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, more dynamic, lispsyntaxed Dylan [4] and an object-oriented Scheme [3]. 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. 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 GOO objects are described with definitions of the following form:

```
Name Signature Documentation
```

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

<table>
<thead>
<tr>
<th>N</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Lexical</td>
</tr>
<tr>
<td>S</td>
<td>Syntax</td>
</tr>
<tr>
<td>G</td>
<td>Generic</td>
</tr>
<tr>
<td>M</td>
<td>Method</td>
</tr>
<tr>
<td>F</td>
<td>Function</td>
</tr>
<tr>
<td>C</td>
<td>Class</td>
</tr>
<tr>
<td>P</td>
<td>Property</td>
</tr>
<tr>
<td>I</td>
<td>Instance</td>
</tr>
<tr>
<td>K</td>
<td>Command</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>; ...</td>
<td>Line comment</td>
</tr>
<tr>
<td>#/ ... /#</td>
<td>Nested comment</td>
</tr>
<tr>
<td>. + - [0-9]+</td>
<td>Number</td>
</tr>
<tr>
<td>#e #i #b #o #d #x #t #f #k</td>
<td>Special number</td>
</tr>
<tr>
<td>#t #f</td>
<td>Logical</td>
</tr>
<tr>
<td>#\name</td>
<td>Character</td>
</tr>
<tr>
<td>(a [b c] d)</td>
<td>Identifier</td>
</tr>
<tr>
<td>( ... )</td>
<td>List</td>
</tr>
<tr>
<td>#( ... )</td>
<td>Tuple</td>
</tr>
<tr>
<td>#( ... )</td>
<td>Vector</td>
</tr>
<tr>
<td>* ... *</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Special character’s within strings</td>
</tr>
</tbody>
</table>
| | Typed variable within list \( (k t) \).
| | Escaped vertical bar.
| \[ ... \] | Lightweight function syntax.
| " ... " | Collection indexing and slicing.

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 "~name" 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:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;...&gt;</td>
<td>Type variable</td>
</tr>
<tr>
<td><em>...</em></td>
<td>Global variable</td>
</tr>
<tr>
<td>...?</td>
<td>Predicate</td>
</tr>
<tr>
<td>.../</td>
<td>Destructor function</td>
</tr>
<tr>
<td>.../~</td>
<td>Setter</td>
</tr>
</tbody>
</table>

2 Expressions

Once tokenized, GOO evaluates s-expressions in the usual lisp manner:
where
returns the value of binding named ,name in the current environment.

lit
syntactic literals that are self-evaluating.

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

DEF
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
otherwise lists represent function calls.

3 Namespaces and Bindings

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, 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.

- (D. ,var ,form)
defines a global constant named (var-name ,var) with an initial value ,form (cf. Dylan's DEFINE CONSTANT).

- (DV ,var ,form)
defines a global variable named (var-name ,var) with an initial value ,form (cf. Dylan's DEFINE VARIABLE).

- (DEF ,var ,val)
locally binds ,var to ,val and evaluates remainder of current body in the context of that binding.

- (DEF (TUP ,var ... ) ,val)
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 ((,var ,val) ...) ,@body)
SEQ (SEQ ,@body)

SEQ abbreviation ≡ (SEQ ,@body).

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

AND
AND (AND ,form ,@forms)

UNLESS
UNLESS (UNLESS ,test ,@body)

COND
COND (COND (,test ,@body) ...)

4 Program Control

Goo provides a variety of program control constructs including function calls, conditional execution, and nonlocal control flow.

SEQ
SEQ (SEQ ,@forms)

returns false

( ... ) ( ,@body )
SEQ abbreviation ≡ (SEQ ,@body).

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

AND
AND (AND ,form ,@forms)

UNLESS
UNLESS (UNLESS ,test ,@body)

COND
COND (COND (,test ,@body) ...)

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 ] ,@body

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 =>'s.

GBP
GBP (GBP ,place ,expr)

evaluates (SEQ ,@forms) of first form 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 =>'s.

SWAPF
SWAPF (SWAPF ,x ,y)

evaluates (SEQ ,@forms) of first form 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 =>'s.

ROTF
ROTF (ROTF ,x ,y ,@places)

evaluates (SEQ ,@forms) of first form 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 =>'s.

call
(,f ,@args)

evaluates $f$ and then $\texttt{\_\_\_\_\_\_}$ in left to right order and then calls $f$ with the evaluated arguments.

$\texttt{\_\_\_\_\_\_}$ defines a recursive loop (cf., Dylan’s \texttt{ITERATE} or Scheme’s \texttt{(LET \texttt{\_\_\_\_\_\_ ...)}).

$\texttt{\_\_\_\_\_\_}$ evaluates $\texttt{\_\_\_\_\_\_}$ with an exit function of a single parameter, $x$, bound to $\texttt{\_\_\_\_\_\_}$ that if called, will cause $\texttt{\_\_\_\_\_\_}$ to return the value of $x$ (cf., Dylan’s \texttt{BLOCK/RETURN}). It is illegal to call the exit function after the execution of the creating $\texttt{\_\_\_\_\_\_}$ form (i.e., no upward continuations).

$\texttt{\_\_\_\_\_\_}$ ensures that $\texttt{\_\_\_\_\_\_}$ is evaluated whether or not an $\texttt{\_\_\_\_\_\_}$ upwards exit is taken during the dynamic-extent of $\texttt{\_\_\_\_\_\_}$ (cf. Dylan’s \texttt{BLOCK/CLEANUP} form and CL’s \texttt{UNWIND-PROTECT}). The result of a $\texttt{\_\_\_\_\_\_}$ form is the result of evaluating its protected form.

$\texttt{\_\_\_\_\_\_}$ ensures that $\texttt{\_\_\_\_\_\_}$ is evaluated whether or not an $\texttt{\_\_\_\_\_\_}$ upwards exit is taken during the dynamic-extent of $\texttt{\_\_\_\_\_\_}$ (cf. Dylan’s \texttt{BLOCK/CLEANUP} form and CL’s \texttt{UNWIND-PROTECT}). The result of a $\texttt{\_\_\_\_\_\_}$ form is the result of evaluating its protected form.

