GOO Reference Manual v27

for release 0.114

Jonathan Bachrach MIT AI Lab

March 14. 2002

1 Introduction

 \mathcal{GOO} is a dynamic type-based object-oriented language. It is designed to be simple, productive, powerful, extensible, dynamic, efficient and real-time. It heavily leverages features from many earlier languages. In particular, it attempts to be a simpler lisp-syntaxed Dylan [4], an object-oriented Scheme [3], and a lispified Cecil [2].

This manual is very preliminary and relies heavily on an understanding of Scheme and Dylan.

1.1 Notation

Throughout this document \mathcal{GOO} objects are described with definitions of the following form:

Name	Signature	\mathcal{N}	
Documentatio	n		

where the rightmost kind field has a one letter code as follows:

N	Notation	\mathcal{N}
L	Lexical	\mathcal{N}
S	Syntax	\mathcal{N}
G	Generic	\mathcal{N}
М	Method	\mathcal{N}
F	Function	\mathcal{N}
С	Class	\mathcal{N}
I	Instance	\mathcal{N}
K	Command	\mathcal{N}

1.2 Lexical Structure

The lexical structure is mostly the same as Scheme [3] with the notable exceptions being that identifiers can start with numeric digits if they are clearly distinguishable from floating point numbers and no syntax is provided for specifying improper lists. Vertical bars are tokenized immediately and separately and have special meaning within lists, providing syntactic sugar for typed variables.

The following is a very brief and incomplete description of how characters are tokenized into s-expressions, where s-expressions are either tokens or lists of s-expressions:

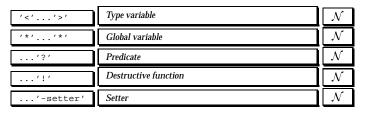
/* */	Nested comment	\mathcal{N}
//	Line comment	\mathcal{N}
. + - [0-9]+	Number	\mathcal{N}
#e #i #b #o #d #x	Special number	\mathcal{N}
#t #f	Logical	\mathcal{N}
#\name	Character	\mathcal{N}
[a-zA-Z0-9]+	Identifier	\mathcal{N}
()	List	\mathcal{N}
#()	Vector	\mathcal{N}
" "	String	\mathcal{N}
\ <i>c</i>	Character	\mathcal{N}
x t	Typed variable within list $\equiv (x t)$.	\mathcal{N}
#	Escaped vertical bar.	\mathcal{N}

1.3 Meta Syntax

 \mathcal{GOO} 's syntax is described almost entirely as \mathcal{GOO} patterns. \mathcal{GOO} patterns in turn are defined with a quasiquote metasyntax. Pattern variables are prefixed with a "," or ",®" to indicate the matching of one or many elements respectively. The default is for a pattern variable to match one or many s-expressions. Alternatively, a pattern variable's shape may be defined with another pattern. The 'name shape is builtin and matches only identifiers. The '['...']' metasyntax is used to indicate optional patterns, '...' is used to indicate zero or more of the preceding pattern element, and ## is used to denote infix string concatenation. Finally, in this manual, uppercase indicates a special form or macro.

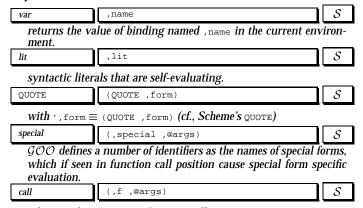
1.4 Conventions

The following naming conventions are used throughout this manual:



2 Expressions

Once tokenized, \mathcal{GOO} evaluates s-expressions in the usual lisp manner:

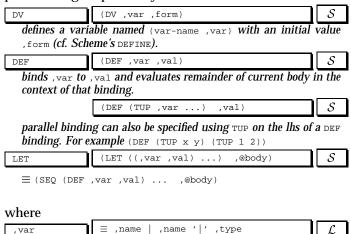


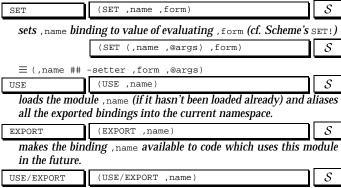
otherwise lists represent function calls.

3 Namespaces and Bindings

 \mathcal{GOO} is a lexically scoped language. Bindings contain values and are looked up by name. Lexical bindings are visible from only particular textual ranges in a program. Lexical bindings shadow bindings of the same name.

At the topmost level, \mathcal{GOO} provides simple modules that map from names to bindings. Each file introduces a new module with the same name as the file. Modules can import bindings exported by other modules.

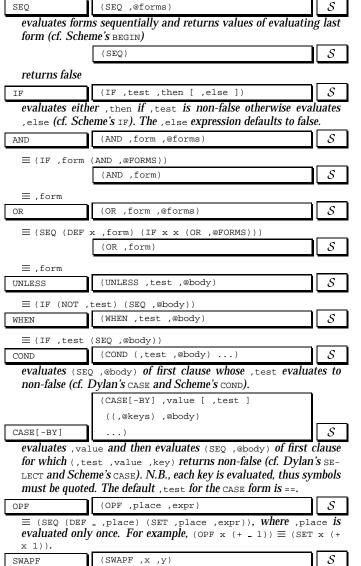




same as use plus reexports all imported bindings.

4 Program Control

 \mathcal{GOO} provides a variety of program control constructs including function calls, conditional execution, and nonlocal control flow.



