FFI Helper User Guide
Matt Wette
October 2022
With NYACC Version 1.09.1

Introduction

The acronym FFI stands for "Foreign Function Interface". It refers to the Guile facility for binding functions and variables from C source libraries into Guile programs. This distribution provides utilities for generating a loadable Guile module from a set of C declarations and associated libraries. The C declarations can, and conventionally do, come from naming a set of C include files. The nominal method for use is to write a *ffi-module* specification in a file which includes a `define-ffi-module` declaration, and then use the command `guild compile-ffi` to produce an associated file of Guile Scheme code.

```
$ guild compile-ffi ffi/cairo.ffi
wrote ‘ffi/cairo.scm’
```

The FH does not generate C code. The hooks to access functions in the Cairo library are provided in 100% Guile Scheme via `(system foreign)`.

To generate Guile Scheme for smaller C code units one can write a ffi-module with the `#:api-code` or import the `ffi-help` module an use the functions `load-include-file`, `C-decl->scm` or `C-fun-decl->scm`. The latter functions are not well tested, though.

The compiler for the FFI Helper (FH) is based on the C parser and utilities which are included in the NYACC (`https://www.nongnu.org/nyacc`) package. Within the NYACC distribution, there are a number of example dot-ffi files in the directory `examples/ffi`.

Use of the FFI-helper module depends on the *scheme-bytestructure* package available from `https://github.com/TaylanUB/scheme-bytestructures`. Releases are available at `https://github.com/TaylanUB/scheme-bytestructures/releases`.

At runtime, after the FFI Helper has been used to create Scheme code, the modules `(system ffi-help-rt)` and `(bytestructures guile)` are required. No other code from the NYACC distribution is needed. However, note that the process of creating the Scheme output depends on reading system headers, so the generated code may well contain operating system and machine dependencies. If you copy code to a new machine, you should re-run `guild compile-ffi`.

You are probably hoping to see an example, so let's try one.

This is a small FH example to illustrate its use. We will start with the Cairo (`cairographics.org`) package because that is the first one I started with in developing the FFI Helper. Say you are an avid Guile user and want to be able to use Cairo in Guile. On most systems Cairo comes with the associated *pkg-config* support files; this demo depends on that support.

Warning: The FFI Helper package is under active development and there is some chance the following example will cease to work in the future.

If you want to follow along and are working in the distribution tree, you should source the file `env.sh` in the `examples` directory.

By practice, I like to put all FH generated modules under a directory called `ffi/`, so we will do that. We start by generating, in the `ffi` directory, a file named `cairo.ffi` with the following contents:

```
(define-ffi-module (ffi cairo)
  #:pkg-config "cairo"
  #:include '("cairo.h" "cairo-pdf.h" "cairo-svg.h"))
```

To generate a Guile module you execute `guild` as follows:

```
$ guild compile-ffi ffi/cairo.ffi
wrote `ffi/cairo.scm'
```

Though the file `cairo/cairo.ffi` is only three lines long, the file `ffi/cairo.scm` will be over five thousand lines long. It looks like the following:

```
(define-module (ffi cairo)
  #:use-module (system ffi-help-rt)
  #:use-module ((system foreign) #:prefix ffi:)
  #:use-module (bytestructures guile))
(define link-libs
  (list (dynamic-link "libcairo")))

;; int cairo_version(void);
(define ~cairo_version
  (delay (fh-link-proc ffi:int "cairo_version" (list) link-libs)))
(define (cairo_version)
  (let () ((force ~cairo_version))))
(export cairo_version)

...

;; typedef struct _cairo_matrix {
;;   double xx;
;;   double yx;
;;   double xy;
;;   double yy;
;;   double x0;
;;   double y0;
;; } cairo_matrix_t;
(define-public cairo_matrix_t-desc
  (bs:struct
    (list `(xx ,double) `(yx ,double) `(xy ,double)
          `(yy ,double) `(x0 ,double) `(y0 ,double))))
(define-fh-compound-type cairo_matrix_t cairo_matrix_t-desc
 cairo_matrix_t? make-cairo_matrix_t)
(export cairo_matrix_t cairo_matrix_t? make-cairo_matrix_t)

... many, many more declarations ...
```

```
;; access to enum symbols and #define'd constants:
(define ffi-cairo-symbol-val
  (let ((sym-tab
         '((CAIRO_SVG_VERSION_1_1 . 0)
           (CAIRO_SVG_VERSION_1_2 . 1)
           (CAIRO_PDF_VERSION_1_4 . 0)
           (CAIRO_PDF_VERSION_1_5 . 1)
           (CAIRO_REGION_OVERLAP_IN . 0)
           (CAIRO_REGION_OVERLAP_OUT . 1)
           ... more constants ...
           (CAIRO_MIME_TYPE_JBIG2_GLOBAL_ID
             .
             "application/x-cairo.jbig2-global-id"))))
    (lambda (k) (or (assq-ref sym-tab k)))))
(export ffi-cairo-symbol-val)
(export cairo-lookup)

... more ...
```