5 Types, Classes and Properties

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

$\texttt{GOO}$ supports the following types in order of specificity (with the exact ordering defined in Appendix B):

- 
  - 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:

$\texttt{\texttt{\_\_\_\_\_\_}}$ returns true if $x$ is a subtype of $y$.

$\texttt{\texttt{\_\_\_\_\_\_}}$ returns new $\texttt{\_\_\_\_\_\_}$.

5.1 Singletons

Singleton types match exactly one value using $\Rightarrow\Rightarrow$. Singletons are the most specific types.

$\texttt{\texttt{\_\_\_\_\_\_}}$ returns singleton constrained to $x$.

$\texttt{\texttt{\_\_\_\_\_\_}}$ 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> (\texttt{\_\_\_\_\_\_})

$\texttt{\texttt{\_\_\_\_\_\_}}$ returns subclass type constrained to subclasses of $x$.

<type-class> (\texttt{\_\_\_\_\_\_})

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> (\texttt{\_\_\_\_\_\_})

<product> (\texttt{\_\_\_\_\_\_})

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

<product> (\texttt{\_\_\_\_\_\_})

<product> (\texttt{\_\_\_\_\_\_})

<product> (\texttt{\_\_\_\_\_\_})

<product> (\texttt{\_\_\_\_\_\_})

<product> (\texttt{\_\_\_\_\_\_})

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> (\texttt{\_\_\_\_\_\_})

<product> (\texttt{\_\_\_\_\_\_})

<product> (\texttt{\_\_\_\_\_\_})

<product> (\texttt{\_\_\_\_\_\_})

<product> (\texttt{\_\_\_\_\_\_})

5.5 Classes

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

<class> (\texttt{\_\_\_\_\_\_})

<class> (\texttt{\_\_\_\_\_\_})

<class> (\texttt{\_\_\_\_\_\_})

<class> (\texttt{\_\_\_\_\_\_})

<class> (\texttt{\_\_\_\_\_\_})

DC (\texttt{\_\_\_\_\_\_})
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 \( \langle \text{prop} \rangle \). 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.

```plaintext
prop
(prop-owner \langle \text{prop} \rangle => \langle \text{any} \rangle)
```

Class on which property was directly defined.

```plaintext
prop-getter \langle \text{prop} \rangle => \langle \text{gen} \rangle
```

Reader accessor generic.

```plaintext
prop-setter \langle \text{prop} \rangle => \langle \text{gen} \rangle
```

Writer accessor generic.

```plaintext
prop-type \langle \text{prop} \rangle => \langle \text{type} \rangle
```

Type constraining property value.

```plaintext
prop-init \langle \text{prop} \rangle => \langle \text{fun} \rangle
```

Lazy initialization function.

```plaintext
find-getter \langle [ \text{class} \text{getter} \langle \text{gen} \rangle => \langle \text{met} \rangle] \rangle
```

Finds getter method defined on given class.

```plaintext
find-setter \langle [ \text{class} \text{setter} \langle \text{gen} \rangle => \langle \text{met} \rangle] \rangle
```

Finds setter method defined on given class.

```plaintext
find-bound? \langle \langle x \| \langle \text{gen} \rangle => \langle \text{clo} \rangle \rangle \rangle => \langle \text{bool} \rangle
```

Returns true if property with getter \( g \) is bound in instance \( x \).

```plaintext
add-prop \langle \langle \text{owner getter} \langle \text{gen} \rangle \text{setter} \langle \text{gen} \rangle \text{type} \langle \text{type} \rangle \text{init} \langle \text{fun} \rangle \rangle \rangle
```

Where \( \text{init} \) is a one parameter function that returns the initial value for the prop and gets called lazily with the new instance as the argument.

```plaintext
\langle \langle \langle \text{dp} \langle \text{name} \langle \text{oname} \rangle, \text{owner} => \langle \text{type} \rangle \rangle \langle \text{init} \rangle \rangle \rangle => \langle \text{any} \rangle
```

Add's an immutable property \( \langle \text{name} \rangle \) \( \langle \text{oname} \rangle \) \( \langle \text{owner} \rangle \) \( \langle \text{type} \rangle \) \( \langle \text{init} \rangle \) to \( \langle \text{dp} \rangle \), lazily and optionally initial value \( \langle \text{init} \rangle \). The initial value function is evaluated lazily when prop's value is first requested.

```plaintext
\langle \langle \langle \text{dp!} \langle \text{name} \langle \text{oname} \rangle, \text{owner} => \langle \text{type} \rangle \rangle \langle \text{init} \rangle \rangle \rangle
```

Same as \( \text{dp} \) but mutable with setter named \( \langle \text{name} \rangle \# "\langle \text{setter} \rangle" \).

6 Functions

All operations in \( \text{GOO} \) 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 \( \text{GOO} \).

All \( \text{GOO} \) functions are objects, instances of \( \langle \text{fun} \rangle \). Generic functions are instances of \( \langle \text{gen}\rangle \) and methods are instances of \( \langle \text{met}\rangle \).
(LET ((,name #f) ...)
  (SET ,name (fun ,sig ,@body)) ...)

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

(setf ,name (fun ,sig ,@body))

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 non-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.

 defendants a method.

preserves the original order of methods.

returns x's methods.

adds method y to generic x.

returns both the list of sorted applicable methods and any ambiguous methods when generic x is called with arguments.

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

6.2 Methods

Methods are GOO's code objects. Methods can optionally be added to generics.

evaluates ,form in the midst of a QUASIQUOTE expression, abbreviated "\"."

evaluates ,form in the midst of a QUASIQUOTE expression and splices it in, abbreviated "\"\".

evaluates ,val corresponding to first ,pat matching ,exp. The pattern is much the same as QUASIQUOTE and can contain either QUASIQUOTE'd variables or QUASIQUOTE-SPlicing-UNQUOTE variables. For example,

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

defines the macro in GOO.

where

7 Macros

Macros provide a facility for extending the base syntax of 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.

 hatırlar the return value of \(eval-when (:compile-toplevel :execute)\) ...

in Common LISP .
8 Scalars

GOO provides a rich set of simple objects.

