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, lisp-syntaxed 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>Kind</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>N</td>
<td>Notation</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>Lexical</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>Syntax</td>
</tr>
<tr>
<td>G</td>
<td>G</td>
<td>Generic</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
<td>Method</td>
</tr>
<tr>
<td>P</td>
<td>P</td>
<td>Function</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>Class</td>
</tr>
<tr>
<td>P</td>
<td>P</td>
<td>Property</td>
</tr>
<tr>
<td>I</td>
<td>I</td>
<td>Instance</td>
</tr>
<tr>
<td>K</td>
<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:

```
Line comment                N
Nested comment              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 's' 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               X
Global variable             X
Predicate                   X
Destructive function        X
Setter                      X
```

2 Expressions

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

can occur in the module use heterarchy. Furthermore, no

can be excluded or renamed imported bindings. Moreover, no cy-

noded 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 ex-

ported by other modules, but currently there is no way to selec-

tively exclude or rename imported bindings. Furthermore, no cy-

cles can occur in the module use heterarchy.

GOO defines a number of identifiers as the names of special forms, which if

seen in function call position cause special form specific evaluation.

3.1 Namespaces

3.1.1 Names

3.1.2 Constants

3.1.3 Variables

3.1.4 Initialization

3.1.5 Binding

3.1.6 Importing

4 Program Control

GOO provides a variety of program control constructs including

function calls, conditional execution, and nonlocal control flow.

SEQ

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

SEQ

returns false

SEQ abbreviation $\equiv$ (SEQ ,@body).

IF

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

AND

evaluates either ,then if ,test is non-false otherwise evaluates (AND ,form ,@forms).

OR

evaluates either (OR ,form ,@forms), (OR ,form).

UNLESS

evaluates (UNLESS ,test ,@body).

COND

evaluates (COND ((,@keys) ,@body) ,@body)...

4.1 Conditional Control

IF

evaluates (IF ,test (SEQ ,@body)) in the same as (USE/EXPORT ,name)

same as USE plus reexports all imported bindings.

SEQ


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

\begin{align*}
\text{REP} & \text{ (REP ,name ((,var ,init) ...) ,@body)} \quad S \\
\text{defines a recursive loop (cf., Dylan's \textsc{iterate} or Scheme's (\textsc{let},var ...)).}
\end{align*}

\begin{align*}
\text{ESC} & \text{ (ESC ,name ,@body)} \quad S \\
\text{evaluates (SEQ ,@body) with an exit function of a single parameter, } x, \text{ bound to name that if called, will cause \text{esc} to return the value of } x \text{ (cf. Dylan's BLOCK/\textsc{return} form and CL's UNWIND-\textsc{protect}). It is illegal to call the exit function after the execution of the creating \text{esc} form (i.e., no upward continuations).}
\end{align*}

\begin{align*}
\text{FIN} & \text{ (FIN ,protected ,@cleanups)} \quad S \\
\text{ensures that (SEQ ,@cleanups) is evaluated whether or not an \text{esc} upwards exit is taken during the dynamic-extent of ,protected (cf. Dylan's BLOCK/CLEANUP form and CL's UNWIND-\textsc{protect}). The result of a \text{fin} form is the result of evaluating its protected form.}
\end{align*}

\begin{align*}
\text{ASSERT} & \text{ (ASSERT ,test ,message ,@args)} \quad S \equiv \text{ (UNLESS ,test (ERROR ,message ,@args))}
\end{align*}

5 Types, Classes and Properties

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

$\text{GOO}$ 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:

\begin{align*}
\text{<type>} \quad & \text{(any)} \quad \mathbb{C} \\
\text{isa?} & \text{ (x|<any> y|<type> => <loog}) \quad \mathbb{G} \\
\text{subtype?} & \text{ (x|<type> y|<type> => <log>)} \quad \mathbb{G} \\
\text{new} & \text{ (type|<type> prop-inits|...) } \quad \mathbb{G} \\
\text{returns true iff } x \text{ is a subtype of } y. \\
\text{creation protocol taking type and creation options where } \text{prop-inits contains getter / initial value pairs.}
\end{align*}

5.1 Singletons

Singleton types match exactly one value using $\text{=}$. Singletons are the most specific types.

\begin{align*}
\text{<singleton>} \quad & \text{(type)} \quad \mathbb{C} \\
\text{t=} & \text{ (x|<any> => <singleton>)} \quad \mathbb{G} \\
\text{returns singleton constrained to } x. \\
\text{type-object} & \text{ (x|<singleton> => <any>)} \quad \mathbb{G} \\
\text{object that singleton type matches.}
\end{align*}

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.

\begin{align*}
\text{<subclass>} \quad & \text{(type)} \quad \mathbb{C} \\
\text{t=} & \text{ (x|<class> => <subclass>)} \quad \mathbb{U} \\
\text{returns subclass type constrained to subclasses of } x. \\
\text{type-class} & \text{ (x|<subclass> => <class>)} \quad \mathbb{U} \\
\text{object that subclass type matches.}
\end{align*}

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.

