Black Hole

Module system for Gambit

Core documentation, revision 1.

Documentation distributed in PDF, OpenOffice and Microsoft Word versions.

Please email any feedback, questions and suggestions to the authors.

Table of Contents

Introduction

Black Hole ("BH") is a module system abstraction that provides recurring module file dependency compilation and loading, including export of macros, for regular R5RS Scheme.

It is built atop Gambit's `(load)`, `(compile-file)` and namespace and macro expansion.

BH requires no runtime library or alike for output code to execute, it is a code processor only.

BH is installed into the Gambit environment (i.e. `gsc`) through that BH's main file is `(load)`:ed upon each start of Gambit. Typically this is done by creating a shellscript by the name of `bsc`, that starts `gsc` with arguments specifying that BH should be loaded on startup. More on this later.

BH requires Gambit version 4.4.2 or later.

Concepts

BH works with Scheme code at the level of granularity of modules. Each module is typically one Scheme file.

Modules are subsectioned into module-resolvers. A module-resolver is a function that gives BH a module's sourcecode, given its name.

One thing the concept of module-resolvers provides with, is for all modules not to be required to be located in or under one single root directory (i.e. say the/home/user/project-name directory, with all Scheme files directly put into it).

BH is bundled with the startup default module-resolver, that takes files from the current directory, and with the `std` module-resolver containing the bundled libraries.

Quick tour

As soon as you have BH installed and up running, just type `bsc` in your shell. You get the REPL prompt, `>`.

The `(import)` form loads modules. Its calling convention is `(import (module-resolver argument-to-module-resolver [another-argument [..]]) [(another-module-resolver another-argument-to-module-resolver [..]) [..]).`

If you evaluate `string-upcase`, you of course get an error, since SRFI 13 is not loaded.

Evaluate `(import (std srfi/13))` to load it. Now evaluate `(string-upcase "Try`

`this")`, and see that it is loaded.

Into the current directory, put the file ”a.scm” with the following contents:

```
(import (std srfi/13))
(define (proc)
  (let loop ((x 10000))
    (string-upcase "s")
    (if (not (zero? x))
      (loop (- x 1))
      "Hello world")))
```

Evaluate `(import a) (proc)` in the REPL.

However, running it will take some time as it is executed in interpreted mode, you can see how much by evaluating `(time (proc))`.

Now do `(module-compile! 'a)` to compile it, `(import a)` to reimport it and then `(time (proc))` again, and you will see that the numbers are much lower.

Create the file “b.scm” in the same directory, with the contents:

```
(import a)
(define mirror proc)
(print "Module B\n")
```

When evaluating `(import b)` the first time, `Module B` will be printed to the console. Evaluating `(mirror)` gives the same result as evaluating `(proc)`.

What it does

BH automatically manages symbol export/import between modules – variables, procedures and macros. Functionality exists to import symbols from modules under other names than the names under which they are exported, i.e. the symbol `a` of module `b` can be imported under the name `c` into your module or REPL.

BH automatically loads dependencies on `(import)`. Upon load, if a compiled version of the module exists, it is loaded rather than the source file. If the compiled version is older than the source file, the module is automatically recompiled.

For macros, `define-syntax` / `let-syntax` / `letrec-syntax` , `define-macro` and `syntax-rules` / `sc-macro-transformer` are supported.

Additionally, `rsc-macro-transformer` (a variant of `sc-macro-transformer`) and `nh-macro-transformer` (using which `define-macro` is implemented) are supported.

(See the design limits section for current constraints of these features.)

Procedures/macros exist to explicitly compile a module, compile a module and its dependencies, force recompilation, remove the compiled version of a module, etc. .

Functionality exists to compile all modules addressed by an `(import)` command into one object file. Also, functionality exists to gather the compiled object files addressed by an `(import)`

command into one list, for you to link together, in order to create executable files for distribution to end-users. Please see the ”How to export binaries” section for this.

Installation

Installing the files

Look to it that you have GIT (http://www.git-scm.org) installed.

Download a local copy of BH using GIT. BH will be downloaded into the `modules` subdirectory of the directory that you execute git from.