 \equiv (SEQ (DEF tmp ,x) (SET ,x ,y) (SET ,y tmp)), where ,x and ,y are evaluated only once. (,f ,@args) evaluates ,f and then ,@args in left to right order and then calls ,f with the evaluated arguments. (REP ,name ((,var ,init) ...) ,@body) \mathcal{S} defines a recursive loop (cf., Dylan's iterate or Scheme's (Let \mathcal{S} (ESC ,name ,@body) evaluates (SEQ ,@body) with an exit function of a single parameter, x, bound to , name that if called, will cause ESC to return the value of x (cf. Dylan's block/return). It is illegal to call the exit function after the execution of the creating ESC form (i.e., no upward continuations) (FIN ,protected ,@cleanups) ensures that (SEQ ,@cleanups) is evaluated whether or not an ESC upwards exit is taken during the dynamic-extent of ,protected (cf. Dylan's block/cleanup form and CL's unwind-protect). The result of a fin form is the result of evaluating its protected form. (ASSERT ,test ,message ,@args) ASSERT

5 Types, Classes and Properties

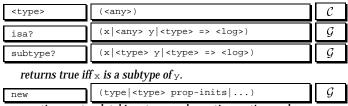
 \equiv (UNLESS ,test (ERROR ,message ,@args))

 \mathcal{GOO} types categorize objects. Types are first class. They are used to annotate bindings. Binding types restrict the type of objects bindable to associated bindings.

 \mathcal{GOO} supports the following types in order of specificity:

- Singleton types specify a unique instance,
- *Classes* and *properties* specify the structure, inheritance, and initialization of objects. Every object is a direct instance of a particular class,
- Product types specify a cross product of types,
- Subclass types specify a lineage of classes, and
- *Union* types specify a union of types.

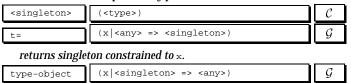
The basic type protocol is:



creation protocol taking type and creation options where propinits contains getter / initial value pairs.

5.1 Singletons

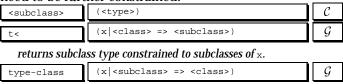
Singleton types match exactly one value using ==. Singletons are the most specific types.



object that singleton type matches.

5.2 Subclasses

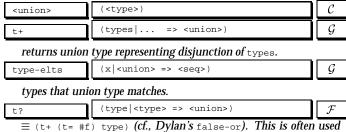
Subclass types match classes and their subclasses. They are quite useful in situations that involve class arguments that need to be further constrained.



object that subclass type matches.

5.3 Unions

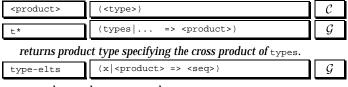
Union types represent the disjunction of types. In conjunction with singleton types, they can be used to represent C-style enums.



 \equiv (t+ (t= #f) type) (CI., Dylan's false-or). This is often used to widen a type to include the convenient false null.

5.4 Product

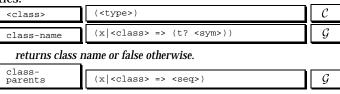
Product types represent tuples formed as the cartesian product of types. They are often used to describe multiple value return types.



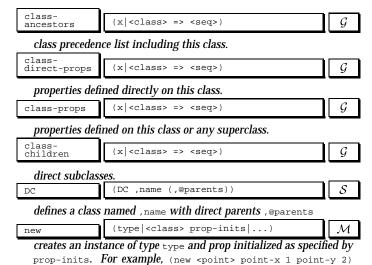
types that product type matches.

5.5 Classes

Classes are types that specify an inheritance relationship and can have associated structured data through properties.



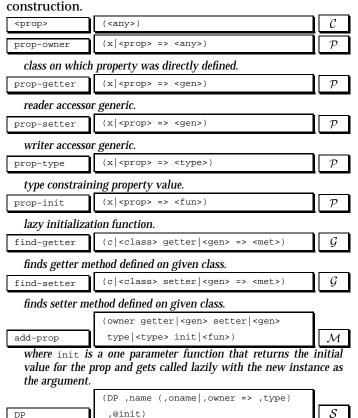
direct superclasses.



5.5.1 Properties

creates a point with x=1 and y=2.

Properties are named data associated with classes. Their values are accessed exclusively through generic functions, called getters and setters. Descriptions of properties are instances of <code>cprop></code>. Property values can either be specified at creation time with keyword arguments, by calling a property setter, or through a property initialization function called lazily the first time a getter is called if the property is otherwise uninitialized. Property initialization functions are called with a single argument, the object under construction.



add's a property to <code>,owner</code> with getter named <code>,name</code>, setter named <code>,name</code> ## "-setter", type <code>,type</code>, and optionally initial value <code>,init</code>. The initial value function is evaluated lazily when prop's value is first requested.

6 Functions

All operations in \mathcal{GOO} are functions.

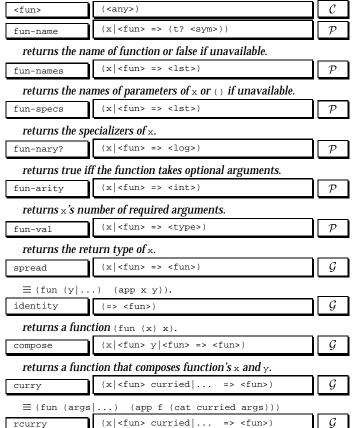
Functions accept zero or more arguments, and return one value. The parameter list of the function describes the number and types of the arguments that the function accepts, and the type of the value it returns.

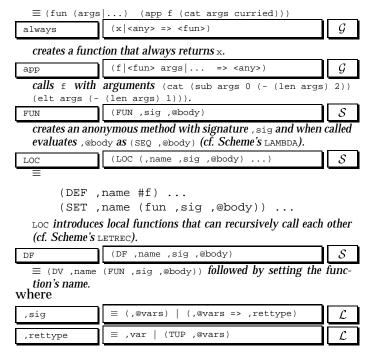
There are two kinds of functions, methods and generic functions. Both are invoked in the same way. The caller does not need to know whether the function it is calling is a method or a generic function.

A method is the basic unit of executable code. A method accepts a number of arguments, creates local bindings for them, executes an implicit body in the scope of these bindings, and then returns a value.