\begin{align*}
\text{<union>} \quad & \text{(type)} \quad \mathbb{C} \\
\text{t=} & \text{ (type|... => <union>)} \quad \mathbb{U} \\
\text{returns union type representing disjunction of types.}
\end{align*}

5.4 Product

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

\begin{align*}
\text{<product>} \quad & \text{(type)} \quad \mathbb{C} \\
\text{t=} & \text{ (types|... => <product>)} \quad \mathbb{U} \\
\text{returns product type specifying the cross product of types.}
\end{align*}

5.5 Classes

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

\begin{align*}
\text{<class>} \quad & \text{(type)} \quad \mathbb{C} \\
\text{class-name} & \text{ (x|<class> => (t? <sym>))} \quad \mathbb{G} \\
\text{returns class name or false otherwise.} \\
\text{class-ancestors} & \text{ (x|<class> => <seq>)} \quad \mathbb{G} \\
\text{class precedence list including this class. See Appendix A for details.} \\
\text{class-direct-props} & \text{ (x|<class> => <seq>)} \quad \mathbb{G} \\
\text{properties defined directly on this class.} \\
\text{class-props} & \text{ (x|<class> => <seq>)} \quad \mathbb{G} \\
\text{properties defined on this class or any superclass.} \\
\text{class-children} & \text{ (x|<class> => <seq>)} \quad \mathbb{G} \\
\text{direct subclasses.}
\end{align*}
defines a class named \( \text{name} \) with direct parents \( \text{parents} \):

\[
\text{new} \begin{cases} \text{type}\langle\text{class}\rangle \text{prop-init}\ldots \end{cases}
\]

creates an instance of type \( \text{type} \) and \( \text{prop} \) initialized as specified by \text{prop-init}. For example, \((\text{new} \begin{cases} \text{point}\langle x \text{=1} \rangle \text{point}\langle y \text{=2} \rangle \end{cases})\) 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 \( \text{prop} \). Property values can either be specified at creation time with key-value pairs and setters. Descriptions of properties are instances of \( \text{gen} \).

The following functions are called with a single argument, the object under construction.

- \( \text{prop}\langle \rangle \rightarrow \langle \text{any} \rangle \)
- \( \text{prop}\langle \rangle \rightarrow \langle \text{gen} \rangle \)
- \( \text{class on which property was directly defined} \)
- \( \text{prop-getter} \langle \rangle \rightarrow \langle \text{gen} \rangle \)
- \( \text{reader accessor generic} \)
- \( \text{prop-setter} \langle \rangle \rightarrow \langle \text{gen} \rangle \)
- \( \text{writer accessor generic} \)
- \( \text{prop-type} \langle \rangle \rightarrow \langle \text{type} \rangle \)
- \( \text{prop-init} \langle \rangle \rightarrow \langle \text{fun} \rangle \)
- \( \text{lazy initialization function} \)
- \( \text{prop-owner} \langle \rangle \rightarrow \langle \text{gen} \rangle \)
- \( \text{prop-bound?} \langle \rangle \rightarrow \langle \text{prop} \rangle \)
- \( \text{type constraining property value} \)
- \( \text{find-getter} \langle \text{class} \rangle \langle \text{getter} \rangle \langle \text{gen} \rangle \rightarrow \langle \text{met} \rangle \)
- \( \text{find setter method defined on given class} \)
- \( \text{find-setter} \langle \text{class} \rangle \langle \text{setter} \rangle \langle \text{gen} \rangle \rightarrow \langle \text{met} \rangle \)
- \( \text{find setter method defined on given class} \)
- \( \text{prop-find-g Setter} \langle \rangle \rightarrow \langle \text{loc} \rangle \)
- \( \text{returns true if property with getter \( g \) is bound in instance \( x \).} \)
- \( \text{add-prop} \langle \text{type}\langle\text{type}\rangle \text{init}\langle\text{fun}\rangle \rangle \)

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.

\[
\text{fig!} \langle \text{DP}, \text{name}, \langle \text{oname}, \text{owner} \rangle, \langle \text{type}, \langle \text{init} \rangle \rangle \rangle
\]

add's an immutable property to \( \text{owner} \) with getter named \( \text{name} \), type \( \text{type} \), and optionally initial value \( \text{init} \). The initial value function is evaluated lazily when prop's value is first requested.

\[
\text{fig!} \langle \text{DP}, \text{name}, \langle \text{oname}, \text{owner} \rangle, \langle \text{type}, \langle \text{init} \rangle \rangle, \langle \text{init} \rangle \rangle
\]

same as \fig! but mutable with setter named \( \text{name} \) \# "setter"

### 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 \( \text{fun} \). Generic functions are instances of \( \text{gen} \) and methods are instances of \( \text{met} \).

\[
\text{fun} \langle \rangle \rightarrow \langle \text{any} \rangle
\]

\( \text{fun-name} \langle \rangle \rightarrow \langle \text{symbol} \rangle \)

returns the name of function or false if unavailable.

\[
\text{fun-specs} \langle \rangle \rightarrow \langle \text{type} \rangle
\]

returns the specializers of \( x \) or nil if unavailable.

