GNU G-Golf

Edition 0.1.0, revision 1, for use with GNU G-Golf 0.1.0

The GNU G-Golf Developpers

This manual documents GNU G-Golf version 0.1.0.

Table of Contents

Preface

This manual describes how to use G-Golf. It relates particularly to G-Golf version 0.1.0.

Contributors to this Manual

Like G-Golf itself, the G-Golf reference manual is a living entity. Right now, the contributor to this manual is:

- David Pirotte

who is also the author and maintainer of G-Golf.

You are most welcome to join and help. Visit G-Golf's web site at `http://www.gnu.org/software/g-golf/` to find out how to get involved.

The G-Golf License

I. Introduction

About G-Golf

GNU G-Golf
GNOME: (Guile Object Library for).

Description

G-Golf is a Guile[1] Object Library for GNOME (`https://www.gnome.org/`).

G-Golf low level API comprises a binding to - (most of) the GObject Introspection (`https://developer.gnome.org/stable/gi`) and (some of) the GObject (`https://developer.gnome.org/gobject/stable/`) and Glib (`https://developer.gnome.org/glib/stable/`) libraries, as well as additional (G-Golf) utilities - used to import GObject libraries and build their corresponding G-Golf high level API.

> **Note:** to be precise, G-Golf imports (and depends on the exitence of) a Typelib (`https://gi.readthedocs.io/en/latest`) - a binary, readonly, memory-mappable database containing reflective information about a GObject library.

[1] GNU Guile (`http://www.gnu.org/software/guile`)
an interpreter and compiler for the Scheme (`http://schemers.org`) programming language.

G-Golf high level API makes (imported) GOBject classes and methods available using GOOPS, the Guile Object Oriented System (see Section "GOOPS" in *The GNU Guile Reference Manual*).

G-Golf is a tool to develop modern graphical applications.

Savannah

GNU G-Golf also has a project page on Savannah (`https://savannah.gnu.org/projects/g-golf`).

Obtaining and installing G-Golf

GNU G-Golf can be obtained from the following archive site `http://ftp.gnu.org/gnu/g-golf/`. The file will be named g-golf-version.tar.gz. The current version is 0.1.0, so the file you should grab is:

```
http://ftp.gnu.org/gnu/g-golf/g-golf-0.1.0.tar.gz
```

Dependencies

GNU G-Golf needs the following software to run:

- Autoconf >= 2.69
- Automake >= 1.14
- Makeinfo >= 6.6
- Guile (`http://www.gnu.org/software/guile`) >= 2.0.14 [allows 2.2 3.0]
- Guile-Lib (`http://www.nongnu.org/guile-lib`) >= 0.2.5
- Glib-2.0 (`https://developer.gnome.org/glib/stable/`) >= 2.48.0
- Gobject-2.0 (`https://developer.gnome.org/gobject/stable/`) >= 2.48.0
- Gtk+-3.0 (`https://developer.gnome.org/gtk3/stable/`) >= 3.24.0

  The `gtk+-3.0` module is required because it is the module that contains and installs `libgdk-3`, which G-Golf requires (as any other GI language binding) to support `Gdk` (in general) and `Gdk Events` (in particular).
- GObject-Introspection-1.0 (`https://developer.gnome.org/stable/gi`) >= 1.48.0

Install from the tarball

Assuming you have satisfied the dependencies, open a terminal and proceed with the following steps:

```
cd <download-path>
tar zxf g-golf-0.1.0.tar.gz
cd g-golf-0.1.0
./configure [--prefix=/your/prefix] [--with-guile-site=yes]
make
make install
```

Happy G-Golf (`http://www.gnu.org/software/g-golf/`)!

Install from the source

G-Golf (`http://www.gnu.org/software/g-golf/`) uses Git (`https://git-scm.com/`) for revision control, hosted on Savannah (`https://savannah.gnu.org/projects/g-golf`), you may browse the sources repository here (`http://git.savannah.gnu.org/cgit/g-golf.git`).

There are currently 2 [important] branches: `master` and `devel`. G-Golf (`http://www.gnu.org/software/g-golf/`) stable branch is master, developments occur on the devel branch.

So, to grab, compile and install from the source, open a terminal and:

```
git clone git://git.savannah.gnu.org/g-golf.git
cd g-golf
./autogen.sh
./configure [--prefix=/your/prefix] [--with-guile-site=yes]
make
make install
```

The above steps ensure you're using G-Golf (`http://www.gnu.org/software/g-golf/`) bleeding edge `stable` version. If you wish to participate to developments, checkout the `devel` branch:

```
git checkout devel
```

Happy `hacking!`

Notes:

1. The `default` and `--prefix` installation locations for source modules and compiled files (in the absence of `--with-guile-site=yes`) are:

   ```
   $(datadir)/g-golf
   $(libdir)/g-golf/guile/$(GUILE_EFFECTIVE_VERSION)/site-ccache
   ```

   If you pass `--with-guile-site=yes`, these locations become the Guile global site and site-ccache directories, respectively.

   The configure step reports these locations as the content of the `sitedir` and `siteccachedir` variables, respectivelly the source modules and compiled files install locations. After installation, you may consult these variables using pkg-config:

   ```
   pkg-config g-golf-1.0 --variable=sitedir
   pkg-config g-golf-1.0 --variable=siteccachedir
   ```

   You will need - unless you have used `--with-guile-site=yes`, or unless these locations are already 'known' by Guile - to define or augment your `GUILE_LOAD_PATH` and `GUILE_COMPILED_PATH` environment variables with these locations, respectively (or `%load-path` and `%load-compiled-path` at run time if you prefer[2] (See Environment Variables (`https://www.gnu.org/software/guile/manual/guile.html#Environment-Variables`)

[2] In this case, you may as well decide to either alter your `$HOME/.guile` personal file, or, if you are working in a mult-user environmet, you may also opt for a global configuration. In this case, the file must be named `init.scm` and placed it here (evaluate the following expression in a terminal): `guile -c "(display (%global-site-dir))(newline)"`.

and Load Path (`https://www.gnu.org/software/guile/manual/guile.html#Load-Paths`) in the Guile Reference Manual).

2. G-Golf also installs its `libg-golf.*` library files, in `$(libdir)`. The configure step reports its location as the content of the `libdir` variable, which depends on on the content of the `prefix` and `exec_prefix` variables (also reported). After nstallation, you may consult these variables using pkg-config:

   ```
   pkg-config g-golf-1.0 --variable=prefix
   pkg-config g-golf-1.0 --variable=exec_prefix
   pkg-config g-golf-1.0 --variable=libdir
   ```

   You will need - unless the `$(libdir)` location is already 'known' by your system - to either define or augment your `$LD_LIBRARY_PATH` environment variable, or alter the `/etc/ld.so.conf` (or add a file in `/etc/ld.so.conf.d`) and run (as root) `ldconfig`, so that G-Golf finds its `libg-golf.*` library files[3].
3. To install G-Golf, you must have write permissions to the default or `$(prefix)` directory and its subdirs, as well as to both Guile's site and site-ccache directories if `--with-guile-site=yes` was passed.
4. Like for any other GNU Tool Chain compatible software, you may install the documentation locally using `make install-info`, `make install-html` and/or `make install-pdf`.
5. Last but not least :), G-Golf comes with a `test-suite`, which we recommend you to run (especially before [Reporting Bugs], page 4):

   ```
   make check
   ```

Contact Information

Mailing list

G-Golf uses Guile's mailing lists:

- `guile-user@gnu.org` is for general user help and discussion.
- `guile-devel@gnu.org` is used to discuss most aspects of G-Golf, including development and enhancement requests.

Please use '`G-Golf -` ' to preceed the subject line of G-Golf related emails, thanks!

You can (un)subscribe to the one or both of these mailing lists by following instructions on their respective list information page (`https://lists.gnu.org/mailman/listinfo/`).

IRC

Most of the time you can find me on irc, channel *#guile*, *#guix* and *#scheme* on *irc.freenode.net*, *#clutter* and *#introspection* on *irc.gnome.org*, under the nickname *daviid*.

Reporting Bugs

G-Golf uses a bug control and manipulation mailserver. You may send your bugs report here:

[3] Contact your administrator if you opt for the second solution but don't have `write` priviledges on your system.

- `bug-g-golf@gnu.org`

You can (un)subscribe to the bugs report list by following instructions on the list information page (`https://lists.gnu.org/mailman/listinfo/bug-g-golf`).

Further information and a list of available commands are available here (`https://debbugs.gnu.org/server-control.html`).

II. Using G-Golf

Before you start

Naming Conventions

G-Golf is, or at least tries to be, consistent in the way '`things`' are being named, whether the functionality being '`exposed`' is from an imported Gnome library or is part of a G-Golf's core reference module.

Gnome Libraries

When G-Golf imports a Gnome library, its classes, properties, methods, functions, types and constant are renamed, which is achieved by calling [g-name->class-name], page 92, and [g-name->name], page 92, appropriately.

As described in their respective documentation entry, as well as in the [Customizing G-Golf], page 9, section, G-Golf offers a way to either ignore or partially customize the renaming process.

- **Classes**

Gnome library classes are imported as GOOPS classes (the Guile Object Oriented System, see Section "GOOPS" in *The GNU Guile Reference Manual*), and their respective name is given by the result of calling [g-name->class-name], page 92, for example:

```
GtkWindow -> <gtk-window>
ClutterActor -> <clutter-actor>
WebKitWebView -> <webkit-web-view>[4]
...
```

- **Properties**

Gnome library class properties are imported as GOOPS class slots, and their respective name is given by calling [g-name->name], page 92. Each property slot defines an `init-keyword` and an `accessor`, following G-Golf's accessors naming conventions (see [GOOPS Notes and Conventions], page 7.

As an example, the `<gtk-label>` class has a `label` slot, with the `#:label` init-keyword and `!label` accessor.

[4] By default, G-Golf sets `WebKit` as a renaming exception token, otherwise, the class name would be <web-kit-web-view>.

- **Methods**

Gnome library methods are imported as GOOPS class methods and added to their respective generic function, the name of which is given by calling [g-name->name], page 92.

In addition, unless otherwise specified (See [Customizing G-Golf], page 9, section), G-Golf also defines so called short name methods, obtained by dropping the container name (and its trailing hyphen) from the GI typelib method full/long names.

For example, the `<gtk-container>` class, which defines a `gtk-container-add` method, would also define, using G-Golf's default settings, an `add` method. To be more precise, G-Golf would create (if it does not exist) or reuse (if it exists) the `add` generic function, make and add a method with its specializers, in this case `<gtk-container>` and `<gtk-widget>`.

- **Functions**

Gnome library functions are imported as procedures, renamed by calling [g-name->name], page 92. For example:

```
gtk_window_new -> gtk-window-new
clutter_actor_new -> clutter-actor-new
...
```

- **Enums, Flags and Boxed types**

Gnome library enums, flags and boxed types are renamed by calling [g-name->name], page 92, (and cached, See [G-Golf Cache - Accessing], page 12, section).

Enum and flag type members are renamed by calling [g-name->name], page 92. To illustrate, here is an example:

```
,use (g-golf)

(gi-import-by-name "Gtk" "WindowPosition")
$2 = #<<gi-enum> 5618c7a18090>

(describe $2)
#<<gi-enum> 5618c7a18090> is an instance of class <gi-enum>
Slots are:
     enum-set = ((none . 0) (center . 1) (mouse . 2) (center-always . 3) (center-on-pa
     g-type = 94664428197600
     g-name = "GtkWindowPosition"
     name = gtk-window-position
```

G-Golf Core Reference

- **Procedures**

G-Golf procedure names that bind a Glib, GObject, Gdk or GObject GObject Introspection function always use the '`original`' name, except that `_` are replaced by `-`. For example:

```
g_main_loop_new
-> [g-main-loop-new], page 16

g_irepository_get_loaded_namespaces
-> [g-irepository-get-loaded-namespaces], page 50
```

G-Golf also comes with its own set of procedures, syntax and variables, aimed at not just reading a typelib, but making its functionality available from Guile (`http://www.gnu.org/software/guile`). Naming those, whenever possible, is done following the '`traditional way`' scheme name its procedures, syntax and variables. For example:

- procedure names that start with `call-with-input-`, `call-with-output-` followed by a Glib, GObject. Gdk or GI type, such as:

  `[call-with-input-typelib], page 80`

- syntax names that start as `with-` followed by a Glib, GObject, Gdk or GI type, such as:

  `[with-gerror], page 82`

When an '`obvious`' name can't be find '`on its own`', or to avoid possible conflict outside G-Golf[5], then the name starts using the `gi-` prefix, and equally for variables, using `%gi-`.

- **Types and Values**

G-Golf variables that bind Glib, GObject, Gdk and GI types and values use the same convention as for procedures, except that they always start with `%` and their original type names are transformed by the same rules that those applied when calling [g-studly-caps-expand], page 92.

For example, from the `GIBaseInfo` section:

```
GIInfoType
->
[%gi-info-type], page 54
```

GOOPS Notes and Conventions

G-Golf extensively uses GOOPS, the Guile Object Oriented System (see Section "GOOPS" in *The GNU Guile Reference Manual*), in a way that is largely inspired by Guile-Gnome (`https://www.gnu.org/software/guile-gnome`).

Here are some notes and the GOOPS conventions used by G-Golf.

- **Slots are not Immutable**

Except for virtual slots, there is currently no way to effectively prohibit (block) a user to mutate a goops class instance (one can always use `slot-set! instance slot-name value`)[6].

However, you will find a few places in this manual using phrase excerpts like '`instances of this <class> are immutable`', or '`this <slot> is immutable`'. In these contexts, what is actually meant is that these (insances or slots) are not meant to be mutated. Doing so is not only at your own risks, but likely to cause a crash.

- **Merging Generics**

---

[5] As an example, it would not be a good idea to use (the name) `import` for the G-Golf procedure that reads and build the interface for a `GIR` library, since it is an R6RS reserved word.

[6] Actually, to be complete, there is a way, which is to define the slot using `#:class <read-only-slot>`, but (a) it is undocumented and (b), it requires the use use of libguile to initialize the slot value, something that I don't wan't to do in G-Golf. If you are interested by this (undocumented) feature for your own project though, I suggest you look for some exmples in the Guile-Gnome (`https://www.gnu.org/software/guile-gnome`), source tree, where it is extensively used.

In G-Golf, generic functions are always merged (see Section "Merging Generics" in *The GNU Guile Reference Manual*).

Users are (highly) recommended to do the same, in their `repl`, application/library modules and script(s). In its modules - those that import (oop goops) - G-Golf uses the following duplicate binding handler set:

```
  #:duplicates (merge-generics
replace
warn-override-core
warn
last)
```

In a `repl` or in scripts, these maybe set - after importing (oop goops) - by calling `default-duplicate-binding-handler`:

```
(use-modules (oop goops))

(default-duplicate-binding-handler
    '(merge-generics replace warn-override-core warn last))
```

- **Accessors Naming Convention**

In G-Golf, all slots define an accessor (and no getter, no setter), the name of which is the `slot-name` prefixed using `!`. For example:

```
(define-class <gtype-class> (<class>)
  (info #:accessor !info
        #:init-keyword #:info)
  ...)
```

The principal reasons are (not in any particular order):

- It is a good idea, we think, to be able to visually (and somehow immediately) spot and distinct accessors from the rest of the scheme code your are looking at or working on.
- Accessors are exported, and with this convention, we almost certainly avoid all '`name clashes`' with user namespaces, that otherwise would be extremelly frequent[7].
- Users quite often want or even need to cash slot values in a closure. By using this `!` prefixing convention, we leave users with the (quite usefull) possibility to name their local variables using the respective slot names.
- Accessors may always be used to mutate a slot value (except for virtual slots, for which you can '`block`' that feature), like in `(set! (!name an-actor) "Mike")`. In scheme, it is a tradition to signal mutability by postfixing the procedure name using the `!` character.
- Accessors are not procedures though, there are methods, and to effectively mutate a slot value, one must use `set!`. Therefore, prefixing makes sence (and preserves the first reason announced here, where posfixing would break it).

[7] Slot names tends to be extremelly common, like `name`, `color`, ... and naming their respective accessor using the slot name would very likely provoque numerous name clashes with user variables, procedures and methods names.

- We should also add that we are well aware that Java also prefixes its accessors, using a `.` as its prefix character, but GOOPS is radically different from Java in its design, and therefore, we really wanted another character.

Customizing G-Golf

There are three '`domains`' for which G-Golf offers a series of customization variables. The first is related to '`name transformation`', or how things are being named as they are being imported. The second is related to the so called '`short name methods`', wether G-Golf should create them. The third is related to the so called '`syntax name protect`' mechanism, or how G-Golf should address syntax name '`clash`' exceptions, if/when short name methods are created.

Variables listed in this subsection of the manual are somewhat briefly introduced, make sure to follow each cross-reference to also read their reference documentation entry.

- **Name Transformation**

As stated previously (See [Naming Conventions], page 5), when G-Golf imports a Gnome library, its classes, properties, methods, functions, types and constant are renamed, mainly to avoid '`Camel Case (https://en.wikipedia.org/wiki/Camel_case)`', to surround class names by '`<`' '`>`' and to avoid '`_`' (underscore) and use '`-`' (hyphen) instead.

G-Golf provides two variables, that may be used to fully or partially customize the name transformation output:

```
[%g-name-transform-exceptions], page 94
[%g-studly-caps-expand-token-exceptions], page 94
```

Here is a summary of how name transformation happens and how the above two variables are used:

- Class names are obtained by calling [g-name->class-name], page 92, which calls [g-name->name], page 92,

- [g-name->name], page 92, first checks if its argument has an entry in [%g-name-transform-exceptions], page 94, and returns its value if it found one, otherwise, it calls [g-studly-caps-expand], page 92,

- [g-studly-caps-expand], page 92, which does the core of the job, uses [%g-studly-caps-expand-token-exceptions], page 94, to specially treat its listed token exceptions.

- **Short Name Methods**

By default, when G-Golf imports a GI typelib, it creates so called '`short name methods`', obtained by dropping the container name (and its trailing hyphen) from the GI typelib method full/long names. Users may change this default, by setting the following variable:

```
[%gi-method-short-names-skip], page 94
```

- **Syntax Name Protect**

When a short name method is created, which is obtained by dropping the container name (and its trailing hyphen) from the GI typelib method full/long name, it may lead to a so called name '`clash`', with an already defined procedure or syntax. Name '`clashes`' against procedures are not a concern (this is explained in the [syntax-name->method-name], page 93, documentation entry).

However, the '`magic`' applied for name '`clashes`' against procedures can not work for syntaxes, and those syntax names must be '`protected`', which is achieved by (automatically) calling [syntax-name->method-name], page 93, (users should normally not call this procedure themselves - except for testing purposes).

Three variables are provided to customize the '`Syntax Name Protect`' default mechanism:

```
[%syntax-name-protect-prefix], page 94
[%syntax-name-protect-postfix], page 94
[%syntax-name-protect-renamer], page 94
```

Getting Started with G-Golf

G-Golf will let you import and work with any GObject-Introspectable GNOME library, called `Typelib`[8]. Since we need to make a choice among so many, to guide new comers and let them started with G-Golf, let's pick-up Gtk (`https://developer.gnome.org/gtk3/stable/`), and show how to Create interfaces that users just love (`https://gtk.org/`).

Following the tradition, we will first look at and work on a couple of versions of the often seen '`Hello World!`' familiar, friendly greeting program.

We will then complete the section by walking through the necessary and/or recommended steps to build applications,

Hello World!

Following the tradition, let's first see how the often seen '`Hello World!`' familiar, minimal, friendly greeting program looks like in G-Golf:

```
;; Load Gtk
(use-modules (g-golf))
(gi-import "Gtk")

;; When the application is launched..
(define (activate app)
  ;; - Create a new window and a new button
  (let ((window (make <gtk-application-window>
                  #:title "Hello"
                  #:application app))
        (button (make <gtk-button>
                  #:label "Hello, World!")))
    ;; - Which closes the window when clicked
    (connect button
```

[8] A `Typelib` is a binary, readonly, memory-mappable database containing reflective information about a GObject library

```
                'clicked
                (lambda (b)
                  (close window)))
        (add window button)
        (show-all window)))

;; Create a new application
(let ((app (make <gtk-application>
             #:application-id "com.example.GtkApplication")))
  (connect app 'activate activate)
  ;; Run the application
  (run app 0 '()))
```

Providing you successfully installed G-Golf, you may run the above code in a Guile REPL (Read Evaluate Print Loop), which as described in its comments, starts the application, resulting in opening a (small) window named '`Hello`', with one button named '`Hello, World!`', that will close the widow when clicked.