A generic function contains a number of methods. When a generic function is called, it compares the arguments it received with the parameter lists of the methods it contains. It selects the most appropriate method and invokes it on the arguments. This technique of method dispatch is the basic mechanism of polymorphism in \mathcal{GOO} .

All \mathcal{GOO} functions are objects, instances of <code><fun></code>. Generic functions are instances of <code><gen></code> and methods are instances of <code><met></code>.

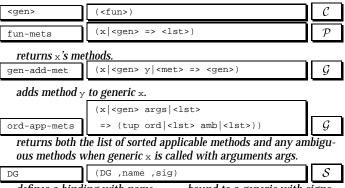




6.1 Generics

Generic functions provide a form of polymorphism allowing many implementation methods with varying parameter types, called *specializers*. Methods on a given generic function are chosen according to applicability and are then ordered by specificity. A method is applicable if each argument is an instance of each corresponding specializer. A method A is more specific than method B if all of A's specializers are subtypes of B's. During method dispatch three cases can occur:

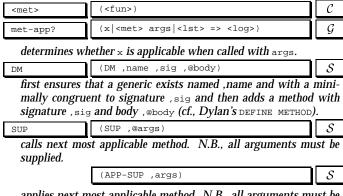
- if no methods are applicable then a no-applicablemethod error is signaled,
- if methods are applicable but are not orderable then an ambiguous-method error is signaled,
- if methods are applicable and are orderable then the most specific method is called and the next methods are established.



defines a binding with name ,name bound to a generic with signature ,sig.

6.2 Methods

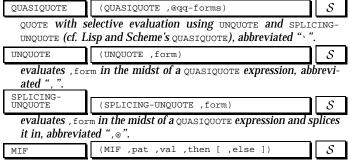
Methods are \mathcal{GOO} 's code objects. Methods can optionally be added to generics.



applies next most applicable method. N.B., all arguments must be supplied.

6.3 Macros

Macros provide a facility for extending the base syntax of \mathcal{GOO} . The design is based on quasiquote code templates and a simple list pattern matching facility.



is the "matching if", evaluating then with pattern variables bound to matched parts of value if matching succeeds and otherwise evaluates, else. The pattern is much the same as quasiquote and can contain either unquote'd variables or unquote-splicing variables. For example,

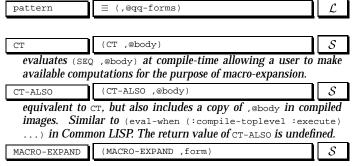
```
(MIF (,a,b)'(12) (lst a b)) \\ \rightarrow (12) and (MIF (,a,@b)'(12) (lst a b)) \\ \rightarrow (1(2)) (MATCH (MATCH,exp(,pat,val)...)) (MATCH (MATCH,exp(,pat,val)...)) (DS,pattern,@body) S (DS,pattern,@body) S (DS,pattern,pattern and expanding accord-
```

defines a macro matching pattern ,pattern and expanding according to ,@body. The pattern matching occurs as in MIF and makes available pattern variables during the evaluation of (SEQ ,@body). For example,

```
(DS (when ,test ,@body)
    '(if (not ,test) (seq ,@body)))

defines the when macro in \mathcal{GOO}.
```

where



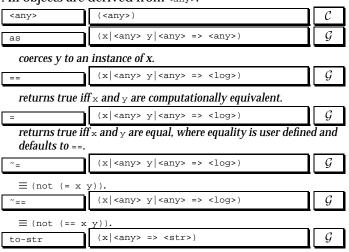
recursively expands macros in expression ,form.

Scalars

 \mathcal{GOO} provide a rich set of simple objects.

7.1 **Any**

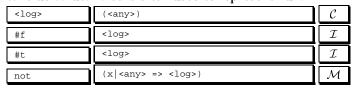
All objects are derived from <any>.



returns string representation of object.

7.2 Booleans

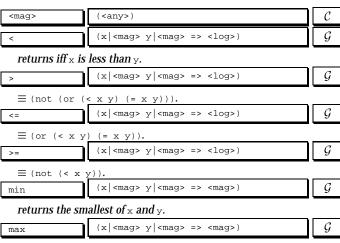
In \mathcal{GOO} , for convenience sake, true is often represented by anything that is not false, but #t is reserved for the canonical true value. False is often used to represent null.



 $\equiv (if x \#f x)$

Magnitudes 7.3

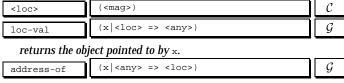
Magnitudes are totally orderable objects. Users are only required to implement < and =.



returns the largest of x and y.

7.4 Locatives

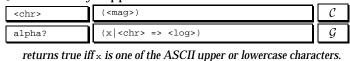
Locatives are word aligned pointers to memory.



returns address of particular object.

7.5 Characters

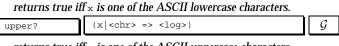
GOO currently supports 8 bit ASCII characters.



digit?	(x <chr> => <log>)</log></chr>	\mathcal{G}

 \mathcal{G}

returns true iff x is one of the ten ASCII numeric characters. (x|<chr> => <log>)



returns true iff x is one of the ASCII uppercase characters. (x|<chr> => <int>)

to-digit	(x <chr> => <int>)</int></chr>	\mathcal{G}
converts ascii	representation of digit to an integer one.	
to-lower	(x <chr> => <chr>)</chr></chr>	\mathcal{G}
notunna lorrion	age vargion of unnergoes alphabetic abarestars	othon

