Perl in Lisp 0.1

by Stuart Sierra

July 24, 2006

Abstract

This document describes the source code of a Common Lisp interface to the Perl 5 API. It consists two layers: 1. CFFI definitions for the C API of Perl and 2. a Lisp library on top of them that offers convenient entry points to evaluate strings of Perl code, call Perl functions, and convert Perl data types to and from their Common Lisp equivalents.

This is a beta release. Some parts are incomplete, but the overall package is usable.

This documentation was generated with Noweb and LaTeX.

Contents

1 Introduction

This document is a “literate program,” containing both its source code and full documentation of that source code. The Makefile in Section 17.1 produces two output files. The first, `perlapi.lisp`, defines the `perl-api` package, which contains CFFI definitions for the Perl C API. The second, `perl-in.lisp`, defines the `perl-in-lisp` package, which exports Lisp functions that provide convenient ways to use Perl from Common Lisp.

Unit tests for both packages are defined with the Lisp-Unit testing framework.

2 The Perl API

On Unix/Linux, the Perl library is called simply “libperl” and this is sufficient for CFFI to find it. On Windows, I do not know where the Perl DLL file will be located or what it will be called. This code should work fine on Windows, but you will need to alter this chunk to tell CFFI where the Perl DLL file is located.

3a ⟨*Libperl foreign library definition* 3a⟩≡ (52b)

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

(use-foreign-library libperl)
```

Most of the public Perl API is implemented as C preprocessor macros. Obviously, those macros cannot be called through a foreign function interface. There are two possible ways to proceed here. One could write a small library of C code to wrap the API macros in functions, and that's exactly what I did in early versions of this library. This proved tricky to compile and awkward to use. So I decided to dig into the Perl source and find the underlying functions those macros call. Then I can reimplement the macros in Lisp.

3 Perl API Primitive Types

The Perl API defines abbreviations for common C types. They are copied here to make the FFI definitions match the C source. `I32`, `U32`, `IV`, and `UV` are, usually, all 32-bit integers. `I32` is actually 64 bits on Crays. If this code ever gets run on a Cray, I will eat my keyboard.

3b ⟨*Perl API Types* 3b⟩≡ (52b) 3c▷

```
(defctype :i32 :int32)
(defctype :u32 :uint32)
```

A more difficult problem is the width of `IV` (signed integer) and `UV` (unsigned). They are usually 32 bits, but could be 64 bits on some architectures. I do not know how to determine this without crawling through the preprocessed Perl source, so I cheat and assume 32 bits. This is a bad thing and should be fixed.

3c ⟨*Perl API Types* 3b⟩+≡ (52b) ◁3b 3d▷

```
(defctype :iv  :int32)
(defctype :uv  :uint32)
```

`NV` is always a `double`. `PV` is always a `char*`, although Perl PV strings may contain NULL characters and may not be NULL-terminated like proper C strings, so we cannot treat them as CFFI `:string` types.

3d ⟨*Perl API Types* 3b⟩+≡ (52b) ◁3c 4a▷

```
(defctype :nv  :double)
(defctype :pv  :pointer) ; char*
```

STRLEN is a typedef, like the traditional size_t, for an unsigned integer type that can hold the length of the largest string Perl can handle. Again, this can vary by platform, so I cheat and assume 32 bits. Bad me.

4a ⟨*Perl API Types* 3b⟩+≡ (52b) ◁3d 4b▷

```
(defctype :strlen :uint32)
```

4 The Perl Interpreter

We treat the interpreter as an opaque void pointer; there is no need to access its memory directly.

4b ⟨*Perl API Types* 3b⟩+≡ (52b) ◁4a 8a▷

```
(defctype :interpreter :pointer :translate-p nil)
```

4.1 Initializing the Interpreter

There are four Perl API functions necessary to set up the Perl interpreter, perl_alloc, perl_construct, perl_parse, and perl_run. Despite what the perlembed man page says, my tests indicate that the PERL_SYS_INIT3 macro is not actually necessary for running an embedded interpreter.

4c ⟨*CFFI Definitions* 4c⟩≡ (52b) 6a▷

```
(defcfun "perl_alloc" :interpreter)

(defcfun "perl_construct" :void
  (interpreter :interpreter))

(defcfun "perl_parse" :void
  (interpreter :interpreter)
  (xsinit :pointer)
  (argc :int)
  (argv :pointer)
  (env :pointer))

(defcfun "perl_run" :int
  (interpreter :interpreter))
```

Defines:
perl-alloc, used in chunks 4d and 5a.
perl-construct, used in chunks 4d and 5a.
perl-parse, used in chunks 4d and 5a.
perl-run, used in chunks 4d and 5d.

4d ⟨*perl-api Exports* 4d⟩≡ (51a) 6b▷

```
#:perl-alloc #:perl-construct #:perl-parse #:perl-run
```

Uses perl-alloc 4c, perl-construct 4c, perl-parse 4c, and perl-run 4c.

We can wrap up the complete process necessary to initialize the interpreter in a single function. It returns the pointer to the interpreter instance. This pointer will be needed later to destroy the interpreter and free the memory.

5a ⟨*Wrapper Library Internal Functions* 5a⟩≡ (53a) 6e ▷

```
(defun make-interpreter ()
  (let ((interpreter (perl-alloc))
        (arguments (foreign-alloc :pointer :count 3)))
    (perl-construct interpreter)
    ⟨Create Command-Line Argument Array 5c⟩
    (perl-parse interpreter (null-pointer)
                3 arguments (null-pointer))
    ⟨Start Interpreter Running 5d⟩
    interpreter))
```

Defines:
`make-interpreter`, used in chunk 7b.
Uses `perl-alloc` 4c, `perl-construct` 4c, and `perl-parse` 4c.

`perl-parse` receives an array of strings, which in a normal Perl executable would be the command-line arguments. To run an embedded interpreter, we need to pass three arguments: an empty string, `-e`, and `0`. These are similar to the arguments that would be used when calling snippets of Perl code from a shell. Initializing this as a C `char**` array looks like this (copied from the perlembed man page):

5b ⟨*Embedding Command Line Arguments In C* 5b⟩≡

```
char *embedding[] = { "", "-e", "0" };
```

But in CFFI-speak it's a little more complicated. The `let` in the function above allocates a foreign array, `arguments`, of pointers. Then we create the three strings:

5c ⟨*Create Command-Line Argument Array* 5c⟩≡ (5a)

```
(setf (mem-aref arguments :pointer 0) (foreign-string-alloc ""))
(setf (mem-aref arguments :pointer 1) (foreign-string-alloc "-e"))
(setf (mem-aref arguments :pointer 2) (foreign-string-alloc "0"))
```

To start the interpreter, we call `perl_run`, which returns zero on success. Any other return value signals a critical error.

5d ⟨*Start Interpreter Running* 5d⟩≡ (5a)

```
(let ((run (perl-run interpreter)))
  (unless (zerop run)
    (error "perl_run failed (return value: ~A)" run)))
```

Uses `perl-run` 4c.

4.2 Destroying the Perl Interpreter

There are separate Perl API functions to shut down the interpreter, `perl_destruct`, and free its memory, `perl_free`.

6a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁4c 6c▷

```
(defcfun "perl_destruct" :void
  (interpreter :interpreter))

(defcfun "perl_free" :void
  (interpreter :interpreter))
```

Defines:
`perl-destruct`, used in chunk 6.
`perl-free`, used in chunks 6 and 37.

6b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁4d 6d▷

```
#:perl-destruct #:perl-free
```

Uses `perl-destruct` 6a and `perl-free` 6a.

To ensure that these functions really clean out all the memory used by Perl, we have to set the global variable `PL_perl_destruct_level` to one.

6c ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁6a 8d▷

```
(defcvar "PL_perl_destruct_level" :i32)
```

Defines:
`*pl-perl-destruct-level*`, used in chunk 6e.

6d ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁6b 8c▷

```
#:*pl-perl-destruct-level*
```

Uses `perl-destruct` 6a.

We wrap the whole process in a single function, which takes as its argument the pointer returned by `make-interpreter`.

6e ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁5a 7d▷

```
(defun destroy-interpreter (interpreter)
  (setf *pl-perl-destruct-level* 1)
  (perl-destruct interpreter)
  (perl-free interpreter))
```

Defines:
`destroy-interpreter`, used in chunk 7b.
Uses `*pl-perl-destruct-level*` 6c, `perl-destruct` 6a, and `perl-free` 6a.

4.3 Maintaining the Interpreter

We store the pointer to the Perl interpreter instance as a private (non-exported) global variable. Two functions will be exported to start and stop the interpreter. They are safe to call at any time; `start-perl` will do nothing if the interpreter is already running and `stop-perl` will do nothing if the interpreter is *not* running. Both functions explicitly return nothing with `(values)` so that no potentially confusing return values will be printed at the REPL.

7a ⟨*Wrapper Library Globals* 7a⟩≡ (53a) 11d ▷

```
(defvar *perl-interpreter* nil)
```

Defines:
`*perl-interpreter*`, used in chunk 7.

7b ⟨*Wrapper Library Public Functions* 7b⟩≡ (53a) 44a ▷

```
(defun start-perl ()
  (unless *perl-interpreter*
    (setq *perl-interpreter* (make-interpreter)))
  (values))

(defun stop-perl ()
  (when *perl-interpreter*
    (destroy-interpreter *perl-interpreter*)
    (setq *perl-interpreter* nil))
  (values))
```

Defines:
`start-perl`, used in chunk 7.
`stop-perl`, used in chunk 7c.
Uses `*perl-interpreter*` 7a, `destroy-interpreter` 6e, and `make-interpreter` 5a.

7c ⟨*Wrapper Library Exports* 7c⟩≡ (51b) 44b ▷

```
#:start-perl #:stop-perl
```

Uses `start-perl` 7b and `stop-perl` 7b.

To make this code idiot-proof, we will ensure that a Perl interpreter is running before calling any of the API functions. We can define a function, `need-perl`, to be called a the top of every function that needs the interpreter. Since this function will be called very often, we declare it `inline`.

7d ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁ 6e 10d ▷

```
(declaim (inline need-perl))

(defun need-perl ()
  (unless *perl-interpreter* (start-perl)))
```

Defines:
`need-perl`, used in chunks 19b, 25d, 44a, 47b, and 49.
Uses `*perl-interpreter*` 7a and `start-perl` 7b.

5 Perl Scalars

A Perl scalar value (abbreviated SV) can be a number, a string, or a reference. At the API level, i.e. not in Perl source code, it may also contain a pointer to other values, such as arrays and hashes.

8a ⟨*Perl API Types* 3b⟩+≡ (52b) ◁4b 8b▷

```
(defctype :sv :pointer)
```

Defines:
`:sv`, used in chunks 8–10, 12–14, 16–18, 21–24, 26d, 28c, 30, 37a, 39a, 45a, and 46c.

We will usually interact with scalars as opaque pointers, but it may be occasionally useful to have access to parts of their structure, particularly the reference count.

8b ⟨*Perl API Types* 3b⟩+≡ (52b) ◁8a 11a▷

```
(defcstruct sv
  (any :pointer)
  (refcnt :uint32)
  (flags :uint32))
```

Uses `refcnt` 10d.

8c ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁6d 8e▷

```
#:sv #:any #:refcnt #:flags
```

Uses `:sv` 8a and `refcnt` 10d.

5.1 Creating Scalars

`Perl_newSV` creates a generic, empty scalar with the supplied number of bytes of storage space allocated. It sets the scalar's reference count to one, as do all of the "shortcut" functions below.

8d ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁6c 9a▷

