Screamer

A Nondeterministic Extension to Common Lisp

Happy Constraining!

Table of Contents

Introduction

Screamer provides a nondeterministic choice-point operator, a backtracking mechanism, and a forward propagation facility.

Screamer was originally written by Jeffrey Mark Siskind and David Allen McAllester.

The copy of Screamer this documentation refers to is maintained courtesy of Steel Bank Studio Ltd by Nikodemus Siivola.

The Google Group

```
http://groups.google.com/group/screamer/
```

exists for Screamer-related discussions.

Screamer is maintained in Git:

```
git clone git://github.com/nikodemus/screamer.git
```

will get you a local copy.

```
http://github.com/nikodemus/screamer
```

is the GitHub project page.

1 Original Publications

Following original publications by Siskind and McAllester form the basis of this manual:

Screaming Yellow Zonkers, 1991. Old Screamer manual, doesn't always hold true for Screamer 3.20 on which this "modern" Screamer is based, but still probably the most complete discussion of various operators and the overall design of Screamer outside of this manual.

Screamer: A Portable Efficient Implementation of Nondeterministic Common Lisp, 1993. A paper describing the fundamentals of Screamer.

Nondeterministic Lisp as a Substrate for Constraint Logic Programming, 1993. A paper describing the constaints propagation features of Screamer.

2 Overview

2.1 Important Note on Packages

Screamer shadows `defun`, and a couple of other symbols.

Examples in this manual are expected to be entered in the package `SCREAMER-USER`, which has the correct `defun`.

Packages using Screamer are best defined using [Macro define-screamer-package], page 30, which is like `defpackage` using `CL` and `SCREAMER` packages, but does the additional shadowing imports.

This is however by no means necessary: you can also explicitly use `screamer::defun`.

2.2 Choice-Points, Failure, and Backtracking

Screamer adds nondeterminism by providing the *choice-point* operator [Macro either], page 9 and the *failure* operator [Function fail], page 9.

A choice-point is a point in program where more than one thing can happen. When a choice-point is encountered, one of the possibilities occurs, and execution continues. Should a failure subsequently occur, the system backtracks to the last choice-point where multiple possibilities were still present, and tries again.

Backtracking is controlled by a form providing a nondeterministic context, eg. [Macro all-values], page 9, [Macro one-value], page 9, or [Macro for-effects], page 10.

```
(all-values
  (let ((x (either 1 2 3 4)))
    (if (oddp x)
        x
        (fail)))) ; => (1 3)
```

At first `(either 1 2 3 4)` evaluates to 1, which is `oddp`, so `all-values` receives it and sets about producing the next value. Now `either` returns 2, which isn't `oddp`. Hence `fail` is called causing the system to backtrack.

Starting again from the choice-point 3 is produced, which is `oddp`, and is received by `all-values`, which in turn requests the next value. Now `either` returns 4, which again causes `fail` to backtrack.

Since the only choice-point available cannot produce any more alternatives, control passes back to `all-values` which returns the collected values.

Had we wanted only one answer instead of an enumeration of all possible answers we could have used `one-value` instead of `all-values`.

If you're familiar with Prolog, `all-values` and `one-value` are analogous to Prolog's `bagof` and `cut` primitives.

2.3 Generators

Given `either` and `fail` we can write functions returning arbitrary sequences of nondeterministic values. Such functions are called *generators*, and by convention have names starting with `a-` or `an-`.

Consider for example `an-integer-between`:

```
;;; Screamer already provides an-integer-between, so we'll
;;; call this by a different name.
(defun an-int-between (min max)
  (if (> min max)
      (fail)
      (either min (an-int-between (1+ min) max))))

(all-values (an-int-between 41 43)) ; => (41 42 43)
```

Called with two integers, this function produces nondeterministic values in the given range – finally backtracking to a previous choice-point when all possibilities have been exhausted.

Given `an-integer-between` and `fail` we can write eg. a generator for square numbers:

```
;;; Square numbers are numbers produced by squaring an integer.
(defun a-square-number (min max)
  (let* ((x (an-integer-between 0 max))
         (square (* x x)))
    (if (<= min square max)
        square
        (fail))))

(all-values (a-square-number 12 80)) ; => (16 25 36 49 64)
```

We're not restricted to numbers, of course. Writing a generator for potential comedy duos works just the same:

```
(defun a-comedic-actor ()
  (list (either :tall :short) (either :thin :fat)))

(defun a-comedy-duo ()
  (let ((a (a-comedic-actor))
        (b (a-comedic-actor)))
    (if (or (eq (first a) (first b))
            (eq (second a) (second b)))
        (fail)
        (list a :and b))))

(one-value (a-comedy-duo)) ; => ((:TALL :THIN) :AND (:SHORT :FAT))
```

2.4 Side-Effects

What should happen to side-effects when a nondeterministic function backtracks? It depends. Some side-effects should be retained, and some undone – and it is impossible for the system to know in general what is the right choice in a given case.

Screamer is able to undo effects of `setf` and `setq` (including calls to user-defined setf-functions), but cannot undo calls to other functions with side-effects such as `set`, `rplaca`, or `sort`.

By default all side-effects are retained:

```
(let ((n 0))
  (list :numbers
        (all-values (let ((x (an-integer-between 0 3)))
                      (incf n)
                      x))
        :n n)) ; => (:NUMBERS (0 1 2 3) :N 4)
```

Macros [Macro local], page 12 and [Macro global], page 12 can be used to turn undoing of side-effects on and off lexically.

```
(let ((m 0)
      (n 0))
  (list :numbers
        (all-values (let ((x (an-integer-between 0 3)))
                      (local
                        (incf n)
                        (global
                          (incf m)))
                      x))
        :m m
        :n n)) ; => (:NUMBERS (0 1 2 3) :M 4 :N 0)
```

2.5 Constraint Propagation

In addition to nondeterminism via backtracking as discussed so far, Screamer also provides for forward constraint propagation via *logic variables* constructed using [Function make-variable], page 14. Screamer provides a variety of primitives for constraining variables.

By convention suffix `v` is used to denote operators that accept (and potentially return) variables in addition to values. Any `foov` is generally just like `foo`, except its arguments can also be logic variables, and that it may assert facts about them and will possibly return another variable.

The operator [Macro assert!], page 14 is the primary tool about asserting facts about variables.

Expression such as `(foov myvar)` typically returns another variable depending on `myvar`, which can be constrained to be true using `assert!`.

Operator [Function bound?], page 15 can be used to test if variable is bound to a specific value, and [Function value-of], page 15 can be used to obtain its value.

```
;;; Make a variable
(defparameter *v* (make-variable "The Answer"))

;;; It is initially unconstrained.
*v* ; => ["The Answer"]

;;; Constrain it to be an integer.
(assert! (integerpv *v*))

*v* ; => ["The Answer" integer]
```

```
;;; Constrain 40 to be 2 less than *v*
(assert! (=v 40 (-v *v* 2)))

;;; And we have our answer.
*v* ; => 42

;;; However, there's a catch. Even though *v* printed as 42 above, *v*
;;; is still a variable. This is convenience for working in the REPL:
;;; unbound variables print using square brackets, bound ones print
;;; as their value.
(type-of *v*) ; => SCREAMER::VARIABLE

;;; To obtain its value:
(value-of *v*) ; => 42
```

Assertions – and constraint operators in general – can cause failure and backtracking, in which case constraints from the last attempt are undone.

This allows us to search the solution space using backtracking:

```
(defparameter *x* (make-variable "X"))
(defparameter *y* (make-variable "Y"))

(assert! (integerpv *x*))
(assert! (integerpv *y*))

(assert! (=v 0 (+v *x* *y* 42)))

(all-values (let ((x (an-integer-between -50 -30))
                  (y (an-integer-between 2 5)))
              (assert! (=v *x* x))
              (assert! (=v *y* y))
              (list x y))) ; => ((-47 5) (-46 4) (-45 3) (-44 2))
```

A possibly less intuitive, but usually more efficient method is to assert range constraints as variables instead of nondeterministic values, and force a solution:

```
(assert! (=v *x* (an-integer-betweenv -50 -30)))
(assert! (=v *y* (an-integer-betweenv 2 5)))

(all-values
  (solution (list *x* *y*)
            (static-ordering #'linear-force)))
; => ((-47 5) (-46 4) (-45 3) (-44 2))
```

In this case backtracking occurs only inside [Function solution], page 27, when the system is trying to apply different solution to the given constraints, whereas in the first one we backtracked over the entire `let`.

