Screaming Yellow Zonkers

Jeffrey Mark Siskind*
M. I. T. Artificial Intelligence Laboratory
545 Technology Square, Room NE43–800b
Cambridge MA 02139
617/253–5659
internet: Qobi@AI.MIT.EDU

DRAFT of Sunday, 29–September–1991 11:32:08 EDT

Abstract

Nondeterministic LISP is a variant of LISP with a nondeterministic choice operator. This manual describes an efficient implementation of nondeterministic LISP called SCREAMER. SCREAMER is implemented as a fully portable macro package built on top of COMMON LISP. SCREAMER functions inter-operate in the same environment as ordinary LISP functions and a large subset of COMMON LISP is available when writing SCREAMER functions. In addition to the nondeterministic choice operator, SCREAMER provides a forward checking constraint propagation facility as well. Together they make SCREAMER an efficient mechanism for building search programs.

TOPIC AREAS: nondeterministic search, AI programming languages

Caution! This product may drive you zonkers!
From the box of Screaming Yellow Zonkers

1 Introduction

2 Nondeterministic Expressions, Functions and Contexts

In order to provide the ability for backtracking, SCREAMER compiles nondeterministic functions differently than deterministic functions. Accordingly, it also compiles calls to nondeterministic functions differently than calls to deterministic functions. For the most part the user need not be aware of this difference as SCREAMER automatically determines whether or not a function is deterministic and compiles code appropriately. There are certain restriction however. In order to explain these restrictions some terminology must be defined.

A *deterministic expression* is an expression which SCREAMER can determine to yield at most one value. A *nondeterministic expression* is an expression which SCREAMER cannot determine to be deterministic. An expression might yield no more than one value when actually evaluated and still be classified as nondeterministic if SCREAMER cannot determine at compile time that it is deterministic. SCREAMER classifies the following expressions as deterministic:

*Need thanks.

- A call to the `map-values` primitive.

- A call to the `one-value` primitive provided that the *default-expression* is either absent or a deterministic expression.

- A call to the `all-values` primitive.

- A call to the `ith-value` primitive provided both that the argument i is a deterministic expression and that the the argument *default-expression* is either absent or a deterministic expression.

- A call to the `print-values` primitive.

- A `function` special form. The expression `#'(lambda (...) ...)` always yields a single value and is thus deterministic even though that value may itself be a nondeterministic function object. This will be the case if one of the expressions in the body of the lambda expression is nondeterministic. Likewise, the expression `#'`*function-name* always is deterministic. It will evaluate to yield a nondeterministic function object if *function-name* names a nondeterministic function.

- An `either` special form with no arguments.

- An `either` special form with a single deterministic argument.

- A special form or a call to a deterministic function provided that every evaluated subexpression is deterministic.

A *deterministic function* is a function which SCREAMER can determine to yield at most one value when called. A *nondeterministic function* is a function which SCREAMER cannot determine to be deterministic. Like expressions, a function might yield no more than one value when actually called and still be classified as nondeterministic if SCREAMER cannot determine at compile time that it is deterministic. Nondeterministic functions, like deterministic functions, are defined with the COMMON LISP primitive `defun`. A function defined with `defun` is deterministic if every expression in its body is deterministic; otherwise the function is nondeterministic. Furthermore, all primitive COMMON LISP functions are deterministic.

COMMON LISP allows one to access function objects using the `function` special form. COMMON LISP function objects always refer to deterministic functions and are termed *deterministic function objects*. SCREAMER supports a new type of object, the *nondeterministic function object*. The expression `#'(lambda (...) ...)` will (deterministically) evaluate to yield an ordinary COMMON LISP deterministic function object if every expression in the body of the lambda expression is deterministic. It will (deterministically) evaluate to yield a nondeterministic function object if some expression in the body of the lambda expression is nondeterministic. Likewise, the expression `#'`*function-name* will (deterministically) evaluate to yield an ordinary COMMON LISP deterministic function object if *function-name* names a deterministic function. It will (deterministically) evaluate to yield a nondeterministic function object if *function-name* names a nondeterministic function (defined using `defun`). The SCREAMER primitive `nondeterministic-function?` can be used to determine whether or not an object is a nondeterministic function. While deterministic function objects can be called using the COMMON LISP primitives `funcall` and `apply`, nondeterministic function objects must be called using the analogous SCREAMER primitives `funcall-nondeterministic` and `apply-nondeterministic`. Attempting to call a nondeterministic function object with `funcall` or `apply` will signal a run time error. The SCREAMER calling primitives accept either deterministic or nondeterministic function objects. Irrespective of whether the *function* argument to `funcall-nondeterministic` or `apply-nondeterministic` is a deterministic or nondeterministic function object, a call to `funcall-nondeterministic` or `apply-nondeterministic` is always a nondeterministic expression since it is impossible, in general, to determine the type of the *function* argument at compile time.

SCREAMER includes several nondeterministic functions as primitives. These include:

- `funcall-nondeterministic`,
- `apply-nondeterministic`,
- `member-of`,
- `integer-between`,
- `decide`,
- `linear-force`,
- `divide-and-conquer-force`,
- `solution` and
- `best-value`.

(Actually, `decide` and `best-value` are macros.) The manual entries in this document explicitly annotate when a primitive is nondeterministic in the first line of the entry for that primitive. Primitives not annotated as nondeterministic are deterministic.

A mentioned previously, SCREAMER must compile nondeterministic expressions specially. It does so by walking the code and performing an operation known as CPS conversion on nondeterministic expressions (see section 14). SCREAMER only walks code appearing in certain contexts. The main implication of this is that nondeterministic expression may appear only in contexts which SCREAMER walks. These contexts are termed *nondeterministic contexts*. A context which SCREAMER does not walk is termed a *deterministic context*. The following is a list of all nondeterministic contexts:

- The body of `defun`. This is accomplished by having SCREAMER shadow the ordinary definition of `defun`.
- The *expression* argument of a call to the `map-values` primitive.
- The *expression* argument of a call to the `one-value` primitive.
- The body of a call to the `all-values` primitive.
- The *expression* argument of a call to the `ith-value` primitive.
- The body of a call to the `print-values` primitive.

Currently, the only top-level defining expression which allows a nondeterministic body is `defun`. This excludes other defining expressions such as `defvar`, `defconstant`, `defparameter`, `defmacro`, `defstruct`, `defclass`, and particularly `defmethod`. Future implementations of SCREAMER may support additional defining expressions beyond `defun`. The general rule is simply:

Nondeterministic expressions must appear in nondeterministic contexts.

A run time or compile time error should be signalled if this rule is violated.

Since, the COMMON LISP primitive `eval` is a deterministic function, it can only evaluate lists representing deterministic expressions. Like any call to a deterministic function, the argument to `eval` can be a nondeterministic expression, but the argument cannot evaluate to a list which represents a nondeterministic expression. This implies that you may not type a nondeterministic expression directly to a LISP listener. To evaluate a nondeterministic expression, convert it to a deterministic expression by enclosing it in one of the primitives `map-values`, `one-value`, `all-values`, `ith-value` or `print-values`.

The current implementation imposes one further restriction. The initialization expressions for `&aux`, `&optional` and `&key` variables appearing in a lambda list for a `defun` or lambda expression are considered deterministic contexts. Thus such initialization expressions may not be nondeterministic. A future version of SCREAMER may remove this restriction.

3 Side Effects

Adding nondeterminism to an imperative language like COMMON LISP offers two design alternatives for side effects. Should side effects be undone upon backtracking? SCREAMER supports both alternatives since both turn out to be useful. *Local* side effects are undone upon backtracking while *global* side effects are not. SCREAMER provides two new special forms, `local` and `global`, which control whether side effects are local or global. The expression `(local (setf x 0))` causes a local side effect to x, while `(global (setf x 0))` causes a global side effect.

The code generated for global side effects consists of ordinary COMMON LISP side effects. Local side effects generate code which first stores the prior value of the variable on a *trail* before updating the variable, and then restoring the prior value when backtracking. This has two implications. First, one can only perform a local side effect on a variable that is bound. SCREAMER cannot perform a local side effect on an unbound variable since it will not be able to access and store the prior value of the variable. Second, local side effects can be performed only when there is at least one choice point on the choice point stack, since the portion of the trail to be unwound upon backtracking is kept as part of a choice point. Depending on the COMMON LISP implementation, an error may or may not be generated if these restrictions are violated.

`Local` and `global` special forms behave like `progn`. They allow a body consisting of several expressions, returning the value of the last expression. All `setf` and `setq` operations that are *lexically nested* in a `local` or `global` expression generate side effects of the appropriate type. Furthermore, `local` and `global` expressions may be nested inside one another. In this case, the most closely nested `local` or `global` expressions determines whether a `setf` or `setq` causes a local or global side effect. Finally, `setf` and `setq` expression which are not nested in either a `local` or `global` expression cause global side effects. Thus the side effects to y and w in the following will be local, while the side effects to v and z will be global.

```
(defun f (x u) (setf v u))

(local (if (p x)
           (setf y (f x 1))
           (global (setf z (f x 2))))
       (setq w (f x 3)))
```

Note that a `local` or `global` expression affects only side effects that are lexically nested in that expression and not side effects in functions called by those expressions. Thus the side effect to v inside f is always a global side effect, even when f is called from inside a `local` expression.

`Local` and `global` expression affect any side effect caused by a lexically nested `setq` or `setf` expression. Side effects caused by other COMMON LISP primitives are always global. Thus `(local (delete x y))` will cause a global side effect *not* a local one.[1] `Local` and `global` do however, affect macros which expand into `setq` or `setf` expressions. COMMON LISP contains a number of side effect primitives implemented as macros. These include `psetq`, `psetf`, `shiftf`, `rotatef`, `incf`, `decf`, `push`, `pushnew` and `pop`. According to the COMMON LISP spec, the expansion of primitive macros is allowed to be implementation dependent.[2] Thus some implementations may expand primitive side effect macros, either directly or indirectly into expressions containing `setf` or `setq` expressions. In those implementations, calls to COMMON LISP primitive side effect macros will be affected by surrounding `local` and `global` expressions. In other implementations, calls to primitive side effect macros do not expand into expressions containing `setf` and `setq` and thus cause global side effects irrespective of their context.[3]

SCREAMER supports local side effects to generalized variables using `setf`. Thus it is possible to write expressions like `(local (setf (car x)) y)` and `(local (setf (aref x n)) y)` and achieve the expected results. When evaluating an expression like `(local (setf` v e`))`, SCREAMER may access v several times, once to trail its value prior to the local side effect and twice per backtrack, once to update its value and once to restore its prior value. SCREAMER takes care to evaluate any subexpressions in v only once and use the result of that evaluation for all such accesses. Local side effects are supported for any `setf` method including slot accessors created by `defstruct` and `defclass`. There is one minor exception however. While it is possible to do local side effects on hash tables, i.e. `(local (setf (gethash key table) value))`, SCREAMER is not able to distinguish between the situation where `table` lacked an entry for `key` prior to the local side effect and the situation where the entry for `key` was `nil`. In either case, `table` will have an entry pairing `key` with `nil` after backtracking. The reason for this is that when SCREAMER evaluates `(gethash key table)` to access its value prior to the local side effect, it utilizes only the first value returned by `gethash`. In both cases this value is `nil`. SCREAMER is not able to utilize the second value returned by `gethash` which would differentiate these situations.

One must be careful using iteration constructs containing nondeterministic expressions. These include the COMMON LISP primitives `do`, `do*`, `dolist`, `dotimes`, `do-all-symbols`, `do-external-symbols`, `do-symbols` and `loop` as well as the `iterate` macro. Such macros usually perform side effect during the iteration. The desired iteration behavior over a nondeterministic body usually requires the side effects to be local. It may be possible to get the correct behavior by wrapping the looping expression inside a `local` expression though for reasons described above, whether this works or not is implementation dependent.

Local side effects interact with SCREAMER primitives like `all-values` in predictable ways which may appear counterintuitive at first glance. Local side effect performed inside the body of an `all-values` expression are undone upon exit from the `all-values` expression. One may try unsuccessfully to use the following code to produce a list of all bit strings of length n.

```
(all-values (local (loop for i from 0 below n collect (either 0 1))))
```

The above code does not produce the desired result, even for an implementation which makes all side effects caused by `loop` to be local, since `collect` adds each new element to the list being generated by side effecting the tail of that list. If such side effects were local, the constructed list would be unavailable for use upon exiting the `all-values` expression.

[1] Technically, the COMMON LISP definition allows an implementation to in-line expand a call to a primitive such as `delete`. Furthermore, in some implementations, the function `macroexpand` performs in-line substitution, even though this is not allowed by the COMMON LISP spec. On an implementation which exhibited both of these flaws, `local` and `global` expressions could affect in-line expanded calls to COMMON LISP side effect primitives such as `delete`. I know of no implementation which suffers from this problem.

[2] In my opinion this is a flaw in the design of COMMON LISP for reasons made apparent in this paragraph.

[3] The Symbolics implementation behaves like the former while the Lucid implementation behaves partially like the former and partially like the later. In Lucid, side effect macros expand into `setq` when called on variables but not when called on generalized variables. Thus expressions like `(local (push a x))` cause local side effects while ones like `(local (push a (aref x n)))` cause global side effects. This is an unfortunate screw.

