

Σ
A Library for ANSI Common Lisp

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

Friday, July 5th, AD 2013

Contents

1	Copyright	9
2	Introduction	11
2.1	Getting Lisp	11
2.2	Getting EMACS and SLIME	12
2.3	Using the Library	12
3	The Behave Package	13
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	15
3.1.7	The <code>Should-Be-False</code> Macro	15
3.1.8	The <code>Should-Be-A</code> Macro	16
3.1.9	The <code>Should=</code> Macro	16
3.1.10	The <code>Should-Not=</code> Macro	16
3.1.11	The <code>Should/=</code> Macro	17
3.1.12	The <code>Should-Not/=</code> Macro	17
3.1.13	The <code>Should<</code> Macro	17
3.1.14	The <code>Should-Not<</code> Macro	17
3.1.15	The <code>Should></code> Macro	18
3.1.16	The <code>Should-Not></code> Macro	18
3.1.17	The <code>Should<=</code> Macro	18
3.1.18	The <code>Should-Not<=</code> Macro	19
3.1.19	The <code>Should>=</code> Macro	19
3.1.20	The <code>Should-Not>=</code> Macro	19
3.1.21	The <code>Should-Eq</code> Macro	20
3.1.22	The <code>Should-Not-Eq</code> Macro	20
3.1.23	The <code>Should-Eql</code> Macro	20
3.1.24	The <code>Should-Not-Eql</code> Macro	20
3.1.25	The <code>Should-Equal</code> Macro	20

3.1.26	The Should-Not-Equal Macro	20
3.1.27	The Should-EqualP Macro	20
3.1.28	The Should-Not-EqualP Macro	20
3.1.29	The Should-String= Macro	20
3.1.30	The Should-Not-String= Macro	20
3.1.31	The Should-String/= Macro	20
3.1.32	The Should-Not-String/= Macro	20
3.1.33	The Should-String< Macro	20
3.1.34	The Should-Not-String< Macro	20
3.1.35	The Should-String> Macro	20
3.1.36	The Should-Not-String> Macro	20
3.1.37	The Should-String<= Macro	20
3.1.38	The Should-Not-String<= Macro	20
3.1.39	The Should-String>= Macro	20
3.1.40	The Should-Not-String>= Macro	20
3.1.41	The Should-String-Equal Macro	20
3.1.42	The Should-Not-String-Equal Macro	20
3.1.43	The Should-String-Not-Equal Macro	20
3.1.44	The Should-Not-String-Not-Equal Macro	20
3.1.45	The Should-String-LessP Macro	20
3.1.46	The Should-Not-String-LessP Macro	20
3.1.47	The Should-String-GreaterP Macro	20
3.1.48	The Should-Not-String-GreaterP Macro	20
3.1.49	The Should-String-Not-GreaterP Macro	20
3.1.50	The Should-Not-String-Not-GreaterP Macro	20
3.1.51	The Should-String-Not-LessP Macro	20
3.1.52	The Should-Not-String-Not-LessP Macro	20
4	The Control Package	21
4.1	Macros	22
4.1.1	The AIf Macro	22
4.1.2	The A?If Macro	22
4.1.3	The AAnd Macro	22
4.1.4	The A?And Macro	22
4.1.5	The ALambda Macro	22
4.1.6	The A?Lambda Macro	22
4.1.7	The ABlock Macro	22
4.1.8	The A?Block Macro	22
4.1.9	The ACond Macro	22
4.1.10	The A?Cond Macro	22
4.1.11	The AWhen Macro	22
4.1.12	The A?When Macro	22
4.1.13	The AWhile Macro	22
4.1.14	The A?While Macro	22
4.1.15	The DeleteF Macro	22
4.1.16	The Do-While Macro	22