2.6 Current Limitations

Screamer is implemented using a code-walker, which does not unfortunately currently support the full ANSI Common Lisp.

2.6.1 Not Supported At All

Following special operators signal an error if they appear in code processed by the code walker:

- load-time-value
- symbol-macrolet
- macrolet

2.6.2 Limited Support

Following special operators are accepted, but they cannot contain nondeterministic forms:

- progv
- unwind-protect
- catch

Additionally, functions defined using `flet` and `labels` are not in nondeterministic context, even if the surrounding context is nondeterministic.

2.6.3 Limitations in Undoing Side-Effects

Undoing side-effects via `local` is reliable only if the `setf` and `setq` forms are lexically apparent:

```
(local (incf (foo)))
```

may or may not work as expected, depending on how `foo` is implemented. If `(incf (foo))` expands using eg. `set-foo`, the code-walker will not notice the side-effect.

Undoing side-effects via `local` when there is no prior value might not work as expected, depending on the implementation of the place:

- If reading a non-existent value causes an error to be signalled, the initial assignment inside `local` will cause that to happen.

  Example: assignment to an unbound variable inside `local` signals an error.

- If reading a non-existent value causes a marker object (eg. `nil`) to be returned, undoing the side-effect means assigning the marker object back to the place.

  Example: undoing `(setf gethash)` of a previously unknown key will cause `nil` to be stored in the table instead of removing the new key and its value entirely via `remhash`.

3 Examples

3.1 Einstein's Riddle

Solving the "Einstein's Riddle" using nondeterministic features of Screamer, ie. backtracking search.

[HTML] [Source]

3.2 The Zebra Puzzle

Solving the "The Zebra Puzzle", using forward constraint propagation features of Screamer.

(This puzzle is virtually identical to "Einstein's Riddle", but the solution is very different.)

[HTML] [Source]

3.3 The Sudoku Puzzle

Solving a sudoku puzzle using forward constraint propagation features of Screamer.

[HTML] [Source]

4 Dictionary

4.1 Nondetermism

`either` *&body alternatives* [Macro]

Nondeterministically evaluates and returns the value of one of its `alternatives`.

`either` takes any number of arguments. With no arguments, `(either)` is equivalent to `(fail)` and is thus deterministic. With one argument, (EITHER X) is equivalent to `x` itself and is thus deterministic only when `x` is deterministic. With two or more argument it is nondeterministic and can only appear in a nondeterministic context.

It sets up a choice-point and evaluates the first `alternative` returning its values. When backtracking follows to this choice-point, the next `alternative` is evaluated and its values are returned. When no more `alternatives` remain, the current choice-point is removed and backtracking continues to the next most recent choice-point.

`fail` [Function]

Backtracks to the most recent choice-point.

`fail` is deterministic function and thus it is permissible to reference `#'fail`, and write `(funcall #'fail)` or `(apply #'fail)`.

Calling `fail` when there is no choice-point to backtrack to signals an error.

`trail` *function* [Function]

When called in non-deterministic context, adds `function` to the trail. Outside non-deterministic context does nothing.

Functions on the trail are called when unwinding from a nondeterministic selection (due to either a normal return, or calling `fail`.)

`all-values` *&body body* [Macro]

Evaluates `body` as an implicit `progn` and returns a list of all of the nondeterministic values yielded by the it.

These values are produced by repeatedly evaluating the body and backtracking to produce the next value, until the body fails and yields no further values.

Accordingly, local side effects performed by the body while producing each value are undone before attempting to produce subsequent values, and all local side effects performed by the body are undone upon exit from `all-values`.

Returns a list containing `nil` if `body` is empty.

An `all-values` expression can appear in both deterministic and nondeterministic contexts. Irrespective of what context the `all-values` appears in, the `body` is always in a nondeterministic context. An `all-values` expression itself is always deterministic.

`all-values` is analogous to the 'bagof' primitive in Prolog.

`one-value` *form &optional default* [Macro]

Returns the first nondeterministic value yielded by `form`.

No further execution of `form` is attempted after it successfully returns one value.

If `form` does not yield any nondeterministic values (i.e. it fails) then `default` is evaluated and its value returned instead. `default` defaults to `(fail)` if not present.

Local side effects performed by `form` are undone when `one-value` returns, but local side effects performed by `default` are not undone when `one-value` returns.

A `one-value` expression can appear in both deterministic and nondeterministic contexts. Irrespective of what context the `one-value` appears in, `form` is always in a nondeterministic context, while `default` is in whatever context the `one-value` form appears.

A `one-value` expression is nondeterministic if `default` is present and is nondeterministic, otherwise it is deterministic.

If `default` is present and nondeterministic, and if `form` fails, then it is possible to backtrack into the `default` and for the `one-value` form to nondeterministically return multiple times. `one-value` is analogous to the cut primitive ('!') in Prolog.

`for-effects` *&body body* [Macro]

Evaluates `body` as an implicit `progn` in a nondeterministic context and returns `nil`.

The body is repeatedly backtracked to its first choice-point until the body fails.

Local side effects performed by `body` are undone when `for-effects` returns.

A `for-effects` expression can appear in both deterministic and nondeterministic contexts. Irrespective of what context the `for-effects` appears in, `body` are always in a nondeterministic context. A `for-effects` expression is is always deterministic.

`ith-value` *i form &optional default* [Macro]

Returns the Ith nondeterministic value yielded by `form`.

I must be an integer. The first nondeterministic value yielded by `form` is numbered zero, the second one, etc. The Ith value is produced by repeatedly evaluating `form`, backtracking through and discarding the first I values and deterministically returning the next value produced.

No further execution of `form` is attempted after it successfully yields the desired value.

If `form` fails before yielding both the I values to be discarded, as well as the desired Ith value, then `default` is evaluated and its value returned instead. `default` defaults to `(fail)` if not present.

Local side effects performed by `form` are undone when `ith-value` returns, but local side effects performed by `default` and by I are not undone when `ith-value` returns.

An `ith-value` expression can appear in both deterministic and nondeterministic contexts. Irrespective of what context the `ith-value` appears in, `form` is always in a nondeterministic context, while `default` and I are in whatever context the `ith-value` appears in.

An `ith-value` expression is nondeterministic if `default` is present and is nondeterministic, or if I is nondeterministic. Otherwise it is deterministic.

If `default` is present and nondeterministic, and if `form` fails, then it is possible to backtrack into the `default` and for the `ith-value` expression to nondeterministically return multiple times.

If I is nondeterministic then the `ith-value` expression operates nondeterministically on each value of I. In this case, backtracking for each value of `form` and `default` is nested in, and restarted for, each backtrack of I.

`print-values` *&body body* [Macro]

Evaluates `body` as an implicit `progn` and prints each of the nondeterministic values yielded by it using `print`.

After each value is printed, the user is queried as to whether or not further values are desired. These values are produced by repeatedly evaluating the body and backtracking to produce the next value, until either the user indicates that no further values are desired or until the body fails and yields no further values.

Returns the last value printed.

Accordingly, local side effects performed by the body while producing each value are undone after printing each value, before attempting to produce subsequent values, and all local side effects performed by the body are undone upon exit from `print-values`, either because there are no further values or because the user declines to produce further values.

A `print-values` expression can appear in both deterministic and nondeterministic contexts. Irrespective of what context the `print-values` appears in, the `body` are always in a nondeterministic context. A `print-values` expression itself is always deterministic.

`print-values` is analogous to the standard top-level user interface in Prolog.

`possibly?` *&body body* [Macro]

Evaluates `body` as an implicit `progn` in nondeterministic context, returning true if the body ever yields true.