4 Gotchas

5 The Constraint Package

It is often said that PROLOG added two innovations to programming language design: nondeterminism and logic variables (unification).[4] The basic SCREAMER mechanisms of `either` and `fail` described in the previous sections add nondeterminism to COMMON LISP. This section describes an additional set of mechanisms provided with SCREAMER which add logic variables to COMMON LISP. This set of mechanisms is collectively called the *constraint package*. These mechanisms go a lot further than providing simple logic variables and unification; they provide much of the capability typically incorporated in constraint-based logic programming languages such as CLP($\Re$) (Jaffar and Michaylov, 1987; Heintze et al. 1987) and CHIP (Van Hentenryck, 1989). In contrast, SCREAMER uses constraint satisfaction methods based on range propagation rather than the linear programming techniques used by CLP($\Re$) and CHIP. Hence the name constraint package.

The constraint package adds a single new data type to COMMON LISP: the *variable*. Variables provide a superset of the functionality of logic variables. In addition to being able to be bound to arbitrary values, they may also be dynamically annotated at run time with constraints on the values to which they may be bound. Variables are created with the function `make-variable`. Initially they are *unbound* but may become *bound* as a result of constraints which are asserted between variables. Variables are bound if SCREAMER can determine that only a single value can satisfy the constraints in which the variable participates. Otherwise, variables remain unbound. All of the primitives provided by SCREAMER accept either bound variables, unbound variables or non-variables as arguments. Collectively we will refer to any such entity as a *value*. Unlike PROLOG and its derivatives, which usually allow all primitives to take variables as arguments, the primitives provided by COMMON LISP however, can take only non-variable values as arguments. Depending on the implementation and the setting of compiler optimization switches, most COMMON LISP primitives will signal an error message when passed a variable as an argument. The SCREAMER primitives `value-of` and `apply-substitution` can be used to access the value of a bound variable before calling a COMMON LISP primitive.[5] In most cases unbound variables can be forced to take on a unique value by using the SCREAMER primitives `linear-force` and `divide-and-conquer-force`. These are nondeterministic functions which provide the interface between constraint propagation and backtracking search. Bound variables are always dereferenced when printed. Unbound variables print in some form which is not parsable by the LISP reader, designed to make the identity and simple properties of the variable readily apparent. The SCREAMER primitives `bound?` and `ground?` can be used to determine whether a value is bound or unbound. The primitive `bound?` is analogous to the extra-logic predicates `var` and `nonvar` typically included in a PROLOG implementation.

Variables can participate in constraints. SCREAMER provides a rich set of primitives for mutually constraining variables. Constraint satisfaction is implemented in SCREAMER by a combination of constraint propagation and backtracking search. The constraint primitives attach procedures called *noticers* onto variables. Whenever the domain, range or type of a variable is restricted, the noticers attached onto that variable are run to propagate the effect of that restriction to other mutually constrained variables. Noticers implement constraint propagation. Since constraint propagation alone is not a complete constraint satisfaction technique, it is augmented by backtracking search. Constraint propagation and

[4] Some people add a third innovation, pattern directed invocation. In my opinion, pattern directed invocation is primarily a syntactic convenience while the other two innovations add significant expressiveness to any programming language.

[5] This operation is sometimes called *dereferencing*. PROLOG and its derivatives provide automatic dereferencing. It would be nice if COMMON LISP provided a hook to allow an arbitrary dereferencing function to be defined on a per type basis and specify that any primitive operation that was undefined given the types of arguments it was called with would first call the appropriate dereferencing functions on it arguments. Such an extension would be useful not only for logic variables but for lazy evaluation and futures (Halstead, 19??) as well.

backtracking search run automatically in an interleaved opportunistic fashion. The user never directly deals with the attaching and running noticers; that is automatically handled by SCREAMER. The user simply creates variables, asserts constraints between them and asks SCREAMER for solutions. Nonetheless, it is helpful to know a little bit about the implementation details since at one level, the precise semantics of the SCREAMER primitives depends on the implementation.

Several general principles pertaining to noticers hold throughout the implementation. First, all noticers are deterministic. Constraint propagation thus is a deterministic process. Noticers may fail however. Like all failures, this causes backtracking to the most recent choice point. Choice points are created by the primitive SCREAMER solution forcing functions `linear-force` and `divide-and-conquer-force` in addition to any other choice points explicitly created by the user. Second, all noticers are attached to variables by local side effect. Thus constraints asserted between variables disappear upon backtracking.

Constraint propagation may restrict the value that a variable may take on. Several forms of restriction are currently implemented in SCREAMER. These include domain restrictions, range restrictions and type restrictions. The kinds of restriction will be discussed in greater detail in the following sections. Several general principles pertaining to restrictions hold throughout the implementation. First, there is only one kind of variable. Any variable may have any combination of restrictions placed on it and may participate in any form of constraint. Second, variables are always restricted by local side effect. Thus any restrictions placed on the values of variables disappear upon backtracking. See section 3 for a discussion of the implications of this for the interaction between local side effects and SCREAMER primitives like `one-value`.

6 Domain, Range and Type Restrictions

The SCREAMER constraint package incorporates the following simple type system. All values are either *numeric* or *non-numeric*. Numeric values are in turn either *real* or *non-real*. Real numbers include both floating point numbers as well as rational numbers. Non-real numbers include complex numbers. Real values are in turn either *integer* or *non-integer*. The above classes are not disjoint. In the SCREAMER type system, all integers are reals and all real are numbers. Any value however, can be classified in one of the following four disjoint classes: integer, non-integer real, non-real number, non-number.

One must bear in mind that the SCREAMER notion of type is derived from the COMMON LISP representation of an object and not its abstract mathematical properties. Thus SCREAMER does not consider the number `1.0` to be an integer since its underlying COMMON LISP representation is a floating point number and COMMON LISP never automatically converts a floating point number to an integer. On the other hand, SCREAMER does consider `4/1` to be an integer since COMMON LISP will automatically convert a rational number with one as its denominator into an integer. Likewise, SCREAMER considers `#C(4 0)` to be an integer since COMMON LISP will convert a complex number whose components are integers and whose imaginary part is zero into an integer. SCREAMER however, does not consider `#C(4.0 0.0)` to be real since COMMON LISP will not convert a complex number whose components are reals into a real, even though its imaginary part is zero. Since COMMON LISP represents both parts of a complex number as the same data type, both `#C(4.0 0)` and `#C(4 0.0)` are treated as `#C(4.0 0.0)`.

SCREAMER can optionally restrict a variable to take on values only of a given type. Variables whose values are restricted to be numeric are termed numeric variables. Similar terminology is used for non-numeric, real, non-real, integer and non-integer variables. Variables created with `make-variable` initially carry no type restrictions. Subsequent type restrictions can be added using the primitives `integerv`, `realv` and `numberv`, potentially combined with `notv`, along with `assertv!`. For example, a real variable can be created using:

```
(let ((x (make-variable)))
 (assert! (realv x))
 ...)
```

a non-numeric variable using:

```
(let ((x (make-variable)))
 (assert! (notv (numberv x)))
 ...)
```

and a non-integer real using:

```
(let ((x (make-variable)))
 (assert! (notv (integerv x)))
 (assert! (realv x))
 ...).
```

As the last example shows, several type restrictions can be placed simultaneously on the same variable subject to the provision that the intersection of the types in nonempty. An attempt, such as the following, to restrict a variable to an empty set of possible values fails and backtracks to the previous choice point.

```
(let ((x (make-variable)))
 (assert! (notv (numberv x)))
 (assert! (integerv x))
 ...).
```

This does not cause an error unless the assertion is performed outside the context of any choice point. Remember that in SCREAMER, restricting a variable to be an integer inherently restricts it to be real and restricting a variable to be real inherently restricts it to be numeric. Likewise, restricting a variable to be non-numeric restricts it to be non-real and restricting a variable to be non-real restricts it to be non-integer. In the above example, `(assert! (integerv x))` will fail since `x` is already restricted to be non-numeric and thus non-integer.

All SCREAMER constraint primitives accept both bound and unbound variables, as well as non-variable values, as their arguments. This includes the type primitives `integerv`, `realv` and `numberv`. Thus `(assert! (integerv 4.0))` is acceptable and will fail immediately. Likewise `(assert! (integerv x))` will also fail immediately if `x` evaluates to `4.0` or if `x` evaluates to a variable which is bound to `4.0`.

In the above examples, the types of variables were explicitly restricted using the `assert!` primitive. SCREAMER may restrict the types of variables in two other ways. First, a number of SCREAMER primitives create and return variables which are already typed. In particular the primitives foo return integer variables, the primitives bar return real variables and the primitives baz return numeric variables. Second, a number of SCREAMER primitives restrict the types of their arguments. In particular ... Remember that all SCREAMER constraint primitives accept both bound and unbound variables, as well as non-variable values as arguments. A primitive which restricts an argument to a given type will fail immediately if called with a non-variable value not meeting the type restriction. Similarly, a bound variable whose value does not meet the type restriction will also cause immediate failure. When an unbound variable is passed to a restricted argument, the primitive immediately restricts that variable to be of the requisite type before any further processing. That restriction may cause immediate failure if the restriction would cause the variable's type to become empty.

The type hierarchy implies that at any point in time a variable can be in one of ten type restriction states: integer, non-integer, non-integer number, non-integer real, real, non-real, non-real number, number, non-number and unknown. In all but the last, some information is known about the type of value which the variable may take on. Thus a variable which is

7 Binding, Dereferencing and Sharing

8 Argument Restrictions

9 Aggregate Objects

A number of SCREAMER primitives deal with aggregate objects, traversing the slots of such objects recursively to access all of their sub-objects. These primitives include `equalv`, `ground?`, `apply-substitution`, `domain-size`, `solution` and `template`. The only aggregate objects which SCREAMER currently knows how to traverse are cons cells. All other aggregate objects are treated as atomic. Furthermore, any primitive which traverses aggregate data may loop if the aggregate object contains circular pointers.

10 Unwedging Screamer

To compile efficient code, SCREAMER must know whether or not an expression is deterministic since SCREAMER can generate more efficient code for deterministic expressions, subexpressions and functions than for nondeterministic ones. Whether or not an expression is deterministic can depend on whether or not the functions it calls are deterministic. Thus SCREAMER maintains a database of the function calling structure of any functions which have been compiled or loaded into the LISP world. This 'who-calls' database is implemented in portable COMMON LISP and is part of SCREAMER. It is independent of any other native who-calls database that may be built into the COMMON LISP implementation for other purposes.

For the most part, the SCREAMER who-calls database is completely transparent from the user. There are however, a number of quirky situations where it is helpful to understand its operation. Some of these situations are unavoidable. Others are simply bugs in the current implementation.

When compiling an expression e which calls a function f, SCREAMER needs to know whether or not f is deterministic to know how to compile e. There will be no problem if the definition of f appears before e in the program, i.e. if the call to f is a backward reference. SCREAMER will already know whether or not f is deterministic and thus how to compile e. The situation is more complex for forward references. SCREAMER assumes by default that f is deterministic and compiles e accordingly. It also saves the complete source code definition for the function g which contains e. If SCREAMER later discovers that f is actually nondeterministic, then SCREAMER recompiles g. This recompilation happens automatically. The user normally need not be aware that this is happening.

When using the COMMON LISP `compile-file` function, this recompilation causes the binary file to contain two definitions for the function g—one where the original definition appeared and one at the place of recompilation just prior to the definition of f.[6] Some COMMON LISP implementations will issue a warning that the binary file contains multiple definitions for functions. Such warnings can safely be ignored.[7] Note that there is no run time performance penalty for forward references and recompilation. The compiled code resulting at the end is the same as for backward references. Thus there should be no temptation to restructure program files to minimize forward references. The only penalties are increased compilation time due to recompilation, slightly increased binary file size and slightly increased load time. Recompiling a file a second time in the same LISP world will eliminate all duplicate definitions

[6] It is actually possible for a function to have several recompilations. If f has forward references to g and h, both of which are later found to be nondeterministic, f will be recompiled twice, once prior to the definition of g and again prior to the definition of h. It is also possible for recompilations to cascade. If f has a forward reference to g which in turn has a forward reference to h then discovering that h is nondeterministic will cause both g and then f to be recompiled when h is compiled. It is our experience that such multiple recompilations are very rare.

[7] It would be nice if COMMON LISP provided a declaration to disable such warnings.

and alleviate the increased binary file size and load time penalties since SCREAMER will have a complete and up-to-date who-calls database for the second compilation.

This same who-calls database allows users to safely redefine functions, changing them from deterministic to nondeterministic and vica versa. SCREAMER will track all such changes using the who-calls database and recompile the necessary functions to keep the code optimal and correct at all times. It is possible however, for there to be bugs in the who-calls database code which might cause SCREAMER to incorrectly determine whether or not a function is deterministic and thus generate incorrect or inefficient code. We request that you inform us if you discover such a bug by sending a bug report to `Bug-Screamer@AI.MIT.EDU` including as much information as possible to allow us to repeat that bug. Nonetheless, even if you are bitten by a bug in the who-calls database management code, it is usually possible for you to recover and still compile and run your program. We provide two functions to allow you to manually remove entries from the SCREAMER who-calls database. The first, `purge`, allows you to remove individual entries. This function would typically be used if you encounter problems compiling a function f. In this case, you should purge f along with all functions called by f and then recompile each of the functions just purged. The second, `unwedge-screamer`, is for more catastrophic situations. It will purge all user defined functions. You will then need to recompile and reload all user defined functions.

There is one currently known bug in the who-calls database code. It arises when a function calls itself, either directly or indirectly, through a chain of other functions (i.e. f calls g which in turn calls h which in turn calls f). If any function in the call cycle is nondeterministic than the all functions in the cycle will be nondeterministic. SCREAMER has no problem detecting that this is the case. If however, each of the functions in the cycle are redefined so that they are deterministic, the entire cycle can be made deterministic. SCREAMER will not detect this and will still think that the entire cycle is nondeterministic. Until this bug is fixed, you can manually recover from such a situation by simply purging each of the functions in the call cycle and then recompiling them. Remember to first purge *all* of the functions before recompiling *any* of them or SCREAMER will still insist that the entire call cycle is nondeterministic.

