\mathcal{GOO} Reference Manual v28

for release 0.116

Jonathan Bachrach MIT AI Lab

March 19, 2002

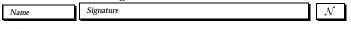
1 Introduction

 \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.

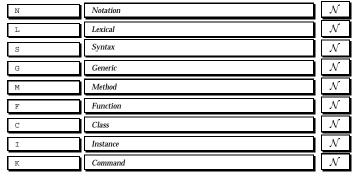
1.1 Notation

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



Documentation

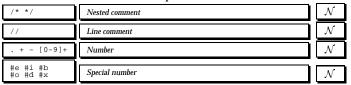
where the rightmost kind field has a one letter code as follows:

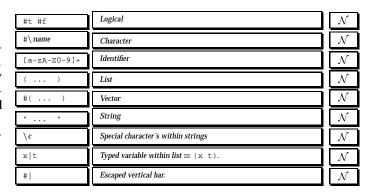


1.2 Lexical Structure

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:



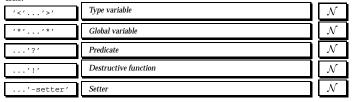


1.3 Meta Syntax

 \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 ", $_{\text{o}}$ " 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 \mathcal{COO} metasyntax is used to indicate optional patterns, \mathcal{COO} 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

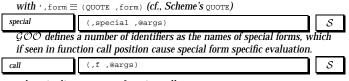
The following naming conventions are used throughout this manual:



2 Expressions

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

var	, name	$\mathcal S$
returns the va	lue of binding named , name in the current environment	
lit	,lit	$\mathcal S$
syntactic literals that are self-evaluating.		
QUOTE	(QUOTE ,form)	$\mathcal S$

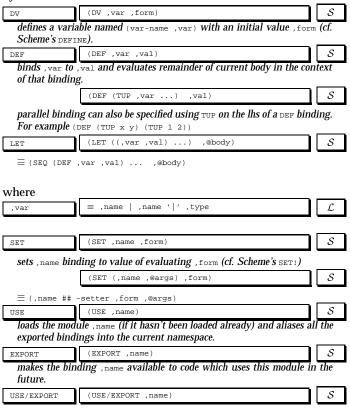


otherwise lists represent function calls.

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.

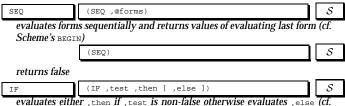
At the topmost level, \mathcal{GOO} provides simple modules that map from names to bindings. Each file introduces a new module with the same name as the file. Modules can import bindings exported by other modules.



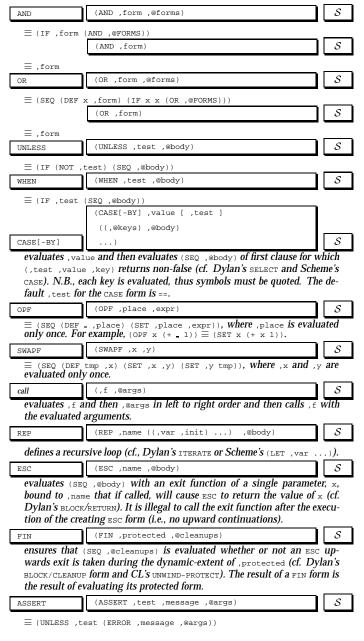
same as use plus reexports all imported bindings.

4 Program Control

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



evaluates either ,then if ,test is non-false otherwise evaluates ,else (cf. Scheme's IF). The ,else expression defaults to false.