Example 1

Wonderful! But you probably rightfully think that it was a bit slow. This is not because G-Golf nor Guile are slow, but because the `Gtk` namespace is absolutely huge, and although we only use a few components, we asked to import the all namespace. We will see how to only selectively import the namespace components we need in a moment, but let's first try the following, (a) close the window and (b) re-evaluate the last expression:

```
(let ((app (make <gtk-application>
             #:application-id "com.example.GtkApplication")))
  (connect app 'activate activate)
  (run app 0 '()))
```

Great! Now, the application was launched instantaneously. Since everything it needs was already imported, the time it takes to execute the code is nearly identical to the time it would take to execute the same code from C - if you accurately measure the execution time in both situation, you would see a difference in the results, but small enough that it is safe to declare it imperceptible.

- **Selective Import**

To selectively import namespace components, use [gi-import-by-name], page 109, which takes two arguments, a namespace and a (component) name. Let's try on our minimal '`Hello World!`' example and see how it goes. All we need to do, is to substitute the `(gi-import "Gtk")` call by the following expression:

```
(for-each (lambda (name)
            (gi-import-by-name "Gtk" name))
    '("Application"
      "ApplicationWindow"
      "Button"))
```

With this change, everything else kept equal, if you (quit and) restart Guile, evaluate the updated '`Hello World!`' example code, you will notice how the elapse time before the application window appears is now substantially reduced, compared to the version that inport the all `Gtk` namespace.

Substantially reduced but not instantaneous: that is expected, although we only import a few `Gtk` namespace components, three GObject classes in this example, G-Golf will import those classes, their interface(s) if any, methods, enums, flags ... and do the same for their parent class, recursively - so, even for a tiny '`Hello World!`' example, G-Golf has to import (and dynamically define) tens of classes, interfaces, enums, flags . . . as well as hundreds of methods and procedures . . .

G-Golf Cache - Accessing

G-Golf Cache - Accessing.

Procedures

[gi-cache-show], page 12
[gi-cache-ref], page 12

Variables

[%gi-cache], page 13

Description

G-Golf has and uses a cache '`mechanism`' - actually several, but only one is (partially) exposed to users (and with reserves, see below), also referred to as G-Golf `main cache` - not only for internal needs, but also to avoid reconstructing things '`on-the-fly`' unnecessarily, such as already imported [<gi-enum>], page 86, [<gi-flag>], page 87, and [<gi-struct>], page 88, instances.

G-Golf `main cache` exposed functionality is '`access only`' - users should not (never) attempt to change its content - and its design is not (yet) '`set in stone`', so interfaces here exposed, may (have to be) change(d).

So, keeping the above reserves in mind, G-Golf `main cache` current data structure is composed of two nested association lists, to which we refer using *m-key* (main key) and *s-key* (secondary key).

Procedures

`gi-cache-show` [*m-key #f*] [Procedure]
Returns nothing.

Displays the content of G-Golf main cache. If *m-key* (main key) is `#f` (the default), it displays the list of the main keys present in the cache. Otherwise, it retrieves the content of the main cache for *m-key* and displays its content if any, or `-- is empty --` if none.

`gi-cache-ref` *m-key s-key* [Procedure]
Returns a [%gi-cache], page 13, entry or `#f`.

Obtains and returns the [%gi-cache], page 13, entry for *m-key* and *s-key*, or `#f` if none is found.

Remember that you may (always) view the list of main and secondary key names (which is '`dynamic`', depending on what you have imported) by calling [gi-cache-show], page 12, (without or with an *m-key* arg appropriately), but as a user, the two most important *m-key* are `'enum` and `'flag`, so you may check their member names, or bind their instance locally.

Main key names are given by G-Golf. Secondary key names are always the result of calling [g-name->name], page 92, upon the '`object`' original name. For example, let's retreive and visualize the content of the `GdkEventType` (enum) type (which is pre-imported in G-Golf):

```
,use (g-golf)
(gi-cache-ref 'enum 'gdk-event-type)
⊣
$2 = #<<gi-enum> 55a9665d9e10>

(describe $2)
#<<gi-enum> 55a9665d9e10> is an instance of class <gi-enum>
Slots are:
     enum-set = ((nothing . -1) (delete . 0) (destroy . 1) (expose . 2) (motion-n
     g-type = #f
     g-name = "GdkEventType"
     name = gdk-event-type
```

Variables

`%gi-cache` [Variable]

Holds a reference the the G-Golf `main cache`, which as said earlier, currently is composed of two nested association lists.

III. G-Golf Core Reference

Overview

Project and Documentation Structure

The project and documentation both structure and naming is, whenever it is possible, based on the '`original`' documentation structure and naming of the corresponding library. So far, these are the `glib` (GLib), `gobject` (GObject), `gdk` (Gdk) and `gi` (GObject Introspection) directories. To illustrate, here are just a few examples:

Glib

`Memory Allocation`
`(https://developer.gnome.org/glib/stable/glib-Memory-Allocation.html)`
[Memory Allocation], page 15,
`(g-golf glib mem-alloc)`

Exceptions to the above are the G-Golf modules that provide support to the core project itself or additional functionality, so far organized in their respective `support`, `override` and `hl-api` directory.

Glib

G-Golf Glib modules are defined in the `glib` subdirectory, such as `(g-golf glib main-event-loop)`.

Where you may load these modules individually, the easiest way to use G-Golf Glib is to import its main module, which imports and re-exports the public interface of (oop goops), (system foreign), all G-Golf support and G-Golf Glib modules:

```
(use-modules (g-golf glib))
```

G-Golf Glib low level API modules correspond to a Glib section, though they might be some exception in the future.

Memory Allocation

G-Golf Glib Memory Allocation low level API.
Memory Allocation — general memory-handling

Procedures

Description

These functions provide support for allocating and freeing memory.

Please read the Memory Allocation (`https://developer.gnome.org/glib/stable/glib-Memory-Allocatio`
section from the Glib reference manual for a complete description.

Procedures

`g-malloc` *n-bytes* [Procedure]
`g-malloc0` *n-bytes* [Procedure]

Returns a pointer to the allocated memory, or `#f`.

Allocates *n-bytes* of memory. If *n-bytes* is `0` it returns `#f`. When using `g-malloc0`, the allocated memory is initialized to `0`.

`g-free` *mem* [Procedure]

Returns nothing.

Frees the memory pointed to by mem.

`g-memdup` *mem n-bytes* [Procedure]

Returns a pointer to the allocated memory, or `#f`.

Allocates *n-bytes* of memory and copies *n-bytes* into it from *mem*. If *mem* is the `%null-pointer` or *n-bytes* is `0` it returns `#f`.

The Main Event Loop

G-Golf Glib Main Event Loop low level API.
The Main Event Loop — manages all available sources of events

Procedures

Description

The main event loop manages all the available sources of events for GLib and GTK+ applications. These events can come from any number of different types of sources such as file descriptors (plain files, pipes or sockets) and timeouts. New types of event sources can also be added using `g-source-attach`.

Please read The Main Event Loop (`https://developer.gnome.org/glib/stable/glib-The-Main-Event-Loo`
section from the Glib reference manual for a complete description.

Procedures

Note: in this section, the *loop*, *context* and *source* arguments are [must be] pointers to a `GMainLoop`, a `GMainContext` and a *GSource* respectively.

`g-main-loop-new` [*context #f*] [*is-running? #f*] [Procedure]

Returns a pointer to a new `GMainLoop`.

Creates a new `GMainLoop` structure.

The *context* must be a pointer to a `GMainContext` of `#f`, in which case the default context is used. When *is-running?* is `#t`, it indicates that the loop is running. This is not very important since calling `g-main-loop-run` will set this to `#t` anyway.

`g-main-loop-ref` *loop* [Procedure]
Returns loop.

Increases the *loop* reference count by one.

`g-main-loop-unref` *loop* [Procedure]
Returns nothing.

Decreases the *loop* reference count by one. If the result is zero, free the loop and free all associated memory.

`g-main-loop-run` *loop* [Procedure]
Returns nothing.

Runs a main loop until [g-main-loop-quit], page 17, is called on the *loop*. If this is called for the thread of the loop's `GMainContext`, it will process events from the *loop*, otherwise it will simply wait.

`g-main-loop-quit` *loop* [Procedure]
Returns nothing.

Stops a `GMainLoop` from running. Any calls to [g-main-loop-run], page 17, for the *loop* will return.

Note that sources that have already been dispatched when `g-main-loop-quit` is called will still be executed.

`g-main-context-new` [Procedure]
Returns a pointer.

Creates and returns a (pointer to a) new `GMainContext` structure.

`g-main-context-default` [Procedure]
Returns a pointer.

Returns the global default main context. This is the main context used for main loop functions when a main loop is not explicitly specified, and corresponds to the '`main`' main loop.

`g-timeout-source-new` *interval* [Procedure]
Returns a pointer.

Creates and returns (a pointer to) a new (timeout) `GSource`.

The source will not initially be associated with any `GMainContext` and must be added to one with [g-source-attach], page 18, before it will be executed.

The timeout *interval* is in milliseconds.

`g-timeout-source-new-seconds` *interval* [Procedure]
Returns a pointer.

Creates and returns (a pointer to) a new (timeout) `GSource`.

The source will not initially be associated with any `GMainContext` and must be added to one with [g-source-attach], page 18, before it will be executed.

The timeout *interval* is in seconds.

`g-idle-source-new` [Procedure]
Returns a pointer.

Creates and returns (a pointer to) a new (idle) `GSource`.

The source will not initially be associated with any `GMainContext` and must be added to one with [g-source-attach], page 18, before it will be executed. Note that the default priority for idle sources is `200`, as compared to other sources which have a default priority of `300`.

`g-source-ref-count` *source* [Procedure]
Returns an integer.

Obtains and returns the reference count of *source*.

`g-source-ref` *source* [Procedure]
Returns *source*.

Increases the *source* reference count by one.

`g-source-unref` *source* [Procedure]
Returns nothing.

Decreases the *source* reference count by one. If the resulting reference count is zero the source and associated memory will be destroyed.

`g-source-free` *source* [Procedure]
Returns nothing.

Calls [g-source-destroy], page 18, and decrements the reference count of *source* to 0 (so *source* will be destroyed and freed).

`g-source-attach` *source context* [Procedure]
Returns an integer.

Adds *source* to *context* so that it will be executed within that context.

Returns the ID (greater than 0) for the *source* within the *context*.

Remove it by calling [g-source-destroy], page 18.

`g-source-destroy` *source* [Procedure]
Returns nothing.

Removes *source* from its `GMainContext`, if any, and mark it as destroyed. The source cannot be subsequently added to another context. It is safe to call this on sources which have already been removed from their context.

This does not unref *source*: if you still hold a reference, use g-source-unref to drop it.

`g-source-is-destroyed?` *source* [Procedure]
Returns #t if *source* has been destroyed. Otherwise, it returns #f.

Once a source is destroyed it cannot be un-destroyed.

`g-source-set-priority` *source priority* [Procedure]
Returns nothing.

Sets the *source* priority. While the main loop is being run, a source will be dispatched if it is ready to be dispatched and no sources at a higher (numerically smaller) priority are ready to be dispatched.

A child source always has the same priority as its parent. It is not permitted to change the priority of a source once it has been added as a child of another source.

`g-source-get-priority` *source priority* [Procedure]
Returns an integer.

Obtains and returns the *source* priority.

`g-source-remove` *id* [Procedure]
Returns #t.

Removes the source with the given *id* from the default main context. You must use [g-source-destroy], page 18, for sources added to a non-default main context.

It is an error to attempt to remove a non-existent source.

Source IDs can be reissued after a source has been destroyed. This could lead to the removal operation being performed against the wrong source, unless you are cautious.

For historical reasons, this procedure always returns #t.

IO Channels

G-Golf Glib IO Channels low level API.
IO Channels — portable support for using files, pipes and sockets

Procedures

Types and Values

Description

The `GIOChannel` data type aims to provide a portable method for using file descriptors, pipes, and sockets, and integrating them into the main event loop. Currently, full support is available on UNIX platforms, support for Windows is only partially complete.

Please read the IO Channels (`https://developer.gnome.org/glib/stable/glib-IO-Channels.html`) section from the Glib reference manual for a complete description.

Procedures

Note: in this section, the *fd*, *channel* and *condition* arguments are [must be] respectively an integer (a '`valid`' file descriptor), a pointer to a `GIOChannel` and a list of one or more [%g-io-condition], page 20, flags.

`g-io-channel-unix-new` *fd* [Procedure]
Returns a pointer.

Creates and returns a pointer to a new `GIOChannel` for *fd* (file descriptor). On UNIX systems this works for plain files, pipes, and sockets.

The newly created `GIOChannel` has a reference count of 1.

The default encoding for `GIOChannel` is UTF-8. If your application is reading output from a command using via pipe, you may need to set the encoding to the encoding of the current locale (FIXME - still missing a binding to g_io_channel_set_encoding).

`g-io-channel-ref` *channel* [Procedure]
Returns *channel*.

Increments the *channel* reference count.

`g-io-channel-unref` *channel* [Procedure]
Returns nothing.

Decrements the *channel* reference count.

`g-io-create-watch` *channel condition* [Procedure]
Returns a pointer.

Creates and returns a pointer to a `GSource` that's dispatched when condition is met for the given *channel*. For example, if condition is `'(in)`, the source will be dispatched when there's data available for reading.

Types and Values

`%g-io-condition` [Instance Variable of `<gi-flag>`]
An instance of `<gi-flag>`, who's members are the scheme representation of the `GIOCondition` flags:

g-name: GIOCondition
name: gio-condition
enum-set:

`in`	There is data to read.
`out`	Data can be written (without blocking).
`pri`	There is urgent data to read.
`err`	Error condition.
`hup`	Hung up (the connection has been broken, usually for pipes and sockets).
`nval`	Invalid request. The file descriptor is not open.

UNIX-specific utilities and integration

G-Golf Glib UNIX-specific utilities and integration low level API.
UNIX-specific utilities and integration — pipes, signal handling.

Procedures

Description

Most of GLib is intended to be portable; in contrast, this set of functions is designed for programs which explicitly target UNIX, or are using it to build higher level abstractions which would be conditionally compiled if the platform matches G_OS_UNIX.

Procedures

Note: in this section, the *fd* and *condition* arguments are [must be] respectively an integer (a 'valid' file descriptor) and a list of one or more [%g-io-condition], page 20, flags.

`g-unix-fd-source-new` *fd condition* [Procedure]

Returns a pointer.

Creates and returns a pointer to a new `GSource` to watch for a particular IO *condition* on *fd*.

The source will never close the file descriptor, you must do it yourself.

Doubly-Linked Lists

G-Golf Glib Doubly-Linked Lists low level API.
Doubly-Linked Lists — linked lists that can be iterated over in both directions

Procedures

Description

The `GList` structure and its associated functions provide a standard doubly-linked list data structure.

Each element in the list contains a piece of data, together with pointers which link to the previous and next elements in the list. Using these pointers it is possible to move through the list in both directions (unlike the singly-linked `GSList`, which only allows movement through the list in the forward direction).

Please read the Doubly-Linked-Lists (`https://developer.gnome.org/glib/stable/glib-Doubly-Linked-Li`
section from the Glib reference manual for a complete description.

Procedures

`g-list-data` *g-list* [Procedure]

Returns a pointer.

Obtains and returns a pointer to the data in *g-list*, or any integer value, in which case, it is the responsibility of the caller to apply the appropriate type conversion procedure.

`g-list-next` *g-list* [Procedure]
Returns a pointer or #f.

Obtains and returns the next element in *g-list*, or #f if there are no more elements.

`g-list-prev` *g-list* [Procedure]
Returns a pointer or #f.

Obtains and returns the previous element in *g-list*, or #f if there are no previous element.

`g-list-free` *g-list* [Procedure]
Returns nothing.

Frees all of the memory used by *g-list*.

`g-list-length` *g-list* [Procedure]
Returns an integer.

Obtains and returns the number of elements in *g-list*. This function iterates over the whole list to count its elements.

`g-list-nth-data` *g-list n* [Procedure]
Returns a pointer or #f.

Obtains and returns a pointer to the data of the *n*-th element of *g-list*. This iterates over the list until it reaches the n-th position. If *n* is off the end of *g-list*, it returns #f.

Singly-Linked Lists

G-Golf Glib Singly-Linked Lists low level API.
Singly-Linked Lists — Linked lists that can be iterated over in one direction

Procedures

Description

The `GSList` structure and its associated functions provide a standard singly-linked list data structure.

Each element in the list contains a piece of data, together with a pointer which links to the next element in the list. Using this pointer it is possible to move through the list in one

direction only (unlike the [Doubly-Linked Lists], page 21, which allow movement in both directions).

Please read the Singly-Linked-Lists (`https://developer.gnome.org/glib/stable/glib-Singly-Linked-Lis` section from the Glib reference manual for a complete description.

Procedures

`g-slist-data` *g-slist* [Procedure]
Returns a pointer.

Obtains and returns a pointer to the data in *g-slist*, or any integer value, in which case, it is the responsibility of the caller to apply the appropriate type conversion procedure.

`g-slist-next` *g-slist* [Procedure]
Returns a pointer or #f.

Obtains and returns the next element in *g-slist*, or #f if there are no more elements.

`g-slist-append` *g-slist data* [Procedure]
Returns a pointer.

Adds *data* - which is (must be) a pointer - to the end of *g-slist* and returns a pointer to the (possibly new) start of the list (so make sure you store the new value).

Note that [g-slist-append], page 23, has to traverse the entire list to find the end, which is inefficient when adding multiple elements. A common idiom to avoid the inefficiency is to prepend the elements and reverse the list when all elements have been added.

`g-slist-prepend` *g-slist data* [Procedure]
Returns a pointer.

Adds *data* - which is (must be) a pointer - to the start of *g-slist* and returns a pointer to the (possibly new) start of the list (so make sure you store the new value).

`g-slist-free` *g-slist* [Procedure]
Returns nothing.

Frees all of the memory used by *g-slist*.

`g-slist-length` *g-slist* [Procedure]
Returns an integer.

Obtains and returns the number of elements in *g-slist*. This function iterates over the whole list to count its elements.

`g-slist-nth-data` *g-slist n* [Procedure]
Returns a pointer or #f.

Obtains and returns a pointer to the data of the *n*-th element of *g-slist*. This iterates over the list until it reaches the n-th position. If *n* is off the end of *g-slist*, it returns #f.

Quarks

G-Golf Glib Quarks low level API.
Quarks — a 2-way association between a string and a unique integer identifier.

Procedures

Description

Quarks are associations between strings and integer identifiers. Given either the string or the `GQuark` identifier it is possible to retrieve the other.

Procedures

`g-quark-from-string` *str* [Procedure]
Returns an integer.

Obtains and returns the `GQuark` identifying the string given by *str*. If the string does not currently have an associated `GQuark`, a new `GQuark` is created, using a copy of the string.

`g-quark-to-string` *g-quark* [Procedure]
Returns a string.

Obtains and returns the string associated with the `GQuark` given by *g-quark*.

GObject

G-Golf GObject modules are defined in the `gobject` subdirectory, such as `(g-golf gobject enum-flags)`.

Where you may load these modules individually, the easiest way to use G-Golf is to import its main module, which imports and re-exports the public interface of (oop goops), (system foreign), all G-Golf support and G-Golf GObject modules:

```
(use-modules (g-golf gobject))
```

G-Golf GObject low level API modules correspond to a GObject section, though they might be some exception in the future.

Type Information

G-Golf GObject Type Information low level API.
Type Information — The GLib Runtime type identification and management system

Procedures

Types and Values

Object Hierarchy

gpointer
  +— GType

Description

The `GType API` is the foundation of the GObject system. It provides the facilities for registering and managing all fundamental data types, user-defined object and interface types.

Please read the Type Information (`https://developer.gnome.org/gobject/stable/gobject-Type-Informa` section from the GObject reference manual for a complete description.

Procedures

`g-type->symbol` *g-type* [Procedure]

Returns a symbol.

Get the symbol that correspond to the type ID *g-type*. Note that this function (like all other GType API) cannot cope with invalid type IDs. It accepts validly registered type ID, but randomized type IDs should not be passed in and will most likely lead to a crash.

`symbol->g-type` *symbol* [Procedure]

Returns a type ID.

Get the type ID for *symbol*. Note that this function (like all other GType API) cannot cope with invalid type ID symbols. It accepts validly registered type ID symbol, but randomized type IDs should not be passed in and will most likely lead to a crash.

`g-type-name` *g-type* [Procedure]

Returns a string.

Get the unique name that is assigned to *g-type*, a type ID. Note that this function (like all other GType API) cannot cope with invalid type IDs. It accepts validly

registered type ID, but randomized type IDs should not be passed in and will most likely lead to a crash.

`g-type-is-a` *g-type is-a-g-type* [Procedure]
Returns #t if *g-type* is a *is-a-g-type*.

If *is-a-g-type* is a derivable type, check whether *g-type* is a descendant of *is-a-g-type*. If *is-a-g-type* is an interface, check whether *g-type* conforms to it.

`g-type-class-ref` *g-type* [Procedure]
Returns a pointer.