8.1 Any

All objects are derived from <any>.

&asymp; coerces y to an instance of x.

class-of returns concrete class of x.

== returns true iff x and y are computationally equivalent.

= returns true iff x and y are equal, where equality is user defined and defaults to ==.

not (x|<any> y|<any> => <log>)

≡ (not (= x y)).

to-str returns string representation of object.

8.2 Booleans

In 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.

#f

#t

if x #f x returns true if x is one of the ASCII upper or lowercase characters.

digit? returns true if x is one of the ten ASCII numeric characters.

lower? returns true if x is one of the ASCII lowercase characters.

upper? returns true if x is one of the ASCII uppercase characters.

to-digit converts ascii representation of digit to an integer one.

to-lower returns lowercase version of uppercase alphabetic characters otherwise returns x.

to-upper returns uppercase version of lowercase alphabetic characters otherwise returns x.

8.3 Magnitudes

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

returns if x is less than y.

returns the sum of its arguments.

returns the difference of its arguments.

returns the product of its arguments.

returns the quotient of its arguments.

8.4 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 GOO object data.

returns the object pointed to by x.

returns address of particular object.

8.5 Characters

GOO currently supports 8 bit ASCII characters.

alpha? returns true if x is one of the ASCII upper or lowercase characters.

digit? returns true if x is one of the ten ASCII numeric characters.

lower? returns true if x is one of the ASCII lowercase characters.

upper? returns true if x is one of the ASCII uppercase characters.

converts ascii representation of digit to an integer one.

returns lowercase version of uppercase alphabetic characters otherwise returns x.

returns uppercase version of lowercase alphabetic characters otherwise returns x.

8.6 Numbers
round \[ x \mapsto \text{closest integer to } x \]. If \( x \) is exactly between two integers then the implementation is free to return either integer.

\[
\text{round-to}(\langle x \rangle, n) \mapsto \text{closest flo } n \text{ digits precision.}
\]

\[
\text{floor}(\langle x \rangle) \mapsto \text{integer truncated towards negative infinity.}
\]

\[
\text{ceil}(\langle x \rangle) \mapsto \text{integer truncated towards positive infinity.}
\]

\[
\text{trunc}(\langle x \rangle) \mapsto \text{integer truncated towards zero.}
\]

\[
\text{mod}(\langle x \rangle, y) \mapsto \text{remainder after taking the floor of the quotient of } x \text{ and } y.
\]

\[
\text{div}(\langle x \rangle, y) \mapsto \text{trunc of the quotient of } x \text{ and } y.
\]

\[
\text{rem}(\langle x \rangle, y) \mapsto \text{remainder after dividing } x \text{ by } y.
\]

\[
\text{pow}(\langle x \rangle, e) \mapsto x \text{ raised to the } e \text{ power.}
\]

\[
\text{exp}(\langle x \rangle) \equiv (\text{pow } e x)
\]

\[
\sqrt{x} \mapsto \text{square root of } x.
\]

\[
\text{pos?}(\langle x \rangle) \equiv (> x 0)
\]

\[
\text{zero?}(\langle x \rangle) \equiv (= x 0)
\]

\[
\text{neg?}(\langle x \rangle) \equiv (< x 0)
\]

\[
\text{neg}(\langle x \rangle) \equiv (- 0 x)
\]

\[
\text{abs}(\langle x \rangle) \mapsto \text{absolute value of } x.
\]

\[
\text{bit?}(\langle x \rangle) \equiv \text{true iff } n \text{th bit is } 1.
\]

8.6.1 Integers

\(\text{GOO currently represents integers as 30 bit fixnums.}\)

\[
\text{even?}(\langle x \rangle) \equiv (\text{bit? } (\text{num-to-str-base } x 2))
\]

\[
\text{odd?}(\langle x \rangle) \equiv (\text{bit? } (\text{num-to-str-base } x 2))
\]

\[
\text{gcd}(\langle x \rangle, y) \equiv \text{greatest common denominator.}
\]

\[
\text{lcm}(\langle x \rangle, y) \equiv \text{least common multiple.}
\]

\[
\text{shl}(\langle x \rangle, n) \equiv \langle x \rangle \text{ bit shift left of } x.
\]

\[
\text{shr}(\langle x \rangle, n) \equiv \langle x \rangle \text{ bit shift right of } x.
\]

\[
\text{ushr}(\langle x \rangle, n) \equiv \langle x \rangle \text{ bit shift right of } x.
\]
8.6.2 Floats

GOO currently only supports single-precision floating point numbers.

```plaintext
8.6.2 Floats
GOO currently only supports single-precision floating point numbers.
```

9 Collections

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

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

9.1 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.

```plaintext
9.1 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.
```

1When optimization is in place, the ′ suffixed functions will be deprecated.
<table>
<thead>
<tr>
<th><strong>9.2 Enumerators</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>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: <code>enum</code>, <code>fin?</code>, <code>nxt</code>, and <code>now</code>. For efficiency, users might choose to override more methods such as <code>len</code>, <code>elt</code>, <code>elt-setter</code>, etc. Enumeration behavior is undefined if an enumerator is modified during enumeration.</td>
</tr>
</tbody>
</table>

```lisp
<enum> (x|<any>) C
enum (x|<any> => <enum>) G
returns initial enum for iterating over x.
fin? (x|<enum> => <log>) G
returns true iff no more elements exist from given enum x.
now (x|<enum> => <any>) G
returns current element given enum x.
own-key (x|<enum> => <any>) G
returns current key given enum x.
own-setter (x|x|<enum>) M
sets current element given enum x to v.
```

9.3 Packers

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

```lisp
(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:

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

```lisp
<packer> (any) C
packer-add (p|<packer> x => <packer>) G
returns a copy packer p augmented with element x.
packer-res (p|<packer> => <any>) G
returns result of packings over p.
packer (init add|<fun> res|<fun>) G
returns a simple packer that starts its value out with init, is augmented with add, and whose final value is computed with res.
```

9.4 Maps

Maps represent collections with explicit keys.

```lisp
<map> (<col>) C
<tab> (<map> <col!>) 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.