Note that from the *pkg-config* spec the FH compiler picks up the required libraries to bind in. Also, `#define` based constants, as well as those defined by enums, are provided in a lookup function `ffi-cairo-symbol-val`. So, for example

```
guile> (use-modules (ffi cairo))
;;; ffi/cairo.scm:6112:11: warning:
    possibly unbound variable ‘cairo_raster_source_acquire_func_t*’
;;; ffi/cairo.scm:6115:11: warning:
    possibly unbound variable ‘cairo_raster_source_release_func_t*’
guile> (ffi-cairo-symbol-val 'CAIRO_FORMAT_ARGB32))
$1 = 0
```

We will discuss the warnings later. They are signals that extra code needs to be added to the ffi module. But you see how the constants (but not CPP function macros) can be accessed.

Let's try something more useful: a real program. Create the following code in a file, say `cairo-demo.scm`, then fire up a Guile session and `load` the file.

```
(use-modules (ffi cairo))
(define srf (cairo_image_surface_create 'CAIRO_FORMAT_ARGB32 200 200))
(define cr (cairo_create srf))
(cairo_move_to cr 10.0 10.0)
(cairo_line_to cr 190.0 10.0)
(cairo_line_to cr 190.0 190.0)
(cairo_line_to cr 10.0 190.0)
(cairo_line_to cr 10.0 10.0)
(cairo_stroke cr)
(cairo_surface_write_to_png srf "cairo-demo.png")
(cairo_destroy cr)
```

```
(cairo_surface_destroy srf)
guile> (load "cairo-demo.scm")
...
;;; compiled /.../cairo.scm.go
;;; compiled /.../cairo-demo.scm.go
guile>
```

If we set up everything correctly we should have generared the target file `cairo-demo.png` which contains the image of a square. A few items in the above code are notable. First, the call to `cairo_image_surface_create` accepted a symbolic form `'CAIRO_FORMAT_ARGB32` for the format argument. It would have also accepted the associated constant `0`. In addition, procedures declared in `(ffi cairo)` will accept Scheme strings where the C function wants "pointer to string."

Now try this in your Guile session:

```
guile> srf
$4 = #<cairo_surface_t* 0x7fda53e01880>
guile> cr
$5 = #<cairo_t* 0x7fda54828800>
```

Note that the FH keeps track of the C types you use. This can be useful for debugging (at a potential cost of bloating the namespace). The constants you see are the pointer values. But it goes further. Let's generate a matrix type:

```
guile> (define m (make-cairo_matrix_t))
guile> m
$6 = #<cairo_matrix_t 0x10cc26c00>
guile> (use-modules (system ffh-help-rt))
guile> (pointer-to m)
$7 = #<cairo_matrix_t* 0x10cc26c00>
```

When it comes to C APIs that expect the user to allocate memory for a structure and pass the pointer address to the C function, FH provides a solution:

```
guile> (cairo_get_matrix cr (pointer-to m))
guile> (fh-object-ref m 'xx)
$9 = 1.0
```

But the FFI helper can also be used on a per declaration basis, but you must first import the proper modules and libraries. This functionality is still under development.

