GOO Reference Manual v32

Jonathan Bachrach
MIT AI Lab

April 2, 2002

1 Introduction

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] and an object-oriented Scheme [3]. GOO's main goal is to offer the best of both scripting and deliver languages. 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 GOO objects are described with definitions of the following form:

Name	*Signature*	$\mathcal{N}$

Documentation

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*	$\mathcal{N}$
`M`	*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:

`;`	*Line comment*	$\mathcal{N}$
`#/ /#`	*Nested 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* ≡ `(x t)`.	$\mathcal{N}$
`#|`	*Escaped vertical bar.*	$\mathcal{N}$

1.3 Meta Syntax

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

1.4 Conventions

The following naming conventions are used throughout this manual:

`'<'...'>'`	*Type variable*	$\mathcal{N}$
`'*'...'*'`	*Global variable*	$\mathcal{N}$
`...'?'`	*Predicate*	$\mathcal{N}$
`...'!'`	*Destructive function*	$\mathcal{N}$
`...'-setter'`	*Setter*	$\mathcal{N}$

2 Expressions

Once tokenized, GOO evaluates s-expressions in the usual lisp manner:

var	`,name`	$\mathcal{S}$

returns the value of binding named `,name` *in the current environment.*

lit	`,lit`	$\mathcal{S}$

syntactic literals that are self-evaluating.

QUOTE	`(QUOTE ,form)`	$\mathcal{S}$

with `',form` ≡ `(QUOTE ,form)` *(cf., Scheme's* `QUOTE`*)*

special	`(,special ,@args)`	$\mathcal{S}$

$\mathcal{GOO}$ defines a number of identifiers as the names of special forms, which if seen in function call position cause special form specific evaluation.

call	`(,f ,@args)`	$\mathcal{S}$

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 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.

DV	`(DV ,var ,form)`	$\mathcal{S}$

defines a global variable named `(var-name ,var)` *with an initial value* `,form` *(cf. Scheme's* `DEFINE`*).*

DEF	`(DEF ,var ,val)`	$\mathcal{S}$

locally binds `,var` *to* `,val` *and evaluates remainder of current body in the context of that binding.*

	`(DEF (TUP ,var ...) ,val)`	$\mathcal{S}$

parallel binding can also be specified using `TUP` *on the lhs of a* `DEF` *binding. For example* `(DEF (TUP x y) (TUP 1 2))`

LET	`(LET ((,var ,val) ...) ,@body)`	$\mathcal{S}$

≡ `(SEQ (DEF ,var ,val) ... ,@body)`

where

,var	≡ `,name	(,name ,type)`	$\mathcal{L}$

with `,name | ,type` ≡ `(,name ,type)` *within lists.*

SET	`(SET ,name ,form)`	$\mathcal{S}$

sets `,name` *binding to value of evaluating* `,form` *(cf. Scheme's* `SET!`*)*

	`(SET (,name ,@args) ,form)`	$\mathcal{S}$

≡ `(,name ## -setter ,form ,@args)`

USE	`(USE ,name)`	$\mathcal{S}$

loads the module `,name` *(if it hasn't been loaded already) and aliases all the exported bindings into the current namespace.*

EXPORT	`(EXPORT ,name)`	$\mathcal{S}$

makes the binding `,name` *available to code which uses this module in the future.*

USE/EXPORT	`(USE/EXPORT ,name)`	$\mathcal{S}$

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.

SEQ	`(SEQ ,@forms)`	$\mathcal{S}$

evaluates forms sequentially and returns values of evaluating last form (cf. Scheme's `BEGIN`*)*

	`(SEQ)`	$\mathcal{S}$

returns false

IF	`(IF ,test ,then [ ,else ])`	$\mathcal{S}$

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

AND	`(AND ,form ,@forms)`	$\mathcal{S}$

≡ `(IF ,form (AND ,@FORMS))`

	`(AND ,form)`	$\mathcal{S}$

≡ `,form`

OR	`(OR ,form ,@forms)`	$\mathcal{S}$

≡ `(SEQ (DEF x ,form) (IF x x (OR ,@FORMS)))`

	`(OR ,form)`	$\mathcal{S}$

≡ `,form`

UNLESS	`(UNLESS ,test ,@body)`	$\mathcal{S}$

≡ `(IF (NOT ,test) (SEQ ,@body))`

WHEN	`(WHEN ,test ,@body)`	$\mathcal{S}$

≡ `(IF ,test (SEQ ,@body))`

COND	`(COND (,test ,@body) ...)`	$\mathcal{S}$

evaluates `(SEQ ,@body)` *of first clause whose* `,test` *evaluates to non-false (cf. Dylan's* `CASE` *and Scheme's* `COND`*).*

CASE[-BY]	`(CASE[-BY] ,value [ ,test ]`
	`((,@keys) ,@body)`
	`...)`
		$\mathcal{S}$

evaluates `,value` *and then evaluates* `(SEQ ,@body)` *of first clause for which* `(,test ,value ,key)` *returns non-false (cf. Dylan's* `SELECT` *and Scheme's* `CASE`*). N.B., each key is evaluated, thus symbols must be quoted. The default* `,test` *for the* `CASE` *form is* `==`*.*

OPF	`(OPF ,place ,expr)`	$\mathcal{S}$

≡ `(SEQ (DEF _ ,place) (SET ,place ,expr))`*, where* `,place` *is evaluated only once. For example,* `(OPF x (+ _ 1))` ≡ `(SET x (+ x 1))`.

SWAPF	`(SWAPF ,x ,y)`	$\mathcal{S}$

≡ `(SEQ (DEF tmp ,x) (SET ,x ,y) (SET ,y tmp))`*, where* `,x` *and* `,y` *are evaluated only once.*

call	`(,f ,@args)`	$\mathcal{S}$

evaluates `,f` *and then* `,@args` *in left to right order and then calls* `,f` *with the evaluated arguments.*

REP	`(REP ,name ((,var ,init) ...) ,@body)`	$\mathcal{S}$

defines a recursive loop (cf., Dylan's `ITERATE` *or Scheme's* `(LET ,var ...)`*).*

ESC	`(ESC ,name ,@body)`	$\mathcal{S}$

evaluates `(SEQ ,@body)` *with an exit function of a single parameter,* `x`*, bound to* `,name` *that if called, will cause* `ESC` *to return the value of* `x` *(cf. Dylan's* `BLOCK/RETURN`*). It is illegal to call the exit function after the execution of the creating* `ESC` *form (i.e., no upward continuations).*

FIN	`(FIN ,protected ,@cleanups)`	$\mathcal{S}$

ensures that (SEQ ,@cleanups) *is evaluated whether or not an* ESC *upwards exit is taken during the dynamic-extent of* ,protected *(cf. Dylan's* BLOCK/CLEANUP *form and CL's* UNWIND-PROTECT*). The result of a* FIN *form is the result of evaluating its protected form.*

ASSERT — (ASSERT ,test ,message ,@args) — $\mathcal{S}$

≡ (UNLESS ,test (ERROR ,message ,@args))

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> — (<any>) — $\mathcal{C}$

isa? — (x|<any> y|<type> => <log>) — $\mathcal{G}$

subtype? — (x|<type> y|<type> => <log>) — $\mathcal{G}$

returns true iff x *is a subtype of* y.

new — (type|<type> prop-inits|...) — $\mathcal{G}$

creation protocol taking type and creation options where prop-inits *contains getter / initial value pairs.*

5.1 Singletons

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

<singleton> — (<type>) — $\mathcal{C}$

t= — (x|<any> => <singleton>) — $\mathcal{G}$

returns singleton constrained to x.

type-object — (x|<singleton> => <any>) — $\mathcal{G}$

object that singleton type matches.

