GOO Reference Manual v30

Jonathan Bachrach MIT AI Lab

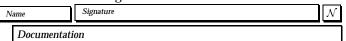
March 22, 2002

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]. \mathcal{GOO} 's main goal is to offer the best of both scripting and deliver languages. \mathcal{GOO} is freely available from www.googaga.org under LGPL. 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:



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}
C	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. Furthermore, 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}
\c	Special character's within strings	\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 ", $_{\circ}$ " 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 <code>,name</code> shape is builtin and matches only identifiers. The <code>/[/.../]</code> metasyntax is used to indicate optional patterns, <code>/.../</code> is used to indicate zero or more of the preceding pattern element, and <code>##</code> is used to denote infix string concatenation. Finally, in this manual, uppercase indicates a special form or macro.

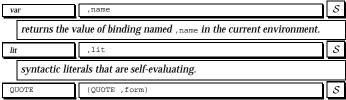
1.4 Conventions

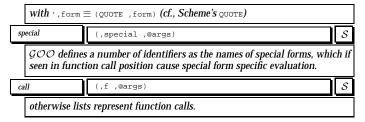
The following naming conventions are used throughout this man-

'<''>'	Type variable	\mathcal{N}
********	Global variable	\mathcal{N}
'?'	Predicate	\mathcal{N}
'1'	Destructive function	\mathcal{N}
'-setter'	Setter	\mathcal{N}

2 Expressions

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



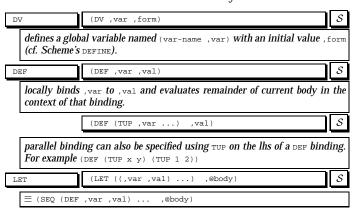


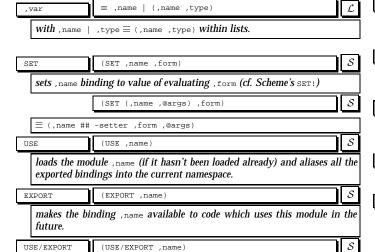
3 Namespaces and Bindings

where

 \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 visible 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. Nested modules are supported by way of slashes in module names. Modules can import bindings exported by other modules, but currently there is no way to selectively exclude or rename imported bindings. Furthermore, no cycles can occur in the module use heterarchy.

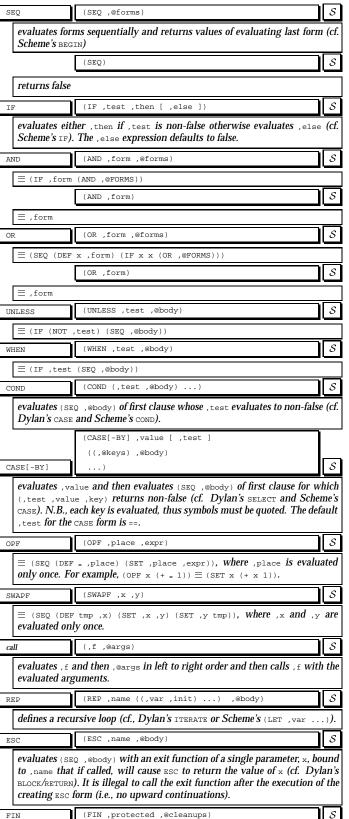




same as use plus reexports all imported bindings.

4 Program Control

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



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.