The body is repeatedly backtracked as long as it yields `nil`. Returns the first true value yielded by the body, or `nil` if body fails before yielding true.

Local side effects performed by the body are undone when `possibly?` returns.

A `possibly?` expression can appear in both deterministic and nondeterministic contexts. Irrespective of what context the `possibly?` appears in, its body is always in a nondeterministic context. A `possibly?` expression is always deterministic.

`necessarily?` *&body body* [Macro]

Evaluates `body` as an implicit `progn` in nondeterministic context, returning true if the body never yields false.

The body is repeatedly backtracked as long as it yields true. Returns the last true value yielded by the body if it fails before yielding `nil`, otherwise returns `nil`.

Local side effects performed by the body are undone when `necessarily?` returns.

A `necessarily?` expression can appear in both deterministic and nondeterministic contexts. Irrespective of what context the `necessarily?` appears in, its body is always in a nondeterministic context. A `necessarily?` expression is always deterministic.

`global` *&body body* [Macro]

Evaluates `body` in the same fashion as `progn` except that all `setf` and `setq` forms lexically nested in its body result in global side effects which are not undone upon backtracking.

Note that this affects only side effects introduced explicitly via `setf` and `setq`. Side effects introduced by Common Lisp builtin functions such as `rplaca` are always global anyway.

`local` and `global` may be nested inside one another. The nearest lexically surrounding one determines whether or not a given `setf` or `setq` results in a local or global side effect.

Side effects default to be global when there is no surrounding `local` or `global` expression. Global side effects can appear both in deterministic as well as nondeterministic contexts. In nondeterministic contexts, `global` as well as `setf` are treated as special forms rather than macros. This should be completely transparent to the user.

`local` *&body body* [Macro]

Evaluates `body` in the same fashion as `progn` except that all `setf` and `setq` forms lexically nested in its body result in local side effects which are undone upon backtracking.

This affects only side effects introduced explicitly via `setf` and `setq`. Side effects introduced by either user defined functions or builtin Common Lisp functions such as `rplaca` are always global.

Behaviour of side effects introduced by macro-expansions such as `incf` depends on the exact macro-expansion. If `(incf (foo))` expands using eg. `set-foo`, `local` is unable to undo the side-effect.

`local` cannot distinguish between initially uninitialized and intialized places, such as unbound variables or hash-table keys with no prior values. As a result, an attempt to assign an unbound variable inside `local` will signal an error due to the system's attempt to first read the variable. Similarly, undoing a `(setf gethash)` when the key did not previously exist in the table will insert a `nil` into the table instead of doing a `remhash`. Easiest way to work around this is by using `trail`.

`local` and `global` may be nested inside one another. The nearest lexically surrounding one determines whether or not a given `setf` or `setq` results in a local or global side effect.

Side effects default to be global when there is no surrounding `local` or `global` expression. Local side effects can appear both in deterministic as well as nondeterministic contexts though different techniques are used to implement the trailing of prior values for restoration upon backtracking. In nondeterministic contexts, `local` as well as `setf` are treated as special forms rather than macros. This should be completely transparent to the user.

`a-boolean` [Function]

Equivalent to `(either t nil)`.

`a-member-of` *sequence* [Function]
Nondeterministically returns an element of `sequence`. The elements are returned in the order that they appear in `sequence`. The `sequence` must be either a list or a vector.

`an-integer` [Function]
Generator yielding integers in sequence 0, 1, -1, 2, -2, ...

`an-integer-above` *low* [Function]
Generator yielding integers starting from `low` and continuing sequentially in increasing direction.

`an-integer-below` *high* [Function]
Generator yielding integers starting from `high` and continuing sequentially in decreasing direction.

`an-integer-between` *low high* [Function]
Nondeterministically returns an integer in the closed interval [`low`, `high`]. The results are returned in ascending order. Both `low` and `high` must be integers. Fails if the interval does not contain any integers.

`apply-nondeterministic` *function &rest arguments* [Function]
Analogous to the `cl:apply`, except `function` can be either a nondeterministic function, or an ordinary deterministic function.

You must use `apply-nondeterministic` to apply a nondeterministic function. An error is signalled if a nondeterministic function object is used with `cl:apply`.

You can use `apply-nondeterministic` to apply either a deterministic or nondeterministic function, though even if all of the `arguments` are deterministic and `function` is a deterministic function object, the call expression will still be nondeterministic (with presumably a single value), since it is impossible to determine at compile time that a given call to `apply-nondeterministic` will be passed only deterministic function objects for function.

`funcall-nondeterministic` *function &rest arguments* [Function]
Analogous to `cl:funcall`, except `function` can be either a nondeterministic function, or an ordinary determinisitic function.

You must use `funcall-nondeterministic` to funcall a nondeterministic function. An error is signalled if you attempt to funcall a nondeterministic function object with `cl:funcall`.

You can use `funcall-nondeterministic` to funcall either a deterministic or nondeterministic function, though even if all of the `arguments` are deterministic and `function` is a deterministic function object, the call expression will still be nondeterministic (with presumably a single value), since it is impossible to determine at compile time that a given call to `funcall-nondeterministic` will be passed only deterministic function objects for function.

`multiple-value-call-nondeterministic` *function-form &rest values-forms* [Function]
Analogous to the `cl:multiple-value-call`, except `function-form` can evaluate to either a nondeterministic function, or an ordinary deterministic function.

You must use `multiple-value-call-nondeterministic` to multiple-value-call a nondeterministic function. An error is signalled if a nondeterministic function object is used with `cl:multiple-value-call`.

You can use `multiple-value-call-nondeterministic` to call either a deterministic or nondeterministic function, though even if all of the `values-forms` are deterministic and `function-form` evaluates to a deterministic function object, the call expression will still be nondeterministic (with presumably a single value), since it is impossible to determine at compile time that a given call to `multiple-value-call-nondeterministic` will be passed only deterministic function objects for function.

While `multiple-value-call-nondeterministic` appears to be a function, it is really a special-operator implemented by the code-walkers processing nondeterministic source contexts.

`nondeterministic-function?` *x* [Function]

Returns `t` if `x` is a nondeterministic function and `nil` otherwise.

`#'foo` returns a nondeterministic function object iff it is used in nondeterminisitc context and `foo` is either a nondeterministic `lambda` form, or the name of a nondeterministic function defined using `screamer::defun`.

Currently, if `foo` is a nondeterministic function defined using `screamer::defun`, `#'foo` and `(symbol-function 'foo)` in deterministic context will return an ordinary deterministic Common Lisp function, which will signal an error at runtime.

`when-failing` (**&body** *failing-forms*) *&body body* [Macro]

Whenever `fail` is called during execution of `body`, executes `failing-forms` before unwinding.

`count-failures` *&body body* [Macro]

Executes `body` keeping track of the number of times `fail` has been called without unwinding from `body`. After `body` completes, reports the number of failures to `*standard-output*` before returning values from `body`.

4.2 Constraints

4.2.1 Basics

`make-variable` *&optional name* [Function]

Creates and returns a new variable. Variables are assigned a name which is only used to identify the variable when it is printed. If the parameter `name` is given then it is assigned as the name of the variable. Otherwise, a unique name is assigned. The parameter `name` can be any Lisp object.

`assert!` *x* [Macro]

Restricts `x` to `t`. No meaningful result is returned. The argument `x` can be either a variable or a non-variable.

This assertion may cause other assertions to be made due to noticers attached to `x`.

A call to `assert!` fails if `x` is known not to equal `t` prior to the assertion or if any of the assertions performed by the noticers result in failure.

Except for the fact that one cannot write `#'assert!`, `assert!` behaves like a function, even though it is implemented as a macro.

The reason it is implemented as a macro is to allow a number of compile time optimizations. Expressions like `(assert! (notv x))`, `(assert! (numberpv x))` and `(assert! (notv (numberv x)))` are transformed into calls to functions internal to Screamer which eliminate the need to create the boolean variable(s) normally returned by functions like `notv` and `numberpv`. Calls to the functions `numberpv`, `realpv`, `integerpv`, `memberv`, `booleanpv`, `=v`, `<v`, `<=v`, `>v`, `>=v`, `/=v`, `notv`, `funcallv`, `applyv` and `equalv` which appear directly nested in a call to `assert!`, or directly nested in a call to `notv` which is in turn directly nested in a call to `assert!`, are similarly transformed.

