

$\Sigma$

# A Library for ANSI Common Lisp

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

**INCOMPLETE DRAFT**

Wednesday, February 19<sup>th</sup>, AD 2014



# Contents

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

3.1.26	The <code>should-not-equal</code> Macro . . . . .	23
3.1.27	The <code>should-equalp</code> Macro . . . . .	23
3.1.28	The <code>should-not-equalp</code> Macro . . . . .	24
3.1.29	The <code>should-string=</code> Macro . . . . .	24
3.1.30	The <code>should-not-string=</code> Macro . . . . .	25
3.1.31	The <code>should-string/=</code> Macro . . . . .	25
3.1.32	The <code>should-not-string/=</code> Macro . . . . .	25
3.1.33	The <code>should-string&lt;</code> Macro . . . . .	26
3.1.34	The <code>should-not-string&lt;</code> Macro . . . . .	26
3.1.35	The <code>should-string&gt;</code> Macro . . . . .	27
3.1.36	The <code>should-not-string&gt;</code> Macro . . . . .	27
3.1.37	The <code>should-string&lt;=</code> Macro . . . . .	27
3.1.38	The <code>should-not-string&lt;=</code> Macro . . . . .	28
3.1.39	The <code>should-string&gt;=</code> Macro . . . . .	28
3.1.40	The <code>should-not-string&gt;=</code> Macro . . . . .	29
3.1.41	The <code>should-string-equal</code> Macro . . . . .	29
3.1.42	The <code>should-not-string-equal</code> Macro . . . . .	29
3.1.43	The <code>should-string-not-equal</code> Macro . . . . .	30
3.1.44	The <code>should-not-string-not-equal</code> Macro . . . . .	30
3.1.45	The <code>should-string-lessp</code> Macro . . . . .	30
3.1.46	The <code>should-not-string-lessp</code> Macro . . . . .	31
3.1.47	The <code>should-string-greaterp</code> Macro . . . . .	31
3.1.48	The <code>should-not-string-greaterp</code> Macro . . . . .	32
3.1.49	The <code>should-string-not-greaterp</code> Macro . . . . .	32
3.1.50	The <code>should-not-string-not-greaterp</code> Macro . . . . .	33
3.1.51	The <code>should-string-not-lessp</code> Macro . . . . .	33
3.1.52	The <code>should-not-string-not-lessp</code> Macro . . . . .	33
<b>4</b>	<b>The Sigma/Control Package</b>	<b>35</b>
4.1	Macros . . . . .	35
4.1.1	The <code>AIf</code> Macro . . . . .	35
4.1.2	The <code>A?If</code> Macro . . . . .	36
4.1.3	The <code>AAnd</code> Macro . . . . .	37
4.1.4	The <code>A?And</code> Macro . . . . .	37
4.1.5	The <code>ALambda</code> Macro . . . . .	37
4.1.6	The <code>A?Lambda</code> Macro . . . . .	37
4.1.7	The <code>ABlock</code> Macro . . . . .	38
4.1.8	The <code>A?Block</code> Macro . . . . .	38
4.1.9	The <code>ACond</code> Macro . . . . .	38
4.1.10	The <code>A?Cond</code> Macro . . . . .	39
4.1.11	The <code>AWhen</code> Macro . . . . .	39
4.1.12	The <code>A?When</code> Macro . . . . .	39
4.1.13	The <code>AWhile</code> Macro . . . . .	40
4.1.14	The <code>A?While</code> Macro . . . . .	40
4.1.15	The <code>DeleteF</code> Macro . . . . .	40
4.1.16	The <code>Do-While</code> Macro . . . . .	40