5 Types, Classes and Properties

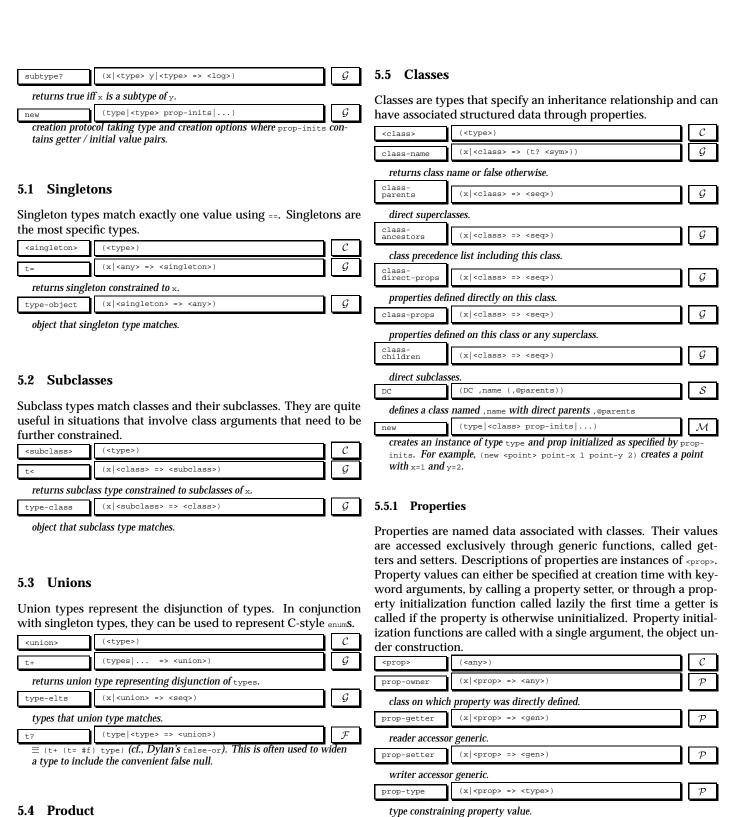
 \mathcal{GOO} types categorize objects. Types are first class. They are used to annotate bindings. Binding types restrict the type of objects bindable to associated bindings.

 \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.

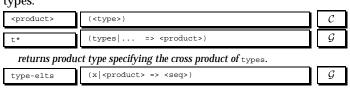
The basic type protocol is:

<type></type>	(<any>)</any>	\mathcal{C}
isa?	(x <any> y <type> => <log>)</log></type></any>	\mathcal{G}



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.

where init is a one parameter function that returns the initial value for the prop and gets called lazily with the new instance as the argument.

(c|<class> getter|<gen> => <met>)

(c|<class> setter|<gen> => <met>)

(owner getter|<gen> setter|<gen>

type | <type > init | <fun >)

(x|<prop> => <fun>)

finds getter method defined on given class.

finds setter method defined on given class.

prop-init

find-getter

find-setter

add-prop

lazy initialization function.

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

6 Functions

All operations in \mathcal{GOO} are functions.

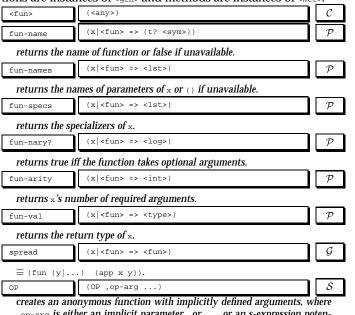
Functions accept zero or more arguments, and return one value. The parameter list of the function describes the number and types of the arguments that the function accepts, and the type of the value it returns.

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.

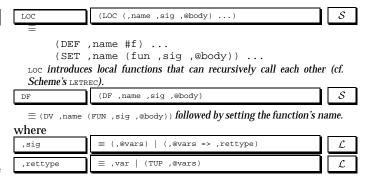
A generic function contains a number of methods. When a generic function is called, it compares the arguments it received with the parameter lists of the methods it contains. It selects the most appropriate method and invokes it on the arguments. This technique of method dispatch is the basic mechanism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



creates an anonymous function with implicitly defined arguments, where <code>.op-arg</code> is either an implicit parameter <code>.or</code> or an s-expression potentially containing further op-args. The required parameters <code>.'s</code> are found ordered according to a depth-first walk of the op-args. The following are typical examples:

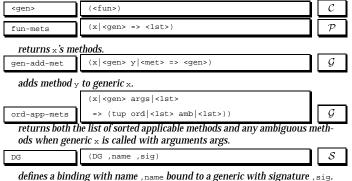
creates an anonymous method with signature, signand when called evaluates, whody as (SEQ , whody) (cf. Scheme's LAMBDA).