Obtains and returns a pointer to the `GTypeClass` structure for *g-type* (a GObject class GType). The reference count of the class is incremented, and the class is '`created`' (instanciated) if/when it doesn't exist already.

`g-type-class-peek` *g-type* [Procedure]
Returns a pointer.

Obtains and returns a pointer to the `GTypeClass` structure for *g-type* (a GObject class GType). The reference count of the class isn't incremented. As a consequence, this function may return `#f` - if the class of the type passed in does not currently exist (hasn't been referenced before).

`g-type-class-unref` *g-class* [Procedure]
Returns nothing.

Decrements the reference count for *g-class* (a pointer to a `GTypeClass` structure). Once the last reference count of a class has been released, it may be finalized by the type system. Attempting to further dereference a finalized class is invalid.

`g-type-fundamental` *g-type* [Procedure]
Returns a type ID.

Extracts the fundamental type ID portion for *g-type*.

`g-type-ensure` *g-type* [Procedure]
Returns nothing.

Ensures that the indicated *g-type* has been registered with the type system, and that its `_class_init` method has been run.

Types and Values

`%g-type-fundamental-flags` [Instance Variable of `<gi-enum>`]
Bit masks used to check or determine specific characteristics of a fundamental type.

An instance of `<gi-enum>`, who's members are the scheme representation of the `GTypeFundamentalFlags`:

g-name: GTypeFundamentalFlags
name: g-type-fundamental-flags
enum-set:

`classed` Indicates a classed type

`instantiable`
: Indicates an instantiable type (implies classed)

`derivable`
: Indicates a flat derivable type

`deep-derivable`
: Indicates a deep derivable type (implies derivable)

`%g-type-fundamental-types` [Instance Variable of `<gi-enum>`]

An instance of `<gi-enum>`, who's members are the scheme representation of the `GType` obtained from the fundamentl types defined using `G_TYPE_MAKE_FUNDAMENTAL`, which starts with `G_TYPE_INVALID` and ends with `G_TYPE_OBJECT`.

g-name: #f[9]

name: g-type-fundamental-types
enum-set:

`invalid` An invalid GType used as error return value in some functions which return a GType.

`none` A fundamental type which is used as a replacement for the C void return type.

`interface`
: The fundamental type from which all interfaces are derived.

`char` The fundamental type corresponding to gchar. It is unconditionally an 8-bit signed integer. This may or may not be the same type a the C type "gchar".

`uchar` The fundamental type corresponding to guchar.

`boolean` The fundamental type corresponding to gboolean.

`int` The fundamental type corresponding to gint.

`uint` The fundamental type corresponding to guint.

`long` The fundamental type corresponding to glong.

`ulong` The fundamental type corresponding to gulong.

`int64` The fundamental type corresponding to gint64.

`uint64` The fundamental type corresponding to guint64.

`enum` The fundamental type from which all enumeration types are derived.

`flags` The fundamental type from which all flags types are derived.

`float` The fundamental type corresponding to gfloat.

[9] There is no corresponding `enum` in GOject. These fundamental types (in GObject) are defined using a macro, `G_TYPE_MAKE_FUNDAMENTAL`, that applies bitwise arithmetic shift given by `G_TYPE_FUNDAMENTAL_SHIFT` (which we also have to apply, to get to the type ID for the fundamental number `x`).

`double`	The fundamental type corresponding to gdouble.
`string`	The fundamental type corresponding to nul-terminated C strings.
`pointer`	The fundamental type corresponding to gpointer.
`boxed`	The fundamental type from which all boxed types are derived.
`param`	The fundamental type from which all [GParamSpec], page 35, types are derived.
`object`	The fundamental type for [GObject_], page 28.

GObject

G-Golf GObject low level API.
GObject — The base object type

Procedures

Object Hierarchy

GObject
+— GBinding
+— GInitiallyUnowned
+— GTypeModule

Description

`GObject` is the fundamental type providing the common attributes and methods for all object types in `GTK+`, `Pango` and other libraries based on GObject. The GObject class provides methods for object construction and destruction, property access methods, and signal support.

Please read the GObject (`https://developer.gnome.org/gobject/stable/gobject-The-Base-Object-Typ` section from the GObject reference manual for a complete description.

Procedures

Note: in this section, unless otherwise specified, the *object* argument is [must be] a pointer to a `GObject` (instance).

`g-object-new` *gtype* [Procedure]
Returns a pointer.

Creates and returns a (pointer to) a new instance of a GObject subtype *gtype*. All properties are set to there default values.

`g-object-new-with-properties` *gtype n-prop names g-values* [Procedure]
Returns a pointer.

Creates and returns a (pointer to) a new instance of a GObject subtype *gtype*. The other arguments are *n-prop* the number of properties, *names* a pointer to an array of pointers to strings with the names of each property to be set and *values* an array of `GValue` containing the values of each property to be set.

Properties that are not explicitly specified are set to there default values.

`g-object-ref` *object* [Procedure]
Returns a pointer.

Increases the reference count of *object*.

`g-object-unref` *object* [Procedure]
Returns nothing.

Decreases the reference count of *object*. When its reference count drops to 0, the object is finalized (i.e. its memory is freed).

If the pointer to the GObject may be reused in future (for example, if it is an instance variable of another object), it is recommended to clear the pointer to NULL rather than retain a dangling pointer to a potentially invalid GObject instance. Use `g-clear-object` for this.

`g-object-ref-sink` *object* [Procedure]
Returns a pointer.

If *object* has a floating reference, then this call '`assumes ownership`' of the floating reference, converting it to a normal reference by clearing the floating flag while leaving the reference count unchanged.

If *object* is not floating, then this call adds a new normal reference increasing the reference count by one.

`g-object-ref-count` *object* [Procedure]
Returns an integer.

Obtains and returns the (public GObject struct field) `ref_count` value for *object*.

`g-object-is-floating` *object* [Procedure]
Returns `#t` if *object* has a floating reference, otherwise it returns `#f`.

`g-object-type` *object* [Procedure]
Returns the *GType* (the type id) for *object*.

`g-object-type-name` *object* [Procedure]
Returns the *GType* name for *object*.

`g-object-get-property` *object property* [*g-type #f*] [Procedure]
Returns the *property* value for *object*.

The *property* argument is (must be) a pointer to a valid `GIPropertyInfo` (*property* must point to one of the properties infos of the class of *object*). The *g-type* argument must be a valid `GType` value. If `#f`, which is the default, [gi-property-g-type], page 77, is called.

`g-object-set-property` *object property value* [*g-type #f*] [Procedure]
Returns *value*.

Sets the *object property* to *value*. The *property* argument is (must be) a pointer to a valid `GIPropertyInfo` (*property* must point to one of the properties infos of the class of *object*). The *g-type* argument must be a valid `GType` value. If `#f`, which is the default, [gi-property-g-type], page 77, is called.

Enumeration and Flag Types

G-Golf GObject Enumeration and Flag Types low level API.
Enumeration and Flag Types — Enumeration and flags types.

Description

The GLib type system provides fundamental types for enumeration and flags types. (Flags types are like enumerations, but allow their values to be combined by bitwise or). A registered enumeration or flags type associates a name and a nickname with each allowed value. When an enumeration or flags type is registered with the GLib type system, it can be used as value type for object properties.

Generic Values

G-Golf GObject Generic Values low level API.
Generic values — A polymorphic type that can hold values of any other type.

Procedures

Object Hierarchy

GBoxed
+— GValue

Description

The `GValue` structure is basically a variable container that consists of a type identifier and a specific value of that type. The type identifier within a GValue structure always determines the type of the associated value. To create a undefined GValue structure, simply call [g-value-new], page 31, which create a zero-filled GValue structure. To create and initialize a GValue, use the [g-value-init], page 31, procedure. A GValue cannot be used until it is

initialized. The basic type operations (such as freeing and copying) are determined by the `GTypeValueTable` associated with the type ID stored in the GValue.

Please read the Generic Values (`https://developer.gnome.org/gobject/stable/gobject-Generic-Values` section from the GObject reference manual for a complete description.

Procedures

`g-value-size` [Procedure]

Returns an integer.

Obtains and returns the size of a `GValue`.

`g-value-new` [Procedure]

Returns a pointer to a `GValue`.

Creates and returns (a pointer to) an empty (uninitialized) `GValue`.

`g-value-init` *g-type* [Procedure]

Returns a pointer to a `GValue`.

Creates and initializes a `GValue` with the default value for *g-type*, which can either be an integer - a `GType` static or dynamic value, or a symbol - a member of the [%g-type-fundamental-types], page 27.

`g-value-unset` *g-value* [Procedure]

Returns nothing.

Clears the current value in *g-value* (if any) and '`unsets`' the type. This releases all resources associated with *g-value*. An unset `GValue` is the same as an uninitialized (zero-filled) `GValue` structure.

Parameters and Values

G-Golf GObject Parameters and Values low level API.
Parameters and Values — Standard Parameter and Value Types

Procedures

Description

`GValue` provides an abstract container structure which can be copied, transformed and compared while holding a value of any (derived) type, which is registered as a GType with a GTypeValueTable in its GTypeInfo structure. Parameter specifications for most value types can be created as `GParamSpec` derived instances, to implement e.g. `GObject properties` which operate on GValue containers.

Parameter names need to start with a letter (a-z or A-Z). Subsequent characters can be letters, numbers or a '-'. All other characters are replaced by a '-' during construction.

Procedures and Methods

Note: in this section, the *g-value* argument is [must be] a pointer to a `GValue`.

`g-value-type` *g-value* [Procedure]
`g-value-type-tag` *g-value* [Procedure]
`g-value-type-name` *g-value* [Procedure]
Returns an integer, a symbol or a string, respectively.

Obtains and returns the `GType`, the `GType` tag (see [%g-type-fundamental-types], page 27) or the `GType` name (see [g-type-name], page 25, for *g-value*, respectively.

`g-value-ref` *g-value* [Procedure]
Returns the content of *g-value*.

Obtains and returns the content of *g-value*. Supported `GType` (their scheme representaion) for *g-value* are: `boolean`, `uint`, `int`, `float`, `double`, `enum`, `flags`, `string`, `boxed`, `pointer`, `object`, `interface`.

`g-value-set!` *g-value value* [Procedure]
Returns nothing.

Sets the content of *g-value* to *value*. Supported `GType` (their scheme representaion) for *g-value* are: `boolean`, `uint`, `int`, `float`, `double`, `enum`, `flags`, `string`, `boxed`, `pointer`, `object`, `interface`.

Note that this procedure cannot cope with invalid values (the type of *value* must correspond to the `GType` for *g-value*, otherwise it will most likely lead to a crash.

`g-value-get-boolean` *g-value* [Procedure]
Returns `#t` or `#f`.

Obtains the content of *g-value* and returns `#f` if it is `0`, otherwise it returns `#t`.

`g-value-set-boolean` *g-value val* [Procedure]
Returns nothing.

Sets the content of *g-value* to `0` if *val* is `#f`, otherwise sets the content to `1`.

`g-value-get-int` *g-value* [Procedure]
Returns a integer.

Obtains and returns the content of *g-value*.

`g-value-set-int` *g-value int* [Procedure]
Returns nothing.

Sets the content of *g-value* to *int*.

`g-value-get-uint` *g-value* [Procedure]
Returns an unsigned integer.

Obtains and returns the content of *g-value*.

`g-value-set-uint` *g-value uint* [Procedure]
Returns nothing.

Sets the content of *g-value* to *uint*.

`g-value-get-float` *g-value* [Procedure]
Returns a float.

Obtains and returns the content of *g-value*.

`g-value-set-float` *g-value float* [Procedure]
Returns nothing.

Sets the content of *g-value* to *float*.

`g-value-get-double` *g-value* [Procedure]
Returns a double.

Obtains and returns the content of *g-value*.

`g-value-set-double` *g-value double* [Procedure]
Returns nothing.

Sets the content of *g-value* to *double*.

`g-value-get-enum` *g-value* [Procedure]
Returns a symbol.

Obtains and returns the (registered) enum type info symbol for *g-value*.

`g-value-set-enum` *g-value* (*id* <*integer*>) [Method]
`g-value-set-enum` *g-value* (*sym* <*symbol*>) [Method]
Returns nothing.

Sets the content of *g-value* to *id*, or to the id corresponding to *sym* respectively. The *id* or the *sym* must be valid (as in being a valid member of the (registered) enum type info for *g-value*), otherwise an exception is raised.

`g-value-get-flags` *g-value* [Procedure]
Returns a list.

Obtains and returns the (registered) list of flags for *g-value*.

`g-value-set-flags` *g-value* (*val* <*integer*>) [Method]
`g-value-set-flags` *g-value* (*flags* <*list*>) [Method]
Returns nothing.

Sets the content of *g-value* to *val*, or to the value given by calling [gi-gflags->integer], page 88, upon the list of *flags*, respectively. The *val* or the *flags* must be valid (as in being a valid member of the (registered) gflags type for *g-value*), otherwise an exception is raised.

`g-value-get-string` *g-value* [Procedure]
Returns a string.

Obtains and returns the content of *g-value*.

`g-value-set-string` *g-value str* [Procedure]
Returns nothing.

Sets the content of *g-value* to *str*.

`g-value-get-boxed` *g-value* [Procedure]
Returns either a list of values, or a pointer.

Obtains and returns the content of *g-value*. If the boxed type [!is-opaque?], page 89, or [!is-semi-opaque?], page 89, it '`blindingly`' returns the boxed instance *g-value* pointer. Otherwise, the boxed instance is '`decoded`', and a list of its field values is returned.

`g-value-set-boxed` *g-value boxed* [Procedure]
Returns nothing.

Sets the content of *g-value* to *boxed*. If the boxed type [!is-opaque?], page 89, or [!is-semi-opaque?], page 89, then *boxed* is (supposed to be) a pointer, used to '`blindingly`' set *g-value*. Otherwise, the boxed instance is (supposed to be) a list of values, that are '`encoded`', and its (newly created) pointer is used to set *g-value*.

`g-value-get-pointer` *g-value* [Procedure]
Returns a pointer.

Obtains and returns the content of *g-value*.

`g-value-set-pointer` *g-value pointer* [Procedure]
Returns nothing.

Sets the content of *g-value* to *pointer*.

`g-value-get-object` *g-value* [Procedure]
Returns a pointer.

Obtains and returns the content of *g-value*.

`g-value-set-object` *g-value object* [Procedure]
Returns nothing.

Sets the content of *g-value* to *object* (a pointer to a `GObject` instance) and increases the *object* reference count.

GParamSpec

G-Golf GObject GParamSpec low level API.
GParamSpec — Metadata for parameter specifications.

Procedures

Types and Values

Description

`GParamSpec` is an object structure that encapsulates the metadata required to specify parameters, such as e.g. GObject properties.

Procedures

Note: in this section, the *p-spec* argument is [must be] a pointer to a `GParamSpec`.

`g-param-spec-type` *p-spec* [Procedure]
`g-param-spec-type-name` *p-spec* [Procedure]
Returns an integer or a (symbol) name, respectively.

Obtains and returns the `GType` or the `GType` (symbol) name for *p-spec*, respectively.

`g-param-spec-get-default-value` *p-spec* [Procedure]
Returns a pointer.

Obtains and returns the *p-spec* default value as pointer to a `GValue`, which will remain valid for the life of *p-spec* and must not be modified.

`g-param-spec-get-name` *p-spec* [Procedure]
`g-param-spec-get-nick` *p-spec* [Procedure]
`g-param-spec-get-blurb` *p-spec* [Procedure]
Returns a string.

Obtains and returns the name, nickname or short description for *p-spec*, respectively.

Types and Values

`%g-param-flags` [Instance Variable of `<gi-enum>`]
An instance of `<gi-enum>`, who's members are the scheme representation of the `GParamFlags`:

type-name: GParamFlags
name: g-param-flags
enum-set:

`readable` the parameter is readable

`writable` the parameter is writable

`readwrite`
alas for readable writable

`construct`
the parameter will be set upon object construction

`construct-only`
the parameter can only be set upon object construction

`lax-validation`
upon parameter conversion, strict validation is not required

`static-name`
the string used as name when constructing the parameter is guaranteed to remain valid and unmodified for the lifetime of the parameter. Since 2.8

`private` internal

`static-nick`
: the string used as nick when constructing the parameter is guaranteed to remain valid and unmmodified for the lifetime of the parameter. Since 2.8

`static-blurb`
: the string used as blurb when constructing the parameter is guaranteed to remain valid and unmodified for the lifetime of the parameter. Since 2.8

`explicit-notify`
: calls to `g_object_set_property` for this property will not automatically result in a '`notify`' signal being emitted: the implementation must call `g_object_notify` themselves in case the property actually changes. Since: 2.42

`deprecated`
: the parameter is deprecated and will be removed in a future version. A warning will be generated if it is used while running with `G_ENABLE_DIAGNOSTIC=1`. Since 2.26

Closures

G-Golf GObject Closures low level API.

Closures - Functions as first-class objects

Procedures

Object Hierarchy

GBoxed
+— GClosure

Description

A `GClosure` represents a callback supplied by the programmer. It will generally comprise a function of some kind and a marshaller used to call it. It is the responsibility of the marshaller to convert the arguments for the invocation from `GValues` into a suitable form,

perform the callback on the converted arguments, and transform the return value back into a `GValue`.

Please read the Closures (`https://developer.gnome.org/gobject/stable/gobject-Closures.html`) section from the GObject reference manual for a complete description.

Procedures

Note: in this section, the *closure*, *marshal*, *source* and *function* arguments are [must be] pointers to a `GClosure`, a `GSource`, a `GClosureMarshal` and a `GClosureNotify` respectively.

`g-closure-size` [Procedure]

Returns an integer.

Obtains and returns the size (the number of bytes) that a `GClosure` occupies in memory.

`g-closure-ref-count` *closure* [Procedure]

Returns an integer.

Obtains and returns the reference count of *closure*.

`g-closure-ref` *closure* [Procedure]

Returns a pointer.

Increments the reference count of *closure*, to force it staying alive while the caller holds a pointer to it.

`g-closure-sink` *closure* [Procedure]

Returns nothing.

Takes over the initial ownership of *closure*. Each closure is initially created in a '`floating`' state, which means that the initial reference count is not owned by any caller. [g-closure-sink], page 38, checks to see if the object is still floating, and if so, unsets the floating state and decreases the reference count. If the closure is not floating, [g-closure-sink], page 38, does nothing.

Because [g-closure-sink], page 38, may decrement the reference count of *closure* (if it hasn't been called on closure yet) just like [g-closure-unref], page 38, [g-closure-ref], page 38, should be called prior to this function.

`g-closure-unref` *closure* [Procedure]

Returns nothing.

Decrements the reference count of *closure* after it was previously incremented by the same caller. If no other callers are using *closure*closure, then it will be destroyed and freed.

`g-closure-free` *closure* [Procedure]

Returns nothing.

Decrements the reference count of *closure* to 0 (so *closure* will be destroyed and freed).

`g-closure-invoke` *closure return-value n-param param-vals invocation-hit* [Procedure]

Returns nothing.

Invokes the *closure*, i.e. executes the callback represented by the closure.

The arguments are *closure* (a pointer to a `GClosure`), *return-value* (a pointer to a `GValue`), *n-param* (the length of the param-vals array), *param-vals* (a pointer to an array of `GValue`) and *invocation-hint* (a context dependent invocation hint).

`g-closure-add-invalidate-notifier` *closure data function* [Procedure]
Returns nothing.

Registers an invalidation notifier which will be called when the closure is invalidated with `g-closure-invalidate`. Invalidation notifiers are invoked before finalization notifiers, in an unspecified order.

The *data* argumet is (must be) a pointer to the notifier data (or #f).

`g-closure-new-simple` *size data* [Procedure]
Returns a pointer.

Allocates a structure of the given *size* and initializes the initial part as a `GClosure`. The *data* (if any) are used to iitialize the data fields of the newly allocated `GClosure`.

The returned value is a floating reference (a pointer) to a new `GClosure`.

`g-closure-set-marshal` *closure marshal* [Procedure]
Returns nothing.

Sets the *closure* marshaller to *marshal*.

`g-source-set-closure` *source closure* [Procedure]
Returns nothing.

Set the *source* callback to *closure*.

If the source is not one of the standard GLib types, the `closure_callback` and `closure_marshal` fields of the `GSourceFuncs` structure must have been filled in with pointers to appropriate functions.

Signals

G-Golf GObject Signals low level API.
Signals — A means for customization of object behaviour and a general purpose notification mechanism

Procedures

`[g-signal-query]`, page 40
`[g-signal-lookup]`, page 40
`[g-signal-list-ids]`, page 40

Types and Values

`[%g-signal-flags]`, page 41

Description

The basic concept of the signal system is that of the emission of a signal. Signals are introduced per-type and are identified through strings. Signals introduced for a parent

type are available in derived types as well, so basically they are a per-type facility that is inherited.

Please read the Signals (`https://developer.gnome.org/gobject/stable/gobject-Signals.html`) section from the GObject reference manual for a complete description.