```
(defcfun "Perl_newSV" :sv
  (size :strlen))
```

Defines:
`perl-newsv`, used in chunks 8e, 10e, 12, 17c, 19a, 24c, 30f, 36d, and 50a.
Uses `:sv` 8a.

8e ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁8c 9b▷

```
#:perl-newsv
```

Uses `perl-newsv` 8d.

There are “shortcut” functions for creating new scalars with numeric values:

9a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁8d 9c▷

```
(defcfun "Perl_newSViv" :sv
  (int :iv))

(defcfun "Perl_newSVuv" :sv
  (uint :uv))

(defcfun "Perl_newSVnv" :sv
  (double :nv))
```

Defines:
  perl-newsviv, used in chunks 9b, 12, 13c, 15–17, 21c, 22a, 38a, 40b, 41a, and 49.
  perl-newsvnv, used in chunks 9b, 12b, 13c, 15b, and 49.
  perl-newsvuv, used in chunks 9b, 13c, and 49.
Uses :sv 8a.

9b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁8e 9d▷

```
#:perl-newsviv #:perl-newsvuv #:perl-newsvnv
```

Uses perl-newsviv 9a, perl-newsvnv 9a, and perl-newsvuv 9a.

There are two functions for creating scalars from strings. Both take the length of the string as an argument, but Perl_newSVpv will automatically calculate the length if it is given as zero. Perl_newSVpvn, which does not perform this check, is recommended as more efficient.

9c ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁9a 9e▷

```
(defcfun "Perl_newSVpv" :sv
  (string :string)
  (length :strlen)) ; automatically computed if zero

(defcfun "Perl_newSVpvn" :sv
  (string :string)
  (length :strlen)) ; NOT automatically computed
```

Defines:
  perl-newsvpv, used in chunks 9d, 12b, 15b, 23, 46, 47, and 49.
  perl-newsvpvn, used in chunks 9d and 15b.
Uses :sv 8a.

9d ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁9b 10a▷

```
#:perl-newsvpv #:perl-newsvpvn
```

Uses perl-newsvpv 9c and perl-newsvpvn 9c.

To copy existing scalars:

9e ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁9c 10b▷

```
(defcfun "Perl_newSVsv" :sv
  (scalar :sv))
```

Defines:
  perl-newsvsv, used in chunk 10a.
Uses :sv 8a.

10a ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁9d 10c▷

```
#:perl-newsvsv
```

Uses `perl-newsvsv` 9e.

5.2 Scalar Reference Counting

Perl's garbage collection works by reference counting. In Perl code, this is invisible, but when using the C interface we must explicitly increment and decrement the reference counts of the variables we use.

10b ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁9e 12c▷

```
(defcfun "Perl_sv_newref" :sv
  (scalar :sv))

(defcfun "Perl_sv_free" :void
  (scalar :sv))
```

Defines:
`perl-sv-free`, used in chunk 10.
`perl-sv-newref`, used in chunk 10.
Uses `:sv` 8a.

10c ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁10a 11b▷

```
#:perl-sv-newref #:perl-sv-free
```

Uses `perl-sv-free` 10b and `perl-sv-newref` 10b.

`perl-sv-newref` will increment the reference count of the scalar; `perl-sv-free` will decrement the reference count and, if it drops to zero, clear the scalar and deallocate all its memory.

We can also get a scalar's reference count directly from its structure:

10d ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁7d 12a▷

```
(defun refcnt (scalar)
  (foreign-slot-value scalar 'sv 'refcnt))
```

Defines:
`refcnt`, used in chunks 8 and 10e.

Testing the reference count functions.

10e ⟨*Wrapper Library Tests* 10e⟩≡ (53b) 12b▷

```
(define-test refcnts
  (let ((s (perl-newsv 0)))
    (assert-equal 1 (refcnt s))
    (assert-equal 2 (refcnt (perl-sv-newref s)))
    (perl-sv-free s)
    (assert-equal 1 (refcnt s))
    (perl-sv-free s)
    (assert-equal 0 (refcnt s))))
```

Uses `perl-newsv` 8d, `perl-sv-free` 10b, `perl-sv-newref` 10b, and `refcnt` 10d.

Technically the scalar gets deallocated from memory at the next-to-last line of that test, but the structure survives long enough for the final =0 test to pass.

5.3 Determining the Type of Scalars

Since scalars can contain multiple types of values, we need tests to determine what they actually are.

11a ⟨*Perl API Types* 3b⟩+≡ (52b) ◁8b 20a▷

```
(defcenum svtype
  :null ; undef
  :iv   ; Scalar (integer)
  :nv   ; Scalar (double float)
  :rv   ; Scalar (reference)
  :pv   ; Scalar (string)
  :pviv ; a pointer to an IV (used in hashes)
  :pvnv ; a pointer to an NV (used in hashes)
  :pvmg ; blessed or magical scalar
  :pvbm ; ??
  :pvlv ; ??
  :pvav ; Array
  :pvhv ; Hash
  :pvcv ; Code reference
  :pvgv ; typeglob (possibly a file handle)
  :pvfm ; ??
  :pvio ; an I/O handle?
  )
```

Defines:
`svtype`, used in chunks 11b, 12b, and 50b.

11b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁10c 12d▷

```
#:svtype
```

Uses `svtype` 11a 12a.

Type checking of scalars is implemented in the Perl API with macros that directly access bits of the SV structure. Copied from `sv.h` in the Perl source, they are:

11c ⟨*SvTYPE macros in C* 11c⟩≡

```
#define SVTYPEMASK 0xff
#define SvTYPE(sv) ((sv)->sv_flags & SVTYPEMASK)
```

In Lisp, these become:

11d ⟨*Wrapper Library Globals* 7a⟩+≡ (53a) ◁7a

```
(defvar *sv-type-mask* #Xff)
```

Defines:
`*sv-type-mask*`, used in chunk 12a.

12a ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁10d 15a▷

```
(defun svtype (scalar)
  (foreign-enum-keyword
   'svtype
   (logand (foreign-slot-value scalar 'sv 'flags)
           *sv-type-mask*)))
```

Defines:
svtype, used in chunks 11b, 12b, and 50b.
Uses *sv-type-mask* 11d.

Here are tests for the most common types:

12b ⟨*Wrapper Library Tests* 10e⟩+≡ (53b) ◁10e 15b▷

```
(define-test sv-type
  (assert-eq :null (svtype (perl-newsv 0)))
  (assert-eq :iv   (svtype (perl-newsviv 100)))
  (assert-eq :nv   (svtype (perl-newsvnv 3.14d0)))
  (assert-eq :rv   (svtype (perl-newrv (perl-newsv 0))))
  (assert-eq :pv   (svtype (perl-newsvpv "hello" 0)))
  (assert-eq :pvav (svtype (perl-newav)))
  (assert-eq :pvhv (svtype (perl-newhv))))
```

Uses perl-newav 20b, perl-newhv 26b, perl-newrv 30a, perl-newsv 8d, perl-newsviv 9a, perl-newsvnv 9a, perl-newsvpv 9c, and svtype 11a 12a.

5.4 Converting Scalars to C Types

Perl_sv_true returns the boolean value (automatically converted from an integer to t or nil by CFFI's :boolean type) of the scalar by the Perl definition of a boolean. In Perl, the value undef, the number 0, the string "0", and the empty string are all false; anything else is true.

12c ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁10b 13a▷

```
(defcfun "Perl_sv_true" :boolean
  (scalar :sv))
```

Defines:
perl-sv-true, used in chunks 12, 13c, 15b, and 16d.
Uses :sv 8a.

12d ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁11b 13b▷

```
#:perl-sv-true
```

Uses perl-sv-true 12c.

12e ⟨*API Tests* 12e⟩≡ (53b) 13c▷

```
(define-test scalars-true-false
  (assert-true (pointerp (perl-newsv 0)))
  (assert-equal nil (perl-sv-true (perl-newsv 0)))
  (assert-equal t (perl-sv-true (perl-newsviv 5))))
```

Uses perl-newsv 8d, perl-newsviv 9a, and perl-sv-true 12c.

Three functions convert scalars to numeric types. These functions will attempt to coerce the scalar to an IV (signed integer), UV (unsigned integer), or NV (double float), respectively.

13a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁12c 14a▷

```
(defcfun "Perl_sv_2iv" :iv
  (scalar :sv))

(defcfun "Perl_sv_2uv" :uv
  (scalar :sv))

(defcfun "Perl_sv_2nv" :nv
  (scalar :sv))
```

Defines:
  `perl-sv-2iv`, used in chunks 13, 17c, 19a, 21c, 38a, 40b, 41a, 45–47, and 50b.
  `perl-sv-2nv`, used in chunks 13, 19a, and 50b.
  `perl-sv-2uv`, used in chunks 13, 19, and 46.
Uses `:sv` 8a.

13b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁12d 14b▷

```
#:perl-sv-2iv #:perl-sv-2uv #:perl-sv-2nv
```

Uses `perl-sv-2iv` 13a, `perl-sv-2nv` 13a, and `perl-sv-2uv` 13a.

13c ⟨*API Tests* 12e⟩+≡ (53b) ◁12e 16d▷

```
(define-test new-scalar-integers
  (assert-equal t (perl-sv-true (perl-newsviv -256)))
  (assert-equal t (perl-sv-true (perl-newsvuv 17)))
  (assert-equal nil (perl-sv-true (perl-newsviv 0)))
  (assert-equal nil (perl-sv-true (perl-newsvuv 0)))
  (assert-equal -256 (perl-sv-2iv (perl-newsviv -256)))
  (assert-equal 17 (perl-sv-2uv (perl-newsvuv 17))))

(define-test new-scalar-floats
  (assert-equal nil (perl-sv-true (perl-newsvnv 0d0)))
  (assert-equal t (perl-sv-true (perl-newsvnv 3.1459d0)))
  (assert-equal 3.1459d0 (perl-sv-2nv (perl-newsvnv 3.1459d0))))
```

Uses `perl-newsviv` 9a, `perl-newsvnv` 9a, `perl-newsvuv` 9a, `perl-sv-2iv` 13a, `perl-sv-2nv` 13a, `perl-sv-2uv` 13a, and `perl-sv-true` 12c.

In the Perl API documentation, scalars are normally converted to strings with the SvPV macro, which first checks if the scalar is actually storing a string and, if it is, returns a pointer directly to that string. If it is not, it uses Perl_sv_2pv_flags to coerce the scalar into a string.

However, Perl_sv_2pv_flags returns a bogus pointer when called on a scalar which already contains string, so it will not work as a functional substitute for SvPV. Instead, we must use Perl_sv_pvn_force_flags, which works for both string and non-string scalars.

The flags in the name refers to a bitfield argument. I do not know what all of the flags are for; they are simply included here for completeness. SV_GMAGIC is the standard recommended flag for use when converting scalars to strings.

14a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁13a 16b▷

```
(defbitfield sv-flags
  (:immediate-unref 1)
  (:gmagic 2)
  (:cow-drop-pv 4) ; Unused in Perl 5.8.x
  (:utf8-no-encoding 8)
  (:nosteal 16))

(defcfun "Perl_sv_pvn_force_flags" :pointer
  (scalar :sv)
  (length :pointer) ; STRLEN*
  (flags :i32))
```

Defines:
perl-sv-pvn-force-flags, used in chunks 14b and 15a.
sv-flags, used in chunk 14b.
Uses :sv 8a.

14b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁13b 16c▷

```
#:sv-flags #:perl-sv-pvn-force-flags
```

Uses :sv 8a, perl-sv-pvn-force-flags 14a, and sv-flags 14a.

Converting scalars to strings is a complicated by the fact that Perl strings can contain NULL characters, so `foreign-string-to-lisp` must be called with `null-terminated-p` set to `nil`. The length of the string comes from the `length` pointer passed to `Perl_sv_pvn_force_flags`.

Does this leak memory when new strings are created? I'm sure I don't know.

15a ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁12a 17d▷