\[
\text{fun-mary?} \langle \rangle \rightarrow \langle \text{int} \rangle
\]

returns true iff the function takes optional arguments.

\[
\text{fun-arity} \langle \rangle \rightarrow \langle \text{int} \rangle
\]

returns \( x \)'s number of required arguments.

\[
\text{fun-val} \langle \rangle \rightarrow \langle \text{type} \rangle
\]

returns the return type of \( x \).

\[
\text{fun} \langle \text{FUN}, \langle \text{sig}, \langle \text{body} \rangle \rangle \rangle
\]

creates an anonymous method with signature \( \text{sig} \) and when called evaluates \( \text{body} \) \( \equiv \) \( \text{fun} \) \( \langle \text{sig}, \langle \text{body} \rangle \rangle \) (cf. Scheme's \( \lambda \) abstraction). The following a few example functions and their application:

\[
\begin{align*}
\langle \text{fun} \langle x \rangle \langle 1 \rangle \rangle & \equiv 1 \\
\langle \text{fun} \langle x \langle \text{int} \rangle \langle 1 \rangle \rangle \langle x \rangle \langle 2 \rangle \rangle & \equiv 2 \\
\langle \text{fun} \langle x \rangle \langle 1 \rangle \langle 2 \rangle \rangle & \equiv 2 \\
\langle \text{fun} \langle x \rangle \langle \langle \text{tup} \langle \text{int} \rangle \rangle \rangle \langle x \rangle \langle 1 \rangle \rangle & \equiv \langle \text{tup} \langle 1 \rangle \rangle
\end{align*}
\]

\[
\text{fun} \langle \langle \langle \text{...} \rangle \rangle \rangle \equiv \langle \langle \text{...} \rangle \rangle \text{. This is particularly useful for lightweight thunks (e.g., } \langle x \rangle \langle 1 \rangle \rangle \equiv \langle \langle \text{fun} \langle x \rangle \langle 1 \rangle \rangle \rangle \text{. N.B. this is an experimental syntax and might change in the future.)}
\]

where

\[
\begin{align*}
\text{sig} & \equiv \langle \text{params}, \langle \text{param} \rangle \rangle \\
\text{params} & \equiv \langle \text{vars}, \langle \text{name}, \rangle \rangle \\
\text{where parameter lists can include an optional nary parameter which binds to all arguments beyond required arguments.}
\end{align*}
\]

\[
\begin{align*}
\text{...} & \equiv \langle \langle \text{...} \rangle \rangle \\
\text{fun abbreviation} & \equiv \langle \text{fun}, \langle \text{sig}, \langle \text{body} \rangle \rangle \rangle
\end{align*}
\]
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 specialization. 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.

### Example

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(fun x y) ...</code></td>
<td>calls <code>t</code> with arguments (<code>cat (sub args 0 (- (len args) 2)) (elt args (- (len args) 1)))</code>.</td>
</tr>
</tbody>
</table>

6.2 Methods

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

- `(defmethod (meth ...) (funmeth ...))` determines whether `meth` is applicable when called with `args`.
- `(define-method (meth ...) (funmeth ...))` 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` (cf. Dylan’s `define-method`).

6.3 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.

### Example

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(quote ...)</code></td>
<td>evaluates <code>form</code> in the midst of a quasiquote expression, abbreviated <code>&quot;&quot;</code>.</td>
</tr>
<tr>
<td><code>(quote-unquote ...)</code></td>
<td>evaluates <code>form</code> in the midst of a quasiquote expression and splices it in, abbreviated <code>&quot;.&quot;</code>.</td>
</tr>
</tbody>
</table>
| `(match ...)`      | evaluates `val` corresponding to first `pat` matching `exp`. The pattern is much the same as QUASIQUOTE and can contain either QUOTE or UNQUOTE-SPLICING variables. For example,
  | `(match (1 2) ((,a ,b) (1st a b)) ...)` | equivalent to `(if (not ,test) (seq ,@body)))` defines the when macro in GOO. |

where

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pattern</code></td>
<td>evaluates <code>(SEQ ,@body)</code> at compile-time allowing a user to make available computations for the purpose of macro-expansion.</td>
</tr>
<tr>
<td><code>ct</code></td>
<td>equivalent to ct, but also includes a copy of ,@body in compiled images. Similar to <code>(eval-when (:compile-toplevel :execute) ...)</code> in Common LISP. The return value of <code>ct-also</code> is undefined.</td>
</tr>
<tr>
<td><code>macro-expand</code></td>
<td>recursively expands macros in expression ,form.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gensym</code></td>
<td>defines a binding with name ,name bound to a generic with signature ,sig.</td>
</tr>
<tr>
<td><code>(LET ((,name #f) ...))</code></td>
<td>returns <code>s</code>’s methods.</td>
</tr>
<tr>
<td><code>(SET ,name (fun ,sig ,@body)) ...</code></td>
<td>adds method <code>y</code> to generic <code>x</code></td>
</tr>
<tr>
<td><code>(LET ((,name #f) ...))</code></td>
<td>returns both the list of sorted applicable methods and any ambiguous methods when generic <code>x</code> is called with arguments <code>args</code>.</td>
</tr>
</tbody>
</table>
7 Scalars

GOO provide a rich set of simple objects.

