GOO Reference Manual v41

Jonathan Bachrach MIT AI Lab

August 16, 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, more dynamic, lisp-syntaxed Dylan [?] and an object-oriented Scheme [?]. \mathcal{GOO} 's main goal is to offer the best of both scripting and delivery languages while at the same time incorporating an extreme back-to-basics philosophy. \mathcal{GOO} is freely available from www.googoogaga.org under GPL. This manual is preliminary and relies 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:

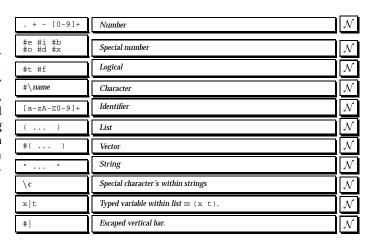
N	Notation	\mathcal{N}
L	Lexical	\mathcal{N}
S	Syntax	\mathcal{N}
G	Generic	N
М	Method	N
F	Function	\mathcal{N}
C	Class	\mathcal{N}
P	Property	N
I	Instance	\mathcal{N}
K	Command	\mathcal{N}

1.2 Lexical Structure

The lexical structure is mostly the same as Scheme [?] 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:

i	Line comment	\mathcal{N}
#/ /#	Nested comment	\mathcal{N}

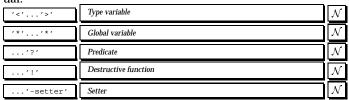


1.3 Meta Syntax

 \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 ", $_{\text{e}}$ " 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

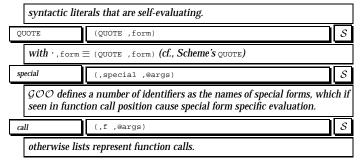
The following naming conventions are used throughout this manual:



2 Expressions

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

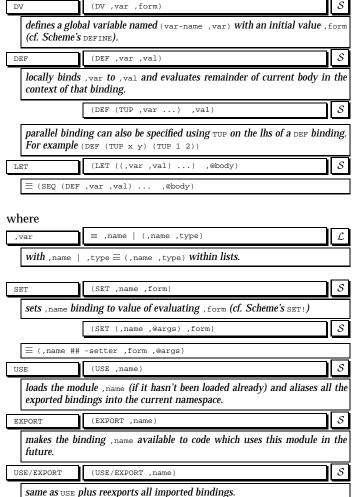
110	minci.		_
Vã	ır	,name	\mathcal{S}
	returns the value of binding named , name in the current environment.		
lit		,lit	\mathcal{S}



3 Namespaces and Bindings

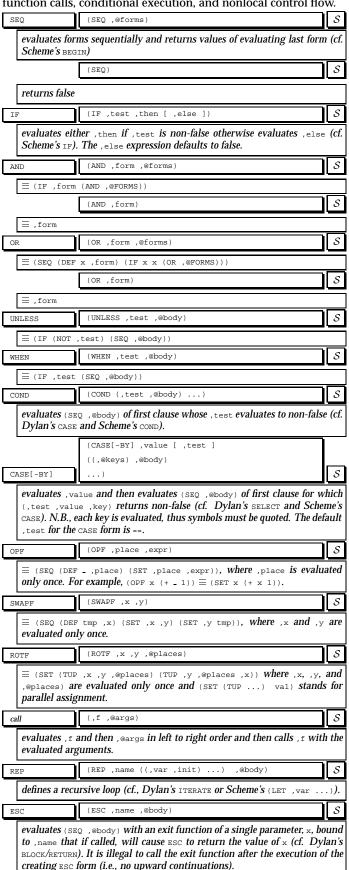
 \mathcal{GOO} is a lexically scoped language. Bindings contain values and are looked up by name. Lexical bindings are visible from only particular textual ranges in a program. Lexical bindings shadow visible bindings of the same name.

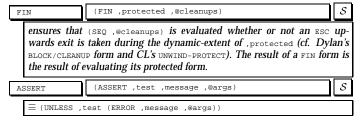
At the topmost level, \mathcal{GOO} provides simple modules that map from names to bindings. Each file introduces a new module with the same name as the file. Nested modules are supported by way of slashes in module names. Modules can import bindings exported by other modules, but currently there is no way to selectively exclude or rename imported bindings. Furthermore, no cycles can occur in the module use heterarchy.