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

factor by which to grow capacity.
9.5 Sequences

Sequences are collections with nonnegative integer keys.

\[
\text{<seq>} \quad ((\text{<col>})
\]

Immutable sequence.

\[
\text{1st} \quad (x|\text{<seq>} \Rightarrow \text{<any>}) \quad \mathbb{G}
\]

1st \(x\) in \(x\) else returns false.

\[
\text{2nd} \quad (x|\text{<seq>} \Rightarrow \text{<any>}) \quad \mathbb{G}
\]

2nd \(x\) in \(x\) else returns false.

\[
\text{3rd} \quad (x|\text{<seq>} \Rightarrow \text{<any>}) \quad \mathbb{G}
\]

3rd \(x\) in \(x\) else returns false.

\[
\text{last} \quad (x|\text{<seq>} \Rightarrow \text{<any>}) \quad \mathbb{G}
\]

Last \(x\) in \(x\) else returns false.

\[
\text{pos} \quad (x|\text{<seq>} v|\text{<any>} \Rightarrow (??\text{<int>})) \quad \mathbb{G}
\]

Finds position of \(v\) in \(x\) else returns false.

\[
\text{finds} \quad (x|\text{<seq>} y|\text{<any>} \Rightarrow (??\text{<int>})) \quad \mathbb{G}
\]

Finds position of \(y\) in \(x\) else returns false.

\[
\text{add} \quad (x|\text{<seq>} y|\text{<any>} \Rightarrow \text{<seq>}) \quad \mathbb{M}
\]

Returns sequence with \(y\) added to the end of \(x\).

\[
\text{push} \quad (x|\text{<seq>} y|\text{<any>} \Rightarrow \text{<seq>}) \quad \mathbb{G}
\]

Returns sequence with \(y\) added to \(x\).

\[
\text{rev} \quad (x|\text{<seq>} \Rightarrow \text{<seq>}) \quad \mathbb{G}
\]

Returns reversed sequence.

\[
\text{cat} \quad (x|\text{<seq>} \Rightarrow \text{<seq>}) \quad \mathbb{G}
\]

Returns concatenated sequences.

\[
\text{sub} \quad (x|\text{<seq>} \Rightarrow \text{<seq>}) \quad \mathbb{G}
\]

Returns subsequence of \(x\) from \(\text{from}\) to \(\text{below}\).

\[
\text{split} \quad (x|\text{<seq>} \Rightarrow \text{<seq>}) \quad \mathbb{G}
\]

Returns sequence of subsequences of \(x\) separated by \(\text{sep}\).

\[
\text{join} \quad (x|\text{<seq>} \Rightarrow \text{<seq>}) \quad \mathbb{G}
\]

Returns sequence composed of sequences in \(x\) joined with \(\text{sep}\).

9.5.1 Mutable Sequences

\[
\text{<seq!>} \quad ((\text{<col!>})
\]

Mutable sequence.

\[
\text{rev!} \quad (x|\text{<seq!>} \Rightarrow \text{<seq!>}) \quad \mathbb{G}
\]

Returns destructively reversed sequence.

\[
\text{cat!} \quad (x|\text{<seq!>} \Rightarrow \text{<seq!>}) \quad \mathbb{G}
\]

Returns destructively concatenated sequences.

\[
\text{add!} \quad (x|\text{<seq!>} y|\text{<any>} \Rightarrow \text{<seq!>}) \quad \mathbb{G}
\]

Returns collection with \(y\) added to the end of \(x\).

\[
\text{push!} \quad (x|\text{<seq!>} y|\text{<any>} \Rightarrow \text{<seq!>}) \quad \mathbb{G}
\]

Returns collection with \(y\) added to the front of \(x\).

\[
\text{pop!} \quad (x|\text{<seq!>} \Rightarrow \text{<seq!>}) \quad \mathbb{G}
\]

Pops element from front of sequence.

\[
\text{PUSHF} \quad (\text{PUSHF}, \text{place}, \text{val}) \quad \mathbb{S}
\]

Pushes \(\text{val}\) onto the sequence stored in \(\text{place}\), updates \(\text{place}\) to contain the new sequence, and returns the new sequence.

\[
\text{POPF} \quad (\text{POPF}, \text{place}) \quad \mathbb{S}
\]

Pops a value from the sequence stored in \(\text{place}\), replaces the sequence with an updated sequence, and returns the value.

\[
\text{ins!} \quad (x|\text{<seq!>} v|\text{<any>} i|\text{<int>} \Rightarrow \text{<seq!>}) \quad \mathbb{G}
\]

Inserts \(v\) before \(i\) in \(x\).
replaces subsequence in range between \texttt{from} and \texttt{below} of \texttt{dst} with contents of \texttt{src}. Provides insertion, deletion, and replacement operations rolled into one.

\text{sub*-setter} (\texttt{dst} <\texttt{seq}!> \texttt{src} <\texttt{seq}> \texttt{from} <\texttt{int}>)

\texttt{G} \equiv \text{sub-setter} \texttt{dst} \texttt{src} \texttt{from} (\text{len} \texttt{dst})

\text{del-vals!} (\texttt{x} <\texttt{seq}!> \texttt{v} <\texttt{any}> \Rightarrow <\texttt{seq}!>)

\texttt{G} \equiv \text{del-vals} \texttt{x} \texttt{v} \Rightarrow <\texttt{seq}!>

\text{del-dups!} (\texttt{x} <\texttt{seq}!> \Rightarrow <\texttt{seq}!>)

\texttt{G} \equiv \text{del-dups} \texttt{x} \Rightarrow <\texttt{seq}!>

\text{sort-by!} (\texttt{s} <\texttt{seq}> \texttt{f} <\texttt{fun}> \Rightarrow <\texttt{seq}>)

\texttt{G} \equiv \text{sort-by!} \texttt{s} \texttt{f} \Rightarrow <\texttt{seq}>

\text{sort!} (\texttt{s} <\texttt{seq}> \Rightarrow <\texttt{seq}>)

\texttt{G} \equiv \text{sort} \texttt{s}

\text{9.5.2 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 \texttt{GOO}.