7.1 Any

All objects are derived from <any>.

\[ \text{class-of} \quad \text{[x<any> y<any> => <class>]} \]

returns concrete class of \( x \).

\[ \text{==} \quad \text{[x<any> y<any> => <log>]} \]

returns true iff \( x \) and \( y \) are computationally equivalent.

\[ \text{=} \quad \text{[x<any> y<any> => <log>]} \]

returns true iff \( x \) and \( y \) are equal, where equality is user defined and defaults to \( \text{==} \).

\[ \text{˜=} \quad \text{[x<any> y<any> => <log>]} \]

\[ \equiv \quad \text{[not (=} x y)} \]

\[ \text{to-str} \quad \text{[x<any> => <str>]} \]

returns string representation of object.

7.2 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.

\[ \text{#t} \quad \text{[<log>]} \]

\[ \text{#f} \quad \text{[<log>]} \]

\[ \text{not} \quad \text{[x<any> => <log>]} \]

\[ \text{lo-str} \quad \text{[x<any> => <str>]} \]

returns string representation of object.

7.3 Magnitudes

Magnitudes are totally orderable objects. Users are only required to implement \( < \) and \( \geq \).

\[ \text{min} \quad \text{[x<map> y<map> => <map>]} \]

returns the smallest of \( x \) and \( y \).

\[ \text{max} \quad \text{[x<map> y<map> => <map>]} \]

returns the largest of \( x \) and \( y \).

7.4 Locatives

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

\[ \text{loc} \quad \text{[<map>]} \]

\[ \text{loc-val} \quad \text{[x<loc> => <any>]} \]

returns the object pointed to by \( x \).

\[ \text{address-of} \quad \text{[x<any> => <loc>]} \]

returns address of particular object.

7.5 Characters

GOO currently supports 8 bit ASCII characters.

\[ \text{alpha?} \quad \text{[x<chr> => <log>]} \]

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

\[ \text{digit?} \quad \text{[x<chr> => <log>]} \]

returns true iff \( x \) is one of the ten ASCII numeric characters.

\[ \text{lower?} \quad \text{[x<chr> => <log>]} \]

returns true iff \( x \) is one of the ASCII lower case characters.

\[ \text{upper?} \quad \text{[x<chr> => <log>]} \]

returns true iff \( x \) is one of the ASCII uppercase characters.

\[ \text{to-digit} \quad \text{[x<chr> => <int>]} \]

converts ascii representation of digit to an integer one.

\[ \text{to-lower} \quad \text{[x<chr> => <chr>]} \]

returns lowercase version of uppercase alphabetic characters otherwise returns \( x \).

\[ \text{to-upper} \quad \text{[x<chr> => <chr>]} \]

returns uppercase version of lowercase alphabetic characters otherwise returns \( x \).

7.6 Numbers

\[ \text{<num>} \quad \text{[<map>]} \]

\[ + \quad \text{[x<num> y<num> => <num>]} \]

returns the sum of its arguments.

\[ - \quad \text{[x<num> y<num> => <num>]} \]

returns the difference of its arguments.

\[ \ast \quad \text{[x<num> y<num> => <num>]} \]

returns the product of its arguments.

\[ / \quad \text{[x<num> y<num> => <num>]} \]

returns the quotient of its arguments.
round \( x \mapsto \text{int} \)

returns closest integer to \( x \). If \( x \) is exactly between two integers then the implementation is free to return either integer.

round-tos \( (x \mapsto \text{int}) \)

returns \( x \) to closest \( \text{flo} \) \( \text{digits} \) precision.

floor \( x \mapsto (\text{tup} \text{int} \text{rem} \text{num}) \)

returns an integer by truncating \( x \) towards negative infinity.

ceil \( x \mapsto (\text{tup} \text{int} \text{rem} \text{num}) \)

returns an integer by truncating \( x \) towards positive infinity.

trunc \( x \mapsto (\text{tup} \text{int} \text{rem} \text{num}) \)

returns an integer by truncating \( x \) towards zero.

mod \( x \mapsto \text{num} \)

returns the remainder after taking the floor of the quotient of \( x \) and \( y \).

div \( x \mapsto \text{num} \)

returns the trunc of the quotient of \( x \) and \( y \).

rem \( x \mapsto \text{num} \)

returns remainder after dividing \( x \) by \( y \).

pow \( x \mapsto \text{num} \)

returns \( x \) raised to the \( e \) power.

exp \( x \mapsto \text{num} \)

\( \equiv (\text{pow} e x) \)

sqrt \( x \mapsto \text{num} \)

returns the square root of \( x \).

pos? \( x \mapsto \text{num} \)

\( \equiv (> x 0) \)

zero? \( x \mapsto \text{num} \)

\( \equiv (= x 0) \)

neg? \( x \mapsto \text{num} \)

\( \equiv (< x 0) \)

neg \( x \mapsto \text{num} \)

\( \equiv (- x) \)

abs \( x \mapsto \text{num} \)

\( \equiv (\text{if} (\text{neg?} x) (- x) x) \)

num-to-str-base \( x \mapsto \text{num} \)

returns string representation of \( x \) in base \( b \).

num-to-str \( x \mapsto \text{num} \)

equiv \( (\text{num-to-str-base} x 10) \)

str-to-num \( x \mapsto \text{num} \)

returns closest number corresponding to string \( x \).