4.1.17	The <code>Do-Until</code> Macro . . . . .	40
4.1.18	The <code>For</code> Macro . . . . .	41
4.1.19	The <code>Forever</code> Macro . . . . .	41
4.1.20	The <code>Multicond</code> Macro . . . . .	41
4.1.21	The <code>OpF</code> Macro . . . . .	41
4.1.22	The <code>Swap</code> Macro . . . . .	41
4.1.23	The <code>Swap-Unless</code> Macro . . . . .	41
4.1.24	The <code>Swap-When</code> Macro . . . . .	41
4.1.25	The <code>Until</code> Macro . . . . .	41
4.1.26	The <code>While</code> Macro . . . . .	41
4.2	Functions . . . . .	41
4.2.1	The <code>Compose</code> Function . . . . .	41
4.2.2	The <code>Conjoin</code> Function . . . . .	41
4.2.3	The <code>Curry</code> Function . . . . .	42
4.2.4	The <code>Disjoin</code> Function . . . . .	42
4.2.5	The <code>Function-Alias</code> Function . . . . .	42
4.2.6	The <code>Operator-To-Function</code> Function . . . . .	42
4.2.7	The <code>RCompose</code> Function . . . . .	42
4.2.8	The <code>RCurry</code> Function . . . . .	42
4.2.9	The <code>Unimplemented</code> Function . . . . .	42
4.3	Generics . . . . .	42
4.3.1	The <code>Duplicate</code> Generic . . . . .	42
<b>5</b>	<b>The Hash Package</b>	<b>43</b>
5.1	Macros . . . . .	43
5.1.1	The <code>sethash</code> Macro . . . . .	43
5.2	Functions . . . . .	43
5.2.1	The <code>populate-hash-table</code> Function . . . . .	43
5.2.2	The <code>inchash</code> Function . . . . .	43
5.2.3	The <code>dechash</code> Function . . . . .	43
5.2.4	The <code>gethash-in</code> Function . . . . .	44
<b>6</b>	<b>The Numeric Package</b>	<b>45</b>
6.1	Macros . . . . .	45
6.1.1	The <code>DivF</code> Macro . . . . .	45
6.1.2	The <code>MultF</code> Macro . . . . .	45
6.2	Functions . . . . .	45
6.2.1	The <code>Bit?</code> Function . . . . .	45
6.2.2	The <code>Choose</code> Function . . . . .	45
6.2.3	The <code>Factorial</code> Function . . . . .	45
6.2.4	The <code>Fractional-Part</code> Function . . . . .	45
6.2.5	The <code>Fractional-Value</code> Function . . . . .	46
6.2.6	The <code>Integer-Range</code> Function . . . . .	46
6.2.7	The <code>Nonnegative?</code> Function . . . . .	46
6.2.8	The <code>Nonnegative-Integer?</code> Function . . . . .	46
6.2.9	The <code>Positive-Integer?</code> Function . . . . .	46

6.2.10	The <code>Product</code> Function . . . . .	46
6.2.11	The <code>Sum</code> Function . . . . .	46
6.2.12	The <code>Unsigned-Integer?</code> Function . . . . .	46
6.3	Types . . . . .	46
6.3.1	The <code>Nonnegative-Float</code> Type . . . . .	46
6.3.2	The <code>Nonnegative-Integer</code> Type . . . . .	46
6.3.3	The <code>Positive-Float</code> Type . . . . .	46
6.3.4	The <code>Positive-Integer</code> Type . . . . .	47
<b>7</b>	<b>The OS Package</b>	<b>49</b>
7.1	Functions . . . . .	49
7.1.1	The <code>Perl</code> Function . . . . .	49
7.1.2	The <code>Python</code> Function . . . . .	49
7.1.3	The <code>Read-File</code> Function . . . . .	49
7.1.4	The <code>Read-Lines</code> Function . . . . .	49
7.1.5	The <code>Ruby</code> Function . . . . .	49
7.2	Parameters . . . . .	49
7.2.1	The <code>*Perl-Path*</code> Parameter . . . . .	49
7.2.2	The <code>*Python-Path*</code> Parameter . . . . .	49
7.2.3	The <code>*Ruby-Path*</code> Parameter . . . . .	50
<b>8</b>	<b>The Probability Package</b>	<b>51</b>
8.1	Macros . . . . .	51
8.1.1	The <code>Decaying-Probabiliity?</code> Macro . . . . .	51
8.2	Functions . . . . .	51
8.2.1	The <code>Probability?</code> Function . . . . .	51
8.3	Types . . . . .	51
8.3.1	The <code>Probability</code> Type . . . . .	51
<b>9</b>	<b>The Random Package</b>	<b>53</b>
9.1	Macros . . . . .	53
9.1.1	The <code>NShuffle</code> Macro . . . . .	53
9.2	Functions . . . . .	53
9.2.1	The <code>Gauss</code> Function . . . . .	53
9.2.2	The <code>Random-Argument</code> Function . . . . .	53
9.2.3	The <code>Coin-Toss</code> Function . . . . .	53
9.2.4	The <code>Random-In-Range</code> Function . . . . .	53
9.2.5	The <code>Random-In-Ranges</code> Function . . . . .	53
9.2.6	The <code>Random-Range</code> Function . . . . .	53
9.2.7	The <code>Randomize-Array</code> Function . . . . .	54
9.2.8	The <code>Random-Array</code> Function . . . . .	54
9.3	Generics . . . . .	54
9.3.1	The <code>Random-Element</code> Generic . . . . .	54
9.3.2	The <code>Shuffle</code> Generic . . . . .	54

