

$\Sigma$

# A Library for ANSI Common Lisp

Christopher Mark Gore  
<http://www.cgore.com>  
[cgore@cgore.com](mailto:cgore@cgore.com)

**INCOMPLETE DRAFT**

Wednesday, March 12<sup>th</sup>, AD 2014



# Contents

<b>Copyright</b>	<b>ix</b>
<b>Introduction</b>	<b>xi</b>
0.1 Getting Lisp . . . . .	xi
0.2 Getting EMACS and SLIME . . . . .	xi
0.3 Using the Library . . . . .	xii
<b>1 The sigma/behave Package</b>	<b>1</b>
1.1 Macros . . . . .	1
1.1.1 The behavior Macro . . . . .	1
1.1.2 The spec Macro . . . . .	2
1.1.3 The should Macro . . . . .	2
1.1.4 The should-not Macro . . . . .	3
1.1.5 The should-be-null Macro . . . . .	3
1.1.6 The should-be-true Macro . . . . .	4
1.1.7 The should-be-false Macro . . . . .	4
1.1.8 The should-be-a Macro . . . . .	5
1.1.9 The should= Macro . . . . .	5
1.1.10The should-not= Macro . . . . .	5
1.1.11The should/= Macro . . . . .	6
1.1.12The should-not/= Macro . . . . .	6
1.1.13The should< Macro . . . . .	7
1.1.14The should-not< Macro . . . . .	7
1.1.15The should> Macro . . . . .	7
1.1.16The should-not> Macro . . . . .	8
1.1.17The should<= Macro . . . . .	8
1.1.18The should-not<= Macro . . . . .	9
1.1.19The should>= Macro . . . . .	9
1.1.20The should-not>= Macro . . . . .	9
1.1.21The should-eq Macro . . . . .	10
1.1.22The should-not-eq Macro . . . . .	10
1.1.23The should-eql Macro . . . . .	11
1.1.24The should-not-eql Macro . . . . .	11
1.1.25The should-equal Macro . . . . .	11

1.1.26	The should-not-equal Macro . . . . .	12
1.1.27	The should-equalp Macro . . . . .	12
1.1.28	The should-not-equalp Macro . . . . .	13
1.1.29	The should-string= Macro . . . . .	13
1.1.30	The should-not-string= Macro . . . . .	13
1.1.31	The should-string/= Macro . . . . .	14
1.1.32	The should-not-string/= Macro . . . . .	14
1.1.33	The should-string< Macro . . . . .	15
1.1.34	The should-not-string< Macro . . . . .	15
1.1.35	The should-string> Macro . . . . .	15
1.1.36	The should-not-string> Macro . . . . .	16
1.1.37	The should-string<= Macro . . . . .	16
1.1.38	The should-not-string<= Macro . . . . .	17
1.1.39	The should-string>= Macro . . . . .	17
1.1.40	The should-not-string>= Macro . . . . .	17
1.1.41	The should-string-equal Macro . . . . .	18
1.1.42	The should-not-string-equal Macro . . . . .	18
1.1.43	The should-string-not-equal Macro . . . . .	19
1.1.44	The should-not-string-not-equal Macro . . . . .	19
1.1.45	The should-string-lessp Macro . . . . .	19
1.1.46	The should-not-string-lessp Macro . . . . .	20
1.1.47	The should-string-greaterp Macro . . . . .	20
1.1.48	The should-not-string-greaterp Macro . . . . .	21
1.1.49	The should-string-not-greaterp Macro . . . . .	21
1.1.50	The should-not-string-not-greaterp Macro . . . . .	22
1.1.51	The should-string-not-lessp Macro . . . . .	22
1.1.52	The should-not-string-not-lessp Macro . . . . .	22
<b>2</b>	<b>The sigma/control Package</b>	<b>25</b>
2.1	Macros . . . . .	25
2.1.1	The aif Macro . . . . .	25
2.1.2	The a?if Macro . . . . .	26
2.1.3	The aand Macro . . . . .	27
2.1.4	The a?and Macro . . . . .	27
2.1.5	The alambda Macro . . . . .	27
2.1.6	The a?lambda Macro . . . . .	28
2.1.7	The ablock Macro . . . . .	28
2.1.8	The a?block Macro . . . . .	28
2.1.9	The acond Macro . . . . .	28
2.1.10	The a?cond Macro . . . . .	29
2.1.11	The awhen Macro . . . . .	29
2.1.12	The a?when Macro . . . . .	29
2.1.13	The awhile Macro . . . . .	30
2.1.14	The a?while Macro . . . . .	30
2.1.15	The deletef Macro . . . . .	30
2.1.16	The do-while Macro . . . . .	31