INCF \( \text{name} \)

\( \equiv (\text{SET} \text{name} (+ \text{name} 1)) \)

DECF \( \text{name} \)

\( \equiv (\text{SET} \text{name} (+ \text{name} 1)) \)
8 Collections

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

8.1 Mutable Collections

Mutation is seen as a necessary evil and is supported but segregated in hopes of trying to isolate and optimize the nondestructive cases. Mutation includes the notion of modifying values and adding/removing keys. The hope is that functional (nondestructive) programs will be both more succinct, understandable, and efficient than equivalent destructive programs. Only core collection operators are given destructive versions. All others can be built out of nondestructive operators followed by \texttt{into} \(^1\).

\(^1\)When optimization is in place, the \texttt{!} suffixed functions will be deprecated.
8.2 Enumerators

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

8.3 Packers

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

\[
\text{packing (for ((e ' (1 2 3 4 5)))}
\]
\[
\text{(when (odd? e) (pack e)))}
\]
\[
\Rightarrow (1 3 5)
\]

They can also be used for summing values etc:

\[
\text{packing-in (x (int))}
\]
\[
\text{(for ((e ' (1 2 3 4 5)))}
\]
\[
\text{(when (odd? e) (pack-in x e)))}
\]
\[
\Rightarrow 9
\]

9.8.4 Maps

Maps represent collections with explicit keys.

\[
\text{map} \leftrightarrow \text{<col>}
\]

\[
\text{<tab>} \leftrightarrow \text{<map> <col!>}
\]

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.

\[
\text{tab-growth-factor} \leftrightarrow \text{<tab>} \Rightarrow \text{<fio>}
\]

factor by which to grow capacity.
8.5 Sequences

Sequences are collections with nonnegative integer keys.

\[ \text{<seq>} \]
\[ \text{<seq>} \text{<col>} \] = \text{C}

\[ \text{<seq>} \text{<seq>} \text{<col>} \] = \text{C}

\[ \text{Immutable sequence.} \]

\[ \text{1st} \] \( (x|<\text{seq}> => <\text{any}> ) \] = \( \ell x 0 \)
\[ \text{2nd} \] \( (x|<\text{seq}> => <\text{any}> ) \] = \( \ell x 1 \)
\[ \text{3rd} \] \( (x|<\text{seq}> => <\text{any}> ) \] = \( \ell x 2 \)
\[ \text{last} \] \( (x|<\text{seq}> => <\text{any}> ) \] = \( \ell x (- (\text{len} x) 1) \)

\[ \text{pos} \]
\[ (x|<\text{seq}> \text{v} |<\text{any}> => (t? <\text{int}>)) \] = \text{G}

finds position of \text{v} in \text{x} else returns false.

\[ \text{finds} \]
\[ (x|<\text{seq}> \text{y} |<\text{seq}> => (t? <\text{int}>)) \] = \text{G}

finds position of \text{y} in \text{x} else returns false.

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

returns sequence with \text{y} added to the end of \text{x}.

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

returns sequence with \text{y} added to \text{x}.

\[ \text{pop} \]
\[ (x|<\text{seq}> => (\text{tup} \text{val} |<\text{any}> <\text{seq}>)) \] = \text{G}

returns last pushed element of \text{x} and new sequence with that element removed from \text{x}.

\[ \text{rev} \]
\[ (x|<\text{seq}> => <\text{seq}> ) \] = \text{G}

returns reversed sequence.

\[ \text{cat} \]
\[ (x|<\text{seq}> \text{more} . . . => <\text{seq}> ) \] = \text{G}

returns concatenated sequence.

\[ \text{sub} \]
\[ (x|<\text{seq}> \text{from} |<\text{int}> \text{below} |<\text{int}> => <\text{seq}> ) \] = \text{G}

subsequence of \text{x} between from and below.

\[ \text{pos*} \]
\[ ((x|<\text{seq}> \text{from} |<\text{int}> \text{below} |<\text{int}> ) \] = \text{G}

\[ \text{sub} \]
\[ ((x|<\text{seq}> \text{from} |<\text{int}> => <\text{seq}> ) \] = \text{G}

\[ \text{sub*} \]
\[ ((x|<\text{seq}> \text{from} |<\text{int}> ) \] = \text{G}

\[ \text{inserts} \]
\[ (x|<\text{seq}> \text{v} |<\text{any}> \text{pos} |<\text{int}> => <\text{seq}> ) \] = \text{G}

inserts \text{v} before \text{pos} in \text{x}.