11 Manual Entries

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

Returns `t` if x is a nondeterministic function object and `nil` otherwise. Nondeterministic function objects can be produced in two ways. First, the special form `(function foo)` (i.e. `#'foo`) will (deterministically) evaluate to a nondeterministic function object if `foo` names a nondeterministic function defined by `defun`. Second, the special form `(function (lambda (...) ...))` (i.e. `#'(lambda (...) ...))` will (deterministically) evaluate to a nondeterministic function object if the body of the lambda expression contains a nondeterministic expression.

`either` $\{expression\}^*$ [*Nondeterministic Special Form*]

Nondeterministically evaluates and returns the value of one of its *expressions*. It sets up a choice point and evaluates the first *expression* returning its result. Whenever backtracking proceeds to this choice point, the next *expression* is evaluated and its result returned. When no more *expressions* remain, the current choice point is removed and backtracking continues to the next most recent choice point. As an optimization, the choice point created for this expression is removed before the evaluation of the last *expression* so that a failure during the evaluation of the last *expression* will backtrack directly to the parent choice point of the `either` expression. `Either` takes any number of arguments. With no arguments, `(either)` is equivalent to `(fail)` and is thus deterministic. With one argument, `(either` *expression*`)` is equivalent to *expression* itself and is thus deterministic only when *expression* is deterministic. `Either` is a special form, not a function. It is an error for the expression `#'either` to appear in a program. Thus `(funcall #'either` ...`)` or `(apply #'either` ...`)` are in error and will yield unpredictable results. With two or more argument it is nondeterministic and can only appear in a nondeterministic context. See section 2 for a discussion of the contexts in which `either` may appear.

`fail` [*Deterministic Function*]

Backtracks to the most recent choice point. Equivalent to `(either)`. Note that `fail` is a deterministic function and thus it is permissible to reference `#'fail` and write `(funcall #'fail)` and `(apply #'fail nil)`. In nondeterministic contexts, the expression `(fail)` is optimized to generate inline backtracking code.

`local` {*expression*}* [*Macro*]

Evaluates *expressions* in the same fashion as `progn` except that all `setf` and `setq` expressions *lexically nested* in its body result in *local* side effects which are undone upon backtracking. Note that this affects only side effects introduced explicitly via `setf` and `setq`. Side effects introduced by Common Lisp built-in functions such as `rplaca` are always global. Furthermore, it affects only occurrences of `setf` and `setq` which appear textually nested in the body of the `local` expression—not those appearing in functions called from the body. `Local` and `global` expressions may be nested inside one another. The nearest surrounding declaration 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. See sections 14, 3 and 4 for details. In nondeterministic contexts, `local` as well as `setf` are treated as special forms rather than macros. This should be completely transparent to the user.

`global` {*expression*}* [*Macro*]

Evaluates *expressions* in the same fashion as `progn` except that all `setf` and `setq` expressions *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 built-in functions such as `rplaca` are always global anyway. Furthermore, it affects only occurrences of `setf` and `setq` which appear textually nested in the body of the `global` expression—not those appearing in functions called from the body. `Local` and `global` expressions may be nested inside one another. The nearest surrounding declaration 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. See sections 3 and 4 for further details. In nondeterministic contexts, `global` as well as `setf` are treated as special forms rather than macros. This should be completely transparent to the user.

`map-values` *function expression* [*Deterministic Macro*]

Function must be a deterministic function object which takes one argument. This function is called on each value returned by evaluating *expression*, backtracking after calling *function* on each value to produce another value of *expression* until evaluating *expression* fails and produces no further values. The results returned by calling *function* on each value are discarded. Local side effects performed by *expression* are available during the corresponding call to *function* but are undone when that call returns. Likewise, local side effects performed by *function* are undone when it returns. Thus by the time `map-values` returns, all local side effects performed either by *expression* or *function* are undone. A `map-values` expression itself is always deterministic and always returns `nil`. In nondeterministic contexts, `map-values` is treated as a special form rather than a macro. This should be completely transparent to the user.

`one-value` *expression* [*default-expression*] [*Macro*]

Returns the first value of a nondeterministic expression. *Expression* is evaluated, deterministically returning only its first nondeterministic value, if any. No further execution of *expression* is attempted after it successfully returns one value. If *expression* does not return any nondeterministic values (i.e. it fails) then *default-expression* is evaluated and its value returned instead. *Default-expression* defaults to `(fail)` if not present. Local side effects performed by *expression* are undone when `one-value` returns. On the other hand, local side effects performed by *default-expression* 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` expression appears in, *expression* is always in a nondeterministic context, while *default-expression* is in whatever context the `one-value` expression appears. A `one-value` expression is nondeterministic if *default-expression* is present and is nondeterministic, otherwise it is deterministic. If *default-expression* is present and nondeterministic, and if *expression* fails, then it is possible to backtrack into the *default-expression* and for the `one-value` expression to nondeterministically return multiple times. `One-value` is analogous to the cut primitive (`!`) in PROLOG.

`all-values` $\{$*expression*$\}^*$ [*Deterministic Macro*]

Evaluates *expressions* (wrapped in an implicit `progn`) and returns a list of all of the nondeterministic values returned by the last *expression*. 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 the list containing `nil` if there are no *expressions*. An `all-values` expression can appear in both deterministic and nondeterministic contexts. Irrespective of what context the `all-values` expression appears in, the *expressions* are always in a nondeterministic context. An `all-values` expression itself is always deterministic. `All-values` is analogous to the `bagof` primitive in PROLOG.

`ith-value` *i expression* [*default-expression*] [*Macro*]

Returns the i^{th} value of a nondeterministic expression. *Expression* is evaluated, deterministically returning only its i^{th} nondeterministic value, if any. *I* must be an integer. The first nondeterministic value returned by *expression* is numbered *zero*, the second *one*, etc. The i^{th} value is produced by repeatedly evaluating *expression*, backtracking through and discarding the first *i* values and deterministically returning the next value produced. No further execution of *expression* is attempted after it successfully returns the desired value. If *expression* fails before returning both the *i* values to be discarded, as well as the desired i^{th} value, then *default-expression* is evaluated and its value returned instead. *Default-expression* defaults to `(fail)` if not present. Local side effects performed by *expression* are undone when `ith-value` returns. On the other hand, local side effects performed by *default-expression* 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` expression appears in, *expression* is always in a nondeterministic context, while *default-expression* and *i* are in whatever context the `ith-value` expression appears. An `ith-value` expression is nondeterministic if *default-expression* is present and is nondeterministic, or if *i* is nondeterministic. Otherwise it is deterministic. If *default-expression* is present and nondeterministic, and if *expression* fails, then it is possible to backtrack into the `default-expression` 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 *expression* and *default-expression* is nested in, and restarted for, each backtrack of *i*.

`print-values` {*expression*}* [*Deterministic Macro*]

Evaluates *expressions* (wrapped in an implicit `progn`) and prints each of the nondeterministic values returned by the last *expression* in succession (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. 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` expression appears in, the *expressions* are always in a nondeterministic context. A `print-values` expression itself is always deterministic and always returns `nil`. `Print-values` is analogous to the standard top-level user interface in PROLOG.

`funcall-nondeterministic` *function* {*argument*}* [*Nondeterministic Function*]

Analogous to the COMMON LISP built-in function `funcall` except that it accepts both ordinary COMMON LISP (deterministic) function objects as well as nondeterministic function objects for *function*. You must use `funcall-nondeterministic` to funcall a nondeterministic function object. A runtime error will be signalled if you attempt to funcall a nondeterministic function object with `funcall`. You can use `funcall-nondeterministic` to funcall either a deterministic or nondeterministic function object though even if all of the arguments to `funcall-nondeterministic` 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*.

`apply-nondeterministic` *function* {*argument*}+ [*Nondeterministic Function*]

Analogous to the COMMON LISP built-in function `apply` except that it accepts both ordinary COMMON LISP (deterministic) function objects as well as nondeterministic function objects for *function*. You must use `apply-nondeterministic` to apply a nondeterministic function object. A runtime error will be signalled if you attempt to apply a nondeterministic function object with `apply`. You can use `apply-nondeterministic` to apply either a deterministic or nondeterministic function object though even if all of the arguments to `apply-nondeterministic` 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*.

`purge` *function-name* [*Deterministic Function*]

Removes any information about *function-name* from SCREAMER's who-calls database. See section 10 for an explanation of when this would be used.

`unwedge-screamer` [*Deterministic Function*]

Removes any information about all user defined functions from SCREAMER's who-calls database. See section 10 for an explanation of when this would be used.

`member-of` *sequence* [*Nondeterministic Function*]

Nondeterministically returns an element of *sequence*. The elements are returned in the order that they appear in *sequence*. *Sequence* must be either a list or a vector.

`integer-between` *low high* [*Nondeterministic Function*]

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

`make-variable` [*name*] [*Deterministic 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.

`numberv` x [*Deterministic Function*]

If when `numberv` is called, x is known to be numeric then `numberv` returns `t`. Alternatively, if when `numberv` is called, x is known to be non-numeric then `numberv` returns `nil`. If it is not known whether or not x is numeric when `numberv` is called then `numberv` 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 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`. Furthermore, 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.

`realv` x [*Deterministic Function*]

If when `realv` is called, x is known to be real then `realv` returns `t`. Alternatively, if when `realv` is called, x is known to be non-real then `realv` returns `nil`. If it is not known whether or not x is real when `realv` is called then `realv` 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 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 noticers 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`. Furthermore, 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.

`integerv` x [*Deterministic Function*]

If when `integerv` is called, x is known to be integer valued then `integerv` returns `t`. Alternatively, if when `integerv` is called, x is known to be non-integer valued then `integerv` returns `nil`. If it is not known whether or not x is integer valued when `integerv` is called then `integerv` 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.

`memberv` x y [*Deterministic Function*]

The current implementation imposes two constraints on the parameter y. First, y must be bound when `memberv` is called. Second, y must not contain any unbound variables when `memberv` is called. The value of parameter y must be a sequence, i.e. either a list or a vector. If when `memberv` is called, x is known to be a member of y (using the COMMON LISP function `eql` as a test function) then `memberv` returns `t`. Alternatively, if when `memberv` is called, x is known not to be a member of y then `memberv` returns `nil`. If it is not known whether or not x is a member of y when `memberv` is called then `memberv` 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 a member of y and v is equal to `nil` if and only if x is known not to be a member of y. If x later becomes known to be a member of y, a noticer attached to x restricts v to equal `t`. Likewise, if x later becomes known not to be a member of y, 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 a member of y. Likewise, if v ever becomes known to equal `nil` then a noticer attached to v restricts x not to be a member of y.

`booleanv` x [*Deterministic Function*]

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

`assert!` x [*Deterministic Macro*]

The argument x can be either a variable or a non-variable. The expression `(assert!` x`)` restricts x to equal `t`. 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. No meaningful result is returned. 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! (numberv` 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 `numberv`. Calls to the functions `numberv`, `realv`, `integerv`, `memberv`, `booleanv`, `=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.

`known?` x [*Deterministic Macro*]

The argument x can be either a variable or a non-variable. The expression `(known?` x`)` restricts x to be boolean. This assertion 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. If x is equal to `t` after being restricted to be boolean then `known?` returns `t`. If x is equal to `nil` or if the value of x is unknown then `known?` returns `nil`. 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? (numberv` x`))` and `(known? (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 `numberv`. Calls to the functions `numberv`, `realv`, `integerv`, `memberv`, `booleanv`, `=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 [*Nondeterministic Macro*]

The argument x can be either a variable or a non-variable. The expression `(decide` x`)` restricts x to be boolean. This assertion 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. After x is restricted to be boolean 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. Except for implementation optimizations `(decide` x`)` is equivalent to:

```
(either (progn (assert! x) t) (progn (assert! (notv x)) nil)).
```

The implementation guarantees that x is evaluated only once so it may safely contain side effects. 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 (numberv` x`))` and `(decide (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 `numberv`. Calls to the functions `numberv`, `realv`, `integerv`, `memberv`, `booleanv`, `=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.