\texttt{<list>} <\texttt{seq}!> \texttt{G} \equiv \text{list}

\texttt{<list>} <\texttt{seq}!> \texttt{P} \equiv \text{head} \texttt{x} \Rightarrow <\texttt{any}>

\texttt{<list>} <\texttt{seq}!> \texttt{P} \equiv \text{tail} \texttt{x} \Rightarrow <\texttt{list}>

\texttt{<list>} <\texttt{seq}!> \texttt{G} \equiv \text{lst} <\texttt{elts} ... \Rightarrow <\texttt{lst}>

\texttt{returns list of arguments.}

\texttt{<list>} <\texttt{seq}!> \texttt{G} \equiv \text{list}\ast <\texttt{elts} ... \Rightarrow <\texttt{list}>

\texttt{returns list of arguments with last argument tacked onto end.}

\texttt{nil} <\texttt{list}>

\texttt{G} \equiv \text{nil} <\texttt{list}>

\texttt{pair} \texttt{x} <\texttt{any}> \texttt{y} <\texttt{list}> <\texttt{lst}>

\texttt{G} \equiv \text{pair} \texttt{x} \texttt{y} <\texttt{list}>

\texttt{9.5.3 Zips}

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

\texttt{<zip>} <\texttt{seq}!> \texttt{G} \equiv \text{zip} <\texttt{cs} ... <\texttt{seq}!> \Rightarrow <\texttt{zip}>

\texttt{returns a zip over sequences cs.}

\texttt{unzip} <\texttt{z} <\texttt{zip} \Rightarrow <\texttt{tup}>

\texttt{G} \equiv \text{unzip} <\texttt{z} <\texttt{zip} \Rightarrow <\texttt{tup}>

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

\texttt{tup} <\texttt{elts} ... \Rightarrow <\texttt{tup}>

\texttt{G} \equiv \text{tup} <\texttt{elts} ... \Rightarrow <\texttt{tup}>

\texttt{9.5.4 Flat Sequences}

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

\texttt{<flat>} <\texttt{seq}!> \texttt{G} \equiv \text{flat}

\texttt{<flat-enum>} <\texttt{enum}>

\texttt{G} \equiv \text{flat-enum} <\texttt{enum}>

\texttt{<tup>} <\texttt{seq}!> \texttt{G} \equiv \text{tup}

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

\texttt{tup} <\texttt{elts} ... \Rightarrow <\texttt{tup}>

\texttt{G} \equiv \text{tup} <\texttt{elts} ... \Rightarrow <\texttt{tup}>

\texttt{9.6 Lazy Series’}

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

\texttt{<range>} <\texttt{seq}!> \texttt{G} \equiv \text{range} \texttt{from} \texttt{to} \texttt{test} \texttt{lim} \texttt{by} \texttt{range}

\texttt{G} \equiv \text{range} \texttt{from} \texttt{lim} \texttt{test} \texttt{by} <\texttt{range}>

\texttt{<step>} <\texttt{seq}!> \texttt{G} \equiv \text{step} <\texttt{seq}!> \texttt{G} \equiv \text{step}

\texttt{Cycles provide a mechanism to create infinite sequences repeating a certain sequence over and over again.}

\texttt{cycle} <\texttt{x} ... \Rightarrow <\texttt{cycle}>

\texttt{G} \equiv \text{cycle} <\texttt{x} ... \Rightarrow <\texttt{cycle}>

\texttt{Strings}

\texttt{GOO currently implements ASCII strings.}

\texttt{<str>} <\texttt{seq}!> <\texttt{mag}!> <\texttt{seq}!> \texttt{G} \equiv \text{str} <\texttt{elts} ... \Rightarrow <\texttt{str}>

\texttt{G} \equiv \text{str} <\texttt{elts} ... \Rightarrow <\texttt{str}>

\texttt{9.6 Lazy Series’}
10 Symbols

Symbols are uniquified (aka interned) strings.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sym</td>
<td>A symbol table class.</td>
</tr>
<tr>
<td>sym-tab</td>
<td>Returns a symbol formed by concatenating the string representations of elts.</td>
</tr>
<tr>
<td>cat-sym</td>
<td>Coerces a string to a symbol.</td>
</tr>
<tr>
<td>condition-name</td>
<td>Returns a system specific unique symbol.</td>
</tr>
<tr>
<td>error</td>
<td>Signals an error.</td>
</tr>
<tr>
<td>simple-error</td>
<td>Signals a simple error.</td>
</tr>
<tr>
<td>serious-condition</td>
<td>A condition consisting of a msg message and arguments.</td>
</tr>
<tr>
<td>condition</td>
<td>Returns msg string.</td>
</tr>
<tr>
<td>condition-arguments</td>
<td>Returns msg string arguments.</td>
</tr>
<tr>
<td>arithmetic-error</td>
<td>A condition that indicates something is invalid about the program.</td>
</tr>
<tr>
<td>stack-overflow-error</td>
<td>A condition that can not be safely ignored.</td>
</tr>
<tr>
<td>keyboard-interrupt</td>
<td>A condition that can not be safely ignored.</td>
</tr>
<tr>
<td>internal-error</td>
<td>A system fault was detected.</td>
</tr>
</tbody>
</table>

11 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.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>A condition consisting of a msg message and arguments.</td>
</tr>
<tr>
<td>error</td>
<td>Signals an error.</td>
</tr>
<tr>
<td>simple-error</td>
<td>Signals a simple error.</td>
</tr>
<tr>
<td>serious-condition</td>
<td>An error that consists of a msg message and arguments.</td>
</tr>
<tr>
<td>condition-type-name</td>
<td>The handler function should take two arguments: the 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 return 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 GOO does not unwind the stack before calling handlers!</td>
</tr>
</tbody>
</table>

where

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handler</td>
<td>Installs handler as a condition handler for the duration of TRY,try-options,handler,try-option-list with zero or more of the following options:</td>
</tr>
<tr>
<td>try-option</td>
<td>(TYPE ,expr) =&gt; An expression returning the type of condition to handle.</td>
</tr>
<tr>
<td>try-option</td>
<td>(TEST ,body) =&gt; 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.</td>
</tr>
<tr>
<td>try-option</td>
<td>(DESCRIPTION ,message ,@arguments) =&gt; A human-readable description of this handler. Used by the debugger.</td>
</tr>
<tr>
<td>try-option</td>
<td>build-condition-interactively =&gt; 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.</td>
</tr>
<tr>
<td>try-option</td>
<td>arg =&gt; A condition, msg, or a handler function.</td>
</tr>
<tr>
<td>try-option</td>
<td>TRY =&gt; (TRY ,try-option-list ,body)</td>
</tr>
</tbody>
</table>