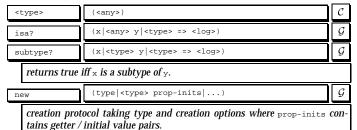
5 Types, Classes and Properties

 \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 :

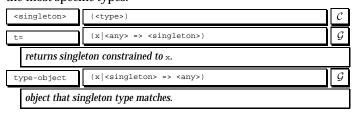
- Singleton types specify a unique instance,
- Classes and properties specify the structure, inheritance, and initialization of objects. Every object is a direct instance of a particular class.
- · Product types specify a cross product of types,
- Subclass types specify a lineage of classes, and
- ·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

The basic type protocol is:



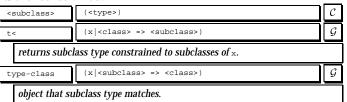
5.1 Singletons

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



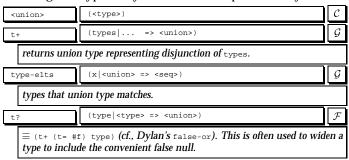
5.2 Subclasses

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



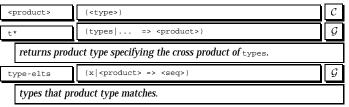
5.3 Unions

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



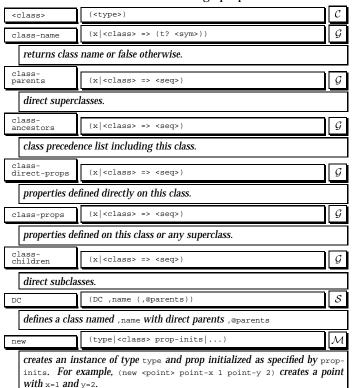
5.4 Product

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



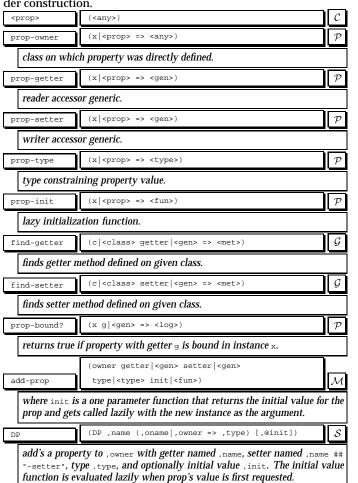
5.5 Classes

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



5.5.1 Properties

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 cproperty 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.



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.

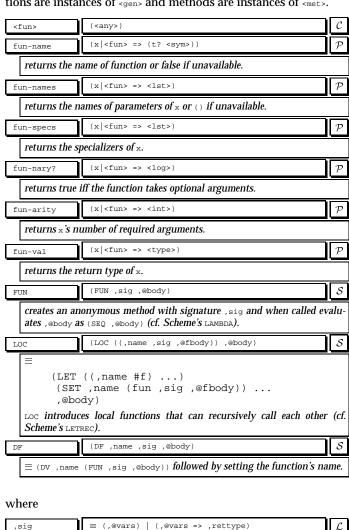
There are two kinds of functions, methods and generic functions. Both are invoked in the same way. The caller does not need to know whether the function it is calling is a method or a generic function.

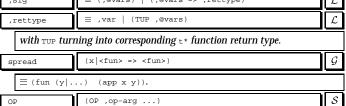
A method is the basic unit of executable code. A method accepts a number of arguments, creates local bindings for them, executes an implicit body in the scope of these bindings, and then returns a value.

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 required parameter "_" or rest parameter "..." or an s-expression potentially containing further op-args. The required parameters are found ordered according to a depth-first walk of the op-args. The following are typical examples:

```
((op _) 1) ==> 1
((op 2) 1) ==> 2
((op tail (tail _)) '(1 2 3)) ==> (3)
((op + _ 1) 3) ==> 4
((op lst ... 1) 3 2) ==> (3 2 1)
(f|<fun> args|... => <any>)
```

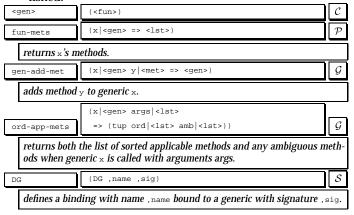
calls f $with \ arguments$ (cat (sub args 0 (- (len args) 2)) (elt args (- (len args) 1))).