Procedures

`g-signal-query` *id* [Procedure]

Returns a list.

Obtains and returns a list composed of the signal id, name, interface-type[10], flags, return-type, number of arguments and their types. For example[11]:

```
,use (g-golf)
(gi-import "Clutter")

(make <clutter-actor>)
$2 = #<<clutter-actor> 565218c88a80>

(!g-type (class-of $2))
$3 = 94910597864000

(g-signal-list-ids $3)
$4 = (5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)

(g-signal-query 20)
$5 = (20 "enter-event" 94910597864000 (run-last) boolean 1 (boxed))
```

As you may have noticed, the signal query argument(s) list does not include the instance (and its type) upon which the signal is called, but both at C level and within the context of `GClosure`, callbacks must assume that the instance upon which a signal is called is always the first argument of the callback.

`g-signal-lookup` *name g-type* [Procedure]

Returns an integer.

Obtains and returns the signal's identifying integer, given the *name* of the signal and the object *g-type* it connects to. If a signal identifier can't be find for the given *name* and *g-type*, an exception is raised.

`g-signal-list-ids` *g-type* [Procedure]

Returns a list of integers.

Obtains and returns the list of signal's identifying integers for *g-type* (Note that at least one *g-type* instance must have been created prior to attempt to list or query signal's identifying integers for a given *g-type*).

---

[10] Within this context, the interface-type is the `GType` of the `GObject` subclass the signal is '`attached to`' - knowing that signals are inhereted.

[11] At least one `GObject` subclass instance must have been created prior to attempt to query any of its class signal(s).

Types and Values

`%g-signal-flags` [Instance Variable of `<gi-enum>`]

The signal flags are used to specify a signal's behaviour, the overall signal description outlines how especially the RUN flags control the stages of a signal emission.

An instance of `<gi-enum>`, who's members are the scheme representation of the `GSignalFlags`:

g-name: GSignalFlags
name: g-signal-flags
enum-set:

`run-first`
Invoke the object method handler in the first emission stage.

`run-last` Invoke the object method handler in the third emission stage.

`run-cleanup`
Invoke the object method handler in the last emission stage.

`no-recurse`
Signals being emitted for an object while currently being in emission for this very object will not be emitted recursively, but instead cause the first emission to be restarted.

`detailed` This signal supports "::detail" appendices to the signal name upon handler connections and emissions.

`action` Action signals are signals that may freely be emitted on alive objects from user code via `g-signal-emit` and friends, without the need of being embedded into extra code that performs pre or post emission adjustments on the object. They can also be thought of as object methods which can be called generically by third-party code.

`no-hooks` No emissions hooks are supported for this signal.

`must-collect`
Varargs signal emission will always collect the arguments, even if there are no signal handlers connected. Since 2.30.

`deprecated`
The signal is deprecated and will be removed in a future version. A warning will be generated if it is connected while running with `G_ENABLE_DIAGNOSTIC=1`. Since 2.32.

Gdk

G-Golf Gdk modules are defined in the `gdk` subdirectory, such as `(g-golf gdk events)`.

Where you may load these modules individually, the easiest way to use G-Golf Gdk is to import its main module, which imports and re-exports the public interface of (oop goops), (system foreign), all G-Golf support and G-Golf Gdk modules:

```
(use-modules (g-golf gdk))
```

G-Golf Gdk low level API modules correspond to a Gdk section, though they might be some exception in the future.

Events

G-Golf Gdk Events low level API.
Events — Functions for handling events from the window system

Procedures

Types and Values

Description

This section describes functions dealing with events from the window system.

Procedures

Note: in this section, the *event* argument is [must be] a pointer to a `GdkEvent`.

`gdk-event-get-button` *event* [Procedure]
Returns an integer or #f.

Extracts and returns the button number from *event*. If *event* did not deliver a button number, it returns #f.

`gdk-event-get-click-count` *event* [Procedure]
Returns an integer or #f.

Extracts and returns the click count from *event*. If *event* did not deliver a click count, it returns #f.

`gdk-event-get-coords` *event* [Procedure]
Returns a list or #f.

Extracts and returns a list of the x and y window relative coordinates from *event*. If *event* did not deliver window coordinates, it returns #f.

`gdk-event-get-keycode` *event* [Procedure]
Returns an integer or #f.

Extracts and returns the hardware keycode from *event*. If *event* did not deliver a hardware keycode, it returns #f.

`gdk-event-get-keyval` *event* [Procedure]
Returns an integer or #f.

Extracts and returns the keyval from *event*. If *event* did not deliver a key symbol, it returns #f.

`gdk-event-get-root-coords` *event* [Procedure]
Returns a list or #f.

Extracts and returns a list of the x and y root window relative coordinates from *event*. If *event* did not deliver root window coordinates, it returns #f.

`gdk-event-get-state` *event* [Procedure]
Returns a (possibly empty) list of `GdkModifierType`[12] or #f.

Obtains and returns the list of (the scheme representation of) `GdkModifierType` for *event*. If *event* did not contain a '`state`' field, it returns #f.

`gdk-event-get-time` *event* [Procedure]
Returns an integer.

Obtains and returns the time stamp for *event*, if there is one, otherwise, it returns `GDK_CURRENT_TIME` (and so does it if *event* is `%null-pointer`).

`gdk-event-get-window` *event* [Procedure]
Returns a pointer.

Extracts and returns (a pointer to) the `GdkWindow` associated with *event*.

`gdk-event-get-event-type` *event* [Procedure]
Returns the event type (symbol) name.

Obtains and returns the [%gdk-event-type], page 44, (symbol) name for *event*.

`gdk-event-get-changed-mask` *event* [Procedure]
Returns a list of flags.

Obtains and returns a list of [%gdk-window-state], page 47, flags that have changed for *event*.

`gdk-event-get-new-window-state` *event* [Procedure]
Returns a list of flags.

Obtains and returns a list of [%gdk-window-state], page 47, flags for *event*.

[12] G-Golf imports the `GdkModifierType` from the Gdk namespace as a `<gi-flag>` instance, which you may get executing `(gi-cache-ref 'flag 'gdk-modifier-type)`, then visualize running `describe` on the former result. Please refer to the enum GdkModifierType (`https://developer.gnome.org/gdk3/stable/gdk3-Windows.html#GdkModifierType`) section of the Gdk Reference Manual for a complete list and description of all modifier bit-flags.

Types and Values

`%gdk-event-type` [Instance Variable of `<gi-enum>`]

Specifies the type of the event.

Do not confuse these events with the signals that GTK+ widgets emit. Although many of these events result in corresponding signals being emitted, the events are often transformed or filtered along the way.

An instance of `<gi-enum>`, who's members are the scheme representation of the `GdkEventType`.

g-name: GdkEventType
name: gdk-event-type
enum-set:

`nothing` A special code to indicate a null event.

`delete`
The window manager has requested that the toplevel window be hidden or destroyed, usually when the user clicks on a special icon in the title bar.

`destroy` The window has been destroyed.

`expose` All or part of the window has become visible and needs to be redrawn.

`motion-notify`
The pointer (usually a mouse) has moved.

`button-press`
A mouse button has been pressed.

`2button-press`
A mouse button has been double-clicked (clicked twice within a short period of time). Note that each click also generates a button-press event.

`double-button-press`
Alias for 2button-press, added in 3.6.

`3button-press`
A mouse button has been clicked 3 times in a short period of time. Note that each click also generates a button-press event.

`triple-button-press`
Alias for 3button-press, added in 3.6.

`button-release`
A mouse button has been released.

`key-press`
A key has been pressed.

`key-release`
: A key has been released.

`enter-notify`
: The pointer has entered the window.

`leave-notify`
: The pointer has left the window.

`focus-change`
: The keyboard focus has entered or left the window.

`configure`
: The size, position or stacking order of the window has changed. Note that GTK+ discards these events for window-child windows.

`map`
: The window has been mapped.

`unmap`
: The window has been unmapped.

`property-notify`
: A property on the window has been changed or deleted.

`selection-clear`
: The application has lost ownership of a selection.

`selection-request`
: Another application has requested a selection.

`selection-notify`
: A selection has been received.

`proximity-in`
: An input device has moved into contact with a sensing surface (e.g. a touchscreen or graphics tablet).

`proximity-out`
: An input device has moved out of contact with a sensing surface.

`drag-enter`
: The mouse has entered the window while a drag is in progress.

`drag-leave`
: The mouse has left the window while a drag is in progress.

`drag-motion`
: The mouse has moved in the window while a drag is in progress.

`drag-status`
: The status of the drag operation initiated by the window has changed.

`drop-start`
: A drop operation onto the window has started.

`drop-finished`
: The drop operation initiated by the window has completed.

`client-event`
: A message has been received from another application.

`visibility-notify`
: The window visibility status has changed.

`scroll` The scroll wheel was turned

`window-state`
: The state of a window has changed. See GdkWindowState for the possible window states

`setting` A setting has been modified.

`owner-change`
: The owner of a selection has changed. This event type was added in 2.6

`grab-broken`
: A pointer or keyboard grab was broken. This event type was added in 2.8.

`damage` The content of the window has been changed. This event type was added in 2.14.

`touch-begin`
: A new touch event sequence has just started. This event type was added in 3.4.

`touch-update`
: A touch event sequence has been updated. This event type was added in 3.4.

`touch-end`
: A touch event sequence has finished. This event type was added in 3.4.

`touch-cancel`
: A touch event sequence has been canceled. This event type was added in 3.4.

`touchpad-swipe`
: A touchpad swipe gesture event, the current state is determined by its phase field. This event type was added in 3.18.

`touchpad-pinch`
: A touchpad pinch gesture event, the current state is determined by its phase field. This event type was added in 3.18.

`pad-button-press`
: A tablet pad button press event. This event type was added in 3.22.

`pad-button-release`
: A tablet pad button release event. This event type was added in 3.22.

`pad-ring` A tablet pad axis event from a "ring". This event type was added in 3.22.

`pad-strip`
: A tablet pad axis event from a "strip". This event type was added in 3.22.

`pad-group-mode`
: A tablet pad group mode change. This event type was added in 3.22.

`event-last`
: marks the end of the GdkEventType enumeration. Added in 2.18

`%gdk-window-state` [Instance Variable of `<gi-flag>`]

Specifies the state of a toplevel window.

An instance of `<gi-flag>`, who's members are the scheme representation of the `GdkWindowState`.

g-name: GdkWindowState
name: gdk-window-state
enum-set:

`withdrawn`
: The window is not shown.

`iconified`
: The window is minimized.

`maximized`
: The window is maximized.

`sticky` The window is sticky

`fullscreen`
: The window is maximized without decorations.

`above` The window is kept above other windows.

`below` The window is kept below other windows.

`focused` The window is presented as focused (with active decorations).

`tiled` The window is in a tiled state, Since 3.10. Since 3.22.23, this is deprecated in favor of per-edge information.

`top-tiled`
: Whether the top edge is tiled, Since 3.22.23

`top-resizable`
: Whether the top edge is resizable, Since 3.22.23

`right-tiled`
: Whether the right edge is tiled, Since 3.22.23

`right-resizable`
: Whether the right edge is resizable, Since 3.22.23

`bottom-tiled`
: Whether the bottom edge is tiled, Since 3.22.23

`bottom-resizable`
: Whether the bottom edge is resizable, Since 3.22.23

`left-tiled`
: Whether the left edge is tiled, Since 3.22.23

`left-resizable`
: Whether the left edge is resizable, Since 3.22.23

Key Values

G-Golf Gdk Key Values low level API.
Key Values — Functions for manipulating keyboard codes

Procedures

`[gdk-keyval-name], page 48`

Struct Hierarchy

GObject
  +— GdkKeymap

Description

Key values are the codes which are sent whenever a key is pressed or released. They appear in the `GdkEventKey.keyval` field of the `GdkEventKey` structure, which is passed to signal handlers for the '`key-press-event`' and '`key-release-event`' signals.

Please refer to the Key Values (`https://developer.gnome.org/gdk3/stable/gdk3-Keyboard-Handling.htm`
section of the Gdk Reference Manual for a complete description of both `Key Values` and `Groups`.

Procedures

`gdk-keyval-name` *keyval* [Procedure]
: Returns a symbol or #f.

  Obtains and returns the symbol name for *keyval*. If *keyval* is not a valid key, it returns #f.

GObject Introspection

G-Golf GObject Introspection modules are defined in the `gi` subdirectory, such as `(g-golf gi repository)`.

Where you may load these modules individually, the easiest way to use G-Golf GObject Introspection is to import the `g-golf` module, which imports and re-exports the public interface of all modules used and defined by G-Golf (for a complete list, visit its source definition):

```
(use-modules (g-golf))
```

Most G-Golf GObject Introspection modules correspond to a GObject Intropection (manual) section, but there are some exceptions, such as `init` and `utils` ...

Repository

G-Golf Introspection Repository low level API.
GIRepository — GObject Introspection repository manager.

Procedures

Object Hierarchy

GObject
|___ GIRepository

Description

`GIRepository` is used to manage repositories of namespaces. Namespaces are represented on disk by type libraries (.typelib files).

Procedures

Note: in this section, the *repository* optional keyword argument is [must be] a pointer to a `GIRepository` or `%null-pointer`.

`g-irepository-get-default` [Procedure]

Returns a pointer to the singleton process-global default `GIRepository`.

GObject Introspection does not currently support multiple repositories in a particular process, but this procedure is provided in the unlikely eventuality that it would become possible.

All G-Golf low level API procedures on GIRepository also accept an optional *#:repository* keyword argument which defaults to `%null-pointer`, meaning this singleton process-global default GIRepository.

`g-irepository-get-dependencies` *namespace #:key repository* [Procedure]
Returns a list of all (transitive) versioned dependencies for *namespace*. Returned string are of the form `namespace-version`.

Note: The *namespace* must have already been loaded using a procedure such as `g-irepository-require` before calling this procedure.

`g-irepository-get-loaded-namespaces` *#:key repository* [Procedure]
Return the list of currently loaded namespaces.

`g-irepository-get-n-infos` *namespace #:key repository* [Procedure]
Returns the number of metadata entries in *namespace*. The *namespace* must have already been loaded before calling this procedure.

`g-irepository-get-info` *namespace index #:key repository* [Procedure]
Returns a pointer to a particular metadata entry in the given *namespace*.

The *namespace* must have already been loaded before calling this procedure. See `g-irepository-get-n-infos` to find the maximum number of entries.

index is a 0-based offset into *namespace* for entry.

`g-irepository-get-typelib-path` *namespace #:key repository* [Procedure]
Returns the full path to the .typelib file *namespace* was loaded from, if loaded. If *namespace* is not loaded or does not exist, it will return `#f`. If the typelib for *namespace* was included in a shared library, it returns the special string `"<builtin>"`.

`g-irepository-require` *namespace #:key version repository* [Procedure]
Returns a pointer a `GITypelib` structure, if the `Typelib` file for *namespace* exists. Otherwise, it raises an error.

Force the *namespace* to be loaded if it isn't already. If *namespace* is not loaded, this procedure will search for a `".typelib"` file using the repository search path. In addition, a version version of namespace may be specified. If version is not specified, the latest will be used.

`g-irepository-get-c-prefix` *namespace #:key repository* [Procedure]
Returns the "C prefix", or the C level namespace associated with the given introspection *namespace*. Each C symbol starts with this prefix, as well each `GType` in the library.

Note: The *namespace* must have already been loaded using a procedure such as `g-irepository-require` before calling this procedure.

`g-irepository-get-shared-library` *namespace #:key repository* [Procedure]
Returns a list of paths to the shared C libraries associated with the given *namespace*. There may be no shared library path associated, in which case this procedure will return an empty list.

`g-irepository-get-version` *namespace #:key repository* [Procedure]
Returns the loaded version associated with the given *namespace*.

Note: The *namespace* must have already been loaded using a procedure such as `g-irepository-require` before calling this procedure.

`g-irepository-find-by-gtype` *gtype #:key repository* [Procedure]
Returns a pointer to a `GIBaseInfo` representing metadata about *gtype*, or `#f`.

Searches all loaded namespaces for a particular `GType`. Note that in order to locate the metadata, the namespace corresponding to the type must first have been loaded. There is currently no mechanism for determining the namespace which corresponds to an arbitrary `GType` - thus, this procedure will operate most reliably when you know the `GType` to originate from be from a loaded namespace.

`g-irepository-find-by-name` *namespace name #:key repository* [Procedure]
Returns a pointer to a `GIBaseInfo` representing metadata about type, or `#f`.

Searches for a particular entry in *namespace*. Before calling this function for a particular namespace, you must call `g-irepository-require` once to load the *namespace*, or otherwise ensure the *namespace* has already been loaded.

Common Types

G-Golf Common Types low level API.
common types - TODO

Types and Values

`%gi-type-tag` [Instance Variable of `<gi-enum>`]
An instance of `<gi-enum>`, who's members are the type tag of a `GITypeInfo`:

g-name: GITypeTag
name: gi-type-tag
enum-set:

void
boolean
int8
uint8
int16
uint16
int32
uint32
int64
uint64
float
double
gtype
utf8
filename
array
interface
glist
gslist

ghash
error
unichar

`%gi-array-type` [Instance Variable of `<gi-enum>`]
An instance of `<gi-enum>`, who's members are the type of array in a `GITypeInfo`:

g-name: GIArrayType
name: gi-array-type
enum-set:

c
array
ptr-array
byte-array

Base Info

G-Golf Base Info low level API.
GIBaseInfo — Base struct for all GITypelib structs.

Procedures

Types and Values

Struct Hierarchy

GIBaseInfo
+— GIArgInfo
+— GICallableInfo
+— GIConstantInfo
+— GIFieldInfo
+— GIPropertyInfo
+— GIRegisteredTypeInfo
+— GITypeInfo

Description

`GIBaseInfo` is the common base struct of all other *Info structs accessible through the `GIRepository API`.

Most `GIRepository APIs` returning a `GIBaseInfo` is actually creating a new struct, in other words, [g-base-info-unref], page 53, has to be called when done accessing the data. `GIBaseInfos` are normally accessed by calling either [g-irepository-find-by-name], page 51, [g-irepository-find-by-gtype], page 51, or [g-irepository-get-info], page 50.

Example: `Getting the Button of the Gtk typelib`

```
,use (g-golf gi)
(g-irepository-require "Gtk")
(g-irepository-find-by-name "Gtk" "Button")
⊣
$4 = #<pointer 0x20e0000>
... use button info ...
(g-base-info-unref $4)
```

Procedures

Note: in this section, the *info*, *info1* and *info2* arguments are [must be] pointers to a `GIBaseInfo`.

`g-base-info-ref` *info* [Procedure]

Returns the same *info*.

Increases the reference count of *info*.

`g-base-info-unref` *info* [Procedure]

Returns nothing.

Decreases the reference count of *info*. When its reference count drops to 0, the *info* is freed.

`g-base-info-equal` *info1 info2* [Procedure]

Returns #t if and only if *info1* equals *info2*.

Compares two `GIBaseInfo`.

Using pointer comparison is not practical since many functions return different instances of `GIBaseInfo` that refers to the same part of the `TypeLib`: use this procedure instead to do `GIBaseInfo` comparisons.

`g-base-info-get-type` *info* [Procedure]

Returns the info type of *info*.

`g-base-info-get-typelib` *info* [Procedure]

Returns a pointer to the `GITypelib` the *info* belongs to.

`g-base-info-get-namespace` *info* [Procedure]

Returns the namespace of *info*

`g-base-info-get-name` *info* [Procedure]
Returns the name of *info* or `#f` if it lacks a name.

What the name represents depends on the `GIInfoType` of the info. For instance for `GIFunctionInfo` it is the name of the function.

`g-base-info-get-attribute` *info name* [Procedure]
Returns the value of the attribute or `#f` if not such attribute exists.

`g-base-info-iterate-attributes` *info proc* [Procedure]
Returns nothing.

Iterate and calls *proc* over all attributes associated with this node. *proc* must be a procedure of two arguments, the *name* and the *value* of the attribute.

`g-base-info-get-container` *info* [Procedure]
Returns a pointer to a `GIBaseInfo`.

The container is the parent `GIBaseInfo`. For instance, the parent of a `GIFunctionInfo` is an `GIObjectInfo` or `GIInterfaceInfo`.

`g-base-info-is-deprecated` *info* [Procedure]
Returns `#t` if deprecated.

Obtain whether *info* represents a metadata which is deprecated or not.

Types and Values

`%gi-info-type` [Instance Variable of `<gi-enum>`]
An instance of `<gi-enum>`, who's members are the scheme representation of the type of a `GIBaseInfo` struct:

g-name: GIInfoType
name: gi-info-type
enum-set:

invalid
function
callback
struct
boxed
enum
flags
object
interface
constant
error-domain
union
value
signal
vfunc
property

field
arg
type
unresolved

Callable Info

G-Golf Callable Info low level API.
GICallableInfo — Struct representing a callable.

