CFFI User Manual

Table of Contents

1 Introduction

CFFI is the Common Foreign Function Interface for ANSI Common Lisp systems. By *foreign function* we mean a function written in another programming language and having different data and calling conventions than Common Lisp, namely, C. CFFI allows you to call foreign functions and access foreign variables, all without leaving the Lisp image.

We consider this manual ever a work in progress. If you have difficulty with anything CFFI-specific presented in the manual, please contact the developers (`cffi-devel@common-lisp.net`) with details.

Motivation

See Section 4.1 [What makes Lisp different], page 4, for an argument in favor of FFI in general.

CFFI's primary role in any image is to mediate between Lisp developers and the widely varying FFIs present in the various Lisp implementations it supports. With CFFI, you can define foreign function interfaces while still maintaining portability between implementations. It is not the first Common Lisp package with this objective; however, it is meant to be a more malleable framework than similar packages.

Design Philosophy

- Pointers do not carry around type information. Instead, type information is supplied when pointers are dereferenced.
- A type safe pointer interface can be developed on top of an untyped one. It is difficult to do the opposite.
- Functions are better than macros. When a macro could be used for performance, use a compiler-macro instead.

2 Installation

CFFI can be obtained through one of the following means available through its website:

- official release tarballs
- git repository

In addition, you will need to obtain and install the following dependencies:

- Babel, a charset encoding/decoding library.
- Alexandria, a collection of portable public-domain utilities.
- trivial-features, a portability layer that ensures consistent `*features*` across multiple Common Lisp implementations.

Furthermore, if you wish to run the testsuite, RT is required.

You may find mechanisms such as Quicklisp (`https://www.quicklisp.org/beta/`) (recommended) or clbuild (for advanced uses) helpful in getting and managing CFFI and its dependencies.

3 Implementation Support

CFFI supports various free and commercial Lisp implementations: ABCL, Allegro CL, Clasp, CLISP, Clozure CL, CMUCL, Corman CL, ECL, GCL, LispWorks, MCL, SBCL and the Scieneer CL.

In general, you should work with the latest versions of each implementation since those will usually be tested against recent versions of CFFI more often and might include necessary features or bug fixes. Reasonable patches for compatibility with earlier versions are welcome nevertheless.

3.1 Limitations

Some features are not supported in all implementations.

Allegro CL

- Does not support the `:long-long` type natively.
- Unicode support is limited to the Basic Multilingual Plane (16-bit code points).

Clasp

- Only supports a flat namespace.

CMUCL

- No Unicode support. (8-bit code points)

Corman CL

- Does not support `foreign-funcall`.

ECL

- On platforms where ECL's dynamic FFI is not supported (ie. when `:dffi` is not present in `*features*`), `cffi:load-foreign-library` does not work and you must use ECL's own `ffi:load-foreign-library` with a constant string argument.

Lispworks

- Does not completely support the `:long-long` type natively in 32-bit platforms.
- Unicode support is limited to the Basic Multilingual Plane (16-bit code points).

SBCL

- Not all platforms support callbacks.

4 An Introduction to Foreign Interfaces and CFFI

Users of many popular languages bearing semantic similarity to Lisp, such as Perl and Python, are accustomed to having access to popular C libraries, such as GTK, by way of "bindings". In Lisp, we do something similar, but take a fundamentally different approach. This tutorial first explains this difference, then explains how you can use CFFI, a powerful system for calling out to C and C++ and access C data from many Common Lisp implementations.

The concept can be generalized to other languages; at the time of writing, only CFFI's C support is fairly complete. Therefore, we will interchangeably refer to *foreign functions* and *foreign data*, and "C functions" and "C data". At no time will the word "foreign" carry its usual, non-programming meaning.

This tutorial expects you to have a working understanding of both Common Lisp and C, including the Common Lisp macro system.

4.1 What makes Lisp different

The following sums up how bindings to foreign libraries are usually implemented in other languages, then in Common Lisp:

Perl, Python, Java, other one-implementation languages
: Bindings are implemented as shared objects written in C. In some cases, the C code is generated by a tool, such as SWIG, but the result is the same: a new C library that manually translates between the language implementation's objects, such as `PyObject` in Python, and whatever C object is called for, often using C functions provided by the implementation. It also translates between the calling conventions of the language and C.

Common Lisp
: Bindings are written in Lisp. They can be created at-will by Lisp programs. Lisp programmers can write new bindings and add them to the image, using a listener such as SLIME, as easily as with regular Lisp definitions. The only foreign library to load is the one being wrapped—the one with the pure C interface; no C or other non-Lisp compilation is required.

We believe the advantages of the Common Lisp approach far outweigh any disadvantages. Incremental development with a listener can be as productive for C binding development as it is with other Lisp development. Keeping it "in the [Lisp] family", as it were, makes it much easier for you and other Lisp programmers to load and use the bindings. Common Lisp implementations such as CMUCL, freed from having to provide a C interface to their own objects, are thus freed to be implemented in another language (as CMUCL is) while still allowing programmers to call foreign functions.

Perhaps the greatest advantage is that using an FFI doesn't obligate you to become a professional binding developer. Writers of bindings for other languages usually end up maintaining or failing to maintain complete bindings to the foreign library. Using an FFI, however, means if you only need one or two functions, you can write bindings for only those functions, and be assured that you can just as easily add to the bindings if need be.

The removal of the C compiler, or C interpretation of any kind, creates the main disadvantage: some of C's "abstractions" are not available, violating information encapsulation. For example, `struct`s that must be passed on the stack, or used as return values, without corresponding functional abstractions to create and manage the `struct`s, must be declared explicitly in Lisp. This is fine for structs whose contents are "public", but is not so pleasant when a struct is supposed to be "opaque" by convention, even though it is not so defined.[1]

Without an abstraction to create the struct, Lisp needs to be able to lay out the struct in memory, so must know its internal details.

In these cases, you can create a minimal C library to provide the missing abstractions, without destroying all the advantages of the Common Lisp approach discussed above. In the case of `struct`s, you can write simple, pure C functions that tell you how many bytes a struct requires or allocate new structs, read and write fields of the struct, or whatever operations are supposed to be public.[2] Chapter 13 [The Groveller], page 119, automates this and other processes.

Another disadvantage appears when you would rather use the foreign language than Lisp. However, someone who prefers C to Lisp is not a likely candidate for developing a Lisp interface to a C library.

4.2 Getting a URL

The widely available `libcurl` is a library for downloading files over protocols like HTTP. We will use `libcurl` with CFFI to download a web page.

Please note that there are many other ways to download files from the web, not least the CL-CURL project to provide bindings to `libcurl` via a similar FFI.[3]

libcurl-tutorial(3) is a tutorial for `libcurl` programming in C. We will follow that to develop a binding to download a file. We will also use `curl.h`, `easy.h`, and the `man` pages for the `libcurl` function, all available in the '`curl-dev`' package or equivalent for your system, or in the cURL source code package. If you have the development package, the headers should be installed in `/usr/include/curl/`, and the `man` pages may be accessed through your favorite `man` facility.

4.3 Loading foreign libraries

First of all, we will create a package to work in. You can save these forms in a file, or just send them to the listener as they are. If creating bindings for an ASDF package of yours, you will want to add `:cffi` to the `:depends-on` list in your `.asd` file. Otherwise, just use the `asdf:load-system` function to load CFFI.

```
(asdf:load-system :cffi)
```

[1] Admittedly, this is an advanced issue, and we encourage you to leave this text until you are more familiar with how CFFI works.

[2] This does not apply to structs whose contents are intended to be part of the public library interface. In those cases, a pure Lisp struct definition is always preferred. In fact, many prefer to stay in Lisp and break the encapsulation anyway, placing the burden of correct library interface definition on the library.

[3] Specifically, UFFI, an older FFI that takes a somewhat different approach compared to CFFI. I believe that these days (December 2005) CFFI is more portable and actively developed, though not as mature yet. Consensus in the free UNIX Common Lisp community seems to be that CFFI is preferred for new development, though UFFI will likely go on for quite some time as many projects already use it. CFFI includes the `UFFI-COMPAT` package for complete compatibility with UFFI.

```
;;; Nothing special about the "CFFI-USER" package.  We're just
;;; using it as a substitute for your own CL package.
(defpackage :cffi-user
  (:use :common-lisp :cffi))

(in-package :cffi-user)

(define-foreign-library libcurl
  (:darwin (:or "libcurl.3.dylib" "libcurl.dylib"))
  (:unix (:or "libcurl.so.3" "libcurl.so"))
  (t (:default "libcurl")))

(use-foreign-library libcurl)
```

Using `define-foreign-library` and `use-foreign-library`, we have loaded `libcurl` into Lisp, much as the linker does when you start a C program, or `common-lisp:load` does with a Lisp source file or FASL file. We special-cased for UNIX machines to always load a particular version, the one this tutorial was tested with; for those who don't care, the `define-foreign-library` clause `(t (:default "libcurl"))` should be satisfactory, and will adapt to various operating systems.

4.4 Initializing `libcurl`

After the introductory matter, the tutorial goes on to present the first function you should use.

```
CURLcode curl_global_init(long flags);
```

Let's pick this apart into appropriate Lisp code:

```
;;; A CURLcode is the universal error code.  curl/curl.h says
;;; no return code will ever be removed, and new ones will be
;;; added to the end.
(defctype curl-code :int)

;;; Initialize libcurl with FLAGS.
(defcfun "curl_global_init" curl-code
  (flags :long))
```

> **Implementor's note:** *By default, CFFI assumes the UNIX viewpoint that there is one C symbol namespace, containing all symbols in all loaded objects. This is not so on Windows and Darwin, but we emulate UNIX's behaviour there.* *[defcfun], page 92, for more details.*

Note the parallels with the original C declaration. We've defined `curl-code` as a wrapping type for `:int`; right now, it only marks it as special, but later we will do something more interesting with it. The point is that we don't have to do it yet.

Looking at `curl.h`, `CURL_GLOBAL_NOTHING`, a possible value for `flags` above, is defined as '`0`'. So we can now call the function:

```
CFFI-USER> (curl-global-init 0)
```

```
⇒ 0
```

Looking at `curl.h` again, `0` means `CURLE_OK`, so it looks like the call succeeded. Note that CFFI converted the function name to a Lisp-friendly name. You can specify your own name if you want; use `("curl_global_init" your-name-here)` as the *name* argument to `defcfun`.

The tutorial goes on to have us allocate a handle. For good measure, we should also include the deallocator. Let's look at these functions:

```
CURL *curl_easy_init( );
void curl_easy_cleanup(CURL *handle);
```

Advanced users may want to define special pointer types; we will explore this possibility later. For now, just treat every pointer as the same:

```
(defcfun "curl_easy_init" :pointer)

(defcfun "curl_easy_cleanup" :void
  (easy-handle :pointer))
```

Now we can continue with the tutorial:

```
CFFI-USER> (defparameter *easy-handle* (curl-easy-init))
⇒ *EASY-HANDLE*
CFFI-USER> *easy-handle*
⇒ #<FOREIGN-ADDRESS #x09844EE0>
```

Note the print representation of a pointer. It changes depending on what Lisp you are using, but that doesn't make any difference to CFFI.

4.5 Setting download options

The `libcurl` tutorial says we'll want to set many options before performing any download actions. This is done through `curl_easy_setopt`:

```
CURLcode curl_easy_setopt(CURL *curl, CURLoption option, ...);
```

We've introduced a new twist: variable arguments. There is no obvious translation to the `defcfun` form, particularly as there are four possible argument types. Because of the way C works, we could define four wrappers around `curl_easy_setopt`, one for each type; in this case, however, we'll use the general-purpose macro `foreign-funcall` to call this function.

To make things easier on ourselves, we'll create an enumeration of the kinds of options we want to set. The `enum CURLoption` isn't the most straightforward, but reading the `CINIT` C macro definition should be enlightening.

```
(defmacro define-curl-options (name type-offsets &rest enum-args)
  "As with CFFI:DEFCENUM, except each of ENUM-ARGS is as follows:

    (NAME TYPE NUMBER)

Where the arguments are as they are with the CINIT macro defined
in curl.h, except NAME is a keyword.
```

```
TYPE-OFFSETS is a plist of TYPEs to their integer offsets, as
defined by the CURLOPTTYPE_LONG et al constants in curl.h."
  (flet ((enumerated-value (type offset)
           (+ (getf type-offsets type) offset)))
    `(progn
       (defcenum ,name
         ,@(loop for (name type number) in enum-args
              collect (list name (enumerated-value type number))))
       ',name)))                ;for REPL users' sanity

(define-curl-options curl-option
    (long 0 objectpoint 10000 functionpoint 20000 off-t 30000)
  (:noprogress long 43)
  (:nosignal long 99)
  (:errorbuffer objectpoint 10)
  (:url objectpoint 2))
```

With some well-placed Emacs `query-replace-regexps`, you could probably similarly define the entire `CURLoption` enumeration. I have selected to transcribe a few that we will use in this tutorial.

If you're having trouble following the macrology, just macroexpand the `curl-option` definition, or see the following macroexpansion, conveniently downcased and reformatted:

```
(progn
  (defcenum curl-option
    (:noprogress 43)
    (:nosignal 99)
    (:errorbuffer 10010)
    (:url 10002))
  'curl-option)
```

That seems more than reasonable. You may notice that we only use the *type* to compute the real enumeration offset; we will also need the type information later.

First, however, let's make sure a simple call to the foreign function works:

```
CFFI-USER> (foreign-funcall "curl_easy_setopt"
               :pointer *easy-handle*
               curl-option :nosignal :long 1 curl-code)
⇒ 0
```

`foreign-funcall`, despite its surface simplicity, can be used to call any C function. Its first argument is a string, naming the function to be called. Next, for each argument, we pass the name of the C type, which is the same as in `defcfun`, followed by a Lisp object representing the data to be passed as the argument. The final argument is the return type, for which we use the `curl-code` type defined earlier.

`defcfun` just puts a convenient façade on `foreign-funcall`.[4] Our earlier call to `curl-global-init` could have been written as follows:

```
CFFI-USER> (foreign-funcall "curl_global_init" :long 0
```

[4] This isn't entirely true; some Lisps don't support `foreign-funcall`, so `defcfun` is implemented without it. `defcfun` may also perform optimizations that `foreign-funcall` cannot.

```
                                 curl-code)
  ⇒ 0
```

Before we continue, we will take a look at what CFFI can and can't do, and why this is so.

4.6 Breaking the abstraction

In Section 4.1 [What makes Lisp different], page 4, we mentioned that writing an FFI sometimes requires depending on information not provided as part of the interface. The easy option `CURLOPT_WRITEDATA`, which we will not provide as part of the Lisp interface, illustrates this issue.

Strictly speaking, the `curl-option` enumeration is not necessary; we could have used `:int 99` instead of `curl-option :nosignal` in our call to `curl_easy_setopt` above. We defined it anyway, in part to hide the fact that we are breaking the abstraction that the C `enum` provides. If the cURL developers decide to change those numbers later, we must change the Lisp enumeration, because enumeration values are not provided in the compiled C library, `libcurl.so.3`.

CFFI works because the most useful things in C libraries — non-static functions and non-static variables — are included accessibly in `libcurl.so.3`. A C compiler that violated this would be considered a worthless compiler.

The other thing `define-curl-options` does is give the "type" of the third argument passed to `curl_easy_setopt`. Using this information, we can tell that the `:nosignal` option should accept a long integer argument. We can implicitly assume `t` ≡ 1 and `nil` ≡ 0, as it is in C, which takes care of the fact that `CURLOPT_NOSIGNAL` is really asking for a boolean.

The "type" of `CURLOPT_WRITEDATA` is `objectpoint`. However, it is really looking for a `FILE*`. `CURLOPT_ERRORBUFFER` is looking for a `char*`, so there is no obvious CFFI type but `:pointer`.

The first thing to note is that nowhere in the C interface includes this information; it can only be found in the manual. We could disjoin these clearly different types ourselves, by splitting `objectpoint` into `filepoint` and `charpoint`, but we are still breaking the abstraction, because we have to augment the entire enumeration form with this additional information.[5]

The second is that the `CURLOPT_WRITEDATA` argument is completely incompatible with the desired Lisp data, a stream.[6] It is probably acceptable if we are controlling every file we might want to use as this argument, in which case we can just call the foreign function `fopen`. Regardless, though, we can't write to arbitrary streams, which is exactly what we want to do for this application.

Finally, note that the `curl_easy_setopt` interface itself is a hack, intended to work around some of the drawbacks of C. The definition of `Curl_setopt`, while long, is far less

[5] Another possibility is to allow the caller to specify the desired C type of the third argument. This is essentially what happens in a call to the function written in C.

[6] See Section "Other Kinds of Streams" in *GNU C Library Reference*, for a GNU-only way to extend the `FILE*` type. You could use this to convert Lisp streams to the needed C data. This would be quite involved and far outside the scope of this tutorial.

cluttered than the equivalent disjoint-function set would be; in addition, setting a new option in an old `libcurl` can generate a run-time error rather than breaking the compile. Lisp can just as concisely generate functions as compare values, and the "undefined function" error is just as useful as any explicit error we could define here might be.

4.7 Option functions in Lisp

We could use `foreign-funcall` directly every time we wanted to call `curl_easy_setopt`. However, we can encapsulate some of the necessary information with the following.

```
;;; We will use this type later in a more creative way.  For
;;; now, just consider it a marker that this isn't just any
;;; pointer.
(defctype easy-handle :pointer)

(defmacro curl-easy-setopt (easy-handle enumerated-name
                            value-type new-value)
  "Call `curl_easy_setopt' on EASY-HANDLE, using ENUMERATED-NAME
as the OPTION.  VALUE-TYPE is the CFFI foreign type of the third
argument, and NEW-VALUE is the Lisp data to be translated to the
third argument.  VALUE-TYPE is not evaluated."
  `(foreign-funcall "curl_easy_setopt" easy-handle ,easy-handle
                    curl-option ,enumerated-name
                    ,value-type ,new-value curl-code))
```

Now we define a function for each kind of argument that encodes the correct `value-type` in the above. This can be done reasonably in the `define-curl-options` macroexpansion; after all, that is where the different options are listed!