 \mathcal{G}

6.1 Generics

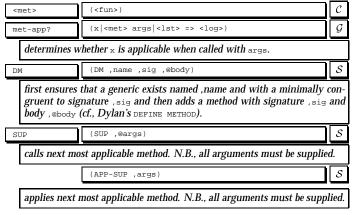
Generic functions provide a form of polymorphism allowing many implementation methods with varying parameter types, called *specializers*. Methods on a given generic function are chosen according to applicability and are then ordered by specificity. A method is applicable if each argument is an instance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



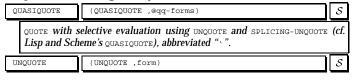
6.2 Methods

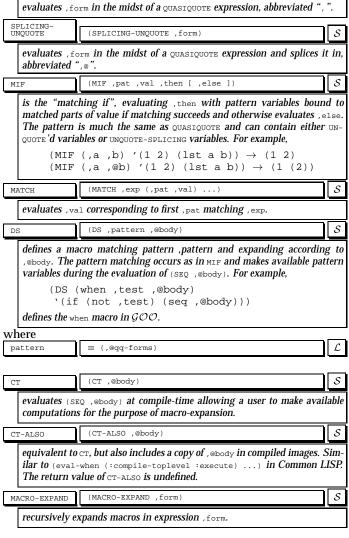
Methods are \mathcal{GOO} 's code objects. Methods can optionally be added to generics.



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.



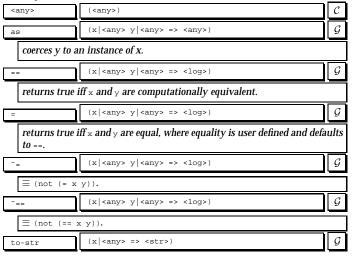


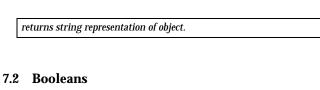
7 Scalars

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

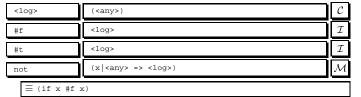
7.1 Any

All objects are derived from <any>.





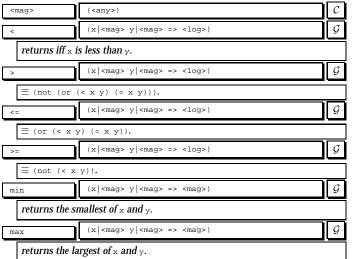
In \mathcal{GOO} , for convenience sake, true is often represented by anything that is not false, but $\#^{t}$ is reserved for the canonical true value. False is often used to represent null.



(x|<chr> => <log>) \mathcal{G} lower? returns true iff x is one of the ASCII lowercase characters. (x|<chr> => <log>) \mathcal{G} upper? returns true iff x is one of the ASCII uppercase characters. (x|<chr> => <int>) \mathcal{G} converts ascii representation of digit to an integer one. (x|<chr> => <chr>) \mathcal{G} to-lower returns lowercase version of uppercase alphabetic characters otherwise returns x. \mathcal{G} to-upper (x|<chr> => <chr>) returns uppercase version of lowercase alphabetic characters otherwise re-

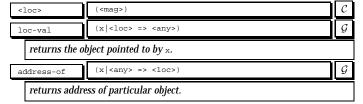
7.3 Magnitudes

Magnitudes are totally orderable objects. Users are only required to implement ${\mbox{\tiny <}}$ and ${\mbox{\tiny =}}.$



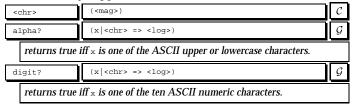
7.4 Locatives

Locatives are word aligned pointers to memory.