2.1.17	The do-until Macro	31
2.1.18	The fop Macro	31
2.1.19	The for Macro	32
2.1.20	The forever Macro	32
2.1.21	The multicond Macro	32
2.1.22	The opf Macro	33
2.1.23	The swap Macro	33
2.1.24	The swap-unless Macro	33
2.1.25	The swap-when Macro	34
2.1.26	The until Macro	34
2.1.27	The while Macro	34
2.2	Functions	35
2.2.1	The compose Function	35
2.2.2	The conjoin Function	35
2.2.3	The curry Function	36
2.2.4	The disjoin Function	36
2.2.5	The function-alias Function	36
2.2.6	The operator-to-function Function	37
2.2.7	The rcompose Function	37
2.2.8	The rcurry Function	37
2.2.9	The unimplemented Function	37
2.3	Generics	38
2.3.1	The duplicate Generic	38
<b>3</b>	<b>The sigma/hash Package</b>	<b>39</b>
3.1	Macros	39
3.1.1	The sethash Macro	39
3.2	Functions	39
3.2.1	The populate-hash-table Function	39
3.2.2	The inchash Function	39
3.2.3	The dechash Function	39
3.2.4	The gethash-in Function	40
<b>4</b>	<b>The sigma/numeric Package</b>	<b>41</b>
4.1	Macros	41
4.1.1	The +f Macro	41
4.1.2	The -f Macro	41
4.1.3	The *f Macro	41
4.1.4	The /f Macro	41
4.1.5	The divf Macro	41
4.1.6	The f+ Macro	42
4.1.7	The f- Macro	42
4.1.8	The f* Macro	42
4.1.9	The f/ Macro	43
4.1.10	The multf Macro	43
4.2	Functions	43

4.2.1	The bit? Function . . . . .	43
4.2.2	The choose Function . . . . .	44
4.2.3	The factorial Function . . . . .	44
4.2.4	The fractional-part Function . . . . .	44
4.2.5	The fractional-value Function . . . . .	44
4.2.6	The integer-range Function . . . . .	44
4.2.7	The nonnegative? Function . . . . .	44
4.2.8	The nonnegative-integer? Function . . . . .	44
4.2.9	The positive-integer? Function . . . . .	44
4.2.10	The product Function . . . . .	44
4.2.11	The sum Function . . . . .	44
4.2.12	The unsigned-integer? Function . . . . .	44
4.3	Types . . . . .	45
4.3.1	The nonnegative-float Type . . . . .	45
4.3.2	The nonnegative-integer Type . . . . .	45
4.3.3	The positive-float Type . . . . .	45
4.3.4	The positive-integer Type . . . . .	45
<b>5</b>	<b>The sigma/os Package</b>	<b>47</b>
5.1	Functions . . . . .	47
5.1.1	The perl Function . . . . .	47
5.1.2	The python Function . . . . .	47
5.1.3	The read-file Function . . . . .	47
5.1.4	The read-lines Function . . . . .	47
5.1.5	The ruby Function . . . . .	47
5.2	Parameters . . . . .	47
5.2.1	The *perl-path* Parameter . . . . .	47
5.2.2	The *python-path* Parameter . . . . .	48
5.2.3	The *ruby-path* Parameter . . . . .	48
<b>6</b>	<b>The sigma/probability Package</b>	<b>49</b>
6.1	Macros . . . . .	49
6.1.1	The decaying-probability? Macro . . . . .	49
6.2	Functions . . . . .	49
6.2.1	The probability? Function . . . . .	49
6.3	Types . . . . .	49
6.3.1	The probability Type . . . . .	49
<b>7</b>	<b>The sigma/random Package</b>	<b>51</b>
7.1	Macros . . . . .	51
7.1.1	The nshuffle Macro . . . . .	51
7.2	Functions . . . . .	51
7.2.1	The gauss Function . . . . .	51
7.2.2	The random-argument Function . . . . .	51
7.2.3	The coin-toss Function . . . . .	51
7.2.4	The random-in-range Function . . . . .	51