5.2 Subclasses

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

<subclass> — (<type>) — $\mathcal{C}$

t< — (x|<class> => <subclass>) — $\mathcal{G}$

returns subclass type constrained to subclasses of x.

type-class — (x|<subclass> => <class>) — $\mathcal{G}$

object that subclass type matches.

5.3 Unions

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

<union> — (<type>) — $\mathcal{C}$

t+ — (types|... => <union>) — $\mathcal{G}$

returns union type representing disjunction of types.

type-elts — (x|<union> => <seq>) — $\mathcal{G}$

types that union type matches.

t? — (type|<type> => <union>) — $\mathcal{F}$

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

5.4 Product

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

<product> — (<type>) — $\mathcal{C}$

t* — (types|... => <product>) — $\mathcal{G}$

returns product type specifying the cross product of types.

type-elts — (x|<product> => <seq>) — $\mathcal{G}$

types that product type matches.

5.5 Classes

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

<class> — (<type>) — $\mathcal{C}$

class-name — (x|<class> => (t? <sym>)) — $\mathcal{G}$

returns class name or false otherwise.

class-parents — (x|<class> => <seq>) — $\mathcal{G}$

direct superclasses.

class-ancestors — (x|<class> => <seq>) — $\mathcal{G}$

class precedence list including this class.

class-direct-props — (x|<class> => <seq>) — $\mathcal{G}$

properties defined directly on this class.

class-props — (x|<class> => <seq>) — $\mathcal{G}$

properties defined on this class or any superclass.

class-children — (x|<class> => <seq>) — $\mathcal{G}$

direct subclasses.

DC — (DC ,name (,@parents)) — $\mathcal{S}$

defines a class named ,name *with direct parents* ,@parents

new — (type|<class> prop-inits|...) — $\mathcal{M}$

creates an instance of type type *and prop initialized as specified by* prop-inits. *For example,* (new <point> point-x 1 point-y 2) *creates a point with* x=1 *and* y=2.

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 `<prop>`. 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.

`<prop>` `(<any>)` $\mathcal{C}$

`prop-owner` `(x|<prop> => <any>)` $\mathcal{P}$

class on which property was directly defined.

`prop-getter` `(x|<prop> => <gen>)` $\mathcal{P}$

reader accessor generic.

`prop-setter` `(x|<prop> => <gen>)` $\mathcal{P}$

writer accessor generic.

`prop-type` `(x|<prop> => <type>)` $\mathcal{P}$

type constraining property value.

`prop-init` `(x|<prop> => <fun>)` $\mathcal{P}$

lazy initialization function.

`find-getter` `(c|<class> getter|<gen> => <met>)` $\mathcal{G}$

finds getter method defined on given class.

`find-setter` `(c|<class> setter|<gen> => <met>)` $\mathcal{G}$

finds setter method defined on given class.

`prop-bound?` `(x g|<gen> => <log>)` $\mathcal{P}$

returns true if property with getter `g` *is bound in instance* `x`.

`add-prop` `(owner getter|<gen> setter|<gen> type|<type> init|<fun>)` $\mathcal{M}$

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.*

`DP` `(DP ,name (,oname|,owner => ,type) [,@init])` $\mathcal{S}$

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 `<fun>`. Generic functions are instances of `<gen>` and methods are instances of `<met>`.

`<fun>` `(<any>)` $\mathcal{C}$

`fun-name` `(x|<fun> => (t? <sym>))` $\mathcal{P}$

returns the name of function or false if unavailable.

`fun-names` `(x|<fun> => <lst>)` $\mathcal{P}$

returns the names of parameters of `x` *or* `()` *if unavailable.*

`fun-specs` `(x|<fun> => <lst>)` $\mathcal{P}$

returns the specializers of `x`.

`fun-nary?` `(x|<fun> => <log>)` $\mathcal{P}$

returns true iff the function takes optional arguments.

`fun-arity` `(x|<fun> => <int>)` $\mathcal{P}$

returns `x`*'s number of required arguments.*

`fun-val` `(x|<fun> => <type>)` $\mathcal{P}$

returns the return type of `x`.

`FUN` `(FUN ,sig ,@body)` $\mathcal{S}$

creates an anonymous method with signature `,sig` *and when called evaluates* `,@body` *as* `(SEQ ,@body)` *(cf. Scheme's* LAMBDA*).*

`LOC` `(LOC ((,name ,sig ,@fbody)) ,@body)` $\mathcal{S}$

≡

```
(LET ((,name #f) ...)
 (SET ,name (fun ,sig ,@fbody)) ...
 ,@body)
```

LOC *introduces local functions that can recursively call each other (cf. Scheme's* LETREC*).*

`DF` `(DF ,name ,sig ,@body)` $\mathcal{S}$

≡ `(DV ,name (FUN ,sig ,@body))` *followed by setting the function's name.*

where

`,sig` ≡ `(,@vars) | (,@vars => ,rettype)` $\mathcal{L}$

`,rettype` ≡ `,var | (TUP ,@vars)` $\mathcal{L}$

with TUP *turning into corresponding* `t*` *function return type.*

`spread` `(x|<fun> => <fun>)` $\mathcal{G}$

≡ `(fun (y|...) (app x y))`.

`OP` `(OP ,op-arg ...)` $\mathcal{S}$

creates an anonymous function with implicitly defined arguments, where `,op-arg` *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>)` $\mathcal{G}$

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.