11.1 Conditions Hierarchy

GOO has a builtin hierarchy of conditions.
<table>
<thead>
<tr>
<th>Error Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion failure</td>
<td>An assertion failure occurred.</td>
</tr>
<tr>
<td>Unbound error</td>
<td>An binding or property was found to be unbound.</td>
</tr>
<tr>
<td>Unbound variable error</td>
<td>A binding was found to be unbound.</td>
</tr>
<tr>
<td>Property error</td>
<td>Property accessor if available.</td>
</tr>
<tr>
<td>Property error-owner</td>
<td>Property owner if available.</td>
</tr>
<tr>
<td>Unbound property error</td>
<td>Unbound property was discovered.</td>
</tr>
<tr>
<td>Property type-error</td>
<td>Attempt was made to store an invalid object in a property.</td>
</tr>
<tr>
<td>Property not-found error</td>
<td>Attempt was made to find a missing property, most likely during a call to new.</td>
</tr>
<tr>
<td>As error</td>
<td>No as method was found.</td>
</tr>
<tr>
<td>Range error</td>
<td>A key lookup on a collection failed.</td>
</tr>
<tr>
<td>Type error</td>
<td>A type check has failed.</td>
</tr>
<tr>
<td>Value error</td>
<td>Returns value on which type check failed.</td>
</tr>
<tr>
<td>Type error-type</td>
<td>Returns type on which type check failed.</td>
</tr>
<tr>
<td>Call error</td>
<td>A function call failed.</td>
</tr>
<tr>
<td>Call error-function</td>
<td>Returns function on which call failed.</td>
</tr>
<tr>
<td>Call error-arguments</td>
<td>Returns arguments on which call failed.</td>
</tr>
<tr>
<td>Arity error</td>
<td>Wrong number of arguments supplied in function call.</td>
</tr>
<tr>
<td>Narity error</td>
<td>Too few arguments supplied in nary function call.</td>
</tr>
<tr>
<td>Unknown arguments error</td>
<td>Too few arguments supplied in nary function call.</td>
</tr>
<tr>
<td>Argument type-error</td>
<td>Invalid argument used function call.</td>
</tr>
<tr>
<td>Return type-error</td>
<td>Invalid result returned from function call.</td>
</tr>
<tr>
<td>Ambiguous method error</td>
<td>Unable to sort applicable methods.</td>
</tr>
<tr>
<td>No applicable methods error</td>
<td>No methods were applicable.</td>
</tr>
<tr>
<td>No next-methods error</td>
<td>No next-methods were found during a sup or app-sup call.</td>
</tr>
<tr>
<td>Incongruent method error</td>
<td>A method is incongruent with a generic.</td>
</tr>
<tr>
<td>Incongruent method error-generic</td>
<td>Incongruent method error-method</td>
</tr>
<tr>
<td>Incongruent method error-method</td>
<td>Incongruent method error-method</td>
</tr>
<tr>
<td>Cpl error</td>
<td>Unable to find a consistent class precedence list.</td>
</tr>
<tr>
<td>Io error</td>
<td>An input/output operation failure.</td>
</tr>
<tr>
<td>File opening error</td>
<td>Unable to open a file.</td>
</tr>
<tr>
<td>Compiler error</td>
<td>A macro expansion failure.</td>
</tr>
<tr>
<td>Syntax error</td>
<td>A macro expansion failure.</td>
</tr>
<tr>
<td>Ast error</td>
<td>An ast conversion failure.</td>
</tr>
<tr>
<td>Namespace error</td>
<td>A namespace form failure.</td>
</tr>
</tbody>
</table>

13
12 Input / Output

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

12.1 Ports

Ports represent character-oriented input/output devices.

\[
\begin{align*}
\text{port} & \quad \text{(read)} \\
\text{open} & \quad \text{[x|<port> \ x|<str> => <port>]} \\
\text{close} & \quad \text{[x|<port>]} \\
\text{noop default.} \\
\text{WITH-PORT} & \quad \text{(WITH-PORT ,name ,port) ,@body} \\
\text{port-contents} & \quad \text{[x|<str-port> => <str>]} \\
\text{port-index} & \quad \text{[x|<str-port> => <int>]} \\
\text{PORT-TO-STR} & \quad \text{(PORT-TO-STR ,name ,@body)}
\end{align*}
\]

12.1.1 File Ports

File ports are ports which map to files.

\[
\begin{align*}
\text{file-port} & \quad \text{(read)} \\
\text{open} & \quad \text{[t|<file-port> \ name|<str> => <file-port>]} \\
\text{close} & \quad \text{[t|<file-port>]} \\
\text{noop default.} \\
\text{WITH-PORT} & \quad \text{(WITH-PORT ,name ,port) ,@body} \\
\text{port-contents} & \quad \text{[x|<str-port> => <str>]} \\
\text{port-index} & \quad \text{[x|<str-port> => <int>]} \\
\text{PORT-TO-STR} & \quad \text{(PORT-TO-STR ,name ,@body)}
\end{align*}
\]

12.1.2 String Ports

String ports provide port interface mapped onto strings.

\[
\begin{align*}
\text{str-port} & \quad \text{(read)} \\
\text{open} & \quad \text{[t|<str-port> \ dat|<str> => <str-port>]} \\
\text{noop default.} \\
\text{WITH-PORT} & \quad \text{(WITH-PORT ,name ,port) ,@body} \\
\end{align*}
\]

12.2 Formatted I/O

GOO provides convenient s-expression reading/writing facilities.

\[
\begin{align*}
\text{read} & \quad \text{[x|<in-port> => <any>]} \\
\text{write} & \quad \text{[x|<out-port> \ y|<any>]} \\
\text{writeln} & \quad \text{[x|<out-port> \ y|<any>]} \\
\text{emit} & \quad \text{[x|<out-port> \ y|<any>]} \\
\text{msg} & \quad \text{[x|<out-port> \ message|<seq> args|...]} \\
\end{align*}
\]
formatted output using special commands embedded in message. supported commands are:

- `> ` → (write x arg)
- `>` → (display x arg)
- `: ` → (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).

