo)         ; No.
          (c (* 4 foo)))) ; Yes, returns 12 = 4*3, the value of c.
```

2.1.11 The awhen Macro

The awhen macro is an anaphoric variant of when built-in. This is based on [1, p. 191]. It works in a similar manner to aif, defining it as the default anaphor. This is useful when the conditional is the result of a complicated computation, so you don't have to compute it twice or wrap the computation in a let block yourself.

Syntax

(awhen *conditional* &body *body*)

Examples

```
(awhen (get-user-name)
  (do-something-with-name it)
  (do-more-stuff)
  (format t "Hello,␣~A!~%" it))
```

2.1.12 The a?when Macro

The a?when macro is similar to the awhen, except that it allows you to specify the anaphor to use, instead of just the default of it.

Syntax

(a?when *conditional* &body *body*)

Examples

```
(a?when user (get-user-name)
  (do-something-with-name user)
  (do-more-stuff)
  (format t "Hello,␣~A!~%" user))
```

2.1.13 The awhile Macro

The awhile macro is an anaphoric variant of while. This is based on [1, p. 191]. This is useful if you need to consume input repeatedly for all input.

Syntax

(awhile *expression* &body *body*)

Examples

```
(awhile (get-input)
  (do-something it)) ; Operate on input for all input.
```

2.1.14 The `a?while` Macro

The `a?while` macro is a variant of `awhile` that allows you to specify the anaphor to use, instead of just the default `it`.

Syntax

```
(awhile anaphor expression &body body)
```

Examples

```
(awhile input (get-input)
  (do-something input)) ; Operate on input for all input.
```

2.1.15 The `deletef` Macro

The `deletef` macro deletes `item` from `sequence` in-place.

Syntax

```
(deletef item sequence &rest rest)
```

Examples

```
(let ((men '(good bad ugly)))
  (deletef 'bad men)
  (deletef 'ugly men)
 men) ; Only the good is left.
```

2.1.16 The `do-while` Macro

The `do-while` macro operates like a `do {BODY} while (CONDITIONAL)` in the C programming language.

Syntax

```
(do-while conditional &body body)
```

Examples

```
(let ((t-minus 10))
  (do-while (<= 0 t-minus)
    (format t "~A␣...␣" t-minus)
    (decf t-minus)))
(format t "Liftoff!~%")
```

2.1.17 The `do-until` Macro

The `do-until` macro operates like a `do {`*`body`*`} while (! `*`conditional`*`)` in the C programming language.

Syntax

(`do-until` *`conditional`* `&body` *`body`*)

Examples

```
(let ((t-minus 10))
  (do-until (< t-minus 0)
    (format t "~A␣...␣" t-minus)
    (decf t-minus)))
(format t "Liftoff!~%")
```

2.1.18 The `fop` Macro

`fop` is like the `opf` macro, but as a post-assignment variant. The difference is similar to the difference between `x++` and `++x` in the C Programming Language, with opf being like `++x` and fop being like `x++`.

Syntax

(`fop` *`operator variable`* `&rest` *`arguments`*)

Examples

```
(let ((x 10))
  (while (<= 0 x)
    (format t "~A␣...␣" (fop #'- x 1))))
(format t "Liftoff!~%")
```

2.1.19 The `for` Macro

A `for` macro, much like the `for` in the C programming language.

Syntax

(`for` *`initial conditional step-action`* `&body` *`body`*)

Examples

```
(for ((i 0))
     (< i 10)
     (incf i)
  (format t \"~%~A\"␣i))
```

2.1.20 The `forever` Macro

The `forever` macro is just a way to say `(while t ...)` with a bit of added expressiveness and explicitness.

Examples

```
(forever (let ((in (read)))
           (if (eq in 'quit))
               (format t "I␣can't␣let␣you␣do␣that,␣Dave.")
               (format t "You␣entered␣~A" in)))
```

2.1.21 The `multicond` Macro

The `multicond` macro is much like `cond`, but where multiple clauses may be evaluated.

Examples