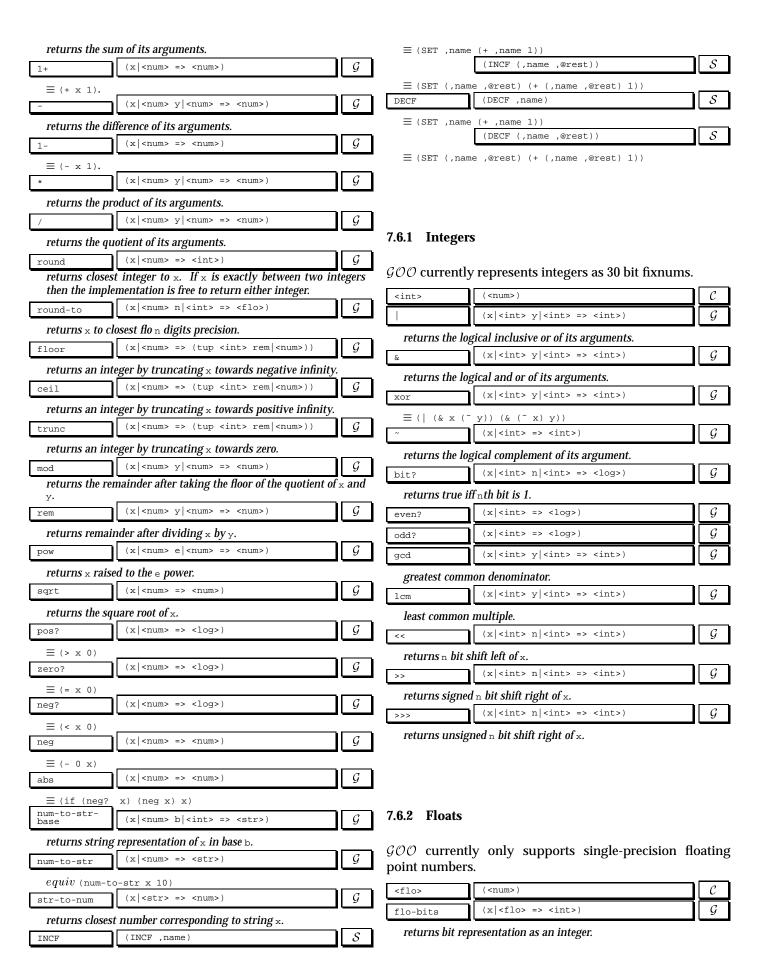
returns lowercase version of uppercase alphabetic characters otherwise returns x

(x|<chr> => <chr>) to-upper returns uppercase version of lowercase alphabetic characters other-

7.6 Numbers

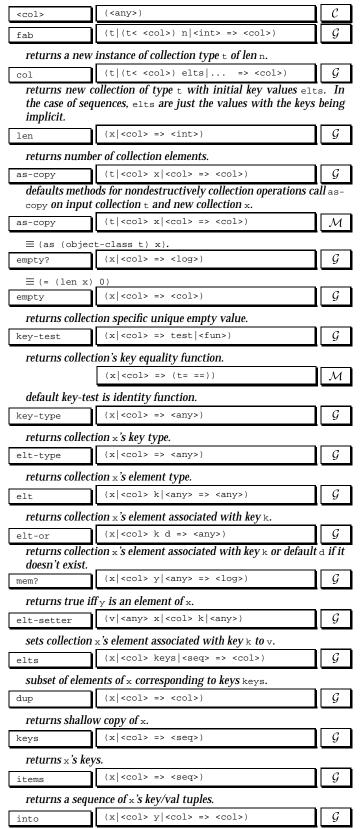
wise returns x.

<num></num>	(<mag>)</mag>	\mathcal{C}
+	(x <num> y <num> => <num>)</num></num></num>	\mathcal{G}



8 Collections

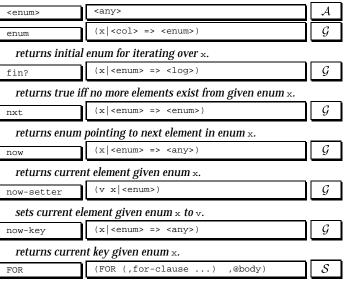
Collections are aggregate data structures mapping keys to values. Collections can be almost entirely defined in terms of an enumeration class.



replaces eleme	ents of x with elements of y .	
del	(x <col/> key <any> => <col/>)</any>	${\cal G}$
returns copy o	of x's without element corresponding to key.	
zap	(x <col/> => <col/>)	$\mathcal G$
returns empty	copy of x.	
fill	(x <col/> y <any> => <col/>)</any>	${\cal G}$
returns copy o	of × with all values being y.	
any?	(f <fun> x <col/> => <log>)</log></fun>	$\mathcal G$
returns true ii	If any of x 's element satisfies given predicate f .	
find	(f <fun> x <col/> => <any>)</any></fun>	$\mathcal G$
returns key as	sociated with first of x's values to satisfy predic	cate £.
find-or	(f <fun> x <col/> default => <any>)</any></fun>	${\cal G}$
returns key as or default if i	sociated with first of \mathbf{x} 's values to satisfy predinct found.	cate f
all?	(f <fun> x <col/> => <log>)</log></fun>	\mathcal{G}
returns true ii	If all of x 's elements satisfies given predicate f .	
fold	(f <fun> x <col/> => <col/>)</fun>	\mathcal{G}
<pre></pre>	(f (elt x 0) (elt x 1)) (elt x (- n 1)))	2)))
do	(f <fun> x <col/>)</fun>	$\mathcal G$
iterates functi	on f over values of x for side-effect.	
map	(f <fun> x <col/> => <col/>)</fun>	\mathcal{G}
iterates functi results.	on £ over values of given collections and collec	ts the

8.1 Enumerators

Enumerations are the foundation of collections and are designed to provide the convenience of Lisp's list interface (e.g., <code>null</code>, <code>car</code>, <code>cdr</code>) for all collections. In defining a new collection class, a user must implement at minimum an enumeration class and the enumeration protocol: <code>enum</code>, <code>fin?</code>, <code>nxt</code>, and <code>now</code>. For efficiency, users might choose to override more methods such as <code>len</code>, <code>elt</code>, <code>elt-setter</code>, etc.



parallel iteration over collections using enumerations. where

(,var ,col) ((tup ,keyvar ,var) ,col) \mathcal{L} .for-clause

specifies one parallel iteration over a collection , col binding successive values to , var and optionally keys to , keyvar.