13 System

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

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

13.1 Files and Directories

A preliminary set of file and directory facilities are provided.

- `filename-time` → ([filename|<str> => <flo>])
  - return the last modification time of a file in seconds (relative to the n GOOepoch) as a floating point number.
- `filename-exists?` → ([filename|<str> => <log>])
  - return true if and only if a file (or a directory, etc.) exists with the given name.
- `filename-type` → ([filename|<str> => <sym>])
  - return the file, directory or some other symbol, depending on the type of the file.
- `create-directory` → ([filename|<str> => <sym>])
  - create a directory with the given name. The parent directory must already exist, and must contain no item with the given name.
- `find-parent-directory` → ([name|<str> => <str>])
  - find the parent directory of the current filename.
- `probe-directory` → ([name|<str> => <str>])
  - make sure that the named directory exists.

13.2 Pathnames

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

- `pathname-to-components` → ([pathname|<str> => <lst>])
  - 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>])
  - reassemble components created by the above function.
- `label-components` → ([components|<lst> => <lst>])
  - get any leading directory label.
- `hierarchical-components` → ([components|<lst> => <lst>])
  - get rid of any leading directory label, etc.
- `components-last` → ([components|<lst> => <any>])
  - return the last item in a list of components.
- `components-basename` → ([components|<lst> => <str>])
  - 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>])
  - calculate the parent directory of a pathname.
- `<pathname-error>` → (<simple-error>)
  - indicates an error while processing a pathname.
SPAWN (SPAWN ,@body)

-thread-yield ()

Surrenders processor to another thread.

-thread-join (thread)<thread>

Causes current thread to wait for the termination of thread.

-thread-current (-><thread>)

-all-threads (-><up>)

Sleep (secs)<flo>

Pauses current thread for secs seconds.

-clock>(<any>)

Represents a mutex.

-lock-name (lock)<lock> => (t? <sym>)

Locks lock waiting if necessary.

-lock-lock (lock)<lock>

Obtain exclusive access to lock waiting if necessary.

-lock-unlock (lock)<lock>

Free up exclusive access to lock potentially allowing another thread access.

-lock-unlock-lock (lock)<lock>

WITH-LOCK (WITH-LOCK ,lock ,@body)

Obtains lock on lock waiting before executing body.

-event-name (event)<event>

With an initial value (var-name ,var) with an initial value.

-lock-event (event<event> lock|<lock>) secs|<flo> => <log>

Unlocks lock and places thread in waiting state to be resumed when event is signaled or broadcasted upon which time lock is reacquired and thread resumed.

-event-wait-timed (event<event> lock|<lock> secs|<flo> => <log>)

Unlocks lock and places thread in waiting state to be resumed when event is signaled or broadcasted or timeout secs is reached upon which time lock is reacquired and thread resumed.

-DDV (DDV ,var ,form)

Defines a thread local variable named (var-name ,var) with an initial value ,form.

-LET ((,old-var ,var) ...) ,@body)

Extracts a value from a variable.

<pipe> (pipe)<pipe> x => <pipe>

Represents a synchronized FIFO queue allowing multiple readers and writers in separate threads.

Adds x to pipe.

<seq!> (pipe)<pipe> => <any>

Removes and returns element from pipe or waits for one to be available.

15 C Interface

A simple mechanism (available through the g2c module) is provided to inline C code directly into GOO, escaping back into GOO when necessary, and relying on C for its type system instead of having to mirror it in GOO. On the downside, there is no static checking, and errors can occur during C compilation.

-#* ... #*

Allows for the easy specification of strings (especially C code snippets) with special character escaping turned off.

-C-MENT (C-MENT ,c-snippet [,goo-form ,c-snippet]*)

Specifies a C statement formed as the concatenation of c-snippets and GOO forms 2. The GOO forms are evaluated at runtime as embedded C expressions in the lexical context of the c-ment expression. If specified at top level, then the c-ment form will be evaluated prior to the evaluation of non-c-ment top level forms.

-C-EXP (C-EXP ,c-snippet [,goo-form ,c-snippet]*)

Same as c-ment, but specifies a value producing C expression instead.

-to-c (x)

user extensible protocol for converting a GOO object to a C value. Methods are provided for <int> <flo> <str> <chr> <log> and <loc>.

-#( ... )

Inlined C Statement

Short-hand for c-ment allowing GOO forms to be specified with a $ prefix and the conversion of GOO objects into C values specified with an $ prefix, with $x ≡ $<to-c x>. For example,

$(df f (x) #

...)

prints out the GOO value. A callback can be defined at top level as follows:

#( int gl_idle(int x) { $gl_idle; } )

C headers can be included similarly:

#( #include "GL.h" )

C expressions can be nested within embedded GOO expressions as follows:

#( int gl_idle(int x) {$gl_idle #e1 x} )

in order to access lexical apparent C variables etc.

-#e( ... )

Inlined C Expression

Similar shorthand for c-exp, also allowing a single character code c for specifying the conversion of C values back to GOO objects. The valid codes are: for <int>, for <flo>, for <str>, for <chr>, for <log>, for <loc>, s for none. For example,

(d. $gl-line-loop #e1 ( GLLINELOOP ))

defines a top-level OpenGL constant.

-USE/LIBRARY (USE/LIBRARY ,name)

adds ,name to list of libraries to be linked against.

-USE/INCLUDE (USE/INCLUDE ,name)

adds ,name to include directory search path.

16 Compiler

GOO’s compiler, g2c, compiles 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.
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 src directory.