`=v` x^+ *[Deterministic 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 (see section 8). Returns `t` when called with one argument. A call such as `(=v` x_1 $x_2 \ldots x_n$`)` 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))
```

Behaves as follows when called with two arguments. Returns `t` if x_1 is known to be equal to x_2 at the time of call. Two numeric values are known to be equal only when they are both bound and equal according to the COMMON LISP function `=`. Returns `nil` if x_1 is known not to be equal to x_2 at the time of call. Two numeric values are known not to be equal when their domains are disjoint (see section 6). 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 (see section 6). If it is not known whether or not x_1 is equal to x_2 when `=v` is called then `=v` creates and returns a new boolean variable v. The values of x_1, x_2 and v are mutually constrained via noticers so that v is equal to `t` if and only if x_1 is known to be equal to x_2 and v is equal to `nil` if and only if x_1 is known not to be equal to x_2. If it later becomes known that x_1 is equal to x_2, noticers attached to x_1 and x_2 restrict v to equal `t`. Likewise, if it later becomes known that x_1 is not equal to x_2, noticers attached to x_1 and x_2 restrict v to equal `nil`. Furthermore, if v ever becomes known to equal `t` then a noticer attached to v restricts x_1 to be equal to x_2. Likewise, if v ever becomes known to equal `nil` then a noticer attached to v restricts x_1 not to be equal to x_2. Restricting two values x_1 and x_2 to be equal is performed by attaching noticers to x_1 and x_2. These noticers continually restrict the domains of x_1 and x_2 to be equivalent sets (using the COMMON LISP function `=` as a test function) as their domains are restricted. Furthermore, if x_1 is known to be real then the noticer attached to x_2 continually restrict the upper bound of x_1 to be no higher than the upper bound of x_2 and the lower bound of x_1 to be no lower than the lower bound of x_2. The noticer of x_2 performs a symmetric restriction on the bounds of x_1 if it is known to be real. Restricting two values x_1 and x_2 to not be equal is also performed by attaching noticers to x_1 and x_2. These noticers however, do not restrict the domains or ranges of x_1 or x_2. They simply monitor their continually restrictions and fail when any assertion causes x_1 to be known to be equal to x_2. See section 6 for a discussion of variable upper and lower bounds, and domains.

`<v` x^+ *[Deterministic Function]*

Returns a boolean value which is constrained to be `t` if each argument x_i is less than the following argument x_{i+1} and constrained to be `nil` if some argument x_i is greater than or equal to the following argument x_{i+1}. This function takes one or more arguments. All of the arguments are restricted to be real (see section 8). Returns `t` when called with one argument. A call such as `(<v` x_1 $x_2 \ldots x_n$`)` with more than two arguments behaves like a conjunction of two argument calls:

`(andv (<v` x_1 x_2`)...(<v` x_i x_{i+1}`)...(<v` x_{n-1} x_n`))`

Behaves as follows when called with two arguments. Returns `t` if x_1 is known to be less than x_2 at the time of call. A real value x_1 is known to be less than a real value x_2 if x_1 has an upper bound, x_2 has a lower bound and the upper bound of x_1 is less than the lower bound of x_2 (see section 6). Returns `nil` if x_1 is known to be greater than or equal to x_2 at the time of call. A real value x_1 is known to be greater than or equal to a real value x_2 if x_1 has a lower bound, x_2 has an upper bound and the lower bound of x_1 is greater than or equal to the upper bound of x_2 (see section 6). If it is not known whether or not x_1 is less than x_2 when `<v` is called then `>v` creates and returns a new boolean variable v. The values of x_1, x_2 and v are mutually constrained via noticers so that v is equal to `t` if and only if x_1 is known to be less than x_2 and v is equal to `nil` if and only if x_1 is known to be greater than or equal to x_2. If it later becomes known that x_1 is less than x_2, noticers attached to x_1 and x_2 restrict v to equal `t`. Likewise, if it later becomes known that x_1 is greater than or equal to x_2, noticers attached to x_1 and x_2 restrict v to equal `nil`. Furthermore, if v ever becomes known to equal `t` then a noticer attached to v restricts x_1 to be less than x_2. Likewise, if v ever becomes known to equal `nil` then a noticer attached to v restricts x_1 to be greater than or equal to x_2. Restricting a real value x_1 to be less than a real value x_2 is performed by attaching noticers to x_1 and x_2. The noticer attached to x_1 continually restricts the lower bound of x_2 to be no lower than the upper bound of x_1 if x_1 has an upper bound. The noticer attached to x_2 continually restricts the upper bound of x_1 to be no higher than the lower bound of x_2 if x_2 has a lower bound. Since these restrictions only guarantee that x_1 be less than or equal to x_2, the constraint that x_1 be strictly less than x_2 is enforced by having the noticers fail when both x_1 and x_2 become known to be equal. Restricting a real value x_1 to be greater than or equal to a real value x_2 is performed by an analogous set of noticers without this last equality check. See section 6 for a discussion of variable upper and lower bounds, and domains.

`<=v` x^+ *[Deterministic Function]*

Returns a boolean value which is constrained to be `t` if each argument x_i is less than or equal to the following argument x_{i+1} and constrained to be `nil` if some argument x_i is greater than the following argument x_{i+1}. This function takes one or more arguments. All of the arguments are restricted to be real (see section 8). Returns `t` when called with one argument. A call such as `(<=v` x_1 $x_2 \ldots x_n$`)` with more than two arguments behaves like a conjunction of two argument calls:

`(andv (<=v` x_1 x_2`)...(<=v` x_i x_{i+1}`)...(<=v` x_{n-1} x_n`))`

Behaves as follows when called with two arguments. Returns `t` if x_1 is known to be less than or equal to x_2 at the time of call. A real value x_1 is known to be less than or equal to a real value x_2 if x_1 has an upper bound, x_2 has a lower bound and the upper bound of x_1 is less than or equal to the lower bound of x_2 (see section 6). Returns `nil` if x_1 is known to be greater than x_2 at the time of call. A real value x_1 is known to be greater than a real value x_2 if x_1 has a lower bound, x_2 has an upper bound and the lower bound of x_1 is greater than the upper bound of x_2 (see section 6). If it is not known whether or not x_1 is less than or equal to x_2 when `>=v` is called then `>=v` creates and returns a new boolean variable v. The values of x_1, x_2 and v are mutually constrained via noticers so that v is equal to `t` if and only if x_1 is known to be less than or equal to x_2 and v is equal to `nil` if and only if x_1 is known to be greater than x_2. If it later becomes known that x_1 is less than or equal to x_2, noticers attached to x_1 and x_2 restrict v to equal `t`. Likewise, if it later becomes known that x_1 is greater than x_2, noticers attached to x_1 and x_2 restrict v to equal `nil`. Furthermore, if v ever becomes known to equal `t` then a noticer attached to v restricts x_1 to be less than or equal to x_2. Likewise, if v ever becomes known to equal `nil` then a noticer attached to v restricts x_1 to be greater than x_2. Restricting a real value x_1 to be less than or equal to a real value x_2 is performed by attaching noticers to x_1 and x_2. The noticer attached to x_1 continually restricts the lower bound of x_2 to be no lower than the upper bound of x_1 if x_1 has an upper bound. The noticer attached to x_2 continually restricts the upper bound of x_1 to be no higher than the lower bound of x_2 if x_2 has a lower bound. Restricting a real value x_1 to be greater than a real value x_2 is performed by an analogous set of noticers. Since these restrictions only guarantee that x_1 be greater than or equal to x_2, the constraint that x_1 be strictly greater than x_2 is enforced by having the noticers fail when both x_1 and x_2 become known to be equal. See section 6 for a discussion of variable upper and lower bounds, and domains.

`>v` x^+ [*Deterministic Function*]

Returns a boolean value which is constrained to be `t` if each argument x_i is greater than the following argument x_{i+1} and constrained to be `nil` if some argument x_i is less than or equal to the following argument x_{i+1}. This function takes one or more arguments. All of the arguments are restricted to be real (see section 8). Returns `t` when called with one argument. A call such as `(>v` x_1 $x_2 \ldots x_n$`)` with more than two arguments behaves like a conjunction of two argument calls:

`(andv (>v` x_1 x_2`)...(>v` x_i x_{i+1}`)...(>v` x_{n-1} x_n`))`

Behaves as follows when called with two arguments. Returns `t` if x_1 is known to be greater than x_2 at the time of call. A real value x_1 is known to be greater than a real value x_2 if x_1 has a lower bound, x_2 has an upper bound and the lower bound of x_1 is greater than the upper bound of x_2 (see section 6). Returns `nil` if x_1 is known to be less than or equal to x_2 at the time of call. A real value x_1 is known to be less than or equal to a real value x_2 if x_1 has an upper bound, x_2 has a lower bound and the upper bound of x_1 is less than or equal to the lower bound of x_2 (see section 6). If it is not known whether or not x_1 is greater than x_2 when `>v` is called then `>v` creates and returns a new boolean variable v. The values of x_1, x_2 and v are mutually constrained via noticers so that v is equal to `t` if and only if x_1 is known to be greater than x_2 and v is equal to `nil` if and only if x_1 is known to be less than or equal to x_2. If it later becomes known that x_1 is greater than x_2, noticers attached to x_1 and x_2 restrict v to equal `t`. Likewise, if it later becomes known that x_1 is less than or equal to x_2, noticers attached to x_1 and x_2 restrict v to equal `nil`. Furthermore, if v ever becomes known to equal `t` then a noticer attached to v restricts x_1 to be greater than x_2. Likewise, if v ever becomes known to equal `nil` then a noticer attached to v restricts x_1 to be less than or equal to x_2. Restricting a real value x_1 to be greater than a real value x_2 is performed by attaching noticers to x_1 and x_2. The noticer attached to x_1 continually restricts the upper bound of x_2 to be no higher than the lower bound of x_1 if x_1 has a lower bound. The noticer attached to x_2 continually restricts the lower bound of x_1 to be no lower than the upper bound of x_2 if x_2 has an upper bound. Since these restrictions only guarantee that x_1 be greater than or equal to x_2, the constraint that x_1 be strictly greater than x_2 is enforced by having the noticers fail when both x_1 and x_2 become known to be equal. Restricting a real value x_1 to be less than or equal to a real value x_2 is performed by an analogous set of noticers without this last equality check. See section 6 for a discussion of variable upper and lower bounds, and domains.

`>=v` x^+ *[Deterministic Function]*

Returns a boolean value which is constrained to be `t` if each argument x_i is greater than or equal to the following argument x_{i+1} and constrained to be `nil` if some argument x_i is less than the following argument x_{i+1}. This function takes one or more arguments. All of the arguments are restricted to be real (see section 8). Returns `t` when called with one argument. A call such as `(>=v` x_1 $x_2 \ldots x_n$`)` with more than two arguments behaves like a conjunction of two argument calls:

`(andv (>=v` x_1 x_2`)...(>=v` x_i x_{i+1}`)...(>=v` x_{n-1} x_n`))`

Behaves as follows when called with two arguments. Returns `t` if x_1 is known to be greater than or equal to x_2 at the time of call. A real value x_1 is known to be greater than or equal to a real value x_2 if x_1 has a lower bound, x_2 has an upper bound and the lower bound of x_1 is greater than or equal to the upper bound of x_2 (see section 6). Returns `nil` if x_1 is known to be less than x_2 at the time of call. A real value x_1 is known to be less than a real value x_2 if x_1 has an upper bound, x_2 has a lower bound and the upper bound of x_1 is less than the lower bound of x_2 (see section 6). If it is not known whether or not x_1 is greater than or equal to x_2 when `>=v` is called then `>=v` creates and returns a new boolean variable v. The values of x_1, x_2 and v are mutually constrained via noticers so that v is equal to `t` if and only if x_1 is known to be greater than or equal to x_2 and v is equal to `nil` if and only if x_1 is known to be less than x_2. If it later becomes known that x_1 is greater than or equal to x_2, noticers attached to x_1 and x_2 restrict v to equal `t`. Likewise, if it later becomes known that x_1 is less than x_2, noticers attached to x_1 and x_2 restrict v to equal `nil`. Furthermore, if v ever becomes known to equal `t` then a noticer attached to v restricts x_1 to be greater than or equal to x_2. Likewise, if v ever becomes known to equal `nil` then a noticer attached to v restricts x_1 to be less than x_2. Restricting a real value x_1 to be greater than or equal to a real value x_2 is performed by attaching noticers to x_1 and x_2. The noticer attached to x_1 continually restricts the upper bound of x_2 to be no higher than the lower bound of x_1 if x_1 has a lower bound. The noticer attached to x_2 continually restricts the lower bound of x_1 to be no lower than the upper bound of x_2 if x_2 has an upper bound. Restricting a real value x_1 to be less than a real value x_2 is performed by an analogous set of noticers. Since these restrictions only guarantee that x_1 be less than or equal to x_2, the constraint that x_1 be strictly less than x_2 is enforced by having the noticers fail when both x_1 and x_2 become known to be equal. See section 6 for a discussion of variable upper and lower bounds, and domains.

`/=v` x^+ [*Deterministic Function*]

Returns a boolean value which is constrained to be **t** if two or more of the arguments numerically differ and constrained to be **nil** all of the arguments are numerically equal. This function takes one or more arguments. All of the arguments are restricted to be numeric (see section 8). Returns **nil** when called with one argument. A call such as `(/=v` x_1 $x_2 \ldots x_n$`)` 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))
```

Behaves as follows when called with two arguments. Returns **t** if x_1 is known not to be equal to x_2 at the time of call. Two numeric values are known not to be equal when their domains are disjoint (see section 6). 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 (see section 6). Returns **nil** if x_1 is known to be equal to x_2 at the time of call. Two numeric values are known to be equal only when they are both bound and equal according to the Common Lisp function `=`. If it is not known whether or not x_1 is equal to x_2 when `/=v` is called then `/=v` creates and returns a new boolean variable v. The values of x_1, x_2 and v are mutually constrained via noticers so that v is equal to **t** if and only if x_1 is known not to be equal to x_2 and v is equal to **nil** if and only if x_1 is known to be equal to x_2. If it later becomes known that x_1 is not equal to x_2, noticers attached to x_1 and x_2 restrict v to equal **t**. Likewise, if it later becomes known that x_1 is equal to x_2, noticers attached to x_1 and x_2 restrict v to equal **nil**. Furthermore, if v ever becomes known to equal **t** then a noticer attached to v restricts x_1 to not be equal to x_2. Likewise, if v ever becomes known to equal **nil** then a noticer attached to v restricts x_1 to be equal to x_2. Restricting two values x_1 and x_2 to be equal is performed by attaching noticers to x_1 and x_2. These noticers continually restrict the domains of x_1 and x_2 to be equivalent sets (using the Common Lisp function `=` as a test function) as their domains are restricted. Furthermore, if x_1 is known to be real then the noticer attached to x_2 continually restrict the upper bound of x_1 to be no higher than the upper bound of x_2 and the lower bound of x_1 to be no lower than the lower bound of x_2. The noticer of x_2 performs a symmetric restriction on the bounds of x_1 if it is known to be real. Restricting two values x_1 and x_2 to not be equal is also performed by attaching noticers to x_1 and x_2. These noticers however, do not restrict the domains or ranges of x_1 or x_2. They simply monitor their continually restrictions and fail when any assertion causes x_1 to be known to be equal to x_2. See section 6 for a discussion of variable upper and lower bounds, and domains.