We could make `cl:defun` forms in the expansion that simply call `curl-easy-setopt`; however, it is probably easier and clearer to use `defcfun`. `define-curl-options` was becoming unwieldy, so I defined some helpers in this new definition.

```
(defun curry-curl-option-setter (function-name option-keyword)
  "Wrap the function named by FUNCTION-NAME with a version that
curries the second argument as OPTION-KEYWORD.

This function is intended for use in DEFINE-CURL-OPTION-SETTER."
  (setf (symbol-function function-name)
        (let ((c-function (symbol-function function-name)))
          (lambda (easy-handle new-value)
            (funcall c-function easy-handle option-keyword
                     new-value)))))

(defmacro define-curl-option-setter (name option-type
                                     option-value foreign-type)
  "Define (with DEFCFUN) a function NAME that calls
curl_easy_setopt.  OPTION-TYPE and OPTION-VALUE are the CFFI
foreign type and value to be passed as the second argument to
easy_setopt, and FOREIGN-TYPE is the CFFI foreign type to be used
for the resultant function's third argument.

This macro is intended for use in DEFINE-CURL-OPTIONS."
  `(progn
     (defcfun ("curl_easy_setopt" ,name) curl-code
```

```
       (easy-handle easy-handle)
       (option ,option-type)
       (new-value ,foreign-type))
     (curry-curl-option-setter ',name ',option-value)))

(defmacro define-curl-options (type-name type-offsets &rest enum-args)
  "As with CFFI:DEFCENUM, except each of ENUM-ARGS is as follows:

    (NAME TYPE NUMBER)

Where the arguments are as they are with the CINIT macro defined
in curl.h, except NAME is a keyword.

TYPE-OFFSETS is a plist of TYPEs to their integer offsets, as
defined by the CURLOPTTYPE_LONG et al constants in curl.h.

Also, define functions for each option named
set-`TYPE-NAME'-`OPTION-NAME', where OPTION-NAME is the NAME from
the above destructuring."
  (flet ((enumerated-value (type offset)
           (+ (getf type-offsets type) offset))
         ;; map PROCEDURE, destructuring each of ENUM-ARGS
         (map-enum-args (procedure)
           (mapcar (lambda (arg) (apply procedure arg)) enum-args))
         ;; build a name like SET-CURL-OPTION-NOSIGNAL
         (make-setter-name (option-name)
           (intern (concatenate
                    'string "SET-" (symbol-name type-name)
                    "-" (symbol-name option-name)))))
    `(progn
       (defcenum ,type-name
         ,@(map-enum-args
            (lambda (name type number)
              (list name (enumerated-value type number)))))
       ,@(map-enum-args
          (lambda (name type number)
            (declare (ignore number))
            `(define-curl-option-setter ,(make-setter-name name)
               ,type-name ,name ,(ecase type
                                   (long :long)
                                   (objectpoint :pointer)
                                   (functionpoint :pointer)
                                   (off-t :long)))))
       ',type-name)))
```

Macroexpanding our `define-curl-options` form once more, we see something different:

```
(progn
  (defcenum curl-option
    (:noprogress 43)
    (:nosignal 99)
    (:errorbuffer 10010)
    (:url 10002))
  (define-curl-option-setter set-curl-option-noprogress
    curl-option :noprogress :long)
  (define-curl-option-setter set-curl-option-nosignal
    curl-option :nosignal :long)
```

```
  (define-curl-option-setter set-curl-option-errorbuffer
    curl-option :errorbuffer :pointer)
  (define-curl-option-setter set-curl-option-url
    curl-option :url :pointer)
  'curl-option)
```

Macroexpanding one of the new `define-curl-option-setter` forms yields the following:

```
(progn
  (defcfun ("curl_easy_setopt" set-curl-option-nosignal) curl-code
    (easy-handle easy-handle)
    (option curl-option)
    (new-value :long))
  (curry-curl-option-setter 'set-curl-option-nosignal ':nosignal))
```

Finally, let's try this out:

```
CFFI-USER> (set-curl-option-nosignal *easy-handle* 1)
⇒ 0
```

Looks like it works just as well. This interface is now reasonably high-level to wash out some of the ugliness of the thinnest possible `curl_easy_setopt` FFI, without obscuring the remaining C bookkeeping details we will explore.

4.8 Memory management

According to the documentation for `curl_easy_setopt`, the type of the third argument when *option* is `CURLOPT_ERRORBUFFER` is `char*`. Above, we've defined `set-curl-option-errorbuffer` to accept a `:pointer` as the new option value. However, there is a CFFI type `:string`, which translates Lisp strings to C strings when passed as arguments to foreign function calls. Why not, then, use `:string` as the CFFI type of the third argument? There are two reasons, both related to the necessity of breaking abstraction described in Section 4.6 [Breaking the abstraction], page 9.

The first reason also applies to `CURLOPT_URL`, which we will use to illustrate the point. Assuming we have changed the type of the third argument underlying `set-curl-option-url` to `:string`, look at these two equivalent forms.

```
(set-curl-option-url *easy-handle* "http://www.cliki.net/CFFI")

≡ (with-foreign-string (url "http://www.cliki.net/CFFI")
    (foreign-funcall "curl_easy_setopt" easy-handle *easy-handle*
                     curl-option :url :pointer url curl-code))
```

The latter, in fact, is mostly equivalent to what a foreign function call's macroexpansion actually does. As you can see, the Lisp string `"http://www.cliki.net/CFFI"` is copied into a `char` array and null-terminated; the pointer to beginning of this array, now a C string, is passed as a CFFI `:pointer` to the foreign function.

Unfortunately, the C abstraction has failed us, and we must break it. While `:string` works well for many `char*` arguments, it does not for cases like this. As the `curl_easy_setopt` documentation explains, "The string must remain present until curl no longer needs it, as it doesn't copy the string." The C string created by `with-foreign-string`, however, only has dynamic extent: it is "deallocated" when the body (above containing the `foreign-funcall` form) exits.

If we are supposed to keep the C string around, but it goes away, what happens when some `libcurl` function tries to access the URL string? We have reentered the dreaded world of C "undefined behavior". In some Lisps, it will probably get a chunk of the Lisp/C stack. You may segfault. You may get some random piece of other data from the heap. Maybe, in a world where "dynamic extent" is defined to be "infinite extent", everything will turn out fine. Regardless, results are likely to be almost universally unpleasant.[7]

Returning to the current `set-curl-option-url` interface, here is what we must do:

```
(let (easy-handle)
  (unwind-protect
    (with-foreign-string (url "http://www.cliki.net/CFFI")
      (setf easy-handle (curl-easy-init))
      (set-curl-option-url easy-handle url)
      #|do more with the easy-handle, like actually get the URL|#)
    (when easy-handle
      (curl-easy-cleanup easy-handle))))
```

That is fine for the single string defined here, but for every string option we want to pass, we have to surround the body of `with-foreign-string` with another `with-foreign-string` wrapper, or else do some extremely error-prone pointer manipulation and size calculation in advance. We could alleviate some of the pain with a recursively expanding macro, but this would not remove the need to modify the block every time we want to add an option, anathema as it is to a modular interface.

Before modifying the code to account for this case, consider the other reason we can't simply use `:string` as the foreign type. In C, a `char *` is a `char *`, not necessarily a string. The option `CURLOPT_ERRORBUFFER` accepts a `char *`, but does not expect anything about the data there. However, it does expect that some `libcurl` function we call later can write a C string of up to 255 characters there. We, the callers of the function, are expected to read the C string at a later time, exactly the opposite of what `:string` implies.

With the semantics for an input string in mind — namely, that the string should be kept around until we `curl_easy_cleanup` the easy handle — we are ready to extend the Lisp interface:

```
(defvar *easy-handle-cstrings* (make-hash-table)
  "Hashtable of easy handles to lists of C strings that may be
safely freed after the handle is freed.")

(defun make-easy-handle ()
  "Answer a new CURL easy interface handle, to which the lifetime
of C strings may be tied.  See `add-curl-handle-cstring'."
  (let ((easy-handle (curl-easy-init)))
    (setf (gethash easy-handle *easy-handle-cstrings*) '())
    easy-handle))

(defun free-easy-handle (handle)
```

[7] "*But I thought Lisp was supposed to protect me from all that buggy C crap!*" Before asking a question like that, remember that you are a stranger in a foreign land, whose residents have a completely different set of values.

```
  "Free CURL easy interface HANDLE and any C strings created to
be its options."
  (curl-easy-cleanup handle)
  (mapc #'foreign-string-free
        (gethash handle *easy-handle-cstrings*))
  (remhash handle *easy-handle-cstrings*))

(defun add-curl-handle-cstring (handle cstring)
  "Add CSTRING to be freed when HANDLE is, answering CSTRING."
  (car (push cstring (gethash handle *easy-handle-cstrings*))))
```

Here we have redefined the interface to create and free handles, to associate a list of allocated C strings with each handle while it exists. The strategy of using different function names to wrap around simple foreign functions is more common than the solution implemented earlier with `curry-curl-option-setter`, which was to modify the function name's function slot.[8]

Incidentally, the next step is to redefine `curry-curl-option-setter` to allocate C strings for the appropriate length of time, given a Lisp string as the `new-value` argument:

```
(defun curry-curl-option-setter (function-name option-keyword)
  "Wrap the function named by FUNCTION-NAME with a version that
curries the second argument as OPTION-KEYWORD.

This function is intended for use in DEFINE-CURL-OPTION-SETTER."
  (setf (symbol-function function-name)
          (let ((c-function (symbol-function function-name)))
            (lambda (easy-handle new-value)
              (funcall c-function easy-handle option-keyword
                       (if (stringp new-value)
                         (add-curl-handle-cstring
                          easy-handle
                          (foreign-string-alloc new-value))
                         new-value))))))
```

A quick analysis of the code shows that you need only reevaluate the `curl-option` enumeration definition to take advantage of these new semantics. Now, for good measure, let's reallocate the handle with the new functions we just defined, and set its URL:

```
CFFI-USER> (curl-easy-cleanup *easy-handle*)
⇒ NIL
CFFI-USER> (setf *easy-handle* (make-easy-handle))
⇒ #<FOREIGN-ADDRESS #x09844EE0>
CFFI-USER> (set-curl-option-nosignal *easy-handle* 1)
⇒ 0
CFFI-USER> (set-curl-option-url *easy-handle*
                                "http://www.cliki.net/CFFI")
⇒ 0
```

[8] There are advantages and disadvantages to each approach; I chose to `(setf symbol-function)` earlier because it entailed generating fewer magic function names.

For fun, let's inspect the Lisp value of the C string that was created to hold "http://www.cliki.net/CFFI". By virtue of the implementation of add-curl-handle-cstring, it should be accessible through the hash table defined:

```
CFFI-USER> (foreign-string-to-lisp
            (car (gethash *easy-handle* *easy-handle-cstrings*)))
⇒ "http://www.cliki.net/CFFI"
```

Looks like that worked, and libcurl now knows what URL we want to retrieve.

Finally, we turn back to the :errorbuffer option mentioned at the beginning of this section. Whereas the abstraction added to support string inputs works fine for cases like CURLOPT_URL, it hides the detail of keeping the C string; for :errorbuffer, however, we need that C string.

In a moment, we'll define something slightly cleaner, but for now, remember that you can always hack around anything. We're modifying handle creation, so make sure you free the old handle before redefining free-easy-handle.

```
(defvar *easy-handle-errorbuffers* (make-hash-table)
  "Hashtable of easy handles to C strings serving as error
writeback buffers.")

;;; An extra byte is very little to pay for peace of mind.
(defparameter *curl-error-size* 257
  "Minimum char[] size used by cURL to report errors.")

(defun make-easy-handle ()
  "Answer a new CURL easy interface handle, to which the lifetime
of C strings may be tied.  See `add-curl-handle-cstring'."
  (let ((easy-handle (curl-easy-init)))
    (setf (gethash easy-handle *easy-handle-cstrings*) '())
    (setf (gethash easy-handle *easy-handle-errorbuffers*)
          (foreign-alloc :char :count *curl-error-size*
                         :initial-element 0))
    easy-handle))

(defun free-easy-handle (handle)
  "Free CURL easy interface HANDLE and any C strings created to
be its options."
  (curl-easy-cleanup handle)
  (foreign-free (gethash handle *easy-handle-errorbuffers*))
  (remhash handle *easy-handle-errorbuffers*)
  (mapc #'foreign-string-free
        (gethash handle *easy-handle-cstrings*))
  (remhash handle *easy-handle-cstrings*))

(defun get-easy-handle-error (handle)
  "Answer a string containing HANDLE's current error message."
  (foreign-string-to-lisp
   (gethash handle *easy-handle-errorbuffers*)))
```

Be sure to once again set the options we've set thus far. You may wish to define yet another wrapper function to do this.

4.9 Calling Lisp from C

If you have been reading curl_easy_setopt(3), you should have noticed that some options accept a function pointer. In particular, we need one function pointer to set as CURLOPT_

WRITEFUNCTION, to be called by libcurl rather than the reverse, in order to receive data as it is downloaded.

A binding writer without the aid of FFI usually approaches this problem by writing a C function that accepts C data, converts to the language's internal objects, and calls the callback provided by the user, again in a reverse of usual practices.

The CFFI approach to callbacks precisely mirrors its differences with the non-FFI approach on the "calling C from Lisp" side, which we have dealt with exclusively up to now. That is, you define a callback function in Lisp using defcallback, and CFFI effectively creates a C function to be passed as a function pointer.

> **Implementor's note:** *This is much trickier than calling C functions from Lisp, as it literally involves somehow generating a new C function that is as good as any created by the compiler. Therefore, not all Lisps support them. See Chapter 3 [Implementation Support], page 3, for information about CFFI support issues in this and other areas. You may want to consider changing to a Lisp that supports callbacks in order to continue with this tutorial.*

Defining a callback is very similar to defining a callout; the main difference is that we must provide some Lisp forms to be evaluated as part of the callback. Here is the signature for the function the :writefunction option takes:

```
size_t
function(void *ptr, size_t size, size_t nmemb, void *stream);
```

> **Implementor's note:** *size_t is almost always an unsigned int. You can get this and many other types using feature tests for your system by using cffi-grovel.*

The above signature trivially translates into a CFFI defcallback form, as follows.

```
;;; Alias in case size_t changes.
(defctype size :unsigned-int)

;;; To be set as the CURLOPT_WRITEFUNCTION of every easy handle.
(defcallback easy-write size ((ptr :pointer) (size size)
                              (nmemb size) (stream :pointer))
  (let ((data-size (* size nmemb)))
    (handler-case
      ;; We use the dynamically-bound *easy-write-procedure* to
      ;; call a closure with useful lexical context.
      (progn (funcall (symbol-value '*easy-write-procedure*)
                      (foreign-string-to-lisp ptr :count data-size))
             data-size)         ;indicates success
      ;; The WRITEFUNCTION should return something other than the
      ;; #bytes available to signal an error.
      (error () (if (zerop data-size) 1 0)))))
```

First, note the correlation of the first few forms, used to declare the C function's signature, with the signature in C syntax. We provide a Lisp name for the function, its return type, and a name and type for each argument.

In the body, we call the dynamically-bound *easy-write-procedure* with a "finished" translation, of pulling together the raw data and size into a Lisp string, rather than deal with the data directly. As part of calling `curl_easy_perform` later, we'll bind that variable to a closure with more useful lexical bindings than the top-level `defcallback` form.

Finally, we make a halfhearted effort to prevent non-local exits from unwinding the C stack, covering the most likely case with an `error` handler, which is usually triggered unexpectedly.[9] The reason is that most C code is written to understand its own idiosyncratic error condition, implemented above in the case of `curl_easy_perform`, and more "undefined behavior" can result if we just wipe C stack frames without allowing them to execute whatever cleanup actions as they like.

Using the `CURLoption` enumeration in `curl.h` once more, we can describe the new option by modifying and reevaluating `define-curl-options`.

```
(define-curl-options curl-option
    (long 0 objectpoint 10000 functionpoint 20000 off-t 30000)
  (:noprogress long 43)
  (:nosignal long 99)
  (:errorbuffer objectpoint 10)
  (:url objectpoint 2)
  (:writefunction functionpoint 11)) ;new item here
```

Finally, we can use the defined callback and the new `set-curl-option-writefunction` to finish configuring the easy handle, using the `callback` macro to retrieve a CFFI `:pointer`, which works like a function pointer in C code.

```
CFFI-USER> (set-curl-option-writefunction
            *easy-handle* (callback easy-write))
⇒ 0
```

4.10 A complete FFI?

With all options finally set and a medium-level interface developed, we can finish the definition and retrieve `http://www.cliki.net/CFFI`, as is done in the tutorial.

```
(defcfun "curl_easy_perform" curl-code
  (handle easy-handle))
```

```
CFFI-USER> (with-output-to-string (contents)
             (let ((*easy-write-procedure*
                     (lambda (string)
                       (write-string string contents))))
               (declare (special *easy-write-procedure*))
               (curl-easy-perform *easy-handle*)))
⇒ "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01//EN\"
...
Now fear, comprehensively</P>
"
```

---

9 Unfortunately, we can't protect against *all* non-local exits, such as `return`s and `throw`s, because `unwind-protect` cannot be used to "short-circuit" a non-local exit in Common Lisp, due to proposal `minimal` in ANSI issue EXIT-EXTENT (`http://www.lisp.org/HyperSpec/Issues/iss152-writeup.html`). Furthermore, binding an `error` handler prevents higher-up code from invoking restarts that may be provided under the callback's dynamic context. Such is the way of compromise.

Of course, that itself is slightly unwieldy, so you may want to define a function around it that simply retrieves a URL. I will leave synthesis of all the relevant REPL forms presented thus far into a single function as an exercise for the reader.

The remaining sections of this tutorial explore some advanced features of CFFI; the definition of new types will receive special attention. Some of these features are essential for particular foreign function calls; some are very helpful when trying to develop a Lispy interface to C.

4.11 Defining new types

We've occasionally used the `defctype` macro in previous sections as a kind of documentation, much what you'd use `typedef` for in C. We also tried one special kind of type definition, the `defcenum` type. See [defcstruct], page 35, for a definition macro that may come in handy if you need to use C `struct`s as data.

However, all of these are mostly sugar for the powerful underlying foreign type interface called *type translators*. You can easily define new translators for any simple named foreign type. Since we've defined the new type `curl-code` to use as the return type for various `libcurl` functions, we can use that to directly convert cURL errors to Lisp errors.

`defctype`'s purpose is to define simple `typedef`-like aliases. In order to use *type translators* we must use the `define-foreign-type` macro. So let's redefine `curl-code` using it.

```
(define-foreign-type curl-code-type ()
  ()
  (:actual-type :int)
  (:simple-parser curl-code))
```

`define-foreign-type` is a thin wrapper around `defclass`. For now, all you need to know in the context of this example is that it does what `(defctype curl-code :int)` would do and, additionally, defines a new class `curl-code-type` which we will take advantage of shortly.

The `CURLcode` enumeration seems to follow the typical error code convention of '`0`' meaning all is well, and each non-zero integer indicating a different kind of error. We can apply that trivially to differentiate between normal exits and error exits.

```
(define-condition curl-code-error (error)
  (($code :initarg :curl-code :reader curl-error-code))
  (:report (lambda (c stream)
             (format stream "libcurl function returned error ~A"
                            (curl-error-code c))))
  (:documentation "Signalled when a libcurl function answers
a code other than CURLE_OK."))

(defmethod translate-from-foreign (value (type curl-code-type))
  "Raise a CURL-CODE-ERROR if VALUE, a curl-code, is non-zero."
  (if (zerop value)
      :curle-ok
      (error 'curl-code-error :curl-code value)))
```

The heart of this translator is new method `translate-from-foreign`. By specializing the *type* parameter on `curl-code-type`, we immediately modify the behavior of every function that returns a `curl-code` to pass the result through this new method.

To see the translator in action, try invoking a function that returns a `curl-code`. You need to reevaluate the respective `defcfun` form so that it picks up the new `curl-code` definition.

```
CFFI-USER> (set-curl-option-nosignal *easy-handle* 1)
⇒ :CURLE-OK
```

As the result was '0', the new method returned `:curle-ok`, just as specified.[10] I will leave disjoining the separate `CURLcode`s into condition types and improving the `:report` function as an exercise for you.

The creation of `*easy-handle-cstrings*` and `*easy-handle-errorbuffers*` as properties of `easy-handle`s is a kluge. What we really want is a Lisp structure that stores these properties along with the C pointer. Unfortunately, `easy-handle` is currently just a fancy name for the foreign type `:pointer`; the actual pointer object varies from Common Lisp implementation to implementation, needing only to satisfy `pointerp` and be returned from `make-pointer` and friends.

