

Σ

A Library for ANSI Common Lisp

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

INCOMPLETE DRAFT

Thursday, February 20th, AD 2014

Contents

1 Copyright	9
2 Introduction	11
2.1 Getting Lisp	11
2.2 Getting EMACS and SLIME	12
2.3 Using the Library	12
3 The sigma/behave Package	13
3.1 Macros	13
3.1.1 The behavior Macro	13
3.1.2 The spec Macro	14
3.1.3 The should Macro	14
3.1.4 The should-not Macro	15
3.1.5 The should-be-null Macro	15
3.1.6 The should-be-true Macro	16
3.1.7 The should-be-false Macro	16
3.1.8 The should-be-a Macro	17
3.1.9 The should= Macro	17
3.1.10The should-not= Macro	17
3.1.11The should/= Macro	18
3.1.12The should-not/= Macro	18
3.1.13The should< Macro	19
3.1.14The should-not< Macro	19
3.1.15The should> Macro	19
3.1.16The should-not> Macro	20
3.1.17The should<= Macro	20
3.1.18The should-not<= Macro	21
3.1.19The should>= Macro	21
3.1.20The should-not>= Macro	21
3.1.21The should-eq Macro	22
3.1.22The should-not-eq Macro	22
3.1.23The should-eql Macro	23
3.1.24The should-not-eql Macro	23
3.1.25The should-equal Macro	23

3.1.26	The should-not-equal Macro	24
3.1.27	The should-equalp Macro	24
3.1.28	The should-not-equalp Macro	25
3.1.29	The should-string= Macro	25
3.1.30	The should-not-string= Macro	25
3.1.31	The should-string/= Macro	26
3.1.32	The should-not-string/= Macro	26
3.1.33	The should-string< Macro	27
3.1.34	The should-not-string< Macro	27
3.1.35	The should-string> Macro	27
3.1.36	The should-not-string> Macro	28
3.1.37	The should-string<= Macro	28
3.1.38	The should-not-string<= Macro	29
3.1.39	The should-string>= Macro	29
3.1.40	The should-not-string>= Macro	29
3.1.41	The should-string-equal Macro	30
3.1.42	The should-not-string-equal Macro	30
3.1.43	The should-string-not-equal Macro	31
3.1.44	The should-not-string-not-equal Macro	31
3.1.45	The should-string-lessp Macro	31
3.1.46	The should-not-string-lessp Macro	32
3.1.47	The should-string-greaterp Macro	32
3.1.48	The should-not-string-greaterp Macro	33
3.1.49	The should-string-not-greaterp Macro	33
3.1.50	The should-not-string-not-greaterp Macro	34
3.1.51	The should-string-not-lessp Macro	34
3.1.52	The should-not-string-not-lessp Macro	34
4	The sigma/control Package	37
4.1	Macros	37
4.1.1	The aif Macro	37
4.1.2	The a?if Macro	38
4.1.3	The aand Macro	39
4.1.4	The a?and Macro	39
4.1.5	The alambda Macro	39
4.1.6	The a?lambda Macro	40
4.1.7	The ablock Macro	40
4.1.8	The a?block Macro	40
4.1.9	The acond Macro	40
4.1.10	The a?cond Macro	41
4.1.11	The awhen Macro	41
4.1.12	The a?when Macro	41
4.1.13	The awhile Macro	42
4.1.14	The a?while Macro	42
4.1.15	The deletef Macro	42
4.1.16	The do-while Macro	43

4.1.17	The do-until Macro	43
4.1.18	The for Macro	43
4.1.19	The forever Macro	44
4.1.20	The multicond Macro	44
4.1.21	The opf Macro	44
4.1.22	The swap Macro	44
4.1.23	The swap-unless Macro	44
4.1.24	The swap-when Macro	44
4.1.25	The until Macro	44
4.1.26	The while Macro	44
4.2	Functions	44
4.2.1	The compose Function	44
4.2.2	The conjoin Function	45
4.2.3	The curry Function	45
4.2.4	The disjoin Function	45
4.2.5	The function-alias Function	45
4.2.6	The operator-to-function Function	45
4.2.7	The rcompose Function	45
4.2.8	The rcurry Function	45
4.2.9	The unimplemented Function	45
4.3	Generics	45
4.3.1	The duplicate Generic	45
5	The sigma/hash Package	47
5.1	Macros	47
5.1.1	The sethash Macro	47
5.2	Functions	47
5.2.1	The populate-hash-table Function	47
5.2.2	The inchash Function	47
5.2.3	The dechash Function	47
5.2.4	The gethash-in Function	48
6	The sigma/numeric Package	49
6.1	Macros	49
6.1.1	The divf Macro	49
6.1.2	The multf Macro	49
6.2	Functions	49
6.2.1	The bit? Function	49
6.2.2	The choose Function	49
6.2.3	The factorial Function	49
6.2.4	The fractional-part Function	50
6.2.5	The fractional-value Function	50
6.2.6	The integer-range Function	50
6.2.7	The nonnegative? Function	50
6.2.8	The nonnegative-integer? Function	50
6.2.9	The positive-integer? Function	50