BH must reside in the directory that Gambit relates to as the `~~/lib/modules/` directory. That is, the `modules` subdirectory of the `lib` subdirectory directory of your Gambit installation.

On non-Cygwin Windows, this is the directory into which you must actually download BH. Thus on windows, `cd` to the `lib` directory of your Gambit installation, and follow the download instructions below.

On Unix systems, you can have BH in any directory, and create a symlink from Gambit's `lib` directory to it (by ensuring that you are logged in as root (`su`), and running `ln -s [the absolute path to your BH modules directory] /usr/local/Gambit-C/current/lib/modules`).

Download BH using GIT by:

```
git clone http://mwaza.dyndns.org/apps/files/modules.git
```

When downloaded, go to BH's directory (`cd modules`), and compile it by `gsc build` . Compiling BH is not necessary, but it does provide significant speed improvements over running it in interpreted mode.

Installation into the Gambit environment

BH is loaded by evaluating `(load "~~/lib/modules/build")` .

While this can be done manually on each start of g`sc`, typically you want a helper `bsc` script that automatizes the task.

For Unix, a `bsc` script is bundled with BH. To install it, ensure that you are logged in as administrator (`su`), and execute `ln -s [the absolute path to your BH modules directory]/bsc /usr/local/bin/bsc` .

On non-Cygwin Windows, a `bsc.bat` script is bundled with BH. To install it, copy it from the modules directory, to the `bin` subdirectory of your Gambit directory.

Ensure that this directory is in the system PATH environment variable, as found in the ”System variables” panel, under ”Environment variables”, in the ”Advanced” tab of the ”System properties” option in the Control panel.

For your reference, all that the `bsc` script typically contains is:

```
#!/bin/sh
gsc -:dar -e '(load "~~/lib/modules/build")' $@ -
```

Setting it up

BH needs no configuration, however you may benefit of compiling the standard libraries. To do this, start `bsc` and evaluate

```
(modules-compile! (modules-in-dir "~~/lib/modules/std") #t)
```

(In a future version, we will probably advice you to compile all the bundled libraries into one single object file. However the set of bundled libraries is currently not so stable as to justify that.)

How to use for a project

Set up files

Chose a root directory for your project, and put your sourcecode files in it.

Set up modules

In the beginning of each module, describe the import dependencies by `(import)`.

The module references you make in `(import)` should be relative to the respective module file's own directory.

I.e., for each Scheme file, `(import a)` imports a.scm from the same directory as the Scheme file, `(import b/a)` imports `a.scm` of the subdirectory `b`, and `(import ../a)` imports `a.scm` from the file's parent directory.

Review the “Special forms for use in modules” chapter for options applicable to your modules.

In incremental work

Originate development out of the REPL as usual.

When you restart `bsc`, use Gambit's history feature (i.e. the arrow up key on the keyboard) in order not to need to re-type any `(import)`:s you made in the previous `bsc` instance.

Keep your use of global state low, both because it is a good design principle to adhere to, and because BH's implementation of the syntactic tower benefits from this.

Export binaries

The object files generated by `(module-compile!)` etc. require only their module dependencies to be already `(load)`:ed in order to function.

`(module-deps)` returns all of a module's dependencies, sorted in the order of descending depth of the dependency tree. Therefore, the return value of `(map module-path (module-deps module))` is usable as object file load order.

`(modules-compile-bunch)` compiles multiple modules into one single object file. For information and examples, see its documentation section.

The basic principle to make you understand how to export binaries is this:
Object files do not contain the module-information of the module of which it is a compiled version, and neither does it contain export declarations for macros it contains.
`(load)` of object file version of a module performs the same action as `(import)`:ing it would have done, presuming that dependencies have been loaded rightly.

How to migrate to BH

Create a directory structure, and follow the other guidance, as described in the "How to use for a project" section.

Replace your Scheme file loading scripts, and any usage of `(include)` and `(load)` by `(import)` :s.