Procedures

Struct Hierarchy

GIBaseInfoInfo
+— GICallableInfo
+— GIFunctionInfo
+— GISignalInfo
+— GIVFuncInfo

Description

`GICallableInfo` represents an entity which is callable. Currently a function (`GIFunctionInfo`), virtual function, (`GIVFuncInfo`) or callback (`GICallbackInfo`).

A callable has a list of arguments (`GIArgInfo`), a return type, direction and a flag which decides if it returns null.

Procedures

Note: in this section, the *info* argument is [must be] a pointer to a `GICallableInfo`.

`g-callable-info-get-n-args` *info* [Procedure]
Returns the number of arguments this *info* expects.

Obtain the number of arguments (both IN and OUT) for this *info*.

`g-callable-info-get-arg` *info n* [Procedure]
Returns a pointer to the *n*th `GIArgInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-callable-info-get-instance-ownership-transfer` *info* [Procedure]
Returns a `GITransfer` enumerated value.

Obtains the ownership transfer for the instance argument. See [%gi-transfer], page 74, for the list of possible values.

`g-callable-info-get-caller-owns` *info* [Procedure]
Returns a `GITransfer` enumerated value.

See whether the caller owns the return value of this callable. See [%gi-transfer], page 74, for the list of possible values.

`g-callable-info-get-return-type` *info* [Procedure]
Returns a pointer to the `GITypeInfo`.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-callable-info-may-return-null` *info* [Procedure]
Returns `#t` if the callable *info* could return `NULL`.

See if a callable could return NULL.

Signal Info

G-Golf Signal Info low level API.
GISignalInfo — Struct representing a signal.

Procedures

`[g-signal-info-get-flags], page 56`

Description

`GISignalInfo` represents a signal. It's a sub-struct of `GICallableInfo` and contains a set of flags and a class closure.

See also [Callable Info], page 55, for information on how to retreive arguments and other metadata from the signal.

Struct Hierarchy

GIBaseInfoInfo
+— GICallableInfo
+— GIFunctionInfo
+— GISignalInfo
+— GIVFuncInfo

Procedures

Note: in this section, the *info* argument is [must be] a pointer to a `GISignalInfo`.

`g-signal-info-get-flags` *info* [Procedure]
Returns a list of [%g-signal-flags], page 41.

Obtain the flags for this signal info. See [%g-signal-flags], page 41, for more information about posible flag values.

Function Info

G-Golf Function Info low level API.
GIFunctionInfo — Struct representing a function.

Procedures

Types and Values

Struct Hierarchy

GIBaseInfoInfo
+— GICallableInfo
+— GIFunctionInfo
+— GISignalInfo
+— GIVFuncInfo

Description

`GIFunctionInfo` represents a function, method or constructor. To find out what kind of entity a GIFunctionInfo represents, call [g-function-info-get-flags], page 57.

See also [Callable Info], page 55, for information on how to retreive arguments and other metadata.

Procedures

Note: in this section, the *info* argument is [must be] a pointer to a `GIFunctionInfo`.

`gi-function-info-is-method?` *info* [*flags #f*] [Procedure]
Returns `#t` if *info* is a method, that is if `is-method` is a member of the *info* flags. Otherwise, it returns `#f`.

The optional *flags* argument, if passed, must be the list of the function info flags as returned by [g-function-info-get-flags], page 57.

`g-function-info-get-flags` *info* [Procedure]
Returns a list of [%g-function-info-flags], page 58.

Obtain the `GIFunctionInfoFlags` for *info*.

`g-function-info-get-property` *info* [Procedure]
Returns a pointer or `#f`.

Obtains the `GIPropertyInfo` associated with *info*. Only `GIFunctionInfo` with the flag `is-getter` or `is-setter` have a property set. For other cases, `#f` will be returned.

The `GIPropertyInfo` must be freed by calling [g-base-info-unref], page 53, when done.

`g-function-info-get-symbol` *info* [Procedure]
Returns a string.

Obtain the 'symbol' of the function[13].

g-function-info-get-vfunc *info* [Procedure]

Returns a pointer or #f.

Obtains the GIVFuncInfo associated with *info*. Only GIFunctionInfo with the flag wraps-vfunc has its virtual function set. For other cases, #f will be returned.

The GIVFuncInfo must be freed by calling [g-base-info-unref], page 53, when done.

g-function-info-invoke *info in-args n-in out-args n-out r-val g-error* [Procedure]

Returns #t if the function has been invoked, #f if an error occured.

Invokes the function described in *info* with the given arguments. Note that inout parameters must appear in both argument lists. The arguments are:

info a pointer to a GIFunctionInfo describing the function to invoke.

in-args a pointer to an array of GIArguments, one for each in and inout parameter of *info*. If there are no in parameter, *in-args* must be the %null-pointer.

n-in the length of the *in-args* array.

out-args a pointer to an array of GIArguments, one for each out and inout parameter of *info*. If there are no out parameter, *out-args* must be the %null-pointer.

n-out the length of the *out-args* array.

r-val a pointer to a GIArguments, the return location for the return value of the function. If the function returns void, *r-val* must be the %null-pointer.

g-error a pointer to a newly allocated (and 'empty') GError (the recommended way for procedure calls that need such a pointer is to 'surround' the call using [with-gerror], page 82).

Types and Values

%g-function-info-flags [Instance Variable of <gi-flag>]

An instance of <gi-flag>, who's members are the scheme representation of the GIFunctionInfoFlags:

g-name: GIFunctionInfoFlags
name: gi-function-info-flags
enum-set:

is-method
Is a method.

[13] As you have noticed already, since g-function-info-get-symbol returns a string, in the Glib, GObject and GObject Instrospection worlds, symbol has a different meaning then in the Lisp/Scheme worlds. However, since the procedure is part of the G-Golf low-level API, we decided to keep its name as close as the original name as possible, which in Glib terms is the name of the exported function, 'suitable to be used as an argument to g_module_symbol()'

`is-constructor`
: Is a constructor.

`is-getter`
: Is a getter of a `GIPropertyInfo`.

`is-setter`
: Is a setter of a `GIPropertyInfo`.

`wraps-vfunc`
: Represent a virtul function.

`throws`
: The function may throw an error.

Registered Type Info

G-Golf Registered Type Info low level API.
GIRegisteredTypeInfo — Struct representing a struct with a `GType`.

Procedures

Struct Hierarchy

GIBaseInfo
+—-GIRegisteredTypeInfo
+—-GIEnumInfo
+—-GIInterfaceInfo
+—-GIObjectInfo
+—-GIStructInfo
+—-GIUnionInfo

Description

`GIRegisteredTypeInfo` represents an entity with a `GType` associated. Could be either a `GIEnumInfo`, `GIInterfaceInfo`, `GIObjectInfo`, `GIStructInfo` or a `GIUnionInfo`.

A registered type info struct has a name and a type function.

Procedures

Note: in this section, the *info* argument is [must be] a pointer to a `GIRegisteredTypeInfo`.

`g-registered-type-info-get-type-name` *info* [Procedure]
: Returns the type name.

  Obtain the type name of the struct within the GObject type system. This type can be passed to g_type_name() to get a GType.

`g-registered-type-info-get-type-init` *info* [Procedure]
: Returns the name of the type init function.

Obtain the type init function for *info*. The type init function is the function which will register the `GType` within the `GObject` type system. Usually this is not called by langauge bindings or applications.

`g-registered-type-info-get-g-type` *info* [Procedure]

Returns the `GType` for *info*.

Obtain the `GType` for this registered type or `G_TYPE_NONE` which has a special meaning. It means that either there is no type information associated with this info or that the shared library which provides the type_init function for this info cannot be called.

Enum Info

G-Golf Enum Info low level API.
GIEnumInfo — Structs representing an enumeration and its values.

Procedures

Struct Hierarchy

GIBaseInfo
+— GIRegisteredTypeInfo
+— GIEnumInfo

Description

`GIEnumInfo` represents an argument. An argument is always part of a `GICallableInfo`.

Procedures

Note: in this section, unless otherwise specified, the *info* argument is [must be] a pointer to a `GIEumInfo`.

`gi-enum-import` *info* [Procedure]

Returns a `<gi-enum>` instance.

Obtains the values this enumeration contains, then makes and returns a `<gi-enum>` instance.

`gi-enum-value-values` *info* [Procedure]

Returns an alist.

Obtains and returns the list pairs `(symbol . id)` the enum GI definition pointed by *info* contains. If you think the name is strange, compare it with, for example [gi-struct-field-types], page 62: just like a `GIStructInfo` holds a list of pointers to `GIFieldInfo` from which we get the (field) type, a`GIEnumInfo` holds a list of pointers

to `GIValueInfo` from which we get the (enum) value - which in the GI world is a name (a string) that we transform, in the scheme world, to a symbol.

`g-enum-info-get-n-values` *info* [Procedure]
Returns the number of values.

Obtains the number of values this enumeration contains.

`g-enum-info-get-value` *info index* [Procedure]
Returns a pointer to a `GIValueInfo` or #f if type tag is wrong.

Obtains a value for this enumeration. The `GIValueInfo` must be free'd using `g-base-info-unref` when done.

index is a 0-based offset into *info* for a value.

`g-enum-info-get-n-methods` *info* [Procedure]
Returns the number of methods.

Obtains the number of methods this enumeration has.

`g-enum-info-get-method` *info index* [Procedure]
Returns a pointer to a `GIFunctionInfo` or #f if type tag is wrong.

Obtains a method for this enumeration. The `GIFunctionInfo` must be free'd using `g-base-info-unref` when done.

index is a 0-based offset into *info* for a method.

`g-value-info-get-value` *info* [Procedure]
Returns the enumeration value.

Obtains a value of the `GIValueInfo`.

info is [must be] a pointer to a `GIValueInfo`.

Struct Info

G-Golf Struct Info low level API.
GIStructInfo — Structs representing a C structure.

Procedures

Struct Hierarchy

GIBaseInfo
  +— GIRegisteredTypeInfo
    +— GIStructInfo

Description

`GIStructInfo` represents a generic C strucuture type.

A structure has methods and fields.

Procedures

Note: in this section, unless otherwise specified, the *info* argument is [must be] a pointer to a `GIStructInfo`.

`gi-struct-import` *info* [Procedure]
Returns a `<gi-struct>` instance.

Obtains the list of (field) types the C struct GI definition pointed by *info* contains, then makes and returns a `<gi-struct>` instance.

`gi-struct-field-types` *info* [Procedure]
Returns a list.

Obtains and returns the list of (field) types the C struct GI definition pointed by *info* contains.

`g-struct-info-get-alignment` *info* [Procedure]
Returns an integer.

Obtains and returns the required alignment for *info*.

`g-struct-info-get-size` *info* [Procedure]
Returns an integer.

Obtains and returns the total size of the structure specified *info*.

`g-struct-info-is-gtype-struct` *info* [Procedure]
Returns #t or #f.

Return true if the structure specified by *info* represents the "class structure" for some GObject or GInterface.

`g-struct-info-is-foreign` *info* [Procedure]
Returns #t or #f.

FIXME. No upstream documentation, though the procedure works.

`g-struct-info-get-n-fields` *info* [Procedure]
Returns an integer.

Obtains the number of fields for *info*.

`g-struct-info-get-field` *info n* [Procedure]
Returns a pointer.

Obtains and returns the *info* type information (a pointer to a `GIFieldInfo`) for the field at the specified *n* index.

The `GIFieldInfo` must be freed by calling [g-base-info-unref], page 53, when done.

`g-struct-info-get-n-methods` *info* [Procedure]

Returns an integer.

Obtains the number of methods for *info*.

`g-struct-info-get-method` *info n* [Procedure]

Returns a pointer.

Obtains and returns the *info* type information (a pointer to a `GIFunctionInfo`) for the method at the specified *n* index.

The `GIFunctionInfo` must be freed by calling [g-base-info-unref], page 53, when done.

Union Info

G-Golf Union Info low level API.
GIUnionInfo — Struct representing a C union.

Procedures

Description

`GIUnionInfo` represents a union type.

A union has methods and fields. Unions can optionally have a discriminator, which is a field deciding what type of real union fields is valid for specified instance.

Struct Hierarchy

GIBaseInfo
+— GIRegisteredTypeInfo
+— GIUnionInfo

Procedures

Note: in this section, unless otherwise specified, the *info* argument is [must be] a pointer to a `GIUnionInfo`.

`g-union-info-get-n-fields` *info* [Procedure]

Returns an integer.

Obtains and returns the number of fields the *info* union has.

`g-union-info-get-field` *info n* [Procedure]
Returns a pointer.

Obtains and returns a pointer to the `GIFieldInfo` for *info*, given its *n*. The `GIFieldInfo` must be free'd by calling [g-base-info-unref], page 53, when done.

`g-union-info-get-n-methods` *info* [Procedure]
Returns an integer.

Obtains and returns the number of methods the *info* union has.

`g-union-info-get-method` *info n* [Procedure]
Returns a pointer.

Obtains and returns a pointer to the `GIFunctionInfo` for *info*, given its *n*, which must be free'd by calling [g-base-info-unref], page 53, when done.

`g-union-info-is-discriminated?` *info* [Procedure]
Returns #t if *info* contains a discriminator field, otherwise it returns #f.

`g-union-info-get-discriminator-offset` *info* [Procedure]
Returns an integer.

Obtains and returns the offset of the discriminator field for *info*.

`g-union-info-get-discriminator-type` *info* [Procedure]
Returns a pointer.

Obtains and returns a pointer to the `GITypeInfo` for *info*, which must be free'd by calling [g-base-info-unref], page 53, when done.

`g-union-info-get-discriminator` *info n* [Procedure]
Returns a pointer.

Obtains and returns a pointer to the `GIConstantInfo` assigned for the *info* *n*-th union field - i.e. the *n*-th union field is the active one if discriminator contains this constant (value) - which must be free'd by calling [g-base-info-unref], page 53, when done.

`g-union-info-get-size` *info* [Procedure]
Returns an integer.

Obtains and returns the total size of the union specified by *info*.

`g-union-info-get-alignment` *info* [Procedure]
Returns an integer.

Obtains and returns the required alignment for *info*.

Object Info

G-Golf Object Info low level API.
GIObjectInfo — Structs representing a GObject.

Procedures

Struct Hierarchy

GIBaseInfo
  +— GIRegisteredTypeInfo
      +— GIObjectInfo

Description

`GIObjectInfo` represents a GObject (`https://developer.gnome.org/gobject/stable/gobject-The-Base-`
This doesn't represent a specific instance of a GObject, instead this represent the object type (eg class).

A GObject has methods, fields, properties, signals, interfaces, constants and virtual functions.

Procedures

Note: in this section, unless otherwise specified, the *info* argument is [must be] a pointer to a `GIObjectInfo`.

`gi-object-show` *info* [Procedure]

Returns nothing.

Obtains and displays the following informations about the object (and its parent) pointed to by *info*:

```
,use (g-golf)
```

```
(g-irepository-require "Clutter")
$2 = #<pointer 0x56396a4f9f80>

(g-irepository-find-by-name "Clutter" "Actor")
$3 = #<pointer 0x56396a4fdc00>

(gi-object-show $3)
⊣
#<pointer 0x56396a4fdc00> is a (pointer to a) GIObjectInfo:

  Parent:
          namespace: "GObject"
               name: "InitiallyUnowned"
             g-type: 94804596757600
       g-type-name: "GInitiallyUnowned"

  Object:
          namespace: "Clutter"
               name: "Actor"
             g-type: 94804596864480
       g-type-name: "ClutterActor"
          abstract: #f
       n-constants: 0
          n-fields: 4
     n-interfaces: 4
          n-methods: 238
     n-properties: 82
          n-signals: 26
          n-vfuncts: 35
```

`gi-object-property-names` *info* [Procedure]

Returns a (possibly empty) list.

Obtains and returns the (possibly empty) list of the (untranslated) GI property names for *info* (see [g-name->name], page 92, to obtain their scheme representation).

`g-object-info-get-abstract` *info* [Procedure]

Returns #t if the *info* object type is abstract.

Obtain if the object type is an abstract type, eg if it cannot be instantiated.

`g-object-info-get-parent` *info* [Procedure]

Returns a pointer or #f.

Obtains and returns a pointer to the *info*'s parent `GIObjectInfo`, or #f if *info* has no parent.

`g-object-info-get-type-name` *info* [Procedure]

Returns the name of the object type for *info*.

Obtain the name of the object class/type for *info*.

`g-object-info-get-type-init` *info* [Procedure]
Returns a function name (a string).

Obtain the function name which when called will return the GType function for which this object type is registered.

`g-object-info-get-n-constants` *info* [Procedure]
Returns the number of constants for *info*.

Obtain the number of constants that this object type has.

`g-object-info-get-constant` *info n* [Procedure]
Returns a pointer to the *n*th `GIConstantInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-object-info-get-n-fields` *info* [Procedure]
Returns the number of fields for *info*.

Obtain the number of fields that this object type has.

`g-object-info-get-field` *info n* [Procedure]
Returns a pointer to the *n*th `GIFieldInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-object-info-get-n-interfaces` *info* [Procedure]
Returns the number of interfaces for *info*.

Obtain the number of interfaces that this object type has.

`g-object-info-get-interface` *info n* [Procedure]
Returns a pointer to the *n*th `GIInterfaceInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-object-info-get-n-methods` *info* [Procedure]
Returns the number of methods for *info*.

Obtain the number of methods that this object type has.

`g-object-info-get-method` *info n* [Procedure]
Returns a pointer to the *n*th `GIFunctionInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-object-info-find-method` *info name* [Procedure]
Returns a pointer to a `GIFunctionInfo` or #f if there is no method available with that name.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-object-info-get-n-properties` *info* [Procedure]
Returns the number of properties for *info*.

Obtain the number of properties that this object type has.

`g-object-info-get-property` *info n* [Procedure]
Returns a pointer to the *n*th `GIPropertyInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-object-info-get-n-signals` *info* [Procedure]
Returns the number of signals for *info*.

Obtain the number of signals that this object type has.

`g-object-info-get-signal` *info n* [Procedure]
Returns a pointer to the *n*th `GISignalInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-object-info-find-signal` *info name* [Procedure]
Returns a pointer to a `GISignalInfo` or #f if there is no signal available with that name.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-object-info-get-n-vfuncs` *info* [Procedure]
Returns the number of vfuncs for *info*.

Obtain the number of vfuncs that this object type has.

`g-object-info-get-vfunc` *info n* [Procedure]
Returns a pointer to the *n*th `GIVfuncInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-object-info-get-class-struct` *info* [Procedure]
Returns a pointer to the *n*th `GIStructInfo` of *info*, or #f.

Every `GObject` has two structures: an instance structure and a class structure. This function returns a pointer to the *info* class structure.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

Interface Info

G-Golf Interface Info low level API.
GIInterfaceInfo — Structs representing a GInterface.

Procedures

Description

`GIInterfaceInfo` represents a GInterface (`https://developer.gnome.org/gobject/stable/GTypeModule.h`

A GInterface has methods, properties, signals, constants, virtual functions and prerequisites.

Struct Hierarchy

GIBaseInfo
  +— GIRegisteredTypeInfo
      +— GIInterfaceInfo

Procedures

Note: in this section, unless otherwise specified, the *info* argument is [must be] a pointer to a `GIInterfaceInfo`.

`gi-interface-import` *info* [Procedure]

Returns a list.

In the current version of G-Golf, interfaces are '`opaques`'. Returns a list composed of the 'interface (type-tag) symbol, the interface (scheme and symbol) name, g-name, g-type and #t (a boolean that means the type is confirmed). Here is an example:

```
(interface gtk-orientable "GtkOrientable" 94578771473520 #t)
```

`gi-interface-show` *info* [Procedure]

Returns nothing.

Obtains and displays the following informations about the interface pointed to by *info*:

```
,use (g-golf)
```

```
(g-irepository-require "Gtk")
$2 = #<pointer 0x55649014c780>

(g-irepository-find-by-name "Gtk" "Orientable")
$3 = #<pointer 0x5564901531e0>

(gi-interface-show $3)
⊣
#<pointer 0x5564901531e0> is a (pointer to a) GIInterfaceInfo:

          namespace: "Gtk"
               name: "Orientable"
             g-type: 93890405098944
        g-type-name: "GtkOrientable"
    n-prerequisites: 0
       n-properties: 1
          n-methods: 2
          n-signals: 0
          n-vfuncts: 0
         n-constants: 0
        iface-struct: #<pointer 0x556490153140>
  iface-struct-name: "OrientableIface"
```

`g-interface-info-get-n-prerequisites` *info* [Procedure]
Returns the number of prerequisites for *info*.

Obtain the number of prerequisites for this interface type. A prerequisites is another interface that needs to be implemented for interface, similar to a base class for GObjects.

`g-interface-info-get-prerequisite` *info n* [Procedure]
Returns a pointer to the *n*th prerequisite for *info*.