6.1 Generics

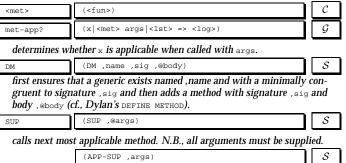
Generic functions provide a form of polymorphism allowing many implementation methods with varying parameter types, called *specializers*. Methods on a given generic function are chosen according to applicability and are then ordered by specificity. A method is applicable if each argument is an instance 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-applicable-method 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.

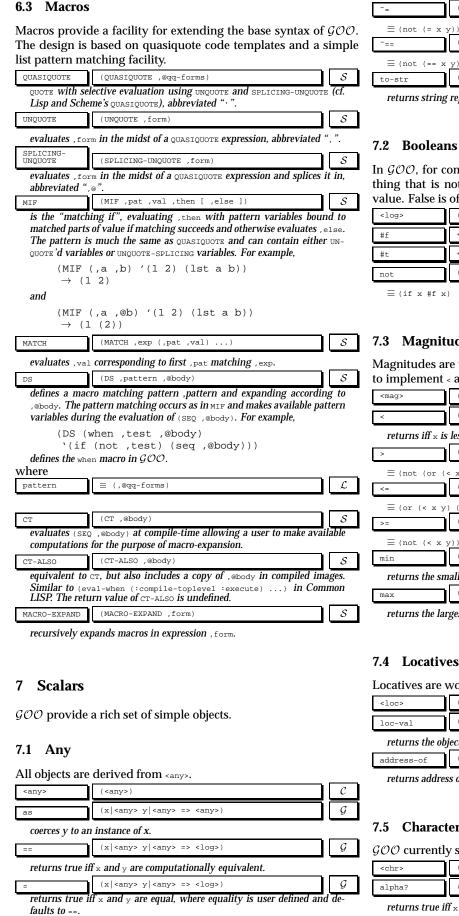


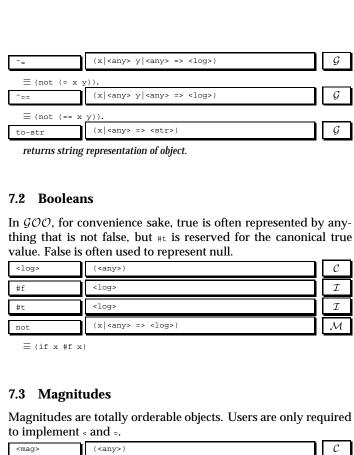
6.2 Methods

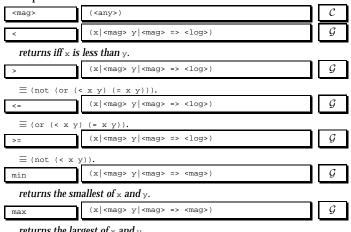
Methods are \mathcal{GOO} 's code objects. Methods can optionally be added to generics.



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

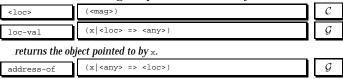






returns the largest of x and y.

Locatives are word aligned pointers to memory.