One solution that would allow us to define a new Lisp structure to represent `easy-handle`s would be to write a wrapper around every function that currently takes an `easy-handle`; the wrapper would extract the pointer and pass it to the foreign function. However, we can use type translators to more elegantly integrate this "translation" into the foreign function calling framework, using `translate-to-foreign`.

```
(defclass easy-handle ()
  ((pointer :initform (curl-easy-init)
            :documentation "Foreign pointer from curl_easy_init")
   (error-buffer
    :initform (foreign-alloc :char :count *curl-error-size*
                             :initial-element 0)
    :documentation "C string describing last error")
   (c-strings :initform '()
              :documentation "C strings set as options"))
  (:documentation "I am a parameterization you may pass to
curl-easy-perform to perform a cURL network protocol request."))

(defmethod initialize-instance :after ((self easy-handle) &key)
  (set-curl-option-errorbuffer self (slot-value self 'error-buffer)))

(defun add-curl-handle-cstring (handle cstring)
  "Add CSTRING to be freed when HANDLE is, answering CSTRING."
  (car (push cstring (slot-value handle 'c-strings))))

(defun get-easy-handle-error (handle)
  "Answer a string containing HANDLE's current error message."
  (foreign-string-to-lisp
   (slot-value handle 'error-buffer)))

(defun free-easy-handle (handle)
  "Free CURL easy interface HANDLE and any C strings created to
be its options."
```

[10] It might be better to return `(values)` than `:curle-ok` in real code, but this is good for illustration.

```
  (with-slots (pointer error-buffer c-strings) handle
    (curl-easy-cleanup pointer)
    (foreign-free error-buffer)
    (mapc #'foreign-string-free c-strings)))

(define-foreign-type easy-handle-type ()
  ()
  (:actual-type :pointer)
  (:simple-parser easy-handle))

(defmethod translate-to-foreign (handle (type easy-handle-type))
  "Extract the pointer from an easy-HANDLE."
  (slot-value handle 'pointer))
```

While we changed some of the Lisp functions defined earlier to use CLOS slots rather than hash tables, the foreign functions work just as well as they did before.

The greatest strength, and the greatest limitation, of the type translator comes from its generalized interface. As stated previously, we could define all foreign function calls in terms of the primitive foreign types provided by CFFI. The type translator interface allows us to cleanly specify the relationship between Lisp and C data, independent of where it appears in a function call. This independence comes at a price; for example, it cannot be used to modify translation semantics based on other arguments to a function call. In these cases, you should rely on other features of Lisp, rather than the powerful, yet domain-specific, type translator interface.

4.12 What's next?

CFFI provides a rich and powerful foundation for communicating with foreign libraries; as we have seen, it is up to you to make that experience a pleasantly Lispy one. This tutorial does not cover all the features of CFFI; please see the rest of the manual for details. In particular, if something seems obviously missing, it is likely that either code or a good reason for lack of code is already present.

> **Implementor's note:** *There are some other things in CFFI that might deserve tutorial sections, such as free-translated-object, or structs. Let us know which ones you care about.*

5 Wrapper generators

CFFI's interface is designed for human programmers, being aimed at aesthetic as well as technical sophistication. However, there are a few programs aimed at translating C and C++ header files, or approximations thereof, into CFFI forms constituting a foreign interface to the symbols in those files.

These wrapper generators are known to support output of CFFI forms.

Verrazano (`http://www.cliki.net/Verrazano`)
: Designed specifically for Common Lisp. Uses GCC's parser output in XML format to discover functions, variables, and other header file data. This means you need GCC to generate forms; on the other hand, the parser employed is mostly compliant with ANSI C.

SWIG (`http://www.cliki.net/SWIG`)
: A foreign interface generator originally designed to generate Python bindings, it has been ported to many other systems, including CFFI in version 1.3.28. Includes its own C declaration munger, not intended to be fully-compliant with ANSI C.

First, this manual does not describe use of these other programs; they have documentation of their own. If you have problems using a generated interface, please look at the output CFFI forms and verify that they are a correct CFFI interface to the library in question; if they are correct, contact CFFI developers with details, keeping in mind that they communicate in terms of those forms rather than any particular wrapper generator. Otherwise, contact the maintainers of the wrapper generator you are using, provided you can reasonably expect more accuracy from the generator.

When is more accuracy an unreasonable expectation? As described in the tutorial (see Section 4.6 [Breaking the abstraction], page 9), the information in C declarations is insufficient to completely describe every interface. In fact, it is quite common to run into an interface that cannot be handled automatically, and generators should be excused from generating a complete interface in these cases.

As further described in the tutorial, the thinnest Lisp interface to a C function is not always the most pleasant one. In many cases, you will want to manually write a Lispier interface to the C functions that interest you.

Wrapper generators should be treated as time-savers, not complete automation of the full foreign interface writing job. Reports of the amount of work done by generators vary from 30% to 90%. The incremental development style enabled by CFFI generally reduces this proportion below that for languages like Python.

6 Foreign Types

Foreign types describe how data is translated back and forth between C and Lisp. CFFI provides various built-in types and allows the user to define new types.

6.1 Built-In Types

`:char` [Foreign Type]

`:unsigned-char` [Foreign Type]

`:short` [Foreign Type]

`:unsigned-short` [Foreign Type]

`:int` [Foreign Type]

`:unsigned-int` [Foreign Type]

`:long` [Foreign Type]

`:unsigned-long` [Foreign Type]

`:long-long` [Foreign Type]

`:unsigned-long-long` [Foreign Type]

These types correspond to the native C integer types according to the ABI of the Lisp implementation's host system.

`:long-long` and `:unsigned-long-long` are not supported natively on all implementations. However, they are emulated by `mem-ref` and `mem-set`.

When those types are **not** available, the symbol `cffi-sys::no-long-long` is pushed into `*features*`.

`:uchar` [Foreign Type]

`:ushort` [Foreign Type]

`:uint` [Foreign Type]

`:ulong` [Foreign Type]

`:llong` [Foreign Type]

`:ullong` [Foreign Type]

For convenience, the above types are provided as shortcuts for `unsigned-char`, `unsigned-short`, `unsigned-int`, `unsigned-long`, `long-long` and `unsigned-long-long`, respectively.

`:int8` [Foreign Type]

`:uint8` [Foreign Type]

`:int16` [Foreign Type]

`:uint16` [Foreign Type]

`:int32` [Foreign Type]

`:uint32` [Foreign Type]

`:int64` [Foreign Type]

`:uint64` [Foreign Type]

`:size` [Foreign Type]

`:ssize` [Foreign Type]

`:intptr` [Foreign Type]

`:uintptr` [Foreign Type]

`:ptrdiff` [Foreign Type]

`:offset` [Foreign Type]

Foreign integer types of specific sizes, corresponding to the C types defined in `stdint.h`.

`:float` [Foreign Type]

`:double` [Foreign Type]

On all systems, the `:float` and `:double` types represent a C `float` and `double`, respectively. On most but not all systems, `:float` and `:double` represent a Lisp `single-float` and `double-float`, respectively. It is not so useful to consider the relationship between Lisp types and C types as isomorphic, as simply to recognize the relationship, and relative precision, among each respective category.

`:long-double` [Foreign Type]

This type is only supported on SCL.

`:pointer` *&optional type* [Foreign Type]

A foreign pointer to an object of any type, corresponding to `void *`. You can optionally specify type of pointer (e.g. `(:pointer :char)`). Although CFFI won't do anything with that information yet, it is useful for documentation purposes.

`:void` [Foreign Type]

No type at all. Only valid as the return type of a function.

6.2 Other Types

CFFI also provides a few useful types that aren't built-in C types.

`:string` [Foreign Type]

The `:string` type performs automatic conversion between Lisp and C strings. Note that, in the case of functions the converted C string will have dynamic extent (i.e. it will be automatically freed after the foreign function returns).

In addition to Lisp strings, this type will accept foreign pointers and pass them unmodified.

A method for [free-translated-object], page 54, is specialized for this type. So, for example, foreign strings allocated by this type and passed to a foreign function will be freed after the function returns.

```
CFFI> (foreign-funcall "getenv" :string "SHELL" :string)
⇒ "/bin/bash"
```

```
CFFI> (with-foreign-string (str "abcdef")
        (foreign-funcall "strlen" :string str :int))
⇒ 6
```

`:string+ptr` [Foreign Type]

Like `:string` but returns a list with two values when convert from C to Lisp: a Lisp string and the C string's foreign pointer.

```
CFFI> (foreign-funcall "getenv" :string "SHELL" :string+ptr)
⇒ ("/bin/bash" #.(SB-SYS:INT-SAP #XBFFFFC6F))
```

`:boolean` *&optional* (*base-type :int*) [Foreign Type]

The `:boolean` type converts between a Lisp boolean and a C boolean. It canonicalizes to *base-type* which is `:int` by default.

```
(convert-to-foreign nil :boolean) ⇒ 0
(convert-to-foreign t :boolean) ⇒ 1
(convert-from-foreign 0 :boolean) ⇒ nil
(convert-from-foreign 1 :boolean) ⇒ t
```

`:bool` [Foreign Type]

The `:bool` type represents the C99 `_Bool` or C++ `bool`. Its size is usually 1 byte except on OSX where it's an `int`.

`:wrapper` *base-type &key to-c from-c* [Foreign Type]

The `:wrapper` type stores two symbols passed to the *to-c* and *from-c* arguments. When a value is being translated to or from C, this type `funcall`s the respective symbol.

`:wrapper` types will be typedefs for *base-type* and will inherit its translators, if any.

Here's an example of how the `:boolean` type could be defined in terms of `:wrapper`.

```
(defun bool-c-to-lisp (value)
  (not (zerop value)))

(defun bool-lisp-to-c (value)
  (if value 1 0))

(defctype my-bool (:wrapper :int :from-c bool-c-to-lisp
                                 :to-c bool-lisp-to-c))

(convert-to-foreign nil 'my-bool) ⇒ 0
(convert-from-foreign 1 'my-bool) ⇒ t
```

6.3 Defining Foreign Types

You can define simple C-like `typedef`s through the `defctype` macro. Defining a typedef is as simple as giving `defctype` a new name and the name of the type to be wrapped.

```
;;; Define MY-INT as an alias for the built-in type :INT.
(defctype my-int :int)
```

With this type definition, one can, for instance, declare arguments to foreign functions as having the type `my-int`, and they will be passed as integers.

More complex types

CFFI offers another way to define types through `define-foreign-type`, a thin wrapper macro around `defclass`. As an example, let's go through the steps needed to define a `(my-string &key encoding)` type. First, we need to define our type class:

```
(define-foreign-type my-string-type ()
  ((encoding :reader string-type-encoding :initarg :encoding))
  (:actual-type :pointer))
```

The `:actual-type` class option tells CFFI that this type will ultimately be passed to and received from foreign code as a `:pointer`. Now you need to tell CFFI how to parse a type specification such as `(my-string :encoding :utf8)` into an instance of `my-string-type`. We do that with `define-parse-method`:

```
(define-parse-method my-string (&key (encoding :utf-8))
  (make-instance 'my-string-type :encoding encoding))
```

The next section describes how make this type actually translate between C and Lisp strings.

6.4 Foreign Type Translators

Type translators are used to automatically convert Lisp values to or from foreign values. For example, using type translators, one can take the `my-string` type defined in the previous section and specify that it should:

- convert C strings to Lisp strings;
- convert Lisp strings to newly allocated C strings;
- free said C strings when they are no longer needed.

In order to tell CFFI how to automatically convert Lisp values to foreign values, define a specialized method for the `translate-to-foreign` generic function:

```
;;; Define a method that converts Lisp strings to C strings.
(defmethod translate-to-foreign (string (type my-string-type))
  (foreign-string-alloc string :encoding (string-type-encoding type)))
```

From now on, whenever an object is passed as a `my-string` to a foreign function, this method will be invoked to convert the Lisp value. To perform the inverse operation, which is needed for functions that return a `my-string`, specialize the `translate-from-foreign` generic function in the same manner:

```
;;; Define a method that converts C strings to Lisp strings.
(defmethod translate-from-foreign (pointer (type my-string-type))
  (foreign-string-to-lisp pointer :encoding (string-type-encoding type)))
```

When a `translate-to-foreign` method requires allocation of foreign memory, you must also define a `free-translated-object` method to free the memory once the foreign object is no longer needed, otherwise you'll be faced with memory leaks. This generic function is called automatically by CFFI when passing objects to foreign functions. Let's do that:

```
;;; Free strings allocated by translate-to-foreign.
(defmethod free-translated-object (pointer (type my-string-type) param)
  (declare (ignore param))
  (foreign-string-free pointer))
```

In this specific example, we don't need the *param* argument, so we ignore it. See [free-translated-object], page 54, for an explanation of its purpose and how you can use it.

A type translator does not necessarily need to convert the value. For example, one could define a typedef for `:pointer` that ensures, in the `translate-to-foreign` method, that the value is not a null pointer, signalling an error if a null pointer is passed. This would prevent some pointer errors when calling foreign functions that cannot handle null pointers.

Please note: these methods are meant as extensible hooks only, and you should not call them directly. Use `convert-to-foreign`, `convert-from-foreign` and `free-converted-object` instead.

See Section 4.11 [Defining new types], page 18, for another example of type translators.

6.5 Optimizing Type Translators

Being based on generic functions, the type translation mechanism described above can add a bit of overhead. This is usually not significant, but we nevertheless provide a way of getting rid of the overhead for the cases where it matters.

A good way to understand this issue is to look at the code generated by `defcfun`. Consider the following example using the previously defined `my-string` type:

```
CFFI> (macroexpand-1 '(defcfun foo my-string (x my-string)))
;; (simplified, downcased, etc...)
(defun foo (x)
  (multiple-value-bind (#:G2019 #:PARAM3149)
      (translate-to-foreign x #<MY-STRING-TYPE {11ED5A79}>)
    (unwind-protect
        (translate-from-foreign
         (foreign-funcall "foo" :pointer #:G2019 :pointer)
         #<MY-STRING-TYPE {11ED5659}>)
      (free-translated-object #:G2019 #<MY-STRING-TYPE {11ED51A79}>
                              #:PARAM3149))))
```

In order to get rid of those generic function calls, CFFI has another set of extensible generic functions that provide functionality similar to CL's compiler macros: `expand-to-foreign-dyn`, `expand-to-foreign` and `expand-from-foreign`. Here's how one could define a `my-boolean` with them:

```
(define-foreign-type my-boolean-type ()
  ()
  (:actual-type :int)
  (:simple-parser my-boolean))

(defmethod expand-to-foreign (value (type my-boolean-type))
  `(if ,value 1 0))

(defmethod expand-from-foreign (value (type my-boolean-type))
  `(not (zerop ,value)))
```

And here's what the macroexpansion of a function using this type would look like:

```
CFFI> (macroexpand-1 '(defcfun bar my-boolean (x my-boolean)))
```

```
;; (simplified, downcased, etc...)
(defun bar (x)
  (let ((#:g3182 (if x 1 0)))
    (not (zerop (foreign-funcall "bar" :int #:g3182 :int)))))
```

No generic function overhead.

Let's go back to our `my-string` type. The expansion interface has no equivalent of `free-translated-object`; you must instead define a method on `expand-to-foreign-dyn`, the third generic function in this interface. This is especially useful when you can allocate something much more efficiently if you know the object has dynamic extent, as is the case with function calls that don't save the relevant allocated arguments.

This exactly what we need for the `my-string` type:

```
(defmethod expand-from-foreign (form (type my-string-type))
  `(foreign-string-to-lisp ,form))

(defmethod expand-to-foreign-dyn (value var body (type my-string-type))
  (let ((encoding (string-type-encoding type)))
    `(with-foreign-string (,var ,value :encoding ',encoding)
       ,@body)))
```

So let's look at the macro expansion:

```
CFFI> (macroexpand-1 '(defcfun foo my-string (x my-string)))
;; (simplified, downcased, etc...)
(defun foo (x)
  (with-foreign-string (#:G2021 X :encoding ':utf-8)
    (foreign-string-to-lisp
     (foreign-funcall "foo" :pointer #:g2021 :pointer))))
```

Again, no generic function overhead.

Other details

To short-circuit expansion and use the `translate-*` functions instead, simply call the next method. Return its result in cases where your method cannot generate an appropriate replacement for it. This analogous to the `&whole form` mechanism compiler macros provide.

The `expand-*` methods have precedence over their `translate-*` counterparts and are guaranteed to be used in `defcfun`, `foreign-funcall`, `defcvar` and `defcallback`. If you define a method on each of the `expand-*` generic functions, you are guaranteed to have full control over the expressions generated for type translation in these macros.

They may or may not be used in other CFFI operators that need to translate between Lisp and C data; you may only assume that `expand-*` methods will probably only be called during Lisp compilation.

`expand-to-foreign-dyn` has precedence over `expand-to-foreign` and is only used in `defcfun` and `foreign-funcall`, only making sense in those contexts.

Important note: this set of generic functions is called at macroexpansion time. Methods are defined when loaded or evaluated, not compiled. You are responsible for ensuring that your `expand-*` methods are defined when the `foreign-funcall` or other forms that use them are compiled. One way to do this is to put the method definitions earlier in the file and inside an appropriate `eval-when` form; another way is to always load a separate Lisp or FASL file containing your `expand-*` definitions before compiling files with forms that ought to use them. Otherwise, they will not be found and the runtime translators will be used instead.

6.6 Foreign Structure Types

For more involved C types than simple aliases to built-in types, such as you can make with `defctype`, CFFI allows declaration of structures and unions with `defcstruct` and `defcunion`.

For example, consider this fictional C structure declaration holding some personal information:

```
struct person {
  int number;
  char* reason;
};
```

The equivalent `defcstruct` form follows:

```
(defcstruct person
  (number :int)
  (reason :string))
```

By default, [convert-from-foreign], page 31, (and also [mem-ref], page 71) will make a plist with slot names as keys, and [convert-to-foreign], page 32, will translate such a plist to a foreign structure. A user wishing to define other translations should use the `:class` argument to [defcstruct], page 35, and then define methods for [translate-from-foreign], page 55, and [translate-into-foreign-memory], page 57, that specialize on this class, possibly calling `call-next-method` to translate from and to the plists rather than provide a direct interface to the foreign object. The macro `translation-forms-for-class` will generate the forms necessary to translate a Lisp class into a foreign structure and vice versa.

Please note that this interface is only for those that must know about the values contained in a relevant struct. If the library you are interfacing returns an opaque pointer that needs only be passed to other C library functions, by all means just use `:pointer` or a type-safe definition munged together with `defctype` and type translation. To pass or return a structure by value to a function, load the cffi-libffi system and specify the structure as `(:struct` *structure-name*`)`. To pass or return the pointer, you can use either `:pointer` or `(:pointer (:struct` *structure-name*`))`.

Optimizing translate-into-foreign-memory

Just like how [translate-from-foreign], page 55, had `expand-from-foreign` to optimize away the generic function call and [translate-to-foreign], page 56, had the same in `expand-to-foreign`, [translate-into-foreign-memory], page 57, has `expand-into-foreign-memory`.

Let's use our `person` struct in an example. However, we are going to spice it up by using a lisp struct rather than a plist to represent the person in lisp.