The following example shows how to convert to scheme code using the procedure `C-decl->scm`:

```
guile> (use-modules (nyacc lang c99 ffi-help))
guile> (define struct-foo-desc (C-decl->scm "struct foo { int x; double y; };"))
guile> ,pp struct-foo-desc
$1 = (bs:struct (list `(x ,int) `(y ,double)))
```

If we import more modules we can use the syntax `C-decl` to complete definitions:

```
guile> (use-modules (system ffi-help-rt))
guile> (use-modules (bytestructures guile))
guile> (use-modules ((system foreign) #:prefix ffi:))
```

```
guile> (define ffi-libs '()) ;; hack for now

guile> (define my-sqrt "double sqrt(double);")
guile> (my-sqrt 4.0)
```

Note that for functions like the above to work any dependent libraries must be loaded first, via `(dynamic-link)`.

Note: currently the above defines a bytestructure, but not a FH type. We could define a FH type as follows:

```
guile> (define-fh-compound-type struct-foo
        struct-foo-desc struct-foo? make-struct-foo)
```

The Guile Foreign Function Interface

Guile has an API, called the Foreign Function Interface, which allows one to avoid writing and compiling C wrapper code in order to access C coded libraries. The API is based on `libffi` and is covered in the Guile Reference Manual. We review some important bits here. For more insight you should read the relevant sections in the Guile Reference Manual. For more info on libffi internals visit libffi (`https://github.com/libffi/libffi`).

The relevant procedures used by the FH are

`dynamic-link`
: links libraries into Guile session

`dynamic-func`
: generated Scheme-level pointer to a C function

`pointer->procedure`
: geneates a Scheme lambda given C function signature

`dynamic-pointer`
: provides access to global C variables

Several of the above require import of the module `(system foreign)`.

In order to generate a Guile procedure wrapper for a function, say `int foo(char *str)`, in some foreign library, say `libbar.so`, you can use something like the following:

```
(use-modules (system foreign))
(define foo (pointer->procedure
             int
             (dynamic-func "foo" (dynamic-link "libbar"))
             (list '*)))
```

The argument `int` is a variable name for the return type, the next argument is an expression for the function pointer and the third argument is an expression for the function argument list. To execute the function, which expects a C string, you use something like

```
(define result-code (foo (string->pointer "hello")))
```

If you want to try a real example, this should work:

```
guile> (use-modules (system foreign))
guile> (define strlen
```

```
          (pointer->procedure
           int (dynamic-func "strlen" (dynamic-link)) (list '*)))
guile> (strlen (string->pointer "hello, world"))
$1 = 12
```

It is important to realize that internally Guile takes care of converting Scheme arguments to and from C types. Scheme does not have the same type system as C and the Guile FFI is somewhat forgiving here. When we declare a C function interface with, say, an uint32 argument type, in Scheme you can pass an exact numeric integer. The FH attempts to be even more forgiving, allowing one to pass symbols where C enums (i.e., integers) are expected.

As mentioned, access to libraries not compiled into Guile is accomplished via `dynamic-link`. To link the shared library `libfoo.so` into Guile one would write something like the following:

```
(define foo-lib (dynamic-link "libfoo"))
```

Note that Guile takes care of dealing with the file extension (e.g., `.so`). Where Guile looks for libraries is system dependent, but usually it will find shared objects in the following

- `(assq-ref %guile-build-info 'libdir)`
- `(assq-ref %guile-build-info 'extensiondir)`
- `/usr/lib` on GNU/Linux and macOS
- $DYLD_LIBRARY_PATH on GNU/Linux and macOS
- directories listed in /etc/ld.so.conf on GNU/Linux

When used with no argument `dynamic-link` returns a handle for objects already linked with Guile. The procedure `dynamic-link` returns a library handle for acquiring function and variable handles, or pointers, for objects (e.g., a pointer for a function) in the library. Theoretically, once a library has been dynamically linked into Guile, the expression `(dynamic-link)` (with no argument) should suffice to provide a handle to acquire object handles, but I have found this is not always the case. The FH will try all library handles defined by a ffi module to acquire object pointers.

The FFI Helper Design

In this section we hope to provide some insight into the FH works. The FH specification, via the dot-ffi file, determines the set of declarations which will be included in the target Guile module. If there is no declartion filter, then all the declarations from the specified set of include files are targeted. With the use of a declaration filter, this set can be reduced. By declaration we mean typedefs, aggregate definitions (i.e., structs and unions), function declarations, and external variables.