`value-of` *x* [Function]

Returns `x` if `x` is not a variable. If `x` is a variable then `value-of` dereferences `x` and returns the dereferenced value. If `x` is bound then the value returned will not be a variable. If `x` is unbound then the value returned will be a variable which may be `x` itself or another variable which is shared with `x`.

`apply-substitution` *x* [Function]

If `x` is a `cons`, or a variable whose value is a `cons`, returns a freshly consed copy of the tree with all variables dereferenced. Otherwise returns the value of `x`.

`bound?` *x* [Function]

Returns `t` if `x` is not a variable or if `x` is a bound variable. Otherwise returns `nil`. `bound?` is analogous to the extra-logical predicates 'var' and 'nonvar' typically available in Prolog.

`ground?` *x* [Function]

The primitive `ground?` is an extension of the primitive `bound?` which can recursively determine whether an entire aggregate object is bound. Returns `t` if `x` is bound and either the value of `x` is atomic or a `cons` tree where all atoms are bound.

Otherwise returns nil.

`applyv` *f x &rest xs* [Function]

`f` must be a deterministic function. If all arguments `x` are bound, returns the result of calling `f` on the dereferenced values of spread arguments.

Otherwise returns a fresh variable `v`, constrained to be equal to the result of calling `f` on the dereferenced values of arguments.

Additionally, if all but one of `v` and the argument variables become known, and the remaining variable has a finite domain, then that domain is further restricted to be consistent with other arguments.

`funcallv` *f &rest x* [Function]

`f` must be a deterministic function. If all arguments `x` are bound, returns the result of calling `f` on the dereferenced values of arguments.

Otherwise returns a fresh variable `v`, constrained to be equal to the result of calling `f` on the dereferenced values of arguments.

Additionally, if all but one of `v` and the argument variables become known, and the remaining variable has a finite domain, then that domain is further restricted to be consistent with other arguments.

`equalv` *x y* [Function]

Returns `t` if the aggregate object `x` is known to equal the aggregate object `y`, `nil` if the aggregate object `x` is known not to equal the aggregate object `y`, and a new boolean variable `v` if it is not known whether or not `x` equals `y` when `equalv` is called.

The values of `x`, `y` and `v` are mutually constraints via noticers so that `v` equals `t` if and only if `x` is known to equal `y` and `v` equals `nil` if and only if `x` is known not to equal `y`.

Noticers are attached to `v` as well as to all variables nested in both in `x` and `y`. When the noticers attached to variables nested in `x` and `y` detect that `x` is known to equal `y` they restrict `v` to equal `t`. Likewise, when the noticers attached to variables nested in `x` and `y` detect that `x` is known not to equal `y` they restrict `v` to equal `nil`.

Furthermore, if `v` later becomes known to equal `t` then `x` and `y` are unified. Likewise, if `v` later becomes known to equal `nil` then `x` and `y` are restricted to not be equal. This is accomplished by attaching noticers to the variables nested in `x` and `y` which detect when `x` becomes equal to `y` and fail.

The expression `(known? (equalv x y))` is analogous to the extra-logical predicate '==' typically available in Prolog.

The expression `(known? (notv (equalv x y)))` is analogous to the extra-logical predicate '\=' typically available in Prolog.

The expression `(assert! (equalv x y))` is analogous to Prolog unification.

The expression `(assert! (notv (equalv x y)))` is analogous to the disunification operator available in Prolog-II.

`template` *template* [Function]

Copies an aggregate object, replacing any symbol beginning with a question mark with a newly created variable.

If the same symbol appears more than once in `x`, only one variable is created for that symbol, the same variable replacing any occurrences of that symbol. Thus `(template '(a b (?c d ?e) ?e))` has the same effect as:

```
(LET ((?C (MAKE-VARIABLE))
      (?E (MAKE-VARIABLE)))
  (LIST 'A 'B (LIST C 'D E) E)).
```

This is useful for creating patterns to be unified with other structures.

4.2.2 Booleans

`a-booleanv` *&optional name* [Function]

Returns a boolean variable.

`booleanpv` *x* [Function]

The expression `(booleanpv x)` is an abbreviation for `(memberv x '(t nil))`.

`known?` *x* [Macro]

Restricts `x` to be a boolean. If `x` is equal to `t` after being restricted to be boolean, returns `t`. If `x` is equal to `nil` or if the value of `x` is unknown returns `nil`. The argument `x` can be either a variable or a non-variable.

The initial restriction to boolean may cause other assertions to be made due to noticers attached to `x`. A call to `known`? fails if `x` is known not to be boolean prior to the assertion or if any of the assertions performed by the noticers result in failure.

Restricting `x` to be boolean attaches a noticer on `x` so that any subsequent assertion which restricts `x` to be non-boolean will fail.

Except for the fact that one cannot write `#'known`?, `known`? behaves like a function, even though it is implemented as a macro.

The reason it is implemented as a macro is to allow a number of compile time optimizations. Expressions like `(known? (notv x))`, `(known? (numberpv x))` and `(known? (notv (numberpv x)))` are transformed into calls to functions internal to Screamer which eliminate the need to create the boolean variable(s) normally returned by functions like `notv` and `numberv`. Calls to the functions `numberpv`, `realpv`, `integerpv`, `memberv`, `booleanpv`, `=v`, `<v`, `<=v`, `v`, `>=v`, `/=v`, `notv`, `funcallv`, `applyv` and `equalv` which appear directly nested in a call to `known`?, or directly nested in a call to `notv` which is in turn directly nested in a call to `known`?, are similarly transformed.

`decide` *x* [Macro]

Restricts `x` to a be boolean. After `x` is restricted a nondeterministic choice is made. For one branch, `x` is restricted to equal `t` and `(decide x)` returns `t` as a result. For the other branch, `x` is restricted to equal `nil` and `(decide x)` returns `nil` as a result. The argument `x` can be either a variable or a non-variable.

The initial restriction to boolean may cause other assertions to be made due to noticers attached to `x`. A call to `decide` immediately fails if `x` is known not to be boolean prior to the assertion or if any of the assertions performed by the noticers result in failure.

Restricting `x` to be boolean attaches a noticer on `x` so that any subsequent assertion which restricts `x` to be non-boolean will fail.

Except for implementation optimizations `(decide x)` is equivalent to:

```
(EITHER (PROGN (ASSERT! X) T) (PROGN (ASSERT! (NOTV X)) NIL))
```

Except for the fact that one cannot write `#'decide`, `decide` behaves like a function, even though it is implemented as a macro.

The reason it is implemented as a macro is to allow a number of compile time optimizations. Expressions like `(decide (notv x))`, `(decide (numberpv x))` and `(decide (notv (numberpv x)))` are transformed into calls to functions internal to Screamer which eliminate the need to create the boolean variable(s) normally returned by functions like notv and numberv. Calls to the functions `numberpv`, `realpv`, `integerpv`, `memberpv`, `booleanpv`, `=v`, `<v`, `<=v`, `>v`, `>=v`, `/=v`, `notv`, `funcallv`, `applyv` and `equalv` which appear directly nested in a call to decide, or directly nested in a call to `notv` which is in turn directly nested in a call to decide, are similarly transformed.

`notv` *x* [Function]

Restricts `x` to be a boolean.

Returns `t` if this restricts `x` to `nil`, and `t` if this restricts `x` to `nil`.

Otherwise returns a new boolean variable `v`. `v` and `x` are mutually constrained via noticers, so that if either is later known to equal `t`, the other is restricted to equal `nil` and vice versa.

Note that unlike `cl:not` `notv` does not accept arbitrary values as arguments: it fails if its argument is not `t`, `nil`, or variable that can be restricted to a boolean.

`andv` *&rest xs* [Function]

Restricts each argument to be boolean.

Returns `t` if called with no arguments, or if all arguments are known to equal `t` after being restricted to be boolean, and returns `nil` if any argument is known to equal `nil` after this restriction.