```
(let ((x 12))
  (multicond ((= x 12) ; This will evaluate.
               (format t "X␣is␣12!␣␣My␣favorite␣number!~%"))
             ((< x 100) ; This will evaluate also.
               (format t "X␣is␣small.~%"))
             ((< x 0) ; But this one won't.
               (format t "X␣is␣negative.~%"))))
```

2.1.22 The `opf` Macro

The `opf` macro is a generic operate-and-store, along the lines of `incf` and `decf`, but allowing for any operation.

Syntax

`(opf` *`operator variable`* `&rest` *`arguments`*`)`

Examples

```
;;; Prints 1 ... 2 ... 4 ... 8 ... ··· ... 65535 ... that’s it!
(let ((x 1))
  (while (<= x (expt 2 16))
    (format t "~A␣...␣" x)
    (opf #’* x 2)))
(format t "␣that’s␣it!~%")
```

2.1.23 The swap Macro

This is a simple `swap` macro. The values of the first and second form are swapped with each other.

Syntax

(swap *x y*)

Examples

```
(let ((first "the␣first")
      (last "the␣last"))
  (swap first last)
  `(,first ,last)) ; Returns ’("the last" "the first")
```

2.1.24 The swap-unless Macro

This macro calls `swap` unless the predicate evaluates to true.

Syntax

(swap-unless *predicate x y*)

Examples

```
;;; make smaller and larger in the correct order.
(let ((smaller 12)
      (larger 266))
  (swap-unless #’<= smaller larger))
```

2.1.25 The swap-when Macro

This macro calls `swap` when the predicate evaluates to true.

Syntax

`(swap-when` *`predicate x y`*`)`

Examples

```
;;; make smaller and larger in the correct order.
(let ((smaller 12)
      (larger 6))
  (swap-when #'> smaller larger))
```

2.1.26 The `until` Macro

The `until` macro is similar to the while loop in C, but with a negated conditional.

Syntax

`(until` *`conditional`* `&body` *`body`*`)`

Examples

```
(let ((x 10))
  (until (< x 0)
    (format t "~A␣...␣" x)
    (decf x))
  (format t "Liftoff!~%"))
```

2.1.27 The `while` Macro

This `while` macro is similar to the while loop in C.

Syntax

`(while` *`conditional`* `&body` *`body`*`)`

Examples

```
(let ((x 10))
  (while (<= 0 x)
    (format t "~A␣...␣" x)
    (decf x))
  (format t "Liftoff!~%"))
```

2.2 Functions

2.2.1 The compose Function

The compose function composes a single function from a list of several functions such that the new function is equivalent to calling the functions in succession. This is based upon a compose function in [2], which is based upon the compose function from Dylan.

Syntax

```
(compose &rest functions)
```

Examples

We want to calculate:

$$\sin(\cos(\tan(\pi))) \approx 0.841{,}470{,}984{,}807{,}896{,}5$$

```
(funcall (compose #'sin #'cos #'tan) pi)
(sin (cos (tan pi))) ; This is the same.
```

2.2.2 The conjoin Function

The conjoin function takes in one or more predicates, and returns a predicate that returns true whenever all of the predicates return true. This is from [2] and is based upon the conjoin function from Dylan.

Syntax

```
(conjoin predicate &rest predicates)
```

Examples

```
;;; Returns '(6 12 18 24 30 36 42 48 54 60 66 72 78 84 90 96).
(remove-if-not #'identity
  (flet ((mod-2? (i) (zerop (mod i 2)))
         (mod-3? (i) (zerop (mod i 3))))
    (loop for i from 1 to 100 collect
          (when (funcall (conjoin #'mod-2? #'mod-3?) i)
                i))))
```

2.2.3 The `curry` Function

The `curry` function takes in a function and some of its arguments, and returns a function that expects the rest of the required arguments. This is from [2] and is based upon the `curry` function from Dylan.

Syntax

(curry *function* &rest *arguments*)

Examples