In the C language typedefs define type aliases, so there is no harm in expanding typedefs which appear outside the specification. For example, say the file `foo.h` includes a declaration for the typedef `foo_t` and the file `bar.h` includes a declaration for the typedef `bar_t`. Furthermore, suppose `foo_t` is a struct that references `bar_t`. Then the FH will preserve the typedef `foo_t` but expand `bar_t`. That is, if the declarations are

```
typedef int bar_t;   /* from bar.h */
```

```
typedef struct { bar_t x; double y; } foo_t; /* from foo.h */
```

then the FH will treat `foo_t` as if it had been declared as

```
typedef struct { int x; double y; } foo_t; /* from foo.h */
```

When it comes to handling C types in Scheme the FH tries to leave base types (i.e., numeric types) alone and uses its own type system, based on Guiles *structs* and associated *vtables*, for structs, unions, function types and pointer types. Enum types are handled specially as described below. The FH type system associates with each type a number of procedures. One of these is the printer procedure which provided the association of type with output seen in the demo above.

One of the challenges in automating C-Scheme type conversion is that C code uses a lot of pointers. So as the FH generates types for aggregates, it will automatically generate types for associated pointers. For example, in the case above with `foo_t` the FH will generate an aggregate type named `foo_t` and a pointer type named `foo_t*`. In addition the FH generates code to link these two together so that, given an object `f1` of type `foo_t`, the expression `(pointer-to f1)` will generate an object of type `foo_t*`. This makes the task of generating an object value in Scheme, and then passing the pointer to that value as an argument to a FFI-generated procedure, easy. The inverse operation `value-at` is also provided. Note that sometimes the C code needs to work with pointer pointer types. The FH does not produce double-pointers and in that case, the user must add code to the FH module defintion to support the required additional type (e.g., `foo_t**`).

In addition, the FH type system provide unwrap and wrap procedures used internal to ffi-generated modules for function calls. These convert FH types to and from objects of type expected by Guile's FFI interface. For example, the unwrap procedure associated with the FH pointer type `foo_t*` will convert an `foo_t*` object to a Guile `pointer`. Similarly, on return the wrap procedure are applied to convert to FH types. When the FH generates a type, for example `foo_t` it also generates an exported procedure `make-foo_t` that users can use to build an object of that type. The FH also generates a predicate `foo_t?` to determine if an object is of that type. The `(system ffi-help-rt)` module provides a procedure `fh-object-ref` to convert an object of type `foo_t` to the underlying bytestructures representation. For numeric and pointer types, this will generate a number and for aggregate types, a bytestructure. Additional arguments to `fh-object-ref` for aggregates work as with the bytestructures package and enable selection of components of the aggregate. Note that the underlying type for a bytestructure pointer is an integer.

Enums are handled specially. In C, enums are represented by integers. The FH does not generate types for C enums or C enum typedefs. Instead, the FH defines unwrap and wrap procedures to convert Scheme values to and from integers, where the Scheme values can be integers or symbols. For example, if, in C, the enum typedef `baz_t` has element `OPTION_A` with value 1, a procedure expecting an argument of type `baz_t` will accept the symbol `'OPTION_A` or the integer `1`.

Where the FH generates types, the underlying representation is a *bytestructure descriptor*. That is, the FH types are essentially a layer on top of a bytestructure. The layer provides identification seen at the Guile REPL, unwrap and wrap procedures which are used in function handling (not normally visible to the user) and procedures to convert types to and from pointier-types.

For base types (e.g., `int`, `double`) the FH uses the associated Scheme values or the associated bytestructures values. (I think this is all bytestructure values now.)

The underlying representation of bytestructure values is *bytevectors*. See the Guile Reference Manual for more information on this datatype.