First we redefine `person` very slightly.

```
(defcstruct (person :class c-person)
  (number :int)
```

```
  (reason :string))
```

By adding `:class` we can specialize the `translate-*` methods on the type `c-person`.

Next we define a lisp struct to use instead of the plists.

```
(defstruct lisp-person
  (number 0 :type integer)
  (reason "" :type string))
```

And now let's define the type translators we know already:

```
(defmethod translate-from-foreign (ptr (type c-person))
  (with-foreign-slots ((number reason) ptr (:struct person))
    (make-lisp-person :number number :reason reason)))

(defmethod expand-from-foreign (ptr (type c-person))
  `(with-foreign-slots ((number reason) ,ptr (:struct person))
     (make-lisp-person :number number :reason reason)))

(defmethod translate-into-foreign-memory (value (type c-person) ptr)
  (with-foreign-slots ((number reason) ptr (:struct person))
    (setf number (lisp-person-number value)
          reason (lisp-person-reason value))))
```

At this point everything works, we can convert to and from our `lisp-person` and foreign `person`. If we macroexpand

```
(setf (mem-aref ptr '(:struct person)) x)
```

we get something like:

```
(let ((#:store879 x))
  (translate-into-foreign-memory #:store879 #<c-person person>
                                 (inc-pointer ptr 0))
  #:store879)
```

Which is good, but now we can do better and get rid of that generic function call to `translate-into-foreign-memory`.

```
(defmethod expand-into-foreign-memory (value (type c-person) ptr)
  `(with-foreign-slots ((number reason) ,ptr (:struct person))
     (setf number (lisp-person-number ,value)
           reason (lisp-person-reason ,value))))
```

Now we can expand again so see the changes:

```
;; this:
(setf (mem-aref ptr '(:struct person)) x)

;; expands to this
;; (simplified, downcased, etc..)
(let ((#:store887 x))
  (with-foreign-slots ((number reason) (inc-pointer ptr 0) (:struct person))
    (setf number (lisp-person-number #:store887)
          reason (lisp-person-reason #:store887))) #:store887)
```

And there we are, no generic function overhead.

Compatibility note

Previous versions of CFFI accepted the "bare" *structure-name* as a type specification, which was interpreted as a pointer to the structure. This is deprecated and produces a style warning. Using this deprecated form means that [mem-aref], page 70, retains its prior meaning and returns a pointer. Using the `(:struct` *`structure-name`*`)` form for the type, [mem-aref], page 70, provides a Lisp object translated from the structure (by default a plist). Thus the semantics are consistent with all types in returning the object as represented in Lisp, and not a pointer, with the exception of the "bare" structure compatibility retained. In order to obtain the pointer, you should use the function [mem-aptr], page 69.

See [defcstruct], page 35, for more details.

6.7 Allocating Foreign Objects

See Section 7.2 [Allocating Foreign Memory], page 60.

convert-from-foreign

Syntax

`convert-from-foreign` *foreign-value type* ⇒ *value* [Function]

Arguments and Values

foreign-value
: The primitive C value as returned from a primitive foreign function or from `convert-to-foreign`.

type
: A CFFI type specifier.

value
: The Lisp value translated from *foreign-value*.

Description

This is an external interface to the type translation facility. In the implementation, all foreign functions are ultimately defined as type translation wrappers around primitive foreign function invocations.

This function is available mostly for inspection of the type translation process, and possibly optimization of special cases of your foreign function calls.

Its behavior is better described under `translate-from-foreign`'s documentation.

Examples

```
CFFI-USER> (convert-to-foreign "a boat" :string)
⇒ #<FOREIGN-ADDRESS #x097ACDC0>
⇒ T
CFFI-USER> (convert-from-foreign * :string)
⇒ "a boat"
```

See Also

[convert-to-foreign], page 32,
[free-converted-object], page 53,
[translate-from-foreign], page 55,

convert-to-foreign

Syntax

`convert-to-foreign` *value type* ⇒ *foreign-value, alloc-params* [Function]

Arguments and Values

value The Lisp object to be translated to a foreign object.

type A CFFI type specifier.

foreign-value
The primitive C value, ready to be passed to a primitive foreign function.

alloc-params
Something of a translation state; you must pass it to `free-converted-object` along with the foreign value for that to work.

Description

This is an external interface to the type translation facility. In the implementation, all foreign functions are ultimately defined as type translation wrappers around primitive foreign function invocations.

This function is available mostly for inspection of the type translation process, and possibly optimization of special cases of your foreign function calls.

Its behavior is better described under `translate-to-foreign`'s documentation.

Examples

```
CFFI-USER> (convert-to-foreign t :boolean)
⇒ 1
⇒ NIL
CFFI-USER> (convert-to-foreign "hello, world" :string)
⇒ #<FOREIGN-ADDRESS #x097C5F80>
⇒ T
CFFI-USER> (code-char (mem-aref * :char 5))
⇒ #\,
```

See Also

[convert-from-foreign], page 31,
[free-converted-object], page 53,
[translate-to-foreign], page 56,

defbitfield

Syntax

`defbitfield` *name-and-options* **&body** *masks* [Macro]

masks ::= [docstring] { (symbol value) }*

name-and-options ::= name | (name &optional (base-type :int))

Arguments and Values

name The name of the new bitfield type.

docstring A documentation string, ignored.

base-type A symbol denoting a foreign type.

symbol A Lisp symbol.

value An integer representing a bitmask.

Description

The `defbitfield` macro is used to define foreign types that map lists of symbols to integer values.

If *value* is omitted, it will be computed as follows: find the greatest *value* previously used, including those so computed, with only a single 1-bit in its binary representation (that is, powers of two), and left-shift it by one. This rule guarantees that a computed *value* cannot clash with previous values, but may clash with future explicitly specified values.

Symbol lists will be automatically converted to values and vice versa when being passed as arguments to or returned from foreign functions, respectively. The same applies to any other situations where an object of a bitfield type is expected.

Types defined with `defbitfield` canonicalize to *base-type* which is `:int` by default.

Examples

```
(defbitfield open-flags
  (:rdonly #x0000)
  :wronly                ;#x0001
  :rdwr                  ;...
  :nonblock
  :append
  (:creat  #x0200))
  ;; etc...

CFFI> (foreign-bitfield-symbols 'open-flags #b1101)
⇒ (:WRONLY :NONBLOCK :APPEND)

CFFI> (foreign-bitfield-value 'open-flags '(:rdwr :creat))
⇒ 514   ; #x0202
```

```
(defcfun ("open" unix-open) :int
  (path :string)
  (flags open-flags)
  (mode :uint16)) ; unportable

CFFI> (unix-open "/tmp/foo" '(:wronly :creat) #o644)
⇒ #<an fd>

;;; Consider also the following lispier wrapper around open()
(defun lispier-open (path mode &rest flags)
  (unix-open path flags mode))
```

See Also

[foreign-bitfield-value], page 44,
[foreign-bitfield-symbols], page 43,

defcstruct

Syntax

`defcstruct` *name-and-options* **&body** *doc-and-slots* ⇒ *name* [Macro]

name-and-options ::= structure-name | (structure-name &key size)
doc-and-slots ::= [docstring] { (slot-name slot-type &key count offset) }*

Arguments and Values

structure-name
: The name of new structure type.

docstring A documentation string, ignored.

slot-name A symbol naming the slot. It must be unique among slot names in this structure.

size Use this option to override the size (in bytes) of the struct.

slot-type The type specifier for the slot.

count Used to declare an array of size *count* inside the structure. Defaults to `1` as such an array and a single element are semantically equivalent.

offset Overrides the slot's offset. The next slot's offset is calculated based on this one.

Description

This defines a new CFFI aggregate type akin to C `structs`. In other words, it specifies that foreign objects of the type *structure-name* are groups of different pieces of data, or "slots", of the *slot-types*, distinguished from each other by the *slot-names*. Each structure is located in memory at a position, and the slots are allocated sequentially beginning at that point in memory (with some padding allowances as defined by the C ABI, unless otherwise requested by specifying an *offset* from the beginning of the structure (offset 0).

In other words, it is isomorphic to the C `struct`, giving several extra features.

There are two kinds of slots, for the two kinds of CFFI types:

Simple Contain a single instance of a type that canonicalizes to a built-in type, such as `:long` or `:pointer`. Used for simple CFFI types.

Aggregate Contain an embedded structure or union, or an array of objects. Used for aggregate CFFI types.

The use of CLOS terminology for the structure-related features is intentional; structure definitions are very much like classes with (far) fewer features.

Examples

```
(defcstruct point
  "Point structure."
  (x :int)
  (y :int))

CFFI> (with-foreign-object (ptr 'point)
```

```
          ;; Initialize the slots
          (setf (foreign-slot-value ptr 'point 'x) 42
                (foreign-slot-value ptr 'point 'y) 42)
          ;; Return a list with the coordinates
          (with-foreign-slots ((x y) ptr point)
            (list x y)))
⇒ (42 42)

;; Using the :size and :offset options to define a partial structure.
;; (this is useful when you are interested in only a few slots
;; of a big foreign structure)

(defcstruct (foo :size 32)
  "Some struct with 32 bytes."
                        ; <16 bytes we don't care about>
  (x :int :offset 16)   ; an int at offset 16
  (y :int)              ; another int at offset 16+sizeof(int)
                        ; <a couple more bytes we don't care about>
  (z :char :offset 24)) ; a char at offset 24
                        ; <7 more bytes ignored (since size is 32)>

CFFI> (foreign-type-size 'foo)
⇒ 32

;;; Using :count to define arrays inside of a struct.
(defcstruct video_tuner
  (name :char :count 32))
```

See Also

[foreign-slot-pointer], page 49,
[foreign-slot-value], page 50,
[with-foreign-slots], page 58,

defcunion

Syntax

`defcunion` *name* **&body** *doc-and-slots* ⇒ *name* [Macro]

doc-and-slots ::= [docstring] { (slot-name slot-type &key count) }*

Arguments and Values

name The name of new union type.

docstring A documentation string, ignored.

slot-name A symbol naming the slot.

slot-type The type specifier for the slot.

count Used to declare an array of size *count* inside the structure.

Description

A union is a structure in which all slots have an offset of zero. It is isomorphic to the C `union`. Therefore, you should use the usual foreign structure operations for accessing a union's slots.

Examples

```
(defcunion uint32-bytes
  (int-value :unsigned-int)
  (bytes :unsigned-char :count 4))
```

See Also

[foreign-slot-pointer], page 49,
[foreign-slot-value], page 50,

defctype

Syntax

`defctype` *name base-type* **&optional** *documentation* [Macro]

Arguments and Values

name The name of the new foreign type.

base-type A symbol or a list defining the new type.

documentation
A documentation string, currently ignored.

Description

The `defctype` macro provides a mechanism similar to C's `typedef` to define new types. The new type inherits *base-type*'s translators, if any. There is no way to define translations for types defined with `defctype`. For that, you should use [define-foreign-type], page 41.

Examples

```
(defctype my-string :string
  "My own string type.")

(defctype long-bools (:boolean :long)
  "Booleans that map to C longs.")
```

See Also

[define-foreign-type], page 41,

defcenum

Syntax

`defcenum` *name-and-options* **&body** *enum-list* [Macro]
enum-list ::= [docstring] { keyword | (keyword value) }*
name-and-options ::= name | (name &optional (base-type :int) &key allow-undeclared-values)

Arguments and Values

name The name of the new enum type.

docstring A documentation string, ignored.

base-type A symbol denoting a foreign type.

allow-undeclared-values
Whether to pass through integer values that were not explicitly declared in the enum when translating from foreign memory.

keyword A keyword symbol.

value An index value for a keyword.

Description

The `defcenum` macro is used to define foreign types that map keyword symbols to integer values, similar to the C `enum` type.

If *value* is omitted its value will either be 0, if it's the first entry, or it it will continue the progression from the last specified value.

Keywords will be automatically converted to values and vice-versa when being passed as arguments to or returned from foreign functions, respectively. The same applies to any other situations where an object of an `enum` type is expected.

If a value should be translated to lisp that is not declared in the enum, an error will be signalled. You can elide this error and instead make it pass the original enum value by specifying *allow-undeclared-values*. This can be useful for very large enumerations of which we only care about a subset of values, or for enumerations that should allow for client or vendor extensions that we cannot know about.

Types defined with `defcenum` canonicalize to *base-type* which is `:int` by default.

Examples

```
(defcenum boolean
  :no
  :yes)

CFFI> (foreign-enum-value 'boolean :no)
⇒ 0
```

```
(defcenum numbers
```

```
  (:one 1)
  :two
  (:four 4))

CFFI> (foreign-enum-keyword 'numbers 2)
⇒ :TWO
```

See Also

[foreign-enum-value], page 46,
[foreign-enum-keyword], page 45,

define-foreign-type

Syntax

`define-foreign-type` *class-name supers slots* **&rest** *options* ⇒ *class-name* [Macro]

options ::= (`:actual-type` *type*) | (`:simple-parser` *symbol*) | *regular defclass option*

Arguments and Values

class-name
A symbol naming the new foreign type class.

supers A list of symbols naming the super classes.

slots A list of slot definitions, passed to `defclass`.

Description

The macro `define-foreign-type` defines a new class *class-name*. It is a thin wrapper around `defclass`. Among other things, it ensures that *class-name* becomes a subclass of *foreign-type*, what you need to know about that is that there's an initarg `:actual-type` which serves the same purpose as `defctype`'s *base-type* argument.

Examples

Taken from CFFI's `:boolean` type definition:

```
(define-foreign-type :boolean (&optional (base-type :int))
  "Boolean type. Maps to an :int by default. Only accepts integer types."
  (ecase base-type
    ((:char
      :unsigned-char
      :int
      :unsigned-int
      :long
      :unsigned-long) base-type)))

CFFI> (canonicalize-foreign-type :boolean)
⇒ :INT
CFFI> (canonicalize-foreign-type '(:boolean :long))
⇒ :LONG
CFFI> (canonicalize-foreign-type '(:boolean :float))
;; error signalled by ECASE.
```

See Also

[defctype], page 38,
[define-parse-method], page 42,

define-parse-method

Syntax

`define-parse-method` *name lambda-list* **&body** *body* ⇒ *name* [Macro]

Arguments and Values

type-name
: A symbol naming the new foreign type.

lambda-list
: A lambda list which is the argument list of the new foreign type.

body One or more forms that provide a definition of the new foreign type.

Description

Examples

Taken from CFFI's `:boolean` type definition:

```
(define-foreign-type :boolean (&optional (base-type :int))
  "Boolean type. Maps to an :int by default. Only accepts integer types."
  (ecase base-type
    ((:char
      :unsigned-char
      :int
      :unsigned-int
      :long
      :unsigned-long) base-type)))

CFFI> (canonicalize-foreign-type :boolean)
⇒ :INT
CFFI> (canonicalize-foreign-type '(:boolean :long))
⇒ :LONG
CFFI> (canonicalize-foreign-type '(:boolean :float))
;; error signalled by ECASE.
```

See Also

[define-foreign-type], page 41,

foreign-bitfield-symbols

Syntax

`foreign-bitfield-symbols` *type value* ⇒ *symbols* [Function]

Arguments and Values

type A bitfield type.

value An integer.

symbols A potentially shared list of symbols. `nil`.

Description

The function `foreign-bitfield-symbols` returns a possibly shared list of symbols that correspond to *value* in *type*.

Examples

```
(defbitfield flags
  (flag-a 1)
  (flag-b 2)
  (flag-c 4))

CFFI> (foreign-bitfield-symbols 'flags #b101)
⇒ (FLAG-A FLAG-C)
```

See Also

[defbitfield], page 33,
[foreign-bitfield-value], page 44,

foreign-bitfield-value

Syntax

`foreign-bitfield-value` *type symbols* ⇒ *value* [Function]

Arguments and Values

type A `bitfield` type.

symbol A Lisp symbol.

value An integer.

Description

The function `foreign-bitfield-value` returns the *value* that corresponds to the symbols in the *symbols* list.

Examples

```
(defbitfield flags
  (flag-a 1)
  (flag-b 2)
  (flag-c 4))

CFFI> (foreign-bitfield-value 'flags '(flag-a flag-c))
⇒ 5  ; #b101
```

See Also

[defbitfield], page 33,
[foreign-bitfield-symbols], page 43,

foreign-enum-keyword

Syntax

`foreign-enum-keyword` *type value* **&key** *errorp* ⇒ *keyword* [Function]

Arguments and Values

type An `enum` type.

value An integer.

errorp If true (the default), signal an error if *value* is not defined in *type*. If false, `foreign-enum-keyword` returns `nil`.

keyword A keyword symbol.

Description

The function `foreign-enum-keyword` returns the keyword symbol that corresponds to *value* in *type*.

An error is signaled if *type* doesn't contain such *value* and *errorp* is true.

Examples

```
(defcenum boolean
  :no
  :yes)

CFFI> (foreign-enum-keyword 'boolean 1)
⇒ :YES
```

See Also

[defcenum], page 39,
[foreign-enum-value], page 46,

foreign-enum-value

Syntax

`foreign-enum-value` *type keyword* **&key** *errorp* ⇒ *value* [Function]

Arguments and Values

type An `enum` type.

keyword A keyword symbol.

errorp If true (the default), signal an error if *keyword* is not defined in *type*. If false, `foreign-enum-value` returns `nil`.

value An integer.

Description

The function `foreign-enum-value` returns the *value* that corresponds to *keyword* in *type*.

An error is signaled if *type* doesn't contain such *keyword*, and *errorp* is true.

Examples

```
(defcenum boolean
  :no
  :yes)

CFFI> (foreign-enum-value 'boolean :yes)
⇒ 1
```

See Also

[defcenum], page 39,
[foreign-enum-keyword], page 45,

foreign-slot-names

Syntax

`foreign-slot-names` *type* ⇒ *names* [Function]

Arguments and Values

type A foreign struct type.

names A list.

Description

The function `foreign-slot-names` returns a potentially shared list of slot *names* for the given structure *type*. This list has no particular order.

Examples

```
(defcstruct timeval
  (tv-secs :long)
  (tv-usecs :long))

CFFI> (foreign-slot-names '(:struct timeval))
⇒ (TV-SECS TV-USECS)
```

See Also

[defcstruct], page 35,
[foreign-slot-offset], page 48,
[foreign-slot-value], page 50,
[foreign-slot-pointer], page 49,

foreign-slot-offset

Syntax

`foreign-slot-offset` *type slot-name* ⇒ *offset* [Function]

Arguments and Values

type A foreign struct type.

slot-name A symbol.

offset An integer.

Description

The function `foreign-slot-offset` returns the *offset* in bytes of a slot in a foreign struct type.

Examples

```
(defcstruct timeval
  (tv-secs :long)
  (tv-usecs :long))

CFFI> (foreign-slot-offset '(:struct timeval) 'tv-secs)
⇒ 0
CFFI> (foreign-slot-offset '(:struct timeval) 'tv-usecs)
⇒ 4
```

See Also

[defcstruct], page 35,
[foreign-slot-names], page 47,
[foreign-slot-pointer], page 49,
[foreign-slot-value], page 50,

foreign-slot-pointer

Syntax

`foreign-slot-pointer` *ptr type slot-name* ⇒ *pointer* [Function]

Arguments and Values

ptr A pointer to a structure.

type A foreign structure type.

slot-names
A slot name in the *type*.

pointer A pointer to the slot *slot-name*.

