misc-extn egg

Provides miscellaneous useful stuff.
Extension for Chicken Scheme
Version 1.9

Kon Lovett

Table of Contents

1 About this egg

1.1 Version history

`1.9` Rename ++f* -> f*++, remove mu & nu; me stupid

`1.8` Moved procedures to own extension

`1.7` Exports, additions

`1.6` More stuff, rename fp++, etc -> ++fp

`1.5` Added alist-delete-first

`1.4` Added mu, nu, moved looping constructs to miscmacros

`1.3` Removed use of define-syntax

`1.2` Added assure, nl, rename set*! to stiff-set!, fp++, etc.

`1.1` Added plain repeat, assoc macros signal errors

`1.0` Initial release

1.2 Usage

Load this egg like so:

```
(require-extension misc-extn)
```

2 Documentation

To access the procedures do: `(require-extension misc-extn-procs)`.

2.1 Record

`define-unchecked-record-type` [macro]

```
(define-unchecked-record-type T CTOR PRED [SLOT ...]))
```

SRFI-9 '(define-record-type T CTOR PRED [SLOT ...])', except no checks are made for correct record type before slot access, and the record type symbol is not defined.

For use when slot access is attempted *only* after determining the correct record type explicitly. Do *not* make constructed slot access procedures part of a public API.

2.2 Control

`errorf` [macro]

```
(errorf [ID] FORMAT ARGS ...)
```

Same as '(error [ID] (sprintf FORMAT ARGS ...))'.

`swap-set!` [macro]

```
(swap-set! VAR1 VAR2)
```

Swap settings of `VAR1` & `VAR2`.

`fluid-set!` [macro]

```
(fluid-set! VAR VAL ...)
```

Set each variable `VAR` to the value `VAL` in parallel.

`stiff-set!` [macro]

```
(stiff-set! VAR VAL ...)
```

Set each variable `VAR` to the value `VAL` in series.

`(hash-let (([VAR | (VAR KEY)] ...)  HASH-TABLE) BODY ...)` [macro]

Decompose `HASH-TABLE` entries into variable bindings. Should the `KEY` not be symbol, or the desired variable name `VAR` should not be the key, the '(VAR KEY)' form can be used. The `BODY ...` is evaluated with the specified bindings.

`set-op!` [macro]

```
(set-op! VAR OP ARG ...)
```

Sets `VAR` to the value `(OP VAR ARG ...)`.

`assure` [macro]

```
(assure EXPRESSION ERROR-ARGUMENT ...)
```

When `EXPRESSION` yields `#f` invoke `(error ERROR-ARGUMENT ...)`.

2.3 Lists

`length=0?` [macro]

```
(length=0? LIST)
```

List of length zero?

`length=1?` [macro]

```
(length=1? LIST)
```

List of length one?

`length>1?` [macro]

```
(length>1? LIST)
```

List of length greater than one?

2.4 Arithmetic

`++` [macro]

```
(++ VAL)
```

Read-only increment.

`--` [macro]

```
(-- VAL)
```

Read-only decrement.

`fx++` [macro]

```
(fx++ VAL)
```

Read-only fixnum increment.

`fx--` [macro]

```
(fx-- VAL)
```

Read-only fixnum decrement.

`fp++` [macro]

```
(fp++ VAL)
```

Read-only flonum increment.

`fp--` [macro]

```
(fp-- VAL)
```

Read-only flonum decrement.

`++!` [macro]

```
(++! VAR)
```

Mutable increment.

`--!` [macro]

```
(--! VAR)
```

Mutable decrement.

`fx++!` [macro]

```
(fx++! VAR)
```

Mutable fixnum increment.

`fx--!` [macro]

```
(fx--! VAR)
```

Mutable fixnum decrement.

`fp++!` [macro]

```
(fp++! VAR)
```

Mutable flonum increment.

`fp--!` [macro]

```
(fp--! VAR)
```

Mutable flonum decrement.

`make-log-function` [procedure]

```
(make-log-function BASE)
```

Returns a procedure of one argument, the logarithm function for the `BASE`.

2.5 Association List

`assoc-def` [macro]

```
(assoc-def KEY ALIST [TEST] [DEFAULT])
```

The assoc procedure with an optional test and default value. An error is signaled when not found.

`assv-def` [macro]

```
(assv-def KEY ALIST [DEFAULT])
```

The assv procedure with a default value. An error is signaled when not found.

`assq-def` [macro]

```
(assq-def KEY ALIST [DEFAULT])
```

The assq procedure with a default value. An error is signaled when not found.

`alist-inverse-ref` [procedure]

```
(alist-inverse-ref VALUE ALIST [EQUALITY? eqv?] [DEFAULT #f])
```

Returns the first key associated with `VALUE` in the `ALIST` using the `EQUALITY?` predicate, else `DEFAULT`.

`alist-delete-first` [procedure]

```
(alist-delete-first KEY ALIST [EQUALITY? equal?])
```

Deletes the first association from alist `ALIST` with the given key `KEY`, using key-comparison procedure `EQUALITY?`. The dynamic order in which the various applications of equality are made is from the alist head to the tail.

Return values may share common tails with the alist argument. The alist is not disordered - elements that appear in the result alist occur in the same order as they occur in the argument alist.

The equality procedure is used to compare the element keys, 'key[i: 0 <= i < (length ALIST)]', of the alist's entries to the key parameter in this way: '(EQUALITY? KEY key[i])'.

2.6 Symbol

`defined?` [procedure]

```
(defined? SYMBOL [ENVIRONMENT interaction-environment])
```

Is the `SYMBOL` is defined in the `ENVIRONMENT`?

`make-qualified-symbol` [procedure]

```
(make-qualified-symbol NAMESPACE SYMBOL)
```

Returns the Chicken namespace qualified `SYMBOL` for the `NAMESPACE`.

2.7 Output

`cout` [procedure]

```
(cout EXPR ...)
```

Like cout << arguments << args where argument can be any Scheme object. If it's a procedure (without args) it's executed rather than printed (like newline).

`cerr` [procedure]

```
(cerr EXPR ...)
```

Like cerr << arguments << args where argument can be any Scheme object. If it's a procedure (without args) it's executed rather than printed (like newline).

`nl` [procedure]

```
(nl)
```

Returns a string form of the newline character.

`identify-error` [procedure]

```
(identify-error [CALLER] MSG ARGS ...)
```

Prints a message like `(error ...)` to `(current-error-port)` but does not throw an exception.

2.8 Argument List Processing

`filter-rest-argument` [procedure]

```
(filter-rest-argument REST-LIST [PREDICATE])
```

Remove any keywords & keyword+value items from a #!rest argument list.

When the optional predicate is supplied & it returns #t then the item is kept, otherwise the default processing occurs.

The optional predicate takes the current item & a flag indicating if the previous item was a keyword.

3 Contributions

William Annis - hash-let.

Oleg Kiselyov's Standard Scheme "Prelude" - ++, ...

4 Examples

```
(use misc-extn)

(hash-let ([name (foo "wow")] some-hashtable)
  (print name " " foo #\newline))

(stiff-set! x 1 y 2)    ; x = 1 y = 2
(fluid-set! x y y x)    ; x = 2 y = 1
(swap-set! x y)                         ; x = 1 y = 2
```

5 License

Index