`real-abovev` *low* [*Deterministic Function*]

Returns a real variable whose value is constrained to be greater than or equal to *low*. The expression `(real-above` *low*`)` is an abbreviation for:

```
(let ((v (make-variable)))
 (assert! (realv v))
 (assert! (>=v v low))
 v)
```

`real-belowv` *high* [*Deterministic Function*]

Returns a real variable whose value is constrained to be less than or equal to *high*. The expression `(real-belowv` *high*`)` is an abbreviation for:

```
(let ((v (make-variable)))
 (assert! (realv v))
 (assert! (<=v v high))
 v)
```

`real-betweenv` *low high* [*Deterministic 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 `(real-betweenv` *low high*`)` is an abbreviation for:

```
(let ((v (make-variable)))
 (assert! (realv v))
 (assert! (>=v v low))
 (assert! (<=v v high))
 (value-of v))
```

`integer-abovev` *low* [*Deterministic Function*]

Returns an integer variable whose value is constrained to be greater than or equal to *low*. The expression `(integer-above` *low*`)` is an abbreviation for:

```
(let ((v (make-variable)))
 (assert! (integerv v))
 (assert! (>=v v low))
 v)
```

`integer-belowv` *high* [*Deterministic Function*]

Returns an integer variable whose value is constrained to be less than or equal to *high*. The expression `(integer-belowv` *high*`)` is an abbreviation for:

```
(let ((v (make-variable)))
 (assert! (integerv v))
 (assert! (<=v v high))
 v)
```

`integer-betweenv` *low high* [*Deterministic 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 `(integer-betweenv` *low high*`)` is an abbreviation for:

```
(let ((v (make-variable)))
 (assert! (integerv v))
 (assert! (>=v v low))
 (assert! (<=v v high))
 (value-of v))
```

`member-ofv` *sequence* [*Deterministic Function*]

Returns a variable whose value is restricted to be a member of *sequence*. If *sequence* is a singleton and the resulting variable would be bound, its value is returned instead. Fails if *sequence* is empty. The expression `(member-ofv` *sequence*`)` is an abbreviation for:

```
(let ((v (make-variable)))
 (assert! (memberv v sequence))
 (value-of v))
```

`notv` x [*Deterministic Function*]

Restricts x to be boolean (see section 8). Returns `nil` if x is known to equal `t` after this restriction. Returns `t` if x is known to equal `nil` after this restriction. If it is not known whether x equals `t` or `nil` then `notv` 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 equal `nil` and v is equal to `nil` if and only if x is known to equal `t`. If x later becomes known to equal `t` then a noticer attached to x restricts v to equal `nil`. Likewise, if x later becomes known to equal `nil` then a noticer attached to x restricts v to equal `t`. Furthermore, if v later becomes known to equal `t` then a noticer attached to v restricts x to equal `nil`. Likewise, if v later becomes known to equal `nil` then a noticer attached to v restricts x to equal `t`. Note that `notv` differs from the COMMON LISP primitive `not` in that while `not` accepts any COMMON LISP object as input, treating any non-`nil` value as true, `notv` restricts is argument to be a boolean value, failing if its argument cannot be restricted to be boolean.

`andv` x^* [*Deterministic Function*]

Takes zero or more arguments. Restricts each argument to be boolean (see section 8). Returns `t` if called with no arguments. Returns `nil` if any argument is known to equal `nil` after this restriction. Returns `t` if all arguments are known to equal `t` after this restriction. If neither of the above conditions hold then `andv` creates and returns a new boolean variable v. The values of the arguments x_i and v are mutually constrained via noticers so that v is equal to `t` if and only if all of the arguments x_i are known to equal `t` and v is equal to `nil` if and only if some argument x_i is known to equal `nil`. If all of the arguments x_i later becomes known to equal `t` then noticers attached to the arguments x_i restrict v to equal `t`. Likewise, if some argument x_i later becomes known to equal `nil` then a noticer attached to x_i restricts v to equal `nil`. Furthermore, if v later becomes known to equal `nil` then a noticer attached to v restricts all of the arguments x_i to equal `nil`. SCREAMER implements one further optimization. If when `andv` is called, all of the arguments except for one, say x_i, are known to equal `t` and it is not known whether x_i equals `t` or `nil` then `andv` returns the argument x_i directly as the result. Note that `andv` differs from the COMMON LISP primitive `and` in two important ways. First, while `and` is a macro and evaluates its arguments from left to right halting that evaluation when an argument evaluates to `nil`, `andv` is a function which always evaluates all of its arguments. Since `andv` is a function however, it can be funcalled and mapped, something which cannot be done with `and` since it is a macro. Second, while `and` accepts any COMMON LISP object as input, treating any non-`nil` value as true, `andv` restricts its arguments to be boolean values, failing if any argument cannot be restricted to be boolean.

`orv` x^* [*Deterministic Function*]

Takes zero or more arguments. Restricts each argument to be boolean (see section 8). Returns `nil` if called with no arguments. Returns `t` if any argument is known to equal `t` after this restriction. Returns `nil` if all arguments are known to equal `nil` after this restriction. If neither of the above conditions hold then `orv` creates and returns a new boolean variable v. The values of the arguments x_i and v are mutually constrained via noticers so that v is equal to `nil` if and only if all of the arguments x_i are known to equal `nil` and v is equal to `t` if and only if some argument x_i is known to equal `t`. If all of the arguments x_i later becomes known to equal `nil` then noticers attached to the arguments x_i restrict v to equal `nil`. Likewise, if some argument x_i later becomes known to equal `t` then a noticer attached to x_i restricts v to equal `t`. Furthermore, if v later becomes known to equal `t` then a noticer attached to v restricts all of the arguments x_i to equal `t`. SCREAMER implements one further optimization. If when `orv` is called, all of the arguments except for one, say x_i, are known to equal `nil` and it is not known whether x_i equals `t` or `nil` then `orv` returns the argument x_i directly as the result. Note that `orv` differs from the COMMON LISP primitive `or` in two important ways. First, while `or` is a macro and evaluates its arguments from left to right halting that evaluation when an argument evaluates to `t`, `orv` is a function which always evaluates all of its arguments. Since `orv` is a function however, it can be funcalled and mapped, something which cannot be done with `or` since it is a macro. Second, while `or` accepts any COMMON LISP object as input, treating any non-`nil` value as true, `orv` restricts its arguments to be boolean values, failing if any argument cannot be restricted to be boolean.

`+v` x^* [*Deterministic Function*]

`-v` x^+ [*Deterministic Function*]

`*v` x^* [*Deterministic Function*]

`/v` x^+ [*Deterministic Function*]

`minv` x^+ [*Deterministic Function*]

`maxv` x^+ [*Deterministic Function*]

`funcallv` *function* x^* [*Deterministic Function*]

The argument *function* must be bound to a deterministic COMMON LISP function object. Returns the result of funcalling *function* on the dereferenced values of the arguments x_i if all of the x_i are bound when `funcallv` is called. If some of the arguments x_i are not bound when `funcallv` is called then `funcallv` creates and returns a variable v. The arguments x_i and v are mutually constrained via noticers so that if v and the arguments x_i ever all become bound then v is restricted to equal the result of funcalling *function* on the dereferenced values of the arguments x_i. The noticers furthermore implement forward checking, i.e. if the collection v combined with the arguments x_i all become bound, save one variable, and this one variable has a finite domain then this domain is restricted to contain only those elements which are consistent with the values of the remaining bound variables. The following is a standard cliché used to constrain the variables $x_1 \ldots x_n$ to obey a given predicate:

`(assert! (funcallv #'(lambda (`$x_1 \ldots x_n$`)` *predicate*`)` $x_1 \ldots x_n$`))`

`applyv` *function* x^+ [*Deterministic Function*]

The argument *function* must be bound to a deterministic COMMON LISP function object. Returns the result of applying *function* on the dereferenced values of the arguments x_i if all of the x_i are bound when `applyv` is called. If some of the arguments x_i are not bound when `applyv` is called then `applyv` creates and returns a variable v. The arguments x_i and v are mutually constrained via noticers so that if v and the arguments x_i ever all become bound then v is restricted to equal the result of applying *function* on the dereferenced values of the arguments x_i. The noticers furthermore implement forward checking, i.e. if the collection v combined with the arguments x_i all become bound, save one variable, and this one variable has a finite domain then this domain is restricted to contain only those elements which are consistent with the values of the remaining bound variables.

`equalv` x y [*Deterministic Function*]

Returns `t` if the aggregate object x is known to equal the aggregate object y when `equalv` is called. Returns `nil` if the aggregate object x is known not to equal the aggregate object y when `equalv` is called. If it is not known whether or not x equals y when `equalv` is called then `equalv` creates and returns a boolean variable v. 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. See section 9 for a discussion of aggregate objects.

`bound?` x *[Deterministic 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. See section 7 for a discussion of how SCREAMER determines whether or not a variable is bound.

`value-of` x *[Deterministic 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. See section 7 for a discussion of how SCREAMER determines whether or not a variable is bound and how SCREAMER dereferences variables.

`ground?` x *[Deterministic 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 all of the slots in the value of x are also bound. Otherwise returns `nil`. See section 7 for a discussion of how SCREAMER determines whether or not a variable is bound. See section 9 for a discussion of aggregate objects.

`apply-substitution` x [*Deterministic Function*]

The primitive `apply-substitution` is an extension of the primitive `value-of` which can recursively dereference an entire aggregate object. If x is an unbound variable then `apply-substitution` behaves like `value-of` dereferencing x and returning the dereferenced value. The value returned will be a variable which may be x itself or another variable which is shared with x. Otherwise, returns a copy of the value of x. Each slot of x that contains a bound variable is replaced in the copy with the dereferenced value of that variable produced by a recursive application of `apply-substitution`. See section 7 for a discussion of how SCREAMER determines whether or not a variable is bound and how SCREAMER dereferences variables. See section 9 for a discussion of aggregate objects.

`linear-force` x [*Nondeterministic Function*]

Returns x if x is not a variable. If x is a bound variable then behaves like `value-of` returning the dereferenced value of x. 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 alternative 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. See section 6 for a discussion of restriction to finite domains and to integer values. See section 7 for a discussion of how SCREAMER determines whether or not a variable is bound and how SCREAMER dereferences variables.

`divide-and-conquer-force` x [*Nondeterministic Function*]

Returns x if x is not a variable. If x is a bound variable then behaves like `value-of` returning the dereferenced value of x. If x is an unbound variable then it must be known either to have a finite domain or to be real and have a lower and upper bound. 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 dereferenced value is returned. Otherwise, x itself is returned. 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. 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. This restriction will cause x to become bound if the difference between the lower and upper bounds becomes smaller than `*fuzz*`. 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. `Divide-and-conquer-force` 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. While a single application of `divide-and-conquer-force` may not cause its argument to be bound, repeated application is guaranteed to eventually bind its argument so long as `*fuzz*` is non-zero. See section 6 for a discussion of restriction to finite domains and to real values. See section 7 for a discussion of how SCREAMER determines whether or not a variable is bound and how SCREAMER dereferences variables.

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

This function is a higher-order function which takes a *force-function* as an argument and returns an ordering force function as its result. 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. SCREAMER currently provides two convenient *force-function*s, 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*. 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.

`domain-size` x [*Deterministic Function*]

Returns the domain size of x. This is the number of possible ground values which x may consistently take on. This function recursively traverses aggregate objects when computing domain sizes. The domain size of an aggregate object is the product of the domain sizes of its slots. An atomic non-variable has a domain size of one. The domain size of a bound variable is the domain size of its dereferenced value. If an unbound variable is known to have a finite domain then its domain size is the size of that domain. If an unbound variable is not known to have a finite domain but is known to be integer valued and to have upper and lower bounds then its domain size is the one plus the difference between the upper and lower bounds Otherwise, the domain size of an unbound variable is infinite. `Domain-size` returns `nil` if x, or any slot nested in x, has an infinite domain size. See section 9 for a discussion of aggregate objects.

`reorder` *force-function* [*Deterministic Function*]

This function is a higher-order function which takes a *force-function* as an argument and returns an ordering force function as its result. 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. 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*. 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 applies the *force-function* to the variable in x with the smallest domain size until all variables in x are bound. The ordering force function does not return any meaningful result.

`solution` *list ordering-force-function* [*Nondeterministic Function*]

The argument *list* is a list of values. Typically it is a list of variables but it may also contain non-variables. The specified *ordering-force-function* is used to force each of the variables in *list* to be bound. Returns a list of the dereference 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 four alternatives provided with SCREAMER.

- `(static-ordering #'linear-force)`,
- `(static-ordering #'divide-and-conquer-force)`,
- `(reorder #'linear-force)` and
- `(reorder #'divide-and-conquer-force)`.

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

`best-value` *objective-function expression* [*Nondeterministic Macro*]

`template` *x* [*Deterministic 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. See section 9 for a discussion of aggregate objects.