Description

Returns a pointer to the location of the slot *slot-name* in a foreign object of type *type* at *ptr*. The returned pointer points inside the structure. Both the pointer and the memory it points to have the same extent as *ptr*.

For aggregate slots, this is the same value returned by `foreign-slot-value`.

Examples

```
(defcstruct point
  "Pointer structure."
  (x :int)
  (y :int))

CFFI> (with-foreign-object (ptr '(:struct point))
        (foreign-slot-pointer ptr '(:struct point) 'x))
⇒ #<FOREIGN-ADDRESS #xBFFF6E60>
;; Note: the exact pointer representation varies from lisp to lisp.
```

See Also

[defcstruct], page 35,
[foreign-slot-value], page 50,
[foreign-slot-names], page 47,
[foreign-slot-offset], page 48,

foreign-slot-value

Syntax

`foreign-slot-value` *ptr type slot-name* ⇒ *object* [Accessor]

Arguments and Values

ptr A pointer to a structure.

type A foreign structure type.

slot-name A symbol naming a slot in the structure type.

object The object contained in the slot specified by *slot-name*.

Description

For simple slots, `foreign-slot-value` returns the value of the object, such as a Lisp integer or pointer. In C, this would be expressed as `ptr->slot`.

For aggregate slots, a pointer inside the structure to the beginning of the slot's data is returned. In C, this would be expressed as `&ptr->slot`. This pointer and the memory it points to have the same extent as *ptr*.

There are compiler macros for `foreign-slot-value` and its `setf` expansion that open code the memory access when *type* and *slot-names* are constant at compile-time.

Examples

```
(defcstruct point
  "Pointer structure."
  (x :int)
  (y :int))

CFFI> (with-foreign-object (ptr '(:struct point))
        ;; Initialize the slots
        (setf (foreign-slot-value ptr '(:struct point) 'x) 42
              (foreign-slot-value ptr '(:struct point) 'y) 42)
        ;; Return a list with the coordinates
        (with-foreign-slots ((x y) ptr (:struct point))
          (list x y)))
⇒ (42 42)
```

See Also

[defcstruct], page 35,
[foreign-slot-names], page 47,
[foreign-slot-offset], page 48,
[foreign-slot-pointer], page 49,
[with-foreign-slots], page 58,

foreign-type-alignment

Syntax

`foreign-type-alignment` *type* ⇒ *alignment* [Function]

Arguments and Values

type A foreign type.

alignment An integer.

Description

The function `foreign-type-alignment` returns the *alignment* of *type* in bytes.

Examples

```
CFFI> (foreign-type-alignment :char)
⇒ 1
CFFI> (foreign-type-alignment :short)
⇒ 2
CFFI> (foreign-type-alignment :int)
⇒ 4
(defcstruct foo
  (a :char))

CFFI> (foreign-type-alignment '(:struct foo))
⇒ 1
```

See Also

[foreign-type-size], page 52,

foreign-type-size

Syntax

`foreign-type-size` *type* ⇒ *size* [Function]

Arguments and Values

type A foreign type.

size An integer.

Description

The function `foreign-type-size` return the *size* of *type* in bytes. This includes any padding within and following the in-memory representation as needed to create an array of *type* objects.

Examples

```
(defcstruct foo
  (a :double)
  (c :char))

CFFI> (foreign-type-size :double)
⇒ 8
CFFI> (foreign-type-size :char)
⇒ 1
CFFI> (foreign-type-size '(:struct foo))
⇒ 16
```

See Also

[foreign-type-alignment], page 51,

free-converted-object

Syntax

`free-converted-object` *foreign-value type params* [Function]

Arguments and Values

foreign-value
: The C object to be freed.

type A CFFI type specifier.

params The state returned as the second value from `convert-to-foreign`; used to implement the third argument to `free-translated-object`.

Description

The return value is unspecified.

This is an external interface to the type translation facility. In the implementation, all foreign functions are ultimately defined as type translation wrappers around primitive foreign function invocations.

This function is available mostly for inspection of the type translation process, and possibly optimization of special cases of your foreign function calls.

Its behavior is better described under `free-translated-object`'s documentation.

Examples

```
CFFI-USER> (convert-to-foreign "a boat" :string)
⇒ #<FOREIGN-ADDRESS #x097ACDC0>
⇒ T
CFFI-USER> (free-converted-object * :string t)
⇒ NIL
```

See Also

[convert-from-foreign], page 31,
[convert-to-foreign], page 32,
[free-translated-object], page 54,

free-translated-object

Syntax

`free-translated-object` *value type-name param* [Generic Function]

Arguments and Values

pointer The foreign value returned by `translate-to-foreign`.

type-name
A symbol naming a foreign type defined by `defctype`.

param The second value, if any, returned by `translate-to-foreign`.

Description

This generic function may be specialized by user code to perform automatic deallocation of foreign objects as they are passed to C functions.

Any methods defined on this generic function must EQL-specialize the *type-name* parameter on a symbol defined as a foreign type by the `defctype` macro.

See Also

Section 6.4 [Foreign Type Translators], page 25,
[translate-to-foreign], page 56,

translate-from-foreign

Syntax

`translate-from-foreign` *foreign-value type-name* ⇒ *lisp-value* [Generic Function]

Arguments and Values

foreign-value
: The foreign value to convert to a Lisp object.

type-name
: A symbol naming a foreign type defined by `defctype`.

lisp-value
: The lisp value to pass in place of `foreign-value` to Lisp code.

Description

This generic function is invoked by CFFI to convert a foreign value to a Lisp value, such as when returning from a foreign function, passing arguments to a callback function, or accessing a foreign variable.

To extend the CFFI type system by performing custom translations, this method may be specialized by EQL-specializing `type-name` on a symbol naming a foreign type defined with `defctype`. This method should return the appropriate Lisp value to use in place of the foreign value.

The results are undefined if the `type-name` parameter is specialized in any way except an EQL specializer on a foreign type defined with `defctype`. Specifically, translations may not be defined for built-in types.

See Also

Section 6.4 [Foreign Type Translators], page 25,
[translate-to-foreign], page 56,
[free-translated-object], page 54,

translate-to-foreign

Syntax

`translate-to-foreign` *lisp-value type-name* ⇒ *foreign-value, alloc-param* [Generic Function]

Arguments and Values

lisp-value The Lisp value to convert to foreign representation.

type-name
A symbol naming a foreign type defined by `defctype`.

foreign-value
The foreign value to pass in place of `lisp-value` to foreign code.

alloc-param
If present, this value will be passed to `free-translated-object`.

Description

This generic function is invoked by CFFI to convert a Lisp value to a foreign value, such as when passing arguments to a foreign function, returning a value from a callback, or setting a foreign variable. A "foreign value" is one appropriate for passing to the next-lowest translator, including the low-level translators that are ultimately invoked invisibly with CFFI.

To extend the CFFI type system by performing custom translations, this method may be specialized by EQL-specializing `type-name` on a symbol naming a foreign type defined with `defctype`. This method should return the appropriate foreign value to use in place of the Lisp value.

In cases where CFFI can determine the lifetime of the foreign object returned by this method, it will invoke `free-translated-object` on the foreign object at the appropriate time. If `translate-to-foreign` returns a second value, it will be passed as the `param` argument to `free-translated-object`. This can be used to establish communication between the allocation and deallocation methods.

The results are undefined if the `type-name` parameter is specialized in any way except an EQL specializer on a foreign type defined with `defctype`. Specifically, translations may not be defined for built-in types.

See Also

Section 6.4 [Foreign Type Translators], page 25,
[translate-from-foreign], page 55,
[free-translated-object], page 54,

translate-into-foreign-memory

Syntax

`translate-into-foreign-memory` *lisp-value type-name pointer* [Generic Function]

Arguments and Values

lisp-value The Lisp value to convert to foreign representation.

type-name
A symbol or list `(:struct` *`structure-name`*`)` naming a foreign type defined by `defctype`.

pointer The foreign pointer where the translated object should be stored.

Description

Translate the Lisp value into the foreign memory location given by pointer. The return value is not used.

with-foreign-slots

Syntax

`with-foreign-slots` (*vars ptr type*) **&body** *body* [Macro]

Arguments and Values

vars A list with each element a symbol, or list of length two with the first element `:pointer` and the second a symbol.

ptr A foreign pointer to a structure.

type A structure type.

body A list of forms to be executed.

Description

The `with-foreign-slots` macro creates local symbol macros for each var in *vars* to reference foreign slots in *ptr* of *type*. If the var is a list starting with `:pointer`, it will bind the pointer to the slot (rather than the value). It is similar to Common Lisp's `with-slots` macro.

Examples

```
(defcstruct tm
  (sec :int)
  (min :int)
  (hour :int)
  (mday :int)
  (mon  :int)
  (year :int)
  (wday :int)
  (yday :int)
  (isdst  :boolean)
  (zone   :string)
  (gmtoff :long))

CFFI> (with-foreign-object (time :int)
        (setf (mem-ref time :int)
              (foreign-funcall "time" :pointer (null-pointer) :int))
        (foreign-funcall "gmtime" :pointer time (:pointer (:struct tm))))
⇒ #<A Mac Pointer #x102A30>
CFFI> (with-foreign-slots ((sec min hour mday mon year) * (:struct tm))
        (format nil "~A:~A:~A, ~A/~A/~A"
                hour min sec (+ 1900 year) mon mday))
⇒ "7:22:47, 2005/8/2"
```

See Also

[defcstruct], page 35,
[defcunion], page 37,
[foreign-slot-value], page 50,

7 Pointers

All C data in CFFI is referenced through pointers. This includes defined C variables that hold immediate values, and integers.

To see why this is, consider the case of the C integer. It is not only an arbitrary representation for an integer, congruent to Lisp's fixnums; the C integer has a specific bit pattern in memory defined by the C ABI. Lisp has no such constraint on its fixnums; therefore, it only makes sense to think of fixnums as C integers if you assume that CFFI converts them when necessary, such as when storing one for use in a C function call, or as the value of a C variable. This requires defining an area of memory[1], represented through an effective address, and storing it there.

Due to this compartmentalization, it only makes sense to manipulate raw C data in Lisp through pointers to it. For example, while there may be a Lisp representation of a `struct` that is converted to C at store time, you may only manipulate its raw data through a pointer. The C compiler does this also, albeit informally.

7.1 Basic Pointer Operations

Manipulating pointers proper can be accomplished through most of the other operations defined in the Pointers dictionary, such as `make-pointer`, `pointer-address`, and `pointer-eq`. When using them, keep in mind that they merely manipulate the Lisp representation of pointers, not the values they point to.

`foreign-pointer` [Lisp Type]

> The pointers' representations differ from implementation to implementation and have different types. `foreign-pointer` provides a portable type alias to each of these types.

7.2 Allocating Foreign Memory

CFFI provides support for stack and heap C memory allocation. Stack allocation, done with `with-foreign-object`, is sometimes called "dynamic" allocation in Lisp, because memory allocated as such has dynamic extent, much as with `let` bindings of special variables.

This should not be confused with what C calls "dynamic" allocation, or that done with `malloc` and friends. This sort of heap allocation is done with `foreign-alloc`, creating objects that exist until freed with `foreign-free`.

7.3 Accessing Foreign Memory

When manipulating raw C data, consider that all pointers are pointing to an array. When you only want one C value, such as a single `struct`, this array only has one such value. It is worthwhile to remember that everything is an array, though, because this is also the semantic that C imposes natively.

C values are accessed as the `setf`-able places defined by `mem-aref` and `mem-ref`. Given a pointer and a CFFI type (see Chapter 6 [Foreign Types], page 22), either of these will

---

[1] The definition of *memory* includes the CPU registers.

dereference the pointer, translate the C data there back to Lisp, and return the result of said translation, performing the reverse operation when `setf`-ing. To decide which one to use, consider whether you would use the array index operator `[n]` or the pointer dereference `*` in C; use `mem-aref` for array indexing and `mem-ref` for pointer dereferencing.

foreign-free

Syntax

`foreign-free` *ptr* ⇒ *undefined* [Function]

Arguments and Values

ptr A foreign pointer.

Description

The `foreign-free` function frees a `ptr` previously allocated by `foreign-alloc`. The consequences of freeing a given pointer twice are undefined.

Examples

```
CFFI> (foreign-alloc :int)
⇒ #<A Mac Pointer #x1022E0>
CFFI> (foreign-free *)
⇒ NIL
```

See Also

[foreign-alloc], page 63,
[with-foreign-pointer], page 78,

foreign-alloc

Syntax

`foreign-alloc` *type* **&key** *initial-element initial-contents* (*count* **1**) *null-terminated-p* ⇒ *pointer* [Function]

Arguments and Values

type A foreign type.

initial-element
A Lisp object.

initial-contents
A sequence.

count An integer. Defaults to 1 or the length of *initial-contents* if supplied.

null-terminated-p
A boolean, false by default.

pointer A foreign pointer to the newly allocated memory.

Description

The `foreign-alloc` function allocates enough memory to hold *count* objects of type *type* and returns a *pointer*. This memory must be explicitly freed using `foreign-free` once it is no longer needed.

If *initial-element* is supplied, it is used to initialize the *count* objects the newly allocated memory holds.

If an *initial-contents* sequence is supplied, it must have a length less than or equal to *count* and each of its elements will be used to initialize the contents of the newly allocated memory.

If *count* is omitted and *initial-contents* is specified, it will default to `(length initial-contents)`.

initial-element and *initial-contents* are mutually exclusive.

When *null-terminated-p* is true, `(1+ (max count (length initial-contents)))` elements are allocated and the last one is set to `NULL`. Note that in this case *type* must be a pointer type (ie. a type that canonicalizes to `:pointer`), otherwise an error is signaled.

Examples

```
CFFI> (foreign-alloc :char)
⇒ #<A Mac Pointer #x102D80>     ; A pointer to 1 byte of memory.

CFFI> (foreign-alloc :char :count 20)
⇒ #<A Mac Pointer #x1024A0>     ; A pointer to 20 bytes of memory.

CFFI> (foreign-alloc :int :initial-element 12)
⇒ #<A Mac Pointer #x1028B0>
```

```
CFFI> (mem-ref * :int)
⇒ 12

CFFI> (foreign-alloc :int :initial-contents '(1 2 3))
⇒ #<A Mac Pointer #x102950>
CFFI> (loop for i from 0 below 3
            collect (mem-aref * :int i))
⇒ (1 2 3)

CFFI> (foreign-alloc :int :initial-contents #(1 2 3))
⇒ #<A Mac Pointer #x102960>
CFFI> (loop for i from 0 below 3
            collect (mem-aref * :int i))
⇒ (1 2 3)

;;; Allocate a char** pointer that points to newly allocated memory
;;; by the :string type translator for the string "foo".
CFFI> (foreign-alloc :string :initial-element "foo")
⇒ #<A Mac Pointer #x102C40>
;;; Allocate a null-terminated array of strings.
;;; (Note: FOREIGN-STRING-TO-LISP returns NIL when passed a null pointer)
CFFI> (foreign-alloc :string
                     :initial-contents '("foo" "bar" "baz")
                     :null-terminated-p t)
⇒ #<A Mac Pointer #x102D20>
CFFI> (loop for i from 0 below 4
            collect (mem-aref * :string i))
⇒ ("foo" "bar" "baz" NIL)
CFFI> (progn
        (dotimes (i 3)
          (foreign-free (mem-aref ** :pointer i)))
        (foreign-free **))
⇒ nil
```

See Also

[foreign-free], page 62,
[with-foreign-object], page 77,
[with-foreign-pointer], page 78,

foreign-symbol-pointer

Syntax

`foreign-symbol-pointer` *foreign-name* **&key** *library* ⇒ *pointer* [Function]

Arguments and Values

foreign-name
: A string.

pointer
: A foreign pointer, or `nil`.

library
: A Lisp symbol or an instance of `foreign-library`.

Description

The function `foreign-symbol-pointer` will return a foreign pointer corresponding to the foreign symbol denoted by the string *foreign-name*. If a foreign symbol named *foreign-name* doesn't exist, `nil` is returned.

ABI name manglings will be performed on *foreign-name* by `foreign-symbol-pointer` if necessary. (eg: adding a leading underscore on darwin/ppc)

library should name a foreign library as defined by `define-foreign-library`, `:default` (which is the default) or an instance of `foreign-library` as returned by `load-foreign-library`.

Important note: do not keep these pointers across saved Lisp cores as the foreign-library may move across sessions.

Examples

```
CFFI> (foreign-symbol-pointer "errno")
⇒ #<A Mac Pointer #xA0008130>
CFFI> (foreign-symbol-pointer "strerror")
⇒ #<A Mac Pointer #x9002D0F8>
CFFI> (foreign-funcall-pointer * () :int (mem-ref ** :int) :string)
⇒ "No such file or directory"

CFFI> (foreign-symbol-pointer "inexistent symbol")
⇒ NIL
```

See Also

[defcvar], page 88,

inc-pointer

Syntax

`inc-pointer` *pointer offset* ⇒ *new-pointer* [Function]

Arguments and Values

pointer
new-pointer
A foreign pointer.

offset An integer.

Description

The function `inc-pointer` will return a *new-pointer* pointing *offset* bytes past *pointer*.

Examples

```
CFFI> (foreign-string-alloc "Common Lisp")
⇒ #<A Mac Pointer #x102EA0>
CFFI> (inc-pointer * 7)
⇒ #<A Mac Pointer #x102EA7>
CFFI> (foreign-string-to-lisp *)
⇒ "Lisp"
```

See Also

[incf-pointer], page 67,
[make-pointer], page 68,
[pointerp], page 74,
[null-pointer], page 72,
[null-pointer-p], page 73,

incf-pointer

Syntax

`incf-pointer` *place* **&optional** (*offset* **1**) ⇒ *new-pointer* [Macro]

Arguments and Values

place A `setf` place.

new-pointer
A foreign pointer.

offset An integer.

Description

The `incf-pointer` macro takes the foreign pointer from *place* and creates a *new-pointer* incremented by *offset* bytes and which is stored in *place*.

Examples

```
CFFI> (defparameter *two-words* (foreign-string-alloc "Common Lisp"))
⇒ *TWO-WORDS*
CFFI> (defparameter *one-word* *two-words*)
⇒ *ONE-WORD*
CFFI> (incf-pointer *one-word* 7)
⇒ #.(SB-SYS:INT-SAP #X00600457)
CFFI> (foreign-string-to-lisp *one-word*)
⇒ "Lisp"
CFFI> (foreign-string-to-lisp *two-words*)
⇒ "Common Lisp"
```

See Also

[inc-pointer], page 66,
[make-pointer], page 68,
[pointerp], page 74,
[null-pointer], page 72,
[null-pointer-p], page 73,

make-pointer

Syntax

`make-pointer` *address* ⇒ *ptr* [Function]

Arguments and Values

address An integer.

ptr A foreign pointer.

Description

The function `make-pointer` will return a foreign pointer pointing to *address*.

Examples

```
CFFI> (make-pointer 42)
⇒ #<FOREIGN-ADDRESS #x0000002A>
CFFI> (pointerp *)
⇒ T
CFFI> (pointer-address **)
⇒ 42
CFFI> (inc-pointer *** -42)
⇒ #<FOREIGN-ADDRESS #x00000000>
CFFI> (null-pointer-p *)
⇒ T
CFFI> (typep ** 'foreign-pointer)
⇒ T
```

See Also

[inc-pointer], page 66,
[null-pointer], page 72,
[null-pointer-p], page 73,
[pointerp], page 74,
[pointer-address], page 75,
[pointer-eq], page 76,
[mem-ref], page 71,

mem-aptr

Syntax

`mem-aptr` *ptr type* **&optional** (*index* **0**) [Accessor]

Arguments and Values

ptr A foreign pointer.

type A foreign type.

index An integer.

new-value A Lisp value compatible with *type*.