Effectively, BH requires being the exclusive user of Gambit's namespace functionality.
Therefore, you must remove any namespace definitions (i.e. use of `(namespace)`), and any explicit addressing of namespaces (i.e. addressing symbols in particular namespaces with `namespace-name#symbol-name)` from your code.
(The use of namespace-name#symbol-name is perfectly OK to use in the REPL. The hook with you using this notion is that you have no guarantee that a given module will have the same namespace name on two different computers.)

Special forms for use in modules

(compile-options #!key options cc-options ld-options-prelude ld-options force-compile)

`(compile-options)` may be put into any module, and can take any of the following arguments:

`options:`, `cc-options:`, `ld-options-prelude:`, `ld-options:`

> The respective value will be appended to the respective argument of (compile-file) on module compilation.
>
> For more information, see `(compile-file)` in Gambit's documentation (link).

`force-compile:`

> Tells whether the module must be compiled or not. Must be `#t` for modules that use Gambit FFI functionality such as `(c-define)`. However, the FFI forms imply setting this option to `#t`, so in practice you quite never need to care of this option.

Example use:

```
(compile-options force-compile: #t)
```

(export)

Specifies what symbols of the module to export. If a module does not contain an `(export)` declaration, all of its symbols are exported.

`(export)` takes a list of arguments. Each argument should either be

- a symbol that should be exported, or
- a list, starting by `rename:` , and continuing by an a-list of symbols that should be renamed, and the names they should be renamed to.
- a list, starting by `re-export:` , that contains a list of the modules whose exports this module should proxy, in the same notation as used in `(import)`.

  A module does not need to be imported to be re-exported.

  Using `(import)`'s tools, you can chose to re-export only particular names, or rename symbols on the re-export.

Example module (`a.scm`):

```
(import (std srfi/13))
(export
  output
  a-struct-a
  a
  (rename:    (more-output more-output-renamed)
              (a-struct-b  a-struct-b-renamed))
  (re-export: (std srfi/13)))

(define-type a-struct a b c)
(define a (make-a-struct 1 2 3))
(define (output) (display "Output.\n"))
(define (more-output) (display "More output.\n"))
```

Example use of module:

```
> (import a)
> a
#<a-struct #2 a: 1 b: 2 c: 3>
> (a-struct-a a)
1
> (a-struct-b-renamed a)
2
> (output)
Output.
> (more-output-renamed)
More output.
> (string-upcase "Abc") ; (From SRFI 13)
"ABC"
> (make-a-struct 4 5 6)
*** ERROR IN (console)@16.2 -- Unbound variable: ~#make-a-struct
1> [ctrl+d]
> (a-struct-c a)
*** ERROR IN (console)@19.2 -- Unbound variable: ~#a-struct-c
```

```
1> [ctrl+d]
> (more-output)
*** ERROR IN (console)@13.2 -- Unbound variable: ~#more-output
1>
```

(module #!optional module-name)

For use from the REPL for switching the current namespace to the namespace of a module, or to the REPL's namespace.

`(module a-module)` switches the current namespace to the namespace of the module `a-module`.

`(module)` switches the current namespace to the REPL's namespace.

A typical use scenario would be to first `(import your-module)`, do something with it, and when you realize you need modifications to it, without wanting to update the sourcecode file + `(import your-module)` again.

Thus into the REPL you `(module your-module)`, paste the definitions that are modified, and then `(module)`.

Procedures and macros

(import)

Takes one or more arguments, each argument being either a module name, where the root is the current directory, or a list consisting of a module-resolver, and one or more module names within the module-resolver.

When `(import)` is made from a Scheme file, we define the current directory as the directory that the Scheme file resides in. When `(import)` is made from the REPL, the current directory is the current working directory.

Examples:

Import module a of the current module-resolver (typically = `a.scm` of the current directory):

```
(import a)
```

Import the srfi/13 module of the std module-resolver:

```
(import (std srfi/13))
```

Import modules `srfi/13`, `srfi/14` and `misc/uuid` of the `std` module-resolver, and module `a` of the current module-resolver (typically = `a.scm` of the current directory):

```
(import
  (std srfi/13 srfi/14 misc/uuid)
  a)
```

Import `/tmp/a.scm`:

```
(import /tmp/a.scm)
```

The imports' dependencies are imported also.