```
(defun string-from-sv (sv)
  (with-foreign-object (length :strlen)
    (foreign-string-to-lisp
        (perl-sv-pvn-force-flags sv length
                                 (foreign-bitfield-value
                                  'sv-flags '(:gmagic)))
        (mem-ref length :strlen)
        nil))) ; null-teminated-p
```

Defines:
`string-from-sv`, used in chunks 15b, 16a, 19a, 23, 29a, 45, and 50b.
Uses `perl-sv-pvn-force-flags` 14a.

Note that this does not handle UTF8 (Perl's preferred flavor of Unicode) strings. Unicode and Lisp is whole can of worms that I don't want to deal with yet.

Some tests for strings:

15b ⟨*Wrapper Library Tests* 10e⟩+≡ (53b) ◁12b 16a▷

```
(define-test new-scalar-strings-of-numbers
  (assert-equal nil (perl-sv-true (perl-newsvpv "0" 0)))
  (assert-equal "0" (string-from-sv (perl-newsviv 0)))
  (assert-equal "-256" (string-from-sv (perl-newsviv -256)))
  (assert-equal "3.14" (string-from-sv (perl-newsvnv 3.14d0))))

(define-test new-scalar-strings-to-booleans
  (assert-equal t (perl-sv-true (perl-newsvpv "foo" 0)))
  (assert-equal t (perl-sv-true (perl-newsvpvn "foo" 3)))
  (assert-equal t (perl-sv-true (perl-newsvpv "1" 0)))
  (assert-equal nil (perl-sv-true (perl-newsvpv "" 0)))
  (assert-equal nil (perl-sv-true (perl-newsvpv "0" 0))))

(define-test new-scalar-strings
  (assert-true (pointerp (perl-newsvpv "hello" 0)))
  (assert-equal "hello" (string-from-sv (perl-newsvpv "hello" 0)))
  (assert-equal "good" (string-from-sv (perl-newsvpvn "goodbye" 4))))
```

Uses `perl-newsviv` 9a, `perl-newsvnv` 9a, `perl-newsvpv` 9c, `perl-newsvpvn` 9c, `perl-sv-true` 12c, and `string-from-sv` 15a.

We can also test that we can use strings containing NULL characters, which are allowed in both Perl and Lisp but not in C. The Perl character-escape syntax `\00` will insert a NULL character in an interpolated string. (We have to escape the backslash to insert it in a Lisp string, giving us `\\00`.) We can create the equivalent string in Lisp by treating a string like an array and modifying one character.

16a ⟨*Wrapper Library Tests* 10e⟩+≡ (53b) ◁15b 19a▷

```
(define-test string-containing-null
  (assert-equal (let ((string (copy-seq "abcde")))
                  (setf (aref string 2) (code-char 0))
                  string) ; "ab" + NULL + "de"
                 (string-from-sv (perl-eval-pv "qq{ab\\00de}" 0))))
```

Uses `perl-eval-pv` 45a and `string-from-sv` 15a.

Lastly, we can access a named scalar with `Perl_get_sv`. If the named scalar does not exist and `create` is true, a new scalar will be created.

16b ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁14a 17a▷

```
(defcfun "Perl_get_sv" :sv
  (name :string)
  (create :boolean))
```

Defines:
`perl-get-sv`, used in chunks 16 and 19b.
Uses `:sv` 8a.

16c ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁14b 17b▷

```
#:perl-get-sv
```

Uses `perl-get-sv` 16b.

16d ⟨*API Tests* 12e⟩+≡ (53b) ◁13c 17c▷

```
(define-test create-named-scalars
  (let ((x (perl-get-sv "foo" t)))
    (perl-sv-setsv-flags x (perl-newsviv 1)
                         (foreign-bitfield-value
                          'sv-flags '(:gmagic)))
    (assert-equal t (perl-sv-true
                     (perl-get-sv "foo" nil)))))
```

Uses `perl-get-sv` 16b, `perl-newsviv` 9a, `perl-sv-setsv-flags` 17a, and `perl-sv-true` 12c.

5.5 Setting the Value of Scalars

The standard API function for copying the value of one scalar to another scalar is Perl_sv_setsv_flags. The flags argument is sv-flags, defined above. The recommended standard flag is SV_GMAGIC.

17a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁16b 18a▷

```
(defcfun "Perl_sv_setsv_flags" :void
  (destination :sv)
  (source :sv)
  (flags :i32))
```

Defines:
perl-sv-setsv-flags, used in chunks 16 and 17.
Uses :sv 8a.

17b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁16c 18b▷

```
#:perl-sv-setsv-flags
```

Uses perl-sv-setsv-flags 17a.

17c ⟨*API Tests* 12e⟩+≡ (53b) ◁16d 21c▷

```
(define-test sv-setsv
  (let ((x (perl-newsv 0))
        (y (perl-newsviv 55)))
    (perl-sv-setsv-flags x y (foreign-bitfield-value
                              'sv-flags '(:gmagic)))
    (assert-equal 55 (perl-sv-2iv x))))
```

Uses perl-newsv 8d, perl-newsviv 9a, perl-sv-2iv 13a, and perl-sv-setsv-flags 17a.

We can abstract away the foreign bitfield:

17d ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁15a 19b▷

```
(defun set-sv (destination source &rest flags)
  (perl-sv-setsv-flags destination source
                       (foreign-bitfield-value
                        'sv-flags flags)))
```

Defines:
set-sv, never used.
Uses perl-sv-setsv-flags 17a.

There are also shortcut functions for C types. The _mg suffix means that these functions correctly handle 'set' magic (i.e. tied variables).

18a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁17a 20b▷

```
(defcfun "Perl_sv_setiv_mg" :void
  (destination :sv)
  (source :iv))
(defcfun "Perl_sv_setuv_mg" :void
  (destination :sv)
  (source :uv))
(defcfun "Perl_sv_setnv_mg" :void
  (destination :sv)
  (source :nv))
(defcfun "Perl_sv_setpv_mg" :void
  (destination :sv)
  (source :string)
  (length :strlen)) ; automatically calculated if 0
(defcfun "Perl_sv_setpvn_mg" :void
  (destination :sv)
  (source :string)
  (length :strlen)) ; NOT automatically calculated
```

Defines:
perl-sv-setiv-mg, used in chunks 18b and 19a.
perl-sv-setnv-mg, used in chunks 18b and 19a.
perl-sv-setpv-mg, used in chunks 18b and 19a.
perl-sv-setpvn-mg, used in chunks 18b and 19a.
perl-sv-setuv-mg, used in chunks 18b and 19a.
Uses :sv 8a.

18b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁17b 20c▷

```
#:perl-sv-setiv-mg #:perl-sv-setuv-mg #:perl-sv-setnv-mg
#:perl-sv-setpv-mg #:perl-sv-setpvn-mg
```

Uses perl-sv-setiv-mg 18a, perl-sv-setnv-mg 18a, perl-sv-setpv-mg 18a, perl-sv-setpvn-mg 18a, and perl-sv-setuv-mg 18a.

19a ⟨*Wrapper Library Tests* 10e⟩+≡ (53b) ◁16a 19c▷

```
(define-test sv-set-iv-uv-nv-pv
  (let ((x (perl-newsv 0)))
    (perl-sv-setiv-mg x -256)
    (assert-equal -256 (perl-sv-2iv x))
    (perl-sv-setuv-mg x 88)
    (assert-equal 88 (perl-sv-2uv x))
    (perl-sv-setnv-mg x 3.1459d0)
    (assert-equal 3.1459d0 (perl-sv-2nv x))
    (perl-sv-setpv-mg x "hello" 0)
    (assert-equal "hello" (string-from-sv x))
    (perl-sv-setpvn-mg x "goodbye" 4)
    (assert-equal "good" (string-from-sv x))))
```

Uses `perl-newsv` 8d, `perl-sv-2iv` 13a, `perl-sv-2nv` 13a, `perl-sv-2uv` 13a, `perl-sv-setiv-mg` 18a, `perl-sv-setnv-mg` 18a, `perl-sv-setpv-mg` 18a, `perl-sv-setpvn-mg` 18a, `perl-sv-setuv-mg` 18a, and `string-from-sv` 15a.

5.6 Accessing Scalars By Name

We can also access and/or create a scalar variable by its name. If the named variable does not exist, it will be automatically created. I am not allowing the passing of any value other than `t` to the `create` argument of `perl-get-sv` because it is not obvious what we should do when the given name does not exist: signal an error or return a null pointer? To keep it simple I use a form that will always succeed.

19b ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁17d 20f▷

```
(defun get-scalar-by-name (name)
  (need-perl)
  (perl-get-sv name t))
```

Defines:
`get-scalar-by-name`, used in chunk 19c.
Uses `need-perl` 7d and `perl-get-sv` 16b.

For reasons I cannot divine, a variable must be created with `perl-get-sv` before being used in an `eval` context in order to be later accessed without causing a memory fault. I think it has something to do with the Perl garbage collector.

19c ⟨*Wrapper Library Tests* 10e⟩+≡ (53b) ◁19a 38d▷

```
(define-test get-scalar-by-name
  (let ((x (get-scalar-by-name "nines")))
    (declare (ignore x))
    (perl-eval-pv "$nines = 999;" nil)
    (assert-equal
     999
     (perl-sv-2uv
      (get-scalar-by-name "nines")))))
```

Uses `get-scalar-by-name` 19b, `perl-eval-pv` 45a, and `perl-sv-2uv` 13a.

6 Perl Arrays

A Perl array (type `AV*`) is actually an "upgraded" scalar that points to a C array of other scalars.

20a ⟨*Perl API Types* 3b⟩+≡ (52b) ◁11a 26a▷

```
(defctype :av :pointer)
```

Creating a new array:

20b ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁18a 20d▷

```
(defcfun "Perl_newAV" :av)
```

Defines:
`perl-newav`, used in chunks 12b, 20–24, 30g, and 49.

20c ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁18b 20e▷

```
#:perl-newav
```

Uses `perl-newav` 20b.

To get the SV at a given index `key` in an array:

20d ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁20b 21a▷

```
(defcfun "Perl_av_fetch" :pointer
  (array :av)
  (key :i32)
  (create :boolean))
```

Defines:
`perl-av-fetch`, used in chunk 20.

20e ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁20c 21b▷

```
#:perl-av-fetch
```

Uses `perl-av-fetch` 20d.

If `create` is true, then the function will grow the array to include the given index. `Perl_av_fetch` has a return type of `SV**`. We must derefernce the pointer to get at the regular `SV*`, but first we have to check that it is not NULL. A wrapper function takes care of this, and returns `nil` if the pointer was NULL.

20f ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁19b 25d▷

```
(defun av-fetch-sv (array key create)
  (let ((ptr (perl-av-fetch array key create)))
    (if (null-pointer-p ptr) nil
      (mem-ref ptr :pointer))))
```

Defines:
`av-fetch-sv`, used in chunks 21c, 22a, and 25d.
Uses `perl-av-fetch` 20d.

To store a value in an array:

21a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁20d 21d▷

```
(defcfun "Perl_av_store" :pointer
  (array :av)
  (key :i32)
  (scalar :sv))
```

Defines:
`perl-av-store`, used in chunks 21 and 22a.
Uses `:sv` 8a.

21b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁20e 21e▷