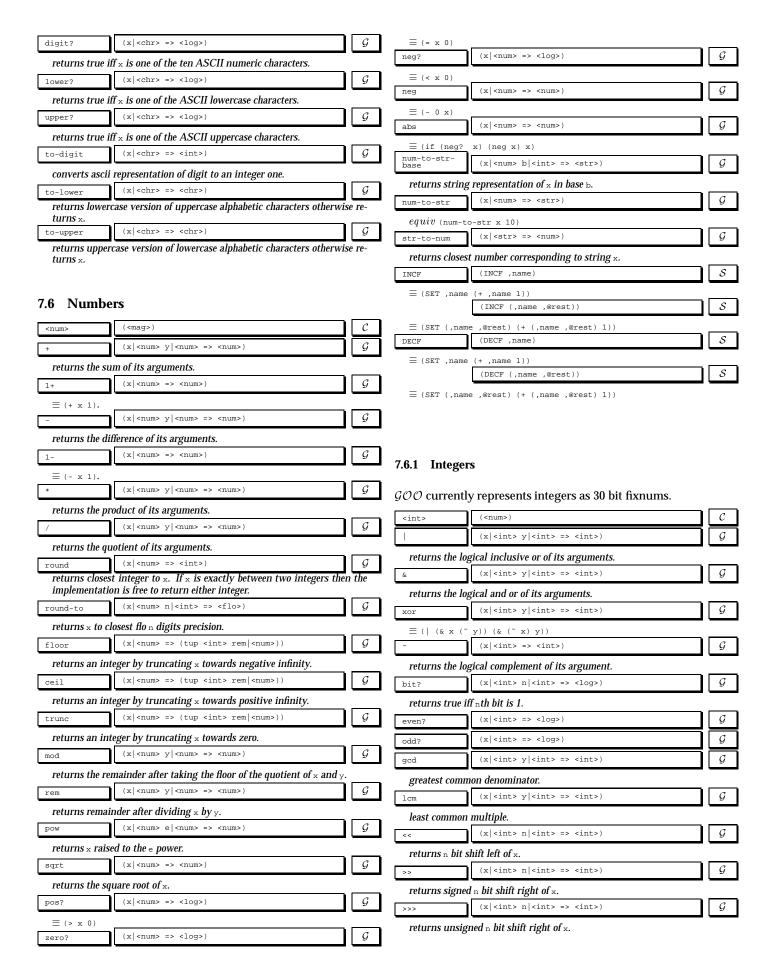
returns address of particular object.

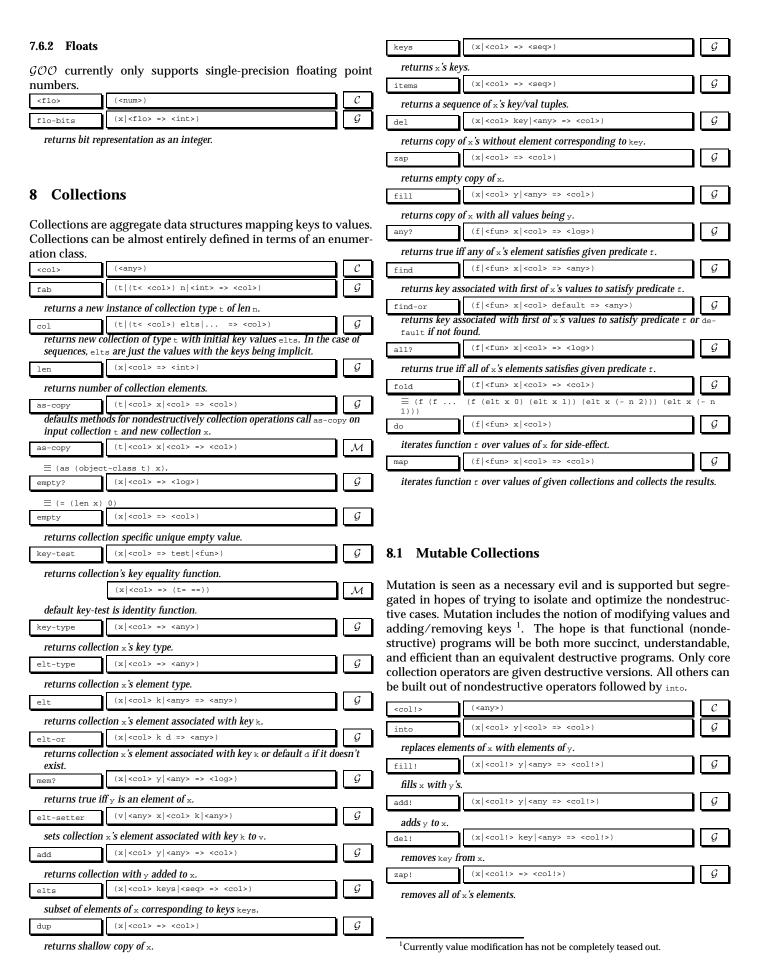
7.5 Characters

GOO currently supports 8 bit ASCII characters.