8.2 **Accumulators**

Accumulators are the complement of enumerators and are the imperative version of fold. The default accumulator returns a list of all accumulated values:

```
(accuming (for ((e '(1 2 3 4 5)))
            (when (odd? e) (accum e))))
 ==> (1 3 5)
```

They can also be used for summing values etc:

```
(accuming-in (x|<int>)
  (for ((e '(1 2 3 4 5)))
    (when (odd? e) (accum-in x e)))
  (accumed x))
 ==> 9
```

<accum></accum>	<any></any>	\mathcal{A}
accum-add	(a <accum> x => <accum>)</accum></accum>	\mathcal{G}

returns an accumulator a augmented with element x.

 \mathcal{G} (a | <accum> => <any> accum-res

returns result of accumulations over a.

(init add | <fun> res | <fun>) \mathcal{G} accumer

returns a simple accumulator with that starts its value out with init, is augmented with add, and whose final value is computed with res.

accum-fab (t|<type> => <accum>) \mathcal{G}

returns a new type t specific accumulator.

(t|(t< <seq>) => <accum>) accum-fab \mathcal{M}

≡ (accumer '() pair (fun (x) (as t (rev! x)))) (t|(t= <int>) => <accum>) \mathcal{M} accum-fab

 \equiv (accumer 0 + identity) (ACCUMING-WITH ((,var ,accum) ...)

mechanism for accumulating objects using given accumulator into

, var. (ACCUMING-IN (,name $' \mid '$,type ...) ACCUMING-IN \mathcal{S}

(ACCUMING-IN (,name) ,@body)

 \equiv (ACCUMING-IN (,name '|' <1st>)

ACCUMING (ACCUMING ,@body) \mathcal{S}

(ACCUM-IN ,accum ,x) ACCUM-IN \mathcal{S}

folds ,x into accumulation in ,accum.

 \mathcal{S} ACCUM (ACCUM ,x) ,name).

(ACCUMED , name) $\mathcal S$ ACCUMED

 \equiv (accum-res ,name).

ACCUMING-WITH

8.3 Maps

Maps represent collections with explicit keys.

	1 3	
<map></map>	(<col/>)	\mathcal{C}
<tab></tab>	(<map>)</map>	\mathcal{C}

Tables are near constant-time aggregate data structures. Users can define their own tables by subclassing and overriding the key-test and tab-hash methods.

tab-growth-factor (x|<tab> => <flo>)

factor by which to grow capacity.

tab-growth-threshold (x|<tab> => <flo>)

when to grow based on proportion of total table capacity.

tab-shrink threshold (x|<tab> => <flo>) \mathcal{P}

when to shrink based on proportion of total table capacity.

(x|<tab> => hash|<fun>) \mathcal{G} tab-hash

returns key equality and hash function.

GC specific.

tab-gc-state (x|<tab> => <any>)

GC specific.

id-hash (x|<tab> => (tup hash gc-state)) \mathcal{G}

hash function based on pointer. Susceptible to rehash if objects are moved. The gc-state reflects movement.

(x|<tab> key => <tab>) \mathcal{G} del!

removes key from x.

(x|<tab> => <tab>) zap!

empties x.

8.4 **Sequences**

Sequences are collections with nonnegative integer keys.

<seq></seq>	(<col/>)	\mathcal{C}
1st	(x <seq> => <any>)</any></seq>	\mathcal{G}
\equiv (elt x 0)		
2nd	(x <seq> => <any>)</any></seq>	\mathcal{G}
\equiv (elt x 1)		
3rd	(x <seq> => <any>)</any></seq>	\mathcal{G}
\equiv (elt x 2)		
last	(x <seq> => <any>)</any></seq>	\mathcal{G}
≡ (elt x (-	(len x) 1))	
pos	(x <seq> v <any> => (t? <int>))</int></any></seq>	\mathcal{G}
finds position	of v in x else returns false.	
find	(x <seq> y <seq> => (t? <int>))</int></seq></seq>	\mathcal{G}

finds position of y in x else returns false.

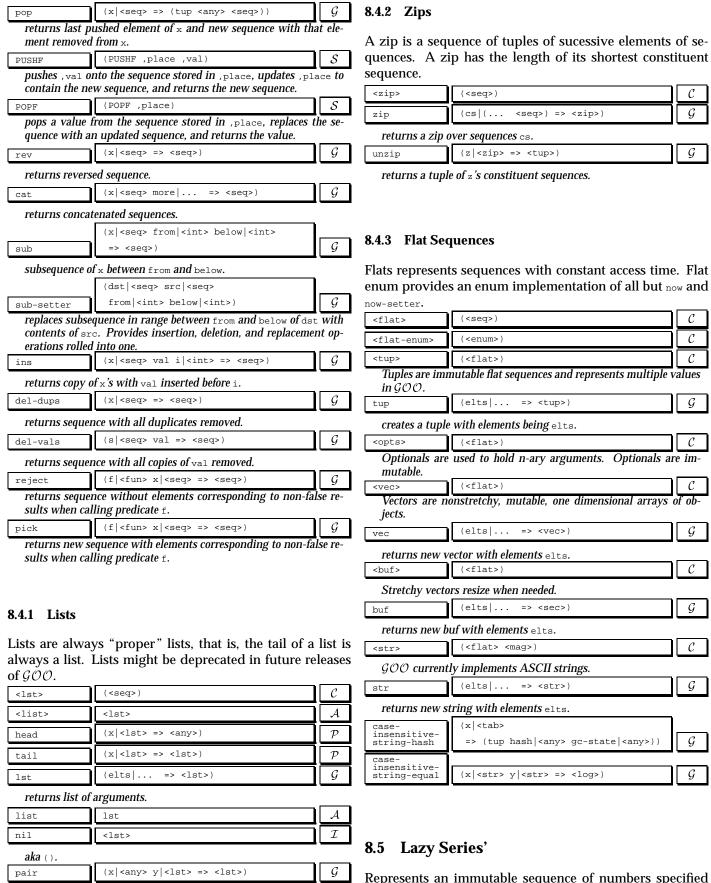
(x | < seq > y | < any > = > < seq >)

 \mathcal{G}

returns sequence with y added to the end of x.