`<gen>`	`(<fun>)`	$\mathcal{C}$
`fun-mets`	`(x|<gen> => <lst>)`	$\mathcal{P}$
	returns `x`*'s methods.*	
`gen-add-met`	`(x|<gen> y|<met> => <gen>)`	$\mathcal{G}$
	adds method `y` *to generic* `x`*.*	
`ord-app-mets`	`(x|<gen> args|...` ` => (tup ord|<lst> amb|<lst>))`	$\mathcal{G}$
	returns both the list of sorted applicable methods and any ambiguous methods when generic `x` *is called with arguments args.*	
`DG`	`(DG ,name ,sig)`	$\mathcal{S}$
	defines a binding with name `,name` *bound to a generic with signature* `,sig`*.*	

6.2 Methods

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

`<met>`	`(<fun>)`	$\mathcal{C}$
`met-app?`	`(x|<met> args|<lst> => <log>)`	$\mathcal{G}$
	determines whether `x` *is applicable when called with* `args`*.*	
`DM`	`(DM ,name ,sig ,@body)`	$\mathcal{S}$
	first ensures that a generic exists named ,name and with a minimally congruent to signature `,sig` *and then adds a method with signature* `,sig` *and body* `,@body` *(cf., Dylan's* `DEFINE METHOD`*).*	
`SUP`	`(SUP ,@args)`	$\mathcal{S}$
	calls next most applicable method. N.B., all arguments must be supplied.	
	`(APP-SUP ,args)`	$\mathcal{S}$
	applies next most applicable method. N.B., all arguments must be supplied.	

6.3 Macros

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

`QUASIQUOTE`	`(QUASIQUOTE ,@qq-forms)`	$\mathcal{S}$
	`QUOTE` *with selective evaluation using* `UNQUOTE` *and* `SPLICING-UNQUOTE` *(cf. Lisp and Scheme's* `QUASIQUOTE`*), abbreviated "'".*	
`UNQUOTE`	`(UNQUOTE ,form)`	$\mathcal{S}$
	evaluates `,form` *in the midst of a* `QUASIQUOTE` *expression, abbreviated ",".*	
`SPLICING-UNQUOTE`	`(SPLICING-UNQUOTE ,form)`	$\mathcal{S}$
	evaluates `,form` *in the midst of a* `QUASIQUOTE` *expression and splices it in, abbreviated ",@".*	
`MATCH`	`(MATCH ,exp (,pat ,val) ...)`	$\mathcal{S}$

	evaluates `,val` *corresponding to first* `,pat` *matching* `,exp`*. The pattern is much the same as* `QUASIQUOTE` *and can contain either* `UNQUOTE`*'d variables or* `UNQUOTE-SPLICING` *variables. For example,*
	`(MATCH '(1 2) ((,a ,b) (lst a b))) → (1 2)`
	`(MATCH '(1 2) ((,a ,@b) (lst a b))) → (1 (2))`	

`DS`	`(DS ,pattern ,@body)`	$\mathcal{S}$

	defines a macro matching pattern ,pattern and expanding according to `,@body`*. The pattern matching occurs as in* `MIF` *and makes available pattern variables during the evaluation of* `(SEQ ,@body)`*. For example,*
	`(DS (when ,test ,@body)`
	` '(if (not ,test) (seq ,@body)))`
	defines the `when` *macro in* $\mathcal{GOO}$*.*	

where

`pattern`	`≡ (,@qq-forms)`	$\mathcal{L}$
`CT`	`(CT ,@body)`	$\mathcal{S}$
	evaluates `(SEQ ,@body)` *at compile-time allowing a user to make available computations for the purpose of macro-expansion.*	
`CT-ALSO`	`(CT-ALSO ,@body)`	$\mathcal{S}$
	equivalent to `CT`*, but also includes a copy of* `,@body` *in compiled images. Similar to* `(eval-when (:compile-toplevel :execute) ...)` *in Common LISP. The return value of* `CT-ALSO` *is undefined.*	
`MACRO-EXPAND`	`(MACRO-EXPAND ,form)`	$\mathcal{S}$
	recursively expands macros in expression `,form`*.*	

7 Scalars

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

7.1 Any

All objects are derived from `<any>`.

`<any>`	`(<any>)`	$\mathcal{C}$
`as`	`(x|<any> y|<any> => <any>)`	$\mathcal{G}$
	coerces y to an instance of x.	
`==`	`(x|<any> y|<any> => <log>)`	$\mathcal{G}$
	returns true iff `x` *and* `y` *are computationally equivalent.*	
`=`	`(x|<any> y|<any> => <log>)`	$\mathcal{G}$
	returns true iff `x` *and* `y` *are equal, where equality is user defined and defaults to* `==`*.*	
`~=`	`(x|<any> y|<any> => <log>)`	$\mathcal{G}$
	`≡ (not (= x y))`.	
`~==`	`(x|<any> y|<any> => <log>)`	$\mathcal{G}$
	`≡ (not (== x y))`.	
`to-str`	`(x|<any> => <str>)`	$\mathcal{G}$
	returns string representation of object.	

7.2 Booleans

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

`<log>` `(<any>)` $\mathcal{C}$

`#f` `<log>` $\mathcal{I}$

`#t` `<log>` $\mathcal{I}$

`not` `(x|<any> => <log>)` $\mathcal{M}$

≡ `(if x #f x)`

7.3 Magnitudes

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

`<mag>` `(<any>)` $\mathcal{C}$

`<` `(x|<mag> y|<mag> => <log>)` $\mathcal{G}$

returns iff `x` *is less than* `y`*.*

`>` `(x|<mag> y|<mag> => <log>)` $\mathcal{G}$

≡ `(not (or (< x y) (= x y)))`.

`<=` `(x|<mag> y|<mag> => <log>)` $\mathcal{G}$

≡ `(or (< x y) (= x y))`.

`>=` `(x|<mag> y|<mag> => <log>)` $\mathcal{G}$

≡ `(not (< x y))`.

`min` `(x|<mag> y|<mag> => <mag>)` $\mathcal{G}$

returns the smallest of `x` *and* `y`*.*

`max` `(x|<mag> y|<mag> => <mag>)` $\mathcal{G}$

returns the largest of `x` *and* `y`*.*

7.4 Locatives

Locatives are word aligned pointers to memory.

`<loc>` `(<mag>)` $\mathcal{C}$

`loc-val` `(x|<loc> => <any>)` $\mathcal{G}$

returns the object pointed to by `x`*.*

`address-of` `(x|<any> => <loc>)` $\mathcal{G}$

returns address of particular object.

7.5 Characters

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

`<chr>` `(<mag>)` $\mathcal{C}$

`alpha?` `(x|<chr> => <log>)` $\mathcal{G}$

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

`digit?` `(x|<chr> => <log>)` $\mathcal{G}$

returns true iff `x` *is one of the ten ASCII numeric characters.*

`lower?` `(x|<chr> => <log>)` $\mathcal{G}$

returns true iff `x` *is one of the ASCII lowercase characters.*

`upper?` `(x|<chr> => <log>)` $\mathcal{G}$

returns true iff `x` *is one of the ASCII uppercase characters.*

`to-digit` `(x|<chr> => <int>)` $\mathcal{G}$

converts ascii representation of digit to an integer one.

`to-lower` `(x|<chr> => <chr>)` $\mathcal{G}$

returns lowercase version of uppercase alphabetic characters otherwise returns `x`*.*

`to-upper` `(x|<chr> => <chr>)` $\mathcal{G}$

returns uppercase version of lowercase alphabetic characters otherwise returns `x`*.*

7.6 Numbers

`<num>` `(<mag>)` $\mathcal{C}$

`+` `(x|<num> y|<num> => <num>)` $\mathcal{G}$

returns the sum of its arguments.