The prerequisite as a `GIBaseInfo`. It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-interface-info-get-n-properties` *info* [Procedure]
Returns the number of properties for *info*.

Obtain the number of properties that this interface type has.

`g-interface-info-get-property` *info n* [Procedure]
Returns a pointer to the *n*th `GIPropertyInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-interface-info-get-n-methods` *info* [Procedure]
Returns the number of methods for *info*.

Obtain the number of methods that this interface type has.

`g-interface-info-get-method` *info n* [Procedure]
Returns a pointer to the *n*th `GIFunctionInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-interface-info-find-method` *info name* [Procedure]
Returns a pointer to a `GIFunctionInfo` or #f if there is no method available with that name.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-interface-info-get-n-signals` *info* [Procedure]
Returns the number of signals for *info*.

Obtain the number of signals that this interface type has.

`g-interface-info-get-signal` *info n* [Procedure]
Returns a pointer to the *n*th `GISignalInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-interface-info-find-signal` *info name* [Procedure]
Returns a pointer to a `GISignalInfo` or #f if there is no signal available with that name.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-interface-info-get-n-vfuncs` *info* [Procedure]
Returns the number of vfuncs for *info*.

Obtain the number of vfuncs that this interface type has.

`g-interface-info-get-vfunc` *info n* [Procedure]
Returns a pointer to the *n*th `GIVfuncInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-interface-info-find-vfunc` *info name* [Procedure]
Returns a pointer to a `GIFunctionInfo` or #f if there is no signal available with that name.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-interface-info-get-n-constants` *info* [Procedure]
Returns the number of constants for *info*.

Obtain the number of constants that this interface type has.

`g-interface-info-get-constant` *info n* [Procedure]
Returns a pointer to the *n*th `GIConstantInfo` of *info*.

It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

`g-interface-info-get-iface-struct` *info* [Procedure]
Returns a pointer to a `GIStructInfo` for *info*, or #f.

Obtains and returns the layout C structure associated with *info*. It must be freed by calling [g-base-info-unref], page 53, when done accessing the data.

Arg Info

G-Golf Arg Info low level API.
GIArgInfo — Struct representing an argument.

Procedures

Types and Values

Struct Hierarchy

GIBaseInfo
+— GIArgInfo

Description

`GIArgInfo` represents an argument. An argument is always part of a `GICallableInfo`.

Procedures

Note: in this section, the *info* argument is [must be] a pointer to a `GIArgInfo`.

`g-arg-info-get-closure` *info* [Procedure]

Returns the index of the user data argument or -1 if there is none.

Obtains the index of the user data argument. This is only valid for arguments which are callbacks.

`g-arg-info-get-destroy` *info* [Procedure]

Returns the index of the `GDestroyNotify` argument or -1 if there is none.

Obtains the index of the `GDestroyNotify` argument. This is only valid for arguments which are callbacks.

`g-arg-info-get-direction` *info* [Procedure]

Returns a symbol.

Obtains and returns the [%gi-direction], page 73, of the argument.

`g-arg-info-get-ownership-transfer` *info* [Procedure]
Returns a symbol.

Obtains and returns the [%gi-transfer], page 74, for this argument.

`g-arg-info-get-scope` *info* [Procedure]
Returns a symbol.

Obtains and returns the [%gi-scope-type], page 74, for this argument. The scope type explains how a callback is going to be invoked, most importantly when the resources required to invoke it can be freed.

`g-arg-info-get-type` *info* [Procedure]
Returns a pointer.

Obtains the `GITypeInfo` holding the type information for *info*. Free it using [g-base-info-unref], page 53, when done.

`g-arg-info-may-be-null` *info* [Procedure]
Returns `#t` or `#f`.

Obtains if the type of the argument includes the possibility of `NULL`. For 'in' values this means that `NULL` is a valid value. For 'out' values, this means that `NULL` may be returned.

`g-arg-info-is-caller-allocates` *info* [Procedure]
Returns `#t` or `#f`.

Obtain if the argument is a pointer to a struct or object that will receive an output of a function. The default assumption for `out` arguments which have allocation is that the callee allocates; if this is TRUE, then the caller must allocate.

`g-arg-info-is-optional` *info* [Procedure]
Returns `#t` or `#f`.

Obtains if the argument is optional. For 'out' arguments this means that you can pass `NULL` in order to ignore the result.

`g-arg-info-is-return-value` *info* [Procedure]
Returns `#t` or `#f`.

Obtains if the argument is a retur value. It can either be a parameter or a return value.

`g-arg-info-is-skip` *info* [Procedure]
Returns `#t` or `#f`.

Obtains if an argument is only useful in C.

Types and Values

`%gi-direction` [Instance Variable of `<gi-enum>`]
An instance of `<gi-enum>`, who's members are the scheme representation of the direction of a `GIArgInfo`:

g-name: GIDirection
name: gi-direction
enum-set:

`in`	in argument.
`out`	out argument.
`inout`	in and out argument.

`%gi-scope-type` [Instance Variable of `<gi-enum>`]

An instance of `<gi-enum>`, who's members are the scheme representation of the scope of a `GIArgInfo`. Scope type of a `GIArgInfo` representing callback, determines how the callback is invoked and is used to decide when the invoke structs can be freed.

g-name: GIScopeType
name: gi-scope-type
enum-set:

`invalid`	The argument is not of callback type.
`call`	The callback and associated user_data is only used during the call to this function.
`async`	The callback and associated user_data is only used until the callback is invoked, and the callback. is invoked always exactly once.
`notified`	The callback and and associated user_data is used until the caller is notfied via the destroy_notify.

`%gi-transfer` [Instance Variable of `<gi-enum>`]

The transfer is the exchange of data between two parts, from the callee to the caller. The callee is either a function/method/signal or an object/interface where a property is defined. The caller is the side accessing a property or calling a function. `GITransfer` specifies who's responsible for freeing the resources after the ownership transfer is complete. In case of a containing type such as a list, an array or a hash table the container itself is specified differently from the items within the container itself. Each container is freed differently, check the documentation for the types themselves for information on how to free them.

An instance of `<gi-enum>`, who's members are the scheme representation of the `GITransfer`:

g-name: GITransfer
name: gi-transfer
enum-set:

`nothing`	transfer nothing from the callee (function or the type instance the property belongs to) to the caller. The callee retains the ownership of the transfer and the caller doesn't need to do anything to free up the resources of this transfer

`container`
: transfer the container (list, array, hash table) from the callee to the caller. The callee retains the ownership of the individual items in the container and the caller has to free up the container resources `g_list_free`, `g_hash_table_destroy`, ... of this transfer

`everything`
: transfer everything, eg the container and its contents from the callee to the caller. This is the case when the callee creates a copy of all the data it returns. The caller is responsible for cleaning up the container and item resources of this transfer

Constant Info

G-Golf Constant Info low level API.
GIConstantInfo — Struct representing a constant.

Procedures

Struct Hierarchy

GIBaseInfo
+— GIConstantInfo

Description

`GIConstantInfo` represents a constant. A constant has a type associated which can be obtained by calling [g-constant-info-get-type], page 75, and a value, which can be obtained by calling [g-constant-info-get-value], page 75.

Procedures

Note: in this section, the *info* and *value* arguments are [must be] pointers to a `GIConstantInfo` and a `GIArgument`, respectively.

`g-constant-info-free-value` *info value* [Procedure]
: Returns nothing.

  Frees the value returned from [g-constant-info-get-value], page 75.

`g-constant-info-get-type` *info* [Procedure]
: Returns a pointer.

  Obtains and returns a pointer to the `GITypeInfo` for *info*. Free it using [g-base-info-unref], page 53, when done.

`g-constant-info-get-value` *info value* [Procedure]
: Returns an integer (the size of a constant).

Obtains the value associated with *info* and store it in the *value* parameter, which must be allocated before passing it.

The size of the constant value stored in argument will be returned. Free the *value* argument with [g-constant-info-free-value], page 75.

Field Info

G-Golf Field Info low level API.
GIFieldInfo — Struct representing a struct or union field.

Procedures

`[g-field-info-get-type], page 76`

Struct Hierarchy

GIBaseInfo
+— GIFieldInfo

Description

A `GIFieldInfo` struct represents a field of a struct (see [Struct Info], page 61), union (see `GIUnionInfo`) or an object (see [Object Info], page 64). The GIFieldInfo is fetched by calling [g-struct-info-get-field], page 62, `g-union-info-get-field` or [g-object-info-get-field], page 67. A field has a size, type and a struct offset asssociated and a set of flags, which is currently `readable` or `writable`.

Procedures

Note: in this section, unless otherwise specified, the *info* argument is [must be] a pointer to a `GIFieldInfo`.

`g-field-info-get-type` *info* [Procedure]
Returns a pointer.

Obtains and returns the `GITypeInfo` for *info*.

The `GITypeInfo` must be freed by calling [g-base-info-unref], page 53, when done.

Property Info

G-Golf Property Info low level API.
GIPropertyInfo — Struct representing a property.

Procedures

`[gi-property-g-type], page 77`
`[g-property-info-get-flags], page 77`
`[g-property-info-get-ownership-transfer], page 77`
`[g-property-info-get-type], page 77`

Struct Hierarchy

GIBaseInfoInfo
+— GIPropertyInfo

Description

`GIPropertyInfo` represents a property. A property belongs to either a `GIObjectInfo` or a `GIInterfaceInfo`.

Procedures

Note: in this section, the *info* argument is [must be] a pointer to a `GIPropertyInfo`.

`gi-property-g-type` *info* [Procedure]
Returns an integer.

Obtains and returns the `GType` value of the property.

`g-property-info-get-flags` *info* [Procedure]
Returns a list of [%g-param-flags], page 36.

Obtain the flags for this property *info*. See [GParamSpec], page 35, for the list of possible flag values.

`g-property-info-get-ownership-transfer` *info* [Procedure]
Returns the ownership transfer for this property.

Obtain the ownership transfer for this property. See [%gi-transfer], page 74, for more information about transfer values.

`g-property-infoxs-get-type` *info* [Procedure]
Returns a pointer to a `GITypeInfo`.

Obtain the type information for this property. The `GITypeInfo` must be free'd using `g-base-info-unref` when done.

Type Info

G-Golf Type Info low level API.
GITypeInfo — Struct representing a type.

Procedures

[g-type-tag-to-string], page 78
[g-info-type-to-string], page 78
[g-type-info-is-pointer], page 78
[g-type-info-get-tag], page 78
[g-type-info-get-param-type], page 78
[g-type-info-get-interface], page 78
[g-type-info-get-array-length], page 79
[g-type-info-get-array-fixed-size], page 79
[g-type-info-is-zero-terminated], page 79
[g-type-info-get-array-type], page 79

Struct Hierarchy

GIBaseInfoInfo
+— GITypeInfo

Description

`GITypeInfo` represents a type. You can retrieve a type info from an argument (see [Arg Info], page 72), a functions return value (see [Function Info], page 56), a field (see `GIFieldInfo`), a property (see [Property Info], page 76), a constant (see `GIConstantInfo`) or for a union discriminator (see `GIUnionInfo`).

A type can either be a of a basic type which is a standard C primitive type or an interface type. For interface types you need to call `g-type-info-get-interface` to get a reference to the base info for that interface.

Procedures

Note: in this section, the *info* argument is [must be] a pointer to a `GITypeInfo`.

`g-type-tag-to-string` *type-tag* [Procedure]
Returns a string or `#f`.

Obtains the string representation for *type-tag* or `#f` if it does not exists.

type-tag can either be a `symbol` or an `id`, a member of the `enum-set` of [%gi-type-tag], page 51, (otherwise, `#f` is returned).

`g-info-type-to-string` *info-type* [Procedure]
Returns a string or `#f`.

Obtains the string representation for *info-type* or `#f` if it does not exists.

info-type can either be a `symbol` or an `id`, a member of the `enum-set` of [%gi-info-type], page 54, (otherwise, `#f` is returned).

`g-type-info-is-pointer` *info* [Procedure]
Returns `#t` or `#f`.

Obtains if the *info* type is passed as a reference.

Note that the types of `out` and `inout` parameters (see [%gi-direction], page 73) will only be pointers if the underlying type being transferred is a pointer (i.e. only if the type of the C function's formal parameter is a pointer to a pointer).

`g-type-info-get-tag` *info* [Procedure]
Returns a symbol.

Obtains the type tag for *info* (see [%gi-type-tag], page 51, for the list of type tags).

`g-type-info-get-param-type` *info n* [Procedure]
Returns a pointer or `#f`.

Obtains the parameter type *n* (the index of the parameter). When there is no such *n* parameter, the procedure returns `#f`.

`g-type-info-get-interface` *info* [Procedure]
Returns a pointer or `#f`.

For `interface` types (see [%gi-type-tag], page 51) such as GObjects and boxed values, this procedure returns a (pointer to a) `GIBaseInfo`, holding full information about the referenced type. You can then inspect the type of the returned `GIBaseInfo` to

further query whether it is a concrete GObject, a GInterface, a structure, etc. using [g-base-info-get-type], page 53.

`g-type-info-get-array-length` *info* [Procedure]
Returns an interger.

Obtain the array length of the type. The type tag must be a `array` (see [%gi-type-tag], page 51), or -1 will returned.

`g-type-info-get-array-fixed-size` *info* [Procedure]
Returns an interger.

Obtain the fixed array syze of the type. The type tag must be a `array` (see [%gi-type-tag], page 51), or -1 will returned.

`g-type-info-is-zero-terminated` *info* [Procedure]
Returns `#t` or `#f`.

Obtains if the last element of the array is `NULL`. The type tag must be a `array` (see [%gi-type-tag], page 51), or `#f` will returned.

`g-type-info-get-array-type` *info* [Procedure]
Returns a symbol or `#f`.

Obtain the array type for this type (see [%gi-array-type], page 52). If the type tag of this type is not array, `#f` will be returned.

Typelib

G-Golf Typelib low level API.
GITypelib — Layout and accessors for typelib.

Procedures

Description

TODO.

Procedures

Note: in this section, the *typelib* argument is [must be] a pointer to a `GITypelib`.

`g-golf-typelib-new` *file* [Procedure]
Returns a pointer to a new `GITypelib`.

file must be a valid typelib filename.

This procedure actually sets things up and calls [g-typelib-new-from-memory], page 80.

`call-with-input-typelib` *file proc* [Procedure]
Returns the value(s) returned by *proc*.

file must be a valid typelib filename. Makes a new `GITypelib` by calling `(g-golf-typelib-new file)` and calls `(proc typelib)` with the resulting `GITypelib`.

When *proc* returns, the `GITypelib` is free'd by calling `g-typelib-free`. Otherwise the [Glib - C] memory chunk might not be free'd automatically, though the scheme pointer returned by `g-golf-typelib-new` will be garbage collected in the usual way if not otherwise referenced.

`g-typelib-new-from-memory` *pointer size gerror* [Procedure]
Returns a pointer to a new `GITypelib`.

pointer must be the address of a memory chunk containing the typelib, *size* is the number of bytes of the memory chunk containing the typelib, and *gerror* a pointer to a `GError`.

Creates a new `GITypelib` from a memory location. The memory block pointed to by typelib will be automatically `g_free()`d when the repository is destroyed.

`g-typelib-free` *typelib* [Procedure]
Returns nothing.

Free a GITypelib.

`g-typelib-get-namespace` *typelib* [Procedure]
Returns the namespace of *typelib*.

Utilities

G-Golf GObject Introspetion Utilities low level API.

Procedures and Syntax

Types and Values

Description

G-Golf GObject Introspection utilities low level API.

Procedures and Syntax

`gi-pointer-new` [Procedure]

Returns a newly allocated (Glib) pointer.

`gi-pointer-inc` *pointer* [*#:offset %gi-pointer-size*] [Procedure]

Returns a foreign pointer object pointing to the address of *pointer* increased by *offset*.

`gi-attribute-iter-new` [Procedure]

Returns a pointer.

Creates and returns a foreign pointer to a C struct for a `GIAttributeIter` (a C struct containg four pointers, initialized to `%null-pointer`).

`with-gerror` *var body* [Procedure]

Returns the result of the execution of *body*, or raises an exception.

var must be an identifier. Evaluate *body* in a lexical environment where *var* is bound to a pointer to a newly allocated (and '`empty`') `GError`. *var* will always be freed. If no exception is raised, the result of the execution of *body* is returned.

`gi->scm` *value type* [*cmpl #f*] [Procedure]

Returns the scheme representation of *value*.

The *type*, a symbol name (also called a `type tag` or just a `tag` in the `GI` terminology) supported values are:

`'boolean` Calls [gi-boolean->scm], page 82.

`'string`
`'pointer` Calls [gi-string->scm], page 82, or [gi-pointer->scm], page 82.

`'n-string`
`'n-pointer`
`'n-gtype` Calls [gi-n-string->scm], page 82, [gi-n-pointer->scm], page 82, or [gi-n-gtype->scm], page 82.

The optional *cmpl* (complement) argument must be passed and set to the number of string(s), pointer(s) or gtype(s) contained in *value*, .

`'strings`
`'pointers`
Calls [gi-strings->scm], page 83, or [gi-pointers->scm], page 83.

`'csv-string`
Calls [gi-csv-string->scm], page 83.

`'glist`
`'gslist` Calls [gi-glist->scm], page 83, or [gi-gslist->scm], page 83, repsectively.

`gi-boolean->scm` *value* [Procedure]

Returns `#t` or `#f`.

The GType of *value* must be a `gboolean`.

`gi-string->scm` *value* [Procedure]
`gi-pointer->scm` *value* [Procedure]

Returns a string, a pointer or `#f` if *value* is the `%null-pointer`.

The GType of *value* must be a `gchar*` or a `gpointer`.

`gi-n-string->scm` *value n-string* [Procedure]
`gi-n-pointer->scm` *value n-pointer* [Procedure]
`gi-n-gtype->scm` *value n-gtype* [Procedure]

Returns a (possibly empty list) of string(s), pointer(s) or GType(s).

The GType of *value* must be a `gchar**`, a `gpointer[]` or a `GType[]`. The *n-string*, *n-pointer* and *n-gtype* argument must be the length of the *value* array.

`gi-strings->scm` *value* [Procedure]
`gi-pointers->scm` *value* [Procedure]
Returns a (possibly empty) list of strings or pointer.

The GType of *value* must be a `gchar**` or `gpointer[]`. The array must be NULL terminated.

`gi-csv-string->scm` *value* [Procedure]
Returns a list of string(s) or `#f` if *value* is the `%null-pointer`.

The GType of *value* is `gchar*`. Unless `#f`, the list of string(s) is obtained by splitting the (comma separated value) string pointed to by *value* using using `#\,` as the `char-pred`.

`gi-glist->scm` *g-list* [Procedure]
`gi-gslist->scm` *g-slist* [Procedure]
Returns a (possibly empty) list.

Obtains and returns a (possibly empty) list of the pointers stored in the `data` field of each element of *g-list* or *g-slist*.

`scm->gi` *value type* [*cmpl #f*] [Procedure]
Returns the GI representation of *value*.

The *type*, a symbol name (also called a `type tag` or just a `tag` in the `GI` terminology) supported values are:

`'boolean` Calls [scm->gi-boolean], page 83.

`'string`
`'pointer` Calls [scm->gi-string], page 83, or [scm->gi-pointer], page 83.

`'n-string`
`'n-pointer`
`'n-gtype` Calls [scm->gi-n-string], page 84, [scm->gi-n-pointer], page 84, or [scm->gi-n-gtype], page 84.

The optional *cmpl* (complement) argument may be passed and set to the number of string(s), pointer(s) or gtype(s) contained in *value*.

`'strings`

`'pointers`
Calls [scm->gi-strings], page 84, or [scm->gi-pointers], page 84.

`'gslist` Calls [scm->gi-gslist], page 84.

`scm->gi-boolean` *value* [Procedure]
Returns `0` if *value* is `#f`, otherwise, it returns `1`.

`scm->gi-string` *value* [Procedure]
`scm->gi-pointer` *value* [Procedure]
Returns a pointer.

If *value* is `#f`, it returns `%null-pointer`. Otherwise, it returns a pointer to the string in *value* or *value*.

`scm->gi-n-string` *value* [*n-string #f*] [Procedure]
`scm->gi-strings` *value* [Procedure]
Returns two values.

If *value* is the empty list, it returns `%null-pointer` and an empty list. Otherwise, it returns a pointer to an array of pointer(s) to the string(s) in *value* and a list of the '`inner`' string pointer(s).

It is the caller's responsibility to maintain a reference to those inner pointer(s), until the array '`itself`' (the first returned value) is no longer needed/used.

The array returned by [scm->gi-strings], page 84, is `NULL` terminated, where as the array returned by [scm->gi-n-string], page 84, is not.

`scm->gi-n-pointer` *value* [*n-pointer #f*] [Procedure]
`scm->gi-n-gtype` *value* [*n-gtype #f*] [Procedure]
Returns a pointer.