<chr></chr>	(<mag>)</mag>	\mathcal{C}
alpha?	(x <chr> => <log>)</log></chr>	$\mathcal G$

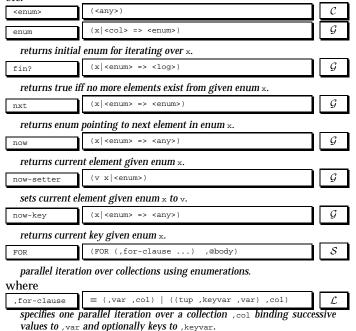
returns true iff x is one of the ASCII upper or lowercase characters.





8.2 Enumerators

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



8.3 Packers

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

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

They can also be used for summing values etc:

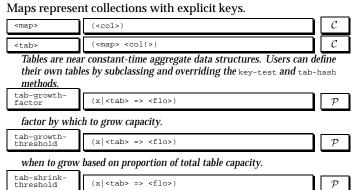
 \equiv (packer 0 + identity)

```
(packing-in (x|<int>)
  (for ((e '(1 2 3 4 5)))
    (when (odd? e) (pack-in x e)))
  (packed x))
 ==> 9
```

<packer></packer>	<any></any>	\mathcal{A}
packer-add	(p <packer> x => <packer>)</packer></packer>	$\mathcal G$
returns a pack	ter a augmented with element x.	
packer-res	(p <packer> => <any>)</any></packer>	\mathcal{G}
returns result	of packings over p.	
packer	(init add <fun> res <fun>)</fun></fun>	\mathcal{G}
	ple packer with that starts its value out with init, is add, and whose final value is computed with res.	aug-
packer-fab	(t <type> => <packer>)</packer></type>	\mathcal{G}
returns a new type t specific packer.		
packer-fab	(t (t< <seq>) => <packer>)</packer></seq>	\mathcal{M}
≡ (packer '() pair (fun (x) (as t (rev! x))))	
packer-fab	(t (t= <int>) => <packer>)</packer></int>	\mathcal{M}

PACKING-WITH	(PACKING-WITH ((,var ,pack)) ,@body)	\mathcal{S}
mechanism for	packing objects using given packer into ,var.	•
PACKING-IN	(PACKING-IN (,name ' ' ,type) ,@body)	S
■ (PACKING-W	ITH (,name (packer-fab ,type)) ,@body).	
	(PACKING-IN (,name) ,@body)	$\mathcal S$
≡ (PACKING-I	N (,name ' ' <lst>) ,@body).</lst>	
PACKING	(PACKING ,@body)	$\mathcal S$
■ (PACKING-I	N (packer-) ,@body (packed packer-)).	
≡ (PACKING-II	N (packer-) ,@body (packed packer-)). (PACK-IN ,pack ,x)	S
PACK-IN		S
PACK-IN	(PACK-IN ,pack ,x)	8
pack-in folds ,x into p	(PACK-IN ,pack ,x) acker in ,pack. (PACK ,x)	8
FACK-IN folds , x into p PACK	(PACK-IN ,pack ,x) acker in ,pack. (PACK ,x)	S S

8.4 Maps



when to shrink based on proportion of total table capacity. (x|<tab> => hash|<fun>) tab-hash

\$permanent hash-state GC specific. tab-gc-state (x|<tab> => <any>) GC specific.

returns key equality and hash function.