4 Program Control

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





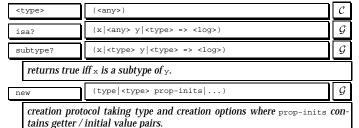
5 Types, Classes and Properties

 \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 (with the exact ordering defined in Appendix ??):

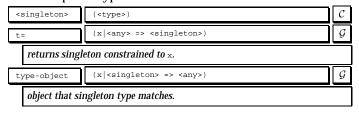
- Singleton types specify a unique instance,
- Classes and properties specify the structure, inheritance, and initialization of objects. Every object is a direct instance of a particular class,
- Product types specify a cross product of types,
- Subclass types specify a lineage of classes, and
- Union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

The basic type protocol is:



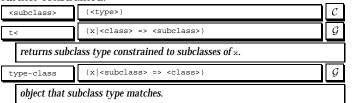
5.1 Singletons

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



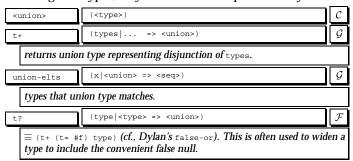
5.2 Subclasses

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



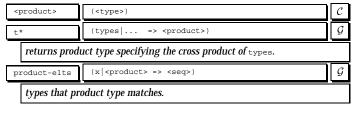
5.3 Unions

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



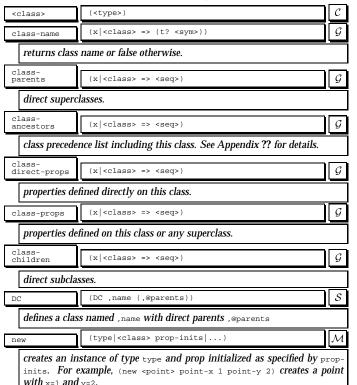
5.4 Product

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



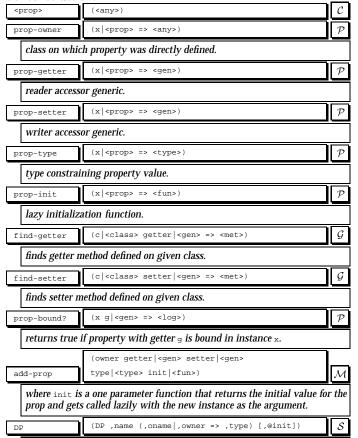
5.5 Classes

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 cprop>. 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.



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.

add's a property to ,owner with getter named ,name, setter named ,name ##

function is evaluated lazily when prop's value is first requested.

"-setter", type, type, and optionally initial value, init. The initial value

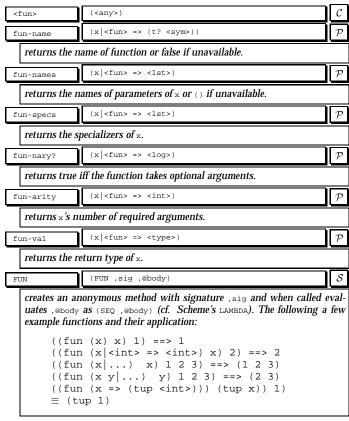
There are two kinds of functions, methods and generic functions. Both are invoked in the same way. The caller does not need to know whether the function it is calling is a method or a generic function.

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.

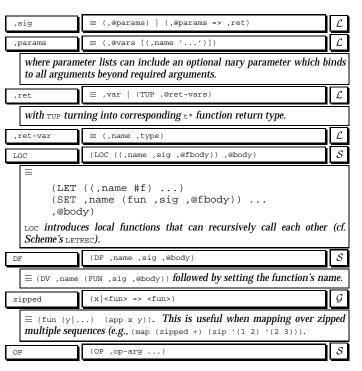
A generic function contains a number of methods. When a generic function is called, it compares the arguments it received with the parameter lists of the methods it contains. It selects the most ap-

propriate 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>.



where



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:

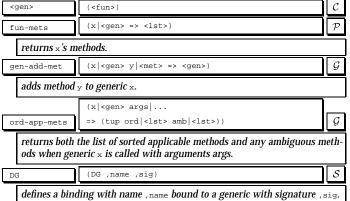
```
((op _) 1) ==> 1
((op 2) 1) ==> 2
((op + _ 1) 3) ==> 4
((op lst ... 1) 3 2) ==> (3 2 1)
((op tail (tail _)) '(1 2 3)) ==> (3)
app
(f|<fun> args|... => <any>)
```

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

6.1 Generics

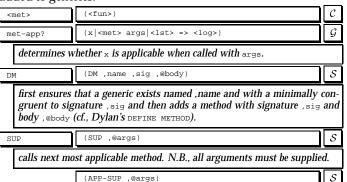
Generic functions provide a form of polymorphism allowing many implementation methods with varying parameter types, called *specializers*. Methods on a given generic function are chosen according to applicability and are then ordered by specificity. A method is applicable if each argument is an instance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

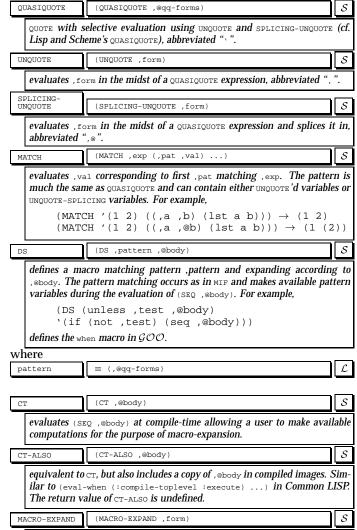


applies next most applicable method. N.B., all arguments must be supplied.