6.2.10	The product Function	50
6.2.11	The sum Function	50
6.2.12	The unsigned-integer? Function	50
6.3	Types	50
6.3.1	The nonnegative-float Type	50
6.3.2	The nonnegative-integer Type	50
6.3.3	The positive-float Type	51
6.3.4	The positive-integer Type	51
7	The sigma/os Package	53
7.1	Functions	53
7.1.1	The perl Function	53
7.1.2	The python Function	53
7.1.3	The read-file Function	53
7.1.4	The read-lines Function	53
7.1.5	The ruby Function	53
7.2	Parameters	53
7.2.1	The *perl-path* Parameter	53
7.2.2	The *python-path* Parameter	54
7.2.3	The *ruby-path* Parameter	54
8	The sigma/probability Package	55
8.1	Macros	55
8.1.1	The decaying-probability? Macro	55
8.2	Functions	55
8.2.1	The probability? Function	55
8.3	Types	55
8.3.1	The probability Type	55
9	The sigma/random Package	57
9.1	Macros	57
9.1.1	The nshuffle Macro	57
9.2	Functions	57
9.2.1	The gauss Function	57
9.2.2	The random-argument Function	57
9.2.3	The coin-toss Function	57
9.2.4	The random-in-range Function	57
9.2.5	The random-in-ranges Function	57
9.2.6	The random-range Function	58
9.2.7	The randomize-array Function	58
9.2.8	The random-array Function	58
9.3	Generics	58
9.3.1	The random-element Generic	58
9.3.2	The shuffle Generic	58

10 The sigma/sequence Package	59
10.1 Macros	59
10.1.1 The arefable? Macro	59
10.1.2 The nconcf Macro	59
10.1.3 The nthable? Macro	59
10.1.4 The set-nthcdr Macro	59
10.2 Functions	59
10.2.1 The array-values Function	59
10.2.2 The nth-from-end Function	59
10.2.3 The sequence? Function	60
10.2.4 The empty-sequence? Function	60
10.2.5 The join-symbol-to-all-following Function	60
10.2.6 The join-symbol-to-all-preceeding Function	60
10.2.7 The list-to-vector Function	61
10.2.8 The set-equal Function	61
10.2.9 The simple-vector-to-list Function	61
10.2.10 The sort-order Function	61
10.2.11 The the-last Function	61
10.2.12 The vector-to-list Function	61
10.3 Generics	61
10.3.1 The best Generic	61
10.3.2 The minimum Generic	61
10.3.3 The minimum? Generic	61
10.3.4 The maximum Generic	61
10.3.5 The maximum? Generic	61
10.3.6 The sort-on Generic	62
10.3.7 The slice Generic	62
10.3.8 The split Generic	62
10.3.9 The worst Generic	62
11 The sigma/string Package	63
11.1 Functions	63
11.1.1 The character-range Function	63
11.1.2 The character-ranges Function	64
11.1.3 The escape-tildes Function	64
11.1.4 The replace-char Function	64
11.1.5 The strcat Function	64
11.1.6 The strmult Function	64
11.1.7 The string-join Function	64
11.1.8 The stringify Function	64
11.1.9 The to-string Function	65
11.2 Methods	65
11.2.1 The split Methods	65