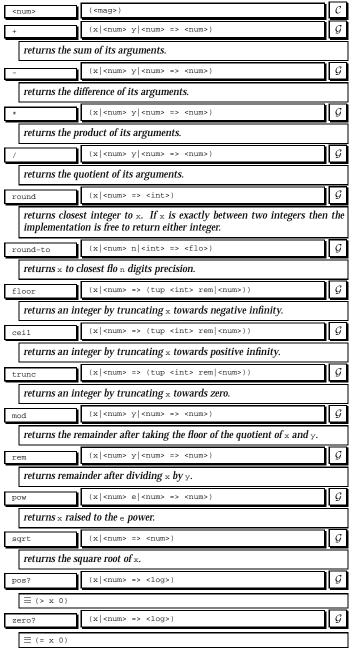


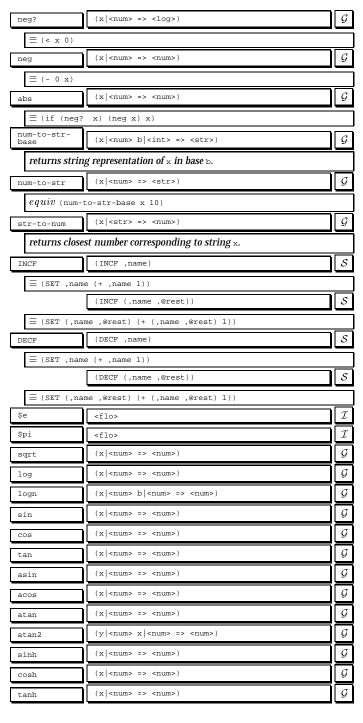
7.5 Characters

 \mathcal{GOO} currently supports 8 bit ASCII characters.



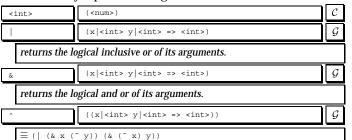
7.6 Numbers

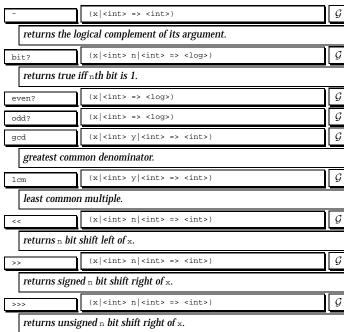




7.6.1 Integers

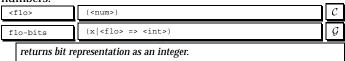
 \mathcal{GOO} currently represents integers as 30 bit fixnums.





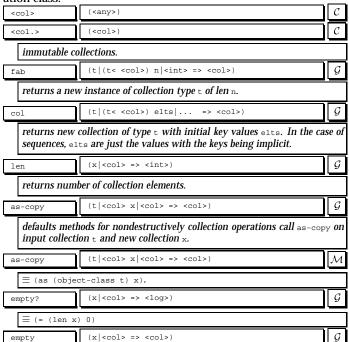
7.6.2 Floats

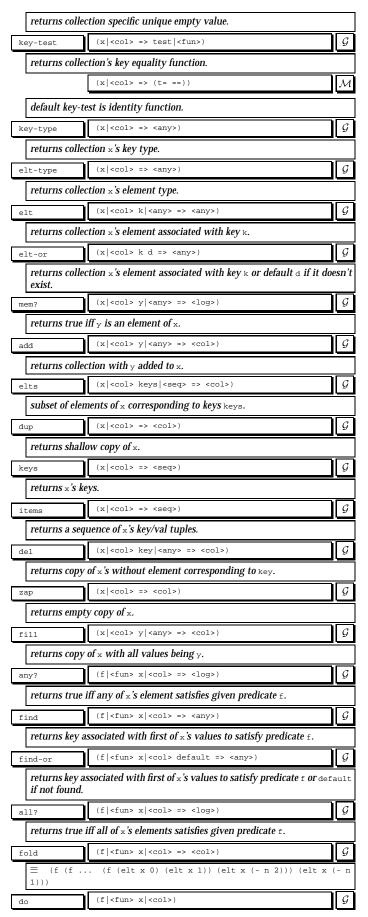
 \mathcal{GOO} currently only supports single-precision floating point numbers.