```
(let ((x 100)
      (f (curry #’+ x)))
  (loop for i from 1 to 10 collect
        (funcall f i))) ; Returns ’(101 102 103 ... 110)
```

2.2.4 The `disjoin` Function

The `disjoin` function takes in one or more predicates, and returns a predicate that returns true whenever any of the predicates return true. This is from [2] and is based upon the disjoin function from Dylan.

Syntax

(disjoin *predicate* &rest *predicates*)

Examples

```
;;; Returns  ’(#\1 #\2 #\3 #\a #\b #\c NIL     NIL)
(let ((chars ’(#\1 #\2 #\3 #\a #\b #\c #\Space #\Newline)))
  (mapcar (lambda (c)
            (when (funcall (disjoin #’alpha-char-p #’digit-char-p)
                           c)
                  c))
          chars))
```

2.2.5 The `function-alias` Function

The `function-alias` function produces one or more aliases (alternate names) for a function.

Syntax

(function-alias *function* &rest *aliases*)

Examples

```
(function-alias 'that-guy-doesnt-know-when-to-stop-typing 'shorter)
```

2.2.6 The `operator-to-function` Function

The `operator-to-function` function takes in any symbol and makes an evaluatable function out of it. The principle purpose for this is so that we can treat macros and other non-function things like a function, for using them with `mapcar` or similar.

Known Issues

[Issue #8]

Syntax

```
(operator-to-function operator)
```

Examples

```
;;; In case you don't like (setf a 1 b 2 c 3).
(mapcar (operator-to-function 'setf)
        '(a b c)
        '(1 2 3))
```

2.2.7 The `rcompose` Function

The `rcompose` function is a reversed variant of the `compose` function.

Syntax

```
(compose &rest functions)
```

Examples

We want to calculate:

$$\tan\left(\cos\left(\sin\left(\pi\right)\right)\right) \approx 1.557{,}407{,}724{,}654{,}902{,}3$$

```
(funcall (rcompose #'sin #'cos #'tan) pi)
(tan (cos (sin pi))) ; This is the same.
```

2.2.8 The `rcurry` Function

This function takes in a function and some of its ending arguments, and returns a function that expects the rest of the required arguments. This is from [2] and is based upon the `rcurry` function from Dylan.

Syntax

(rcurry *function* &rest *arguments*)

Examples

```
(let ((x 100)
      (f (rcurry #'- x)))
  (loop for i from 1 to 10 collect
        (funcall f i))) ; Returns '(-99 -98 -97 ... -90)
```

2.2.9 The `unimplemented` Function

This is a convenience function that merely raises an error. It is for code that is yet to be written.

Syntax

(unimplemented)

Examples

```
(defun turing-test-solver ()
  (unimplemented)) ; TODO: figure out how to program this.
```

2.3 Generics

2.3.1 The `duplicate` Generic

The `duplicate` generic is to provide a deep copy facility for any of your objects. If you define a class and want a deep copy facility for it, implement a version of `duplicate` that is correct for it. This library provides versions of `duplicate` for most built-in classes already.

Chapter 3

The sigma/hash Package

3.1 Macros

3.1.1 The sethash Macro

The `sethash` macro is shortcut for `setf gethash`.

Syntax

(set-partition *key hash-table value*)

Arguments and Values

key The key to add to in the hash table.

hash-table The hash table we are modifying.

value The value to set the hash table key to.

Returns

The new *value* for key.

3.1.2 The set-partition Macro

The `set-partition` macro is a variant of `sethash` that assumes the hash entries are all sequences, allowing for multiple results per key. When you call `gethash` on the hashmap you will get back a sequence of all the entries for that key.

Syntax

(set-partition *key hash-table value*)

Arguments and Values

key The key to add to in the hash table.

hash-table The partition, a hash table, we are modifying.

value The value to add to the hash table key.

Returns

The new set of values for key, a sequence containing your newly added *value* and any previously added values.

3.2 Functions

3.2.1 The populate-hash-table Function

The populate-hash-table function makes initial construction of hash tables a lot easier, just taking in key/value pairs as the arguments to the function, and returning a newly-constructed hash table.

Examples