The following routines are user-level procedures provided by the runtime module `(system ffi-help-rt)`:

`fh-type?` a predicate to indicate whether an object is a FH type

`fh-object?`
: a predicate to indicate whether an object is a FH object

`fh-object-val`
: the underlying bytestructure value

`fh-object-ref`
: a procedure that works like `bytestructure-ref` on the underlying object

`fh-object-set!`
: a procedure that works like `bytestructure-set!` on the underlying object

`pointer-to`
: a procedure, given a FH object, or a bytestructure, that returns an associated pointer object (i.e., a pointer type whose object value is the address of the underlying argument); this may be a FH type or a bytestructure

`value-at` a procedure to dereference an object

`fh-cast` a procedure to cast arguments for varaidic C functions

`make-`*`type`*
: make base type, as listed below; also used to make bytestructure objects for base types (e.g., `(make-double)` for `double`)

Supported base types are

short	unsigned-short	int	unsigned
long	unsigned-long	float	double
size_t	ssize_t	intptr_t	uintptr_t
ptrdiff_t			
int8	uint8	int16	uint16
int32	uint32	int64	uint64

These types are useful for cases where the corresponding types are passed by reference as return types. For example

```
(let ((name (make-char*)))
  (some_function (pointer-to name))
  (display "name: ") (display (char*->string name)) (newline))
(let ((return-val (make-double)))
  (another_function (pointer-to return-val))
  (simple-format #t "val is ~S\n" (fh-object-ref return-val)))
```

You can pass a bytestructure struct value:

```
guile> (make-ENTRY '((key 0) (data 0)))
```

```
#<ENTRY 0x18a10b0>
```

TODO: should we support `(make-ENTRY 0 0)` ?

Creating FFI Modules with `(nyacc lang c99 ffi-help)`

```
(define ffi-module module-name ...)
```

`#:pkg-config`
: This option take a single string argument which provides the name used for the *pkg-config* program. Try `man pkg-config`.

`#:include`
: This form, with expression argument, indicates the list of include files to be processed at the top level. Without use of the `#:inc-filter` form, only declarations in these files will be output. To constrain the set of declarations output use the `#:decl-filter` form.

`#:inc-filter`
: This form, with predicate procedure argument taking the form `(proc file-spec path-spec)`, is used to indicate which includes beyond the top-level should have processed declarations emitted in the output. The `file-spec` argument is a string as parsed from `#include` statements in the C code, including brackets or double quotes (e.g., `"<stdio.h>"`, `"\"foo.h\""`). The `path-spec` is the full path to the file.

`#:use-ffi-module`
: This form, with literal module-type argument (e.g., `(ffi glib)`), indicates dependency on declarations from another processed ffi module. For example, the ffi-module for `(ffi gobject)` includes the form `#:use-ffi-module (ffi glib)`.

`#:decl-filter`
: This form, with a predicate procedure argument, is used to restrict which declarations should be processed for output. The single argument is either a string or a pair. The string form is used for simple identifiers and the pair is used for struct, union and enum forms from the C code (e.g., `(struct . "foo")`).

`#:library`
: This form, with a list of strings, indicates which (shared object) libraries need to be loaded. The formmat of each string in the list should be as provided to the `dynamic-link` form in Guile.

`#:renamer`
: todo

`#:cpp-defs`
: This form, with a list of strings, provides extra C preprodessor definitions to be used in processing the header files. The defines take the form `"SYM=`*val*`"`.

`#:inc-dirs`
: This form, with a list of strings, provides extra directories in which to search for include files.

`#:inc-help`
: todo

`#:api-code`
: todo

`#:def-keepers`
: This form, with a list of strings, provides extra (non-function) C preprocessor macro definitions that should be included in the output.

```
#:library '("libcairo" "libmisc")
#:inc-dirs '("/opt/local/include/cairo" "/opt/local/include")
#:renamer (string-renamer
   (lambda (n)
     (if (string=? "cairo" (substring n 0 5)) n
 (string-append "cairo-" n))))
#:pkg-config "cairo"
#:include '("cairo.h" "cairo-svg.h")
#:inc-help (cond
            ((string-contains %host-type "darwin")
             '(("__builtin" "__builtin_va_list=void*")
               ("sys/cdefs.h" "__DARWIN_ALIAS(X)=")))
            (else '()))
#:decl-filter (string-member-proc
  "cairo_t" "cairo_status_t" "cairo_surface_t"
              "cairo_create" "cairo_svg_surface_create"
              "cairo_destroy" "cairo_surface_destroy")
#:export (make-cairo-unit-matrix)
```