\[ \text{sub-setter} \]
\[ (\text{dst} |<\text{seq}> \text{src} |<\text{seq}> \text{from} |<\text{int}> \text{below} |<\text{int}> ) \] = \text{G}
replaces subsequence in range between from and below of dst with contents of src. Provides insertion, deletion, and replacement operations rolled into one.

\[
\text{sub-setter} \equiv \text{sub-setter dst src from (len dst)}
\]

\[
\text{del-vals!} \equiv (x|<\text{seq}!> v|<\text{any}> => <\text{seq}!>)
\]

\[
\text{del-dups!} \equiv (x|<\text{seq}!> => <\text{seq}!>)
\]

\[
\text{sort-by!} \equiv (s|<\text{seq}> f|<\text{fun}> => <\text{seq}>)
\]

\[
\text{sort!} \equiv (s|<\text{seq}> => <\text{seq}>)
\]

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 GOO.

\[
\text{head} \equiv (s|<\text{seq}> => <\text{any}>)
\]

\[
\text{tail} \equiv (s|<\text{seq}> => <\text{seq}>)
\]

\[
\text{lst} \equiv (\text{elts}|... => <\text{lst}>)
\]

\[
\text{nil} \equiv ()
\]

\[
\text{pair} \equiv (x|<\text{any}> y|<\text{lst}> => <\text{lst}>)
\]

8.5.3 Zips

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

\[
\text{zip} \equiv (\text{cs}|... => <\text{zip}>)
\]

\[
\text{unzip} \equiv (s|<\text{seq}> => <\text{zip}>)
\]

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.

\[
\text{cycle} \equiv (\text{seq} | <\text{seq}>)
\]

\[
\text{cycle} \equiv (x|... => <\text{cycle}>)
\]

Strings

GOO currently implements ASCII strings.

\[
\text{str} \equiv (\text{elts}|... => <\text{str}>)
\]

8.6 Lazy Series’

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

\[
\text{range} \equiv (\text{from}|<\text{num}> \text{test}|<\text{fun}> \text{lim}|<\text{num}> \text{by}|<\text{fun}> => <\text{range}>)
\]

\[
\text{from} \equiv (\text{from}|<\text{num}> => <\text{range}>)
\]

\[
\text{below} \equiv (\text{lim}|<\text{num}> => <\text{range}>)
\]

Steps represent step functions.

\[
\text{first-then} \equiv (\text{first}|<\text{fun}> \text{then}|<\text{fun}> => <\text{step}>)
\]

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

\[
\text{cycle} \equiv (x|... => <\text{cycle}>)
\]

returns a new cycle that repeats elements of x.
9 Symbols

Symbols are uniquified (aka interned) strings.

- `sym` → `(any?)
- `sym-tab` → `(tab?)

symbol table class.

`≡` coerces a string to a symbol.

- `cat-sym` → `(sym-tab)->(sym)
- `gen sym` → `(sym)->(sym)

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

- `fab-setter-name` → `(x|<sym-tab> => <sym>)
- `(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 function used to take care of conditions of a particular type. A handler function 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.

Condition hierarchy - protocol for determining whether a handler handles a particular condition.

- `TRY` → `(TRY ,try-options ,handler ,@body)`

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)` An expression 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.

build-condition-interactively returns a string describing an anonymous handler for this type of condition.

- `condition-args` → `(<simple-condition>)
- `condition-message` → `(<simple-condition>)

returns a string describing an anonymous condition for this type of condition.

- `condition-arguments` → `(x|<condition> args|...)
- `condition-description` → `(x|<condition> args|...)

returns a condition with optional arguments args.

- `simple-condition` → `(<condition>)
- `condition` → `(<condition>)

a condition consisting of a seq message and arguments.

- `condition-arguments` → `(x|<condition> args|...)
- `condition-message` → `(x|<condition> args|...)

returns seq String.

- `condition-arguments` → `(x|<condition> args|...)
- `condition-message` → `(x|<condition> args|...)

returns seq String arguments.

- `condition-arguments` → `(x|<condition> args|...)
- `condition-message` → `(x|<condition> args|...)

returns a condition that can not be safely ignored.

- `condition-arguments` → `(x|<condition> args|...)
- `condition-message` → `(x|<condition> args|...)

returns an error that consists of a seq message and arguments.

- `condition-arguments` → `(x|<condition> args|...)
- `condition-message` → `(x|<condition> args|...)

a condition that indicates something is invalid about the program.

- `condition-arguments` → `(x|<condition> args|...)
- `condition-message` → `(x|<condition> args|...)

a condition that indicates something is invalid about the program.