```
#:perl-av-store
```

Uses `perl-av-store` 21a.

From `av.c`: "The return value will be NULL if the operation failed or if the value did not need to be actually stored within the array (as in the case of tied arrays). Otherwise it can be dereferenced to get the original `SV*`. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL."

I'm not going to write a wrapper function here, because whether or not we actually want to decrement the reference count depends on the reason we're creating the array in the first place—if we're initializing a new array with newly-created scalars, there's no reason to increment the reference count on the scalars before storing them in the array.

21c ⟨*API Tests* 12e⟩+≡ (53b) ◁17c 22a▷

```
(define-test array-store-fetch
  (let ((a (perl-newav)))
    (perl-av-store a 3 (perl-newsviv -17))
    (assert-equal -17 (perl-sv-2iv (perl-in-lisp::av-fetch-sv a 3 nil)))))
```

Uses `av-fetch-sv` 20f, `perl-av-store` 21a, `perl-newav` 20b, `perl-newsviv` 9a, and `perl-sv-2iv` 13a.

To empty an array (does not free the memory):

21d ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁21a 22b▷

```
(defcfun "Perl_av_clear" :void
  (array :av))
```

Defines:
`perl-av-clear`, used in chunks 21e and 22a.

21e ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁21b 22c▷

```
#:perl-av-clear
```

Uses `perl-av-clear` 21d.

22a ⟨*API Tests* 12e⟩+≡ (53b) ◁21c 23a▷

```
(define-test av-clear
  (let ((a (perl-newav)))
    (perl-av-store a 0 (perl-newsviv 34))
    (perl-av-clear a)
    (assert-equal nil (perl-in-lisp::av-fetch-sv a 0 nil))))
```

Uses `av-fetch-sv` 20f, `perl-av-clear` 21d, `perl-av-store` 21a, `perl-newav` 20b, and `perl-newsviv` 9a.

To undefine an array and free its memory;

22b ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁21d 22d▷

```
(defcfun "Perl_av_undef" :void
  (array :av))
```

Defines:
`perl-av-undef`, used in chunk 22c.

22c ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁21e 22e▷

```
#:perl-av-undef
```

Uses `perl-av-undef` 22b.

To push a scalar onto the end of the array, automatically enlarging it if necessary:

22d ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁22b 22f▷

```
(defcfun "Perl_av_push" :void
  (array :av)
  (scalar :sv))
```

Defines:
`perl-av-push`, used in chunks 22–24 and 49.
Uses `:sv` 8a.

22e ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁22c 22g▷

```
#:perl-av-push
```

Uses `perl-av-push` 22d.

And pop a scalar off the end:

22f ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁22d 23b▷

```
(defcfun "Perl_av_pop" :sv
  (array :av))
```

Defines:
`perl-av-pop`, used in chunks 22g and 23a.
Uses `:sv` 8a.

22g ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁22e 23c▷

```
#:perl-av-pop
```

Uses `perl-av-pop` 22f.

23a ⟨*API Tests* 12e⟩+≡ (53b) ◁22a 23d▷

```
(define-test av-push-pop
  (let ((a (perl-newav)))
    (perl-av-push a (perl-newsvpv "a" 0))
    (perl-av-push a (perl-newsvpv "b" 0))
    (assert-equal "b" (perl-in-lisp::string-from-sv (perl-av-pop a)))
    (assert-equal "a" (perl-in-lisp::string-from-sv (perl-av-pop a)))))
```

Uses `perl-av-pop` 22f, `perl-av-push` 22d, `perl-newav` 20b, `perl-newsvpv` 9c, and `string-from-sv` 15a.

To “unshift” an array, i.e. to add `undef` values to the beginning of the array:

23b ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁22f 23e▷

```
(defcfun "Perl_av_unshift" :void
  (array :av)
  (count :i32))
```

Defines:
`perl-av-unshift`, used in chunk 23.

23c ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁22g 23f▷

```
#:perl-av-unshift
```

Uses `perl-av-unshift` 23b.

23d ⟨*API Tests* 12e⟩+≡ (53b) ◁23a 23g▷

```
(define-test av-unshift
  (let ((a (perl-newav)))
    (perl-av-unshift a 10)
    (assert-equal 9 (perl-av-len a))))
```

Uses `perl-av-len` 24a, `perl-av-unshift` 23b, and `perl-newav` 20b.

To shift an SV off the beginning of the array:

23e ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁23b 24a▷

```
(defcfun "Perl_av_shift" :sv
  (array :av))
```

Defines:
`perl-av-shift`, used in chunk 23.
Uses `:sv` 8a.

23f ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁23c 24b▷

```
#:perl-av-shift
```

Uses `perl-av-shift` 23e.

23g ⟨*API Tests* 12e⟩+≡ (53b) ◁23d 24c▷

```
(define-test av-shift
  (let ((a (perl-newav)))
    (perl-av-push a (perl-newsvpv "a" 0))
    (perl-av-push a (perl-newsvpv "b" 0))
    (assert-equal "a" (perl-in-lisp::string-from-sv (perl-av-shift a)))
    (assert-equal "b" (perl-in-lisp::string-from-sv (perl-av-shift a )))))
```

Uses `perl-av-push` 22d, `perl-av-shift` 23e, `perl-newav` 20b, `perl-newsvpv` 9c, and `string-from-sv` 15a.

To get the highest index of the array, or -1 if the array is empty:

24a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁23e 24d▷

```
(defcfun "Perl_av_len" :i32
  (array :av))
```

Defines:
`perl-av-len`, used in chunks 23–25.

24b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁23f 24e▷

```
#:perl-av-len
```

Uses `perl-av-len` 24a.

24c ⟨*API Tests* 12e⟩+≡ (53b) ◁23g 24f▷

```
(define-test av-len
  (let ((a (perl-newav)))
    (perl-av-push a (perl-newsv 0))
    (perl-av-push a (perl-newsv 0))
    (perl-av-push a (perl-newsv 0))
    (assert-equal 2 (perl-av-len a))))
```

Uses `perl-av-len` 24a, `perl-av-push` 22d, `perl-newav` 20b, and `perl-newsv` 8d.

To ensure that an array contains elements indexed at least up to `fill`:

24d ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁24a 24g▷

```
(defcfun "Perl_av_fill" :void
  (array :av)
  (fill :i32))
```

Defines:
`perl-av-fill`, used in chunk 24.

24e ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁24b 25a▷

```
#:perl-av-fill
```

Uses `perl-av-fill` 24d.

24f ⟨*API Tests* 12e⟩+≡ (53b) ◁24c 30f▷

```
(define-test av-fill
  (let ((a (perl-newav)))
    (perl-av-fill a 3)
    (assert-equal 3 (perl-av-len a))))
```

Uses `perl-av-fill` 24d, `perl-av-len` 24a, and `perl-newav` 20b.

To delete the element at index `key` from an array:

24g ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁24d 25b▷

```
(defcfun "Perl_av_delete" :sv
  (array :av)
  (key :i32)
  (flags :i32))
```

Defines:
`perl-av-delete`, used in chunk 25a.
Uses `:sv` 8a.

25a ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁24e 25c▷

```
#:perl-av-delete
```

Uses `perl-av-delete` 24g.

`flags` may be `:discard` from `perl-call-flags`, in which case the SV element is freed and NULL is returned.

To test if an element at index `key` has been initialized:

25b ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁24g 26b▷

```
(defcfun "Perl_av_exists" :boolean
  (array :av)
  (key :i32))
```

Defines:
`perl-av-exists`, used in chunk 25c.

25c ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁25a 26c▷

```
#:perl-av-exists
```

Uses `perl-av-exists` 25b.

We can make `perl-aref` behave like Lisp's `aref`. Given an array and an index into that array, return the scalar at that index. This will not correctly handle references or arrays of arrays. Perl's array access function can potentially return a NULL pointer, which gets translated to `nil` by `av-fetch-sv`.

25d ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁20f 25e▷

```
(defun perl-aref (array index)
  (need-perl)
  (lisp-from-perl (av-fetch-sv array index t)))
```

Defines:
`perl-aref`, used in chunk 25e.
`perl-array`, never used.
Uses `av-fetch-sv` 20f, `lisp-from-perl` 50b, and `need-perl` 7d.

The scalar returned is the same scalar as the one stored in the array, not a copy. So the normal scalar setting functions will work on the return value of `perl-aref`, and we do not need another function to store a value in an array.

Finally, to convert a complete Perl array into the equivalent Lisp list, we have:

25e ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁25d 29a▷

```
(defun list-from-av (av)
  (loop for i from 0 upto (perl-av-len av)
        collecting (perl-aref av i)))
```

Defines:
`list-from-av`, used in chunk 50b.
Uses `perl-aref` 25d and `perl-av-len` 24a.

7 Perl Hash Tables

A Perl hash table, noted in code as `%name`, always uses strings as keys.

Creating a new hash:

26a ⟨*Perl API Types* 3b⟩+≡ (52b) ◁20a 28b▷
```
  (defctype :hv :pointer)
```

26b ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁25b 26d▷
```
  (defcfun "Perl_newHV" :hv)
```
Defines:
  `perl-newhv`, used in chunks 12b, 26c, and 29b.

26c ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁25c 26e▷
```
  #:perl-newhv
```
Uses `perl-newhv` 26b.

Storing a value in it:

26d ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁26b 26f▷
```
  (defcfun "Perl_hv_store" :pointer ; SV**
    (hash :hv)
    (key :string)
    (key-length :u32)
    (value :sv)
    (precomputed-hash-value :u32))
```
Defines:
  `perl-hv-store`, used in chunks 26e and 29b.
Uses `:sv` 8a.

26e ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁26c 26g▷
```
  #:perl-hv-store
```
Uses `perl-hv-store` 26d.

`key-length` *must* be given as it is not automatically calculated. `value`'s reference count is *not* automatically incremented.

Retrieving a value:

26f ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁26d 27a▷
```
  (defcfun "Perl_hv_fetch" :pointer ; SV**
    (hash :hv)
    (key :string)
    (key-length :u32)
    (lvalue :i32))
```
Defines:
  `perl-hv-fetch`, used in chunk 26g.

26g ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁26e 27b▷
```
  #:perl-hv-fetch
```
Uses `perl-hv-fetch` 26f.

To check if a hash table entry exists:

27a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁26f 27c▷

```
(defcfun "Perl_hv_exists" :boolean
  (hash :hv)
  (key :string)
  (key-length :u32))
```

Defines:
`perl-hv-exists`, used in chunk 27b.

27b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁26g 27d▷

```
#:perl-hv-exists
```

Uses `perl-hv-exists` 27a.

To delete the entry:

27c ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁27a 27e▷

```
(defcfun "Perl_hv_delete" :pointer ; SV**
  (hash :hv)
  (key :string)
  (key-length :u32)
  (flags :i32))
```

Defines:
`perl-hv-delete`, used in chunk 27d.

27d ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁27b 27f▷

```
#:perl-hv-delete
```

Uses `perl-hv-delete` 27c.

If `flags` does *not* include `:discard` from `perl-call-flags` then `hv_delete` will create and return a mortal copy of the deleted value.

To delete all the entries in a hash without deleting the hash itself:

27e ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁27c 27g▷

```
(defcfun "Perl_hv_clear" :void
  (hash :hv))
```

Defines:
`perl-hv-clear`, used in chunk 27f.

27f ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁27d 28a▷

```
#:perl-hv-clear
```

Uses `perl-hv-clear` 27e.

And to delete both the entries and the hash itself:

27g ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁27e 28c▷

```
(defcfun "Perl_hv_undef" :void
  (hash :hv))
```

Defines:
`perl-hv-undef`, used in chunk 28a.

28a ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁27f 28d▷

```
#:perl-hv-undef
```

Uses `perl-hv-undef` 27g.

Other functions return complete key/value hash structures, or allow SVs to be used as keys, but I believe these are unnecessary for this implementation.

To copy hash tables into Lisp data structures, we will need to be able to iterate over them. The Perl API provides the following routines for this purpose.

28b ⟨*Perl API Types* 3b⟩+≡ (52b) ◁26a 30c▷