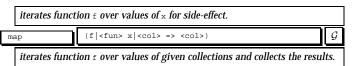


8 Collections

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

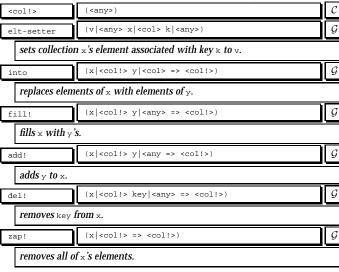






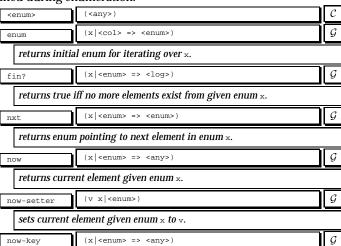
8.1 Mutable Collections

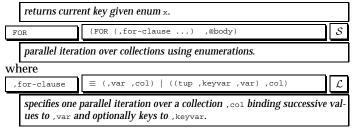
Mutation is seen as a necessary evil and is supported but segregated in hopes of trying to isolate and optimize the nondestructive cases. Mutation includes the notion of modifying values and adding/removing keys. The hope is that functional (nondestructive) programs will be both more succinct, understandable, and efficient than equivalent destructive programs. Only core collection operators are given destructive versions. All others can be built out of nondestructive operators followed by <code>into</code>.



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 enumerator 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. Enumeration behavior is undefined if an enumerator is modified during enumeration.



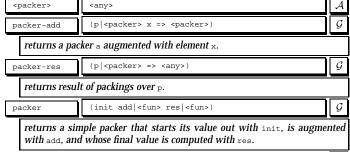


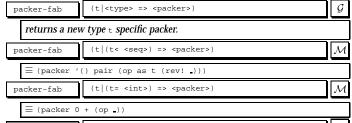
8.3 Packers

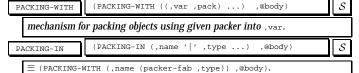
Packers are the complement of enumerators and are the imperative version of <code>fold</code>. The default packer returns a list of all accumulated values:

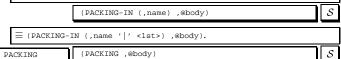
They can also be used for summing values etc:

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







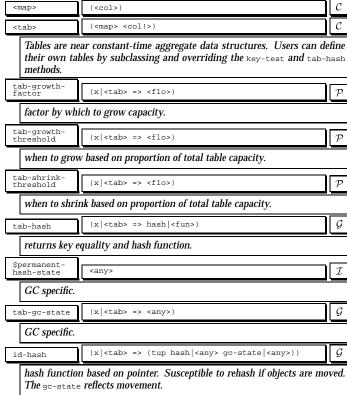




10145 /11 2	mico puener m / puem	
PACK	(PACK ,x)	S
■ (PACK	packer- ,name).	
PACKED	(PACKED ,name)	8
≡ (packe	er-res ,name).	

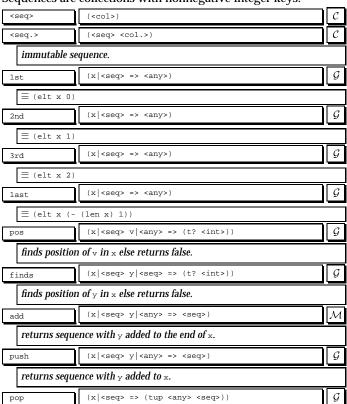
8.4 Maps

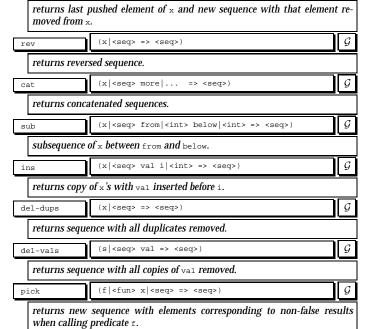
Maps represent collections with explicit keys.