`-` `(x|<num> y|<num> => <num>)` $\mathcal{G}$

returns the difference of its arguments.

`*` `(x|<num> y|<num> => <num>)` $\mathcal{G}$

returns the product of its arguments.

`/` `(x|<num> y|<num> => <num>)` $\mathcal{G}$

returns the quotient of its arguments.

`round` `(x|<num> => <int>)` $\mathcal{G}$

returns closest integer to `x`*. If* `x` *is exactly between two integers then the implementation is free to return either integer.*

`round-to` `(x|<num> n|<int> => <flo>)` $\mathcal{G}$

returns `x` *to closest flo* `n` *digits precision.*

`floor` `(x|<num> => (tup <int> rem|<num>))` $\mathcal{G}$

returns an integer by truncating `x` *towards negative infinity.*

`ceil` `(x|<num> => (tup <int> rem|<num>))` $\mathcal{G}$

returns an integer by truncating `x` *towards positive infinity.*

`trunc` `(x|<num> => (tup <int> rem|<num>))` $\mathcal{G}$

returns an integer by truncating `x` *towards zero.*

`mod` `(x|<num> y|<num> => <num>)` $\mathcal{G}$

returns the remainder after taking the floor of the quotient of `x` *and* `y`*.*

`rem` `(x|<num> y|<num> => <num>)` $\mathcal{G}$

returns remainder after dividing `x` *by* `y`*.*

`pow` `(x|<num> e|<num> => <num>)` $\mathcal{G}$

returns `x` *raised to the* `e` *power.*

`sqrt` `(x|<num> => <num>)` $\mathcal{G}$

returns the square root of `x`*.*

`pos?` `(x|<num> => <log>)` $\mathcal{G}$

≡ `(> x 0)`

`zero?` `(x|<num> => <log>)` $\mathcal{G}$

≡ `(= x 0)`

`neg?` `(x|<num> => <log>)` $\mathcal{G}$

≡ `(< x 0)`

neg — (x|<num> => <num>) — $\mathcal{G}$

≡ (- 0 x)

abs — (x|<num> => <num>) — $\mathcal{G}$

≡ (if (neg? x) (neg x) x)

num-to-str-base — (x|<num> b|<int> => <str>) — $\mathcal{G}$

returns string representation of x *in base* b.

num-to-str — (x|<num> => <str>) — $\mathcal{G}$

$equiv$ (num-to-str-base x 10)

str-to-num — (x|<str> => <num>) — $\mathcal{G}$

returns closest number corresponding to string x.

INCF — (INCF ,name) — $\mathcal{S}$

≡ (SET ,name (+ ,name 1))

(INCF (,name ,@rest)) — $\mathcal{S}$

≡ (SET (,name ,@rest) (+ (,name ,@rest) 1))

DECF — (DECF ,name) — $\mathcal{S}$

≡ (SET ,name (+ ,name 1))

(DECF (,name ,@rest)) — $\mathcal{S}$

≡ (SET (,name ,@rest) (+ (,name ,@rest) 1))

$e — <flo> — $\mathcal{I}$

$pi — <flo> — $\mathcal{I}$

sqrt — (x|<num> => <num>) — $\mathcal{G}$

log — (x|<num> => <num>) — $\mathcal{G}$

logn — (x|<num> b|<num> => <num>) — $\mathcal{G}$

sin — (x|<num> => <num>) — $\mathcal{G}$

cos — (x|<num> => <num>) — $\mathcal{G}$

tan — (x|<num> => <num>) — $\mathcal{G}$

asin — (x|<num> => <num>) — $\mathcal{G}$

acos — (x|<num> => <num>) — $\mathcal{G}$

atan — (x|<num> => <num>) — $\mathcal{G}$

atan2 — (y|<num> x|<num> => <num>) — $\mathcal{G}$

sinh — (x|<num> => <num>) — $\mathcal{G}$

cosh — (x|<num> => <num>) — $\mathcal{G}$

tanh — (x|<num> => <num>) — $\mathcal{G}$

7.6.1 Integers

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

<int> — (<num>) — $\mathcal{C}$

| — (x|<int> y|<int> => <int>) — $\mathcal{G}$

returns the logical inclusive or of its arguments.

& — (x|<int> y|<int> => <int>) — $\mathcal{G}$

returns the logical and or of its arguments.

^ — ((x|<int> y|<int> => <int>)) — $\mathcal{G}$

≡ (| (& x (~ y)) (& (~ x) y))

~ — (x|<int> => <int>) — $\mathcal{G}$

returns the logical complement of its argument.

bit? — (x|<int> n|<int> => <log>) — $\mathcal{G}$

returns true iff n*th bit is 1.*

even? — (x|<int> => <log>) — $\mathcal{G}$

odd? — (x|<int> => <log>) — $\mathcal{G}$

gcd — (x|<int> y|<int> => <int>) — $\mathcal{G}$

greatest common denominator.

lcm — (x|<int> y|<int> => <int>) — $\mathcal{G}$

least common multiple.

<< — (x|<int> n|<int> => <int>) — $\mathcal{G}$

returns n *bit shift left of* x.

>> — (x|<int> n|<int> => <int>) — $\mathcal{G}$

returns signed n *bit shift right of* x.

>>> — (x|<int> n|<int> => <int>) — $\mathcal{G}$

returns unsigned n *bit shift right of* x.

7.6.2 Floats

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

<flo> — (<num>) — $\mathcal{C}$

flo-bits — (x|<flo> => <int>) — $\mathcal{G}$

returns bit representation as an integer.

8 Collections

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

<col> — (<any>) — $\mathcal{C}$

<col.> — (<col>) — $\mathcal{C}$

immutable collections.

fab — (t|(t< <col>) n|<int> => <col>) — $\mathcal{G}$

returns a new instance of collection type t *of len* n.

col — (t|(t< <col>) elts|... => <col>) — $\mathcal{G}$

returns new collection of type t *with initial key values* elts*. In the case of sequences,* elts *are just the values with the keys being implicit.*

len — (x|<col> => <int>) — $\mathcal{G}$

returns number of collection elements.

col-res-type — (t|<col> x|<col> => <col>) — $\mathcal{G}$

appropriate instantiable type for creating collection results, where the default is their own type.

empty? — (x|<col> => <log>) — $\mathcal{G}$

≡ (= (len x) 0)

empty — (x|<col> => <col>) — $\mathcal{G}$

returns collection specific unique empty value.

key-test — (x|<col> => test|<fun>) — $\mathcal{G}$

returns collection's key equality function.