```
(populate-hash-table 'name "Valentinus"
                     'likes '(birds roses)
                     'dislikes '(beheadings epilepsy "false␣idols")
                     'died 269)
```

3.2.2 The inchash Function

The inchash function will increment the value in *key* of the *hash*, initializing it to 1 if it isn't currently defined.

3.2.3 The dechash Function

The dechash function will decrement the value in *key* of the *hash*, initializing it to −1 if it isn't currently defined.

3.2.4 The gethash-in Function

The gethash-in function works like gethash, but allows for multiple keys to be specified at once, to work with nested hash tables.

Syntax

(gethash-in *keys hash-table* &optional *default*)

Arguments and Values

keys A list of objects.

hash-table A hash table.

default An object. The default is nil.

Returns

value An object.

present? A generalized boolean.

Examples

```
(let ((h (make-hash-table)))
  (sethash 'a h 12)
  (gethash-in '(a) h)) ; Returns 12

(let ((h (make-hash-table))
      (i (make-hash-table)))
  (sethash 'b i 123)
  (sethash 'a h i)
  (gethash-in '(a b) h 123)) ; Returns 123
```

3.2.5 The `make-partition` Function

The `make-partition` function is a variant of `make-hash-table` that assumes you're going to use the hash table for partitions with multiple entries per key, not a one-to-one hashmap.

Syntax

```
(make-partition)
```

3.2.6 The `populate-partition` Function

The `populate-partition` function make initial construction of a partition a lot easier, just taking in key/value pairs as the arguments to the function, where there can be multiple entries for any key, and returning a newly-constructed partition.

Syntax

```
(populate-partition &rest pairs)
```

Examples

```
(populate-partition 'a '(1 2 3)
                    'a 4
                    'a '(5 6 7)
                    'b 8
                    'c 9
                    'c 10
                    'd 11
                    'a 147)
```

Chapter 4

The sigma/numeric Package

4.1 Macros

4.1.1 The +f Macro

The +f macro is an alias for incf.

4.1.2 The -f Macro

The -f macro is an alias for decf.

4.1.3 The *f Macro

The *f macro is an alias for multf.

4.1.4 The /f Macro

The /f macro is an alias for divf.

4.1.5 The divf Macro

The divf macro is divide-and-store, along the lines of incf and decf, but with division instead. This is similar to x /= *something* in the C programming language.

Syntax

(divf *variable* &rest *arguments*)

Examples

```
;;; Prints 65536 ... ⋯ 8 ... 4 ... 2 ... 1 ... 0 ... that's it!
(let ((x (expt 2 16)))
  (while (<= 0 x)
    (format t "~A␣...␣" x)
    (divf x 2)))
(format t "␣that's␣it!~%")
```

4.1.6 The `f+` Macro

The `f+` macro is similar to `incf` or `+f`, but it is a post-increment instead of a pre-increment. That is, `f+` works like `x++` in C but `incf` and `+f` work like `++x` in C.

Syntax

(f+ *variable* &rest *addends*)

Examples

```
(let ((x 12))
  (list x (f+ x) x)) ; Returns '(12 12 13).
```

4.1.7 The `f-` Macro

The `f-` macro is similar to `decf` or `-f`, but it is a post-decrement instead of a pre-decrement. That is, `f-` works like `x--` in C but `decf` and `-f` work like `--x` in C.

Syntax

(f- *variable* &rest *subtrahends*)

Examples

