Σ
A Library for ANSI Common Lisp

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

Friday, June 21st, AD 2013

Contents

Chapter 1

Copyright

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 "commutitativity"
                  (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 `spec`s.

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)
(should-not #'= 12 44)
```

3.1.5 The `Should-Be-Null` Macro

3.1.6 The `Should-Be-A` Macro

3.1.7 The `Should=` Macro

3.1.8 The `Should/=` Macro

3.1.9 The `Should<` Macro

3.1.10 The `Should>` Macro

3.1.11 The `Should<=` Macro

3.1.12 The `Should>=` Macro

3.1.13 The `Should-Eq` Macro

3.1.14 The `Should-Not-Eq` Macro

3.1.15 The `Should-Eql` Macro

3.1.16 The `Should-Not-Eql` Macro

3.1.17 The `Should-Equal` Macro

3.1.18 The `Should-Not-Equal` Macro

3.1.19 The `Should-EqualP` Macro

3.1.20 The `Should-Not-EqualP` Macro

3.1.21 The `Should-String=` Macro

3.1.22 The `Should-Not-String=` Macro

3.1.23 The `Should-String/=` Macro

3.1.24 The `Should-Not-String/=` Macro

3.1.25 The `Should-String<` Macro

3.1.26 The `Should-Not-String<` Macro

3.1.27 The `Should-String>` Macro

3.1.28 The `Should-Not-String>` 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-Equal` Macro

3.1.34 The `Should-Not-String-Equal` Macro

3.1.35 The `Should-String-Not-Equal` Macro

3.1.36 The `Should-Not-String-Not-Equal` Macro

3.1.37 The `Should-String-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-Preceeding 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*`)` $\Longrightarrow$ `'(`*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` *start*$_1$ *end*$_1$ … $\Longrightarrow$ `'(`*character*$_1$ …`)`

Arguments and Values

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

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

Examples

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

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