```
(defctype :he :pointer
  :documentation "An entry in a Perl hash table")
```

28c ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁27g 30a▷

```
;; initialize an iterator for the hash
(defcfun "Perl_hv_iterinit" :i32 ; returns # of hash entries
  (hash :hv))

;; advance to the next hash entry
(defcfun "Perl_hv_iternext" :he
  (hash :hv))

;; get the key of the hash entry
(defcfun "Perl_hv_iterkey" :pointer ; char*, may contain NULL
  (hash-entry :he)
  (key-length :pointer)) ; I32*, length of the char*

;; same as above but creates new mortal SV to hold the key
(defcfun "Perl_hv_iterkeysv" :sv
  (hash-entry :he))

;; get the value of the hash entry
(defcfun "Perl_hv_iterval" :sv
  (hash :hv)
  (hash-entry :he))
```

Defines:
`perl-hv-iterinit`, used in chunks 28d and 29a.
`perl-hv-iterkey`, used in chunk 28d.
`perl-hv-iterkeysv`, used in chunks 28d and 29a.
`perl-hv-iternext`, used in chunks 28d and 29a.
`perl-hv-iterval`, used in chunks 28d and 29a.
Uses `:sv` 8a.

28d ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁28a 30b▷

```
#:perl-hv-iterinit #:perl-hv-iternext #:perl-hv-iterkey
#:perl-hv-iterkeysv #:perl-hv-iterval
```

Uses `perl-hv-iterinit` 28c, `perl-hv-iterkey` 28c, `perl-hv-iterkeysv` 28c, `perl-hv-iternext` 28c, and `perl-hv-iterval` 28c.

With these four functions, we can make a function to convert a Perl hash table to a Lisp hash table.

29a ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁25e 29b▷

```
(defun hash-from-hv (perl-hash)
  (perl-scope ;; because SVs may be mortal copies
   (let ((lisp-hash (make-hash-table :test #'equal))
         (size (perl-hv-iterinit perl-hash)))
     (loop repeat size
           do (let ((entry (perl-hv-iternext perl-hash)))
                (setf (gethash (string-from-sv ; does not work w/ lisp-from-perl, why?
                                (perl-hv-iterkeysv entry))
                               lisp-hash)
                      (lisp-from-perl
                       (perl-hv-iterval perl-hash entry)))))
     lisp-hash)))
```

Defines:
  `hash-from-hv`, used in chunk 50b.
Uses `lisp-from-perl` 50b, `perl-hv-iterinit` 28c, `perl-hv-iterkeysv` 28c, `perl-hv-iternext` 28c, `perl-hv-iterval` 28c, `perl-scope` 37c, and `string-from-sv` 15a.

And convert a Lisp hash table to a Perl hash table.

29b ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁29a 30e▷

```
(defun hv-from-hash (lisp-hash)
  (let ((perl-hash (perl-newhv)))
    (maphash #'(lambda (key value)
                 (let ((string-key (string key)))
                   (with-foreign-string
                    (cstring string-key)
                    (perl-hv-store perl-hash
                                   cstring
                                   (length string-key)
                                   (perl-from-lisp value)
                                   0))))
             lisp-hash)
    perl-hash))
```

Defines:
  `hv-from-hash`, used in chunk 49.
Uses `perl-from-lisp` 49, `perl-hv-store` 26d, and `perl-newhv` 26b.

8 Perl References

A Perl reference is a scalar that points to something—anything—else. References are created with newRV, which increments the reference count of the source object, and newRV_noinc, which does not.

30a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁28c 33b▷

```
(defcfun "Perl_newRV" :sv
  (thing :sv))

(defcfun "Perl_newRV_noinc" :sv
  (thing :sv))
```

Defines:
  perl-newrv, used in chunks 12b and 30.
  perl-newrv-noinc, used in chunk 30b.
Uses :sv 8a.

30b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁28d 30d▷

```
#:perl-newrv #:perl-newrv-noinc
```

Uses perl-newrv 30a and perl-newrv-noinc 30a.

The Perl API dereferences with a macro, so we have to do it by digging into the RV/SV structure.

30c ⟨*Perl API Types* 3b⟩+≡ (52b) ◁28b

```
(defcstruct xrv
  (rv :sv))
```

Uses :sv 8a.

30d ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁30b 33c▷

```
#:xrv #:rv
```

30e ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁29b 34d▷

```
(defun deref-rv (ref)
  (foreign-slot-value (foreign-slot-value ref 'sv 'any)
                      'xrv 'rv))
```

Defines:
  deref-rv, used in chunks 30 and 50b.

30f ⟨*API Tests* 12e⟩+≡ (53b) ◁24f 30g▷

```
(define-test references
  (let ((s (perl-newsv 0)))
    (assert-equality #'pointer-eq s
                     (deref-rv (perl-newrv s)))))
```

Uses deref-rv 30e, perl-newrv 30a, and perl-newsv 8d.

30g ⟨*API Tests* 12e⟩+≡ (53b) ◁30f 36d▷

```
(define-test reference-to-array
  (let ((a (perl-newav)))
    (assert-equality #'pointer-eq a
                     (deref-rv (perl-newrv a)))))
```

Uses deref-rv 30e, perl-newav 20b, and perl-newrv 30a.

9 Manipulating the Perl Stack

It's a dirty job, but we have to do it. The only way to pass arguments to and return values from functions is to manipulate the Perl stack directly. Ugh. Here goes.

9.1 A Digression on Pointers

In order to manipulate the Perl stack, we need to modify the values of several global pointers—not the values the they point to, but the addresses stored in the pointers themselves. However, CFFI's `:pointer` type is immutable; once a foreign variable is `defcvar`ed as a `:pointer`, one cannot modify the address it contains.

However, foreign variables declared as integers *can* be modified under CFFI, and in C, pointers are just integers with some additional type information. The only question is which integer type to use, since pointers can be different sizes on different platforms. I cannot cheat here; this has to be correct. Fortunately, CFFI already knows the answer.

31a ⟨*Determine Pointer Size* 31a⟩≡ (33a)

```
(defvar *pointer-size*
  (foreign-type-size :pointer)
  "The size of a pointer on the current platform, in bytes.")
```

Defines:
`*pointer-size*`, used in chunks 31b and 32a.

On this basis, we can create a new "pointer-as-integer" type. I will name this new type `:address`. (This may need to be wrapped in an `eval-when`.)

31b ⟨*:address Type* 31b⟩≡ (33a)

```
(ecase *pointer-size*
  (1 (defctype :address :uint8))   ; unlikely
  (2 (defctype :address :uint16))  ; possible
  (4 (defctype :address :uint32))  ; most common
  (8 (defctype :address :uint64))) ; possible
```

Defines:
`:address`, used in chunks 33–36.
Uses `*pointer-size*` 31a.

Since :address is just another name for an integer, CFFI's definitions for setf and its ilk will work correctly. Arithmetic between addresses will also work correctly.

C's ++ and -- operators increment and decrement pointers by the correct number of bytes, so stack operations can be succintly written as *++stack=object or similar. But Lisp's incf will always increment by one unless given a different value. To avoid mistakes, I will define two macros to increment and decrement an address by the size of a pointer.

32a ⟨*Macros for Using :address* 32a⟩≡ (33a) 32b▷

```
(defmacro address-incf (address &optional (n 1))
  `(incf ,address (* ,n *pointer-size*)))

(defmacro address-decf (address &optional (n 1))
  `(decf ,address (* ,n *pointer-size*)))
```

Defines:
address-decf, used in chunks 33a and 36c.
address-incf, used in chunks 33a, 34d, and 36a.
Uses *pointer-size* 31a.

Lastly, we will need to access the value the address points to. We can create an abbreviation that fills the same role as C's * operator.

32b ⟨*Macros for Using :address* 32a⟩+≡ (33a) ◁32a

```
(defmacro address-ref (address type)
  `(mem-ref (make-pointer ,address) ,type))
```

Defines:
address-ref, used in chunks 33a, 34d, and 36.

Since this is a macro, CFFI's `(setf (mem-ref ...))` magic still works. This allows stack operations such as:

`(setf (address-ref (address-incf x) ...))`

which is equivalent to `*++x = ...` in C.

I will wrap this functionality in its own package, since it is not directly related to Perl.

33a ⟨*address.lisp* 33a⟩≡

```
;;; address.lisp -- CFFI extension to allow mutable pointers

⟨License Header 59⟩

(in-package :common-lisp-user)

(defpackage :cffi-address
  (:use :common-lisp :cffi)
  (:export #:address-incf #:address-decf #:address-ref))

(in-package :cffi-address)

⟨Determine Pointer Size 31a⟩
⟨:address Type 31b⟩
⟨Macros for Using :address 32a⟩
```

Uses `:address` 31b, `address-decf` 32a, `address-incf` 32a, and `address-ref` 32b.

9.2 The Stack Pointer

The Perl API uses a series of macros to create and manipulate a local copy of the stack pointer stored in the global variable `PL_stack_sp`, of type `SV**`. The local pointer is declared and given its initial value with the `dSP` macro.

The only reason I can see for the local copy is so that it can be optimized with the C `register` keyword. For our purposes, it will be simpler to manipulate the global variable directly.

33b ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁30a 34b▷

```
(defcvar "PL_stack_sp" :address)
```

Defines:
  `*pl-stack-sp*`, used in chunks 34–36.
Uses `:address` 31b.

33c ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁30d 34c▷

```
#:*pl-stack-sp*
```

Then we can translate the PUSHMARK and PUTBACK macros, used to keep track of the number of parameters being pushed onto the stack in a function call. I don't claim to understand exactly what these two macros do; I'm just transcribing their definitions into Lisp. Thep PL_* variables have different definitions in different parts of the Perl API source. They are macros, but they appear as symbols in the object code, so I am hoping it is possible to use them directly.

Here is the C definition of PUSHMARK, from pp.h in the Perl source:

34a ⟨PUSHMARK *macro in C* 34a⟩≡

```
#define PUSHMARK(p) if (++PL_markstack_ptr == PL_markstack_max) \
                           markstack_grow();                        \
                        *PL_markstack_ptr = (p) - PL_stack_base
```

Translated to Lisp, using global variables, this becomes:

34b ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁33b 35b▷

```
(defcvar "PL_markstack_ptr" :address)  ; *pl-markstack-ptr*
(defcvar "PL_markstack_max" :address)  ; *pl-markstack-max*
(defcvar "PL_stack_base" :address)     ; *pl-stack-base*
(defcfun "Perl_markstack_grow" :void)  ; (perl-markstack-grow)
```

Uses :address 31b.

34c ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁33c 35c▷

```
#:*pl-markstack-ptr* #:*pl-markstack-max* #:*pl-stack-base* #:perl-markstack-grow
```

34d ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁30e 35d▷

```
(defun pushmark ()
  (when (= (address-incf *pl-markstack-ptr*) *pl-markstack-max*)
    (perl-markstack-grow))
  (setf (address-ref *pl-markstack-ptr* :address)
        (- *pl-stack-sp* *pl-stack-base*)))
```

Defines:
pushmark, used in chunks 40b, 41a, and 44a.
Uses *pl-stack-sp* 33b, :address 31b, address-incf 32a, and address-ref 32b.

If you try to call a function with arguments without first calling pushmark, Perl dies violently with an “Out of memory!” error and takes Lisp down with it.

The PUTBACK macro's purpose purpose in C is to reset the global stack pointer to the value of the local copy. Since we are working directly with the global pointer, we can omit PUTBACK.

9.3 Pushing Arguments Onto the Stack

The XPUSHs macro in the Perl API is used to push new scalar values onto the Perl stack. The first thing it does is extend the stack if necessary, using the EXTEND macro, which looks like this:

35a ⟨EXTEND *macro in C* 35a⟩≡