4.1.17	The Do-Until Macro	22
4.1.18	The For Macro	22
4.1.19	The Forever Macro	22
4.1.20	The Multicond Macro	22
4.1.21	The OpF Macro	22
4.1.22	The Swap Macro	22
4.1.23	The Swap-Unless Macro	22
4.1.24	The Swap-When Macro	22
4.1.25	The Until Macro	22
4.1.26	The While Macro	22
4.2	Functions	22
4.2.1	The Compose Function	22
4.2.2	The Conjoin Function	22
4.2.3	The Curry Function	22
4.2.4	The Disjoin Function	22
4.2.5	The Function-Alias Function	22
4.2.6	The Operator-To-Function Function	22
4.2.7	The RCompose Function	22
4.2.8	The RCurry Function	22
4.2.9	The Unimplemented Function	22
4.3	Generics	22
4.3.1	The Duplicate Generic	22
5	The Hash Package	23
5.1	Functions	23
5.1.1	The IncHash Function	23
5.1.2	The DecHash Function	23
6	The Numeric Package	25
6.1	Macros	25
6.1.1	The DivF Macro	25
6.1.2	The MultF Macro	25
6.2	Functions	25
6.2.1	The Bit? Function	25
6.2.2	The Choose Function	25
6.2.3	The Factorial Function	25
6.2.4	The Fractional-Part Function	26
6.2.5	The Fractional-Value Function	26
6.2.6	The Integer-Range Function	26
6.2.7	The Nonnegative? Function	26
6.2.8	The Nonnegative-Integer? Function	26
6.2.9	The Positive-Integer? Function	26
6.2.10	The Product Function	26
6.2.11	The Sum Function	26
6.2.12	The Unsigned-Integer? Function	26
6.3	Types	26

6.3.1	The Nonnegative-Float Type	26
6.3.2	The Nonnegative-Integer Type	26
6.3.3	The Positive-Float Type	26
6.3.4	The Positive-Integer Type	26
7	The OS Package	27
7.1	Functions	27
7.1.1	The Perl Function	27
7.1.2	The Python Function	27
7.1.3	The Read-File Function	27
7.1.4	The Read-Lines Function	27
7.1.5	The Ruby Function	27
7.2	Parameters	27
7.2.1	The *Perl-Path* Parameter	27
7.2.2	The *Python-Path* Parameter	27
7.2.3	The *Ruby-Path* Parameter	27
8	The Probability Package	29
8.1	Macros	29
8.1.1	The Decaying-Probabiliity? Macro	29
8.2	Functions	29
8.2.1	The Probability? Function	29
8.3	Types	29
8.3.1	The Probability Type	29
9	The Random Package	31
9.1	Macros	31
9.1.1	The NShuffle Macro	31
9.2	Functions	31
9.2.1	The Gauss Function	31
9.2.2	The Random-Argument Function	31
9.2.3	The Coin-Toss Function	31
9.2.4	The Random-In-Range Function	31
9.2.5	The Random-In-Ranges Function	31
9.2.6	The Random-Range Function	31
9.2.7	The Randomize-Array Function	31
9.2.8	The Random-Array Function	31
9.3	Generics	31
9.3.1	The Random-Element Generic	31
9.3.2	The Shuffle Generic	31
10	The Sequence Package	33
10.1	Macros	34
10.1.1	The Arefable? Macro	34
10.1.2	The NConcF Macro	34
10.1.3	The Nthable? Macro	34