Another decl-filter, useful for debugging.

```
#:decl-filter (lambda (k)
                (cond
                 ((member k '(
                              "cairo_t" "cairo_status_t"
                              "cairo_glyph_t" "cairo_path_data_t"
)) #t)
                 ((equal? k '(union . "union-_cairo_glyph_t")) #t)
                 (else #f)))
```

Direct Usage

Work to go here:

`load-include-file` *filename* [*#pkg-config pkg*] [Procedure]
: This is the functionality that Ludo was asking for: to be at guile prompt and be able to issue

```
(use-modules (nyacc lang c99 ffi-help))
(load-include-file "cairo.h" #:pkg-config "cairo")
```

```
guile> ,use (nyacc lang c99 ffi-help)
guile> (load-include-file "cairo.h" #:pkg-config "cairo")
;; wait a while
guile> ...
```

Tuning and Debugging

Since this is not all straightforward you will get errors.

Method

1. compile-ffi with flag to echo declarations
2. compile -O0 the resulting scm file
3. guile -c '(use-modules (ffi mymod))'

MAX_HEAP_SECTS

The message is

> Too many heap sections: Increase MAXHINCR or MAX_HEAP_SECTS

The message comes from the garbage collector. It means you've run out of memory. I found that this actually came from a bug in the ff-compiler which generated this code:

```
(bs:struct
  (list ...
        `(compose_buffer ,(bs:vector #f unsigned-int))
```

The original C declaration was

```
struct _GtkIMContextSimple {
  ...
  guint compose_buffer[7 + 1];
  ...
};
```

This bug, failure to evaluate `7+1` to an integer, was fixed.

Trimming Things Down

After using the FFI Helper to provide code for some packages you may notice that the quantity of code produced is large. For example, to generate a guile interface for gtk2+, along with glib, gobject, pango and gdk you will end up with over 100k lines of scm code. This may seem bulky. Instead it may be preferable to generate a small number of calls for gtk and work from there. In order to achieve this you could use the `#:api-code` or `#:decl-filter` options.

For example, in the expansion of the GLU/GL FFI module, called `glugl.ffi`, I found that a very large number of declarations starting with `PF` were being generated. I removed these using the `#:decl-filter` option:

```
(define-ffi-module (ffi glugl)
  #:include '("GL/gl.h" "GL/glu.h")
  #:library '("libGLU" "libGL")
  #:inc-filter (lambda (spec path) (string-contains path "GL/" 0))
```

```
  #:decl-filter (lambda (n) (not (and (string? n) (string-prefix? "PF" n)))))
```

Using the option reduced `glugl.scm` from 59,274 lines down to 15,354 lines.

As another example, if we wanted to just generate code for the gtk hello world demo we could write

```
(define-ffi-module (hack1)
  #:pkg-config "gtk+-2.0"
  #:api-code "
  #include <gtk2.h>
  void gtk_init(int *argc, char ***argv);
  void gtk_container_set_border_width(GtkContainer *container,
       guint border_width);
  void gtk_container_add(GtkContainer *container, GtkWidget *widget);
  void gtk_widget_show(GtkWidget *widget);
  void gtk_main(void);
  ")
```

Since the above example does not ask the FH to pull in typedef's then the pointer types will be expanded to native. You could invent your own types or echo the typedefs from the package headers

Warning: Possibly Unbound Variable

```
;;; ffi/gtk2+.scm:3564:5: warning:
    possibly unbound variable ‘GtkEnumValue*’
;;; ffi/gtk2+.scm:3581:5: warning:
    possibly unbound variable ‘GtkFlagValue*’
;;; ffi/gtk2+.scm:10717:11: warning:
    possibly unbound variable ‘GtkAllocation*’
;;; ffi/gtk2+.scm:15107:15: warning:
    possibly unbound variable ‘GdkNativeWindow’
;;; ffi/gtk2+.scm:15122:15: warning:
    possibly unbound variable ‘GdkNativeWindow’
;;; ffi/gtk2+.scm:26522:11: warning:
    possibly unbound variable ‘GSignalCMarshaller’
;;; ffi/gtk2+.scm:62440:11: warning:
    possibly unbound variable ‘GdkNativeWindow’
;;; ffi/gtk2+.scm:62453:5: warning:
    possibly unbound variable ‘GdkNativeWindow’
```

When I see this I check the scm file and see one of many things

`(fht-unwrap GtkAllocation*)`
: This usually means that `GtkAllocation` was somehow defined but not the pointer type.