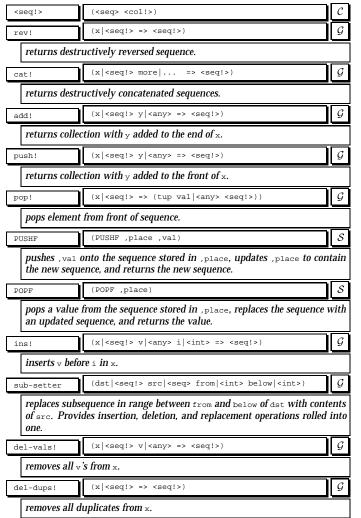
8.5 Sequences

Sequences are collections with nonnegative integer keys.



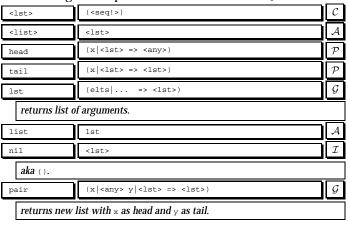


8.5.1 Mutable Sequences



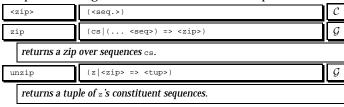
8.5.2 Lists

Lists are always "proper" lists, that is, the tail of a list is always a list. Lists might be deprecated in future releases of \mathcal{GOO} .



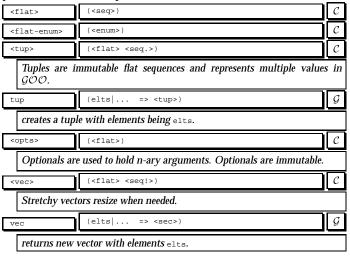
8.5.3 Zips

A zip is a sequence of tuples of sucessive elements of sequences. A zip has the length of its shortest constituent sequence.



8.5.4 Flat Sequences

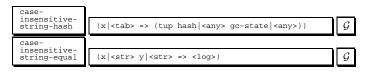
Flats represents sequences with constant access time. Flat enum provides an enum implementation of all but now and now-setter.



Strings

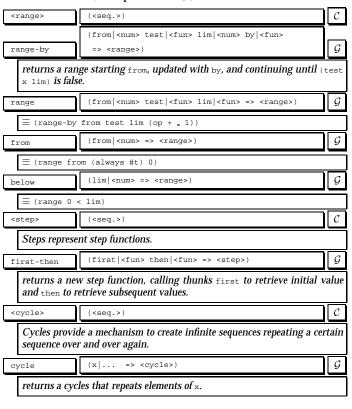
GOO currently implements ASCII strings.

<str></str>	(<flat> <mag> <seq.>)</seq.></mag></flat>	\mathcal{C}
str	(elts => <str>)</str>	\mathcal{G}
returns new	string with elements elts.	-



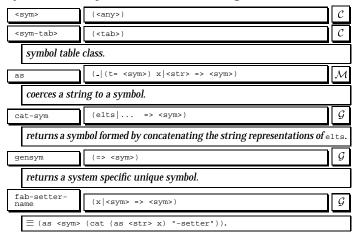
8.6 Lazy Series'

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



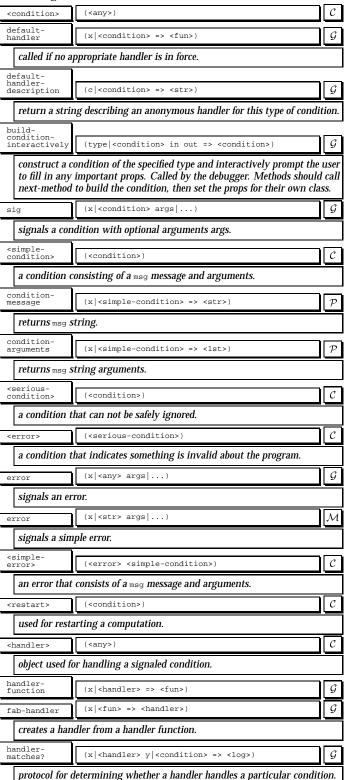
9 Symbols

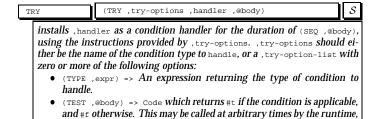
Symbols are uniquified (aka interned) strings.