<b>10 The Sequence Package</b>	<b>55</b>
10.1 Macros	55
10.1.1 The Arefable? Macro	55
10.1.2 The NConcF Macro	55
10.1.3 The Nthable? Macro	55
10.1.4 The Set-NthCdr Macro	55
10.2 Functions	55
10.2.1 The Array-Values Function	55
10.2.2 The Nth-From-End Function	55
10.2.3 The Sequence? Function	55
10.2.4 The Empty-Sequence? Function	56
10.2.5 The Join-Symbol-To-All-Following Function	56
10.2.6 The Join-Symbol-To-All-Preceding Function	56
10.2.7 The List-To-Vector Function	56
10.2.8 The Set-Equal Function	56
10.2.9 The Simple-Vector-To-List Function	57
10.2.10 The Sort-Order Function	57
10.2.11 The The-Last Function	57
10.2.12 The Vector-To-List Function	57
10.3 Generics	57
10.3.1 The Best Generic	57
10.3.2 The Minimum Generic	57
10.3.3 The Minimum? Generic	57
10.3.4 The Maximum Generic	57
10.3.5 The Maximum? Generic	57
10.3.6 The Sort-On Generic	57
10.3.7 The Slice Generic	57
10.3.8 The Split Generic	58
10.3.9 The Worst Generic	58
<b>11 The String Package</b>	<b>59</b>
11.1 Functions	59
11.1.1 The Character-Range Function	59
11.1.2 The Character-Ranges Function	59
11.1.3 The Escape-Tildes Function	60
11.1.4 The Replace-Char Function	60
11.1.5 The StrCat Function	60
11.1.6 The StrMult Function	60
11.1.7 The String-Join Function	60
11.1.8 The Stringify Function	60
11.1.9 The To-String Function	60
11.2 Methods	60
11.2.1 The Split Methods	60

<b>12 The Time-Series Package</b>	<b>61</b>
12.1 Macros	61
12.1.1 The Snap-Index Macro	61
12.2 Functions	61
12.2.1 The Array-Raster-Line Function	61
12.2.2 The Distance Function	61
12.2.3 The Norm Function	61
12.2.4 The Raster-Line Function	61
12.2.5 The Similar-Points? Function	61
12.2.6 The Time-Series? Function	61
12.2.7 The Time-Multiseries? Function	62
12.2.8 The TMSref Function	62
12.2.9 The TMS-Dimensions Function	62
12.2.10 The TMS-Raster-Line Function	62
12.2.11 The TMS-Values Function	62
12.3 Types	62
12.3.1 The Time-Multiseries Type	62
<b>13 The Truth Package</b>	<b>63</b>
13.1 Functions	63
13.1.1 The [?] Function	63
13.1.2 The Toggle Function	63
13.2 Generics	63
13.2.1 The ? Generic	63
<b>14 The Sigma Package</b>	<b>65</b>
14.1 Variables	65
14.1.1 The *Sigma-Packages* Variable	65
14.2 Functions	65
14.2.1 The Use-All-Sigma Function	65



# 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](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.



## Chapter 2

# 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/>

## 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 perfonal 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 "F00" "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 "F00" "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= "F00" "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= "F00" "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/= "F00" "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 "F00" "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 "F00" "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 "F00" "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 [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!~%"))
```

**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 [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.
```

### 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 [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.
```

### 4.1.6 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.
```

#### 4.1.7 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)
```

#### 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 [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.
```

**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 [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))
```

**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 [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.
```

**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**

...TO DO ...

**4.1.16 The Do-While Macro**

...TO DO ...

**4.1.17 The Do-Until Macro**

...TO DO ...



**4.1.18 The For Macro**

...TO DO ...

**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 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 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 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 Probability Package

### 8.1 Macros

#### 8.1.1 The Decaying-Probabiliity? 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 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 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) ⇒ '(start ... end)`

##### Arguments and Values

*Start* The character to start the range with, inclusive.

*End* The character to end the range with, inclusive.

##### Examples

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

#### 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 ...  $\Rightarrow$  '(character1 ...)`

**Arguments and Values**

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

*End<sub>n</sub>* The character to end the nth range with, inclusive.

**Examples**

`(character-ranges #\a #\c #\x #\z)  $\Rightarrow$  '(#\a #\b #\c #\x #\y #\z)`  
`(character-ranges #\a #\c #\a #\c)  $\Rightarrow$  '(#\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] Paul Graham, *On Lisp*. Prentice-Hall, 1993. ISBN 0130305529. Retrived PDF from <http://www.paulgraham.com/onlisp.html>.