6.3 Macros

 \mathcal{G}

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. Macros are currently unhygienic, and users are required to use gensym to avoid name collisions.



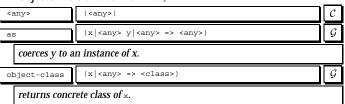
7 Scalars

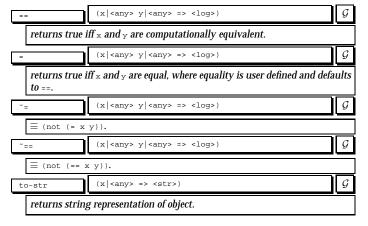
 \mathcal{GOO} provide a rich set of simple objects.

recursively expands macros in expression , form.

7.1 Any

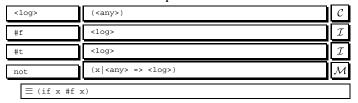
All objects are derived from <any>.



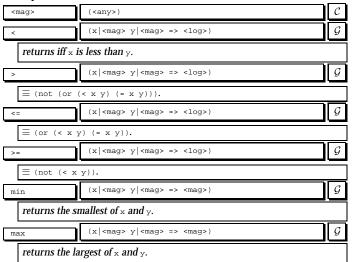


7.2 Booleans

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

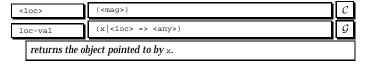


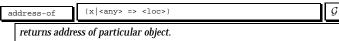
7.3 Magnitudes



7.4 Locatives

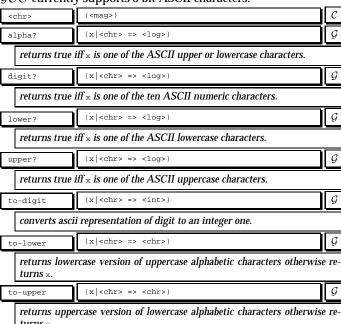
Locatives are word aligned pointers to memory. They are meant to be used to represent pointers to foreign data and not to point to interior \mathcal{GOO} object data.



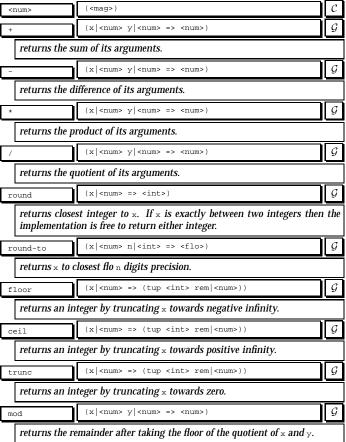


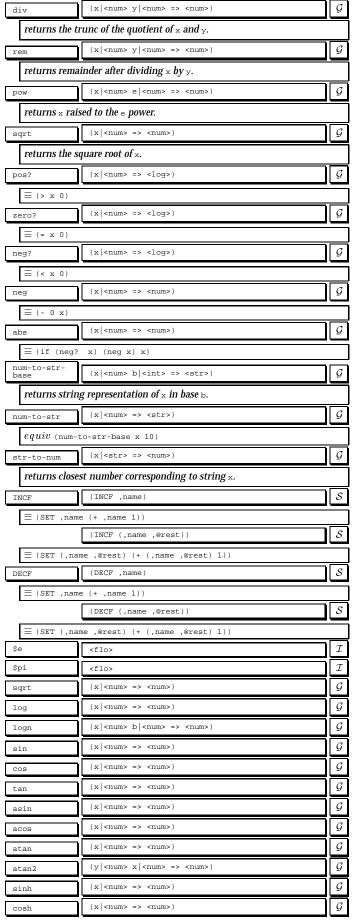
7.5 Characters

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



7.6 Numbers





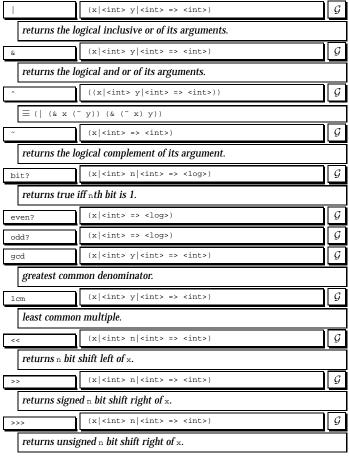


tanh

(x | <num> => <num>)