If compiled versions are found, these are loaded rather than the sourcecode versions. If a module has a compiled version that has an older last modified timestamp than its sourcecode version, it is automatically recompiled.

Upon `(import)`, any code in the top-level will be evaluated, and any global defines marked for export will become available to the caller.

(See the General advice section for why you want to keep any code in the top-level minimal.)

In the current version of BH, the sourcecode version of a module must always be apparent, for `(import)` to be made.

(This is because the exported macro definitions and the internal module-information record are derived from the sourcecode on import. This will not be the case anymore when BH implements the syntactic tower, in a future version, though.)

Filter, prefix, rename symbols on import

You can add certain options to module imports by enclosing the imports like this:

`prefix:` prefixes all imported symbols.

Prefix all srfi/1 symbols by `1-`:

```
(import (prefix: (std srfi/1) 1-))
```

`rename:` renames one or more symbols.

Rename srfi/95's `sort` to `srt` and `merge` to `mrg`:

```
(import (rename: (std srfi/95) (sort srt) (merge mrg)))
```

`only:` specifies that only mentioned symbols should be imported.

Import only `string-upcase` and `string-downcase` from srfi/13:

```
(import (only: (std srfi/13) string-upcase string-downcase))
```

`except:` specifies that all symbols except the mentioned should be imported.

Import all symbols except `string-upcase` and `string-downcase` from srfi/13:

```
(import (except: (std srfi/13) string-upcase string-downcase))
```

You can combine `prefix:`, `rename:` and `only:` or `except:`. Example:

```
(import
```

```
(prefix:
  (rename:
    (only: (std srfi/13) string-upcase string-downcase)
    (string-upcase string-up)
    (string-downcase string-down))
  13-)
(prefix:
  (except: (std srfi/95) merge)
  95-))
```

Behavior of subsequent (import):s of one and the same module

Subsequent `(import)`:s of a module are generally ignored. However, if a module's sourcecode has been modified, it will be re-imported. If the module has a compiled version, then it will be recompiled also. Example:

```
$ echo '(display "hi")(force-output)' > /tmp/a.scm; bsc
> (import /tmp/a)
hi> (import /tmp/a)
> [press ctrl+z]
[1]+  Stopped                 bsc
$ echo '(display " again")(force-output)' >> /tmp/a.scm; fg
bsc
(import /tmp/a)
hi again> (module-compile! '/tmp/a)
> (import /tmp/a)
> [press ctrl+z]
[1]+  Stopped                 bsc
$ echo '(display " again!")(force-output)' >> /tmp/a.scm; fg
bsc
(import /tmp/a)
/tmp/a is being compiled...
hi again again!>
```

Note that imported modules that depend on a module that is reimported, will not be reimported also, and thus any references the depending modules have to a reimported module, will be to its previously imported version, and not the new one.

Though, when the referring code is not evaluated with the strategy `(declare (block))`) (that is, when it is evaluated with the default strategy `(declare (separate))`), variable lookup is made anew on procedure invocations. I.e. given a.scm:

```
(print "Module A import\n")
(define exported-var 'something)
```

And b.scm:

```
(import a)

(define (work-when-separate) exported-var)
(begin (declare (block)) (define (work-when-block) exported-var))

(define work-closure1 (lambda () exported-var))
(define work-closure2 (let ((c exported-var)) (lambda () c)))

(define structure (cons exported-var 'something-tihrd))
```

And then altering a.scm into:

```
(print "Module A import\n")
(define exported-var 'something-else)
```

Will produce the behavior:

```
$ echo "(print \"Module A import\n\") (define exported-var 'something)" >
a.scm
$ echo "(import a) (define (work-when-separate) exported-var) (begin
(declare (block)) (define (work-when-block) exported-var)) (define work-
closure1 (lambda () exported-var)) (define work-closure2 (let ((c
exported-var)) (lambda () c))) (define structure (cons exported-var
'something-third))" > b.scm
$ bsc
> (import b)
Module A import
> (work-with-exported-var)
something
> (more-work)
something
> structure
something . something-third
> [press ctrl+z]
[1]+  Stopped                 bsc
$ echo "(print \"Module A import\n\") (define exported-var 'something-
else)" > a.scm; fg
(import b)
Module A import
> (work-with-exported-var)
something-else
> (more-work)
something
> structure
something . something-third
```

i.e. references from a closure or data structure in a depending module are to the old version, but references from procedures are to the new version.