```
#define EXTEND(p,n) STMT_START { if (PL_stack_max - p < (int)(n)) { \
                                   sp = stack_grow(sp,p, (int) (n));  \
                                 } } STMT_END
```

The STMT_START and STMT_END do nothing; they are macros used by the Perl source to prevent certain C compiler warnings.

We replace this with a Lisp function that serves the same purpose. Since we are treating PL_stack_sp as an :address, we must declare the Perl_stack_grow function to return an :address (instead of a :pointer) as well.

35b ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁34b 37a▷

```
(defcvar "PL_stack_max" :address)
(defcfun "Perl_stack_grow" :address
  (sp :address) (p :address) (n :uint))
```

Uses :address 31b.

35c ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁34c 37b▷

```
#:*pl-stack-max* #:perl-stack-grow
```

35d ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁34d 36a▷

```
(defun ensure-room-on-stack (n)
  (when (< (- *pl-stack-max* *pl-stack-sp*) n)
    (setf *pl-stack-sp*
          (perl-stack-grow *pl-stack-sp* *pl-stack-sp* n))))
```

Defines:
ensure-room-on-stack, used in chunk 36a.
Uses *pl-stack-sp* 33b.

The XPUSHs macro looks like this:

35e ⟨XPUSHs *macro in C* 35e⟩≡

```
#define XPUSHs(s) STMT_START { EXTEND(sp,1); (*++sp = (s)); } STMT_END
```

Now we can define an equivalent to XPUSHs.

36a ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁35d 36c▷

```
(defun pushs (scalar)
  "Push a scalar value (a pointer) onto the Perl stack."
  (ensure-room-on-stack 1) ; EXTEND
  (setf (address-ref (address-incf *pl-stack-sp*) :address)
        (pointer-address scalar)))
```

Defines:
pushs, used in chunks 36d, 38b, 40b, and 41a.
Uses *pl-stack-sp* 33b, :address 31b, address-incf 32a, address-ref 32b, and ensure-room-on-stack 35d.

9.4 Popping Values Off the Stack

Popping a scalar value off the Perl stack is, thankfully, much simpler.

The C macro is:

36b ⟨POPs *macro in C* 36b⟩≡

```
#define POPs (*sp--)
```

Which becomes, in Lisp:

36c ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁36a 38b▷

```
(defun pops ()
  "Pop a scalar pointer off the Perl stack."
  (prog1
      (address-ref *pl-stack-sp* :pointer)
    (address-decf *pl-stack-sp*)))
```

Defines:
pops, used in chunks 36d, 38c, 40b, 41a, 46e, and 47a.
Uses *pl-stack-sp* 33b, address-decf 32a, and address-ref 32b.

Phew. Let's test that, shall we? We will check that pushing a value on to the stack and popping it off gives back the same value, and check that the stack pointer is in the same place after the operation as it was before.

36d ⟨*API Tests* 12e⟩+≡ (53b) ◁30g 38a▷

```
(define-test push-and-pop-one-scalar
  (let ((x (perl-newsv 0))
        (old-stack-address *pl-stack-sp*))
    (pushs x)
    (assert-equality #'pointer-eq x
                     (address-ref *pl-stack-sp* :pointer))
    (assert-equality #'pointer-eq x (pops))
    (assert-equal old-stack-address *pl-stack-sp*)))
```

Uses *pl-stack-sp* 33b, address-ref 32b, perl-newsv 8d, pops 36c, and pushs 36a.

9.5 Scope and Temporary Variables

When creating temporary variables to place on the stack as function arguments, we must define a scope for those variables to live in, and free their memory when we are finished with them. The Perl API provides a set of four macros for this purpose, described in perlcall: ENTER and SAVETMPS begin a new scope for local variables, and FREETMPS and LEAVE end the scope. Within that scope, local variables must be declared "mortal" with the sv_2mortal() function. We can imitate all of this in Lisp.

Normally, the SAVETMPS and FREETMPS macros fiddle with a "temporary value stack" to avoid calling free_tmps if not necessary. To keep it simple, we will always call free_tmps. This does no harm and should not be a major performance drain. As a result of this simplification, we can completely omit SAVETMPS.

37a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁35b 39a▷

```
(defcfun "Perl_push_scope" :void) ; ENTER
(defcfun "Perl_free_tmps" :void)  ; FREETMPS
(defcfun "Perl_pop_scope" :void)  ; LEAVE
(defcfun "Perl_sv_2mortal" :sv
  (scalar :sv))
```

Defines:
perl-free-tmps, used in chunk 37.
perl-pop-scope, used in chunk 37.
perl-push-scope, used in chunk 37.
perl-sv-2mortal, used in chunks 37, 38, 40b, 41a, and 47b.
Uses :sv 8a.

37b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁35c 39b▷

```
#:perl-push-scope #:perl-free-tmps #:perl-pop-scope #:perl-sv-2mortal
```

Uses perl-free 6a, perl-free-tmps 37a, perl-pop-scope 37a, perl-push-scope 37a, and perl-sv-2mortal 37a.

Nowe we can create a Lisp macro that neatly packages up the process of creating a Perl scoping block.

37c ⟨*Wrapper Library Macros* 37c⟩≡ (53a)

```
(defmacro perl-scope (&body body)
  (let ((return-symbol (gensym)))
    `(progn
       (perl-push-scope)  ; ENTER
       ;; SAVETMPS omitted
       (let ((,return-symbol (progn ,@body))) ; execute body
         (perl-free-tmps) ; FREETMPS
         (perl-pop-scope) ; LEAVE
         ,return-symbol))))
```

Defines:
perl-scope, used in chunks 29a, 38, 44a, and 47b.
Uses perl-free 6a, perl-free-tmps 37a, perl-pop-scope 37a, and perl-push-scope 37a.

The `let` in the middle allows us to return values from the `body`, which we will want to do when we start calling Perl functions.

We can test the scope by making sure that a scalar declared "mortal" inside the scope has its reference count set to zero outside of the scope.

38a ⟨*API Tests* 12e⟩+≡ (53b) ◁36d 40b▷

```
(define-test perl-scope
  (let ((x))
    (perl-scope
     (setf x (perl-newsviv 999))
     (perl-sv-2mortal x)
     (assert-equal 999 (perl-sv-2iv x))
     ;; still within the scope block:
     (assert-equal 1 (foreign-slot-value x 'sv 'refcnt)))
    ;; outside the scope block here:
    (assert-equal 0 (foreign-slot-value x 'sv 'refcnt))))
```

Uses `perl-newsviv` 9a, `perl-scope` 37c, `perl-sv-2iv` 13a, and `perl-sv-2mortal` 37a.

9.6 Using the Perl Stack

38b ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁36c 38c▷

```
(defun push-mortals-on-stack (args)
  (loop for arg in args
        do (pushs (perl-sv-2mortal (perl-from-lisp arg)))))
```

Defines:
`push-mortals-on-stack`, used in chunks 38d and 44a.
Uses `perl-from-lisp` 49, `perl-sv-2mortal` 37a, and `pushs` 36a.

38c ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁38b 40c▷

```
(defun get-from-stack (n)
  (nreverse (loop repeat n
                  collecting (lisp-from-perl (pops)))))

(defun get-stack-values (n)
  (values-list (get-from-stack n)))
```

Defines:
`get-from-stack`, used in chunk 38d.
`get-stack-values`, used in chunks 43b and 47b.
Uses `lisp-from-perl` 50b and `pops` 36c.

38d ⟨*Wrapper Library Tests* 10e⟩+≡ (53b) ◁19c 44c▷

```
(define-test stack
  (let ((things (list 1 2 "hello" -27)))
    (perl-scope
     (push-mortals-on-stack things)
     (assert-equal things (get-from-stack (length things))))))
```

Uses `get-from-stack` 38c, `perl-scope` 37c, and `push-mortals-on-stack` 38b.

10 Calling Perl Functions

THIS SECTION INCOMPLETE.

Calling Perl functions (or subroutines, as Perl calls them) always boils down to a single function, `Perl_call_sv`, which takes a scalar argument which can be a the name of function (a string) or an anonymous function reference. All parameter passing to and from the Perl function is done on the Perl stack.

39a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁37a 39c▷

```
(defcfun "Perl_call_sv" :i32
  (name :sv)
  (flags :i32))
```

Defines:
`perl-call-sv`, used in chunks 39b and 40c.
Uses `:sv` 8a.

39b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁37b 39d▷

```
#:perl-call-sv
```

Uses `perl-call-sv` 39a.

The second argument is a bitfield specifying the type of function to call, the context (array, scalar, or void) in which to call it, and how to handle errors. Here are the values, copied from Perl's `cop.h` along with their documenting comments.

39c ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁39a 39e▷

```
(defbitfield perl-call-flags
  (:scalar   0)  ; call in scalar context
  (:array    1)  ; call in array context
  (:void   128)  ; call in void context (no return values)
  (:discard  2)  ; Call FREETMPS.
  (:eval     4)  ; Assume ‘eval {}’ around subroutine call.
  (:noargs   8)  ; Don’t construct a @_ array.
  (:keeperr 16)  ; Append errors to $@, don’t overwrite it.
  (:nodebug 32)  ; Disable debugging at toplevel.
  (:method  64)) ; Calling method.
```

Defines:
`perl-call-flags`, used in chunk 39d.

39d ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁39b 40a▷

```
#:perl-call-flags
```

Uses `perl-call-flags` 39c.

A shortcut exists that takes a C string as its argument instead of a scalar:

39e ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁39c 45a▷

```
(defcfun "Perl_call_pv" :i32
  (name :string)
  (flags :i32))
```

Defines:
`perl-call-pv`, used in chunk 40.

40a ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁39d 45b▷

```
#:perl-call-pv
```

Uses `perl-call-pv` 39e.

40b ⟨*API Tests* 12e⟩+≡ (53b) ◁38a 41a▷

```
(define-test perl-call-pv
  (assert-equal 1  ; 1 value was return on the stack
                (progn
                  (perl-eval-pv
"sub meaning_of_life { print \"\\nThis should be forty-two: \",
 $_[0], \"\\n\"; return $_[0]; }" t)
                  (pushmark)
                  (pushs (perl-sv-2mortal (perl-newsviv 42)))
                  (perl-call-pv "meaning_of_life"
                                (foreign-bitfield-value
                                 'perl-call-flags
                                 '(:scalar)))))
  (assert-equal 42 (perl-sv-2iv (pops))))
```

Uses `perl-call-pv` 39e, `perl-eval-pv` 45a, `perl-newsviv` 9a, `perl-sv-2iv` 13a, `perl-sv-2mortal` 37a, `pops` 36c, `pushmark` 34d, and `pushs` 36a.

We can abstract out the foreign bitfield with a function. We will export this function so that libraries that use this library will not need to import any symbols from CFFI. Here, `flags` should be a list.

40c ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁38c 42▷

```
(defun perl-call-scalar (scalar flags)
  (perl-call-sv scalar (foreign-bitfield-value
                        'perl-call-flags flags)))

(defun perl-call-function (name flags)
  (perl-call-pv name (foreign-bitfield-value
                      'perl-call-flags flags)))
```

Defines:
`perl-call-function`, used in chunk 41a.
`perl-call-scalar`, never used.

Uses `perl-call-pv` 39e and `perl-call-sv` 39a.

41a ⟨*API Tests* 12e⟩+≡ (53b) ◁40b 45c▷