(x|<col> => (t= ==)) $\mathcal{M}$

default key-test is identity function.

key-type (x|<col> => <type>) $\mathcal{G}$

returns collection x*'s key type.*

elt-type (x|<col> => <type>) $\mathcal{G}$

returns collection x*'s element type.*

elt (x|<col> k|<any> => <any>) $\mathcal{G}$

returns collection x*'s element associated with key* k.

elt-or (x|<col> k d => <any>) $\mathcal{G}$

returns collection x*'s element associated with key* k *or default* d *if it doesn't exist.*

mem? (x|<col> y|<any> => <log>) $\mathcal{G}$

returns true iff y *is an element of* x.

add (x|<col> y|<any> => <col>) $\mathcal{G}$

returns collection with y *added to* x.

elts (x|<col> keys|<seq> => <col>) $\mathcal{G}$

subset of elements of x *corresponding to keys* keys.

dup (x|<col> => <col>) $\mathcal{G}$

returns shallow copy of x.

keys (x|<col> => <seq>) $\mathcal{G}$

returns x*'s keys.*

items (x|<col> => <seq>) $\mathcal{G}$

returns a sequence of x*'s key/val tuples.*

del (x|<col> key|<any> => <col>) $\mathcal{G}$

returns copy of x*'s without element corresponding to* key.

zap (x|<col> => <col>) $\mathcal{G}$

returns empty copy of x.

fill (x|<col> y|<any> => <col>) $\mathcal{G}$

returns copy of x *with all values being* y.

any? (f|<fun> x|<col> => <log>) $\mathcal{G}$

returns true iff any of x*'s element satisfies given predicate* f.

find (f|<fun> x|<col> => <any>) $\mathcal{G}$

returns key associated with first of x*'s values to satisfy predicate* f.

find-or (f|<fun> x|<col> default => <any>) $\mathcal{G}$

returns key associated with first of x*'s values to satisfy predicate* f *or* default *if not found.*

all? (f|<fun> x|<col> => <log>) $\mathcal{G}$

returns true iff all of x*'s elements satisfies given predicate* f.

fold (f|<fun> init|<any> x|<col> => <col>) $\mathcal{G}$

≡ (f (f ... (f init (elt x 0)) (elt x (- n 2))) (elt x (- n 1)))

fold+ (f|<fun> x|<col> => <any>) $\mathcal{G}$

≡ (f (f ... (f (elt x 0) (elt x 1)) (elt x (- n 2))) (elt x (- n 1)))

do (f|<fun> x|<col>) $\mathcal{G}$

iterates function f *over values of* x *for side-effect.*

map (f|<fun> x|<col> => <col>) $\mathcal{G}$

iterates function f *over values of given collections and collects the results.*

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. The ! suffixed functions might be deprecated in the future.

<col!> (<any>) $\mathcal{C}$

elt-setter (v|<any> x|<col> k|<any>) $\mathcal{G}$

sets collection x*'s element associated with key* k *to* v.

into (x|<col!> y|<col> => <col!>) $\mathcal{G}$

replaces elements of x *with elements of* y.

fill! (x|<col!> y|<any> => <col!>) $\mathcal{G}$

fills x *with* y*'s.*

add! (x|<col!> y|<any => <col!>) $\mathcal{G}$

adds y *to* x.

del! (x|<col!> key|<any> => <col!>) $\mathcal{G}$

removes key *from* x.

zap! (x|<col!> => <col!>) $\mathcal{G}$

removes all of x*'s elements.*

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.

<enum> (<any>) $\mathcal{C}$

enum (x|<col> => <enum>) $\mathcal{G}$

returns initial enum for iterating over x.

fin? (x|<enum> => <log>) $\mathcal{G}$

returns true iff no more elements exist from given enum x.

nxt (x|<enum> => <enum>) $\mathcal{G}$

returns enum pointing to next element in enum x.

now (x|<enum> => <any>) $\mathcal{G}$

returns current element given enum x.

now-setter (v x|<enum>) $\mathcal{G}$

sets current element given enum x *to* v.

now-key (x|<enum> => <any>) $\mathcal{G}$

returns current key given enum x.

FOR | (FOR (,for-clause ...) ,@body) | $\mathcal{S}$

parallel iteration over collections using enumerations.

where

,for-clause | ≡ (,var ,col) | ((tup ,keyvar ,var) ,col) | $\mathcal{L}$

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

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:

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

<packer> | (<any>) | $\mathcal{C}$

packer-add | (p|<packer> x => <packer>) | $\mathcal{G}$

returns a copy packer p *augmented with element* x.

packer-res | (p|<packer> => <any>) | $\mathcal{G}$

returns result of packings over p.

packer | (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.

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

returns a new type t *specific packer.*

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

≡ (packer '() pair (op as t (rev! _)))

packer-fab | (t|(t= <int>) => <packer>) | $\mathcal{M}$

≡ (packer 0 + (op _))

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) | $\mathcal{S}$

≡ (PACKING-WITH (,name (packer-fab ,type)) ,@body).

(PACKING-IN (,name) ,@body) | $\mathcal{S}$

≡ (PACKING-IN (,name '|' <lst>) ,@body).

PACKING | (PACKING ,@body) | $\mathcal{S}$

≡ (PACKING-IN (packer-) ,@body (packed packer-)).

PACK-IN | (PACK-IN ,pack ,x) | $\mathcal{S}$

folds ,x *into packer in* ,pack.

PACK | (PACK ,x) | $\mathcal{S}$

≡ (PACK packer- ,name).

PACKED | (PACKED ,name) | $\mathcal{S}$

≡ (packer-res ,name).

8.4 Maps

Maps represent collections with explicit keys.

<map> | (<col>) | $\mathcal{C}$

<tab> | (<map> <col!>) | $\mathcal{C}$

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

tab-growth-factor | (x|<tab> => <flo>) | $\mathcal{P}$

factor by which to grow capacity.

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

when to grow based on proportion of total table capacity.

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

when to shrink based on proportion of total table capacity.

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

returns hash function.

id-hash | (x|<tab> => <int>) | $\mathcal{G}$

hash function based on identity.

8.5 Sequences

Sequences are collections with nonnegative integer keys.

<seq> | (<col>) | $\mathcal{C}$

<seq.> | (<seq> <col.>) | $\mathcal{C}$

immutable sequence.

1st | (x|<seq> => <any>) | $\mathcal{G}$

≡ (elt x 0)

2nd | (x|<seq> => <any>) | $\mathcal{G}$

≡ (elt x 1)

3rd | (x|<seq> => <any>) | $\mathcal{G}$

≡ (elt x 2)

last | (x|<seq> => <any>) | $\mathcal{G}$

≡ (elt x (- (len x) 1))

pos | (x|<seq> v|<any> => (t? <int>)) | $\mathcal{G}$

finds position of v *in* x *else returns false.*

finds | (x|<seq> y|<seq> => (t? <int>)) | $\mathcal{G}$

finds position of y *in* x *else returns false.*

add | (x|<seq> y|<any> => <seq>) | $\mathcal{M}$

returns sequence with y *added to the end of* x.

push | (x|<seq> y|<any> => <seq>) | $\mathcal{G}$

returns sequence with y *added to* x.

pop | (x|<seq> => (tup <any> <seq>)) | $\mathcal{G}$

returns last pushed element of x *and new sequence with that element removed from* x.

rev | (x|<seq> => <seq>) | $\mathcal{G}$

returns reversed sequence.

cat | (x|<seq> more|... => <seq>) | $\mathcal{G}$

returns concatenated sequences.