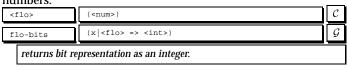
 \mathcal{G}

 \mathcal{C}



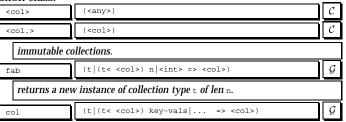
7.6.2 Floats

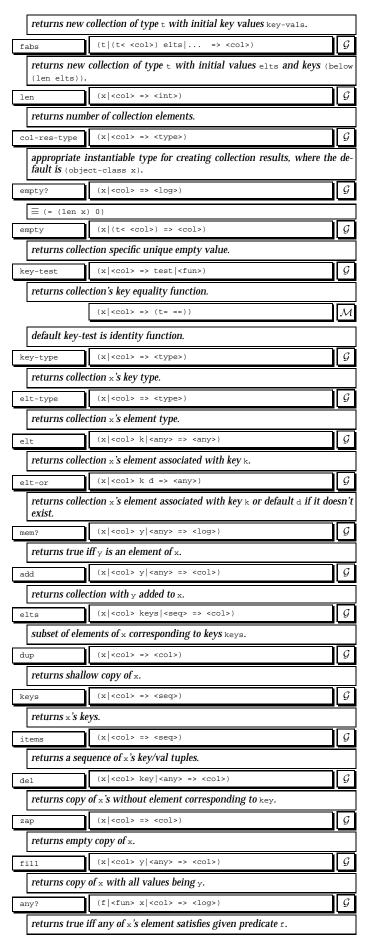
 \mathcal{GOO} currently only supports single-precision floating point numbers.

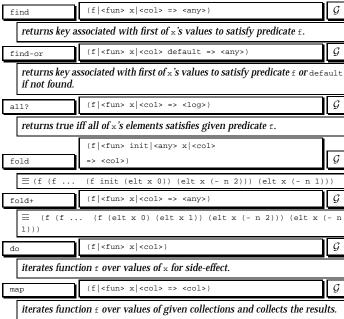


8 Collections

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

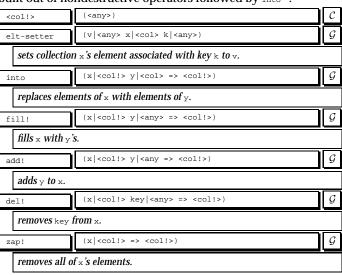






8.1 Mutable Collections

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 into 1.

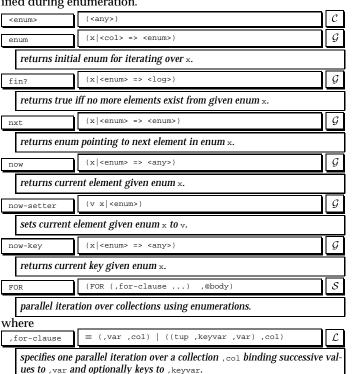


8.2 Enumerators

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,

¹When optimization is in place, the <code>!</code> suffixed functions will be deprecated.

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



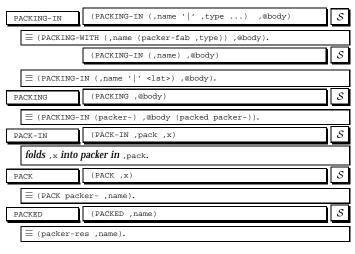
8.3 Packers

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:

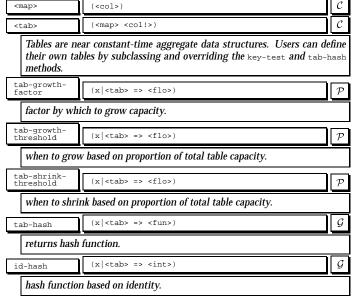
(packing-in (x|<int>)

```
(for ((e '(1 2 3 4 5)))
           (when (odd? e) (pack-in x e)))
       (packed x))
      ==> 9
<packer>
                  (<any>)
                  (p|<packer> x => <packer>)
                                                                             \mathcal{G}
  returns a copy packer p augmented with element x.
                  (p|<packer> => <any>)
                                                                             \mathcal{G}
packer-res
  returns result of packings over p.
                  (init add | <fun> res | <fun>)
                                                                             \mathcal{G}
   returns a simple packer that starts its value out with init, is augmented
   with add, and whose final value is computed with res.
                                                                             \mathcal{G}
                  (t | <type> => <packer>)
packer-fab
  returns a new type t specific packer.
                  (t|(t< <seq>) => <packer>)
packer-fab
                                                                            \mathcal{M}
  \equiv (packer
              '() pair (op as t (rev! _)))
                  (t|(t= <int>) => <packer>)
                                                                            \mathcal{M}
packer-fab
  \equiv (packer 0 + (op _{-}))
                  (PACKING-WITH ((,var ,pack) ...)
                                                                             \mathcal{S}
PACKING-WITH
  mechanism for packing objects using given packer into ,var.
```