Description

The `mem-aptr` function finds the pointer to an element of the array.

```
(mem-aptr ptr type n)

;; is identical to:

(inc-pointer ptr (* n (foreign-type-size type)))
```

Examples

```
CFFI> (with-foreign-string (str "Hello, foreign world!")
        (mem-aptr str :char 6))
⇒ #.(SB-SYS:INT-SAP #X0063D4B6)
```

mem-aref

Syntax

`mem-aref` *ptr type* **&optional** (*index* **0**) [Accessor]
(setf (**mem-aref** *ptr type &optional (index 0)) new-value*)

Arguments and Values

ptr A foreign pointer.

type A foreign type.

index An integer.

new-value A Lisp value compatible with *type*.

Description

The `mem-aref` function is similar to `mem-ref` but will automatically calculate the offset from an *index*.

```
(mem-aref ptr type n)

;; is identical to:

(mem-ref ptr type (* n (foreign-type-size type)))
```

Examples

```
CFFI> (with-foreign-string (str "Hello, foreign world!")
        (mem-aref str :char 6))
⇒ 32
CFFI> (code-char *)
⇒ #\Space

CFFI> (with-foreign-object (array :int 10)
        (loop for i below 10
              do (setf (mem-aref array :int i) (random 100)))
        (loop for i below 10 collect (mem-aref array :int i)))
⇒ (22 7 22 52 69 1 46 93 90 65)
```

Compatibility Note

For compatibility with older versions of CFFI, [mem-aref], page 70, will produce a pointer for the deprecated bare structure specification, but it is consistent with other types for the current specification form (`:struct` *`structure-name`*) and provides a Lisp object translated from the structure (by default a plist). In order to obtain the pointer, you should use the new function [mem-aptr], page 69.

See Also

[mem-ref], page 71,
[mem-aptr], page 69,

mem-ref

Syntax

`mem-ref` *ptr type* **&optional** *offset* ⇒ *object* [Accessor]

Arguments and Values

ptr A pointer.

type A foreign type.

offset An integer (in byte units).

object The value *ptr* points to.

Description

Examples

```
CFFI> (with-foreign-string (ptr "Saluton")
        (setf (mem-ref ptr :char 3) (char-code #\a))
        (loop for i from 0 below 8
              collect (code-char (mem-ref ptr :char i))))
⇒ (#\S #\a #\l #\a #\t #\o #\n #\Null)
CFFI> (setq ptr-to-int (foreign-alloc :int))
⇒ #<A Mac Pointer #x1047D0>
CFFI> (mem-ref ptr-to-int :int)
⇒ 1054619
CFFI> (setf (mem-ref ptr-to-int :int) 1984)
⇒ 1984
CFFI> (mem-ref ptr-to-int :int)
⇒ 1984
```

See Also

[mem-aref], page 70,

null-pointer

Syntax

`null-pointer` ⇒ *pointer* [Function]

Arguments and Values

pointer A `NULL` pointer.

Description

The function `null-pointer` returns a null pointer.

Examples

```
CFFI> (null-pointer)
⇒ #<A Null Mac Pointer>
CFFI> (pointerp *)
⇒ T
```

See Also

[null-pointer-p], page 73,
[make-pointer], page 68,

null-pointer-p

Syntax

`null-pointer-p` *ptr* ⇒ *boolean* [Function]

Arguments and Values

ptr A foreign pointer that may be a null pointer.

boolean `T` or `NIL`.

Description

The function `null-pointer-p` returns true if *ptr* is a null pointer and false otherwise.

Examples

```
CFFI> (null-pointer-p (null-pointer))
⇒ T
(defun contains-str-p (big little)
  (not (null-pointer-p
        (foreign-funcall "strstr" :string big :string little :pointer))))

CFFI> (contains-str-p "Popcorns" "corn")
⇒ T
CFFI> (contains-str-p "Popcorns" "salt")
⇒ NIL
```

See Also

[null-pointer], page 72,
[pointerp], page 74,

pointerp

Syntax

`pointerp` *ptr* ⇒ *boolean* [Function]

Arguments and Values

ptr An object that may be a foreign pointer.

boolean `T` or `NIL`.

Description

The function `pointerp` returns true if *ptr* is a foreign pointer and false otherwise.

Implementation-specific Notes

In Allegro CL, foreign pointers are integers thus in this implementation `pointerp` will return true for any ordinary integer.

Examples

```
CFFI> (foreign-alloc 32)
⇒ #<A Mac Pointer #x102D20>
CFFI> (pointerp *)
⇒ T
CFFI> (pointerp "this is not a pointer")
⇒ NIL
```

See Also

[make-pointer], page 68, [null-pointer-p], page 73,

pointer-address

Syntax

`pointer-address` *ptr* ⇒ *address* [Function]

Arguments and Values

ptr A foreign pointer.

address An integer.

Description

The function `pointer-address` will return the *address* of a foreign pointer *ptr*.

Examples

```
CFFI> (pointer-address (null-pointer))
⇒ 0
CFFI> (pointer-address (make-pointer 123))
⇒ 123
```

See Also

[make-pointer], page 68,
[inc-pointer], page 66,
[null-pointer], page 72,
[null-pointer-p], page 73,
[pointerp], page 74,
[pointer-eq], page 76,
[mem-ref], page 71,

pointer-eq

Syntax

`pointer-eq` *ptr1 ptr2* ⇒ *boolean* [Function]

Arguments and Values

ptr1
ptr2 A foreign pointer.

boolean `T` or `NIL`.

Description

The function `pointer-eq` returns true if *ptr1* and *ptr2* point to the same memory address and false otherwise.

Implementation-specific Notes

The representation of foreign pointers varies across the various Lisp implementations as does the behaviour of the built-in Common Lisp equality predicates. Comparing two pointers that point to the same address with `EQ` Lisps will return true on some Lisps, others require more general predicates like `EQL` or `EQUALP` and finally some will return false using any of these predicates. Therefore, for portability, you should use `POINTER-EQ`.

Examples

This is an example using SBCL, see the implementation-specific notes above.

```
CFFI> (eql (null-pointer) (null-pointer))
⇒ NIL
CFFI> (pointer-eq (null-pointer) (null-pointer))
⇒ T
```

See Also

[inc-pointer], page 66,

with-foreign-object, with-foreign-objects

Syntax

`with-foreign-object` (*var type* **&optional** *count*) **&body** *body* [Macro]

`with-foreign-objects` (*bindings*) **&body** *body* [Macro]

bindings ::= {(var type &optional count)}*

Arguments and Values

var A symbol.

type A foreign type, evaluated.

count An integer.

Description

The macros `with-foreign-object` and `with-foreign-objects` bind *var* to a pointer to *count* newly allocated objects of type *type* during *body*. The buffer has dynamic extent and may be stack allocated if supported by the host Lisp.

Examples

```
CFFI> (with-foreign-object (array :int 10)
        (dotimes (i 10)
          (setf (mem-aref array :int i) (random 100)))
        (loop for i below 10
              collect (mem-aref array :int i)))
⇒ (22 7 22 52 69 1 46 93 90 65)
```

See Also

[foreign-alloc], page 63,

with-foreign-pointer

Syntax

`with-foreign-pointer` (*var size* **&optional** *size-var*) **&body** *body* [Macro]

Arguments and Values

var
size-var A symbol.

size An integer.

body A list of forms to be executed.

Description

The `with-foreign-pointer` macro, binds *var* to *size* bytes of foreign memory during *body*. The pointer in *var* is invalid beyond the dynamic extend of *body* and may be stack-allocated if supported by the implementation.

If *size-var* is supplied, it will be bound to *size* during *body*.

Examples

```
CFFI> (with-foreign-pointer (string 4 size)
        (setf (mem-ref string :char (1- size)) 0)
        (lisp-string-to-foreign "Popcorns" string size)
        (loop for i from 0 below size
              collect (code-char (mem-ref string :char i))))
⇒ (#\P #\o #\p #\Null)
```

See Also

[foreign-alloc], page 63,
[foreign-free], page 62,

8 Strings

As with many languages, Lisp and C have special support for logical arrays of characters, going so far as to give them a special name, "strings". In that spirit, CFFI provides special support for translating between Lisp and C strings.

The `:string` type and the symbols related below also serve as an example of what you can do portably with CFFI; were it not included, you could write an equally functional `strings.lisp` without referring to any implementation-specific symbols.

default-foreign-encoding

Syntax

`*default-foreign-encoding*` [Special Variable]

Value type

A keyword.

Initial value

`:utf-8`

Description

This special variable holds the default foreign encoding.

Examples

```
CFFI> *default-foreign-encoding*
:utf-8
CFFI> (foreign-funcall "strdup" (:string :encoding :utf-16) "foo" :string)
"f"
CFFI> (let ((*default-foreign-encoding* :utf-16))
        (foreign-funcall "strdup" (:string :encoding :utf-16) "foo" :string))
"foo"
```

See also

Section 6.2 [Other Types], page 23, (`:string` type)
[foreign-string-alloc], page 81,
[foreign-string-to-lisp], page 83,
[lisp-string-to-foreign], page 84,
[with-foreign-string], page 85,
[with-foreign-pointer-as-string], page 86,

foreign-string-alloc

Syntax

`foreign-string-alloc` *string* **&key** *encoding null-terminated-p start end* ⇒ *pointer* [Function]

Arguments and Values

string A Lisp string.

encoding Foreign encoding. Defaults to `*default-foreign-encoding*`.

null-terminated-p
Boolean, defaults to true.

start, end Bounding index designators of *string*. 0 and `nil`, by default.

pointer A pointer to the newly allocated foreign string.

Description

The `foreign-string-alloc` function allocates foreign memory holding a copy of *string* converted using the specified *encoding*. *Start* specifies an offset into *string* and *end* marks the position following the last element of the foreign string.

This string must be freed with `foreign-string-free`.

If *null-terminated-p* is false, the string will not be null-terminated.

Examples

```
CFFI> (defparameter *str* (foreign-string-alloc "Hello, foreign world!"))
⇒ #<FOREIGN-ADDRESS #x00400560>
CFFI> (foreign-funcall "strlen" :pointer *str* :int)
⇒ 21
```

See Also

[foreign-string-free], page 82,
[with-foreign-string], page 85,

foreign-string-free

Syntax

`foreign-string-free` *pointer* [Function]

Arguments and Values

pointer A pointer to a string allocated by `foreign-string-alloc`.

Description

The `foreign-string-free` function frees a foreign string allocated by `foreign-string-alloc`.

Examples

See Also

[foreign-string-alloc], page 81,

foreign-string-to-lisp

Syntax

`foreign-string-to-lisp` *ptr* **&key** *offset count max-chars encoding* [Function]
⇒ *string*

Arguments and Values

ptr	A pointer.
offset	An integer greater than or equal to 0. Defauls to 0.
count	Either `nil` (the default), or an integer greater than or equal to 0.
max-chars	An integer greater than or equal to 0. `(1- array-total-size-limit)`, by default.
encoding	Foreign encoding. Defaults to `*default-foreign-encoding*`.
string	A Lisp string.

Description

The `foreign-string-to-lisp` function converts at most *count* octets from *ptr* into a Lisp string, using the defined *encoding*.

If *count* is `nil` (the default), characters are copied until *max-chars* is reached or a `NULL` character is found.

If *ptr* is a null pointer, returns `nil`.

Note that the `:string` type will automatically convert between Lisp strings and foreign strings.

Examples

```
CFFI> (foreign-funcall "getenv" :string "HOME" :pointer)
⇒ #<FOREIGN-ADDRESS #xBFFFFFD5>
CFFI> (foreign-string-to-lisp *)
⇒ "/Users/luis"
```

See Also

[lisp-string-to-foreign], page 84,
[foreign-string-alloc], page 81,

lisp-string-to-foreign

Syntax

`lisp-string-to-foreign` *string buffer bufsize* **&key** *start end offset encoding* ⇒ *buffer* [Function]

Arguments and Values

string A Lisp string.

buffer A foreign pointer.

bufsize An integer.

start, end Bounding index designators of *string*. 0 and `nil`, by default.

offset An integer greater than or equal to 0. Defauls to 0.

encoding Foreign encoding. Defaults to `*default-foreign-encoding*`.

Description

The `lisp-string-to-foreign` function copies at most *bufsize*-1 octets from a Lisp *string* using the specified *encoding* into *buffer*+*offset*. The foreign string will be null-terminated.

Start specifies an offset into *string* and *end* marks the position following the last element of the foreign string.

Examples

```
CFFI> (with-foreign-pointer-as-string (str 255)
        (lisp-string-to-foreign "Hello, foreign world!" str 6))
⇒ "Hello"
```

See Also

[foreign-string-alloc], page 81,
[foreign-string-to-lisp], page 83,
[with-foreign-pointer-as-string], page 86,

with-foreign-string, with-foreign-strings

Syntax

`with-foreign-string` (*var-or-vars string* **&rest** *args*) **&body** *body* [Macro]

`with-foreign-strings` (*bindings*) **&body** *body* [Macro]

var-or-vars ::= var | (var &optional octet-size-var)
bindings ::= {(var-or-vars string &rest args)}*

Arguments and Values

var, byte-size-var
: A symbol.

string
: A Lisp string.

body
: A list of forms to be executed.

Description

The `with-foreign-string` macro will bind *var* to a newly allocated foreign string containing *string*. *Args* is passed to the underlying `foreign-string-alloc` call.

If *octet-size-var* is provided, it will be bound the length of foreign string in octets including the null terminator.

Examples

```
CFFI> (with-foreign-string (foo "12345")
        (foreign-funcall "strlen" :pointer foo :int))
⇒ 5
```

See Also

[foreign-string-alloc], page 81,
[with-foreign-pointer-as-string], page 86,

with-foreign-pointer-as-string

Syntax

`with-foreign-pointer-as-string` (*var size* **&optional** *size-var* **&rest** *args*) **&body** *body* ⇒ *string* [Macro]

Arguments and Values

var A symbol.

string A Lisp string.

body List of forms to be executed.

Description

The `with-foreign-pointer-as-string` macro is similar to `with-foreign-pointer` except that *var* is used as the returned value of an implicit `progn` around *body*, after being converted to a Lisp string using the provided *args*.

Examples

```
CFFI> (with-foreign-pointer-as-string (str 6 str-size :encoding :ascii)
        (lisp-string-to-foreign "Hello, foreign world!" str str-size))
⇒ "Hello"
```

See Also

[foreign-string-alloc], page 81,
[with-foreign-string], page 85,

9 Variables

defcvar

Syntax

`defcvar` *name-and-options type* **&optional** *documentation* ⇒ *lisp-name* [Macro]

name-and-options ::= name | (name &key read-only (library :default))

name ::= lisp-name [foreign-name] | foreign-name [lisp-name]

Arguments and Values

foreign-name
: A string denoting a foreign function.

lisp-name
: A symbol naming the Lisp function to be created.

type
: A foreign type.

read-only
: A boolean.

documentation
: A Lisp string; not evaluated.

Description

The `defcvar` macro defines a symbol macro *lisp-name* that looks up *foreign-name* and dereferences it acording to *type*. It can also be `setf`ed, unless *read-only* is true, in which case an error will be signaled.

When one of *lisp-name* or *foreign-name* is omitted, the other is automatically derived using the following rules:

- Foreign names are converted to Lisp names by uppercasing, replacing underscores with hyphens, and wrapping around asterisks.
- Lisp names are converted to foreign names by lowercasing, replacing hyphens with underscores, and removing asterisks, if any.

Examples

```
CFFI> (defcvar "errno" :int)
⇒ *ERRNO*
CFFI> (foreign-funcall "strerror" :int *errno* :string)
⇒ "Inappropriate ioctl for device"
CFFI> (setf *errno* 1)
⇒ 1
CFFI> (foreign-funcall "strerror" :int *errno* :string)
⇒ "Operation not permitted"
```

Trying to modify a read-only foreign variable:

```
CFFI> (defcvar ("errno" +error-number+ :read-only t) :int)
⇒ +ERROR-NUMBER+
CFFI> (setf +error-number+ 12)
;; error Trying to modify read-only foreign var: +ERROR-NUMBER+.
```

Note that accessing `errno` *this way won't work with every implementation of the C standard library.*

See Also

[get-var-pointer], page 90,

get-var-pointer

Syntax

`get-var-pointer` *symbol* ⇒ *pointer* [Function]

Arguments and Values

symbol A symbol denoting a foreign variable defined with `defcvar`.

pointer A foreign pointer.

Description

The function `get-var-pointer` will return a *pointer* to the foreign global variable *symbol* previously defined with `defcvar`.

Examples

```
CFFI> (defcvar "errno" :int :read-only t)
⇒ *ERRNO*
CFFI> *errno*
⇒ 25
CFFI> (get-var-pointer '*errno*)
⇒ #<A Mac Pointer #xA0008130>
CFFI> (mem-ref * :int)
⇒ 25
```

See Also

[defcvar], page 88,

10 Functions

defcfun

Syntax

`defcfun` *name-and-options return-type* **&body** [*docstring*] *arguments* **[&rest]** ⇒ *lisp-name* [Macro]

name-and-options ::= *name* | (*name* &key *library convention*)
name ::= *lisp-name* [*foreign-name*] | *foreign-name* [*lisp-name*]
arguments ::= { (*arg-name arg-type*) }*

Arguments and Values

foreign-name
A string denoting a foreign function.

lisp-name A symbol naming the Lisp function to be created.

arg-name A symbol.

return-type
arg-type A foreign type.

convention
One of `:cdecl` (default) or `:stdcall`.

library A symbol designating a foreign library.

docstring A documentation string.

Description

The `defcfun` macro provides a declarative interface for defining Lisp functions that call foreign functions.

When one of *lisp-name* or *foreign-name* is omitted, the other is automatically derived using the following rules:

- Foreign names are converted to Lisp names by uppercasing and replacing underscores with hyphens.
- Lisp names are converted to foreign names by lowercasing and replacing hyphens with underscores.

If you place the symbol `&rest` in the end of the argument list after the fixed arguments, `defcfun` will treat the foreign function as a **variadic function**. The variadic arguments should be passed in a way similar to what `foreign-funcall` would expect. Unlike `foreign-funcall` though, `defcfun` will take care of doing argument promotion. Note that in this case `defcfun` will generate a Lisp *macro* instead of a function and will only work for Lisps that support `foreign-funcall.`

If a foreign structure is to be passed or returned by value (that is, the type is of the form `(:struct ...)`), then the cffi-libffi system must be loaded, which in turn depends on libffi (`http://sourceware.org/libffi/`), including the header files. Failure to load that system will result in an error. Variadic functions cannot at present accept or return structures by value.