Other

User is responsible for calling string->pointer and pointer->string.

By definition: wrap is c->scm; unwrap is scm->c.

`define-ffi-module` options:

`#:decl-filter proc`
: proc is a prodicate taking a key of the form `"name"`, `(struct . "name")`, `(union . "name")` or `(enum . "name")`.

`#:inc-filter proc`
`#:include expr`
: expr is string or list or procecure that evaluates to string or list

`#:library expr`
: expr is string or list or procecure that evaluates to string or list

`#:pkg-config string`
`#:renamer proc`
: procdure

Here are the type of hacks I need to parse inside `/usr/include` with NYACC's C99 parser. There is no such thing as a working C standard.

```
(define cpp-defs
  (cond
   ((string-contains %host-type "darwin")
    '("__GNUC__=6")
    (remove (lambda (s)
      (string-contains s "_ENVIRONMENT_MAC_OS_X_VERSION"))
    (get-gcc-cpp-defs)))
   (else '())))
(define fh-inc-dirs
  (append
   `(,(assq-ref %guile-build-info 'includedir) "/usr/include")
   (get-gcc-inc-dirs)))
(define fh-inc-help
  (cond
   ((string-contains %host-type "darwin")
    '(("__builtin"
       "__builtin_va_list=void*"
       "__attribute__(X)="
       "__inline=" "__inline__="
       "__asm(X)=" "__asm__(X)="
       "__has_include(X)=__has_include__(X)"
       "__extension__="
       "__signed=signed"
       )))
   (else
    '(("__builtin"
       "__builtin_va_list=void*" "__attribute__(X)="
       "__inline=" "__inline__="
       "__asm(X)=" "__asm__(X)="
       "__has_include(X)=__has_include__(X)"
       "__extension__="
```

```
)))))
```

The Run-time Module (system ffi-help-rt)

Here we provide details of the run-time support module.

Work to Go

02 if need foo_t pointer then I gen wrapper for foo_t* but add foo_t to *wrappers* so if I later run into need for foo_t may be prob

03 allow user to specify #:renamer (lambda (n) "make_goo" => "make-goo")

04 Now the hard part if we want to reference other ffi-modules for types or other c-routines. Say ffi-module foo defines foo_t now in ffi-module bar we want to reference, but redefine, foo_t

```
(define-ffi-module (cairo cairo) ...)
(define-ffi-module (cairo cairo-svg) #:use-ffi-module (cairo cairo)
```

05 Should setters for `bs:struct` enum fields check for symbolic arg?

06 Use guardians for `cairo_destroy` and `cairo_surface_destroy`?

07 What about vectors? If `foo(foo_t x[]`,

1. user must make vector of foo_t
2. ffi-module author should generate a make-foo_t-vector procedure

Completed

01

```
enum-wrap 0 => 'CAIRO_STATUS_SUCCESS
enum-unwrap 'CAIRO_STATUS_SUCCESS => 0
```

Administrative Items

Installation

```
./configure --prefix=xxx
make install
```

Reporting Bugs

Please report bugs by navigating with your browser to '`https://savannah.nongnu.org/projects/nyacc`' and select the "Submit New" item under the "Bugs" menu. Alternatively, ask on the Guile user's mailing list `guile-user@gnu.org`.

Notes

1. The following situation is a bit tricky for me.

```
typedef struct foo foo_t;
typedef foo_t bar_t;
```

```
struct foo { int a; };
int baz(foo_t *x);
```

Right now, on the first declaration I assign `foo_t` the type `fh-void`. The second declaration is handled as a type-alias. When I get to the third declaration I define the `struct foo` compound type, then re-define the `foo_t` as a compound type, and it's pointer type (missed this first time).

Copyright