12 The time-series Package	67
12.1 Macros	67
12.1.1 The snap-index Macro	67
12.2 Functions	67
12.2.1 The array-raster-line Function	67
12.2.2 The distance Function	67
12.2.3 The norm Function	67
12.2.4 The raster-line Function	67
12.2.5 The similar-points? Function	67
12.2.6 The time-series? Function	68
12.2.7 The time-multiseries? Function	68
12.2.8 The tmsref Function	68
12.2.9 The tms-dimensions Function	68
12.2.10 The tms-raster-line Function	68
12.2.11 The tms-values Function	68
12.3 Types	68
12.3.1 The time-multiseries Type	68
13 The truth Package	69
13.1 Functions	69
13.1.1 The [?] Function	69
13.1.2 The toggle Function	69
13.2 Generics	69
13.2.1 The ? Generic	69
14 The sigma Package	71
14.1 Variables	71
14.1.1 The *sigma-packages* Variable	71
14.2 Functions	71
14.2.1 The use-all-sigma Function	71

Chapter 1

Copyright

Copyright © 2005 – 2014, Christopher Mark Gore,
Soli Deo Gloria,
All rights reserved.

2317 South River Road, Saint Charles, Missouri 63303 USA.

Web: <http://www.cgore.com>

Email: cgore@cgore.com

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Christopher Mark Gore nor the names of other contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF

LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 2

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/>

2.1 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
```

2.2 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
```

2.3 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 personal use is SLIME from within Emacs, but I will demonstrate using the shell itself here.

```
sbcl
```

Now we are in SBCL.

```
(require :asdf) ; Require ASDF
(require :sigma) ; Require the 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 3

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.

3.1 Macros

3.1.1 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 "commutativity"
        (should= (* a b) (* b a))))))
```

3.1.2 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 specs.

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)))
```

3.1.3 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
```

3.1.4 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
```

3.1.5 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
```

3.1.6 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
```

3.1.7 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))
```

3.1.8 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
```

3.1.9 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
```

3.1.10 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
```

3.1.11 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
```

3.1.12 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
```

3.1.13 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
```

3.1.14 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
```

3.1.15 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
```

3.1.16 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
```

3.1.17 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
```

3.1.18 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
```

3.1.19 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
```

3.1.20 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
```

3.1.21 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.
```

3.1.22 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.
```

3.1.23 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.
```

3.1.24 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.
```

3.1.25 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
```

3.1.26 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
```

3.1.27 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
```

3.1.28 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
```

3.1.29 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
```

3.1.30 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
```

3.1.31 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
```

3.1.32 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
```

3.1.33 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
```

3.1.34 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
```

3.1.35 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
```

3.1.36 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
```

3.1.37 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
```

3.1.38 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
```

3.1.39 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
```

3.1.40 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
```

3.1.41 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
```

3.1.42 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
```

3.1.43 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
```

3.1.44 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
```

3.1.45 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
```

3.1.46 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
```

3.1.47 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
```

3.1.48 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
```

3.1.49 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
```

3.1.50 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
```

3.1.51 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
```

3.1.52 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 4

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.

4.1 Macros

4.1.1 The `aif` Macro

The `aif` macro is an anaphoric variation of the built-in `if` control structure. This is based on [3, 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!~%"))
```

4.1.2 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)
```

4.1.3 The aand Macro

The `aand` macro is an anaphoric variation of the built-in `and`. This is based on [3, 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.
```

4.1.4 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)
```

4.1.5 The alambda Macro

The `alambda` macro is an anaphoric variant of the built-in `lambda`. This is based on [3, 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.
```

4.1.6 The a?lambda Macro

The `a?lambda` macro is an variant of `lambda` 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.
```

4.1.7 The ablock Macro

The `ablock` macro is an anaphoric variant of the built-in `block`. This is based on [3, 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)
```

4.1.8 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)
```

4.1.9 The acond Macro

The `acond` macro is an anaphoric variant of the built-in `cond`. This is based on [3, 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.
```

4.1.10 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.
```

4.1.11 The awhen Macro

The `awhen` macro is an anaphoric variant of `when` built-in. This is based on [3, 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))
```

4.1.12 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))
```

4.1.13 The awhile Macro

The `awhile` macro is an anaphoric variant of `while`. This is based on [3, 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.
```

4.1.14 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.
```