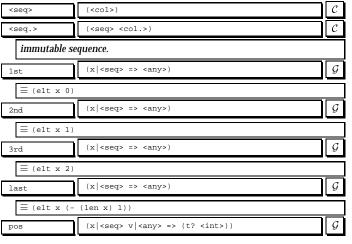
8.4 Maps

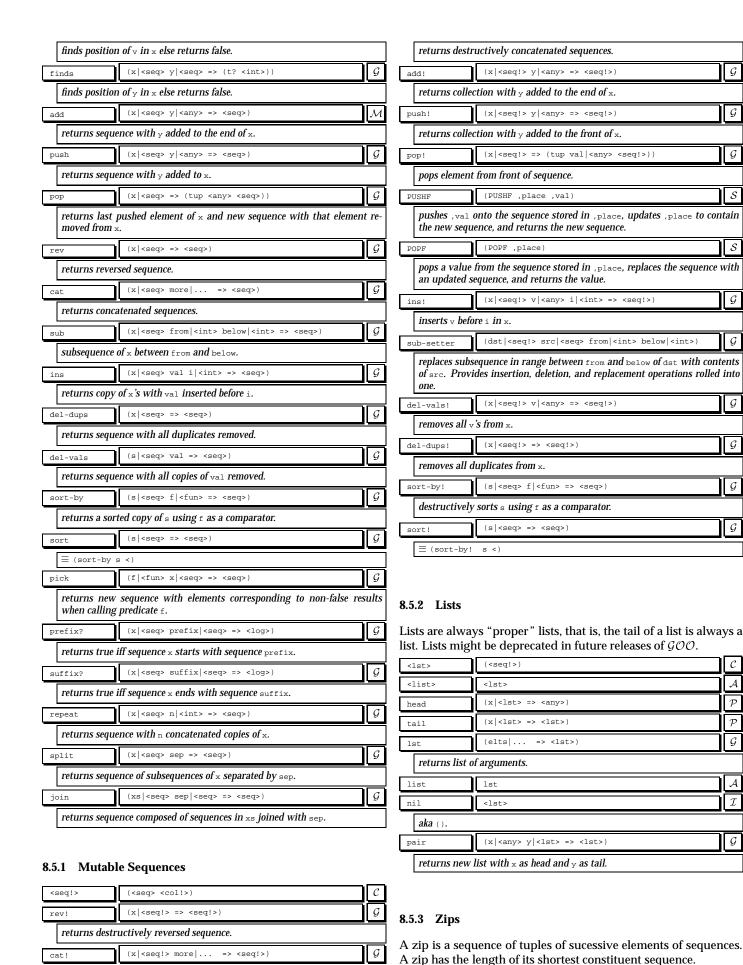
Maps represent collections with explicit keys.

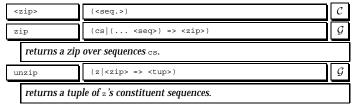


8.5 Sequences

Sequences are collections with nonnegative integer keys.

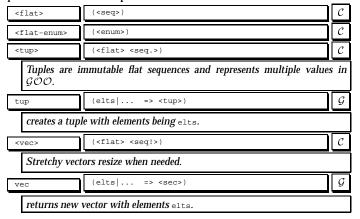






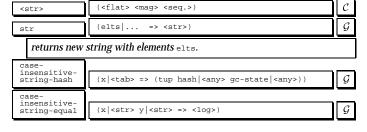
8.5.4 Flat Sequences

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



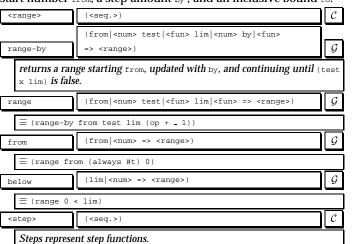
Strings

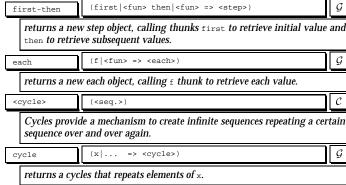
GOO currently implements ASCII strings.



8.6 Lazy Series'

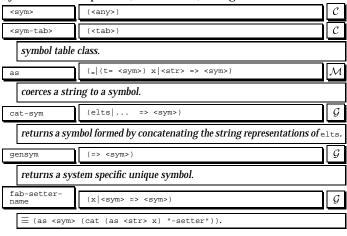
Represents an immutable sequence of numbers specified using a start number from, a step amount by , and an inclusive bound to.





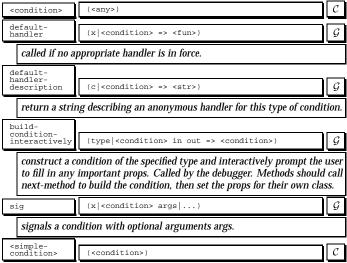
9 Symbols

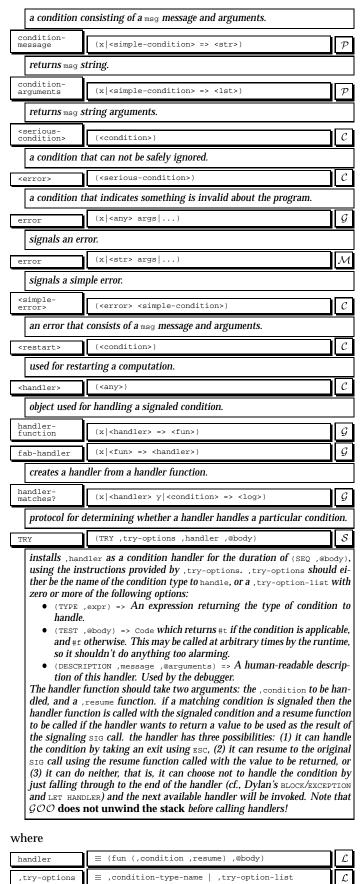
Symbols are uniquified (aka interned) strings.