`*fuzz*` [*Variable*]

The variable `*fuzz*` is used to determine whether a real variable is bound when it is not known to have a finite domain. If a real value has lower and upper bounds and the difference between the lower and upper bounds is less than `*fuzz*` then the variable is considered bound. When such a variable is dereferenced, the lower bound is arbitrarily taken as its value. See section 6 for further discussion.

12 Examples

In this section I will present a number of examples of SCREAMER programs to illustrate typical programming styles and clichés. Figure 1 contains a program for computing all Pythagorean triples of positive integers less than or equal to n. It illustrates a simple generate-and-test paradigm. The `let` bindings of the function `pythagorean-triples` constitute the generator while the `unless` expression constitutes the test. The function `integer-between` nondeterministically returns an integer in the range given by its arguments. This function is so useful that it is built into SCREAMER, though it could easily be defined as in figure 1.

Figure 1 also contains a second program for finding pythagorean triples. While the first one uses pure backtracking search, the second function `pythagorean-triplesv` uses the SCREAMER constraint package. While for this problem, the constraint based version does not offer much computational advantage over the backtracking version, this example does illustrate a standard cliché for reformulating generate-and-test algorithms into constraint propagation ones. Note the intentional similarity in the names of the primitives `integer-between` and `integer-betweenv`. A similar analogy exists between the primitives `member-of` and `member-ofv`.

Figure 2 contains a program for solving the N-Queens problem. The function `attacks` returns `t` if a queen on row q_i attacks a queen on row q_j when the queens are *distance* columns apart. The function `check-queens` checks whether a new queen can be placed on the board without attacking the queens already placed. It recursively checks the new queen against each previously placed queen and fails if it detects an attack. The parameter *queen* gives the row of the new queen to be placed in the column adjacent to columns of the previously placed queens. The parameter *queens* is a list of the rows of the previously placed queens. Note that the functions `attacks` and `check-queens` are deterministic. The function `n-queens` nondeterministically recursively places queens on the board, checking for attacks as each queen is placed.

This example illustrates the cliché of interleaving the generate and test processes to implement early failure detection. Rather than wait until all queens are placed before checking for attacks, consistency

```
(defun integer-between (low high)
 (if (> low high) (fail) (either low (integer-between (1+ low) high))))

(defun pythagorean-triples (n)
 (all-values
  (let ((a (integer-between 1 n))
        (b (integer-between 1 n))
        (c (integer-between 1 n)))
   (unless (= (+ (* a a) (* b b)) (* c c)) (fail))
   (list a b c))))

(defun pythagorean-triplesv (n)
 (let ((a (integer-betweenv 1 n))
       (b (integer-betweenv 1 n))
       (c (integer-betweenv 1 n)))
  (assert! (=v (+v (*v a a) (*v b b)) (*v c c)))
  (all-values (solution (list a b c) (reorder #'divide-and-conquer-force)))))
```

Figure 1: Two SCREAMER programs for finding pythagorean triples. The first uses backtracking search while the second uses the constraint package.

is checked after making each nondeterministic choice. This example also illustrates how COMMON LISP features such as `&optional` arguments are available in SCREAMER.

Figure 3 contains a program for finding all simple paths in a directed graph. A simple path is one which does not visit a vertex more than once. The graph is represented via the `defstruct node`. Each node contains two slots. The `marked?` slot is used to record when a node has already been visited in the path being constructed. The `next-nodes` slot contains a list of all of the nodes reachable from a given node via out-edges. The function `member-of` is used to nondeterministically choose one of the nodes in the `next-nodes` slot as the next node in the path being constructed. This function is so useful that it is built into SCREAMER, though it could easily be defined as in figure 3. The version built into SCREAMER is slightly more complex as it accepts either a vector or a list as its argument.

This example illustrates the utility of local side effects. Local side effects are used to mark a node as being visited so that they are not included twice in the path. The marking is undone upon backtracking as different paths are considered.

The function `simple-path` never overwrites a local side effect. As such, its side effects are single assignment, much like PROLOG's binding of logic variables. SCREAMER allows multiple assignment local side effects, something which is not possible in PROLOG. The second example in figure 3 illustrates the utility of multiple assignment local side effects. Let us extend the definition of simple path to define the notion of a k-simple path. A k-simple path is one which never visits a node more than k times. A program for finding simple paths can be converted into one which finds k-simple paths by keeping a `visits` count for each node instead of a `marked?` flag. Incrementing this `visits` count requires multiple assignment local side effects. This example also illustrates how local side effects can be done on any COMMON LISP generalized variable (such as `defstruct` slots) via the `setf` primitive.

The SCREAMER constraint package is implemented using the basic nondeterministic primitives provided by SCREAMER. No internals of SCREAMER needed to be modified to support the constraint package. Had the constraint package not been provided with SCREAMER, a user could have constructed her own

```
(defun attacks (qi qj distance)
 (or (= qi qj)
     (= qi (+ qj distance))
     (= qi (- qj distance))))

(defun check-queens (queen queens &optional (distance 1))
 (unless (null queens)
  (if (attacks queen (first queens) distance) (fail))
  (check-queens queen (rest queens) (1+ distance))))

(defun n-queens (n &optional queens)
 (if (= (length queens) n)
     queens
     (let ((queen (integer-between 1 n)))
      (check-queens queen queens)
      (n-queens n (cons queen queens)))))

(defun n-queensv (n)
 (let ((q (make-array n)))
  (dotimes (i n)
   (setf (aref q i) (member-ofv (all-values (integer-between 1 n)))))
  (dotimes (i n)
   (dotimes (j n)
    (if (> j i)
        (assert! (notv (funcallv #'attacks (aref q i) (aref q j) (- j i)))))))
  (all-values (solution (coerce q 'list) (reorder #'linear-force)))))
```

Figure 2: Two SCREAMER programs for solving the N-Queens problem. The first uses backtracking search while the second uses the constraint package.

```
(defun member-of (list)
 (if (null list) (fail) (either (first list) (member-of (rest list)))))

(defstruct (node (:conc-name nil) (:print-function print-node))
  name next-nodes (marked? nil) (visits 0))

(defun print-node (node stream print-level)
 (declare (ignore print-level))
 (princ (name node) stream))

(defun simple-path (u v)
 (if (marked? u) (fail))
 (local (setf (marked? u) t))
 (either (progn (unless (eq u v) (fail)) (list u))
         (cons u (simple-path (member-of (next-nodes u)) v))))

(defun k-simple-path (u v k)
 (if (= (visits u) k) (fail))
 (local (setf (visits u) (1+ (visits u))))
 (either (progn (unless (eq u v) (fail)) (list u))
         (cons u (k-simple-path (member-of (next-nodes u)) v k))))
```

Figure 3: A Screamer program for finding simple paths in a graph.

using nothing more than the facilities already provided by Common Lisp and the non constraint package portions of Screamer. This is in striking contrast to constraint logic programming languages which cannot be implemented efficiently on top of existing Prolog implementations but rather necessitate creating a new language implementation.

To illustrate the essential organization of the constraint package and how it is implemented on top of nondeterministic Common Lisp, figure 4 contains a greatly simplified version of the constraint package which supports only boolean constraints. This version is thus analogous to what is commonly called a truth maintenance system or TMS. Variables have two slots, a `value` which is either `t`, `nil` or `:unassigned`, and a list of `noticers`, procedures to be called to perform constraint propagation when a variable is assigned a value. The functions `notv` and `andv` take variables as arguments and create and return a new variable constrained by the appropriate relation to the arguments. The constraints are implemented via noticers attached both to the argument and result variables. The function `set-value` is used to set the value of a variable. It first checks whether the variable can be set to the given value, failing if it cannot. If the variable was not previously bound, `set-value` calls its noticers after setting its value in order to perform constraint propagation. Note that the functions `notv`, `andv` and `set-value` are all deterministic. This is in keeping with the requirement that constraint propagation be a fast operation. Nonetheless, it is an incomplete method for finding solutions to propositional satisfiability problems. Many truth maintenance systems rely solely on constraint propagation and are thus incomplete. Figure 4 shows how a TMS built in Screamer can circumvent this problem by providing a hook into backtracking search. The function `solution` is such a hook which nondeterministically assigns values to unassigned variables, interleaving constraint propagation with each assignment. The final expression in figure 4 shows how this TMS can be used to find all solutions to any propositional satisfiability problem.

Figure 5 contains a program for learning the syntactic categories of words appearing in a corpus of sentences. It is a problem first proposed and solved by Rayner et al. (1988). The basic idea is that you are presented with a context free grammar and a corpus of sentences generated by that grammar but do not know the syntactic categories of the words appearing in the corpus. One can find a lexicon mapping words to categories by nondeterministically enumerating all possible mappings and filtering out the ones which do not allow the sentences to be parsed by the grammar. The key to making this tractable is to order the search so as to delay the process of making nondeterministic choices for lexical entries for as long as possible.

Figure 5 shows a Screamer version of the original Prolog solution. The variable `*grammar*` contains a list of context free rules. The function `lhs` returns the category on the left hand side of a rule while the function `rhs` returns the list of categories forming the right hand side of a rule. The function `parse` returns `t` if a list of *words* can be parsed as a phrase of a given *category* and fails otherwise. It is essentially a simple top-down left-to-right nondeterministic parser. It calls `parse-rule` which nondeterministically tries to parse the *words* using every grammar rule which contains *category* as its left hand side. The core of the parser is the function `parse-categories`. It takes the right hand side of a rule and nondeterministically splits a list of words into sub-phrases to assign to each category in the right hand side, in every possible way, and attempts to recursively parse each sub-phrase.

So far this is a straightforward parser. Nothing about this parser is particular to the problem of category acquisition. The first definition for the function `category` completes the definition of this parser. By replacing this definition with the second one, the parser is transformed into a language acquisition device. While the first definition returns the categories associated with a word derived from a pre-existing lexicon, the second definition creates that lexicon on the fly. It enforces the monosemy constraint by returning the category of a word which is already in the lexicon. New lexical entries are added as new words are encountered, nondeterministically assigning categories to each new word. Notice how lexical entries are added by performing local side effects to a hash table, something possible in Screamer but not in Prolog. The last expression in figure 5 illustrates how to solve a given category acquisition puzzle by forming a series of calls to the parser, one for each sentence in the corpus, and

```lisp
(defstruct (boolean-variable (:conc-name nil)) (value :unassigned) noticers)

(defun notb (x)
 (let ((z (make-boolean-variable)))
  (local (push #'(lambda () (set-value x (not (value z)))) (noticers z))
         (push #'(lambda () (set-value z (not (value x)))) (noticers x)))
  z))

(defun andb (x y)
 (let ((z (make-boolean-variable)))
  (local
   (push #'(lambda ()
            (cond ((value x)
                   (unless (eq (value y) :unassigned) (set-value z (value y)))
                   (unless (eq (value z) :unassigned) (set-value y (value z))))
                  (t (set-value z nil))))
         (noticers x))
   (push #'(lambda ()
            (cond ((value y)
                   (unless (eq (value x) :unassigned) (set-value z (value x)))
                   (unless (eq (value z) :unassigned) (set-value x (value z))))
                  (t (set-value z nil))))
         (noticers y))
   (push #'(lambda ()
            (cond ((value z) (set-value x t) (set-value y t))
                  (t (if (eq (value x) t) (set-value y nil))
                     (if (eq (value y) t) (set-value x nil)))))
         (noticers z))
   z)))

(defun orb (x y) (notb (andb (notb x) (notb y))))

(defun set-value (variable value)
 (cond ((eq (value variable) :unassigned)
        (local (setf (value variable) value))
        (dolist (noticer (noticers variable)) (funcall noticer)))
       (t (unless (eq (value variable) value) (fail)))))

(defun boolean-solution (variables)
 (if (null variables)
     '()
     (let ((variable (first variables)))
      (when (eq (value variable) :unassigned)
       (set-value variable (either t nil)))
      (cons (value variable) (boolean-solution (rest variables))))))

(defun sat-problem ()
 (all-values
  (let ((x (make-boolean-variable))
        (y (make-boolean-variable))
        (z (make-boolean-variable)))
   (set-value (andb (orb x (notb y)) (orb y (notb z))) t)
   (boolean-solution (list x y z)))))
```


Figure 4: A Screamer program for solving propositional satisfiability problems using a combination of constraint propagation and backtracking search.

```
(defvar *grammar* '((s np vp)
                    (np det n)
                    (np n)
                    (vp v)
                    (vp v np)
                    (vp v np np)
                    (vp v pp)
                    (vp v np pp)
                    (pp p np)))

(defun lhs (rule) (car rule))

(defun rhs (rule) (cdr rule))

(defun categories (grammar)
 (remove-duplicates
  (set-difference (reduce #'append grammar) (mapcar #'first grammar)
                  :test #'eq)
  :test #'eq))

(defun parse-categories (categories words1 &optional words2)
 (if (null categories)
     (if (and (null words1) (null words2)) t (fail))
     (either (progn (parse (first categories) words1)
                    (parse-categories (rest categories) words2))
             (if (null words1)
                 (fail)
                 (parse-categories
                  categories
                  (reverse (rest (reverse words1)))
                  (append (last words1) words2))))))

(defun parse-rule (category words rules)
 (if (null rules)
     (fail)
     (either (if (eq (lhs (first rules)) category)
                 (parse-categories (rhs (first rules)) words)
                 (fail))
             (parse-rule category words (rest rules)))))

(defun parse (category words)
 (if (null (rest words))
     (if (eq category (category (first words))) t (fail))
     (parse-rule category words *grammar*)))

(defun category (word) (member-of (categories word)))

(defun category (word)
 (declare (special lexicon))
 (let ((category (gethash word lexicon)))
  (if category
      category
      (local (setf (gethash word lexicon)
                   (member-of (categories *grammar*)))))))

(defun grow-up ()
 (all-values
  (let ((lexicon (make-hash-table :test #'eq)))
   (declare (special lexicon))
   (progn (parse 's '(the cup slid from john to mary))
          (parse 's '(john walked to the table)))
   (iterate (for (word category) in-hashtable lexicon)
            (format t "~%~S:  ~S" word category)))))
```

Figure 5: A Screamer program for learning the syntactic categories of words.

backtracking until a solution is found.