10.1 Conditions Hierarchy

**GOO** has a builtin hierarchy of conditions.

- `arithmetic-error` → `(error)
- `catch-error` → `(error)`
- `keyboard-error` → `(error)`
- `user-interrupt-key` → `(error)`

a system fault was detected.
an assertion failure occurred.

an binding or property was found to be unbound.

a binding was found to be unbound.

an unbound variable was found.

unbound property was discovered.

a property error was discovered.

a property accessor if available.

a property owner if available.

an property type error was discovered.

a property error was discovered.

a range error on a collection failed.

a type check has failed.

a value error on which type check failed.

a type error on which type check failed.

a call error on which type check failed.

a function call failed.

a call error on which function call failed.

returns function on which call failed.

returns arguments on which call failed.

returns arguments on which call failed.

wrong number of arguments supplied in function call.
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>`
- `open` (t | (t < port>) x | <str> => <port>)
- `close` (x | <port>)
- `noop` default.
- `WITH-PORT` (WITH-PORT ,name ,port) ,@body
- `eof-object?` (x | <chr> => <log>)
- `<in-port>`
- `in` <in-port>
- `get` (x | <in-port> => <chr>)
- `peek` (x | <in-port> => <chr>)
- `ready?` (x | <in-port> => <log>)
- `<out-port>`
- `out` <out-port>
- `force-out` (x | <out-port>)
- `newline` (x | <out-port>)
- `say` (x | <out-port> args|...)

11.1.1 File Ports

File ports are ports which map to files.

- `<file-port>`
- `close` (x | <file-port>)
- `open` (t | (t < file-in-port>) name|<str> => <file-in-port>)
- `create` file in port mapped to a file with filename name.
- `file-out-port` (x | <out-port>)
- `open` (t | (t < file-out-port>) name|<str> => <file-out-port>)
- `create` 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>`
- `port-contents` (x | <str-port> => <str>)
- `port-index` (x | <str-port> => <int>)
- `port-index` (x | <str-port> => <int>)
- `PORT-TO-STR` (PORT-TO-STR ,name ,port) ,@body

11.2 Formatted I/O

`GOO` provides convenient s-expression reading/writing facilities.

- `read` (x | <in-port> => <any>)
- `write` (x | <out-port> y | <any>)
- `writeln` (x | <out-port> y | <any>)
- `emit` (x | <out-port> y | <any>)
- `msg` (x | <out-port> message|<seq> args|...)

14
formatted output using special commands embedded in message. supported
commands are:

- `(write x arg)`
- `(display x arg)`
- `(write-char x <sym>)`

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

```
(post [messages[compose] args[]])
```

```s
(app msg out message args)
```

12 System

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

- `app-filename` returns the filename of the application.
- `app-args` returns a list of argument strings with which the application was called.
- `os-name` returns name of current operating-system.
- `os-val` returns the process id of the current process.
- `os-val=` setter returns OS environment variable value.
- `os-val-setter` sets OS environment variable value.
- `process-id` returns the process id of the current GOO process.
- `create-directory` creates directory with the given name. The parent directory must already
exist, and must contain no item with the given name.
- `parent-directory` finds the parent directory of the current filename.
- `probe-directory` makes sure that the named directory exists.

12.1 Files and Directories

A preliminary set of file and directory facilities are provided.

- `file-tmtime` return the last modification time of a file in seconds (relative to the n
GOO epoch) as a floating point number.
- `file-exists?` return true if and only if a file (or a directory, etc.) exists with the given
name.
- `file-type` return 'file, directory or some other symbol, depending on the type of
the file.
- `create-directory` create a directory with the given name. The parent directory must already
exist, and must contain no item with the given name.
- `parent-directory` find the parent directory of the current filename.
- `probe-directory` make sure that the named directory exists.

12.2 Pathnames

Pathnames allow you to work with hierarchical, structured path-
names in a reasonably portable fashion.

- `pathname-to-components` 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 por-
tion of this pathname (everything but the label) must be non-empty at all
times.

- `components-to-pathname` reassemble components created by the above function.
- `parent-name` 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.
- `last` calculate the parent directory of a pathname.

```
(pathname to components) => (components [list] => [str])
```

```
(label components) => (components [list] => [list])
```

```
(hierarchical components) => (components [list] => [list])
```

13 Threads

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

```
(creates thread in separate thread)
```

```
(thread name) => (thread [thread] => (t? [sym]))
```

```
(thread priority) => (thread [thread] => [int])
```

```
(thread function) => (thread [thread] => [fun])
```

```
(thread run) => (thread [thread] => (init ... => [thread]))
```

Creates thread and runs thread-function of created thread in separate
OS thread.
14 Compiler

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

\begin{verbatim}
...<g2c-module-loader> (finite (lock-lock ,lock) (lock-unlock ,lock)) ...
\end{verbatim}

\texttt{g2c} builds an application-specific module loader used in future \texttt{g2c} builds.

\begin{verbatim}
appname|<str> modname|<str> => <any>
\end{verbatim}

\texttt{g2c} builds entire \texttt{GOO} world.

\begin{verbatim}
(load (filename|<str> modname|<sym> => <any>) (eval (x|<any> modname|<sym> => <any>)) ...
\end{verbatim}

\texttt{g2c} changes destination directory to be (cat *g2c-* (to-str name)). This is useful for bootstrapping.