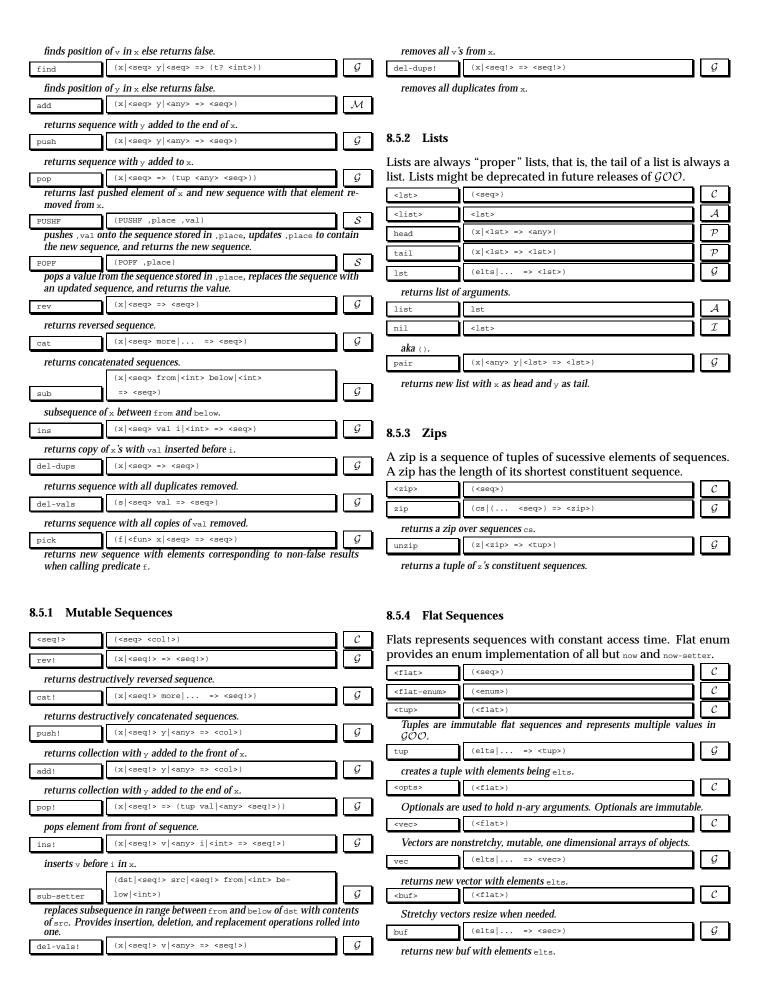
<any>

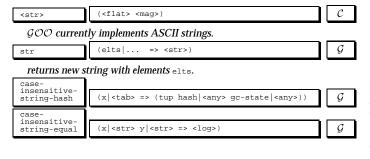
(x | <tab> => (tup hash gc-state)) hash function based on pointer. Susceptible to rehash if objects are moved. The gc-state reflects movement.

8.5 Sequences

Sequences are collections with nonnegative integer keys.

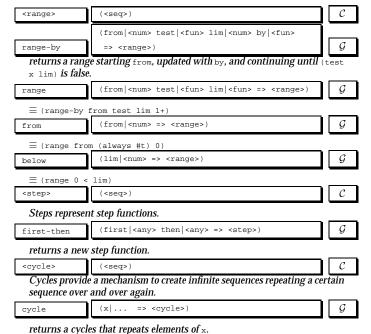






8.6 Lazy Series'

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



9 Symbols

Symbols are uniquified (aka interned) strings.

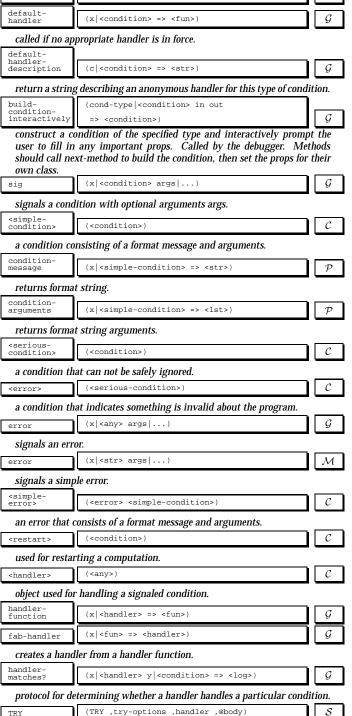
<sym></sym>	(<any>)</any>	\mathcal{C}
<sym-tab></sym-tab>	(<tab>)</tab>	\mathcal{C}
symbol table class.		
as	(_ (t= <sym>) x <str> => <sym>)</sym></str></sym>	\mathcal{M}
coerces a strin	ng to a symbol.	
cat-sym	(elts => <sym>)</sym>	\mathcal{G}
returns a syn	abol formed by concatenating the string representation	ns of
gensym	(=> <sym>)</sym>	$\mathcal G$
returns a system specific unique symbol.		
fab-setter- name	(x <sym> => <sym>)</sym></sym>	\mathcal{G}
= (as <sym></sym>	(cat (as <str> x) "-setter")).</str>	