10 Conditions

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.

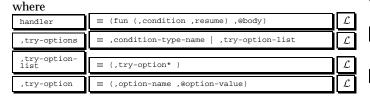




so it shouldn't do anything too alarming.

• (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>lethendler</code> and the next available handler will be invoked. Note that <code>GOO</code> does not unwind the stack before calling handlers!



11 Input / Output

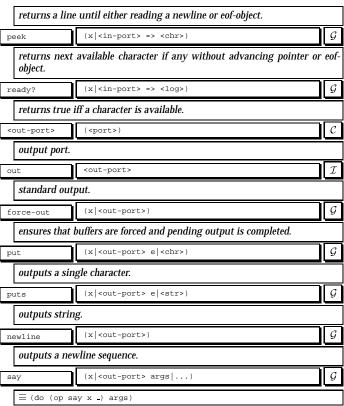
This is a very preliminary I/O system and is mostly just enough with which to write a compiler.

11.1 Ports

gets

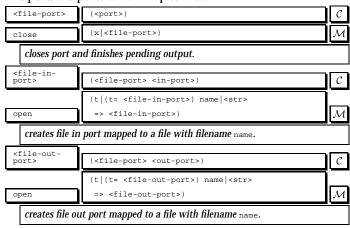
Ports represent character-oriented input/output devices. <port> (<seq>) \mathcal{C} \mathcal{G} (t | (t < port>) x | < str> => < port>)open creates port given port specific spec x. (x | <port>) \mathcal{G} closes and cleanups port. (x|<port>) \mathcal{M} noop default. \mathcal{S} WITH-PORT (WITH-PORT (,name ,port) ,@body) binds , name to the value of , port during the evaluation of (seq ,@body) and finally ensures that the port is closed at the end of evaluation. eof-object? (x|<chr> => <log>) \mathcal{G} \mathcal{C} <in-port> (<port> input port. <in-port> \mathcal{I} standard input. (x|<in-port> => <chr>) \mathcal{G} get returns next available character or eof-object.

(x | < in-port> => < str>)



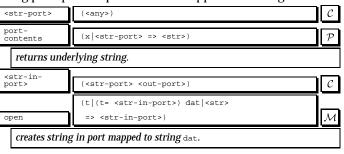
11.1.1 File Ports

File ports are ports which map to files.

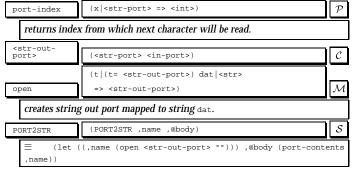


11.1.2 String Ports

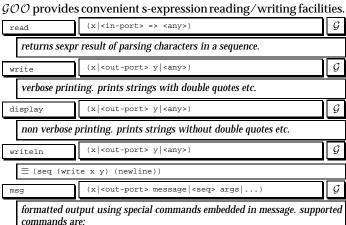
String ports provide port interface mapped onto strings.



 \mathcal{G}



11.2 Formatted I/O

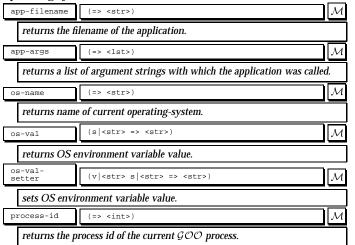


commands are:

- %= → (write x arg)
- %s → (display x arg)
- %d → (write x arg) $%% \rightarrow (write-char x \#\%)$
- which consume one argument at a time. otherwise subsequent message characters are printed to port x (cf. Dylan's and CL's format).