10.1.4	The <code>Set-NthCdr</code> Macro	34
10.2	Functions	34
10.2.1	The <code>Array-Values</code> Function	34
10.2.2	The <code>Nth-From-End</code> Function	34
10.2.3	The <code>Sequence?</code> Function	34
10.2.4	The <code>Empty-Sequence?</code> Function	34
10.2.5	The <code>Join-Symbol-To-All-Following</code> Function	34
10.2.6	The <code>Join-Symbol-To-All-Preceding</code> Function	34
10.2.7	The <code>List-To-Vector</code> Function	34
10.2.8	The <code>Set-Equal</code> Function	34
10.2.9	The <code>Simple-Vector-To-List</code> Function	34
10.2.10	The <code>Sort-Order</code> Function	34
10.2.11	The <code>The-Last</code> Function	34
10.2.12	The <code>Vector-To-List</code> Function	34
10.3	Generics	34
10.3.1	The <code>Best</code> Generic	34
10.3.2	The <code>Minimum</code> Generic	34
10.3.3	The <code>Minimum?</code> Generic	34
10.3.4	The <code>Maximum</code> Generic	34
10.3.5	The <code>Maximum?</code> Generic	34
10.3.6	The <code>Sort-On</code> Generic	34
10.3.7	The <code>Slice</code> Generic	34
10.3.8	The <code>Split</code> Generic	34
10.3.9	The <code>Worst</code> Generic	34
11	The String Package	35
11.1	Functions	35
11.1.1	The <code>Character-Range</code> Function	35
11.1.2	The <code>Character-Ranges</code> Function	35
11.1.3	The <code>Escape-Tildes</code> Function	36
11.1.4	The <code>Replace-Char</code> Function	36
11.1.5	The <code>StrCat</code> Function	36
11.1.6	The <code>StrMult</code> Function	36
11.1.7	The <code>String-Join</code> Function	36
11.1.8	The <code>Stringify</code> Function	36
11.1.9	The <code>To-String</code> Function	36
11.2	Methods	36
11.2.1	The <code>Split</code> Methods	36
12	The Time-Series Package	37
12.1	Macros	37
12.1.1	The <code>Snap-Index</code> Macro	37
12.2	Functions	37
12.2.1	The <code>Array-Raster-Line</code> Function	37
12.2.2	The <code>Distance</code> Function	37
12.2.3	The <code>Norm</code> Function	37

12.2.4	The Raster-Line Function	37
12.2.5	The Similar-Points? Function	37
12.2.6	The Time-Series? Function	37
12.2.7	The Time-Multiseries? Function	37
12.2.8	The TMSref Function	37
12.2.9	The TMS-Dimensions Function	37
12.2.10	The TMS-Raster-Line Function	37
12.2.11	The TMS-Values Function	37
12.3	Types	37
12.3.1	The Time-Multiseries Type	37
13	The Truth Package	39
13.1	Functions	39
13.1.1	The [?] Function	39
13.1.2	The Toggle Function	39
13.2	Generics	39
13.2.1	The ? Generic	39
14	The Sigma Package	41
14.1	Variables	41
14.1.1	The *Sigma-Packages* Variable	41
14.2	Functions	41
14.2.1	The Use-All-Sigma Function	41

Chapter 1

Copyright

Copyright © 2005 – 2013, Christopher Mark Gore,
Soli Deo Gloria,
All rights reserved.

8729 Lower Marine Road, Saint Jacob, Illinois 62281 USA.

Web: <http://cgore.com>

Email: cgore@cgore.com

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

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

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

Chapter 2

Introduction

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

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

You can download the library from GitHub at:

<https://github.com/cgore/sigma>

and I have some other information on it at my own website at:

<http://cgore.com/programming/lisp/sigma/>

2.1 Getting Lisp

Before using this library you need a working Lisp. I use and recommend SBCL, Steel Bank Common Lisp, which is available at:

<http://www.sbcl.org>

This is derived from CMUCL, Carnegie Mellon University Common Lisp, which is still under active development and is: available at:

<http://www.cons.org/cmucl/>

SBCL has information on getting started at:

<http://www.sbcl.org/getting.html>

If you are using Debian or a similar Linux distribution (including Ubuntu), you can just run as root:

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

2.2 Getting Emacs and Slime

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

<http://common-lisp.net/project/slime/>

This can be installed on Debian by:

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

2.3 Using the Library

First we need to clone the utilities.