HTTP URL:s as module names

Is supported. Downloaded files are stored in `~~/lib/modules/work/lib`.

Example:

```
(import "http://github.com/pereckerdal/termite/raw/master/termite.scm")
```

On subsequent `(import)`:s, no checking is done to ensure that you have the latest version on your local machine.

HTTP URL:s are also supported by other procedures:

```
(module-compile/deps!
"http://mwaza.dyndns.org/apps/files/termite/termite.scm")
```

Downloading is done using the tool `wget`. Thus it must be installed for it to work.

(module-compile! module-name #!optional continue-on-error)

Compiles one module. `module-name` may either be a module name, describe in the same way as in `(import)`, or a module structure.

If the module is already compiled, it is recompiled.

If `continue-on-error` is set to `#t`, compilation error will not throw an exception, but `#f` will be returned.

Examples:

Compile `a.scm`:

```
> (module-compile! 'a)
>
```

Compile `srfi/13.scm` of `std`:

```
> (module-compile! '(std srfi/13))
>
```

Use with a module structure for argument:

```
(module-compile! (car (modules-in-dir "/lib/modules/misc")))
```

(modules-compile-bunch to-file files #!key (save-links #t) port)

Compiles a list of Scheme files into one object file.

Example “raw” use:

```
$ echo '(define (p) (display "test\n"))' > a.scm
$ echo '(import a) (p)' > b.scm
$ bsc
> (module-compile-bunch "out.o1" '("a.scm" "b.scm") save-links: #f)
Compiling 2 files...
a.scm ...
b.scm ...
Creating link file..
Compiling link file..
Linking files..
"out.o1"
> ,q
$ gsc
Gambit (version)
> (load "out.o1")
test
>
```

Further example:

```
> (module-compile-bunch "~~/lib/blackhole/std.ob" (module-files-in-dir
```

```
"~~/lib/blackhole/std"))
Compiling 31 files...
~~/lib/blackhole/std/string/xml-to-sxml.scm ...
~~/lib/blackhole/std/string/util.scm ...
(snip)
~~/lib/blackhole/std/misc/al.scm ...
~~/lib/blackhole/std/ds/wt-tree.scm ...
~~/lib/blackhole/std/ds/queue.scm ...
Creating link file..
Compiling link file..
Linking files..
"~~/lib/blackhole/std.ob"
>
```

If `save-links` is `#t`, it saves an `.ol` (note L, not 1) file that contains a row, which is the filename of the object file, which can be used by `(import)`. So if you have a group of modules that do generally not change (for instance the bundled modules of the `std` module-resolver), you can do `(module-compile-bunch)` with `save-links #t`.

This compiles all modules to one binary file, and saves an .ol file per module, in the same directory as the module, so on import the .ol file is loaded, instead of the module's own sourcecode or binary object file. This gives a perfomance benefit when importing many modules.

The current version of this procedure takes filenames for `files` argument only. A future version will be able to take a module and all of its dependencies and make an object file of it automatically.

(This procedure works on Mac OS X, has been shown not to work on Debian, and has not been extensively tested. Please do not hesitate to pass reports or questions to the authors about this procedure – of course after having checked you have the latest version of BH.)

(module-deps module #!optional recursive)

Returns a list of module objects describing module and its dependencies. `module` should be a module name, described in the same way as in `(import)`.

```
(module-deps '(std srfi/13))
```

The returned list is sorted in the order of descending depth of the dependency tree.