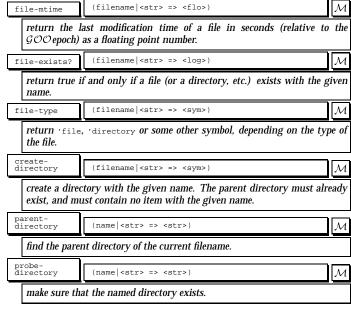
12 System

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



12.1 Files and Directories

A preliminary set of file and directory facilities are provided.



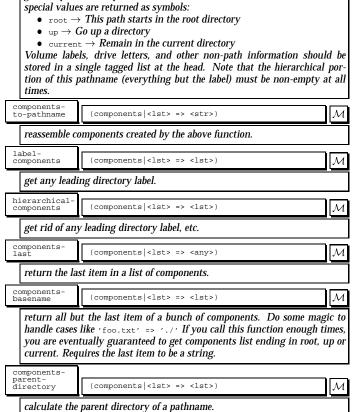
12.2 Pathnames

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

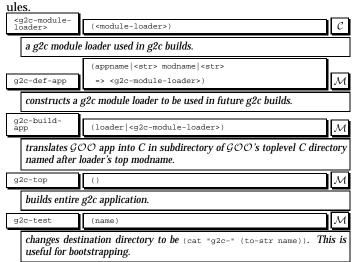
 \mathcal{M}

(pathname | <str> => <lst>)



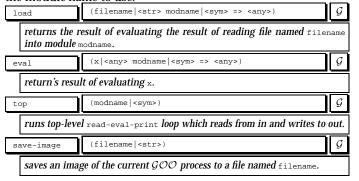
13 Compiler

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



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 STC). 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

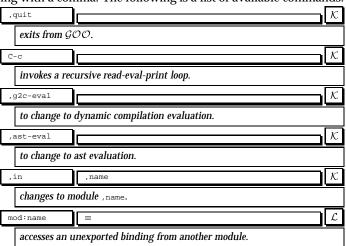
and one from

\${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:



16.1 Development

To compile \mathcal{GOO} :

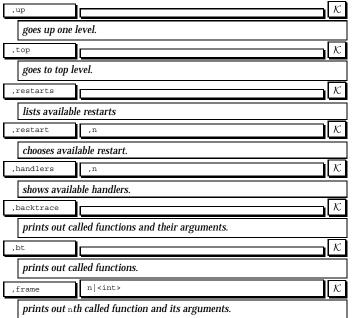
```
goo/user 0<= (use x8r/g2c)
goo/user 0=> #f
goo/user 0<= (g2c-top)
```

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:



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 $.{\tt emacs}$ file:

Useful features include the following. You can add "font-lock" mode by adding <code>(global-font-lock-mode t)</code> to your <code>.emacs</code>: In a given buffer, you can toggle font-lock with <code>M-x font-lock-mode</code>. Finally, check out the "Index" menu item in a \mathcal{GOO} buffer for other options.

For even more fun, load $_{\tt emacs/goo-font-lock.el}$ for a color coded parenthesis nesting aid $^1.$

16.3.2 Emacs Shell

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

```
(autoload 'run-goo "goo-shell" "Run an inferior Goo process." t)
(setq auto-mode-alist
   (cons '("\\.goo\\" . goo-mode) auto-mode-alist))
(setq goo-program-name "/home/ai/jrb/goo/goo")
```

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 $_{\tt make}$ all-tags in the $_{\tt src}$ 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 $_{\tt src}$ 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. Finally, I would like to thank Keith Playford for his continued guidance in language design and implementation and for his ever present sense of good taste.

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.

 $^{^{\}rm 1}{\rm The}$ original idea was dreamed up and first implemented by Andrew Sutherland and then improved by James Knight