```
(let ((x 12))
  (list x (f- x) x)) ; Returns '(12 12 11).
```

4.1.8 The `f*` Macro

The `f*` macro is similar to `multf` or `*f`, but it is a post-multiply instead of a pre-multiply. That is, `f*` works like `x++` in C (just for multiplication instead of addition) but `multf` and `*f` work like `++x` in C (again, just for multiplication instead of addition.)

Syntax

`(f* variable &rest multiplicands)`

Examples

```
(let ((x 12))
  (list x (f* x 2) x)) ; Returns ’(12 12 24).
```

4.1.9 The `f/` Macro

The `f/` macro is similar to `divf` or `/f`, but it is a post-divide instead of a pre-divide. That is, `f/` works like `x++` in C (just for division instead of addition) but `divf` and `/f` work like `++x` in C (again, just for division instead of addition.)

Syntax

`(f/ variable &rest divisors)`

Examples

```
(let ((x 12))
  (list x (f/ x 2) x)) ; Returns ’(12 12 6).
```

4.1.10 The `multf` Macro

The `divf` macro is multiply-and-store, along the lines of `incf` and `decf`, but with multiplication instead. This is similar to `x *=` *something* in the C programming language.

Syntax

`(multf variable &rest arguments)`

Examples

```
;;; Prints 1 ... 2 ... 4 ... 8 ... ⋯ ... 65535 ... that’s it!
(let ((x 1))
  (while (<= x (expt 2 16))
    (format t "~A␣...␣" x)
    (multf x 2)))
(format t "␣that’s␣it!~%")
```

4.2 Functions

4.2.1 The bit? Function

. . . TO DO . . .

4.2.2 The choose Function

The *choose* function computes the binomial coefficient for n and k, typically spoken as *n choose k*, and usually written mathematically as $\binom{n}{k}$.

4.2.3 The factorial Function

The *factorial* function computes $n!$ for positive integers. NB, this isn't intelligent, and uses a loop instead of better approaches.

4.2.4 The fractional-part Function

. . . TO DO . . .

4.2.5 The fractional-value Function

. . . TO DO . . .

4.2.6 The integer-range Function

. . . TO DO . . .

4.2.7 The nonnegative? Function

. . . TO DO . . .

4.2.8 The nonnegative-integer? Function

. . . TO DO . . .

4.2.9 The positive-integer? Function

. . . TO DO . . .

4.2.10 The product Function

. . . TO DO . . .

4.2.11 The sum Function

. . . TO DO . . .

4.2.12 The unsigned-integer? Function

. . . TO DO . . .

4.3 Types

4.3.1 The nonnegative-float Type

. . . TO DO . . .

4.3.2 The nonnegative-integer Type

. . . TO DO . . .

4.3.3 The positive-float Type

. . . TO DO . . .

4.3.4 The positive-integer Type

. . . TO DO . . .

Chapter 5

The sigma/os Package

5.1 Functions

5.1.1 The perl Function

. . . TO DO . . .

5.1.2 The python Function

. . . TO DO . . .

5.1.3 The read-file Function

. . . TO DO . . .

5.1.4 The read-lines Function

. . . TO DO . . .

5.1.5 The ruby Function

. . . TO DO . . .

5.2 Parameters

5.2.1 The *perl-path* Parameter

. . . TO DO . . .

5.2.2 The *python-path* Parameter

. . . TO DO . . .

5.2.3 The *ruby-path* Parameter

. . . TO DO . . .

Chapter 6

The `sigma/probability` Package

6.1 Macros

6.1.1 The `decaying-probabiliity?` Macro

. . . TO DO . . .

6.2 Functions

6.2.1 The `probability?` Function

. . . TO DO . . .

6.3 Types

6.3.1 The `probability` Type

. . . TO DO . . .

Chapter 7

The sigma/random Package

7.1 Macros

7.1.1 The nshuffle Macro

. . . TO DO . . .

7.2 Functions

7.2.1 The gauss Function

. . . TO DO . . .

7.2.2 The random-argument Function

. . . TO DO . . .

7.2.3 The coin-toss Function

. . . TO DO . . .

7.2.4 The random-in-range Function

. . . TO DO . . .

7.2.5 The random-in-ranges Function

. . . TO DO . . .

7.2.6 The `random-range` Function

. . . TO DO . . .

7.2.7 The `randomize-array` Function

. . . TO DO . . .

7.2.8 The `random-array` Function

. . . TO DO . . .

7.3 Generics

7.3.1 The `random-element` Generic

. . . TO DO . . .

7.3.2 The `shuffle` Generic

. . . TO DO . . .

Chapter 8

The sigma/sequence Package

8.1 Macros

8.1.1 The arefable? Macro

. . . TO DO . . .

8.1.2 The nconcf Macro

. . . TO DO . . .

8.1.3 The nthable? Macro

. . . TO DO . . .

8.1.4 The set-nthcdr Macro

. . . TO DO . . .

8.2 Functions

8.2.1 The array-values Function

. . . TO DO . . .

8.2.2 The nth-from-end Function

. . . TO DO . . .

8.2.3 The sequence? Function

. . . TO DO . . .

8.2.4 The empty-sequence? Function

. . . TO DO . . .

8.2.5 The join-symbol-to-all-following Function

This function takes a symbol and a list, and for every occurance of the symbol in the list, it joins it to the item following it. For example:

Syntax

(join-symbol-to-all-following *symbol list*)

Examples