Otherwise returns a boolean variable `v`. The values of the arguments and `v` are mutually constrained:

- If any argument is later known to equal `nil` value of `v` becomes `nil`.
- If all arguments are later known to equal `t`, value of `v` becomes `t`.
- If value of `v` is later known to equal `t`, all arguments become `t`.
- If value of `v` is later known to equal `nil`, and all but one argument is known to be `t`, the remaining argument becomes `nil`.

Note that unlike `cl:and`, `andv` is a function and always evaluates all its arguments. Secondly, any non-boolean argument causes it to fail.

`orv` *&rest xs* [Function]

Restricts each argument to be boolean.

Returns `nil` if called with no arguments, or if all arguments are known to equal `nil` after being restructed to be boolean, and returns `t` if any argument is known to equal `t` after this restriction.

Otherwise returns a boolean variable `v`. The values of arguments and `v` are mutually constrained:

- If any argument is later known to equal `t`, value of `v` becomes `t`.
- If all arguments are later known to equal `nil`, value of `v` becomes `nil`.
- If value of `v` is later known to equal `nil`, all arguments become `nil`.
- If value of `v` is later known to equal `t`, and all but one argument is known to be `nil`, the remaining argument becomes `t`.

Note that unlike `cl:or`, `orv` is a function and always evaluates all its arguments. Secondly, any non-boolean argument causes it to fail.

`count-truesv` *&rest xs* [Function]

Constrains all its arguments to be boolean. If each argument is known, returns the number of `t` arguments. Otherwise returns a fresh constraint variable `v`.

`v` and arguments are mutually constrained:

- Lower bound of `v` is the number arguments known to be `t`.
- Upper bound of `v` is the number arguments minus the number of arguments known to be `nil`.
- If lower bound of `v` is constrained to be equal to number of arguments known to be `nil`, all arguments not known to be `nil` are constrained to be `t`.
- If Upper bound of `v` is constrained to be equal to number of arguments known to be `t`, all arguments not known to be `t` are constrained to be `nil`.

4.2.3 Sequences

`a-member-ofv` *values &optional name* [Function]
Returns a variable whose value is constrained to be one of `values`. `values` can be either a vector or a list designator.

`memberv` *x sequence* [Function]
Returns `t` if `x` is known to be a member of `sequence` (using the Common Lisp function `eql` as a test function), `nil` if `x` is known not to be a member of `sequence`, and otherwise returns a new boolean variable `v`.

When a new variable is created, the values of `x` and `v` are mutually constrained via noticers so that `v` is equal to `t` if and only if `x` is known to be a member of `sequence` and `v` is equal to `nil` if and only if `x` is known not to be a member of `sequence`.

- If `x` later becomes known to be a member of `sequence`, a noticer attached to `x` restricts v to equal `t`. Likewise, if `x` later becomes known not to be a member of `sequence`, a noticer attached to `x` restricts `v` to equal `nil`.
- If `v` ever becomes known to equal `t` then a noticer attached to `v` restricts `x` to be a member of `sequence`. Likewise, if `v` ever becomes known to equal `nil` then a noticer attached to `v` restricts `x` not to be a member of `sequence`.

The current implementation imposes two constraints on the parameter `sequence`. First, `sequence` must be bound when `memberv` is called. Second, `sequence` must not contain any unbound variables when `memberv` is called.

The value of parameter `sequence` must be a sequence, i.e. either a list or a vector.

4.2.4 Numbers

`a-numberv` *&optional name* [Function]
Returns a variable whose value is constained to be a number.

`a-realv` *&optional name* [Function]
Returns a real variable.

`a-real-abovev` *low &optional name* [Function]
Returns a real variable whose value is constrained to be greater than or equal to `low`.

`a-real-belowv` *high &optional name* [Function]
Returns a real variable whose value is constrained to be less than or equal to `high`.

`a-real-betweenv` *low high &optional name* [Function]
Returns a real variable whose value is constrained to be greater than or equal to low and less than or equal to high. If the resulting real variable is bound, its value is returned instead. Fails if it is known that low is greater than high at the time of call.

The expression `(a-real-betweenv low high)` is an abbreviation for:

```
(LET ((V (MAKE-VARIABLE)))
   (ASSERT! (REALPV V))
   (ASSERT! (>=V V LOW))
   (ASSERT! (<=V V HIGH))
   (VALUE-OF V))
```

`an-integerv` *&optional name* [Function]
Returns an integer variable.

`an-integer-abovev` *low &optional name* [Function]
Returns an integer variable whose value is constrained to be greater than or equal to `low`.

`an-integer-belowv` *high &optional name* [Function]
Returns an integer variable whose value is constrained to be less than or equal to `high`.

`an-integer-betweenv` *low high &optional name* [Function]
Returns an integer variable whose value is constrained to be greater than or equal to `low` and less than or equal to `high`. If the resulting integer variable is bound, its value is returned instead. Fails if it is known that there is no integer between `low` and `high` at the time of call.

The expression `(an-integer-betweenv low high)` is an abbreviation for:

```
(LET ((V (MAKE-VARIABLE)))
   (ASSERT! (INTEGERPV V))
   (ASSERT! (>=V V LOW))
   (ASSERT! (<=V V HIGH))
   (VALUE-OF v))
```

`numberpv` *x* [Function]
Returns `t` if `x` is known to be numeric, `nil` if `x` is known to be non-numeric, and otherwise returns a new boolean variable `v`.

The values of `x` and `v` are mutually constrained via noticers so that `v` is equal to `t` if and only if `x` is known to be numeric and `v` is equal to `nil` if and only if `x` is known to be non-numeric.

- If `x` later becomes known to be numeric, a noticer attached to `x` restricts `v` to equal `t`. Likewise, if `x` later becomes known to be non-numeric, a noticer attached to `x` restricts `v` to equal `nil`.
- If `v` ever becomes known to equal `t` then a noticer attached to `v` restricts `x` to be numeric. Likewise, if `v` ever becomes known to equal `nil` then a noticer attached to `v` restricts `x` to be non-numeric.

`realpv` *x* [Function]
Returns `t` if `x` is known to be real, `nil` if `x` is known to be non-real, and otherwise returns a new boolean variable `v`.

The values of `x` and `v` are mutually constrained via noticers so that `v` is equal to `t` if and only if `x` is known to be real and `v` is equal to `nil` if and only if `x` is known to be non-real.

- If `x` later becomes known to be real, a noticer attached to `x` restricts `v` to equal `t`. Likewise, if `x` later becomes known to be non-real, a noticer attached to `x` restricts `v` to equal `nil`.
- If `v` ever becomes known to equal `t` then a noticer attached to `v` restricts `x` to be real. Likewise, if `v` ever becomes known to equal `nil` then a noticer attached to `v` restricts `x` to be non-real.

`integerpv` *x* [Function]

Returns `t` if `x` is known to be integer valued, and `nil` if `x` is known be non-integer value.

If it is not known whether or not `x` is integer valued when `integerpv` is called then `integerpv` creates and returns a new boolean variable `v`.

The values of `x` and `v` are mutually constrained via noticers so that `v` is equal to `t` if and only if `x` is known to be integer valued, and `v` is equal to `nil` if and only if `x` is known to be non-integer valued.

If `x` later becomes known to be integer valued, a noticer attached to `x` restricts `v` to equal `t`. Likewise, if `x` later becomes known to be non-integer valued, a noticer attached to `x` restricts `v` to equal `nil`.

Furthermore, if `v` ever becomes known to equal `t` then a noticer attached to `v` restricts `x` to be integer valued. Likewise, if `v` ever becomes known to equal `nil` then a noticer attached to `v` restricts `x` to be non-integer valued.

`minv` *x &rest xs* [Function]

Constrains its arguments to be real. If called with a single argument, returns its value. If called with multiple arguments, behaves as if a combination of two argument calls:

```
(MINV X1 X2 ... Xn) == (MINV (MINV X1 X2) ... Xn)
```

If called with two arguments, and either is known to be less than or equal to the other, returns the value of that argument. Otherwise returns a real variable `v`, mutually constrained with the arguments:

- Minimum of the values of X1 and X2 is constrained to equal `v`. This includes constraining their bounds appropriately. If it becomes know that cannot be true. `fail` is called.
- If both arguments are integers, `v` is constrained to be an integer.