Examples

```
(defcfun "strlen" :int
  "Calculate the length of a string."
  (n :string))

CFFI> (strlen "123")
⇒ 3
```

```
(defcfun ("abs" c-abs) :int (n :int))

CFFI> (c-abs -42)
⇒ 42
```

Function without arguments:

```
(defcfun "rand" :int)

CFFI> (rand)
⇒ 1804289383
```

Variadic function example:

```
(defcfun "sprintf" :int
  (str :pointer)
  (control :string)
  &rest)

CFFI> (with-foreign-pointer-as-string (s 100)
        (sprintf s "%c %d %.2f %s" :char 90 :short 42 :float pi
                 :string "super-locrian"))
⇒ "A 42 3.14 super-locrian"
```

See Also

[foreign-funcall], page 94,
[foreign-funcall-pointer], page 96,
[foreign-funcall-varargs], page 97,
[foreign-funcall-pointer-varargs], page 98,

foreign-funcall

Syntax

`foreign-funcall` *name-and-options* **&rest** *arguments* ⇒ *return-value* [Macro]
arguments ::= { *arg-type arg* }* [*return-type*]
name-and-options ::= *name* | (*name* &key *library convention*)

Arguments and Values

name A Lisp string.

arg-type A foreign type.

arg An argument of type *arg-type*.

return-type
A foreign type, `:void` by default.

return-value
A lisp object.

library A lisp symbol; not evaluated.

convention
One of `:cdecl` (default) or `:stdcall`.

Description

The `foreign-funcall` macro is the main primitive for calling foreign functions.

If a foreign structure is to be passed or returned by value (that is, the type is of the form `(:struct ...)`), then the cffi-libffi system must be loaded, which in turn depends on libffi (`http://sourceware.org/libffi/`), including the header files. Failure to load that system will result in an error. Variadic functions cannot at present accept or return structures by value.

Note: The return value of foreign-funcall on functions with a :void return type is still undefined.

Implementation-specific Notes

- Corman Lisp does not support `foreign-funcall`. On implementations that **don't** support `foreign-funcall` `cffi-sys::no-foreign-funcall` will be present in `*features*`. Note: in these Lisps you can still use the `defcfun` interface.

Examples

```
CFFI> (foreign-funcall "strlen" :string "foo" :int)
⇒ 3
```

Given the C code:

```
void print_number(int n)
{
    printf("N: %d\n", n);
```

```
}
CFFI> (foreign-funcall "print_number" :int 123456)
⊣ N: 123456
⇒ NIL
```

Or, equivalently:

```
CFFI> (foreign-funcall "print_number" :int 123456 :void)
⊣ N: 123456
⇒ NIL
CFFI> (foreign-funcall "printf" :string (format nil "%s: %d.~%")
                       :string "So long and thanks for all the fish"
                       :int 42 :int)
⊣ So long and thanks for all the fish: 42.
⇒ 41
```

See Also

[defcfun], page 92,
[foreign-funcall-pointer], page 96,

foreign-funcall-pointer

Syntax

`foreign-funcall-pointer` *pointer options* **&rest** *arguments* ⇒ *return-value* [Macro]

arguments ::= { *arg-type arg* }* [*return-type*]
options ::= (&key *convention*)

Arguments and Values

pointer A foreign pointer.

arg-type A foreign type.

arg An argument of type *arg-type*.

return-type
A foreign type, `:void` by default.

return-value
A lisp object.

convention
One of `:cdecl` (default) or `:stdcall`.

Description

The `foreign-funcall` macro is the main primitive for calling foreign functions.

Note: The return value of foreign-funcall on functions with a :void return type is still undefined.

Implementation-specific Notes

- Corman Lisp does not support `foreign-funcall`. On implementations that **don't** support `foreign-funcall cffi-sys::no-foreign-funcall` will be present in `*features*`. Note: in these Lisps you can still use the `defcfun` interface.

Examples

```
CFFI> (foreign-funcall-pointer (foreign-symbol-pointer "abs") ()
                                :int -42 :int)
⇒ 42
```

See Also

[defcfun], page 92,
[foreign-funcall], page 94,

foreign-funcall-varargs

Syntax

`foreign-funcall-varargs` *name-and-options* (*fixed-arguments*) **&rest** *arguments* ⇒ *return-value* [Macro]

fixed-arguments ::= { *arg-type arg* }* [*return-type*]
arguments ::= { *arg-type arg* }* [*return-type*]
name-and-options ::= *name* | (*name* &key *library convention*)

Arguments and Values

name A Lisp string.

arg-type A foreign type.

arg An argument of type *arg-type*.

return-type
A foreign type, `:void` by default.

return-value
A lisp object.

library A lisp symbol; not evaluated.

convention
One of `:cdecl` (default) or `:stdcall`.

Description

The `foreign-funcall-varargs` macro is the main primitive for calling foreign variadic functions. It behaves similarily to `foreign-funcall` except `fixed-arguments` are distinguished from the remaining arguments.

Examples

```
CFFI> (with-foreign-pointer-as-string (s 100)
        (setf (mem-ref s :char) 0)
        (foreign-funcall-varargs
         "sprintf" (:pointer s :string) "%.2f")
         :double (coerce pi 'double-float) :int))
⇒ 3.14
```

foreign-funcall-pointer-varargs

Syntax

`foreign-funcall-pointer-varargs` *pointer options* (*fixed-arguments*) **&rest** *arguments* ⇒ *return-value* [Macro]

fixed-arguments ::= { *arg-type arg* }* [*return-type*]
arguments ::= { *arg-type arg* }* [*return-type*]
options ::= (&key *convention*)

Arguments and Values

pointer A foreign pointer.

arg-type A foreign type.

arg An argument of type *arg-type*.

return-type
A foreign type, `:void` by default.

return-value
A lisp object.

convention
One of `:cdecl` (default) or `:stdcall`.

Description

The `foreign-funcall-pointer-varargs` macro is the main primitive for calling foreign variadic functions. It behaves similarily to `foreign-funcall-pointer` except `fixed-arguments` are distinguished from the remaining arguments.

Examples

```
CFFI> (with-foreign-pointer-as-string (s 100)
        (setf (mem-ref s :char) 0)
        (foreign-funcall-pointer-varargs
         (foreign-symbol-pointer "sprintf") () (:pointer s :string "%.2f")
         :double (coerce pi 'double-float) :int))
⇒ 3.14
```

translate-camelcase-name

Syntax

`translate-camelcase-name` *name* **&key** *upper-initial-p* *special-words* ⇒ *return-value* [Function]

Arguments and Values

name Either a symbol or a string.

upper-initial-p
A generalized boolean.

special words
A list of strings.

return-value
If *name* is a symbol, this is a string, and vice versa.

Description

`translate-camelcase-name` is a helper function for specializations of `translate-name-from-foreign` and `translate-name-to-foreign`. It handles the common case of converting between foreign camelCase names and lisp names. *upper-initial-p* indicates whether the first letter of the foreign name should be uppercase. *special-words* is a list of strings that should be treated atomically in translation. This list is case-sensitive.

Examples

```
CFFI> (translate-camelcase-name some-xml-function)
⇒ "someXmlFunction"
CFFI> (translate-camelcase-name some-xml-function :upper-initial-p t)
⇒ "SomeXmlFunction"
CFFI> (translate-camelcase-name some-xml-function :special-words '("XML"))
⇒ "someXMLFunction"
CFFI> (translate-camelcase-name "someXMLFunction")
⇒ SOME-X-M-L-FUNCTION
CFFI> (translate-camelcase-name "someXMLFunction" :special-words '("XML"))
⇒ SOME-XML-FUNCTION
```

See Also

[translate-name-from-foreign], page 100,
[translate-name-to-foreign], page 101,
[translate-underscore-separated-name], page 102,

translate-name-from-foreign

Syntax

`translate-name-from-foreign` *foreign-name package* **&optional** *varp* ⇒ *symbol* [Function]

Arguments and Values

foreign-name
: A string denoting a foreign function.

package
: A Lisp package

varp
: A generalized boolean.

symbol
: The Lisp symbol to be used a function name.

Description

`translate-name-from-foreign` is used by [defcfun], page 92, to handle the conversion of foreign names to lisp names. By default, it translates using [translate-underscore-separated-name], page 102. However, you can create specialized methods on this function to make translating more closely match the foreign library's naming conventions.

Specialize *package* on some package. This allows other packages to load libraries with different naming conventions.

Examples

```
CFFI> (defcfun "someXmlFunction" ...)
⇒ SOMEXMLFUNCTION
CFFI> (defmethod translate-name-from-foreign ((spec string)
                                              (package (eql *package*))
                                              &optional varp)
        (let ((name (translate-camelcase-name spec)))
          (if varp (intern (format nil "*~a*" name)) name)))
⇒ #<STANDARD-METHOD TRANSLATE-NAME-FROM-FOREIGN (STRING (EQL #<Package "SOME-PACKAGE">
CFFI> (defcfun "someXmlFunction" ...)
⇒ SOME-XML-FUNCTION
```

See Also

[defcfun], page 92,
[translate-camelcase-name], page 99,
[translate-name-to-foreign], page 101,
[translate-underscore-separated-name], page 102,

translate-name-to-foreign

Syntax

`translate-name-to-foreign` *lisp-name package* **&optional** *varp* ⇒ *string* [Function]

Arguments and Values

lisp-name A symbol naming the Lisp function to be created.

package A Lisp package

varp A generalized boolean.

string The string representing the foreign function name.

Description

`translate-name-to-foreign` is used by [defcfun], page 92, to handle the conversion of lisp names to foreign names. By default, it translates using [translate-underscore-separated-name], page 102. However, you can create specialized methods on this function to make translating more closely match the foreign library's naming conventions.

Specialize *package* on some package. This allows other packages to load libraries with different naming conventions.

Examples

```
CFFI> (defcfun some-xml-function ...)
⇒ "some_xml_function"
CFFI> (defmethod translate-name-to-foreign ((spec symbol)
                                            (package (eql *package*))
                                            &optional varp)
        (let ((name (translate-camelcase-name spec)))
          (if varp (subseq name 1 (1- (length name))) name)))
⇒ #<STANDARD-METHOD TRANSLATE-NAME-TO-FOREIGN (STRING (EQL #<Package "SOME-PACKAGE">))
CFFI> (defcfun some-xml-function ...)
⇒ "someXmlFunction"
```

See Also

[defcfun], page 92,
[translate-camelcase-name], page 99,
[translate-name-from-foreign], page 100,
[translate-underscore-separated-name], page 102,

translate-underscore-separated-name

Syntax

`translate-underscore-separated-name` *name* ⇒ *return-value* [Function]

Arguments and Values

name Either a symbol or a string.

return-value
If *name* is a symbol, this is a string, and vice versa.

Description

`translate-underscore-separated-name` is a helper function for specializations of [translate-name-from-foreign], page 100, and [translate-name-to-foreign], page 101. It handles the common case of converting between foreign underscore_separated names and lisp names.

Examples

```
CFFI> (translate-underscore-separated-name some-xml-function)
⇒ "some_xml_function"
CFFI> (translate-camelcase-name "some_xml_function")
⇒ SOME-XML-FUNCTION
```

See Also

[translate-name-from-foreign], page 100,
[translate-name-to-foreign], page 101,
[translate-camelcase-name], page 99,

11 Libraries

11.1 Defining a library

Almost all foreign code you might want to access exists in some kind of shared library. The meaning of *shared library* varies among platforms, but for our purposes, we will consider it to include `.so` files on UNIX, frameworks on Darwin (and derivatives like Mac OS X), and `.dll` files on Windows.

Bringing one of these libraries into the Lisp image is normally a two-step process.

1. Describe to CFFI how to load the library at some future point, depending on platform and other factors, with a `define-foreign-library` top-level form.
2. Load the library so defined with either a top-level `use-foreign-library` form or by calling the function `load-foreign-library`.

See Section 4.3 [Loading foreign libraries], page 5, for a working example of the above two steps.

11.2 Library definition style

Looking at the `libcurl` library definition presented earlier, you may ask why we did not simply do this:

```
(define-foreign-library libcurl
  (t (:default "libcurl")))
```

Indeed, this would work just as well on the computer on which I tested the tutorial. There are a couple of good reasons to provide the `.so`'s current version number, however. Namely, the versionless `.so` is not packaged on most UNIX systems along with the actual, fully-versioned library; instead, it is included in the "development" package along with C headers and static `.a` libraries.

The reason CFFI does not try to account for this lies in the meaning of the version numbers. A full treatment of shared library versions is beyond this manual's scope; see Section "Library interface versions" in *GNU Libtool*, for helpful information for the unfamiliar. For our purposes, consider that a mismatch between the library version with which you tested and the installed library version may cause undefined behavior.[1]

> **Implementor's note:** *Maybe some notes should go here about OS X, which I know little about. –stephen*

[1] Windows programmers may chafe at adding a UNIX-specific clause to `define-foreign-library`. Instead, ask why the Windows solution to library incompatibility is "include your own version of every library you use with every program".

close-foreign-library

Syntax

`close-foreign-library` *library* ⇒ *success* [Function]

Arguments and Values

library A symbol or an instance of `foreign-library`.

success A Lisp boolean.

Description

Closes *library* which can be a symbol designating a library define through `define-foreign-library` or an instance of `foreign-library` as returned by `load-foreign-library`.

See Also

[define-foreign-library], page 106,
[load-foreign-library], page 110,
[use-foreign-library], page 113,

darwin-framework-directories

Syntax

`*darwin-framework-directories*` [Special Variable]

Value type

A list, in which each element is a string, a pathname, or a simple Lisp expression.

Initial value

A list containing the following, in order: an expression corresponding to Darwin path `~/Library/Frameworks/`, `#P"/Library/Frameworks/"`, and `#P"/System/Library/Frameworks/"`.

Description

The meaning of "simple Lisp expression" is explained in [*foreign-library-directories*], page 108. In contrast to that variable, this is not a fallback search path; the default value described above is intended to be a reasonably complete search path on Darwin systems.

Examples

```
CFFI> (let ((lib (load-foreign-library '(:framework "OpenGL"))))
        (foreign-library-pathname lib))
⇒ #P"/System/Library/Frameworks/OpenGL.framework/OpenGL"
```

See also

[*foreign-library-directories*], page 108,
[define-foreign-library], page 106,

define-foreign-library

Syntax

`define-foreign-library` *name-and-options* { *load-clause* }* ⇒ *name* [Macro]

name-and-options ::= name | (name &key canary convention search-path)
load-clause ::= (feature library &key convention search-path)

Arguments and Values

name A symbol.

feature A feature expression.

library A library designator.

canary A string denoting a foreign symbol that will be searched in core before attempting to load the library. If that symbol is found, the library is assumed to be statically linked and `load-foreign-library` only marks the library as loaded.

Some implementations (Clisp, ECL, SBCL) natively support static linking, sometimes referred to as a *link kit*.

convention
One of `:cdecl` (default) or `:stdcall`

search-path
A path or list of paths where the library will be searched if not found in system-global directories. Paths specified in a load clause take priority over paths specified as library option, with *foreign-library-directories* having lowest priority.

Description

Creates a new library designator called *name*. The *load-clauses* describe how to load that designator when passed to `load-foreign-library` or `use-foreign-library`.

When trying to load the library *name*, the relevant function searches the *load-clauses* in order for the first one where *feature* evaluates to true. That happens for any of the following situations:

1. If *feature* is a symbol present in `common-lisp:*features*`.
2. If *feature* is a list, depending on `(first feature)`, a keyword:

   `:and` All of the feature expressions in `(rest feature)` are true.

   `:or` At least one of the feature expressions in `(rest feature)` is true.

   `:not` The feature expression `(second feature)` is not true.
3. Finally, if *feature* is `t`, this *load-clause* is picked unconditionally.

Upon finding the first true *feature*, the library loader then loads the *library*. The meaning of "library designator" is described in [load-foreign-library], page 110.

Functions associated to a library defined by `define-foreign-library` (e.g. through `defcfun`'s `:library` option, will inherit the library's options. The precedence is as follows:

1. `defcfun`/`foreign-funcall` specific options;

2. *load-clause* options;
3. global library options (the *name-and-options* argument)

Examples

See Section 4.3 [Loading foreign libraries], page 5.

See Also

[close-foreign-library], page 104,
[load-foreign-library], page 110,

foreign-library-directories

Syntax

`*foreign-library-directories*` [Special Variable]

Value type

A list, in which each element is a string, a pathname, or a simple Lisp expression.

Initial value

The empty list.

Description

You should not have to use this variable.

Most, if not all, Lisps supported by CFFI have a reasonable default search algorithm for foreign libraries. For example, Lisps for UNIX usually call `dlopen(3)`, which in turn looks in the system library directories. Only if that fails does CFFI look for the named library file in these directories, and load it from there if found.

Thus, this is intended to be a CFFI-only fallback to the library search configuration provided by your operating system. For example, if you distribute a foreign library with your Lisp package, you can add the library's containing directory to this list and portably expect CFFI to find it.

A *simple Lisp expression* is intended to provide functionality commonly used in search paths such as ASDF's[1], and is defined recursively as follows:[2]

1. A list, whose '`first`' is a function designator, and whose '`rest`' is a list of simple Lisp expressions to be evaluated and passed to the so-designated function. The result is the result of the function call.
2. A symbol, whose result is its symbol value.
3. Anything else evaluates to itself.

The result of evaluating the *simple Lisp expression* should yield a *designator* for a *list* of *pathname designators*.

Note: in Common Lisp, `#p"/foo/bar"` designates the *bar* file within the */foo* directory whereas `#p"/foo/bar/"` designates the */foo/bar* directory. Keep that in mind when customising the value of `*foreign-library-directories*`.

Examples

```
$ ls
⊣ liblibli.so    libli.lisp
```

In `libli.lisp`:

```
(pushnew #P"/home/sirian/lisp/libli/" *foreign-library-directories*
```

[1] See Section "Using asdf to load systems" in *asdf: another system definition facility*, for information on `asdf:*central-registry*`.

[2] See `mini-eval` in `libraries.lisp` for the source of this definition. As is always the case with a Lisp `eval`, it's easier to understand the Lisp definition than the english.

```
             :test #'equal)

    (load-foreign-library '(:default "liblibli"))
```

The following example would achieve the same effect:

```
    (pushnew '(merge-pathnames #p"lisp/libli/" (user-homedir-pathname))
              *foreign-library-directories*
              :test #'equal)
    ⇒ ((MERGE-PATHNAMES #P"lisp/libli/" (USER-HOMEDIR-PATHNAME)))

    (load-foreign-library '(:default "liblibli"))
```

See also

[*darwin-framework-directories*], page 105,
[define-foreign-library], page 106,

load-foreign-library

Syntax

`load-foreign-library` *library-designator* ⇒ *library* [Function]

Arguments and Values

library-designator
A library designator.

library-designator
An instance of `foreign-library`.

Description

Load the library indicated by *library-designator*. A *library designator* is defined as follows:

1. If a symbol, is considered a name previously defined with `define-foreign-library`.
2. If a string or pathname, passed as a namestring directly to the implementation's foreign library loader. If that fails, search the directories in `*foreign-library-directories*` with `cl:probe-file`; if found, the absolute path is passed to the implementation's loader.
3. If a list, the meaning depends on `(first library)`:

   `:framework`
   The second list element is taken to be a Darwin framework name, which is then searched in `*darwin-framework-directories*`, and loaded when found.

   `:or` Each remaining list element, itself a *library designator*, is loaded in order, until one succeeds.

   `:default` The name is transformed according to the platform's naming convention to shared libraries, and the resultant string is loaded as a *library designator*. For example, on UNIX, the name is suffixed with `.so`.

If the library is already loaded it will be reloaded.

If the load fails, signal a `load-foreign-library-error`.