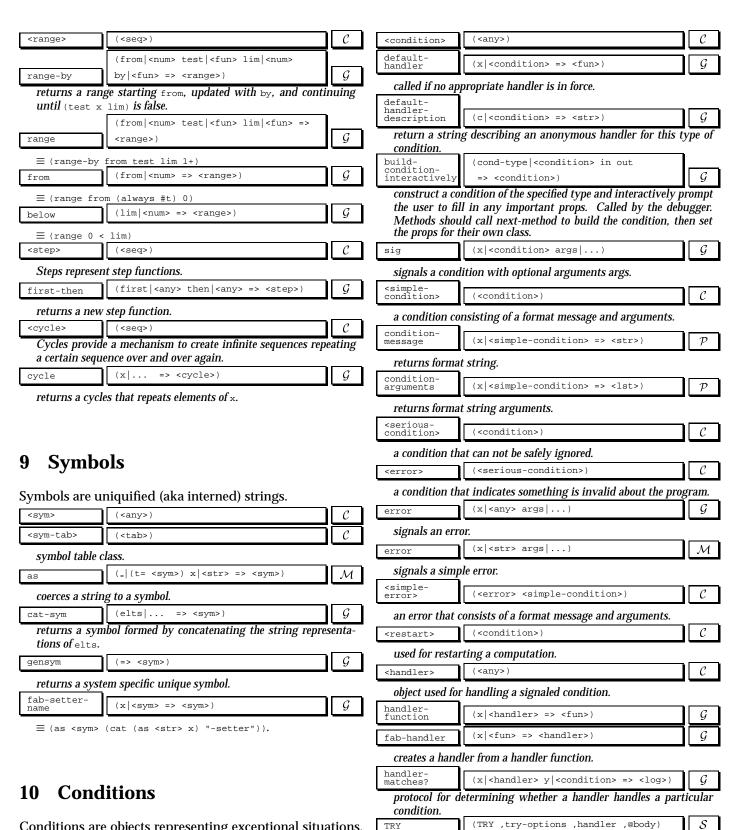
(x | < seq> y | < any> => < seq>)push returns sequence with y added to x.

 \mathcal{S}



returns new list with x as head and y as tail.

Represents an immutable sequence of numbers specified using a start number ${\tt from}$, a step amount ${\tt by}$, and an inclusive bound ${\tt to}$.



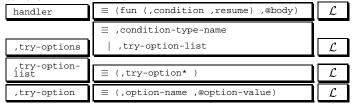
Conditions are objects representing exceptional situations. \mathcal{GOO} provides restartable conditions as well as the more traditional stack unwinding conditions. A condition is an object used to provide information to a handler. A handler is an object with a handler function used to take care of conditions of a particular type. Signalling is a mechanism for finding the most appropriate handler for a given condition. See DRM [4] for more information.

installs ,handler as a condition handler for the duration of (SEQ ,@body), using the instructions provided by ,try-options. ,try-options should either be the name of the condition type to handle, or a ,try-option-list with zero or more of the following options:

- (TYPE , expr) => An expression returning the type of condition to handle.
- (TEST ,@body) => Code which returns #t if the condition is applicable, and #f otherwise. This may be called at arbitrary times by the runtime, so it shouldn't do anything too alarming.
- (DESCRIPTION ,message ,@arguments) => A humanreadable description of this handler. Used by the debugger.

The handler function should take two arguments: the <code>,condition</code> to be handled, and a <code>,resume</code> function. if a matching condition is signaled then the handler function is called with the signaled condition and a resume function to be called if the handler wants to return a value to be used as the result of the signaling <code>sig</code> call. the handler has three possibilities: (1) it can handle the condition by taking an exit using <code>esc</code>, (2) it can resume to the original <code>sig</code> call using the resume function called with the value to be returned, or (3) it can do neither, that is, it can choose not to handle the condition by just falling through to the end of the handler (cf., Dylan's <code>block/exception</code> and <code>lethandler</code>) and the next available handler will be invoked. Note that \mathcal{GOO} does not unwind the stack before calling handlers!





11 Input / Output

This is a very preliminary I/O system and is mostly just enough with which to write a compiler.

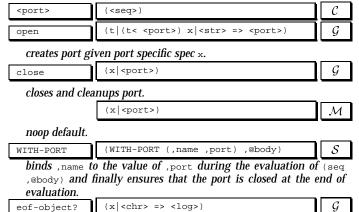
11.1 Ports

<in-port>

in

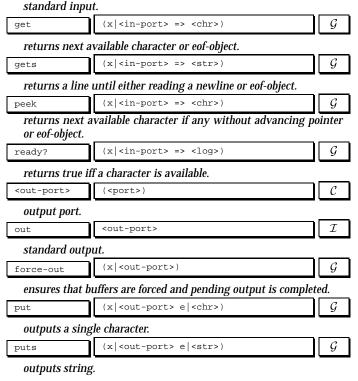
input port.

Ports represent character-oriented input/output devices.