`maxv` *x &rest xs* [Function]

Constrains its arguments to be real. If called with a single argument, returns its value. If called with multiple arguments, behaves as if a combination of two argument calls:

```
(MAXV X1 X2 ... Xn) == (MAXV (MAXV X1 X2) ... Xn)
```

If called with two arguments, and either is known to be greater than or equal to the other, returns the value of that argument. Otherwise returns a real variable `v`, mutually constrained with the arguments:

- Maximum of the values of X1 and X2 is constrained to equal `v`. This includes constraining their bounds appropriately. If it becomes know that cannot be true. `fail` is called.
- If both arguments are integers, `v` is constrained to be an integer.

`+v` *&rest xs* [Function]

Constrains its arguments to be numbers. Returns 0 if called with no arguments. If called with a single argument, returns its value. If called with more than two arguments, behaves as nested sequence of two-argument calls:

```
(+V X1 X2 ... Xn) = (+V X1 (+V X2 (+V ...)))
```

When called with two arguments, if both arguments are bound, returns the sum of their values. If either argument is known to be zero, returns the value of the remaining argument. Otherwise returns number variable `v`.

- Sum of X1 and X2 is constrained to equal `v`. This includes constraining their bounds appropriately. If it becomes known that cannot be true, `fail` is called.
- If both arguments are known to be reals, `v` is constrained to be real.
- If both arguments are known to be integers, `v` is constained to be integer.
- If one argument is known to be a non-integer, and the other is known to be a real, `v` is constrained to be a non-integer.
- If one argument is known to be a non-real, and the other is known to be a real, `v` is constrained to be non-real.

Note: Numeric contagion rules of Common Lisp are not applied if either argument equals zero.

`-v` *x &rest xs* [Function]

Constrains its arguments to be numbers. If called with a single argument, behaves as if the two argument call:

```
(-V 0 X)
```

If called with more than two arguments, behaves as nested sequence of two-argument calls:

```
(-V X1 X2 ... Xn) = (-V X1 (-V X2 (-V ...)))
```

When called with two arguments, if both arguments are bound, returns the difference of their values. If X2 is known to be zero, returns the value of X1. Otherwise returns number variable `v`.

- Difference of X1 and X2 is constrained to equal `v`. This includes constraining their bounds appropriately. If it becomes known that cannot be true, `fail` is called.
- If both arguments are known to be reals, `v` is constrained to be real.
- If both arguments are known to be integers, `v` is constained to be integer.
- If one argument is known to be a non-integer, and the other is known to be a real, `v` is constrained to be a non-integer.
- If one argument is known to be a non-real, and the other is known to be a real, `v` is constrained to be non-real.

Note: Numeric contagion rules of Common Lisp are not applied if X2 equals zero.

`*v` *&rest xs* [Function]

Constrains its arguments to be numbers. If called with no arugments, returns 1. If called with a single argument, returns its value. If called with more than two arguments, behaves as nested sequence of two-argument calls:

```
(*V X1 X2 ... Xn) = (*V X1 (*V X2 (*V ...)))
```

When called with two arguments, if both arguments are bound, returns the product of their values. If either argument is known to equal zero, returns zero. If either argument is known to equal one, returns the value of the other. Otherwise returns number variable `v`.

- Product of X1 and X2 is constrained to equal `v`. This includes constraining their bounds appropriately. If it becomes known that cannot be true, `fail` is called.
- If both arguments are known to be reals, `v` is constrained to be real.
- If both arguments are known to be integers, `v` is constained to be integer.
- If `v` is known to be an integer, and either X1 or X2 is known to be real, both X1 and X2 are constrained to be integers.
- If `v` is known to be an reals, and either X1 or X2 is known to be real, both X1 and X2 are constrained to be reals.

Note: Numeric contagion rules of Common Lisp are not applied if either argument equals zero or one.

`/v` *x &rest xs* [Function]

Constrains its arguments to be numbers. If called with a single argument, behaves as the two argument call:

```
(/V 1 X)
```

If called with more than two arguments, behaves as nested sequence of two-argument calls:

```
(/V X1 X2 ... Xn) = (/V ... (/V (/V X1 X2) X3) ... Xn)
```

When called with two arguments, if both arguments are bound, returns the division of their values. If X1 is known to equal zero, returns 0. If X2 is known to equal zero, `fail` is called. If X2 is known to equal one, returns the value of X1. Otherwise returns number variable `v`.

- Division of X1 and X2 is constrained to equal `v`. This includes constraining their bounds appropriately. If it becomes known that cannot be true, `fail` is called.
- If both arguments are known to be reals, `v` is constrained to be real.
- If both arguments are known to be integers, `v` is constained to be integer.
- If `v` is known to be an integer, and either X1 or X2 is known to be real, both X1 and X2 are constrained to be integers.
- If `v` is known to be an reals, and either X1 or X2 is known to be real, both X1 and X2 are constrained to be reals.

Note: Numeric contagion rules of Common Lisp are not applied if X1 equals zero or X2 equals one.

`<v` *x &rest xs* [Function]

Returns a boolean value which is constrained to be `t` if each argument Xi is less than the following argument Xi+1 and constrained to be `nil` if some argument Xi is greater than or equal to the following argument Xi+1.

This function takes one or more arguments. All of the arguments are restricted to be real.

Returns `t` when called with one argument. A call such as `(<v x1 x2 ... xn)` with more than two arguments behaves like a conjunction of two argument calls:

```
(ANDV (<V X1 X2) ... (<V Xi Xi+1 ) ... (<V Xn-1 Xn))
```

When called with two arguments, returns `t` if X1 is known to be less than X2 at the time of call, `nil` if X1 is known to be greater than or equal to X2 at the time of call, and otherwise a new boolean variable `v`.

A real value X1 is known to be less than a real value X2 if X1 has an upper bound, X2 has a lower bound and the upper bound of X1 is less than the lower bound of X2.

A real value X1 is known to be greater than or equal to a real value X2 if X1 has a lower bound, X2 has an upper bound and the lower bound of X1 is greater than or equal to the upper bound of X2.

When a new variable is created, the values of X1, X2 and v are mutually constrained via noticers so that `v` is equal to `t` if and only if X1 is known to be less than X2 and `v` is equal to `nil` if and only if X1 is known to be greater than or equal to X2.

- If it later becomes known that X1 is less than X2, noticers attached to X1 and X2 restrict `v` to equal `t`. Likewise, if it later becomes known that X1 is greater than or equal to X2, noticers attached to X1 and X2 restrict `v` to equal `nil`.
- If `v` ever becomes known to equal `t` then a noticer attached to `v` restricts X1 to be less than X2. Likewise, if `v` ever becomes known to equal `nil` then a noticer attached to `v` restricts X1 to be greater than or equal to X2.

Restricting a real value X1 to be less than a real value X2 is performed by attaching noticers to X1 and X2. The noticer attached to X1 continually restricts the lower bound of X2 to be no lower than the upper bound of X1 if X1 has an upper bound. The noticer attached to X2 continually restricts the upper bound of X1 to be no higher than the lower bound of X2 if X2 has a lower bound. Since these restrictions only guarantee that X1 be less than or equal to X2, the constraint that X1 be strictly less than X2 is enforced by having the noticers fail when both X1 and X2 become known to be equal.

Restricting a real value X1 to be greater than or equal to a real value X2 is performed by an analogous set of noticers without this last equality check.

`<=v` *x &rest xs* [Function]

All arguments are constrained to be real. Returns `t` when called with one argument. A call such as `(<=v x1 x2 ... xn)` with more than two arguments behaves like a conjunction of two argument calls:

```
(ANDV (<=V X1 X2) ... (<=V Xi Xi+1) ... (<=V Xn-1 Xn))
```

When called with two arguments, returns `t` if X1 is know to be less than or equal to X2 at the time of the call, `nil` if X1 is known to be greater than X2, and otherwise a new boolean variable `v`.