During start up, $GOO will load two patch files: $({$GOO_ROOT}/init.goo $({HOME})/:.goo/init.goo

You can customize your $GOO by adding forms to these files.

19 Usage

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

:quit exits from $GOO.

:eval invokes a recursive read-eval-print loop.

:g2c-eval to change to dynamic compilation evaluation.

:ast-eval to change to ast evaluation.

:GOO_EVALMODE <str> environment variable for setting $GOO's evaluation mode. Valid settings are "ast" or "g2c".

:top <name> changes to module .name.

:mod:name = accesses an unexported binding from another module.

19.1 Development

To batch compile $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)

19.2 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:

K up goes up one level.

K top goes to top level.

17 Top Level

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

load <filename|[str] modname|<sym> => <any>> returns the result of evaluating the result of reading file named <filename> into module modname.

eval <any> modname|<sym> => <any> returns the result of evaluating .

top <modname|<sym>> runs top-level read-eval-print loop which reads from in and writes to out.

save-image <filename|<str>> saves an image of the current $GOO process to a file named <filename>.

refers respectively to last, second to last, and third to last values returned in top-level listener.

18 Installation

Unpack a $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 $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. $GOO requires Boehm's GC to be installed as a shared library on linux or under the source dir.

You can customize your $GOO_ROOT by adding forms to these files.
19.3 Emacs Support

A rudimentary emacs-based development system is provided.

19.3.1 Emacs Mode

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

```lisp
(load 'goo-mode "Major mode for editing Goo source.")
(setq auto-mode-alist
  (cons (".*\." . 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.

19.3.2 Emacs Shell

Put emacs/goo-shell.el in your emacs lisp directory. Add the following to your .emacs:

```lisp
(load 'run-goo "Run an inferior Goo process.")
(setq auto-mode-alist
  (cons (".*\." . goo-mode) auto-mode-alist))
(setq goo-program-name "~/ai/jrb/goo/goo")
```

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

Useful command / key-bindings are:

- C-x goo-send-definition
- C-c C-w goo-send-definition
- C-c M-x goo-send-definition-and-go
- C-z 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.

19.3.3 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

20 Caveats

This is the first release of GOO. 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 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@googoogaga.org. I will fix your bugs asap. The GOO website www.googoogaga.org will have papers, releases, FAQs, etc.

21 Future

The plan is for GOO to evolve in a number of dimensions. First of all, 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, GOO will support a more complete loopless programming protocol inspired by Waters’ series [5].

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.

22 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 co-taught 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 LLLI in the Fall of 2001. Many stimulating conversations on the follow on LLI discussion list inspired 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 \[2\] backend, and has provided useful feedback on GOO’s design. I also wish to thank Boehm, Demers, and Weiser for writing the conservative GC upon which this initial version of 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


A Class Precedence List

This section defines the algorithm for computing a class’s linearized ancestors from its parents, its parent’s parents, etc. GOO uses the C3 class linearization rule [1]. The following is the GOO implementation of this algorithm:

```lisp
(dm class-ordered-ancestors (c|<class> => <lst>)
  (def parents (class-parents c))
  (rep merge-lists
    ((partial-cpl|<class> (lst c))
     (remaining-lists|<lst>)
     (add (map class-ancestors parents) parents)))
  (rev! partial-cpl)
  (loc ((candidate (c)
    (loc (tail? (l|<lst>) (mem? (tail l) c)))
    (and (not (any? tail? remaining-lists) c))
    (candidate-at-head (l|<class>)(candidate (head l))))
    (def next (any? candidate-at-head-remaining-lists))
    (if next
      (loc ((del-next (l|<list>)
        (if (= (head l) next) (tail l) l))
        (merge-lists
          (pair next partial-cpl)
          (map del-next remaining-lists)))
        (error "inconsistent precedence graph"))))
```

B Subtyping Rules

This section defines the subtyping rules for GOO in terms of subtype methods.

```lisp
(dm subtype? (t1|<union> t2|<type> => <log>)
  (all? (op subtype? _ t2) (type-elts t1)))
(dm subtype? (t1|<type> t2|<union> => <log>)
  (any? (op subtype? t1 _) (type-elts t2)))
(dm subtype? (t1|<union> t2|<class> => <log>)
  (subclass? t1 t2))
(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? (type-class t1) (type-class t2)))
(dm subtype? (t1|<union> t2|<subclass> => <log>)
  (all? (op subtype? _ t2) (type-elts t1)))
(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|<product> t2|<type> => <log>) #f)
(dm subtype? (t1|<type> t2|<product> => <log>) #f)
(dm subtype? (t1|<product> t2|<product> => <log>)
  (and (== (len (type-elts t1)) (len (type-elts t2)))
    (all? (zipped subtype?) (zip (type-elts t1) (type-elts t2))))
(dm subtype? (t1|<product> t2|<class> => <log>)
  (subclass? (type-class t1) (type-class t2)))
(dm subtype? (t1|<class> t2|<subclass> => <log>)
  (and (isa? (type-object t1) (class)
    (subclass? (type-object t1) (class)
      (subclass? (type-object t1) (class)
        (subclass? (type-object t1) (class)
          (is? (type-object t1) (class)
            (subclass? (type-object t1) (class)
              (subclass? (type-object t1) (class)
                (subclass? (type-object t1) (class)
                  (isa? (type-object t1) (class)
                    (subclass? (type-object t1) (class)
                      (subclass? (type-object t1) (class)
                        (subclass? (type-object t1) (class)
                          (isa? (type-object t1) (class)
                            (subclass? (type-object t1) (class)
                              (subclass? (type-object t1) (class)
                                (isa? (type-object t1) (class)
                                  (subclass? (type-object t1) (class)
                                    (isa? (type-object t1) (class)
                                      (subclass? (type-object t1) (class)
                                        (isa? (type-object t1) (class)
                                          (subclass? (type-object t1) (class)
                                            (isa? (type-object t1) (class)
                                              (isa? (type-object t1) (class)
                                                (isa? (type-object t1) (class)
                                                  (isa? (type-object t1) (class)
                                                    (isa? (type-object t1) (class)))))))))))))))))))))))))))))))
```

19