7.2.5	The random-in-ranges Function . . . . .	51
7.2.6	The random-range Function . . . . .	52
7.2.7	The randomize-array Function . . . . .	52
7.2.8	The random-array Function . . . . .	52
7.3	Generics . . . . .	52
7.3.1	The random-element Generic . . . . .	52
7.3.2	The shuffle Generic . . . . .	52
<b>8</b>	<b>The sigma/sequence Package</b>	<b>53</b>
8.1	Macros . . . . .	53
8.1.1	The arefable? Macro . . . . .	53
8.1.2	The nconcf Macro . . . . .	53
8.1.3	The nthable? Macro . . . . .	53
8.1.4	The set-nthcdr Macro . . . . .	53
8.2	Functions . . . . .	53
8.2.1	The array-values Function . . . . .	53
8.2.2	The nth-from-end Function . . . . .	53
8.2.3	The sequence? Function . . . . .	54
8.2.4	The empty-sequence? Function . . . . .	54
8.2.5	The join-symbol-to-all-following Function . . . . .	54
8.2.6	The join-symbol-to-all-preceeding Function . . . . .	54
8.2.7	The list-to-vector Function . . . . .	55
8.2.8	The max* Function . . . . .	55
8.2.9	The min* Function . . . . .	55
8.2.10	The set-equal Function . . . . .	55
8.2.11	The simple-vector-to-list Function . . . . .	55
8.2.12	The sort-order Function . . . . .	56
8.2.13	The the-last Function . . . . .	56
8.2.14	The vector-to-list Function . . . . .	56
8.3	Generics . . . . .	56
8.3.1	The best Generic . . . . .	56
8.3.2	The minimum Generic . . . . .	56
8.3.3	The minimum? Generic . . . . .	56
8.3.4	The maximum Generic . . . . .	56
8.3.5	The maximum? Generic . . . . .	56
8.3.6	The sort-on Generic . . . . .	56
8.3.7	The slice Generic . . . . .	56
8.3.8	The split Generic . . . . .	56
8.3.9	The worst Generic . . . . .	57
<b>9</b>	<b>The sigma/string Package</b>	<b>59</b>
9.1	Functions . . . . .	59
9.1.1	The character-range Function . . . . .	59
9.1.2	The character-ranges Function . . . . .	60
9.1.3	The escape-tildes Function . . . . .	60
9.1.4	The replace-char Function . . . . .	60

9.1.5 The <code>strcat</code> Function . . . . .	60
9.1.6 The <code>strmult</code> Function . . . . .	60
9.1.7 The <code>string-join</code> Function . . . . .	60
9.1.8 The <code>stringify</code> Function . . . . .	60
9.1.9 The <code>to-string</code> Function . . . . .	61
9.2 Methods . . . . .	61
9.2.1 The <code>split</code> Methods . . . . .	61
<b>10 The time-series Package</b>	<b>63</b>
10.1 Macros . . . . .	63
10.1.1 The <code>snap-index</code> Macro . . . . .	63
10.2 Functions . . . . .	63
10.2.1 The <code>array-raster-line</code> Function . . . . .	63
10.2.2 The <code>distance</code> Function . . . . .	63
10.2.3 The <code>norm</code> Function . . . . .	63
10.2.4 The <code>raster-line</code> Function . . . . .	63
10.2.5 The <code>similar-points?</code> Function . . . . .	63
10.2.6 The <code>time-series?</code> Function . . . . .	64
10.2.7 The <code>time-multiseries?</code> Function . . . . .	64
10.2.8 The <code>tmsref</code> Function . . . . .	64
10.2.9 The <code>tms-dimensions</code> Function . . . . .	64
10.2.10 The <code>tms-raster-line</code> Function . . . . .	64
10.2.11 The <code>tms-values</code> Function . . . . .	64
10.3 Types . . . . .	64
10.3.1 The <code>time-multiseries</code> Type . . . . .	64
<b>11 The truth Package</b>	<b>65</b>
11.1 Functions . . . . .	65
11.1.1 The <code>[?]</code> Function . . . . .	65
11.1.2 The <code>toggle</code> Function . . . . .	65
11.2 Generics . . . . .	65
11.2.1 The <code>? Generic</code> . . . . .	65
<b>12 The sigma Package</b>	<b>67</b>
12.1 Variables . . . . .	67
12.1.1 The <code>*sigma-packages*</code> Variable . . . . .	67
12.2 Functions . . . . .	67
12.2.1 The <code>use-all-sigma</code> Function . . . . .	67



# 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](mailto: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.



# Introduction

The  $\Sigma$  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/>

## 0.1 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
```

## 0.2 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 avail-

able at:

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

This can be installed on Debian by:

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

## 0.3 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 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 "commutativity"
        (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 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)))
```

**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**

```
(funcall (compose #'sin #'cos #'tan) pi) ; Returns about 0.8414709848078965.
(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 123)
      (f (curry #' + x)))
  (loop for i from 1 to 10 collect
        (funcall f i)))
```

### 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 evaluable 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**

```
(funcall (rcompose #'sin #'cos #'tan) pi) ; Returns about 1.5574077246549023.  
(tan (cos (sin pi))) ; This is the same.
```

**2.2.8 The rcurry Function**

...TO DO ...

**2.2.9 The unimplemented Function**

...TO DO ...

## **2.3 Generics**

### **2.3.1 The duplicate Generic**

...TO DO ...

## Chapter 3

# The `sigma/hash` Package

### 3.1 Macros

#### 3.1.1 The `sethash` Macro

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

### 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
```

## 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-probability? 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 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\*.

**8.2.6 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\*.

### 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)  $\Rightarrow$  ' (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)
```

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

#### Arguments and Values

**start<sub>n</sub>** The character to start the *n*th range with, inclusive.

**end<sub>n</sub>** 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)
```

### 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.