10 Conditions

<condition>

(<any>)

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 human-readable 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>let handler</code>) and the next available handler will be invoked. Note that \mathcal{GOO} does not unwind the stack before calling handlers!

where

handler	≡ (fun (,condition ,resume) ,@body)	\mathcal{L}
try-options,	= ,condition-type-name ,try-option-list	\mathcal{L}
,try-option- list	≡ (,try-option*)	\mathcal{L}
try-option,	≡ (,option-name ,@option-value)	\mathcal{L}

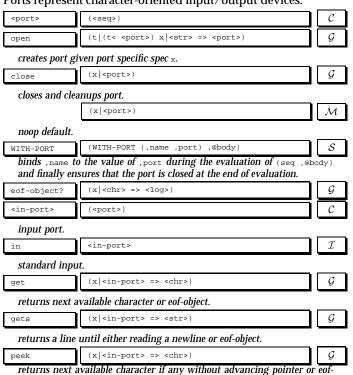
11 Input / Output

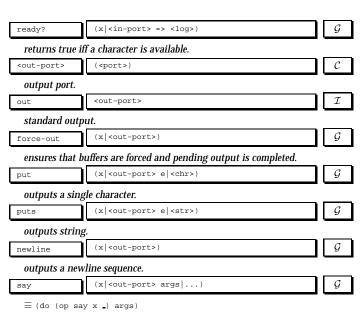
This is a very preliminary I/O system and is mostly just enough with which to write a compiler.

11.1 Ports

object.

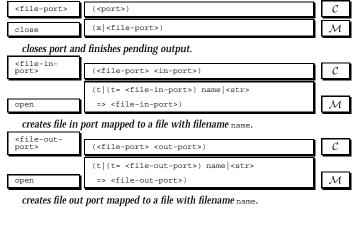
Ports represent character-oriented input/output devices.





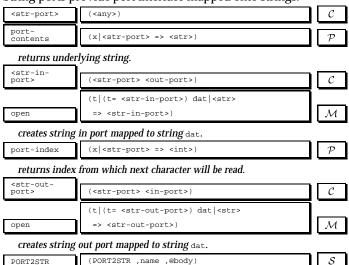
11.1.1 File Ports

File ports are ports which map to files.



11.1.2 String Ports

String ports provide port interface mapped onto strings.



 \mathcal{G} returns sexpr result of parsing characters in a sequence. \mathcal{G} write (x|<out-port> y|<any>) verbose printing. prints strings with double quotes etc. (x|<out-port> y|<any>) \mathcal{G} display non verbose printing. prints strings without double quotes etc. \mathcal{G} (x|<out-port> y|<any>) writeln \equiv (seq (write x y) (newline)) (x | < out-port> message | < seq> args | ...) \mathcal{G} formatted output using special commands embedded in message. ported commands are:

• %= → (write x arg)
• %s → (display x arg)
• %d → (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.

app-filename (=> <str>) \mathcal{M} returns the filename of the application. (=> <lst>) \mathcal{M} app-args returns a list of argument strings with which the application was called. (=> <str>) \mathcal{M} os-name returns name of current operating-system. (s|<str> => <str>) \mathcal{M} os-val returns OS environment variable value. (v|<str> s|<str> => <str>) \mathcal{M} sets OS environment variable value. \mathcal{M} process-id (=> <int>)

returns the process id of the current GOO process.

12.1 Files and Directories

A preliminary set of file and directory facilities are provided.

(filename | <str> => <flo>) \mathcal{M} return the last modification time of a file in seconds (relative to the GOO epoch) as a floating point number. (filename|<str> => <log>) file-exists? \mathcal{M} return true if and only if a file (or a directory, etc.) exists with the given name. (filename | <str> => <sym>) file-type \mathcal{M} return 'file, 'directory or some other symbol, depending on the type of the file. create-(filename | <str> => <sym>) directory \mathcal{M} create a directory with the given name. The parent directory must already exist, and must contain no item with the given name. parent-directory (name | <str> => <str>) \mathcal{M}

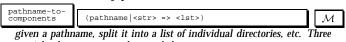
find the parent directory of the current filename.

probedirectory (name|<str> => <str>)

make sure that the named directory exists.