If *value* is an empty list, it returns `%null-pointer`. Otherwise, it returns a pointer to an array the pointer(s) or GType(s) in *value*.

The returned array is not `NULL` nor `0-` terminated.

`scm->gi-pointers` *value* [Procedure]
Returns a pointer.

If *value* is an empty list, it returns `%null-pointer`. Otherwise, it returns a pointer to an array the pointer(s) in *value*.

The returned array is `NULL` terminated.

`scm->gi-gslist` *value* [Procedure]
Returns a pointer.

If *value* is an empty list, it returns `%null-pointer`. Otherwise, it returns a pointer to a `GSList`, with its element's data being (in order), the pointer(s) in *value*.

Types and Values

`%gi-pointer-size` [Variable]
The size (the number of bytes) that a (Glib) pointer occupies in memory (which is architecture dependent).

Support

G-Golf uses a series of support modules, each documented in the following subsections. You may either import them all, like this `(use-modules (g-golf support))`, or individually, such as `(use-modules (g-golf support modules))`, `(use-modules (g-golf support goops))`, ...

Module

x

`[re-export-public-interface], page 84`

`re-export-public-interface` . *args* [Special Form]
Re-export the public interface of a module or modules. Invoked as `(re-export-modules (mod1) (mod2)...)`.

Goops

Syntax, Procedures and Methods

`class-direct-virtual-slots` (*self* <*class*>) [Method]
Returns a list.

Obtains and returns the list of the class direct slots for *self* that satisfy the `(eq? (slot-definition-allocation slot) #:virtual)` predicate.

`class-virtual-slots` (*self* <*class*>) [Method]
Returns a list.

Obtains and returns the list of the class slots for *self* that satisfy the `(eq? (slot-definition-allocation slot) #:virtual)` predicate.

`class-direct-g-property-slots` (*self* <*class*>) [Method]
Returns a list.

Obtains and returns the list of the class direct slots for *self* that satisfy the `(eq? (slot-definition-allocation slot) #:g-property)` predicate.

`class-g-property-slots` (*self* <*class*>) [Method]
Returns a list.

Obtains and returns the list of the class slots for *self* that satisfy the `(eq? (slot-definition-allocation slot) #:g-property)` predicate.

`mslot-set!` *inst s1 v1 s2 v2 s3 v3 ...* [Procedure]
Returns nothing.

Performs a multiple `slot-set!` for *inst*, setting its slot named *s1* to the value *v1*, *s2* to *v2*, *s3* to *v3* ...

`generic?` *value* [Procedure]
Returns `#t` if *value* is a <*generic*> instance. Otherwise, it returns `#f`.

Enum

G-Golf class, accessors, methods and procedures to deal with C enum types.

Classes

Procedures, Accessors and Methods

Description

G-Golf class, accessors, methods and procedures to deal with C enum types.

Classes

`<enum>` [Class]

The `<enum>` class is for enumerated values. Its (unique) slot is:

`enum-set` #:accessor !enum-set
#:init-keyword #:enum-set

Notes:

- the `enum-set` can't be empty and so you must use the `#:enum-set` (#:init-keyword) when creating new `<enum>` instances;
- the `#:enum-set` (#:init-keyword) accepts either a list of symbols or a well-formed `enum-set`;
- a well-formed `enum-set` is a list of `(symbol . id)` pairs, where `id` is a positive integer.
- each `symbol` and each `id` of an `enum-set` must be unique.

Instances of the `<enum>` class are immutable (to be precise, there are not meant to be mutated, see [GOOPS Notes and Conventions], page 7, 'Slots are not Immutable').

`<gi-enum>` [Class]

The `<gi-enum>` class is a subclass of `<enum>`. Its `class-direct-slots` are:

`g-type` #:accessor !g-type
#:init-keyword #:g-type
#:init-value #f

`g-name` #:accessor !g-name
#:init-keyword #:g-name

name #:accessor !name

The `name` slot is automatically initialized.

Instances of the `<gi-enum>` class are immutable (to be precise, there are not meant to be mutated, see [GOOPS Notes and Conventions], page 7, 'Slots are not Immutable').

Procedures, Accessors and Methods

`!enum-set` (*inst* `<`*enum*`>`) [Accessor]

Returns the content of the enum-set slot for *inst*.

`enum->value` (*inst* `<`*enum*`>`) *symbol* [Method]
`enum->values` (*inst* `<`*enum*`>`) [Method]

Returns the *inst* value for *symbol* (or `#f` if it does not exists), or the list of all values for *inst*, respectively.

`enum->symbol` (*inst* `<`*enum*`>`) *value* [Method]
`enum->symbols` (*inst* `<`*enum*`>`) [Method]

Returns the *inst* symbol for *value* (or `#f` if it does not exists), or the list of all symbols for *inst*, respectively.

`enum->name` (*inst* `<`*enum*`>`) *value* [Method]
`enum->names` (*inst* `<`*enum*`>`) [Method]

Returns the *inst* name (the string representation of the symbol) for *value* (or `#f` if it does not exists), or the list of all names for *inst*, respectively.

value can either be a `symbol` or an `id`.

`!g-type` (*inst* `<`*gi-enum*`>`) [Accessor]
`!g-name` (*inst* `<`*gi-enum*`>`) [Accessor]
`!name` (*inst* `<`*gi-enum*`>`) [Accessor]

Returns the content of the g-type, g-name or name slot for *inst*, respectively.

Flag

G-Golf class, accessors, methods and procedures to deal with C flag types.

Classes

`[<gi-flag>], page 87`

Procedures

`[gi-integer->gflags], page 88`
`[gi-gflags->integer], page 88`

Description

G-Golf class, accessors, methods and procedures to deal with C flag types.

Classes

`<gi-flag>` [Class]

The `<gi-flag>` class is a subclass of `<gi-enum>`. Its has no direct slots.

Procedures

`gi-integer->gflags` *gflags n* [Procedure]
Returns a possibly empty) list of symbol(s).

Obtains and returns the list of (symbol) flags for the given `<gi-flag>` instance *gflags* and its integer representation *n*.

`gi-gflags->integer` *gflags flags* [Procedure]
Returns an integer.

Compute and returns the integer representation for the list of (symbol(s)) given by *flags* and the given `<gi-flag>` instance *gflags*.

Struct

G-Golf class, accessors, methods and procedures to deal with C struct types.

Classes

`[<gi-struct>], page 88`

Procedures and Accessors

`[!g-name_], page 89`
`[!name____], page 89`
`[!alignment], page 89`
`[!size], page 89`
`[!is-gtype-struct?], page 89`
`[!is-foreign?], page 89`
`[!field-types], page 89`
`[!scm-types], page 89`
`[!init-vals], page 89`
`[!is-opaque?], page 89`
`[!is-semi-opaque?], page 89`

Description

G-Golf class, accessors, methods and procedures to deal with C struct types.

Classes

`<gi-struct>` [Class]
The `<gi-struct>` class is a subclass of `<struct>`. Its `class-direct-slots` are:

g-name #:accessor !g-name
#:init-keyword #:g-name

name #:accessor !name

alignment
#:accessor !alignment
#:init-keyword #:alignment

size #:accessor !size
#:init-keyword #:size

is-gtype-struct?
#:accessor !is-gtype-struct?
#:init-keyword #:is-gtype-struct?

field-types
#:accessor !field-types
#:init-keyword #:field-types

scm-types
#:accessor !scm-types

init-vals
#:accessor !init-vals

is-opaque?
#:accessor !is-opaque?

is-semi-opaque
#:accessor !is-semi-opaque?

The `name` and `scm-types` slots are automatically initialized.

Instances of the `<gi-struct>` are immutable (to be precise, there are not meant to be mutated, see [GOOPS Notes and Conventions], page 7, 'Slots are not Immutable').

Procedures and Accessors

`!g-name` (*inst* `<`*gi-struct*`>`) [Accessor]
`!name` (*inst* `<`*gi-struct*`>`) [Accessor]
`!alignment` (*inst* `<`*gi-struct*`>`) [Accessor]
`!size` (*inst* `<`*gi-struct*`>`) [Accessor]
`!is-gtype-struct?` (*inst* `<`*gi-struct*`>`) [Accessor]
`!field-types` (*inst* `<`*gi-struct*`>`) [Accessor]
`!scm-types` (*inst* `<`*gi-struct*`>`) [Accessor]
`!init-vals` (*inst* `<`*gi-struct*`>`) [Accessor]
Returns the content of their respective slot for *inst*.

`!is-opaque?` (*inst* `<`*gi-struct*`>`) [Accessor]
Returns `#t` if *inst* is '`opaque`', otherwise, it returns `#f`.

A <gi-struct> instance is said to be '`opaque`' when the call to `g-struct-info-get-size` upon its GIStructInfo pointer returns `zero`. In scheme, these <gi-struct> instances have no fields.

'`Opaque`' boxed types should never be '`decoded`', nor '`encoded`'. Instead, procedures, accessors and methods should '`blindingly`' receive, pass and/or return their pointer(s).

`!is-semi-opaque?` (*inst* `<`*gi-struct*`>`) [Accessor]
Returns `#t` if *inst* is '`semi-opaque`', otherwise, it returns `#f`.

A <gi-struct> instance is said to be 'semi-opaque' when one of its field types is void.

'Semi-opaque' boxed types should never be 'decoded', nor 'encoded'. Instead, procedures, accessors and methods should 'blindingly' receive, pass and/or return their pointer(s).

Union

G-Golf class, accessors, methods and procedures to deal with C union types.

Classes

[<gi-union>], page 90

Procedures, Accessors and Methods

[make-c-union], page 91
[c-union-ref], page 91
[c-union-set!], page 91
[!g-type__], page 91
[!g-name__], page 91
[!name___], page 91
[!size_], page 91
[!alignment_], page 91
[!fields], page 91
[!is-discriminated?], page 91
[!discriminator-offset], page 91
[!discriminator], page 91

Description

G-Golf class, accessors, methods and procedures to deal with C union types.

Classes

<gi-union> [Class]

The <gi-union> class. Its class-direct-slots are:

g-type #:accessor !g-type
#:init-keyword #:g-type

g-name #:accessor !g-name
#:init-keyword #:g-name

name #:accessor !name

size #:accessor !size
#:init-keyword #:size

alignment
#:accessor !alignment
#:init-keyword #:alignment

fields #:accessor !fields
#:init-keyword #:fields

is-discrimanted?
#:accessor !is-discriminated?
#:init-keyword #:is-discriminated?

discriminator-offset
#:accessor !discriminator-offset
#:init-keyword #:discriminator-offset

discriminator
#:accessor !discriminator #:init-keyword #:discriminator #:init-value #f

The `name` slot is automatically initialized.

Instances of the `<gi-union>` are immutable (to be precise, there are not meant to be mutated, see [GOOPS Notes and Conventions], page 7, 'Slots are not Immutable').

Procedures, Accessors and Methods

`make-c-union` *types* [*type #f*] [*val #f*] [Procedure]
Returns a pointer.

Create a foreign pointer to a C union for the list of *types* (see Foreign Types (`https://www.gnu.org/software/guile/manual/guile.html#Foreign-Types`) in the Guile Reference Manual for a list of supported types).

`c-union-ref` *foreign size type* [Procedure]
Returns the content of the C union pointed by *foreign*, for the given *size* and *type*.

`c-union-set!` *foreign size type val* [Procedure]
Returns nothing.

Sets the content of the C union pointed by *foreign* to *val*, given its *size* and *type*.

`!g-type` (*inst* <*gi-union*>) [Accessor]
`!g-name` (*inst* <*gi-union*>) [Accessor]
`!name` (*inst* <*gi-union*>) [Accessor]
`!size` (*inst* <*gi-union*>) [Accessor]
`!alignment` (*inst* <*gi-union*>) [Accessor]
`!fields` (*inst* <*gi-union*>) [Accessor]
`!is-discriminated?` (*inst* <*gi-union*>) [Accessor]
`!discriminator-offset` (*inst* <*gi-union*>) [Accessor]
`!discriminator` (*inst* <*gi-union*>) [Accessor]
Returns the content of their respective slot for *inst*.

Utilities

Procedures

Variables

Description

G-Golf utilities low level API.

Procedures

`g-studly-caps-expand` *str* [Procedure]

Returns a string.

Given a '`Camel Case (https://en.wikipedia.org/wiki/Camel_case)`' string, this procedure[14] returns a new string, with all '`_`' transformed into '`-`', uppercase letters are transformed into their corresponding lowercase letter, and a `#\-` is inserted in between occurrences of two consecutive uppercase letters, unless the sequence analysed is part of a prefix defined in the [%g-studly-caps-expand-token-exceptions], page 94, alist.

Here are two examples:

```
(g-studly-caps-expand "GStudlyCapsExpand")
⊣
$2 = "g-studly-caps-expand"
```

```
(g-studly-caps-expand "WebKitWebContext")
⊣
$3 = "webkit-web-context"
```

`g-name->name` *g-name* [*as-string? #f*] [Procedure]

`g-name->class-name` *g-name* [Procedure]

Return a symbol name (or a string for the former, if the optional parameter is not `#f`).

[14] This procedure, as well as [g-name->name], page 92, and [g-name->class-name], page 92, come from Guile-Gnome (`https://www.gnu.org/software/guile-gnome`), where there are named `GStudlyCapsExpand`, `gtype-name->scm-name` and `gtype-name->class-name`, in the (Guile-Gnome) module (gnome gobject utils). In G-Golf, these will also be used to transform other (Gobject Inptrospection given) names, such as function names, hence their `g-name->` prefix instead

The former first obtains a string, the scheme representation for *g-name*, by looking for a possible entry in [%g-name-transform-exceptions], page 94, then, if it failed, by calling [g-studly-caps-expand], page 92. It then either returns that string, if the optional *as-string?* is not `#f`, or its symbol name, by calling `string->symbol`.

The later uses the former, surrounds the result using `#\<` and `#\>` characters then calls `string->symbol`. For example:

```
(g-name->class-name "ClutterActor")
⊣
$2 = <clutter-actor>
```

`gi-type-tag->ffi` *type-tag* [Procedure]

Returns an integer or `'*` (the symbol `*`).

Obtains the correponding Guile's ffi tag value for *type-tag*, which must be a member of [%gi-type-tag], page 51. If *type-tag* is unknown, an exception is raised. Note that Guile's ffi tag values are integers or `'*` (the symbol `*`, used by convention to denote pointer types.

`gi-type-tag->init-val` *type-tag* [Procedure]

Returns the default init value for *type-tag*.

Obtains and returns the default init value for *type-tag*, which will either be `0` (zero), or `%null-pointer`.

`syntax-name->method-name` *name* [Procedure]

Returns a (symbol) name.

This procedure is used to '`protect`' syntax names, from being redefined as generic functions and methods.

Users should normally not call this procedure - except for testing purposes, if/when they customize its default settings - it is appropriately and automatically called by G-Golf when importing a GI typelib.

Unless otherwise specified (see [%gi-method-short-names-skip], page 94), when a GI typelib is imported, G-Golf also creates so called short name methods, obtained by dropping the container name (and its trailing hyphen) from the GI typelib method full/long names.

GI methods are added to their respective generic function, which is created if it does not already exist. When a generic function is created, G-Golf checks if the name is used, and when it is bound to a procedure, the procedure is 'captured' into an unspecialized method, which is added to the newly created generic function.

However, when the name is used but its variable value is a syntax, the above can't be done and the name must be '`protected`', which is what [syntax-name->method-name], page 93, does, using a renamer, or by adding a prefix, a postfix or both to its (symbol) *name* argument.

By default, the renamer ([%syntax-name-protect-renamer], page 94) and prefix ([%syntax-name-protect-prefix], page 94) variables are set to `#f`. The the postfix ([%syntax-name-protect-postfix], page 94) variable is set to `_` (the underscore symbol).

As an example, using these default settings, the short name method for `gcr-secret-exchange-begin` would be `begin_`.

Variables

`%g-name-transform-exceptions` [Variable]

Contains an alist where each `key` is a `GType` name exception for the [g-name->name], page 92, procedure, and the corresponding `value` is the name [g-name->name], page 92, should use instead.

Its default value contains an entry for `GObject`, which should not (never) be removed:

```
(define %g-name-transform-exceptions
  '(("GObject" . "gobject")))
```

`%g-studly-caps-expand-token-exceptions` [Variable]

Contains an alist where each `key` is a `token` exception for the [g-studly-caps-expand], page 92, procedure, and the corresponding `value` the string that [g-studly-caps-expand], page 92, will use for that `token` transformation instead.

Its default value contains an entry for the `WebKit` token:

```
(define %g-studly-caps-expand-token-exceptions
  '(("WebKit" . "webkit")))
```

Users may add or remove alist pairs to satisfy their needs.

`%gi-method-short-names-skip` [Variable]

This variable is used by G-Golf to decide, while importing a GI typelib, if a short name method should be created or not ('`skipped`', hence the variable name).

It can take the following values:

`'()` This is the default value. In this case, a short name method is created for every full/long name method, or in other words, no short name method creation process is skipped.

`'all` In this case, no short name method is created, or in other words, all short name method creation process is skipped.

`a list of short name(s)`
In this case, for each short name in the list, the short name method creation process is skipped.

`%syntax-name-protect-prefix` [Variable]
`%syntax-name-protect-postfix` [Variable]
`%syntax-name-protect-renamer` [Variable]

These variables are used by [syntax-name->method-name], page 93, and may be customized. Their default values are:

%syntax-name-protect-prefix #f
%syntax-name-protect-postfix '_ (the underscore symbol)
%syntax-name-protect-renamer #f

%syntax-name-protect-prefix and *%syntax-name-protect-postfix* may be defined as `#f` or a symbol name. Unless a renamer is set, at least one of these two variables must be defined as a symbol name.

The *%syntax-name-protect-renamer* may be defined as `#f` or a procedure, that takes one argument - a symbol name - and returns a symbol name.

[syntax-name->method-name], page 93, first checks for a *%syntax-name-protect-renamer*, and calls it if it has been defined, igoring the other variables.

Otherwise, [syntax-name->method-name], page 93, returns a symbol name prefixed using *%syntax-name-protect-prefix* when not `#f` and/or postfixed using *%syntax-name-protect-postfix* when not `#f`. As mentioned above, unless a renamer is set, at least one of these two variables must be defined as a symbol name.

G-Golf High Level API

G-Golf High Level API modules are defined in the `hl-api` subdirectory, such as `(g-golf hl-api gobject)`.

Where you may load these modules individually, the easiest way to use the G-Golf High Level API is to import the `hl-api` module: it imports and re-exports the public interface of (oop goops), some G-Golf support modules and all G-Golf High Level API modules:

```
(use-modules (g-golf hl-api))
```

As stated in the introduction, G-Golf high level API (main) objective is to make (imported) GOBject classes and methods available using GOOPS, the Guile Object Oriented System (see Section “GOOPS” in *The GNU Guile Reference Manual*), in a way that is largely inspired by Guile-Gnome (`https://www.gnu.org/software/guile-gnome`).

Events

G-Golf Events high level API.
The G-Golf integration with Gdk Events.

Class

`[<gdk-event>], page 96`

Accessors and Methods

Description

This section describes classes, procedures, accessors and methods for the G-Golf integration with events from the window system.

Classes

`<gdk-event>` [Class]

The superclass of all Gdk type of event. Its slots are:

event #:accessor !event
#:init-keyword #:event
A pointer to a `GdkEvent`.

Accessors and Methods

`!event` (*inst* <*gdk-event*>) [Accessor]

Returns the content of the `event` slot for *inst*.

`!button` (*inst* <*gdk-event*>) [Method]
`!click-count` (*inst* <*gdk-event*>) [Method]
`!coords` (*inst* <*gdk-event*>) [Method]
`!x` (*inst* <*gdk-event*>) [Method]
`!y` (*inst* <*gdk-event*>) [Method]
`!keycode` (*inst* <*gdk-event*>) [Method]
`!keyval` (*inst* <*gdk-event*>) [Method]
`!keyname` (*inst* <*gdk-event*>) [Method]
`!root-coords` (*inst* <*gdk-event*>) [Method]
`!root-x` (*inst* <*gdk-event*>) [Method]

`!root-y` (*inst* <*gdk-event*>) [Method]
`!state` (*inst* <*gdk-event*>) [Method]
`!time` (*inst* <*gdk-event*>) [Method]
`!window` (*inst* <*gdk-event*>) [Method]
`!type` (*inst* <*gdk-event*>) [Method]
`!changed-mask` (*inst* <*gdk-event*>) [Method]
`!new-window-state` (*inst* <*gdk-event*>) [Method]

Respectively returns the scheme representation of the content of the *inst* event (struct) element - refered to by its name - or #f if the event (struct) does not deliver the element.

The event (struct) elements are:

`button` The button number of the event.

`click-count`
The click-count of the event.

`coords` The list of the x and y window relative coordinates of the event.

`x` The x window relative coordinate of the event.

`y` The y window relative coordinate of the event.