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

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

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

Now we need to start up our Lisp REPL. The best way to do this for 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 Behave Package

The `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)
```

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)
```

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)
```

Examples

```

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

```

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)
```

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)
```

Examples

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

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)
```

Examples

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

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)
```

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**.

```
(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)
```

Examples

```
(should= 12 12)
(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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

Examples

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

- 3.1.21 The Should-Eq Macro
- 3.1.22 The Should-Not-Eq Macro
- 3.1.23 The Should-Eql Macro
- 3.1.24 The Should-Not-Eql Macro
- 3.1.25 The Should-Equal Macro
- 3.1.26 The Should-Not-Equal Macro
- 3.1.27 The Should-EqualP Macro
- 3.1.28 The Should-Not-EqualP Macro
- 3.1.29 The Should-String= Macro
- 3.1.30 The Should-Not-String= Macro
- 3.1.31 The Should-String/= Macro
- 3.1.32 The Should-Not-String/= Macro
- 3.1.33 The Should-String< Macro
- 3.1.34 The Should-Not-String< Macro
- 3.1.35 The Should-String> Macro
- 3.1.36 The Should-Not-String> Macro
- 3.1.37 The Should-String<= Macro
- 3.1.38 The Should-Not-String<= Macro
- 3.1.39 The Should-String>= Macro
- 3.1.40 The Should-Not-String>= Macro
- 3.1.41 The Should-String-Equal Macro
- 3.1.42 The Should-Not-String-Equal Macro
- 3.1.43 The Should-String-Not-Equal Macro
- 3.1.44 The Should-Not-String-Not-Equal Macro
- 3.1.45 The Should-String-LessP Macro
- 3.1.46 The Should-Not-String-LessP Macro
- 3.1.47 The Should-String-GreaterP Macro
- 3.1.48 The Should-Not-String-GreaterP Macro
- 3.1.49 The Should-String-Not-GreaterP Macro
- 3.1.50 The Should-Not-String-Not-GreaterP Macro
- 3.1.51 The Should-String-Not-LessP Macro
- 3.1.52 The Should-Not-String-Not-LessP Macro

Chapter 4

The Control Package

4.1 Macros

- 4.1.1 The AIf Macro
- 4.1.2 The A?If Macro
- 4.1.3 The AAnd Macro
- 4.1.4 The A?And Macro
- 4.1.5 The ALambda Macro
- 4.1.6 The A?Lambda Macro
- 4.1.7 The ABlock Macro
- 4.1.8 The A?Block Macro
- 4.1.9 The ACond Macro
- 4.1.10 The A?Cond Macro
- 4.1.11 The AWhen Macro
- 4.1.12 The A?When Macro
- 4.1.13 The AWhile Macro
- 4.1.14 The A?While Macro
- 4.1.15 The DeleteF Macro
- 4.1.16 The Do-While Macro
- 4.1.17 The Do-Until Macro
- 4.1.18 The For Macro
- 4.1.19 The Forever Macro
- 4.1.20 The Multicond Macro
- 4.1.21 The OpF Macro
- 4.1.22 The Swap Macro
- 4.1.23 The Swap-Unless Macro

Chapter 5

The Hash Package

5.1 Functions

5.1.1 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.

5.1.2 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.

Chapter 6

The Numeric Package

6.1 Macros

6.1.1 The DivF Macro

6.1.2 The MultF Macro

6.2 Functions

6.2.1 The Bit? Function

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**6.2.5 The Fractional-Value Function****6.2.6 The Integer-Range Function****6.2.7 The Nonnegative? Function****6.2.8 The Nonnegative-Integer? Function****6.2.9 The Positive-Integer? Function****6.2.10 The Product Function****6.2.11 The Sum Function****6.2.12 The Unsigned-Integer? Function****6.3 Types****6.3.1 The Nonnegative-Float Type****6.3.2 The Nonnegative-Integer Type****6.3.3 The Positive-Float Type****6.3.4 The Positive-Integer Type**