12.2 Pathnames

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

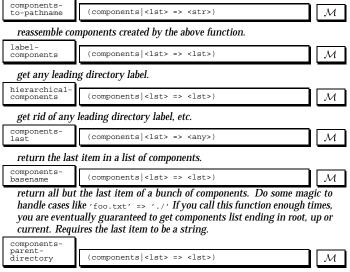


special values are returned as symbols:

• root → This path starts in the root directory

- up → Go up a directory
- current → Remain in the current directory

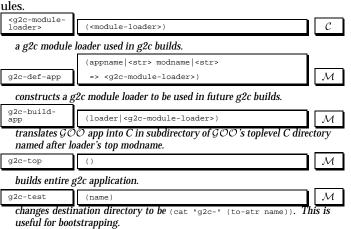
Volume labels, drive letters, and other non-path information should be stored in a single tagged list at the head. Note that the hierarchical portion of this pathname (everything but the label) must be non-empty at all times.



calculate the parent directory of a pathname.

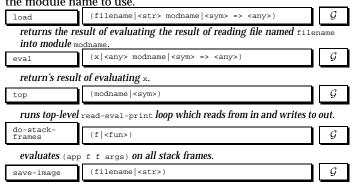
13 Compiler

 \mathcal{GOO} 's compiler, $\mathcal{g2c}$, compiles \mathcal{GOO} source code to C. It lives within the $_{x8x}$ module. 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.



saves an image of the current \mathcal{GOO} process to a file named filename.

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 <code>goo_root</code> to your top level \mathcal{GOO} directory (i.e., containing the subdirectory named <code>src</code>). Make sure to slash terminate the path. For example, my <code>goo_root</code> 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

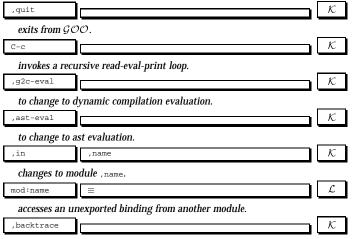
and one from

\${GOO_ROOT}/src/user-patches.goo

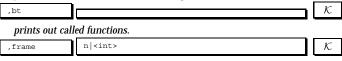
You can customize your \mathcal{GOO} by adding forms to user-patches.

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:



prints out called functions and their arguments.



prints out nth called function and its arguments.

16.1 Development

To compile \mathcal{GOO} :

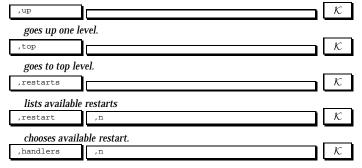
goo/user 0<= (use x8r/g2c)
goo/user 0=> #f
goo/user 0<= (g2c-top)</pre>

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 ${\tt emacs/goo.el}$ in your emacs lisp directory. Add the following to your .emacs file:

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
- Check out the "Index" menu item in a goo buffer.

Load ${\tt emacs/goo-font-lock.el}$ for a color coded parenthesis nesting aid $^2.$

 $^{^2{\}rm The}$ original idea was dreamed up and first implemented by Andrew Sutherland and then improved by James Knight

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 / keybindings. I doubt the compile commands do anything useful cause there isn't a compiler.

16.3.3 TAGS

Emacs TAGS files can be generated by typing make all-tags in the $_{\mathtt{src}}$ directory. Useful tags commands / key-bindings are:

```
M-. find-tag
M-, tags-loop-continue
tags-search
tags-query-replace
```

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.

Please, please send bug reports to <code>jrb@ai.mit.edu</code>. I will fix your bugs asap. The \mathcal{GOO} website <code>www.jbot.org/goo</code> will have papers, releases, FAQS, etc.

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 Prototypebased object system was inadequate for the type system I was interested in supporting and changed over to the present typebased 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 object-orientation 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}