4.1.15 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.
```

4.1.16 The do-while Macro

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

Syntax

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

Examples

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

4.1.17 The do-until Macro

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

Syntax

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

Examples

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

4.1.18 The for Macro

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

Syntax

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

Examples

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

4.1.19 The forever Macro

...TO DO ...

4.1.20 The multicond Macro

...TO DO ...

4.1.21 The opf Macro

...TO DO ...

4.1.22 The swap Macro

...TO DO ...

4.1.23 The swap-unless Macro

...TO DO ...

4.1.24 The swap-when Macro

...TO DO ...

4.1.25 The until Macro

...TO DO ...

4.1.26 The while Macro

...TO DO ...

4.2 Functions**4.2.1 The compose Function**

...TO DO ...

4.2.2 The conjoin Function

...TO DO ...

4.2.3 The curry Function

...TO DO ...

4.2.4 The disjoin Function

...TO DO ...

4.2.5 The function-alias Function

...TO DO ...

4.2.6 The operator-to-function Function

...TO DO ...

4.2.7 The rcompose Function

...TO DO ...

4.2.8 The rcurry Function

...TO DO ...

4.2.9 The unimplemented Function

...TO DO ...

4.3 Generics

4.3.1 The duplicate Generic

...TO DO ...

Chapter 5

The `sigma/hash` Package

5.1 Macros

5.1.1 The `sethash` Macro

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

5.2 Functions

5.2.1 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)
```

5.2.2 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.

5.2.3 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.

5.2.4 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
```


Chapter 6

The sigma/numeric Package

6.1 Macros

6.1.1 The divf Macro

...TO DO ...

6.1.2 The multf Macro

...TO DO ...

6.2 Functions

6.2.1 The bit? Function

...TO DO ...

6.2.2 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 $\binom{n}{k}$.

6.2.3 The factorial Function

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

6.2.4 The fractional-part Function

...TO DO ...

6.2.5 The fractional-value Function

...TO DO ...

6.2.6 The integer-range Function

...TO DO ...

6.2.7 The nonnegative? Function

...TO DO ...

6.2.8 The nonnegative-integer? Function

...TO DO ...

6.2.9 The positive-integer? Function

...TO DO ...

6.2.10 The product Function

...TO DO ...

6.2.11 The sum Function

...TO DO ...

6.2.12 The unsigned-integer? Function

...TO DO ...

6.3 Types**6.3.1 The nonnegative-float Type**

...TO DO ...

6.3.2 The nonnegative-integer Type

...TO DO ...

6.3.3 The positive-float Type

...TO DO ...

6.3.4 The positive-integer Type

...TO DO ...

Chapter 7

The sigma/os Package

7.1 Functions

7.1.1 The perl Function

...TO DO ...

7.1.2 The python Function

...TO DO ...

7.1.3 The read-file Function

...TO DO ...

7.1.4 The read-lines Function

...TO DO ...

7.1.5 The ruby Function

...TO DO ...

7.2 Parameters

7.2.1 The *perl-path* Parameter

...TO DO ...

7.2.2 The **python-path Parameter**

...TO DO ...

7.2.3 The **ruby-path Parameter**

...TO DO ...

Chapter 8

The sigma/probability Package

8.1 Macros

8.1.1 The decaying-probability? Macro

...TO DO ...

8.2 Functions

8.2.1 The probability? Function

...TO DO ...

8.3 Types

8.3.1 The probability Type

...TO DO ...

Chapter 9

The sigma/random Package

9.1 Macros

9.1.1 The nshuffle Macro

...TO DO ...

9.2 Functions

9.2.1 The gauss Function

...TO DO ...

9.2.2 The random-argument Function

...TO DO ...

9.2.3 The coin-toss Function

...TO DO ...

9.2.4 The random-in-range Function

...TO DO ...

9.2.5 The random-in-ranges Function

...TO DO ...

9.2.6 The random-range Function

...TO DO ...

9.2.7 The randomize-array Function

...TO DO ...

9.2.8 The random-array Function

...TO DO ...

9.3 Generics**9.3.1 The random-element Generic**

...TO DO ...

9.3.2 The shuffle Generic

...TO DO ...

Chapter 10

The sigma/sequence Package

10.1 Macros

10.1.1 The arefable? Macro

...TO DO ...

10.1.2 The nconcf Macro

...TO DO ...

10.1.3 The nthable? Macro

...TO DO ...

10.1.4 The set-nthcdr Macro

...TO DO ...

10.2 Functions

10.2.1 The array-values Function

...TO DO ...

10.2.2 The nth-from-end Function

...TO DO ...

10.2.3 The sequence? Function

...TO DO ...

10.2.4 The empty-sequence? Function

...TO DO ...

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

This function takes a symbol and a list, and for every occurrence 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*.

10.2.6 The join-symbol-to-all-preceding Function

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

Syntax

```
(join-symbol-to-all-preceding symbol list)
```

Examples

```
(join-symbol-to-all-preceding :% '(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*.