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 [?] for more information.





(,try-option*

 \equiv (,option-name ,@option-value)

try-option

11 Input / Output

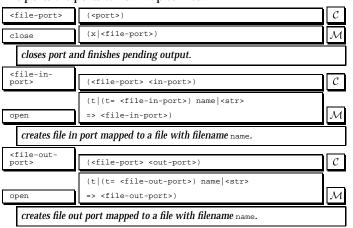
This is a very preliminary I/O system and is mostly just enough with which to write a compiler.

11.1 Ports

Ports represent character-oriented input/output devices. (<seq>) (t|(t< <port>) x|<str> => <port>) open creates port given port specific spec x. \mathcal{G} (x|<port>) close closes and cleanups port. (x | < port > \mathcal{M} noop default. (WITH-PORT (,name ,port) \mathcal{S} WITH-PORT 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. (x|<chr> => <log>) \mathcal{G} eof-object? <in-port> \mathcal{C} (<port>) input port. \mathcal{I} in <in-port> standard input. (x|<in-port> => <chr>) \mathcal{G} returns next available character or eof-object. \mathcal{G} (x|<in-port> => <str> returns a line until either reading a newline or eof-object. \mathcal{G} (x|<in-port> => <chr>) returns next available character if any without advancing pointer or eofobject. \mathcal{G} ready? (x | < in-port> => < log>)returns true iff a character is available. \mathcal{C} <out-port> (<port>) output port. \mathcal{I} <out-port> standard output. \mathcal{G} (x|<out-port>) force-out ensures that buffers are forced and pending output is completed. (x|<out-port> e|<chr>) \mathcal{G} outputs a single character. \mathcal{G} puts (x|<out-port> e|<str>) outputs string. (x|<out-port>) \mathcal{G} newline outputs a newline sequence. (x | < out-port> args | ...) \mathcal{G} \equiv (do (op say x \blacksquare) args)

11.1.1 File Ports

File ports are ports which map to files.



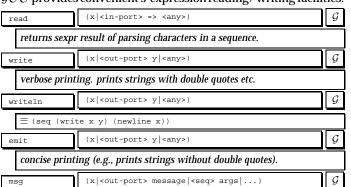
11.1.2 String Ports

String ports provide port interface mapped onto strings.



11.2 Formatted I/O

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



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

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

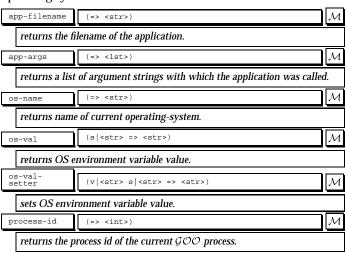
which consume one argument at a time. otherwise subsequent message characters are printed to port x (cf. Dylan's and CL's format).

characters are printed to port \times (cf. Dylan's and CL's format).

Dist (message | < seq> args | ...) g(app msg out message args)

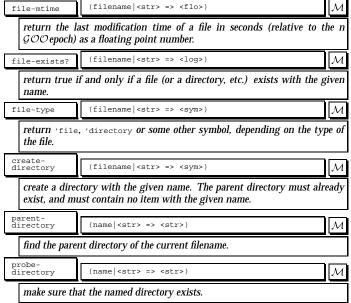
12 System

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



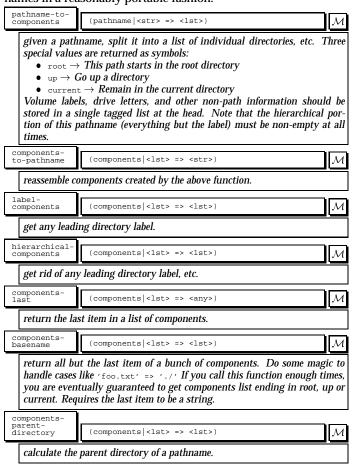
12.1 Files and Directories

A preliminary set of file and directory facilities are provided.