`sub` (x|<seq> from|<int> below|<int> => <seq>) $\mathcal{G}$

subsequence of `x` *between* `from` *and* `below`.

`ins` (x|<seq> val i|<int> => <seq>) $\mathcal{G}$

returns copy of `x`*'s with* `val` *inserted before* `i`.

`del-dups` (x|<seq> => <seq>) $\mathcal{G}$

returns sequence with all duplicates removed.

`del-vals` (s|<seq> val => <seq>) $\mathcal{G}$

returns sequence with all copies of `val` *removed.*

`sort-by` (s|<seq> f|<fun> => <seq>) $\mathcal{G}$

returns a sorted copy of `s` *using* `f` *as a comparator.*

`sort` (s|<seq> => <seq>) $\mathcal{G}$

≡ (sort-by s <)

`pick` (f|<fun> x|<seq> => <seq>) $\mathcal{G}$

returns new sequence with elements corresponding to non-false results when calling predicate `f`.

`prefix?` (x|<seq> prefix|<seq> => <log>) $\mathcal{G}$

returns true iff sequence `x` *starts with sequence* `prefix`.

`suffix?` (x|<seq> suffix|<seq> => <log>) $\mathcal{G}$

returns true iff sequence `x` *ends with sequence* `suffix`.

`repeat` (x|<seq> n|<int> => <seq>) $\mathcal{G}$

returns sequence with `n` *concatenated copies of* `x`.

`split` (x|<seq> sep => <seq>) $\mathcal{G}$

returns sequence of subsequences of `x` *separated by* `sep`.

`join` (xs|<seq> sep|<seq> => <seq>) $\mathcal{G}$

returns sequence composed of sequences in `xs` *joined with* `sep`.

8.5.1 Mutable Sequences

`<seq!>` (<seq> <col!>) $\mathcal{C}$

`rev!` (x|<seq!> => <seq!>) $\mathcal{G}$

returns destructively reversed sequence.

`cat!` (x|<seq!> more|... => <seq!>) $\mathcal{G}$

returns destructively concatenated sequences.

`add!` (x|<seq!> y|<any> => <seq!>) $\mathcal{G}$

returns collection with `y` *added to the end of* `x`.

`push!` (x|<seq!> y|<any> => <seq!>) $\mathcal{G}$

returns collection with `y` *added to the front of* `x`.

`pop!` (x|<seq!> => (tup val|<any> <seq!>)) $\mathcal{G}$

pops element from front of sequence.

`PUSHF` (PUSHF ,place ,val) $\mathcal{S}$

pushes `,val` *onto the sequence stored in* `,place`*, updates* `,place` *to contain the new sequence, and returns the new sequence.*

`POPF` (POPF ,place) $\mathcal{S}$

pops a value from the sequence stored in `,place`*, replaces the sequence with an updated sequence, and returns the value.*

`ins!` (x|<seq!> v|<any> i|<int> => <seq!>) $\mathcal{G}$

inserts `v` *before* `i` *in* `x`.

`sub-setter` (dst|<seq!> src|<seq> from|<int> below|<int>) $\mathcal{G}$

replaces subsequence in range between `from` *and* `below` *of* `dst` *with contents of* `src`*. Provides insertion, deletion, and replacement operations rolled into one.*

`del-vals!` (x|<seq!> v|<any> => <seq!>) $\mathcal{G}$

removes all `v`*'s from* `x`.

`del-dups!` (x|<seq!> => <seq!>) $\mathcal{G}$

removes all duplicates from `x`.

`sort-by!` (s|<seq> f|<fun> => <seq>) $\mathcal{G}$

destructively sorts `s` *using* `f` *as a comparator.*

`sort!` (s|<seq> => <seq>) $\mathcal{G}$

≡ (sort-by! s <)

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 $\mathcal{GOO}$.

`<lst>` (<seq!>) $\mathcal{C}$

`<list>` <lst> $\mathcal{A}$

`head` (x|<lst> => <any>) $\mathcal{P}$

`tail` (x|<lst> => <lst>) $\mathcal{P}$

`lst` (elts|... => <lst>) $\mathcal{G}$

returns list of arguments.

`list` lst $\mathcal{A}$

`nil` <lst> $\mathcal{I}$

aka `()`.

`pair` (x|<any> y|<lst> => <lst>) $\mathcal{G}$

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

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.

`<zip>` (<seq.>) $\mathcal{C}$

`zip` (cs|(... <seq>) => <zip>) $\mathcal{G}$

returns a zip over sequences `cs`.

`unzip` (z|<zip> => <tup>) $\mathcal{G}$

returns a tuple of `z`*'s constituent sequences.*

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`.

`<flat>` (<seq>) $\mathcal{C}$

`<flat-enum>` (<enum>) $\mathcal{C}$

`<tup>` (<flat> <seq.>) $\mathcal{C}$

Tuples are immutable flat sequences and represents multiple values in $\mathcal{GOO}$.

`tup` — `(elts|... => <tup>)` — 𝒢

creates a tuple with elements being `elts`.

`<vec>` — `(<flat> <seq!>)` — 𝒞

Stretchy vectors resize when needed.

`vec` — `(elts|... => <sec>)` — 𝒢

returns new vector with elements `elts`.

Strings

𝒢𝒪𝒪 currently implements ASCII strings.

`<str>` — `(<flat> <mag> <seq.>)` — 𝒞

`str` — `(elts|... => <str>)` — 𝒢

returns new string with elements `elts`.

`case-insensitive-string-hash` — `(x|<tab> => (tup hash|<any> gc-state|<any>))` — 𝒢

`case-insensitive-string-equal` — `(x|<str> y|<str> => <log>)` — 𝒢

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`.

`<range>` — `(<seq.>)` — 𝒞

`range-by` — `(from|<num> test|<fun> lim|<num> by|<fun> => <range>)` — 𝒢

returns a range starting `from`*, updated with* `by`*, and continuing until* `(test x lim)` *is false.*

`range` — `(from|<num> test|<fun> lim|<fun> => <range>)` — 𝒢

≡ `(range-by from test lim (op + _ 1))`

`from` — `(from|<num> => <range>)` — 𝒢

≡ `(range from (always #t) 0)`

`below` — `(lim|<num> => <range>)` — 𝒢

≡ `(range 0 < lim)`

`<step>` — `(<seq.>)` — 𝒞

Steps represent step functions.