Chapter 7

The OS Package

7.1 Functions

7.1.1 The Perl Function

7.1.2 The Python Function

7.1.3 The Read-File Function

7.1.4 The Read-Lines Function

7.1.5 The Ruby Function

7.2 Parameters

7.2.1 The *Perl-Path* Parameter

7.2.2 The *Python-Path* Parameter

7.2.3 The *Ruby-Path* Parameter

Chapter 8

The Probability Package

8.1 Macros

8.1.1 The Decaying-Probabiliity? Macro

8.2 Functions

8.2.1 The Probability? Function

8.3 Types

8.3.1 The Probability Type

Chapter 9

The Random Package

9.1 Macros

9.1.1 The NShuffle Macro

9.2 Functions

9.2.1 The Gauss Function

9.2.2 The Random-Argument Function

9.2.3 The Coin-Toss Function

9.2.4 The Random-In-Range Function

9.2.5 The Random-In-Ranges Function

9.2.6 The Random-Range Function

9.2.7 The Randomize-Array Function

9.2.8 The Random-Array Function

9.3 Generics

9.3.1 The Random-Element Generic

9.3.2 The Shuffle Generic

Chapter 10

The Sequence Package

10.1 Macros

10.1.1 The Arefable? Macro

10.1.2 The NConcF Macro

10.1.3 The Nthable? Macro

10.1.4 The Set-NthCdr Macro

10.2 Functions

10.2.1 The Array-Values Function

10.2.2 The Nth-From-End Function

10.2.3 The Sequence? Function

10.2.4 The Empty-Sequence? Function

10.2.5 The Join-Symbol-To-All-Following Function

10.2.6 The Join-Symbol-To-All-Preceding Function

10.2.7 The List-To-Vector Function

10.2.8 The Set-Equal Function

10.2.9 The Simple-Vector-To-List Function

10.2.10 The Sort-Order Function

10.2.11 The The-Last Function

10.2.12 The Vector-To-List Function

10.3 Generics

10.3.1 The Best Generic

10.3.2 The Minimum Generic

10.3.3 The Minimum? Generic

10.3.4 The Maximum Generic

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) \implies '(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)  $\implies$  '(#\a #\b #\c #\d #\e)
(character-range #\e #\a)  $\implies$  '(#\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 ... \implies '(character1 ...)`

Arguments and Values

Start_n The character to start the nth range with, inclusive.

End_n The character to end the nth range with, inclusive.

Examples

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

`(character-ranges #\a #\c #\a #\c) \implies '(#\a #\b #\c)`

11.1.3 The Escape-Tildes Function**11.1.4 The Replace-Char Function****11.1.5 The StrCat Function****11.1.6 The StrMult Function****11.1.7 The String-Join Function****11.1.8 The Stringify Function****11.1.9 The To-String Function****11.2 Methods****11.2.1 The Split Methods**

Chapter 12

The Time-Series Package

12.1 Macros

12.1.1 The Snap-Index Macro

12.2 Functions

12.2.1 The Array-Raster-Line Function

12.2.2 The Distance Function

12.2.3 The Norm Function

12.2.4 The Raster-Line Function

12.2.5 The Similar-Points? Function

12.2.6 The Time-Series? Function

12.2.7 The Time-Multiseries? Function

12.2.8 The TMSref Function

12.2.9 The TMS-Dimensions Function

12.2.10 The TMS-Raster-Line Function

12.2.11 The TMS-Values Function

12.3 Types

12.3.1 The Time-Multiseries Type

Chapter 13

The Truth Package

13.1 Functions

13.1.1 The `[?]` Function

13.1.2 The `Toggle` Function

13.2 Generics

13.2.1 The `?` Generic

Chapter 14

The Sigma Package

14.1 Variables

14.1.1 The `*Sigma-Packages*` Variable

14.2 Functions

14.2.1 The `Use-All-Sigma` Function