`keycode` The raw code (also called hardware keycode) of the key that was pressed of released.

`keyval` The key value that was pressed of released (See the gdk/gdkkeysyms.h header file for a complete list of GDK key codes.

`keyname` The key name that was pressed of released (There is actually no such element in any (gdk) event, this method calls [gdk-keyval-name], page 48, on the keyval of the event).

`root-coords`
The list of the x and y root window relative coordinates of the event.

`root-x` The root x window relative coordinate of the event.

`root-y` The root y window relative coordinate of the event.

`state` A list representing the state of the modifier keys (e.g. Control, Shift and Alt) and the pointer buttons (see the [gdk-event-get-state], page 43, description for some more information about `GdkModifierType`).

`time` The time of the event in milliseconds.

`window` The (a pointer to the) `GdkWindow` of the event.

`type` The type of the event.

`changed-mask`
The mask specifying what flags have changed.

new-window-state
: The new window state, a combination of [%gdk-window-state], page 47, bits.

GType

G-Golf GType high level API.
The base of the GObject type system.

Classes

Accessors and Methods

`<gtype-class>` [Class]
The `metaclass` of all GType classes. Ensures that GType classes have an `info` slot, holding a pointer to a `GIObjectInfo`. Its slots are:

info #:accessor !info
#:init-keyword #:info

namespace
#:accessor !namespace

g-type #:accessor !g-type

g-name #:accessor !g-name

g-class #:accessor !g-class

derived #:accessor !derived
#:init-keyword #:derived
#:init-value #f
A class is derived when it is user defined (not imported), and inherit a GObject subclass.

The *#:info* #:init-keyword is mandatory, other slots are initialized automatically. All slots are immutable (to be precise, they are not meant to be mutated, see [GOOPS Notes and Conventions], page 7, 'Slots are not Immutable').

`!info` *class* [Accessor]
`!namespace` *class* [Accessor]
`!g-type` *class* [Accessor]
`!g-name` *class* [Accessor]
`!g-class` *class* [Accessor]
`!derived` *class* [Accessor]

Returns the content of their respective slot for *class*.

`<gtype-instance>` [Class]

The root class of all instantiatable GType classes. Adds a slot, `g-inst`, to instances, which holds a pointer to the C value

g-inst #:accessor !g-inst

The *g-inst* slot is initialized automatically and immutable (to be precise, it is not meant to be mutated, see [GOOPS Notes and Conventions], page 7, 'Slots are not Immutable').

`!g-inst` *instance* [Accessor]

Returns the content of the *g-inst* slot for *instance*.

`unref` *instance* [Method]

Returns nothing.

This method calls [g-object-unref], page 29, on the `g-inst` of *instance*.

When the reference count for the `g-inst` reaches 0 (zero), it sets the `g-inst` slot value for *instance* to #f and removes *instance* from the `%g-inst-cache`.

This method must be called upon instances that are not referenced anywhere anymore, so that their memory can be freed by the next gc occurrence.

GObject

G-Golf GObject high level API.
The G-Golf integration with the GObject object system.

Classes

`[<gobject>]`, page 100

Procedures

`[gobject-class?]`, page 100

Description

Classes

`<gobject>` [Class]
The base class for GLib's default object system.

`gobject-class?` *val* [Procedure]
Returns #t if *val* is a class and if `<gobject>` is a member of its class precedence list. Otherwise, it returns #f.

Closure

G-Golf closure high level API.
The G-Golf integration with GObject Closures.

Classes

`[<closure>], page 101`

Procedures and Methods

`[!g-closure], page 101`
`[!function], page 101`
`[!return-type], page 101`
`[!param-types], page 101`
`[invoke], page 101`

Description

The GLib/GObject type system supports the creation and invocation of '`Closures`', which represents a callback supplied by the programmer (see [Closures], page 37, if you are curious about the low-level description and API, though you don't need to to understand and use the high level API described here).

Its infrastructure allows one to pass a Scheme function to C, and have C call into Scheme, and vice versa. In Scheme, a `<closure>` instance holds a pointer to a `GClosure` instance, a Scheme procedure, the type of its return value, and a list of the type of its arguments.

Closures can be invoked with [invoke], page 101, for example:

```
,use (g-golf)

(make <closure>
      #:function (lambda (a b) (+ a b))
      #:return-type 'int
      #:param-types '(int int))
⊣
$2 = #<<closure> 55f24a0228d0>

(invoke $2 3 2)
⊣
$3 = 5
```

Classes

`<closure>` [Class]

Its slots are:

g-closure
#:accessor !g-closure

function #:accessor !function
#:init-keyword #:function

return-type
#:accessor !return-type
#:init-keyword #:return-type

param-types
#:accessor !param-types
#:init-keyword #:param-types

The `#:return-type` and `#:param-types` accept respectively one symbol and a list of symbols that are members of the [%g-type-fundamental-types], page 27.

Instances of the `<closure>` class are immutable (to be precise, there are not meant to be mutated, see [GOOPS Notes and Conventions], page 7, 'Slots are not Immutable').

Accessors and Methods

Note: in this section, the *closure* argument is [must be] a `<closure>` instance.

`!g-closure` *closure* [Accessor]
`!function` *closure* [Accessor]
`!return-type` *closure* [Accessor]
`!param-types` *closure* [Accessor]

Returns the content of their respective slot for *closure*.

`invoke` *closure* . *args* [Method]

Returns the result of the invocation of *closure*, using (the possibly empty list of) *args*.

This is a '`low level`' method, not used internally, provided mainly for debugging (or demonstration) purposes, so you may test and verify your callbacks and signals procedures[15].

Function

G-Golf GI function and argument high level API.
The G-Golf GI function and argument high level API.

Classes

`[<function>]`, page 104
`[<argument>]`, page 105

[15] From scheme, you would '`immediately`' call the procedure instead of course.

Accessors and Methods

Variables

[%gi-strip-boolean-result], page 108

Classes

<function> [Class]

Its slots are:

info #:accessor !info

namespace #:accessor !namespace

g-name #:accessor !g-name

name #:accessor !name

override? #:accessor !override?

i-func #:accessor !i-func

o-func #:accessor !o-func

o-spec-pos #:accessor !o-spec-pos

flags #:accessor !flags

is-method? #:accessor !is-method

n-arg #:accessor !n-arg

caller-owns #:accessor !caller-owns

return-type #:accessor !return-type

type-desc #:accessor !type-desc

may-return-null? #:accessor !may-return-null?

arguments #:accessor !arguments

n-gi-arg-in #:accessor !n-gi-arg-in

args-in #:accessor !args-in

gi-args-in #:accessor !gi-args-in

gi-args-in-bv #:accessor !gi-args-in-bv

n-gi-arg-out
#:accessor !n-gi-arg-out

args-out #:accessor !args-out

gi-args-out
#:accessor !gi-args-out

gi-args-out-bv
#:accessor !gi-args-out-bv

gi-arg-result
#:accessor !gi-arg-result

Instances of the `<function>` class are immutable (to be precise, there are not meant to be mutated, see [GOOPS Notes and Conventions], page 7, 'Slots are not Immutable').

`<argument>` [Class]

Its slots are:

g-name #:accessor !g-name
#:init-keyword #:g-name

name #:accessor !name
#:init-keyword #:name

closure #:accessor !closure

destroy #:accessor !destroy

direction
#:accessor !direction
#:init-keyword #:direction

transfert
#:accessor !transfert

scope #:accessor !scope

type-tag #:accessor !type-tag
#:init-keyword #:type-tag

type-desc
#:accessor !type-desc
#:init-keyword #:type-desc

forced-type
#:accessor !forced-type
#:init-keyword #:forced-type

string-pointer
#:accessor !string-pointer

is-pointer?
#:accessor !is-pointer?
#:init-keyword #:is-pointer?

may-be-null?
#:accessor !may-be-nul?
#:init-keyword #:may-be-null?

is-caller-allocate?
#:accessor !is-caller-allocate?

is-optional?
#:accessor !is-optional?

is-return-value?
#:accessor !is-return-value?

is-skip? #:accessor !is-skip?

arg-pos #:accessor !arg-pos
#:init-keyword #:arg-pos

gi-argument-in
#:accessor !gi-argument-in
#:init-value #f

gi-argument-in-bv-pos
#:accessor !gi-argument-in-bv-pos
#:init-value #f

gi-argument-out
#:accessor !gi-argument-out
#:init-value #f

gi-argument-out-bv-pos
#:accessor !gi-argument-out-bv-pos
#:init-value #f

name #:accessor !gi-argument-field
#:init-keyword #:gi-argument-field

Instances of the `<argument>` class are immutable (to be precise, there are not meant to be mutated, see [GOOPS Notes and Conventions], page 7, 'Slots are not Immutable').

Accessors and Methods

Note: in this section, the *function* and *argument* arguments are [must be] a `<function>` and an `<argument>` instance, respectively.

`!info` *function* [Accessor]
`!namespace` *function* [Accessor]
`!g-name` *function* [Accessor]
`!name` *function* [Accessor]
`!override?` *function* [Accessor]
`!i-func` *function* [Accessor]
`!o-func` *function* [Accessor]
`!o-spec-pos` *function* [Accessor]
`!flags` *function* [Accessor]

`!is-method?` *function* [Accessor]
`!n-arg` *function* [Accessor]
`!caller-owns` *function* [Accessor]
`!return-type` *function* [Accessor]
`!type-desc` *function* [Accessor]
`!may-return-null` *function* [Accessor]
`!arguments` *function* [Accessor]
`!n-gi-arg-in` *function* [Accessor]
`!args-in` *function* [Accessor]
`!gi-args-in` *function* [Accessor]
`!gi-args-in-bv` *function* [Accessor]
`!n-gi-arg-out` *function* [Accessor]
`!args-out` *function* [Accessor]
`!gi-args-out` *function* [Accessor]
`!gi-args-out-bv` *function* [Accessor]
`!gi-arg-result` *function* [Accessor]

Returns the content of their respective slot for *function*.

`!g-name` *argument* [Accessor]
`!name` *argument* [Accessor]
`!closure` *argument* [Accessor]
`!destroy` *argument* [Accessor]
`!direction` *argument* [Accessor]
`!transfert` *argument* [Accessor]
`!scope` *argument* [Accessor]
`!type-tag` *argument* [Accessor]
`!type-desc` *argument* [Accessor]
`!forced-type` *argument* [Accessor]
`!string-pointer` *argument* [Accessor]
`!is-pointer?` *argument* [Accessor]
`!may-be-null?` *argument* [Accessor]
`!is-caller-allocate?` *argument* [Accessor]
`!is-optional?` *argument* [Accessor]
`!is-return-value?` *argument* [Accessor]
`!is-skip?` *argument* [Accessor]
`!arg-pos` *argument* [Accessor]
`!gi-argument-in` *argument* [Accessor]
`!gi-argument-in-bv-pos` *argument* [Accessor]
`!gi-argument-out` *argument* [Accessor]
`!gi-argument-out-bv-pos` *argument* [Accessor]
`!gi-argument-field` *argument* [Accessor]

Returns the content of their respective slot for *argument*.

Variables

`%gi-strip-boolean-result` [Variable]

A list of procedure and method names that that have at least one `'inout` or `'out` argument(s) and return either `#t` or `#f`, solely to indicate that the procedure or method call was successful or not.

These procedures and methods, if (and only if) their name is a member of *%gi-strip-boolean-result*, will see their returned valued eluded if it is `#t`, otherwise, an exception will be raised[16].

Initially, *%gi-strip-boolean-result* is empty, and it is a user responsibility to fill it appropriately, for each namespace they are importing.

Here is a concrete example, for the `"Clutter"` namespace and the `clutter-color-from-string` procedure:

```
,use (g-golf)
(gi-import "Clutter")
(clutter-color-from-string "Blue")
⊣
$2 = #t
$3 = (0 0 255 255)
```

And call it with an undefined color name:

```
(clutter-color-from-string "Bluee")
⊣
$4 = #f
$5 = (0 0 0 0)
```

Now, let's add this procedure name to *%gi-strip-boolean-result*:

```
(push! 'clutter-color-from-string
       %gi-function-call-strip-boolean-result)
$6 = (clutter-color-from-string)

(clutter-color-from-string "Blue")
⊣
$7 = (0 0 255 255)
```

And call it with an undefined color name:

```
(clutter-color-from-string "Bluee")
⊣
scm-error" "clutter-color-from-string" failed."

Entering a new prompt.  Type ‘,bt’ for a backtrace or ‘,q’ to continue.
```

[16] In any other situation, but void, the returned value comes first, then in order, if any, the 'inout and/or 'out argument(s).

Import

G-Golf Import high level API.
The G-Golf GIR namespace (Typelib) import high level API.

Procedures

Variables

Procedures

`gi-import` *namespace* [Procedure]

Returns nothing.

Imports the *namespace* GIR Typelib and exports its interface. For example:

```
,use (g-golf
(gi-import "Clutter")
```

The *namespace* is a case sensitive string. It is an error to call this procedure using an invalid *namespace*.

This procedure is certainly one of the first thing you will want to try and use, but it has a cost: you will not '`feel it`' if the number of objects in *namespace* is relatively small, but importing the "Gtk" namespace, on a laptop equiped with a i5-2450M CPU 2.50GHz × 4 and 6GB of memory takes nearly 2 seconds.

So, either early in the development cycle, or when your application is more stable, at your best convenience, you may consider making a series of selective import instead, see [gi-import-by-name], page 109, here below.

`gi-import-by-name` *namespace name* [*#:with-method #t*] [Procedure]

Returns the object or constant returned by [gi-import-info], page 110, called upon the GIBaseInfo `info` named *name* in *namespace*.

Obtains and imports the GIBaseInfo `info` named *name* in *namespace*. The *namespace* and *name* arguments are case sensitive. It is an error to call this procedure using an invalid *namespace* or *name*.

The optional keyword *#:with-method* argument - which is #t by default - is passed to the `gi-import-enum`, `gi-import-flag` and `gi-import-struct`. When *#:with-method* is #f, then the enum, flag or struct `info` will be imported without their respective methods. This is likely to only be the case if/when you intend to selectively

import an enum, gflag or struct from GLib or GObject, which is what G-Golf itself does, for example, in the top level (g-golf) module:

```
(gi-import-by-name "GLib" "IOChannel" #:with-method #f)
```

`gi-import-info` *info* [Procedure]

Returns the object or constant returned by the one of the `gi-import-enum`, `gi-import-flag`, ..., called upon `info`.

Obtains the GIBaseInfo type for `info` and uses it to dispatch a call to `gi-import-enum`, `gi-import-enum`, ..., and returns the object or constant returned by the procedure that has been called.

You probably will prefer to call [gi-import-by-name], page 109, most of the time, but here is a example:

```
,use (g-golf)
(g-irepository-require "Clutter")
$2 = #<pointer 0x5642cb065e30>

(g-irepository-find-by-name "Clutter" "ActorFlags")
$3 = #<pointer 0x5642cb067de0>

(gi-import-info $3)
$4 = #<<gi-flag> 5642cb13c5d0>

(describe $4)
#<<gi-flag> 5642cb13c5d0> is an instance of class <gi-flag>
Slots are:
     enum-set = ((mapped . 2) (realized . 4) (reactive . 8) (visible . 16) (no-la
     g-type = 94844874149456
     g-name = "ClutterActorFlags"
     name = clutter-actor-flags
```

`gi-import-enum` *info* [*#:with-method #t*] [Procedure]
`gi-import-flag` *info* [*#:with-method #t*] [Procedure]
`gi-import-struct` *info* [*#:with-method #t*] [Procedure]

Returns a [<gi-enum>], page 86, a [<gi-flag>], page 87, or a [<gi-struct>], page 88, instance, respectively.

The *info* argument is (must be) a pointer to `GIEnumInfo`, a `GIEnumInfo` for which `([g-base-info-get-type], page 53, info)` returned `'flags` and a `GIStructInfo` respectively. It is an error to call any of these procedures upon an invalid *info* argument.

The optional keyword *#:with-method* argument - which is #t by default - is passed using #f, then *info* will be imported without its respective methods. A description and an example ware also given here above, as part of the [gi-import-by-name], page 109, documentation entry.

Every imported [<gi-enum>], page 86, [<gi-flag>], page 87, and [<gi-struct>], page 88, instance is cached under the `'enum`, `'flag` and `'boxed` main key (respectively), using

the content of their (symbol) `name` slot as the secondary key. For example, reusing the "Clutter" "ActorFlags" namespace/name introduced above, you would retreive its [<gi-flag>], page 87, instance as is:

```
...
(gi-cache-ref 'flag 'clutter-actor-flags)
$6 = #<<gi-flag> 5642cb13c5d0>
```

`gi-import-function` *info* [Procedure]

Returns a [<function>], page 104, instance.

Imports *info* - a pointer to a `GIFunctionInfo` (see [Function Info], page 56), which represents a function, a method or a constructor - in Guile and exports its interface. This procedure also imports, recursively (and exports the interface of) its argument's type(s) and method(s).

Every imported function, method and constructor is cached under `'function` main key, and using the value of their [<function>], page 104, instance `name` slot as the secondary key. Here is an example:

```
,use (g-golf)
(g-irepository-require "Clutter")
$2 = #<pointer 0x55c191f3fe30>

(g-irepository-find-by-name "Clutter" "init")
$3 = #<pointer 0x55c191f41de0>

(gi-import-function $3)
$4 = #<<function> 55c191e81510>

(describe $4)
#<<function> 55c191e81510> is an instance of class <function>
Slots are:
     info = #<pointer 0x55c191f41de0>
     name = clutter-init
     flags = ()
     n-arg = 2
     caller-owns = nothing
     return-type = interface
...

(gi-cache-ref 'function 'clutter-init)
$5 = #<<function> 55c191e81510>
```

Returned value(s):

In most situations, but when the `return-type` is `'void` (in which case nothing is returned), the function or method returned value comes first, then in order, if any, the `'inout` and/or `'out` argument(s).

However, some function and method, that have at least one `'inout` or `'out` argument(s), do return a `'boolean`, but solely to indicate that the function or method

call was successful or not. It is only if the call is successful that the `'inout` and/or `'out` argument(s) have been '`correctly`' set and may be safely used.

In scheme, when binding such a function or method, we would rather (a) when the call is successful, elude the boolean and return, in order, the `'inout` and/or `'out` argument(s) value(s); and (b), when the call is unsuccessful, raise an exception.

Since it is not possible to automatically '`detect`' these functions and methods, G-Golf defines a [%gi-strip-boolean-result], page 108, variable, initially empty, that users may fill appropriately, using the function or method (symbol) name, as described in its documentation: make sure to carefully read and understand it.

`gi-import-constant` *info* [Procedure]

Returns two values, the constant value and its name.

Obtains and returns the *info* constant value and its name. For example:

```
,use (g-golf)
(g-irepository-require "GLib")
#<pointer 0x55ad58e6ae00>

(g-irepository-find-by-name "GLib" "PRIORITY_DEFAULT_IDLE")
$3 = #<pointer 0x55ad58e6cde0>

(gi-import-constant $3)
$4 = 200
$5 = "PRIORITY_DEFAULT_IDLE"
```

Constants are curently not being automatically imported, though this will probably change in the near future, stay tuned.

Variables

`%gi-base-info-types` [Variable]

`%gi-imported-base-info-types` [Variable]

A (cumulative) list of the distinct (top level) base info types contained in the imported namespace(s).

These two variables have no other purpose then offering a feedback about: (a) the (top level) base info types contained in the namespace(s) passed to [gi-import], page 109; (b) the (top level) base info types that have effectively been imported - when `G-Golf` is complete, both lists should be identical.

Initially, these variables are empty. As [gi-import], page 109, [gi-import-info], page 110, and/or [gi-import-by-name], page 109, are being called, they are filled with new types, which are added to both lists.

Note that theorder in which base info types appear in these two lists is rrelevant, and may slightly vary, depending on the order of the namespace used for the successive [gi-import], page 109, calls and how complete is `G-Golf`.

Utilities

G-Golf additional utilities.

Procedures

[gi-find-by-property-name], page 113

Description

G-Golf additional utilities.

Procedures

`gi-find-by-property-name` *namespace name* [Procedure]

Returns a (possibly empty) list.

Obtains and returns a (possibly empty) list of (pointers to) `GIObjectInfo` in *namespace* that have a property named *name*. Property names are obtained calling g-base-info-get-name, with no translation/transformation - underscore, if any, are kept 'as is', and the comparison with *name* is case sensitive.

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

   A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

   Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or

processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant

Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

\input texinfo

Concept Index

This index contains concepts, keywords and non-Schemey names for several features, to make it easier to locate the desired sections.

C

G

L

Procedure Index

This is an alphabetical list of all the procedures, methods and macros in G-Golf.

!

C

E

G

I

M

R

S

U

W

Variable Index

This is an alphabetical list of all the important variables and constants in G-Golf.

Type Index

This is an alphabetical list of all the important data types defined in the G-Golf Programmers Manual.

List of Examples