```
(join-symbol-to-all-following :# '(:# 10 :# 20 :# 30))
;; Returns '(:#10 :#20 :#30)
```

Affected By

print-escape, *print-radix*, *print-base*, *print-circle*, *print-pretty*, *print-level*, *print-length*, *print-case*, *print-gensym*, *print-array*.

8.2.6 The join-symbol-to-all-preceeding Function

This function takes a symbol and a list, and for every occurance of the symbol in the list, it joins it to the item preceeding it. For example:

Syntax

(join-symbol-to-all-preceeding *symbol list*)

Examples

```
(join-symbol-to-all-preceeding :% '(10 :% 20 :% 30 :%))
;; Returns '(:10% :20% :30%)
```

Affected By

`*print-escape*`, `*print-radix*`, `*print-base*`, `*print-circle*`, `*print-pretty*`, `*print-level*`, `*print-length*`, `*print-case*`, `*print-gensym*`, `*print-array*`.

8.2.7 The `list-to-vector` Function

. . . TO DO . . .

8.2.8 The `max*` Function

The `max*` function is a shortcut for `max`. It takes in one or more lists and finds the maximum value within all of them. This is so you don't have to manually use `apply` and `concatenate`.

Syntax

```
(min &rest lists)
```

Examples

```
(max* '(1 2 3 100 4 5)) ; Returns 100
(max* '(1 2 3 4)
      '(5 6 99 7)
      '(8 9 10)) ; Returns 99
```

8.2.9 The `min*` Function

The `min*` function is a shortcut for `min`. It takes in one or more lists and finds the maximum value within all of them. This is so you don't have to manually use `apply` and `concatenate`.

Syntax

```
(min &rest lists)
```

Examples

```
(min* '(1 2 3 -100 4 5)) ; Returns -100
(min* '(1 2 3 4)
      '(5 6 -99 7)
      '(8 9 10)) ; Returns -99
```

8.2.10 The `set-equal` Function

. . . TO DO . . .

8.2.11 The `simple-vector-to-list` Function

. . . TO DO . . .

8.2.12 The `sort-order` Function

. . . TO DO . . .

8.2.13 The `the-last` Function

. . . TO DO . . .

8.2.14 The `vector-to-list` Function

. . . TO DO . . .

8.3 Generics

8.3.1 The `best` Generic

. . . TO DO . . .

8.3.2 The `minimum` Generic

. . . TO DO . . .

8.3.3 The `minimum?` Generic

. . . TO DO . . .

8.3.4 The `maximum` Generic

. . . TO DO . . .

8.3.5 The `maximum?` Generic

. . . TO DO . . .

8.3.6 The `sort-on` Generic

. . . TO DO . . .

8.3.7 The `slice` Generic

. . . TO DO . . .

8.3.8 The `split` Generic

. . . TO DO . . .

8.3.9 The `worst` Generic

. . . TO DO . . .

Chapter 9

The `sigma/string` Package

The `String` package contains useful tools for working with strings.

9.1 Functions

9.1.1 The `character-range` Function

The `character-range` function returns a list of characters from the *start* to the *end* character. Note that this is returning a list, not a string.

Syntax