(<port>)

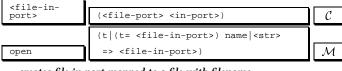
<in-port>



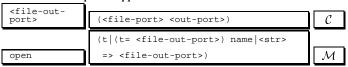
11.1.1 File Ports

File ports are ports which map to files.





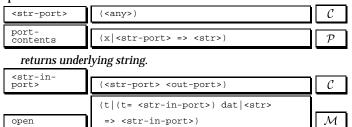
creates file in port mapped to a file with filename name.



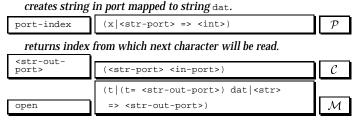
creates file out port mapped to a file with filename name.

11.1.2 Sequence Ports

Sequence ports provide port interface mapped onto sequences.



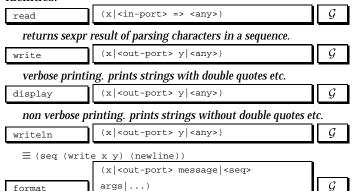
 \mathcal{C}



creates string out port mapped to string dat.

Formatted I/O 11.2

 \mathcal{GOO} provides convenient s-expression reading/writing facilities.



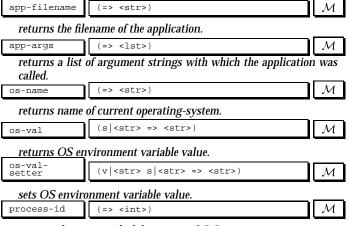
formatted output using special commands embedded in message. supported commands are:

- %= → (write x arg)
- \bullet %s \rightarrow (display x arg)
- %d \rightarrow (write x arg)
- %% → (write-char x #\%)

which consume one argument at a time. otherwise subsequent message characters are printed to port x (cf. Dylan's and CL's format).

12 **System**

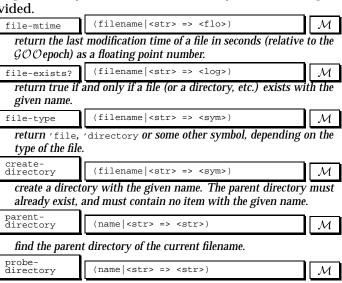
This is a very rudimentary portable interface to an underlying operating system.



returns the process id of the current \mathcal{GOO} process.

12.1 Files and Directories

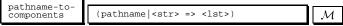
A preliminary set of file and directory facilities are provided.



make sure that the named directory exists.

12.2 Pathnames

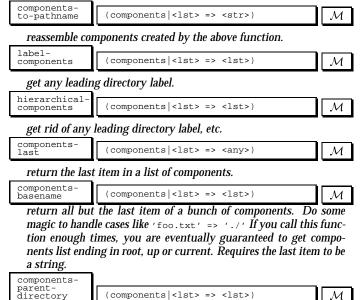
Pathnames allow you to work with hierarchical, structured pathnames in a reasonably portable fashion.



given a pathname, split it into a list of individual directories, etc. Three special values are returned as symbols:

- root → This path starts in the root directory
- up → Go up a directory
- current → Remain in the current directory

Volume labels, drive letters, and other non-path information should be stored in a single tagged list at the head. Note that the hierarchical portion of this pathname (everything but the label) must be non-empty at all times.



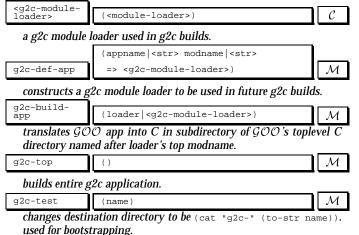
(components|<lst> => <lst>)

 \mathcal{M}

calculate the parent directory of a pathname.

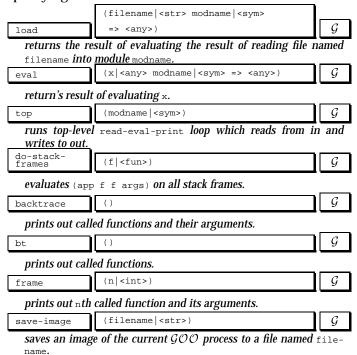
13 Compiler

 \mathcal{GOO} 's compiler, g2c, compiles \mathcal{GOO} source code to C. During a given session, g2c recompiles only used modules that are either modified or use modified modules.



14 Top Level

Functions which load code at runtime require a symbol specifying the module name to use.



15 Installation

Unpack either a linux or windows version of \mathcal{GOO} into an appropriate installation area. There are three directories:

DOC, BIN, SRC, and EMACS.

Set up your os environment variable named GOO_ROOT to your top level \mathcal{GOO} directory (i.e., containing the subdirectory named SRC). Make sure to slash terminate the path. For example, my GOO_ROOT on linux is:

```
SET GOO_ROOT=/home/ai/jrb/goo
```

On linux of course you would use forward slashes and environment variable setting depends on the shell you're using.

During start up, \mathcal{GOO} will load two patch files, one from

```
${GOO_ROOT}/SRC/system-patches.goo
```

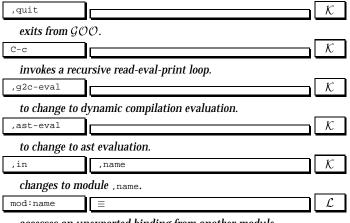
and one from

```
${GOO_ROOT}/SRC/user-patches.goo
```

You can customize your \mathcal{GOO} by adding forms to user-patches.

16 Usage

Typing goo at your shell will start up a \mathcal{GOO} read-eval-print loop, which accepts sexpressions and top-level commands commencing with a comma. The following is a list of available commands:



accesses an unexported binding from another module.