Values of `v`, X1, and X2 are mutually constrained:

- `v` is equal to `t` iff X1 is known to be less than or equal to X2.
- `v` is equal to `nil` iff X2 is known to be greater than X2.
- If `v` is known to be `t`, X1 is constrained to be less than or equal to X2.
- If `v` is known to be `nil`, X1 is constrained to be greater than X2.

`=v` *x &rest xs* [Function]

Returns a boolean value which is constrained to be `t` if all of the arguments are numerically equal, and constrained to be `nil` if two or more of the arguments numerically differ.

This function takes one or more arguments. All of the arguments are restricted to be numeric.

Returns `t` when called with one argument. A call such as `(=v x1 x2 ... xn)` with more than two arguments behaves like a conjunction of two argument calls:

```
(ANDV (=V X1 X2) ... (=V Xi Xi+1) ... (=V Xn-1 Xn))
```

When called with two arguments, returns `t` if X1 is known to be equal to X2 at the time of call, `nil` if X1 is known not to be equal to X2 at the time of call, and a new boolean variable `v` if is not known if the two values are equal.

Two numeric values are known to be equal only when they are both bound and equal according to the Common Lisp function `=`.

Two numeric values are known not to be equal when their domains are disjoint. Furthermore, two real values are known not to be equal when their ranges are disjoint, i.e. the upper bound of one is greater than the lower bound of the other.

When a new variable is created, the values of X1, X2, and `v` are mutually constrained via noticers so that `v` is equal to `t` if and only if X1 is known to be equal to X2, and `v` is equal to `nil` if and only if X1 is known not to be equal to X2.

- If it later becomes known that X1 is equal to X2 noticers attached to X1 and X2 restrict `v` to equal `t`. Likewise if it later becomes known that X1 is not equal to X2 noticers attached to X1 and X2 restrict `v` to equal `nil`.
- If `v` ever becomes known to equal `t` then a noticer attached to `v` restricts X1 to be equal to X2. Likewise if `v` ever becomes known to equal `nil` then a noticer attached to `v` restricts X1 not to be equal to X2.
- If X1 is known to be real then the noticer attached to X2 continually restrict the upper bound of X1 to be no higher than the upper bound of X2 and the lower bound of X1 to be no lower than the lower bound of X2. Likewise for bounds of X1 if X2 is known to be real.

Restricting two values x1 and x2 to be equal is performed by attaching noticers to x1 and x2. These noticers continually restrict the domains of x1 and x2 to be equivalent sets (using the Common Lisp function `=` as a test function) as their domains are restricted.

Restricting two values X1 and X2 to not be equal is also performed by attaching noticers to X1 and X2. These noticers however do not restrict the domains or ranges of X1 or X2. They simply monitor their continually restrictions and fail when any assertion causes X1 to be known to be equal to X2.

`>=v` *x &rest xs* [Function]

All arguments are constrained to be real. Returns `t` when called with one argument. A call such as `(>=v x1 x2 ... xn)` with more than two arguments behaves like a conjunction of two argument calls:

```
(ANDV (>=V X1 X2) ... (>=V Xi Xi+1) ... (>=V Xn-1 Xn))
```

When called with two arguments, returns `t` if X1 is know to be greater than or equal to X2 at the time of the call, `nil` if X1 is known to be less than X2, and otherwise a new boolean variable `v`.

Values of `v`, X1, and X2 are mutually constrained:

- `v` is equal to `t` iff X1 is known to be greater than or equal to X2.
- `v` is equal to `nil` iff X2 is know to be less than X2.
- If `v` is known to be `t`, X1 is constrained to be greater than or equal to X2.
- If `v` is known to be `nil`, X1 is constrained to be less than X2.

`>v` *x &rest xs* [Function]

All arguments are constrained to be real. Returns `t` when called with one argument. A call such as `(>v x1 x2 ... xn)` with more than two arguments behaves like a conjunction of two argument calls:

```
(ANDV (> X1 X2) ... (> Xi Xi+1) ... (> Xn-1 Xn))
```

When called with two arguments, returns `t` if X1 is know to be greater than X2 at the time of the call, `nil` if X1 is known to be less than or equal to X2, and otherwise a new boolean variable `v`.

Values of `v`, X1, and X2 are mutually constrained:

- `v` is equal to `t` iff X1 is known to be greater than X2.
- `v` is equal to `nil` iff X2 is known to be less than or equal to X2.
- If `v` is known to be `t`, X1 is constrained to be greater than X2.
- If `v` is known to be `nil`, X1 is constrained to be less than or equal to X2.

`/=v` *x &rest xs* [Function]

Returns a boolean value which is constrained to be `t` if no two arguments are numerically equal, and constrained to be `nil` if any two or more arguments are numerically equal.

This function takes one or more arguments. All of the arguments are restricted to be numeric.

Returns `t` when called with one argument. A call such as `(/=v x1 x2 ... xn)` with more than two arguments behaves like a conjunction of two argument calls:

```
(ANDV (/=V X1 X2) ... (/=V X1 Xn)
      (/=V X2 X3) ... (/=V X2 Xn)
      ...
      (/=V Xi Xi+1 ... (/=V Xi Xn)
      ...
      (/=V Xn-1 xn))
```

When called with two arguments, returns `t` if X1 is known not to be equal to X2 at the time of call, `nil` if X1 is known to be equal to X2 at the time of call, and otherwise a new boolean variable `v`.

Two numeric values are known not to be equal when their domains are disjoint.

Two real values are known not to be equal when their ranges are disjoint, i.e. the upper bound of one is greater than the lower bound of the other.

Two numeric values are known to be equal only when they are both bound and equal according to the Common Lisp function `=`.

When a new variable is created, the values of X1, X2 and `v` are mutually constrained via noticers so that `v` is equal to `t` if and only if X1 is known not to be equal to X2 and `v` is equal to `nil` if and only if X1 is known to be equal to X2.

- If it later becomes known that X1 is not equal to X2, noticers attached to X1 and X2 restrict `v` to equal `t`. Likewise, if it later becomes known that X1 is equal to X2, noticers attached to X1 and X2 restrict `v` to equal `nil`.
- If `v` ever becomes known to equal `t` then a noticer attached to `v` restricts X1 to not be equal to X2. Likewise, if `v` ever becomes known to equal `nil` then a noticer attached to `v` restricts X1 to be equal to X2.

Restricting two values X1 and X2 to be equal is performed by attaching noticers to X1 and X2. These noticers continually restrict the domains of X1 and X2 to be equivalent sets (using the Common Lisp function `=` as a test function) as their domains are restricted. Furthermore, if X1 is known to be real then the noticer attached to X2 continually restrict the upper bound of X1 to be no higher than the upper bound of X2 and the lower bound of X1 to be no lower than the lower bound of X2. The noticer of X2 performs a symmetric restriction on the bounds of X1 if it is known to be real.

Restricting two values X1 and X2 to not be equal is also performed by attaching noticers to X1 and X2. These noticers however, do not restrict the domains or ranges of X1 or X2. They simply monitor their continually restrictions and fail when any assertion causes X1 to be known to be equal to X2.

4.2.5 Forcing Solutions

`solution` *arguments ordering-force-function* [Function]

`arguments` is a list of values. Typically it is a list of variables but it may also contain nonvariables.

The specified `ordering-force-function` is used to force each of the variables in list to be bound.

Returns a list of the values of the elements of list in the same order that they appear in list, irrespective of the forcing order imposed by the `ordering-force-function`.

The `ordering-force-function` can be any function which takes a list of values as its single argument that is guaranteed to force all variables in that list to be bound upon its return. The returned value of the `ordering-force-function` is ignored.

The user can construct her own `ordering-force-function` or use one of the following alternatives provided with Screamer:

```
(STATIC-ORDERING #'LINEAR-FORCE),
(STATIC-ORDERING #'DIVIDE-AND-CONQUER-FORCE),
(REORDER COST-FUN TERMINATE-TEST ORDER #'LINEAR-FORCE) and
(REORDER COST-FUN TERMINATE-TEST ORDER #'DIVIDE-AND-CONQUER-FORCE).
```

Future implementation of Screamer may provide additional forcing and ordering functions.