Siskind (1990) extended the work of Rayner et al. (1988) in a number of ways. One important extension was using a constraint propagation based method rather than a pure backtracking method to solve the puzzle. The key ideas behind this method are illustrated in figure 6. This is a greatly simplified version of the program described in Siskind (1990).[8] This program was derived from the program in figure 5 by changing only those parts that appear in upper case letters. Like the program in figure 1, this illustrates a typical cliché for converting backtracking programs into constraint propagation based ones. In the derived program, the functions `parse-categories`, `parse-rule`, `parse` and `category` are all deterministic. They read in a corpus and construct a large constraint network corresponding to the category acquisition puzzle. The last expression in figure 6 triggers the constraint solving process. It is the only nondeterministic expression in the program.

Crossword puzzles are an interesting problem for illustrating constraint satisfaction techniques. Consider the problem of consistently fitting a subset of a given set of words into a given crossword puzzle without the clues. Figure 7 shows a sample crossword puzzle and set of words. Puzzles such as these can easily be solved using the SCREAMER constraint package. Figure 9 gives a SCREAMER program for solving crossword puzzles. A variable is created for each across and down entry which we will call placements. Each variable is given a fixed domain ranging over all words of the requisite length. A constraint is established for each pair of variables that represents two intersecting placements to enforce the requirement that they contain the same letter at their intersection point. The function `crossword-variables` creates the variables and asserts the constraints between them. It is a deterministic function. The function `crosswordv` is a nondeterministic function which initiates the search for a solution.

The arithmetic constraint primitives of the SCREAMER constraint package can be used to find solutions to complex systems of nonlinear equations and inequalities over the reals, as well as integer programming problems. Figures 10 illustrates how this can be done. Problems over the reals must be solved using `static-ordering` and `divide-and-conquer-force`. Integer programming problems can be solved using either `static-ordering` or `reorder` as well as either `linear-force` or `divide-and-conquer-force` though the later usually will yield better performance.

Our final example is an illustration of a standard cliché for translating PROLOG programs into SCREAMER. Figure 11 shows the standard `append` function in PROLOG, along with a translation of that PROLOG program into SCREAMER. Automation of this translation is quite straightforward. A future release of SCREAMER will have a facility for doing this. This will allow complete inter-operability between COMMON LISP and PROLOG.

13 Using Screamer

You must include the following three expressions at the top of every file which uses SCREAMER.

```
(in-package :my-package)
(use-package '(:lisp :screamer))
(shadowing-import '(screamer::defun))
```

You may replace `:my-package` with the desired package name for your file and have that package use any other packages that you wish. SCREAMER must be loaded into your COMMON LISP environment before you attempt to compile or load any file which uses SCREAMER.

[8]The program described in Siskind (1990) was written prior to the development of SCREAMER. Siskind (1991) describes continued work along these lines, including a newer program which is written in SCREAMER, though it uses only nondeterministic LISP and not the constraint package.

```
(defun parse-categoriesv (categories words1 &optional words2)
 (if (null categories)
     (if (and (null words1) (null words2)) t NIL)
     (ORV (progn (parsev (first categories) words1)
                 (parse-categoriesv (rest categories) words2))
          (if (null words1)
              NIL
              (parse-categoriesv
               categories
               (reverse (rest (reverse words1)))
               (append (last words1) words2))))))

(defun parse-rulev (category words rules)
 (if (null rules)
     NIL
     (ORV (if (eq (lhs (first rules)) category)
              (parse-categoriesv (rhs (first rules)) words)
              NIL)
          (parse-rulev category words (rest rules)))))

(defun parsev (category words)
 (if (null (rest words))
     (EQUALV CATEGORY (CATEGORYV (FIRST WORDS)))
     (parse-rulev category words *grammar*)))

(defun categoryv (word)
 (declare (special lexicon))
 (let ((category (gethash word lexicon)))
  (if category
      category
      (setf (gethash word lexicon) (MEMBER-OFV (categories *grammar*))))))

(defun grow-upv ()
 (let ((lexicon (make-hash-table :test #'eq)))
  (declare (special lexicon))
  (ASSERT! (ANDV (parsev 's '(the cup slid from john to mary))
                 (parsev 's '(john walked to the table))))
  (all-values
   (FUNCALL (REORDER #'LINEAR-FORCE)
            (ITERATE (FOR (WORD CATEGORY) IN-HASHTABLE LEXICON)
                     (DECLARE (IGNORE WORD))
                     (COLLECT CATEGORY)))
   (iterate (for (word category) in-hashtable lexicon)
            (format t "~%~S:  ~S" word category)))))
```

Figure 6: A constraint-based SCREAMER program for learning the syntactic categories of words.

ad	al	alas	aloha	art	at
atl	bags	bang	base	bore	coat
dad	dart	dime	dine	dive	do
eh	elf	er	evade	even	fan
fee	fine	gate	goat	happy	hares
hem	hide	hire	hive	hoe	hone
inn	largest	learned	lee	lemons	lid
lilac	lip	lo	load	mates	mile
mirror	mist	moon	more	oak	olive
ore	pans	paris	pay	pea	pedal
penny	pier	pile	pins	pits	raise
rips	roe	ropes	roy	salads	see
slam	slat	some	spot	steer	stew
tag	tame	tan	tank	tea	tee
tie	tigers	tire	to	toe	wager
wave	wider	win	wires		

Figure 7: A sample crossword puzzle which can be solved by the the program in figure 8.

```
(defun row (placement) (first placement))

(defun column (placement) (second placement))

(defun direction (placement) (third placement))

(defun placement-length (placement) (fourth placement))

(defun intersect? (placement1 placement2)
 (and
  (not (eq (direction placement1) (direction placement2)))
  (if (eq (direction placement1) 'across)
      (and (>= (row placement1) (row placement2))
           (<= (row placement1)
               (+ (row placement2) (placement-length placement2) -1))
           (>= (column placement2) (column placement1))
           (<= (column placement2)
               (+ (column placement1) (placement-length placement1) -1)))
      (and (>= (row placement2) (row placement1))
           (<= (row placement2)
               (+ (row placement1) (placement-length placement1) -1))
           (>= (column placement1) (column placement2))
           (<= (column placement1)
               (+ (column placement2) (placement-length placement2) -1))))))

(defun consistent-placements?
    (placement1 placement2 placement1-word placement2-word)
 (or (not (intersect? placement1 placement2))
     (if (eq (direction placement1) 'across)
         (char= (aref placement1-word
                      (- (column placement2) (column placement1)))
                (aref placement2-word
                      (- (row placement1) (row placement2))))
         (char= (aref placement2-word
                      (- (column placement1) (column placement2)))
                (aref placement1-word
                      (- (row placement2) (row placement1)))))))

(defun word-of-length (n dictionary)
 (if (null dictionary)
     (fail)
     (if (= (length (first dictionary)) n)
         (either (first dictionary) (word-of-length n (rest dictionary)))
         (word-of-length n (rest dictionary)))))

(defun check-placement (placement word solution)
 (dolist (placement-word solution)
  (if (not (consistent-placements?
            (first placement-word) placement (second placement-word) word))
      (fail))))
```



```
(defun choose-placement (placements solution)
 (block exit
  (dolist (placement placements)
   (if (some #'(lambda (placement-word)
                (intersect? (first placement-word) placement))
             solution)
       (return-from exit placement)))
  (return-from exit (first placements))))

(defun crossword (placements dictionary &optional solution)
 (if (null placements)
     solution
     (let* ((placement (choose-placement placements solution))
            (word (word-of-length (placement-length placement) dictionary)))
      (check-placement placement word solution)
      (crossword (remove placement placements)
                 dictionary
                 (cons (list placement word) solution)))))
```

Figure 8: A backtracking SCREAMER program for solving crossword puzzles.

```
(defun crossword-variables (placements dictionary)
 (iterate
  (with variables =
        (iterate
         (for placement in placements)
         (collect
          (member-ofv
           (all-values
            (let ((word (member-of dictionary)))
             (unless (= (length word)
                        (placement-length placement))
              (fail))
             word))))))
  (for (variable1 .  remaining-variables) on variables)
  (for (placement1 .  remaining-placements) on placements)
  (iterate
   (for variable2 in remaining-variables)
   (for placement2 in remaining-placements)
   (if (intersect? placement1 placement2)
       (let ((placement1 placement1)
             (placement2 placement2))
        (assert!
         (funcallv #'(lambda (word1 word2)
                      (consistent-placements?
                       placement1 placement2 word1 word2))
                   variable1
                   variable2)))))
  (finally (return variables))))

(defun crosswordv (placements dictionary)
 (mapcar #'list
         placements
         (solution (crossword-variables placements dictionary)
                   (reorder #'linear-force))))
```

Figure 9: A constraint-based Screamer program for solving crossword puzzles.

```
(one-value
 (let ((x (real-betweenv -1e40 1e40))
       (y (real-betweenv -1e40 1e40))
       (z (real-betweenv -1e40 1e40)))
  (assert! (andv (orv (=v (+v (*v 4 x x y)
                              (*v 7 y z z)
                              (* 6 x x z z))
                          2)
                      (=v (+v (*v 3 x y)
                              (*v 2 y y)
                              (*v 5 x y z))
                          -4))
                 (>=v (*v (+v x y) (+v y z)) -5))))
  (solution (list x y z) (static-ordering #'divide-and-conquer-force)))
```

Figure 10: A SCREAMER program for solving nonlinear inequalities.

```
append([],x,x).
append([a|x],y,[a|z]):-append(x,y,z).

(defun prolog-append (x y z)
 (either (progn (assert! (equalv x nil))
                (assert! (equalv y z)))
         (let ((x1 (make-variable))
               (y1 (make-variable))
               (z1 (make-variable))
               (a (make-variable)))
          (assert! (equalv x (cons a x1)))
          (assert! (equalv y y1))
          (assert! (equalv z (cons a z1)))
          (prolog-append x1 y1 z1))))

(defun split-list ()
 (all-values
  (let ((x (make-variable))
        (y (make-variable)))
   (prolog-append x y '(a b c d))
   (print (list x y)))))
```

Figure 11: A method for translating PROLOG programs into SCREAMER

$$[(\texttt{either})]_{\mathrm{c}} \leadsto (\texttt{throw 'fail nil}) \tag{1}$$

$$[(\texttt{either}\ e)]_{\mathrm{c}} \leadsto [e]_{\mathrm{c}} \tag{2}$$

$$[(\texttt{either}\ e_1 \ldots e_n)]_{\mathrm{c}} \leadsto \begin{array}{l} (\texttt{let}((a\ c)) \\ \ (\texttt{catch 'fail}\ [e_1]_{\mathrm{a}}) \\ \ \vdots \\ \ (\texttt{catch 'fail}\ [e_{n-1}]_{\mathrm{a}}) \\ \ [e_n]_{\mathrm{a}}) \end{array} \tag{3}$$

$$[(\texttt{fail})]_{\mathrm{c}} \leadsto (\texttt{throw 'fail nil}) \tag{4}$$

$$[(\texttt{nondeterministic-setf}\ v\ e)]_{\mathrm{c}} \leadsto [e]\ \begin{array}{l} (\lambda(d) \\ \ (\texttt{let}((b\ v)) \\ \ \ (\texttt{unwind-protect} \\ \ \ \ (\texttt{progn}\ (\texttt{setf}\ v\ d)\ (\texttt{funcall c}\ d)) \\ \ \ \ (\texttt{setq}\ v\ b)))) \end{array} \tag{5}$$

$$[(\texttt{quote}\ x)]_{\mathrm{c}} \leadsto (\texttt{funcall}\ c\ (\texttt{quote}\ x)) \tag{6}$$

$$[(\texttt{function}\ x)]_{\mathrm{c}} \leadsto (\texttt{funcall}\ c\ (\texttt{function}\ x)) \tag{7}$$

$$[(\texttt{progn})]_{\mathrm{c}} \leadsto (\texttt{funcall}\ c\ \texttt{nil}) \tag{8}$$

$$[(\texttt{progn}\ e_1)]_{\mathrm{c}} \leadsto [e_1]_{\mathrm{c}} \tag{9}$$

$$[(\texttt{progn}\ e_1 \ldots e_n)]_{\mathrm{c}} \leadsto [e_1]_{(\lambda(d_1)\ldots[e_n]_{\mathrm{c}}\cdots)} \tag{10}$$

$$[(\texttt{setq}\ v\ e)]_{\mathrm{c}} \leadsto [e]_{(\lambda(d)\ (\texttt{setq}\ v\ d)\ (\texttt{funcall}\ c\ d))} \tag{11}$$

$$[(\texttt{if}\ e_1\ e_2\ e_3)]_{\mathrm{c}} \leadsto (\texttt{let}((a\ c))\ [e_1]_{(\lambda(\mathrm{d})\ (\texttt{if}\ \mathrm{d}\ [\mathrm{e}_2]_{\mathrm{a}}\ [\mathrm{e}_3]_{\mathrm{a}}))}) \tag{12}$$

$$[(f\ e_1 \ldots e_n)]_{\mathrm{c}} \leadsto [e_1]\ \ (\lambda(d_1)\ \ldots [e_n]_{(\lambda(\mathrm{d_n})\ (\mathrm{f\ c\ d_1 \ldots d_n}))\ldots})\ \text{when } f \text{ is nondeterministic} \tag{13}$$

$$[(f\ e_1 \ldots e_n)]_{\mathrm{c}} \leadsto [e_1]\ \ (\lambda(d_1)\ \ldots [e_n]_{(\lambda(\mathrm{d_n})\ (\texttt{funcall}\ \mathrm{c\ (f\ d_1 \ldots d_n)}))\ldots})\ \text{when } f \text{ is deterministic} \tag{14}$$

$$[x]_{\mathrm{c}} \leadsto (\texttt{funcall}\ c\ x)\text{when} x \text{is a variable or is self evaluating} \tag{15}$$

Table 1: Some of the CPS conversion rules used by SCREAMER.