`first-then` — `(first|<fun> then|<fun> => <step>)` — 𝒢

returns a new step function, calling thunks `first` *to retrieve initial value and* `then` *to retrieve subsequent values.*

`<cycle>` — `(<seq.>)` — 𝒞

Cycles provide a mechanism to create infinite sequences repeating a certain sequence over and over again.

`cycle` — `(x|... => <cycle>)` — 𝒢

returns a cycles that repeats elements of `x`.

9 Symbols

Symbols are uniquified (aka interned) strings.

`<sym>` — `(<any>)` — 𝒞

`<sym-tab>` — `(<tab>)` — 𝒞

symbol table class.

`as` — `(_|(t= <sym>) x|<str> => <sym>)` — ℳ

coerces a string to a symbol.

`cat-sym` — `(elts|... => <sym>)` — 𝒢

returns a symbol formed by concatenating the string representations of `elts`.

`gensym` — `(=> <sym>)` — 𝒢

returns a system specific unique symbol.

`fab-setter-name` — `(x|<sym> => <sym>)` — 𝒢

≡ `(as <sym> (cat (as <str> x) "-setter"))`.

10 Conditions

Conditions are objects representing exceptional situations. 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.

`<condition>` — `(<any>)` — 𝒞

`default-handler` — `(x|<condition> => <fun>)` — 𝒢

called if no appropriate handler is in force.

`default-handler-description` — `(c|<condition> => <str>)` — 𝒢

return a string describing an anonymous handler for this type of condition.

`build-condition-interactively` — `(type|<condition> in out => <condition>)` — 𝒢

construct a condition of the specified type and interactively prompt the user to fill in any important props. Called by the debugger. Methods should call `next-method` *to build the condition, then set the props for their own class.*

`sig` — `(x|<condition> args|...)` — 𝒢

signals a condition with optional arguments args.

`<simple-condition>` — `(<condition>)` — 𝒞

a condition consisting of a `msg` *message and arguments.*

`condition-message` — `(x|<simple-condition> => <str>)` — 𝒫

returns `msg` *string.*

`condition-arguments` — `(x|<simple-condition> => <lst>)` — 𝒫

returns `msg` *string arguments.*

`<serious-condition>` — `(<condition>)` — 𝒞

a condition that can not be safely ignored.

`<error>` — `(<serious-condition>)` — 𝒞

a condition that indicates something is invalid about the program.

`error` — `(x|<any> args|...)` — 𝒢

signals an error.

`error` — `(x|<str> args|...)` — ℳ

signals a simple error.

<simple-error>	(<error> <simple-condition>)	$\mathcal{C}$

an error that consists of a msg *message and arguments.*

<restart>	(<condition>)	$\mathcal{C}$

used for restarting a computation.

<handler>	(<any>)	$\mathcal{C}$

object used for handling a signaled condition.

handler-function	(x|<handler> => <fun>)	$\mathcal{G}$
fab-handler	(x|<fun> => <handler>)	$\mathcal{G}$

creates a handler from a handler function.

handler-matches?	(x|<handler> y|<condition> => <log>)	$\mathcal{G}$

protocol for determining whether a handler handles a particular condition.

TRY	(TRY ,try-options ,handler ,@body)	$\mathcal{S}$

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 ,condition *to be handled, and a* ,resume *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* SIG *call. the handler has three possibilities: (1) it can handle the condition by taking an exit using* ESC*, (2) it can resume to the original* SIG *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* BLOCK/EXCEPTION *and* LET HANDLER*) 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)	$\mathcal{L}$

noop default.

WITH-PORT	(WITH-PORT (,name ,port) ,@body)	$\mathcal{S}$

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>)	$\mathcal{G}$
<in-port>	(<port>)	$\mathcal{C}$

input port.

in	<in-port>	$\mathcal{I}$

standard input.

get	(x|<in-port> => <chr>)	$\mathcal{G}$

returns next available character or eof-object.

gets	(x|<in-port> => <str>)	$\mathcal{G}$

returns a line until either reading a newline or eof-object.

peek	(x|<in-port> => <chr>)	$\mathcal{G}$

returns next available character if any without advancing pointer or eof-object.

ready?	(x|<in-port> => <log>)	$\mathcal{G}$

returns true iff a character is available.

<out-port>	(<port>)	$\mathcal{C}$

output port.

out	<out-port>	$\mathcal{I}$

standard output.

force-out	(x|<out-port>)	$\mathcal{G}$

ensures that buffers are forced and pending output is completed.

put	(x|<out-port> e|<chr>)	$\mathcal{G}$

outputs a single character.

puts	(x|<out-port> e|<str>)	$\mathcal{G}$

outputs string.

newline	(x|<out-port>)	$\mathcal{G}$

outputs a newline sequence.

say	(x|<out-port> args|...)	$\mathcal{G}$

≡ (do (op say x _) args)

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.

<port>	(<seq>)	$\mathcal{C}$
open	(t|(t< <port>) x|<str> => <port>)	$\mathcal{G}$

creates port given port specific spec x*.*

close	(x|<port>)	$\mathcal{G}$

closes and cleanups port.

	(x|<port>)	$\mathcal{M}$

11.1.1 File Ports

File ports are ports which map to files.

<file-port>	(<port>)	$\mathcal{C}$
close	(x|<file-port>)	$\mathcal{M}$

closes port and finishes pending output.

<file-in-port>	(<file-port> <in-port>)	$\mathcal{C}$
open	(t|(t= <file-in-port>) name|<str> => <file-in-port>)	$\mathcal{M}$

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

<file-out-port>	(<file-port> <out-port>)	$\mathcal{C}$
open	(t|(t= <file-out-port>) name|<str> => <file-out-port>)	$\mathcal{M}$

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

11.1.2 String Ports

String ports provide port interface mapped onto strings.

<str-port> — (<any>) — $\mathcal{C}$

port-contents — (x|<str-port> => <str>) — $\mathcal{P}$

returns underlying string.

<str-in-port> — (<str-port> <out-port>) — $\mathcal{C}$

open — (t|(t= <str-in-port>) dat|<str> => <str-in-port>) — $\mathcal{M}$

creates string in port mapped to string dat.

port-index — (x|<str-port> => <int>) — $\mathcal{P}$

returns index from which next character will be read.

<str-out-port> — (<str-port> <in-port>) — $\mathcal{C}$

open — (t|(t= <str-out-port>) dat|<str> => <str-out-port>) — $\mathcal{M}$

creates string out port mapped to string dat.