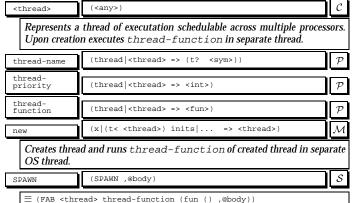
12.2 Pathnames

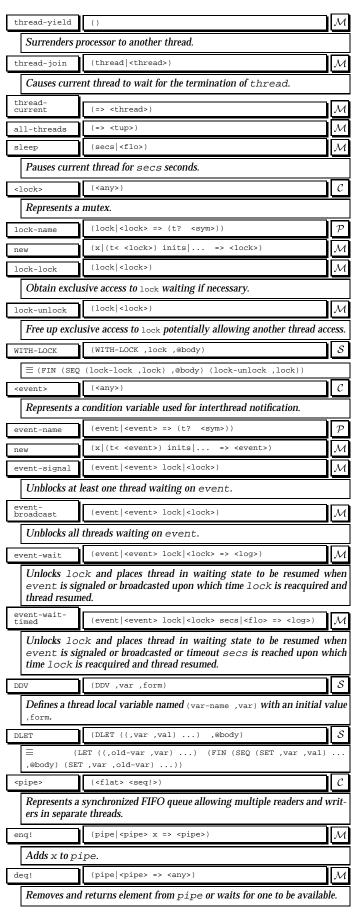
Pathnames allow you to work with hierarchical, structured pathnames in a reasonably portable fashion.



13 Threads

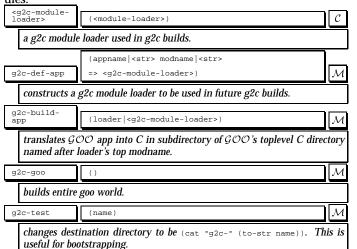
Threads allow for expressing concurrent programs. The assumed model is shared memory with explicit synchronization and symmetric multiprocessing and is based heavily upon pthreads This section is preliminary and might change in the future. There are several limitations in the current implementation. It represents a rudimentary but usable subset of typical thread operations. Tables and vectors require user locks to ensure thread safety and no out of language crashes. Finally, the compiler, interpreter, and (re)definition machinery are thread unsafe and can only reliably be run in one thread.





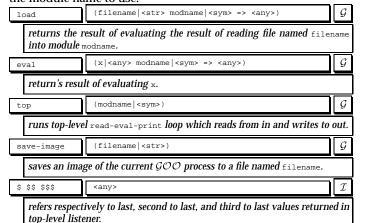
14 Compiler

 \mathcal{GOO} 's compiler, g2c, compiles \mathcal{GOO} source code to C. It lives within the <code>eval</code> module. During a given session, <code>g2c</code> recompiles only used modules that are either modified or use modified modules.



15 Top Level

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



16 Installation

Unpack a \mathcal{GOO} development or platform specific binary tarball into an appropriate staging directory. In the case of a binary tarball, there will be five directories: doc, bin, c, src, and emacs. You can just run \mathcal{GOO} from the bin subdirectory.

In the case of a development tarball, you must install it. After unpacking, there will be five directories: doc, bin, c, src, and emacs. On windows, installation must be conducted from within a cygwin shell. \mathcal{GOO} requires Boehm's GC to be installed as a shared library on linux or under the source dir GOODIR/gc downloadable from http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_source/gc.tar.gz.

Generate makefiles by executing ./configure. Configure takes a --prefix argument which sets the installation root, GOO_ROOT. Normally GOO_ROOT is set to /usr/local but can be set to staging directory for a personal installation. Configure also takes a --with-threads argument which enables threads support on linux. The rest of the

configure arguments can be found by executing ./configure --help. Execute make to build \mathcal{GOO} and make install to install it. Executing make install will build \mathcal{GOO} with the proper roots, creating 11b and mods directories, and installing \mathcal{GOO} in bin and setting up doc. On windows, it is necessary to add GOO_BIN to your PATH.

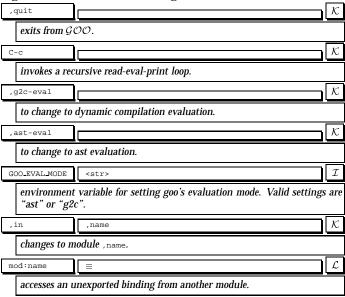
You can override the default GOO_ROOT by setting up your OS environment variable. For example, my GOO_ROOT on linux is: Setenv GOO_ROOT /home/ai/jrb/goo. Environment variable setting depends on the shell you're using. In order to run g2c-goo you need to set GOO_BUILD_ROOT to the directory which includes the STC directory. During start up, \mathcal{GOO} will load two patch files:

```
${GOO_ROOT}/init.goo
${HOME}/.goo/init.goo
```

You can customize your \mathcal{GOO} by adding forms to these files.

17 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:



17.1 Development

To batch compile \mathcal{GOO} to C:

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

To then compile the C:

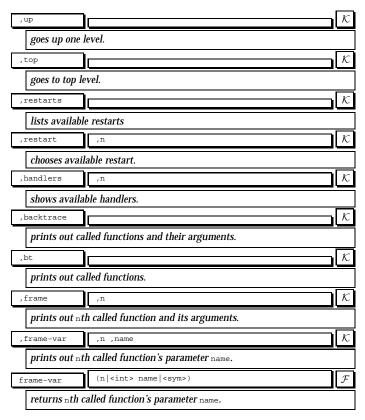
```
cd ${GOO_ROOT}/c make
```

To run the test suites:

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

17.2 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:



17.3 Emacs Support

A rudimentary emacs-based development system is provided.