```
(character-range start end) ⟹ '(start ... end)
```

Arguments and Values

start The character to start the range with, inclusive.

end The character to end the range with, inclusive.

Examples

```
(character-range #\a #\e) ⟹ '(#\a #\b #\c #\d #\e)
(character-range #\e #\a) ⟹ '(#\a #\b #\c #\d #\e)
```

9.1.2 The `character-ranges` Function

The `character-ranges` function is a convenience wrapper for `character-range` function, concatenating several calls and making the resultant list contain only unique instances.

Syntax

`(character-ranges` *start*$_1$ *end*$_1$ ... $\Longrightarrow$ `'(`*character*$_1$ `...)`

Arguments and Values

start$_n$ The character to start the nth range with, inclusive.

end$_n$ The character to end the nth range with, inclusive.

Examples

```
(character-ranges #\a #\c #\x #\z) ⟹ '(#\a #\b #\c #\x #\y
#\z)
(character-ranges #\a #\c #\a #\c) ⟹ '(#\a #\b #\c)
```

9.1.3 The `escape-tildes` Function

. . . TO DO . . .

9.1.4 The `replace-char` Function

. . . TO DO . . .

9.1.5 The `strcat` Function

. . . TO DO . . .

9.1.6 The `strmult` Function

. . . TO DO . . .

9.1.7 The `string-join` Function

. . . TO DO . . .

9.1.8 The `stringify` Function

. . . TO DO . . .

9.1.9 The `to-string` Function

. . . TO DO . . .

9.2 Methods

9.2.1 The `split` Methods

. . . TO DO . . .

Chapter 10

The `time-series` Package

10.1 Macros

10.1.1 The `snap-index` Macro

. . . TO DO . . .

10.2 Functions

10.2.1 The `array-raster-line` Function

. . . TO DO . . .

10.2.2 The `distance` Function

. . . TO DO . . .

10.2.3 The `norm` Function

. . . TO DO . . .

10.2.4 The `raster-line` Function

. . . TO DO . . .

10.2.5 The `similar-points?` Function

. . . TO DO . . .

10.2.6 The `time-series?` Function

. . . TO DO . . .

10.2.7 The `time-multiseries?` Function

. . . TO DO . . .

10.2.8 The `tmsref` Function

. . . TO DO . . .

10.2.9 The `tms-dimensions` Function

. . . TO DO . . .

10.2.10 The `tms-raster-line` Function

. . . TO DO . . .

10.2.11 The `tms-values` Function

. . . TO DO . . .

10.3 Types

10.3.1 The `time-multiseries` Type

. . . TO DO . . .

Chapter 11

The `truth` Package

11.1 Functions

11.1.1 The `[?]` Function

. . . TO DO . . .

11.1.2 The `toggle` Function

. . . TO DO . . .

11.2 Generics

11.2.1 The `?` Generic

. . . TO DO . . .

Chapter 12

The sigma Package

12.1 Variables

12.1.1 The *sigma-packages* Variable

. . . TO DO . . .

12.2 Functions

12.2.1 The use-all-sigma Function

. . . TO DO . . .

Bibliography

[1] GRAHAM, P. *On Lisp*. Prentice-Hall, 1993.

[2] GRAHAM, P. ANSI *Common Lisp*. Prentice-Hall, 1995.