```
(define-test perl-call-function
  (assert-equal 1 (progn
                   (perl-eval-pv
"sub meaning_of_life { print \"\\nThis should be forty-two: \",
 $_[0], \"\\n\"; return $_[0]; }" t)
                   (pushmark)
                   (pushs (perl-sv-2mortal (perl-newsviv 42)))
                   (perl-call-function "meaning_of_life" '(:scalar))))
  (assert-equal 42 (perl-sv-2iv (pops))))
```

Uses `perl-call-function` 40c, `perl-eval-pv` 45a, `perl-newsviv` 9a, `perl-sv-2iv` 13a, `perl-sv-2mortal` 37a, `pops` 36c, `pushmark` 34d, and `pushs` 36a.

10.1 Perl Calling Contexts

Users of this library should not have to worry about the special flags used when calling Perl functions from C. However, we can't entirely shield them from Perl's notion of calling contexts. Perl functions can be called in *scalar context*, *list context*, or *void context*. The interpreter determines the context based on how the return value of the function is used. For example:

41b ⟨*sample Perl code* 41b⟩≡

```
my $scalar = func();  # called in scalar context
my @array = func();  # called in list context
func();  # called in void context (return value not used)
```

In this example, `func` may return entirely different values in each of the contexts in which it is called. We cannot recreate this behavior in Lisp without doing ghastly things to syntax and functional purity.

Furthermore, in Perl the number 5 is indestinguishable from the string "5"—both are scalars. Lisp is dynamically typed, but not that dynamic; it does has *some* modesty.

The simplest albeit not the prettiest way out of this dilemma is to force the user to specify the type of the return value. Thus, in the `eval-perl`, `call-perl`, and `call-perl-method` functions, below, the first argument specifies the return type.

An argument of `nil` will call the function in void context and will return nothing. Most functions will behave the same way in a void context as they do in scalar context, they simply discard their return value.

The following arguments will call the function in a scalar context:

- `:integer`
- `:float` – a double-precision float
- `:string`
- `:object` – a Perl object reference, opaque to Lisp
- `t` – automatically chooses the best representation, in the same order of preference as they are listed above, but always in scalar context

The following arguments will call the function in a list context:

- `:list` – a Lisp list
- `:array` – a Lisp array
- `:alist` – a Lisp association list (actual returned value must be recognizable as a Perl hash table)
- `:hash` – a Lisp hash table (actual returned value must be recognizable as a Perl hash table)

Note that Perl does not have an explicit "hash context" for calling functions. Perl functions that return a hash table actually return a list in the form "key1, value1, key2, value2." Assigning this list to a Perl hash variable causes it to be interpreted as a hash table. Again, Lisp is not quite *that* dynamic, so we must specify the result type.

The following function will return the correct flag, `:void`, `:scalar`, or `:array`, to use when calling Perl, based on the given return type.

42 ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁40c 43a▷

```
(defun context-from-type (type)
  (cond
   ((null type) :void)
   ((find type '(:integer :float :string t)) :scalar)
   ((find type '(:list :array :alist :hash)) :array)
   (t (error "No Perl calling context for type ~a" type))))
```

Defines:
`context-from-type`, used in chunk 43a.

43a ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁42 43b▷

```
(defun calling-flags (type methodp)
  (let ((flags (list :eval (context-from-type type))))
    (when methodp (push :method flags))
    flags))
```

Defines:
`calling-flags`, used in chunk 44a.
Uses `context-from-type` 42.

NOT CORRECTLY IMPLEMENTED YET:

43b ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁43a 46f▷

```
(defun get-stack-by-type (type count)
  (declare (ignore type)) ;; fix me
  (get-stack-values count))
```

Defines:
`get-stack-by-type`, used in chunk 44a.
Uses `get-stack-values` 38c.

10.2 Public Interface

THIS NEEDS TO BE REDESIGNED.

We cannot use `:discard` in the calling flags because that would destroy the return value before we can use it (discovered by trial and error).

44a ⟨*Wrapper Library Public Functions* 7b⟩+≡ (53a) ◁7b 48b▷

```
(defun call-perl (function return-type methodp &rest args)
  (need-perl)
  (perl-scope
   (pushmark)
   (push-mortals-on-stack args)
   (get-stack-by-type
    return-type
    (funcall (if (stringp function) #'perl-call-function
                 ;; either a scalar string or a code reference
                 #'perl-call-scalar)
             function
             (calling-flags return-type methodp)))))
```

Defines:
`call-perl`, used in chunk 44.

Uses `calling-flags` 43a, `get-stack-by-type` 43b, `need-perl` 7d, `perl-scope` 37c, `push-mortals-on-stack` 38b, and `pushmark` 34d.

44b ⟨*Wrapper Library Exports* 7c⟩+≡ (51b) ◁7c 48c▷

```
#:call-perl
```

Uses `call-perl` 44a.

44c ⟨*Wrapper Library Tests* 10e⟩+≡ (53b) ◁38d 48a▷

```
(define-test call-perl
  (eval-perl "use CGI;")
  (assert-equal "<p>Hello, 1999</p>"
                (call-perl "CGI::p" :string nil "Hello," 1999)))
```

Uses `call-perl` 44a and `eval-perl` 47b.

11 Evaluating Perl Code

We can evaluate arbitrary strings of Perl code with the `Perl_eval_pv` function. Its first argument is a string of Perl code, which may be an expression, a semicolon-terminated statement, or multiple semicolon-separated statements. The second argument is a boolean specifying whether or not the process should die if a Perl error occurs.

`Perl_eval_pv` always returns a single scalar as its result, so the given statements or expressions will be evaluated in scalar context.

45a ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁39e 46c▷

```
(defcfun "Perl_eval_pv" :sv
  (code :string)
  (die-on-error :boolean))
```

Defines:
`perl-eval-pv`, used in chunks 16a, 19c, 40b, 41a, 45, and 46.
Uses `:sv` 8a.

45b ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁40a 46d▷

```
#:perl-eval-pv
```

Uses `perl-eval-pv` 45a.

45c ⟨*API Tests* 12e⟩+≡ (53b) ◁41a 45d▷

```
(define-test eval-pv-expressions
  (assert-equal 7 (perl-sv-2iv (perl-eval-pv "3 + 4" nil)))
  (assert-equal "7" (perl-in-lisp::string-from-sv (perl-eval-pv "3 + 4" nil)))
  (assert-equal "olleh" (perl-in-lisp::string-from-sv
                          (perl-eval-pv "reverse 'hello'" nil))))
```

Uses `perl-eval-pv` 45a, `perl-sv-2iv` 13a, and `string-from-sv` 15a.

Anything that can go in a normal Perl script can be used in `Perl_eval_pv`: you can load other modules, create variables, and declare packages.

45d ⟨*API Tests* 12e⟩+≡ (53b) ◁45c 46a▷

```
(define-test eval-pv-multi-statement
  (assert-equal "<p align=\"center\">Hello, world!</p>"
                (perl-in-lisp::string-from-sv
                 (perl-eval-pv "
package PerlInLisp::Tests;
use CGI;
my $cgi = new CGI;
$cgi->p({align=>'center'}, 'Hello, world!');" nil))))
```

Uses `perl-eval-pv` 45a and `string-from-sv` 15a.

Note that a single call to `Perl_eval_pv` defines a block of Perl scope. Local variables declared with `my` will not retain their value between calls.

46a ⟨*API Tests* 12e⟩+≡ (53b) ◁45d 46b▷

```
(define-test eval-pv-local-scope
  (assert-equal 385 (perl-sv-2uv (perl-eval-pv "my $x = 385" nil)))
  (assert-equal 0 (perl-sv-2uv (perl-eval-pv "$x" nil))))
```

Uses `perl-eval-pv` 45a and `perl-sv-2uv` 13a.

To keep values between calls, you must use package-global variables.

46b ⟨*API Tests* 12e⟩+≡ (53b) ◁46a 46e▷

```
(define-test eval-pv-global-scope
  (assert-equal 200 (perl-sv-2uv (perl-eval-pv "$var = 200" nil)))
  (assert-equal 200 (perl-sv-2uv (perl-eval-pv "$var" nil))))
```

Uses `perl-eval-pv` 45a and `perl-sv-2uv` 13a.

`Perl_eval_pv` is actually only a shortcut for the more general `Perl_eval_sv`, which takes the code argument as a scalar. Its second argument is the same set of flags as those used by `Perl_call_sv`. Also like `call_sv`, its integer return value is the number of result values placed on the Perl stack.

46c ⟨*CFFI Definitions* 4c⟩+≡ (52b) ◁45a

```
(defcfun "Perl_eval_sv" :i32
  (code :sv)
  (flags :i32))
```

Defines:
`perl-eval-sv`, used in chunk 46.
Uses `:sv` 8a.

46d ⟨*perl-api Exports* 4d⟩+≡ (51a) ◁45b

```
#:perl-eval-sv
```

Uses `perl-eval-sv` 46c.

46e ⟨*API Tests* 12e⟩+≡ (53b) ◁46b 47a▷

```
(define-test eval-sv
  (assert-equal 1 (perl-eval-sv (perl-newsvpv "20 + 7" 0)
                                (foreign-bitfield-value
                                 'perl-call-flags
                                 '(:scalar :eval))))
  (assert-equal 27 (perl-sv-2iv (pops))))
```

Uses `perl-eval-sv` 46c, `perl-newsvpv` 9c, `perl-sv-2iv` 13a, and `pops` 36c.

We can abstract away the bitfield here just as with the `perl-call` functions.

46f ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁43b 47b▷

```
(defun perl-eval-scalar (scalar flags)
  (perl-eval-sv scalar (foreign-bitfield-value
                        'perl-call-flags flags)))
```

Defines:
`perl-eval-scalar`, used in chunk 47.
Uses `perl-eval-sv` 46c.

47a ⟨*API Tests* 12e⟩+≡ (53b) ◁46e

```
(define-test perl-eval-scalar
  (assert-equal 1 (perl-eval-scalar (perl-newsvpv "33+1" 0) '(:scalar)))
  (assert-equal 34 (perl-sv-2iv (pops))))
```

Uses `perl-eval-scalar` 46f, `perl-newsvpv` 9c, `perl-sv-2iv` 13a, and `pops` 36c.

This wrapper function will evaluate the given string of Perl code in scalar context, returning whatever that code returns, automatically converted to the most likely Lisp type.

47b ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁46f 49▷

```
(defun eval-perl (code)
  (need-perl)
  (perl-scope
   (get-stack-values
    (perl-eval-scalar (perl-sv-2mortal (perl-newsvpv code 0))
                      '(:scalar :eval)))))
```

Defines:
`eval-perl`, used in chunks 44c and 48.
Uses `get-stack-values` 38c, `need-perl` 7d, `perl-eval-scalar` 46f, `perl-newsvpv` 9c, `perl-scope` 37c, and `perl-sv-2mortal` 37a.

48a ⟨*Wrapper Library Tests* 10e⟩+≡ (53b) ◁44c 50c▷

```
(define-test eval-perl
  (assert-equal 7 (eval-perl "3+4"))
  (assert-equal "abcdef" (eval-perl "'abc' . 'def'"))
  (assert-equal '(1 2 3) (eval-perl "[1, 2, 3];")))

(define-test eval-perl-with-hash
  (let ((hash
         (eval-perl
          "{aa => 1, bb => 3.14, cc => 'hello'};")))
    (assert-true (hash-table-p hash))
    (assert-equal 1 (gethash "aa" hash))
    (assert-true (< 3.13 (gethash "bb" hash) 3.15))
    (assert-equal "hello" (gethash "cc" hash))
    ))