PORT2STR — (PORT2STR ,name ,@body) — $\mathcal{S}$

≡ (let ((,name (open <str-out-port> ""))) ,@body (port-contents ,name))

11.2 Formatted I/O

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

read — (x|<in-port> => <any>) — $\mathcal{G}$

returns sexpr result of parsing characters in a sequence.

write — (x|<out-port> y|<any>) — $\mathcal{G}$

verbose printing. prints strings with double quotes etc.

display — (x|<out-port> y|<any>) — $\mathcal{G}$

non verbose printing. prints strings without double quotes etc.

writeln — (x|<out-port> y|<any>) — $\mathcal{G}$

≡ (seq (write x y) (newline))

msg — (x|<out-port> message|<seq> args|...) — $\mathcal{G}$

formatted output using special commands embedded in message. supported 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.

app-args — (=> <lst>) — $\mathcal{M}$

returns a list of argument strings with which the application was called.

os-name — (=> <str>) — $\mathcal{M}$

returns name of current operating-system.

os-val — (s|<str> => <str>) — $\mathcal{M}$

returns OS environment variable value.

os-val-setter — (v|<str> s|<str> => <str>) — $\mathcal{M}$

sets OS environment variable value.

process-id — (=> <int>) — $\mathcal{M}$

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

12.1 Files and Directories

A preliminary set of file and directory facilities are provided.

file-mtime — (filename|<str> => <flo>) — $\mathcal{M}$

return the last modification time of a file in seconds (relative to the $\mathcal{GOO}$*epoch) as a floating point number.*

file-exists? — (filename|<str> => <log>) — $\mathcal{M}$

return true if and only if a file (or a directory, etc.) exists with the given name.

file-type — (filename|<str> => <sym>) — $\mathcal{M}$

return 'file, 'directory *or some other symbol, depending on the type of the file.*

create-directory — (filename|<str> => <sym>) — $\mathcal{M}$

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.

probe-directory — (name|<str> => <str>) — $\mathcal{M}$

make sure that the named directory exists.

12.2 Pathnames

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

pathname-to-components — (pathname|<str> => <lst>) — $\mathcal{M}$

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

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

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

components-to-pathname — (components|<lst> => <str>) — $\mathcal{M}$

reassemble components created by the above function.

label-components — (components|<lst> => <lst>) — $\mathcal{M}$

get any leading directory label.

hierarchical-components — (components|<lst> => <lst>) — $\mathcal{M}$

get rid of any leading directory label, etc.

`components-last`	`(components|<lst> => <any>)`	$\mathcal{M}$
	return the last item in a list of components.	
`components-basename`	`(components|<lst> => <lst>)`	$\mathcal{M}$
	return all but the last item of a bunch of components. Do some magic to handle cases like `'foo.txt' => './'` *If you call this function enough times, you are eventually guaranteed to get components list ending in root, up or current. Requires the last item to be a string.*	
`components-parent-directory`	`(components|<lst> => <lst>)`	$\mathcal{M}$
	calculate the parent directory of a pathname.	

13 Compiler

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

`<g2c-module-loader>`	`(<module-loader>)`	$\mathcal{C}$
	a g2c module loader used in g2c builds.	
`g2c-def-app`	`(appname|<str> modname|<str> => <g2c-module-loader>)`	$\mathcal{M}$
	constructs a g2c module loader to be used in future g2c builds.	
`g2c-build-app`	`(loader|<g2c-module-loader>)`	$\mathcal{M}$
	translates $\mathcal{GOO}$ app into C in subdirectory of $\mathcal{GOO}$'s toplevel C directory named after loader's top modname.	
`g2c-goo`	`()`	$\mathcal{M}$
	builds entire goo world.	
`g2c-test`	`(name)`	$\mathcal{M}$
	changes destination directory to be `(cat "g2c-" (to-str name))`*. This is useful for bootstrapping.*	

14 Top Level

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

`load`	`(filename|<str> modname|<sym> => <any>)`	$\mathcal{G}$
	returns the result of evaluating the result of reading file named `filename` *into module* `modname`.	
`eval`	`(x|<any> modname|<sym> => <any>)`	$\mathcal{G}$
	return's result of evaluating `x`.	
`top`	`(modname|<sym>)`	$\mathcal{G}$
	runs top-level `read-eval-print` *loop which reads from in and writes to out.*	
`save-image`	`(filename|<str>)`	$\mathcal{G}$
	saves an image of the current $\mathcal{GOO}$ process to a file named `filename`.	

15 Installation

Unpack appropriate installation area. There are four directories: `doc`, `bin`, `src`, and `emacs`. Edit the `Makefile` in the top level directory, to configure the installation directory root using the `PREFIX` if `/usr` is inapproriate. Executing `make` will rebuild $\mathcal{GOO}$ with the proper roots, creating `lib` and `mods` directories, and installing $\mathcal{GOO}$ in `bin` and setting up `doc`.
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. 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.

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:

`,quit`		$\mathcal{K}$
	exits from $\mathcal{GOO}$.	
`C-c`		$\mathcal{K}$
	invokes a recursive read-eval-print loop.	
`,g2c-eval`		$\mathcal{K}$
	to change to dynamic compilation evaluation.	
`,ast-eval`		$\mathcal{K}$
	to change to ast evaluation.	
`,in`	`,name`	$\mathcal{K}$
	changes to module `,name`.	
`mod:name`	$\equiv$	$\mathcal{L}$
	accesses an unexported binding from another module.	

16.1 Development

To batch compile $\mathcal{GOO}$ to C:

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

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)
```

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:

`,up`		$\mathcal{K}$
	goes up one level.	
`,top`		$\mathcal{K}$
	goes to top level.	

`,restarts` — $\mathcal{K}$

lists available restarts

`,restart` `,n` — $\mathcal{K}$

chooses available restart.

`,handlers` `,n` — $\mathcal{K}$

shows available handlers.

`,backtrace` — $\mathcal{K}$

prints out called functions and their arguments.

`,bt` — $\mathcal{K}$

prints out called functions.

`,frame` `,n` — $\mathcal{K}$

prints out `n`*th called function and its arguments.*

16.3 Emacs Support

A rudimentary emacs-based development system is provided.

16.3.1 Emacs Mode

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

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

Useful features include the following. You can add "font-lock" mode by adding `(global-font-lock-mode t)` to your `.emacs`: In a given buffer, you can toggle font-lock with `M-x font-lock-mode`. Finally, check out the "Index" menu item in a GOO buffer for other options.
For even more fun, load `emacs/goo-font-lock.el` for a color coded parenthesis nesting aid [1].

16.3.2 Emacs Shell

Put `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 / key-bindings. I doubt the compile commands do anything useful cause there isn't a compiler.

[1] The original idea was dreamed up and first implemented by Andrew Sutherland and then improved by James Knight.

16.3.3 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
```

17 Caveats

This is the first release of GOO. 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 on their lexical structure and special form behavior respectively.
Please, please, please send bug reports to `jrb@googaga.org`. I will fix your bugs asap. The GOO website `www.googaga.org` will have papers, releases, FAQS, etc.

18 History and Acknowledgements

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 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, 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, GOO, and it stuck. Andrew Sutherland became my MEng student in the winter of 2002, wrote a GOO SWIG [1] backend, and has provided useful feedback on 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 and rare 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. Revised[5] 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.