Please note: For system libraries, you should not need to specify the directory containing the library. Each operating system has its own idea of a default search path, and you should rely on it when it is reasonable.

Implementation-specific Notes

On ECL platforms where its dynamic FFI is not supported (ie. when `:dffi` is not present in `*features*`), `cffi:load-foreign-library` does not work and you must use ECL's own `ffi:load-foreign-library` with a constant string argument.

Examples

See Section 4.3 [Loading foreign libraries], page 5.

See Also

[close-foreign-library], page 104,
[*darwin-framework-directories*], page 105,
[define-foreign-library], page 106,
[*foreign-library-directories*], page 108,
[load-foreign-library-error], page 112,
[use-foreign-library], page 113,

load-foreign-library-error

Syntax

`load-foreign-library-error` [Condition Type]

Class precedence list

`load-foreign-library-error`, `error`, `serious-condition`, `condition`, `t`

Description

Signalled when a foreign library load completely fails. The exact meaning of this varies depending on the real conditions at work, but almost universally, the implementation's error message is useless. However, CFFI does provide the useful restarts `retry` and `use-value`; invoke the `retry` restart to try loading the foreign library again, or the `use-value` restart to try loading a different foreign library designator.

See also

[load-foreign-library], page 110,

use-foreign-library

Syntax

`use-foreign-library` *name* [Macro]

Arguments and values

name A library designator; unevaluated.

Description

See [load-foreign-library], page 110, for the meaning of "library designator". This is intended to be the top-level form used idiomatically after a `define-foreign-library` form to go ahead and load the library. Finally, on implementations where the regular evaluation rule is insufficient for foreign library loading, it loads it at the required time.[1]

Examples

See Section 4.3 [Loading foreign libraries], page 5.

See also

[load-foreign-library], page 110,

[1] Namely, CMUCL. See `use-foreign-library` in `libraries.lisp` for details.

12 Callbacks

callback

Syntax

`callback` *symbol* ⇒ *pointer* [Macro]

Arguments and Values

symbol A symbol denoting a callback.

pointer
new-value A pointer.

Description

The `callback` macro is analogous to the standard CL special operator `function` and will return a pointer to the callback denoted by the symbol *name*.

Examples

```
CFFI> (defcallback sum :int ((a :int) (b :int))
        (+ a b))
⇒ SUM
CFFI> (callback sum)
⇒ #<A Mac Pointer #x102350>
```

See Also

[get-callback], page 118,
[defcallback], page 116,

defcallback

Syntax

`defcallback` *name-and-options return-type arguments* **&body** *body* ⇒ *name* [Macro]

name-and-options ::= name | (name &key convention)
arguments ::= ({ (arg-name arg-type) }*)

Arguments and Values

name A symbol naming the callback created.

return-type
The foreign type for the callback's return value.

arg-name A symbol.

arg-type A foreign type.

convention
One of `:cdecl` (default) or `:stdcall`.

Description

The `defcallback` macro defines a Lisp function that can be called from C. The arguments passed to this function will be converted to the appropriate Lisp representation and its return value will be converted to its C representation.

This Lisp function can be accessed by the `callback` macro or the `get-callback` function.

Portability note: `defcallback` will not work correctly on some Lisps if it's not a top-level form.

Examples

```
(defcfun "qsort" :void
  (base :pointer)
  (nmemb :int)
  (size :int)
  (fun-compar :pointer))

(defcallback < :int ((a :pointer) (b :pointer))
  (let ((x (mem-ref a :int))
        (y (mem-ref b :int)))
    (cond ((> x y) 1)
          ((< x y) -1)
          (t 0))))

CFFI> (with-foreign-object (array :int 10)
        ;; Initialize array.
        (loop for i from 0 and n in '(7 2 10 4 3 5 1 6 9 8)
```

```
              do (setf (mem-aref array :int i) n))
        ;; Sort it.
        (qsort array 10 (foreign-type-size :int) (callback <))
        ;; Return it as a list.
        (loop for i from 0 below 10
              collect (mem-aref array :int i)))
  ⇒ (1 2 3 4 5 6 7 8 9 10)
```

See Also

[callback], page 115,
[get-callback], page 118,

get-callback

Syntax

`get-callback` *symbol* ⇒ *pointer* [Accessor]

Arguments and Values

symbol A symbol denoting a callback.

pointer A pointer.

Description

This is the functional version of the `callback` macro. It returns a pointer to the callback named by *symbol* suitable, for example, to pass as arguments to foreign functions.

Examples

```
CFFI> (defcallback sum :int ((a :int) (b :int))
        (+ a b))
⇒ SUM
CFFI> (get-callback 'sum)
⇒ #<A Mac Pointer #x102350>
```

See Also

[callback], page 115,
[defcallback], page 116,

13 The Groveller

CFFI-Grovel is a tool which makes it easier to write CFFI declarations for libraries that are implemented in C. That is, it grovels through the system headers, getting information about types and structures, so you don't have to. This is especially important for libraries which are implemented in different ways by different vendors, such as the UNIX/POSIX functions. The CFFI declarations are usually quite different from platform to platform, but the information you give to CFFI-Grovel is the same. Hence, much less work is required!

If you use ASDF, CFFI-Grovel is integrated, so that it will run automatically when your system is building. This feature was inspired by SB-Grovel, a similar SBCL-specific project. CFFI-Grovel can also be used without ASDF.

13.1 Building FFIs with CFFI-Grovel

CFFI-Grovel uses a specification file (*.lisp) describing the features that need groveling. The C compiler is used to retrieve this data and write a Lisp file (*.cffi.lisp) which contains the necessary CFFI definitions to access the variables, structures, constants, and enums mentioned in the specification.

CFFI-Grovel provides an ASDF component for handling the necessary calls to the C compiler and resulting file management.

13.2 Specification File Syntax

The specification files are read by the normal Lisp reader, so they have syntax very similar to normal Lisp code. In particular, semicolon-comments and reader-macros will work as expected.

There are several forms recognized by CFFI-Grovel:

`progn` **&rest** *forms* [Grovel Form]

Processes a list of forms. Useful for conditionalizing several forms. For example:

```
#+freebsd
(progn
  (constant (ev-enable "EV_ENABLE"))
  (constant (ev-disable "EV_DISABLE")))
```

`include` **&rest** *files* [Grovel Form]

Include the specified files (specified as strings) in the generated C source code.

`in-package` *symbol* [Grovel Form]

Set the package to be used for the final Lisp output.

`ctype` *lisp-name size-designator* [Grovel Form]

Define a CFFI foreign type for the string in *size-designator*, e.g. `(ctype :pid "pid_t")`.

`constant` (*lisp-name* **&rest** *c-names*) **&key** *type documentation optional* [Grovel Form]

Search for the constant named by the first *c-name* string found to be known to the C preprocessor and define it as *lisp-name*.

The *type* keyword argument specifies how to grovel the constant: either `integer` (the default) or `double-float`. If *optional* is true, no error will be raised if all the *c-names* are unknown. If *lisp-name* is a keyword, the actual constant will be a symbol of the same name interned in the current package.

`feature` *lisp-feature-name c-name* **&key** *feature-list* [Grovel Form]

Adds *lisp-feature-name* to the list *feature-list* if the *c-name* string is known to the C preprocessor. *feature-list* defaults to `cl:*features*`.

`define` *name* **&optional** *value* [Grovel Form]

Defines an additional C preprocessor symbol, which is useful for altering the behavior of included system headers.

`cc-flags` **&rest** *flags* [Grovel Form]

Adds *cc-flags* to the command line arguments used for the C compiler invocation.

`pkg-config-cflags` *pkg* **&key** *optional* [Grovel Form]

Adds *pkg* to the command line arguments for the external program `pkg-config` and runs it to retrieve the relevant include flags used for the C compiler invocation. This syntax can be used instead of hard-coding paths using `cc-flags`, and ensures that include flags are added correctly on the build system. Assumes `pkg-config` is installed and working. *pkg* is a string that identifies an installed `pkg-config` package. See the pkg-config manual for more information. If *optional* is true, failure to execute `pkg-config` does *not* abort compilation.

`cstruct` *lisp-name c-name slots* [Grovel Form]

Define a CFFI foreign struct with the slot data specfied. Slots are of the form `(lisp-name c-name &key type count (signed t))`.

`cunion` *lisp-name c-name slots* [Grovel Form]

Identical to `cstruct`, but defines a CFFI foreign union.

`cstruct-and-class` *c-name slots* [Grovel Form]

Defines a CFFI foreign struct, as with `cstruct` and defines a CLOS class to be used with it. This is useful for mapping foreign structures to application-layer code that shouldn't need to worry about memory allocation issues.

`cvar` *namespec type* **&key** *read-only* [Grovel Form]

Defines a foreign variable of the specified type, even if that variable is potentially a C preprocessor pseudo-variable. e.g. `(cvar ("errno" errno) errno-values)`, assuming that errno-values is an enum or equivalent to type `:int`.

The *namespec* is similar to the one used in [defcvar], page 88.

`cenum` *name-and-opts* **&rest** *elements* [Grovel Form]

Defines a true C enum, with elements specified as `((lisp-name &rest c-names) &key optional documentation)`. *name-and-opts* can be either a symbol as name, or a list `(name &key base-type define-constants)`. If *define-constants* is non-null, a Lisp constant will be defined for each enum member.

`constantenum` *name-and-opts* **&rest** *elements* [Grovel Form]

Defines an enumeration of pre-processor constants, with elements specified as `((lisp-name &rest c-names) &key optional documentation)`. *name-and-opts* can be either a symbol as name, or a list `(name &key base-type define-constants)`. If *define-constants* is non-null, a Lisp constant will be defined for each enum member.

This example defines `:af-inet` to represent the value held by `AF_INET` or `PF_INET`, whichever the pre-processor finds first. Similarly for `:af-packet`, but no error will be signalled if the platform supports neither `AF_PACKET` nor `PF_PACKET`.

```
(constantenum address-family
  ((:af-inet "AF_INET" "PF_INET")
   :documentation "IPv4 Protocol family")
  ((:af-local "AF_UNIX" "AF_LOCAL" "PF_UNIX" "PF_LOCAL")
   :documentation "File domain sockets")
  ((:af-inet6 "AF_INET6" "PF_INET6")
   :documentation "IPv6 Protocol family")
  ((:af-packet "AF_PACKET" "PF_PACKET")
   :documentation "Raw packet access"
   :optional t))
```

`bitfield` *name-and-opts* **&rest** *elements* [Grovel Form]

Defines a bitfield, with elements specified as `((lisp-name &rest c-names) &key optional documentation)`. *name-and-opts* can be either a symbol as name, or a list `(name &key base-type)`. For example:

```
(bitfield flags-ctype
  ((:flag-a "FLAG_A")
    :documentation "DOCU_A")
  ((:flag-b "FLAG_B" "FLAG_B_ALT")
    :documentation "DOCU_B")
  ((:flag-c "FLAG_C")
    :documentation "DOCU_C"
    :optional t))
```

13.3 ASDF Integration

An example software project might contain four files; an ASDF file, a package definition file, an implementation file, and a CFFI-Grovel specification file.

The ASDF file defines the system and its dependencies. Notice the use of `eval-when` to ensure CFFI-Grovel is present and the use of `(cffi-grovel:grovel-file name &key cc-flags)` instead of `(:file name)`.

The `example-software.asd` file would look like that:

```
;;; CFFI-Grovel is needed for processing grovel-file components
(defsystem "example-software"
  :defsystem-depends-on ("cffi-grovel")
  :depends-on ("cffi")
  :serial t
  :components
```

```
((:file "package")
 (:cffi-grovel-file "example-grovelling")
 (:cffi-wrapper-file "example-wrappers")
 (:file "example")))
```

The `package.lisp` file would contain one or several `defpackage` forms, to remove circular dependencies and make building the project easier. Note that you may or may not want to `:use` your internal package.

> **Implementor's note:** *Note that it's a not a good idea to* `:use` *when names may clash with, say, CL symbols. Or you could use* `uiop:define-package` *and its* `:mix` *option.*

```
(defpackage #:example-internal
  (:use)
  (:nicknames #:exampleint))

(defpackage #:example-software
  (:export ...)
  (:use #:cl #:cffi #:exampleint))
```

The internal package is created by Lisp code output from the C program written by CFFI-Grovel; if your specification file is `exampleint.lisp`, the `exampleint.cffi.lisp` file will contain the CFFI definitions needed by the rest of your project. See Section 13.2 [Groveller Syntax], page 119.

13.4 Implementation Notes

CFFI-Grovel will generate many files that not only architecture-specific, but also implementation-specific, and should not be distributed. ASDF will generate these files in its output cache; if you build with multiple architectures (e.g. with NFS/AFS home directories) or implementations, it is critical for avoiding clashes to keep this cache in an implementation-dependent directory (as is the default).

For `foo-internal.lisp`, the resulting `foo-internal.c`, `foo-internal`, and `foo-internal.cffi.lisp` are all platform-specific, either because of possible reader-macros in foo-internal.lisp, or because of varying C environments on the host system. For this reason, it is not helpful to distribute any of those files; end users building CFFI-Grovel based software will need `cffi`-Grovel anyway.

> **Implementor's note:** *For now, after some experimentation with* CLISP *having no long-long, it seems appropriate to assert that the generated* `.c` *files are architecture and operating-system dependent, but lisp-implementation independent. This way the same* `.c` *file (and so the same* `.grovel-tmp.lisp` *file) will be shareable between the implementations running on a given system.*

13.5 Wrapper for Inline/Static Functions and Macros

In a shared library, information in static/inlined functions and macros are already removed during the compilation. Wrapper file enables to write an uninlined function wrapping the call to them.

A wrapper file compilation/loading proceeds as follows: Unlike groveller which generates C code that emits lisp files containing cffi definitions, it generates C code, compiles it as a shared library, loads the library, generate the cffi definitions (as lisp code) and then loads the lisp code.

It has asdf integration similar to groveller.

```
(defsystem "example-software"
  :defsystem-depends-on ("cffi-grovel")
  :depends-on ("cffi")
  :serial t
  :components
  ((:file "package")
   (:cffi-grovel-file "example-grovelling")
   (:cffi-wrapper-file "example-wrappers")  ;; <<--- this part
   (:file "example")))
```

`defwrapper` *name-and-options return-type* **&rest** *args* [Wrapper Form]

```
static inline int foo(int i) {
  return 1+i;
};
#define bar(i) (1+(i))
```

```
(in-package :mypackage)
(defwrapper ("foo" foo) :int
  (i :int))
(defwrapper ("bar" bar) :int
  (i :int))
```

Other forms are similar to grovel files.

`progn` **&rest** *forms* [Wrapper Form]

Processes a list of forms. Useful for conditionalizing several forms. For example:

```
#+freebsd
(progn
  (constant (ev-enable "EV_ENABLE"))
  (constant (ev-disable "EV_DISABLE")))
```

`include` **&rest** *files* [Wrapper Form]

Include the specified files (specified as strings) in the generated C source code.

`in-package` *symbol* [Wrapper Form]

Set the package to be used for the final Lisp output.

`flags` **&rest** *flags* [Wrapper Form]

Adds *cc-flags* to the command line arguments used for the C compiler invocation.

`proclaim` **&rest** *proclaimations* [Wrapper Form]

`declaim` **&rest** *declaimations* [Wrapper Form]

14 Static Linking

On recent enough versions of supported implementations (currently, GNU CLISP 2.49 or later, CMUCL 2015-11 or later, and SBCL 1.2.17 or later, except SBCL 2.0.4), and with a recent enough ASDF (3.1.2 or later), you can create a statically linked Lisp executable image (or a standalone application executable) that includes all the C extensions defined via Chapter 13 [The Groveller], page 119's `:cffi-wrapper-file` ASDF components (and any other such objects output by `asdf:compile-op`), as well as those defined by `:c-file` or `:o-file` ASDF components, and your Lisp code. This makes it easier to deliver your code as a single file.

Note that the resulting binary will typically still depend on any shared libraries loaded via See [load-foreign-library], page 110, or See [use-foreign-library], page 113, as well as core libraries such as `libc`.

To dump a statically linked executable image, use:

```
(asdf:load-system :cffi-grovel)
(asdf:operate :static-image-op :example-software)
```

To dump a statically linked executable standalone application, use:

```
(asdf:load-system :cffi-grovel)
(asdf:operate :static-program-op :example-software)
```

See the ASDF manual for documentation about `image-op` and `program-op` which are the parent operation classes that behave similarly except they don't statically link C code.

> **Implementor's note:** *There is also an operation* `:static-runtime-op` *to create the statically linked runtime alone, but it's admittedly not very useful except as an intermediate step dependency towards building* `:static-image-op` *or* `:static-program-op`*.*

15 Limitations

These are CFFI's limitations across all platforms; for information on the warts on particular Lisp implementations, see Chapter 3 [Implementation Support], page 3.

- The tutorial includes a treatment of the primary, intractable limitation of CFFI, or any FFI: that the abstractions commonly used by C are insufficiently expressive. See Section 4.6 [Breaking the abstraction], page 9, for more details.

Appendix A Platform-specific features

Whenever a backend doesn't support one of CFFI's features, a specific symbol is pushed onto `common-lisp:*features*`. The meanings of these symbols follow.

cffi-sys::flat-namespace
: This Lisp has a flat namespace for foreign symbols meaning that you won't be able to load two different libraries with homograph functions and successfully differentiate them through the `:library` option to `defcfun`, `defcvar`, etc...

cffi-sys::no-foreign-funcall
: The macro `foreign-funcall` is **not** available. On such platforms, the only way to call a foreign function is through `defcfun`. See [foreign-funcall], page 94, and [defcfun], page 92.

cffi-sys::no-long-long
: The C `long long` type is **not** available as a foreign type.

However, on such platforms CFFI provides its own implementation of the `long long` type for all of operations in chapters Chapter 6 [Foreign Types], page 22, Chapter 7 [Pointers], page 60, and Chapter 9 [Variables], page 87. The functionality described in Chapter 10 [Functions], page 91, and Chapter 12 [Callbacks], page 114, will not be available.

32-bit Lispworks 5.0+ is an exception. In addition to the CFFI implementation described above, Lispworks itself implements the `long long` type for Chapter 10 [Functions], page 91. Chapter 12 [Callbacks], page 114, are still missing `long long` support, though.

cffi-sys::no-stdcall
: This Lisp doesn't support the `stdcall` calling convention. Note that it only makes sense to support `stdcall` on (32-bit) x86 platforms.

Appendix B Glossary

aggregate type

A CFFI type for C data defined as an organization of data of simple type; in structures and unions, which are themselves aggregate types, they are represented by value.

foreign value

This has two meanings; in any context, only one makes sense.

When using type translators, the foreign value is the lower-level Lisp value derived from the object passed to `translate-to-foreign` (see [translate-to-foreign], page 56). This value should be a Lisp number or a pointer (satisfies `pointerp`), and it can be treated like any general Lisp object; it only completes the transformation to a true foreign value when passed through low-level code in the Lisp implementation, such as the foreign function caller or indirect memory addressing combined with a data move.

In other contexts, this refers to a value accessible by C, but which may only be accessed through CFFI functions. The closest you can get to such a foreign value is through a pointer Lisp object, which itself counts as a foreign value in only the previous sense.

simple type

A CFFI type that is ultimately represented as a builtin type; CFFI only provides extra semantics for Lisp that are invisible to C code or data.

Index

F

G

I

L

M

N

P

R

S

T

U

V

W