(define-test eval-perl-creating-hash
  (let ((hash (make-hash-table)))
    (setf (gethash 'key1 hash) "one")
    (setf (gethash 'key2 hash) "two")
    (setf (gethash 'key3 hash) "three")
    (let ((new-hash (lisp-from-perl (perl-from-lisp hash))))
      (assert-true (hash-table-p new-hash))
      (assert-equal "one" (gethash "KEY1" new-hash))
      (assert-equal "two" (gethash "KEY2" new-hash))
      (assert-equal "three" (gethash "KEY3" new-hash)))))
```

Uses `eval-perl` 47b, `lisp-from-perl` 50b, and `perl-from-lisp` 49.

12 Loading Perl Modules

The Perl API provides the `load_module` function as an equivalent to the `use` directive in Perl code. I could never make it work correctly, and it seems to require a module version number anyway. As a simpler alternative, evaluate a standard Perl `use` statement in an `eval-perl`.

48b ⟨*Wrapper Library Public Functions* 7b⟩+≡ (53a) ◁44a

```
(defun use-perl-module (name &optional version)
  (eval-perl (format nil "use ~A ~@[~A~];" name version)))
```

Defines:
`use-perl-module`, used in chunk 48c.
Uses `eval-perl` 47b.

48c ⟨*Wrapper Library Exports* 7c⟩+≡ (51b) ◁44b

```
#:use-perl-module
```

Uses `use-perl-module` 48b.

13 Automatic Type Conversions

It would be really useful to have generic functions that would convert automatically between appropriate types. Perl arrays can become Lisp lists, hash tables can be hashes, and so on.

Here I follow the slightly out-of-fashion Hungarian notation of naming conversion functions "x FROM y" rather than "y TO x." I find the former easier to read, because it puts the type name closest to the object that is of that type.

49 ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁47b 50a▷

```
(defgeneric perl-from-lisp (value))

(defmethod perl-from-lisp ((value integer))
  (need-perl)
  (cond
   ((and (<= 0 value 4294967295)) ;; 32-bit unsigned integers
    (perl-newsvuv value))
   ((and (> 0 value -2147483648)) ;; 32-bit signed integers
    (perl-newsviv value))
   (t (error "Integer value out of range for Perl;
BigInts not supported"))))

(defmethod perl-from-lisp ((value float))
  (need-perl)
  (perl-newsvnv
   ;; ensure VALUE is a double-float
   (float value 1.0d0)))

(defmethod perl-from-lisp ((value string))
  (need-perl)
  (perl-newsvpv value 0))

(defmethod perl-from-lisp ((value list))
  (let ((a (perl-newav)))
    (loop for i in value
          ;; Perl's "push" pushes to the *end* of the array
          do (perl-av-push a (perl-from-lisp i)))
    a))

(defmethod perl-from-lisp ((value hash-table))
  (hv-from-hash value))
```

Defines:
`perl-from-lisp`, used in chunks 29b, 38b, 48a, and 50.
Uses `hv-from-hash` 29b, `need-perl` 7d, `perl-av-push` 22d, `perl-newav` 20b, `perl-newsviv` 9a, `perl-newsvnv` 9a, `perl-newsvpv` 9c, and `perl-newsvuv` 9a.

While the Perl API uses `&PL_sv_undef` to indicate an `undef` value, the recommended way to add undefined values to arrays and hashes is to create a new empty scalar.

50a ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁49 50b▷

```
(defmethod perl-from-lisp ((value null)) ; NIL isn't a class; NULL is
  (perl-newsv 0))
```

Uses `perl-from-lisp` 49 and `perl-newsv` 8d.

50b ⟨*Wrapper Library Internal Functions* 5a⟩+≡ (53a) ◁50a

```
(defun lisp-from-perl (p)
  (ecase (svtype p)
    (:null nil)
    (:iv (perl-sv-2iv p))
    (:nv (perl-sv-2nv p))
    (:rv (lisp-from-perl (deref-rv p)))
    (:pv (string-from-sv p))
    (:pviv (cffi:mem-ref p :iv))
    (:pvnv (cffi:mem-ref p :nv))
    (:pvmg (error "Blessed or magical scalars not supported yet"))
    (:pvav (list-from-av p))
    (:pvhv (hash-from-hv p))))
```

Defines:
`lisp-from-perl`, used in chunks 25d, 29a, 38c, 48a, and 50c.
Uses `deref-rv` 30e, `hash-from-hv` 29a, `list-from-av` 25e, `perl-sv-2iv` 13a, `perl-sv-2nv` 13a, `string-from-sv` 15a, and `svtype` 11a 12a.

50c ⟨*Wrapper Library Tests* 10e⟩+≡ (53b) ◁48a

```
(define-test lisp-from-perl-scalars
  (assert-equal 42 (lisp-from-perl (perl-from-lisp 42)))
  (assert-equal "Hello, world!"
                (lisp-from-perl (perl-from-lisp "Hello, world!")))
  (assert-true
   ;; we can't get exact equality from floats
   (< 3.14589 (lisp-from-perl (perl-from-lisp 3.1459)) 3.14591))
  (assert-equal nil (lisp-from-perl (perl-from-lisp nil))))
```

Uses `lisp-from-perl` 50b and `perl-from-lisp` 49.

14 Packages

51a ⟨*perl-api Package Definition* 51a⟩≡ (52b)

```
;;;; perlapi.lisp - CFFI definitions for the Perl C API

⟨License Header 59⟩

(cl:in-package :common-lisp-user)

(defpackage :perl-api
  (:use :common-lisp :cffi :cffi-address)
  (:export ⟨perl-api Exports 4d⟩))

(in-package :perl-api)
```

Defines:
`perl-api`, used in chunks 51b and 53b.

51b ⟨*perl-in-lisp Package Definition* 51b⟩≡ (53a)

```
;;;; Perl-in.lisp - Lisp interface to the Perl API

⟨License Header 59⟩

(cl:in-package :common-lisp-user)

(defpackage :perl-in-lisp
  (:use :common-lisp :cffi :cffi-address :perl-api)
  (:nicknames :perl)
  (:export ⟨Wrapper Library Exports 7c⟩))

(in-package :perl-in-lisp)
```

Uses `perl-api` 51a.

15 ASDF System Definition

52a ⟨*perl-in-lisp.asd* 52a⟩≡

```
;;;; perl-in-lisp.asd - ASDF definition for a Lisp interface to Perl

⟨License Header 59⟩

(defpackage :perl-in-lisp.system
  (:documentation "ASDF system package for PERL-IN-LISP.")
  (:use :common-lisp :asdf))

(in-package :perl-in-lisp.system)

(defsystem :perl-in-lisp
  :components ((:static-file "perl-in-lisp.asd")
               (:module :src
                        :serial t
                        :components ((:file "address")
                                     (:file "perlapi")
                                     (:file "perl-in"))))
  :depends-on (:cffi))


(defsystem :perl-in-lisp.test
  :components ((:module :tests
                        :serial t
                        :components ((:file "lisp-unit")
                                     (:file "tests"))))
  :depends-on (:perl-in-lisp))


(defmethod perform ((op asdf:test-op) (system (eql (find-system :perl-in-lisp))))
  (asdf:oos 'asdf:load-op :perl-in-lisp.test)
  (format t "Tests loaded.
Change to package PERL-IN-LISP and execute
(RUN-TESTS) to run all tests."))
```

16 Output Files

52b ⟨*perlapi.lisp* 52b⟩≡

```
⟨perl-api Package Definition 51a⟩
⟨Libperl foreign library definition 3a⟩
⟨Perl API Types 3b⟩
⟨CFFI Definitions 4c⟩
```

53a ⟨*perl-in.lisp* 53a⟩≡

```
⟨perl-in-lisp Package Definition 51b⟩
⟨Wrapper Library Globals 7a⟩
⟨Wrapper Library Macros 37c⟩
⟨Wrapper Library Internal Functions 5a⟩
⟨Wrapper Library Public Functions 7b⟩
```

53b ⟨*tests.lisp* 53b⟩≡

```
;;;; tests.lisp -- Unit tests (with Lisp-Unit) for Perl-in-Lisp

⟨License Header 59⟩

(in-package :common-lisp-user)

;; (defpackage :perl-in-lisp.test
;;   (:use :common-lisp :perl-in-lisp :perl-api
;;      :lisp-unit :cffi :cffi-address)
;;   (:export #:run-tests)

(eval-when (:compile-toplevel :load-toplevel :execute)
  (use-package :lisp-unit :perl-in-lisp))

(in-package :perl-in-lisp)
⟨API Tests 12e⟩
⟨Wrapper Library Tests 10e⟩
```

Uses `perl-api` 51a.

17 Development Aids

17.1 Makefile

Edit this Makefile as needed, then extract it with the following command:

`notangle -t8 perl-in-lisp.nw >Makefile`

To generate the source code and documentation (DVI) run `make`.

The other defined makefile targets are:

`doc` — only generate the documentation

`code` — only extract the source code

`pdf` — generate PDF documentation instead of DVI (requires pdflatex)

`html` — generate HTML documentation

To re-extract this Makefile, run `make remake`.

54 ⟨* 54⟩≡

```
SHELL=/bin/sh
TANGLE=notangle
WEAVE=noweave
LATEX=latex
PDFLATEX=pdflatex
ENSURE_DIR=mkdir -p
FASLS=*.fasl *.fas *.lib *.x86f

all: code doc

code: perl-in-lisp.nw
        $(ENSURE_DIR) src tests
        $(TANGLE) -Rperl-in-lisp.asd  perl-in-lisp.nw >perl-in-lisp.asd
        $(TANGLE) -Raddress.lisp  perl-in-lisp.nw >src/address.lisp
        $(TANGLE) -Rperlapi.lisp  perl-in-lisp.nw >src/perlapi.lisp
        $(TANGLE) -Rperl-in.lisp perl-in-lisp.nw >src/perl-in.lisp
        $(TANGLE) -Rtests.lisp  perl-in-lisp.nw >tests/tests.lisp

doc: perl-in-lisp.nw
        $(ENSURE_DIR) doc
        $(WEAVE) -t8 -latex -delay -index perl-in-lisp.nw >doc/perl-in-lisp.tex
        # run latex twice to get references right
        cd doc; $(LATEX) perl-in-lisp.tex; $(LATEX) perl-in-lisp.tex

# pdf depends on doc to ensure latex was already run once to generate
# references and table of contents
pdf: doc
        cd doc; $(PDFLATEX) perl-in-lisp.tex

html: perl-in-lisp.nw
        $(WEAVE) -index -html -filter l2h perl-in-lisp.nw | htmltoc >doc/perl-in-lisp.h
```

```
remake: perl-in-lisp.nw
        $(TANGLE) -t8 perl-in-lisp.nw >Makefile

clean:
        rm -f *~ *.out $(FASLS)
        cd src; rm -f $(FASLS)
        cd tests; rm -f $(FASLS)
        cd doc; rm -f *.aux *.log *.tex *.toc

dist: remake code doc html clean
        cd doc; rm -f *.dvi *.pdf
```

17.2 List of All Code Chunks

This list is automatically generated by Noweb.

17.3 Symbol Index

This list is automatically generated by Noweb. The underlined number after each symbol specifies the page and code chunk on which that symbol is defined; other numbers specify pages and chunks where that symbol is used.

18 License (LLGPL)

59 ⟨*License Header* 59⟩≡ (33a 51–53)

```
;;; Copyright 2006 Stuart Sierra

;;; This program is free software; you can redistribute it and/or
;;; modify it under the terms of the Lisp Lesser GNU General Public
;;; License (LLGPL) published by Franz, Inc., available at
;;; http://opensource.franz.com/preamble.html

;;; This program is distributed in the hope that it will be useful,
;;; but WITHOUT ANY WARRANTY; without even the implied warranty of
;;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
;;; Lesser GNU General Public License for more details.
```