17.3.1 Emacs Mode

Put <code>emacs/goo.el</code> in your emacs lisp directory. Add the following to your <code>.emacs</code> 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 $^2.$

17.3.2 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-qoo
```

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.

17.3.3 TAGS

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

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

18 Caveats

This is the first release of \mathcal{GOO} . \mathcal{GOO} is relatively slow at this point. There are no compiler optimizations in place. The error reporting is minimal and no source locations are tracked. Also hygiene is not implemented and there are some potential hygiene leaks. Dynamic compilation and image saving work only on Linux.

This manual is preliminary. Please consult the runtime libraries in the ${\tt src}$ directory. Also check out Scheme and Dylan's manuals for information on their lexical structure and special form behavior respectively.

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

19 Future

The plan is for \mathcal{GOO} to evolve in a number of dimensions. First of all, \mathcal{GOO} 's design is incomplete. Parameter lists and dispatch will be improved to allow methods of differing numbers of required parameters and named parameters. Lisp lists will most likely be deprecated and program fragments will be represented by a richer data structure which can capture source locations and hygiene information. This will be accomplished with minimal impact on macro definitions and WYSIWYG program construction and destructuring facilities. The module system will be improved to include in the very least renaming and selective imports. Finally, \mathcal{GOO} will support a more complete loopless programming protocol inspired by Waters' series [?].

Secondly, the overall mission is to crank the implementation until its performance is competitive with Java while at the same time maintaining low-latency interactivity. The basic approach involves incremental whole program optimization using simple dynamic compilation combined with partial evaluation. One important optimization will be side effect analysis combined with a generalized box/unbox optimization to remove unnecessary creation of immutable enumerators and packers for instance. Similar analyses and optimizations will be employed to optimize loopless programming patterns involving map and fold.

20 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

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

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, including by writing the Emacs goo-mode. 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, the speeding up of the runtime, and the improving of the non local exit facility. 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 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 [?] backend, and has provided useful feedback on \mathcal{GOO} 's design. I also wish to thank Boehm, Demers, and Weiser for writing the conservative GC upon which this initial version of \mathcal{GOO} is based. Finally, I would like to thank Keith Playford for his continued guidance in language design and implementation and for his ever present and rare sense of good taste.

References

- [1] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, Andrew L. M. Shalit, and P. Tucker Withington. A monotonic superclass linearization for Dylan. In *Proceedings* of the Conference on Object-Oriented Programming Systems, Languages, and Applications, volume 31, 10 of ACM SIGPLAN Notices, pages 69–82, New York, October 6–10 1996. ACM Press.
- [2] 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.
- [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 Class Precedence List

This section defines the algorithm for computing a class's linearized ancesters from its parents, its parent's parents, etc. \mathcal{GOO} uses the C3 class linearization rule [?]. The following is the \mathcal{GOO} implementation of this algorithm:

```
(dm class-ordered-ancestors (c|<class> => <lst>)
  (def parents (class-parents c))
  (rep merge-lists
          ((partial-cpl|<lst>
                (lst c))
                (remaining-lists|<lst>
```

B Subtyping Rules

This section defines the subtyping rules for \mathcal{GOO} in terms of subtype methods.

```
(dm subtype? (t1|<union> t2|<type> => <log>)
(all? (op subtype? _ t2) (type-elts t1)))
(dm subtype? (t1|<type> t2|<union> => <log>)
(dm subtype? (t1|<class> t2|<class> => <log>)
(subclass? t1 t2))
(dm subtype? (t1|<singleton> t2|<class> => <log>)
  (isa? (type-object t1) t2))
(dm subtype? (t1|<subclass> t2|<class> => <log>)
  (subclass? <class> t2))
(dm subtype? (t1 | <class> t2 | <singleton> => <log>) #f)
(dm subtype? (t1 | <singleton> t2 | <singleton> => <log>)
  (== (type-object t1) t2))
(dm subtype? (t1|<subclass> t2|<singleton> => <log>) #f)
(dm subtype? (t1|<class> t2|<subclass> => <log>)
            (and (== t1 <class>) (== (type-class t2) <class>)))
(dm subtype? (t1|<subclass> t2|<subclass> => <log>)
  (subclass? (type-class t1) (type-class t2)))
(dm subtype? (t1|<singleton> t2|<subclass> => <log>)
  (and (isa? (type-object t1) <class>)
        (subclass? (type-object t1) (type-class t2))))
(dm subtype? (t1 | sproduct> t2 | stype> => <log>) #f)
(dm subtype? (t1 | stype> t2 | sproduct> => <log>) #f)
(dm subtype? (t1|rpoduct> t2|cproduct> => <log>)
 (dm subtype? (t1|<product> t2|<class> => <log>)
  (subtype? <tup> t2))
```