14 Implementation

SCREAMER implements backtracking by performing CPS (Continuation Passing Style) conversion on expressions in nondeterministic contexts. Without CPS conversion, returning values from functions is handled by the underlying LISP function return mechanism. A CPS converted expression returns its value by calling a continuation with the returned value as its single argument. This frees up the underlying LISP function return mechanism to be usurped to handle backtracking. Thus in SCREAMER, a nondeterministic function fails by returning and returns by calling its continuation.

If e and c are non-CPS-converted expressions expressions, we denote by $[e]_{\mathrm{c}}$, the CPS conversion of e so that it calls the continuation denoted by c with its result. Some of the transformations rules for performing CPS conversion of expressions allowed in nondeterministic contexts is given in table 1.

Some comments pertaining to these CPS conversion rules are worthwhile. First, any variable in italics appearing on the right hand side of a rule and not the left hand side denotes a new uninterned symbol created by `gensym`. Second, rules 3 and 12 introduce new variables a to factor out the common continuation subexpression and avoid the potential exponential growth in expression size that would occur if such common subexpression factoring were not performed. As an optimization, if this variable a would only rename another created variable, then the `let` introducing a is β-converted. Third, while backtracking could have been implemented as simple function return, instead choice points are set up as `catch` frames with the tag `fail` and backtracking is implemented via a `throw` to the most recent catch frame named `fail`. Presumably, unwinding a stack via `throw` is faster than returning through all of the intermediate frames. Fourth, the dummy variables $d_1 \ldots d_{n-1}$ in rule 10 are never used. To avoid being flagged with warnings by some COMMON LISP compilers, they are declared as `ignore` variables. Fifth, all of the closures created by the lambda expressions created by the CPS conversion can be stack allocated by COMMON LISP compilers which provide such a capability. As the Symbolics compiler supports stack allocated closures only when lambda expression are declared as such via the declaration `sys:downward-function`, such a declaration is added to all lambda expressions generated by CPS conversion.[9]

The CPS converter performs one additional optimization. Stated in their simple form given in table 1, the CPS conversion rules generate a great deal of redundant lambda expressions which immediately get funcalled. To remove such redundant lambda expressions, the CPS converter incorporates a β-conversion rule. The `let` bindings for the variable a created by rules 3 and 12 can sometimes thwart the β-conversion optimization. The aforementioned optimization whereby `let` bindings which simply bind a to a previously created variable are β-converted may circumvent this thwarting and allow further β-conversion by the current rule.

The actual implementation uses a more efficient CPS conversion algorithm. First, it performs a more efficient CPS conversion on special forms which have only deterministic sub-expressions. Second, it performs a more efficient CPS conversion for the special forms `if`, `progn` and `setq` when it can be shown that their values are not being used and they are only being called for side effect.

15 Intellectual Heritage

Nondeterministic LISP is not new. The addition of a nondeterministic choice operator (once called `amb`) to LISP dates back to McCarthy (1963). Clinger (1982) discusses the difficulties involved in giving a formal semantics to a nondeterministic choice operator in LISP. DEPENDENCY DIRECTED LISP (also known as DDL) (Chapman, unpublished) was an implementation of nondeterministic LISP used to implement TWEAK (Chapman, 1985), a non-linear constraint-posting planner. DDL recorded dependency information during execution to support selective backtracking. SCHEMER (Zabih, 1987; Zabih, McAllester and Chapman, 1987; Zabih, McAllester and Chapman, forthcoming) was an interpreter for nondeterministic SCHEME that recorded and analyzed dependency information to perform both selective backtracking and lateral pruning. SCHEMER and DDL were both interpreters to support retaining the dependency information needed for intelligent backtracking. Because SCREAMER uses chronological backtracking, it can macro-expand into ordinary COMMON LISP which is then compiled into efficient code. LAMBEX (McAllester, unpublished) and new ONTIC (McAllester, unpublished) are dialects of LISP which incorporate a nondeterministic choice operator. They are intended to be used as a declarative input language for stating theorems to a proof checker and no operational evaluators have been constructed for them. Haynes (1987) describes how a nondeterministic choice operator can be added to SCHEME using the `call/cc` function.

The techniques used for implementing backtracking in SCREAMER are analogous to those used when compiling PROLOG into LISP (Kahn, 1982; Kahn, 1983; Kahn and Carrlson, 1984; Siskind, 1989). CPS conversion was used in the RABBIT compiler for SCHEME (Steele and Sussman, 1976; Steele, 1977).

[9] Since efficient CPS conversion relies on being able to stack allocate closures to avoid needing to garbage collect the many closures created during the execution of nondeterministic functions, in would be nice if COMMON LISP incorporated the option of a `sys:downward-function` declaration as a standard feature.

Constraint solvers based on forward checking (sometimes called constraint propagation) also have a long history. Some of the earliest and best known of such systems were SKETCHPAD (Sutherland, 1963), THINGLAB (Borning, 1979), CONSYS (Steele, 1980; Sussman and Steele, 1980) and MAGRITTE (Gosling, 1983). Since constraint propagation is an incomplete technique for solving systems of constraints, systems such as these which rely primarily on constraint propagation are incapable of solving many problems they are given. The constraint handling mechanism used in SCREAMER was adapted from the one used in CHIP (Van Hentenryck, 1989). The novel approach taken by CHIP and SCREAMER is to combine backtracking search with constraint propagation to yield a complete constraint solver. Unlike CHIP which uses linear programming techniques to solve systems of numeric constraints, SCREAMER uses range propagation and supports a divide and conquer approach to solving nonlinear numeric constraint problems. Van Hentenryck's book gives an excellent historical overview of the development of constraint satisfaction techniques and languages and contains an extensive bibliography of that field.

16 Obtaining Screamer

SCREAMER is intended to be portable and should run under any COMMON LISP implementation. At the M. I. T. Artificial Intelligence laboratory, the source code for SCREAMER is available from the file:

```
/src/local/lisplib/src/screamer/screamer.lisp
```

Compiled code for various combinations of machines and COMMON LISP compilers is available from the files:

`/src/local/lisplib/`*machine*`/screamer.`*binary*

where *machine* is the directory for storing compiled LISP code for that architecture and *binary* is the standard extension for that architecture. The following file is useful if you attempt to port SCREAMER to a new machine or COMMON LISP implementation:

```
/src/local/lisplib/src/screamer/primordial.lisp
```

It contains a series of examples designed to exercise SCREAMER. Simply compile that file and run the function `(prime-ordeal)`. It should return `t` if the SCREAMER port was successful and give an error message if not. Additionally, a file containing all of the examples from this manual is available as:

```
/src/local/lisplib/src/screamer/screams.lisp
```

Finally, a postscript file containing this manual is available from the file:

```
/src/local/lisplib/doc/screamer.ps
```

From outside M. I. T., SCREAMER is available by public FTP from the host:

```
ftp.ai.mit.edu
```

in the directory:

```
/com/ftp/pub/screamer/
```

In that directory you will find the files:

```
screamer.lisp
primordial.lisp
screams.lisp
screamer.ps
```

You will have to compile SCREAMER yourself for whichever machine and COMMON LISP implementation you use.

We maintain several mailing lists pertaining to SCREAMER. The mailing list `Info-Screamer` is a channel from the developers to the users and contains announcements of enhancements and bug fixes. To be added to that mailing list, send mail to:

`Info-Screamer-Request@AI.MIT.EDU`

If you detect a bug in SCREAMER please send mail to:

`Bug-Screamer@AI.MIT.EDU`

Please do not send mail directly to the author as the `Bug-Screamer` mailing list is archived separately to aid in maintaining SCREAMER. We also ask that you report *all* bugs whether or not you need them fixed and whether or not you fix them yourself. This will assist us in helping the entire user community. Finally, we ask that you send mail to `Bug-Screamer` whenever you attempt to port SCREAMER to a machine or COMMON LISP implementation not listed at the beginning of the file `screamer.lisp` whether or not that port is successful.

SCREAMER is available free of charge and without any restriction. All we ask is that you abide by the above policy for sending bug reports and also send mail to `Info-Screamer-Request` if you obtain a copy in any way so that we may keep track of who has obtained a copy and keep users informed of enhancements and bug fixes by way of the `Info-Screamer` mailing list.

Acknowledgments

Bug-DDL@AI.MIT.EDU

References

[1] Alan Hamilton Borning. THINGLAB—*A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, July 1979. Also available as Stanford Computer Science Department report STAN-CS-79-746 and as XEROX Palo Alto Research Center report SSL-79-3.

[2] David Chapman. Dependency-Directed LISP. Unpublished manuscript received directly from author.

[3] David Chapman. Planning for conjunctive goals. Master's thesis, Massachusetts Institute of Technology, January 1985. Also available as M. I. T. Artificial Inteligence Laboratory Technical Report 802.

[4] W. Clinger. Nondeterministic call by need is neither lazy nor by name. In *Proceedings of the ACM Conference on* LISP *and Functional Programming*, pages 226–234, 1982.

[5] James Gosling. *Algebraic Constraints*. PhD thesis, Carnegie-Mellon University, 1983.

[6] Christopher T. Haynes. Logic continuations. *Journal of Logic Programming*, 4:157–176, 1987.

[7] Nevin Heintze, Spiro Michaylov, and Peter Stuckey. CLP($\Re$) and some electrical engineering problems. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference*, pages 675–703, Cambridge, MA, May 1987. M. I. T. Press.

[8] Joxan Jaffar and Spiro Michaylov. Methodology and implementation of a CLP system. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference*, pages 196–218, Cambridge, MA, May 1987. M. I. T. Press.

[9] Kenneth M. Kahn. A partial evaluator of LISP written in PROLOG. In *Proceedings of the First Logic Programming Conference*, Marseille, France, 1982.

[10] Kenneth M. Kahn. Unique features of LISP machine PROLOG. UPMAIL Report 14, University of Uppsala, Sweden, 1983.

[11] Kenneth M. Kahn and M. Carlsson. How to implement PROLOG on a LISP machine. In J. A. Campbell, editor, *Implementations of* PROLOG, chapter 2, pages 117–134. Ellis Horwood, Chichester, 1984.

[12] David Allen McAllester. Lambex. Unpublished manuscript received directly from author.

[13] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programing and Formal Systems*. Elsevier North-Holland, Amsterdam, 1963.

[14] Manny Rayner, Åsa Hugosson, and Göran Hagert. Using a logic grammar to learn a lexicon. Technical Report R88001, Swedish Institute of Computer Science, 1988.

[15] Jeffrey Mark Siskind. The culprit pointer method for selective backtracking. Master's thesis, Massachusetts Institute of Technology, January 1989.

[16] Jeffrey Mark Siskind. Acquiring core meanings of words, represented as Jackendoff-style conceptual structures, from correlated streams of linguistic and non-linguistic input. In *Proceedings of the* 28^{th} *Annual Meeting of the Association for Computational Linguistics*, pages 143–156, University of Pittsburgh, Pittsburgh, PA, June 1990.

[17] Jeffrey Mark Siskind. Dispelling myths about language bootstrapping. In *The AAAI Spring Symposium Workshop on Machine Learning of Natural Language and Ontology*, pages 157–164, March 1991.

[18] Ivan E. Southerland. SKETCHPAD*: A Man-Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, January 1963.

[19] Guy Lewis Steele Jr. Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or lambda, the ultimate goto. A. I. Memo 443, M. I. T. Artificial Intelligence Laboratory, October 1977.

[20] Guy Lewis Steele Jr. *The Definition and Implementation of a Computer Programming Language Based on Constraints*. PhD thesis, Massachusetts Institute of Technology, August 1980. Also avilable as M. I. T. VLSI Memo 80–32 and as M. I. T. Artificial Inteligence Laboratory Technical Report 595.

[21] Guy Lewis Steele Jr. and Gerald Jay Sussman. Lambda, the ultimate imperative. A. I. Memo 353, M. I. T. Artificial Intelligence Laboratory, March 1976.

[22] Gerald Jay Sussman and Guy Lewis Steele Jr. CONSTRAINTS—a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, 1980. Also available as M. I. T. Artificial Intelligence Laboratory Memo 502A.

[23] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. M. I. T. Press, Cambridge, MA, 1989.

[24] Ramin D. Zabih. Dependency-directed backtracking in non-deterministic SCHEME. Master's thesis, Massachusetts Institute of Technology, January 1987.

[25] Ramin D. Zabih, David Allen McAllester, and David Chapman. Non-deterministic Lisp with dependency-directed backtracking. In *Proceedings of AAAI–87*, pages 59–64, July 1987.

[26] Ramin D. Zabih, David Allen McAllester, and David Chapman. Dependency-directed backtracking in non-deterministic Lisp. *Artificial Intelligence*, 1988. Submitted for publication.