(modules-in-dir top-dir)

Returns a list of module objects describing the modules in `top-dir`.

```
(modules-in-dir "~~/lib/modules/std")
```

(module-clean! module)

Removes the compiled version of the module. `module` may either be a module name, described in the same way as in `(import)`, or a module structure.

```
(module-clean! 'module)
(module-clean! '(std srfi/13))
(module-clean! (car (modules-in-dir "~~/lib/modules/std/misc")))
```

(modules-compile! modules #!optional continue-on-error port)

Recompiles `modules`. Differs from `(module-compile!)` in the important way that it does not force recompilation – if a compiled version of a module is up-to-date with its sourcecode form, it will not be compiled.

Compilation is made if no compiled version of the module exists, or if the compiled version is outdated. Combine with `(module-clean!)`/`(modules-clean!)`/`(module-clean/deps!)` to create forcing behavior.

```
(modules-compile! '((std srfi/1 srfi/14)))
(modules-compile! (modules-in-dir "~~/lib/modules/std"))
```

(modules-clean! modules)

`(for-each module-clean! modules).`

(module-compile/deps! module #!optional continue-on-error port)

Like `(modules-compile!)`, but compiles one module and its dependencies. `module` may either be a module name, described in the same way as in `(import)`, or a module structure.

```
(module-compile/deps! '(std net/tcpip))
(module-compile/deps! (car (modules-in-dir "~~/lib/modules/std/misc")))
```

(module-clean/deps! module #!optional continue-on-error port)

`(module-clean!)` for the module and each of its dependencies.

(expand-macro macro)

Returns the expanded form of the macro. The argument should be an s-expression describing the macro invocation, just like an argument to `(eval)`.

```
> (define-macro (do-times times code)
    `(begin
      ,@(let loop ((n times) (r '()))
             (if (zero? n) r (loop (- n 1) (cons code r))))
      #!void))
> (expand-macro '(do-times 2 (display "output ")))
(begin (display "output ") (display "output ") #!void)
> (do-times 2 (display "output "))
output output >
```

Valuable for debugging macros.

General advice

Have as little code execute in the top-level of modules as possible. Any such code is executed on `(import)`.

Any errors in code executed then would be easier to track if they were trigged on a procedure invocation, in particular if they happen deep in the dependency graph.

Also, generally you want `(import)` to execute as quickly as possible.

Internals

The sourcecode

BH's sourcecode is of moderate size, and well documented. Originate your reading in `build.scm`.

Why BH must be in ~~/lib/modules/

This choice provided for the most straightforward way to implement the `std` module-resolver. Otherwise than because of that, there's no need for BH to reside in any particular place.

What BH really does with Gambit's namespace functionality

Actually BH does not use Gambit's namespace functionality at all. It parses `gambit#.scm` with a special-written parser in order to extract the symbols Gambit defines, and that's it.

Development status, design limits

BH's core is stable.

For the development status of the bundled libraries, see their respective documentation.

There are certain constraints in the functionality of BH's core:

Currently `(let-syntax)` and `(letrec-syntax)` cannot contain `(define-syntax)`, due to incomplete handling of syntactic closures. Example:

```
(let-syntax
   ((macro-1 (syntax-rules () ((_) #t))))
 (define-syntax macro-2
    (syntax-rules ()
      ((_) (macro-1))))

 (macro-2)) ;; Correctly expands to #t

(macro-2) ;; This should expand to #t, but macro-1 will be forgotten at
          ;; this time, so this doesn't work
```

Currently C type definitions are not exported between modules.

Currently two modules cannot have the same name. The module name is given by the filename. (Minor additions required to support this.)

Note that BH always works with at most one instance of each module.

An instance where this could be relevant, and where BH differs from certain other Scheme environment (in this case at least PLT), is in this case: if you have a module `a` in which there is the global `b`, and you evaluate `(import a) b (syntax-begin (import a) b)`, then the two instances of use of `b` will address the same object instance. If not paid attention to, this could manifest as subtle problems.

Cyclic `(import)`:s are not detected, and cause an infinite loop.

License

BH and its bundled libraries (CURRENLTY WITH SOME EXCEPTIONS) is licensed under the MIT non-academic license:

> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For further references to the MIT non-academic license, including notes on its relation with the BSD license, please see http://en.wikipedia.org/wiki/MIT_license .

The license text is also found on http://opensource.org/licenses/mit-license.php .