`static-ordering` *force-function* [Function]

Returns an ordering force function based on `force-function`.

The ordering force function which is returned is a nondeterministic function which takes a single argument `x`. This argument `x` can be a list of values where each value may be either a variable or a non-variable. The ordering force function applies the `force-function` in turn to each of the variables in `x`, in the order that they appear, repeatedly applying the `force-function` to a given variable until it becomes bound before proceeding to the next variable. The ordering force function does not return any meaningful result.

`force-function` is any (potentially nondeterministic) function which can be applied to a variable as its single argument with the stipulation that a finite number of repeated applications will force the variable to be bound. The `force-function` need not return any useful value.

Screamer currently provides two convenient force-functions, namely `#'linear-force` and `#'divide-and-conquer-force` though future implementations may provide additional ones. (The defined Screamer protocol does not provide sufficient hooks for the user to define her own force functions.)

`reorder` *cost-function terminate? order force-function* [Function]

Returns an ordering force function based on arguments.

The `force-function` is any (potentially nondeterministic) function which can be applied to a variable as its single argument with the stipulation that a finite number of repeated applications will force the variable to be bound. The `force-function` need not return any useful value.

The ordering force function which is returned is a nondeterministic function which takes a single argument `x`. This argument `x` can be a list of values where each value may be either a variable or a non-variable.

The ordering force function repeatedly selects a "best" variable using using `cost-function` and `order`. Eg. using `#'domain-size` and `#'<` as the `cost-function` and `order`, then the variable with the smallest domain will be forced first.

Function `terminate?` is then called with the determined cost of that variable, and unless it returns true, `force-function` is applied to that variable to force constrain it.

Process then iterates until all variables become bound or `terminate?` returns true.

The ordering force function does not return any meaningful result.

Screamer currently provides two convenient force-functions, namely #'linear-force and #'divide-and-conquer-force though future implementations may provide additional ones. (The defined Screamer protocol does not provide sufficient hooks for the user to define her own force functions.)

`linear-force` *x* [Function]

Returns `x` if it is not a variable. If `x` is a bound variable then returns its value.

If `x` is an unbound variable then it must be known to have a countable set of potential values. In this case `x` is nondeterministically restricted to be equal to one of the values

in this countable set, thus forcing `x` to be bound. The dereferenced value of `x` is then returned.

An unbound variable is known to have a countable set of potential values either if it is known to have a finite domain or if it is known to be integer valued.

An error is signalled if `x` is not known to have a finite domain and is not known to be integer valued.

Upon backtracking `x` will be bound to each potential value in turn, failing when there remain no untried alternatives.

Since the set of potential values is required only to be countable, not finite, the set of untried alternatives may never be exhausted and backtracking need not terminate. This can happen, for instance, when `x` is known to be an integer but lacks either an upper of lower bound.

The order in which the nondeterministic alternatives are tried is left unspecified to give future implementations leeway in incorporating heuristics in the process of determining a good search order.

`divide-and-conquer-force` *variable* [Function]

Returns `x` if `x` is not a variable. If `x` is a bound variable then returns its value. Otherwise implements a single binary-branching step of a divide-and-conquer search algorithm. There are always two alternatives, the second of which is tried upon backtracking.

If it is known to have a finite domain `d` then this domain is split into two halves and the value of `x` is nondeterministically restricted to be a member one of the halves. If `x` becomes bound by this restriction then its value is returned. Otherwise, `x` itself is returned.

If `x` is not known to have a finite domain but is known to be real and to have both lower and upper bounds then nondeterministically either the lower or upper bound is restricted to the midpoint between the lower and upper bound. If `x` becomes bound by this restriction then its dereferenced value is returned. Otherwise, `x` itself is returned.

An error is signalled if `x` is not known to be restricted to a finite domain and either is not known to be real or is not known to have both a lower and upper bound.

When the set of potential values may be infinite, users of `divide-and-conquer-force` may need to take care to fail when the range size of the variable becomes too small, unless other constraints on it are sufficient to guarentee failure.

The method of splitting the domain into two halves is left unspecified to give future implementations leeway in incorporating heuristics in the process of determining a good search order. All that is specified is that if the domain size is even prior to splitting, the halves are of equal size, while if the domain size is odd, the halves differ in size by at most one.

`domain-size` *x* [Function]

Returns the domain size of `x`.

If `x` is an integer variable with an upper and lower bound, its domain size is the one greater than the difference of its bounds. Eg. [integer 1:2] has domain size 2.

If `x` is a variable with an enumerated domain, its domain size is the size of that domain.

If `x` is a `cons`, or a variable whose value is a `cons`, its domain size is the product of the domain sizes of its `car` and `cdr`.

Other types of unbound variables have domain size `nil`, whereas non-variables have domain size of 1.

`range-size` *x* [Function]

Returns the range size of `x`. Range size is the size of the range values of `x` may take.

If `x` is an integer or a bound variable whose value is an integer, it has the range size 0. Reals and bound variables whose values are reals have range size 0.0.

Unbound variables known to be reals with an upper and lower bound have a range size the difference of their upper and lower bounds.

Other types of objects and variables have range size `nil`.

4.3 Miscellany

`define-screamer-package` *defined-package-name &body options* [Macro]

Convenience wrapper around `defpackage`. Passes its argument directly to `defpackage`, and automatically injects two additional options:

```
(:shadowing-import-from :screamer :defun :multiple-value-bind :y-or-n-p)
(:use :cl :screamer)
```

`best-value` *form1 objective-form &optional form2* [Macro]

First evaluates `objective-form`, which should evaluate to constraint variable `v`.

Then repeatedly evaluates FORM1 in non-deterministic context till it fails. If previous round of evaluation produced an upper bound `b` for `v`, the during the next round any change to `v` must provide an upper bound higher than `b`, or that that change fails.

If the last successful evaluation of `form` produced an upper bound for `v`, returns a list of two elements: the the primary value of FORM1 from that round, and the upper bound of `v`.

Otherwise if FORM2 is provided, returns the result of evaluating it, or else calls fails.

Note: this documentation string is entirely reverse-engineered. Lacking information on just how `best-value` was intended to work, it is hard to tell what is a bug, an accident of implementation, and what is a feature. If you have any insight into `best-value`, please send email to nikodemus@random-state.net.

`booleanp` *x* [Function]

Returns true iff `x` is `t` or `nil`.

`count-trues` *&rest xs* [Function]

Returns the number of time a non-NIL value occurs in its arguments.

`purge` *function-name* [Function]

Removes any information about `function-name` from Screamer's who-calls database.

`unwind-trail` [Function]

`deprecated`.

Calls all functions installed using `trail`, and removes them from the trail.

Using `unwind-trail` is dangerous, as `trail` is used by Screamer internally to eg. undo effects of local assignments -- hence users should never call it. It is provided at the moment only for backwards compatibility with classic Screamer.

`unwedge-screamer` [Function]

Removes any information about all user defined functions from Screamer's who-calls database.

`*dynamic-extent?*` [Variable]

Set to `t` to enable the dynamic extent optimization, `nil` to disable it. Default is platform dependent.

`*screamer-version*` [Variable]

The version of Screamer which is loaded. This is currently still 3.20, while we're considering how to express versions for this copy of Screamer in the future.

`*iscream?*` [Variable]

`t` if Screamer is running under ILisp/GNUEmacs with iscream.el loaded.

`*maximum-discretization-range*` [Variable]

Discretize integer variables whose range is not greater than this number. Discretize all integer variables if `nil`. Must be an integer or `nil`.

`*minimum-shrink-ratio*` [Variable]

Ignore propagations which reduce the range of a variable by less than this ratio.

`*strategy*` [Variable]

Strategy to use for `funcallv` and `applyv`. Either `:gfc` for Generalized Forward Checking, or `:ac` for Arc Consistency. Default is `:gfc`.

Appendix A Function Index

M

N

O

P

R

S

T

U

V

W

Appendix B Variable Index