15 Top Level

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

\begin{verbatim}
...<g2c-module-loader> (finite (lock-lock ,lock) (lock-unlock ,lock)) ...
\end{verbatim}

\texttt{g2c} builds entire \texttt{GOO} process to a file named \texttt{filename}.

\begin{verbatim}
...<g2c-module-loader> (finite (lock-lock ,lock) (lock-unlock ,lock)) ...
\end{verbatim}

\texttt{g2c} causes \texttt{appname} into directory \texttt{modname}.

\begin{verbatim}
...<g2c-module-loader> (finite (lock-lock ,lock) (lock-unlock ,lock)) ...
\end{verbatim}

\texttt{g2c} builds entire \texttt{GOO} process to a file named \texttt{filename}.

\begin{verbatim}
...<g2c-module-loader> (finite (lock-lock ,lock) (lock-unlock ,lock)) ...
\end{verbatim}

\texttt{g2c} builds entire \texttt{GOO} process to a file named \texttt{filename}.

\begin{verbatim}
...<g2c-module-loader> (finite (lock-lock ,lock) (lock-unlock ,lock)) ...
\end{verbatim}

\texttt{g2c} builds entire \texttt{GOO} process to a file named \texttt{filename}.

\begin{verbatim}
...<g2c-module-loader> (finite (lock-lock ,lock) (lock-unlock ,lock)) ...
\end{verbatim}

\texttt{g2c} builds entire \texttt{GOO} process to a file named \texttt{filename}.

\begin{verbatim}
...<g2c-module-loader> (finite (lock-lock ,lock) (lock-unlock ,lock)) ...
\end{verbatim}

\texttt{g2c} builds entire \texttt{GOO} process to a file named \texttt{filename}.
The document contains information about configuring and installing a software package. It includes details on setting the root directory, configuring with the `--with-threads` argument, and executing `make` to build and install the package. It also provides instructions for setting the root directory using environment variables and details on the `g2c-goo` command.

### Usage

Typing `goo` at the shell will start a GOO read-eval-print loop, which accepts s-expressions and top-level commands. The following are a list of available commands:

- `:quit` exits from GOO.
- `C-c` invokes a recursive read-eval-print loop.
- `:g2c-eval` to change to dynamic compilation evaluation.
- `:ast-eval` to change to ast evaluation.

### 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)
```

### Debugger

A keyboard interrupt or any error enters the user into the debugger, which provides a superset of the commands available at top-level.

### Emacs Support

A rudimentary emacs-based development system is provided.

### Emacs Mode

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

```emacs
(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.

### Emacs Shell

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

```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)
```

The original idea was dreamed up and first implemented by Andrew Sutherland and then improved by James Knight.
Emacs TAGS files can be generated by typing `make all-tags` in the 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-z switch-to-goo
```

Check out `goo-shell.el` for the complete list of commands / key-bindings. I doubt the compile commands do anything useful cause there isn’t a compiler.

### 17.3.3 Emacs 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
M-. tags-search
```

### 18 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. Please, please, please send bug reports to jrb@googoogaga.org. I will fix your bugs asap. The GOO website [www.googoogaga.org](http://www.googoogaga.org) will have papers, releases, FAQS, etc.

### 19 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`.

### 20 History and Acknowledgements

GOO has greatly benefited 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, including by writing the Emacs `goo-mode`. James Knight was one of the 6.894 students and became my MEng student after the course.

He has helped in many many ways including the writing of the `save-image` facility, the speeding up of the runtime, and the improving of the non local exit facility. Eric Kidd worked with me during the summer of 2001 implementing the module system, restarts, and the dependency tracking system. During that summer I decided that a Prototype-based object system was inadequate for the type system I was interested in supporting and changed over to the present type-based system. I presented my ideas on Proto at LL1 in the Fall of 2001. Many stimulating conversations on the follow on LL1 discussion list 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 SWI [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:
B Subtyping Rules

This section defines the subtyping rules for GoO in terms of sub-type methods.

(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|<union> => <log>)
  (all? (op subtype? _ t2) (type-elts t1)))
(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? t1 t2))
(dm subtype? (t1|<class> t2|<singleton> => <log>) #f)
(dm subtype? (t1|<singleton> t2|<class> => <log>)
  (isa? (type-object t1) t2))
(dm subtype? (t1|<subclass> t2|<singleton> => <log>) #f)
(dm subtype? (t1|<class> t2|<subclass> => <log>)
  (and (== t1 (type-class t2)) (== (type-elts t2) (type-elts t1))))
(dm subtype? (t1|<subclass> t2|<class> => <log>)
  (subclass? (type-class t1) (type-class t2)))
(dm subtype? (t1|<singleton> t2|<subclass> => <log>)
  (and (isa? (type-object t1) (type-class t2))
    (isa? (type-object t2) (type-class t1))
    (subclass? (type-class t2) (type-class t1))))
(dm subtype? (t1|<product> t2|<type> => <log>) #f)
(dm subtype? (t1|<type> t2|<product> => <log>) #f)
(dm subtype? (t1|<product> t2|<product> => <log>)
  (all? (zipped subtype?) (zip (type-elts t1) (type-elts t2))))
(dm subtype? (t1|<subclass> t2|<class> => <log>)
  (subtype? <tup> t2))
}