10.2.7 The list-to-vector Function

...TO DO ...

10.2.8 The set-equal Function

...TO DO ...

10.2.9 The simple-vector-to-list Function

...TO DO ...

10.2.10 The sort-order Function

...TO DO ...

10.2.11 The the-last Function

...TO DO ...

10.2.12 The vector-to-list Function

...TO DO ...

10.3 Generics**10.3.1 The best Generic**

...TO DO ...

10.3.2 The minimum Generic

...TO DO ...

10.3.3 The minimum? Generic

...TO DO ...

10.3.4 The maximum Generic

...TO DO ...

10.3.5 The maximum? Generic

...TO DO ...

10.3.6 The sort-on Generic

...TO DO ...

10.3.7 The slice Generic

...TO DO ...

10.3.8 The split Generic

...TO DO ...

10.3.9 The worst Generic

...TO DO ...

Chapter 11

The `string` Package

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

11.1 Functions

11.1.1 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)  $\Rightarrow$  ' (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)  $\Rightarrow$  ' (#\a #\b #\c #\d #\e)  
(character-range #\e #\a)  $\Rightarrow$  ' (#\a #\b #\c #\d #\e)
```

11.1.2 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 start1 end1 ... ⇒ ' (character1 ...)
```

Arguments and Values

start_n The character to start the *n*th range with, inclusive.

end_n The character to end the *n*th 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)
```

11.1.3 The escape-tildes Function

...TO DO ...

11.1.4 The replace-char Function

...TO DO ...

11.1.5 The strcat Function

...TO DO ...

11.1.6 The strmult Function

...TO DO ...

11.1.7 The string-join Function

...TO DO ...

11.1.8 The stringify Function

...TO DO ...

11.1.9 The to-string Function

...TO DO ...

11.2 Methods

11.2.1 The split Methods

...TO DO ...

Chapter 12

The time-series Package

12.1 Macros

12.1.1 The snap-index Macro

...TO DO ...

12.2 Functions

12.2.1 The array-raster-line Function

...TO DO ...

12.2.2 The distance Function

...TO DO ...

12.2.3 The norm Function

...TO DO ...

12.2.4 The raster-line Function

...TO DO ...

12.2.5 The similar-points? Function

...TO DO ...

12.2.6 The `time-series?` Function

...TO DO ...

12.2.7 The `time-multiseries?` Function

...TO DO ...

12.2.8 The `tmsref` Function

...TO DO ...

12.2.9 The `tms-dimensions` Function

...TO DO ...

12.2.10 The `tms-raster-line` Function

...TO DO ...

12.2.11 The `tms-values` Function

...TO DO ...

12.3 Types**12.3.1 The `time-multiseries` Type**

...TO DO ...

Chapter 13

The truth Package

13.1 Functions

13.1.1 The `[?]` Function

...TO DO ...

13.1.2 The `toggle` Function

...TO DO ...

13.2 Generics

13.2.1 The `?` Generic

...TO DO ...

Chapter 14

The sigma Package

14.1 Variables

14.1.1 The `*sigma-packages*` Variable

...TO DO ...

14.2 Functions

14.2.1 The `use-all-sigma` Function

...TO DO ...

Bibliography

- [1] Patrick Henry Winston and Berthold Klaus Paul Horn,
Lisp, 3rd Edition.
Addison-Wesley, 1989.
ISBN: 0201083191.
- [2] Guy L. Steele, Jr.,
Common Lisp, The Language, Second Edition.
Digital Press, 1990.
ISBN: 1555580416.
- [3] Paul Graham,
On Lisp.
Prentice-Hall, 1993.
ISBN 0130305529.
Retrieved PDF from: <http://www.paulgraham.com/onlisp.html>
- [4] Paul Graham,
ANSI Common Lisp.
Prentice-Hall, 1995.
ISBN: 0133708756.
- [5] Doug Hoyte,
Let Over Lambda,
Lulu.com, 2008.
ISBN: 1435712757.