16.1 Development

To compile \mathcal{GOO} :

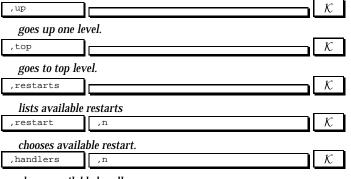
```
goo/user 0<= (use g2c)
goo/user 0=> #f
goo/user 0<= (g2c-top)</pre>
```

To run the test suites:

```
goo/user 0<= (use tests)
goo/user 0=> #f
goo/user 0<= (run-all-tests)</pre>
```

16.2 Debugger

A keyboard interrupt or any error enters the user into the debugger which provides a superset of the commands available at top-level. The following are debugger specific commands:



shows available handlers.

16.3 Emacs Support

A rudimentary emacs-based development system is provided.

16.3.1 Emacs Mode

Put EMACS/goo.el in your emacs lisp directory. Add the following to your .emacs file:

```
(autoload 'goo-mode "goo"
  "Major mode for editing Goo source." t)
(setq auto-mode-alist
        (cons '("\\.goo\\'" . goo-mode) auto-mode-alist))
```

Useful features include:

- You can add "font-lock" mode by adding the following to your .emacs:
 (global-font-lock-mode t)
- In a given buffer, you can toggle font-lock with M-x font-lock-mode
- Check out the "Index" menu item in a goo buffer.

16.3.2 Emacs Shell

Put ${\tt EMACS/goo-shell.el}$ in your emacs lisp directory. Add the following to your .emacs:

make sure to set up the goo-program-name to correspond to your installation area.

Useful command / key-bindings are:

```
M-C-x goo-send-definition
C-c C-e goo-send-definition
C-c M-e goo-send-definition-and-go
C-c C-r goo-send-region
C-c M-r goo-send-region-and-go
C-c C-z switch-to-goo
```

Check out goo-shell.el for the complete list of command / key-bindings. I doubt the compile commands do anything useful cause there isn't a compiler.

17 Caveats

 \mathcal{GOO} is relatively slow at this point. There are no compiler optimizations in place. This will improve in coming releases

This manual is very preliminary. Please consult the runtime libraries in the SRC directory. Also check out Scheme and Dylan's manuals for information of their lexical structure and special form behavior respectively.

The names of functions will probably change in the near future. Please give me feedback on the current names. Please, please, please send bug reports to <code>jrb@ai.mit.edu</code>. I will fix your bugs asap.

18 History and Acknowledgements

 \mathcal{GOO} has greatly benefitted from the help of others. During the winter of 2001, I briefly discussed the early design of Proto, a Prototype-based precursor to \mathcal{GOO} , with Paul Graham and his feedback was very useful. From there, I bootstrapped the first version of Proto for a seminar, called Advanced Topics in Dynamic Object-Oriented Language Design and Compilation (6.894), that I cotaught with Greg Sullivan and Kostas Arkoudas. The 6.894 students were very patient and gave me many helpful suggestions that greatly improved Proto. During and after the seminar, Greg Sullivan reviewed many ideas and helped tremendously. James Knight was one of the 6.894 students and became my MEng student after the course. He has helped in many many ways including the writing of the save-image facility and the speeding up of the runtime. Eric Kidd worked with me during the summer of 2001 implementing the module system, restarts, and the dependency tracking system. During that summer I decided that a Prototype-based object system was inadequate for the type system I was interested in supporting and changed over to the present type-based system. I presented my ideas on Proto at LL1 in the Fall of 2001. Many stimulating conversations on the follow on LL1 discussion list inpired me. In fact, during the course of defending Proto's form of objectorientation on that list I came up with its current name, \mathcal{GOO} , and it stuck. Andrew Sutherland became my MEng student in the winter of 2002, wrote a \mathcal{GOO} SWIG [1] backend, and has provided useful feedback on \mathcal{GOO} 's design.

References

[1] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, 1996.

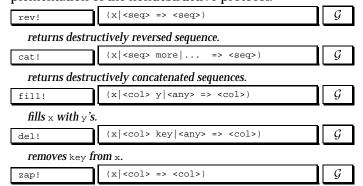
- [2] Craig Chambers. The Cecil language specification and rationale: Version 2.0. Available from http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html, December 1995.
- [3] R. Kelsey, W. Clinger, and J. Rees. Revised⁵ report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.
- [4] A. Shalit. *The Dylan Reference Manual*. Addison Wesley, 1996.
- [5] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.

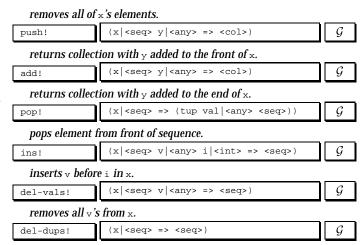
A Math

\$e	<flo></flo>	\mathcal{I}
\$pi	<flo></flo>	\mathcal{I}
sqrt	(x <num> => <num>)</num></num>	\mathcal{G}
log	(x <num> => <num>)</num></num>	\mathcal{G}
logn	(x <num> b <num> => <num>)</num></num></num>	\mathcal{G}
sin	(x <num> => <num>)</num></num>	\mathcal{G}
cos	(x <num> => <num>)</num></num>	\mathcal{G}
tan	(x <num> => <num>)</num></num>	\mathcal{G}
asin	(x <num> => <num>)</num></num>	\mathcal{G}
acos	(x <num> => <num>)</num></num>	\mathcal{G}
atan	(x <num> => <num>)</num></num>	\mathcal{G}
atan2	(y <num> x <num> => <num>)</num></num></num>	\mathcal{G}
sinh	(x <num> => <num>)</num></num>	\mathcal{G}
cosh	(x <num> => <num>)</num></num>	\mathcal{G}
tanh	(x <num> => <num>)</num></num>	\mathcal{G}

B Deprecated Mutation Protocol

These functions are in the process of being deprecated, but are documented because of current inefficiencies in the implementation of the nondestructive protocol.





removes all duplicates from x.