SBCL User Manual

SBCL version 1.0.13
2008-01

Table of Contents

1 Introduction

SBCL is a mostly-conforming implementation of the ANSI Common Lisp standard. This manual focuses on behavior which is specific to SBCL, not on behavior which is common to all implementations of ANSI Common Lisp.

1.1 ANSI Conformance

Essentially every type of non-conformance is considered a bug. (The exceptions involve internal inconsistencies in the standard.) In SBCL the master record of known bugs is in the '`BUGS`' file in the distribution.

The recommended way to report bugs is through the *sbcl-help* or *sbcl-devel* mailing lists. For mailing list addresses, see Section 1.5.1 [SBCL Homepage], page 4.

1.2 Extensions

SBCL comes with numerous extensions, some in core and some in modules loadable with `require`. Unfortunately, not all of these extensions have proper documentation yet.

System Definition Tool
: `asdf` is a flexible and popular protocol-oriented system definition tool by Daniel Barlow. See Info file '`asdf`', node '`Top`', for more information.

Third-party Extension Installation Tool
: `asdf-install` is a tool that can be used to download and install third-party libraries and applications, automatically handling dependencies, etc.

Foreign Function Interface
: `sb-alien` package allows interfacing with C-code, loading shared object files, etc. See Chapter 7 [Foreign Function Interface], page 55.

  `sb-grovel` can be used to partially automate generation of foreign function interface definitions. See Section 15.2 [sb-grovel], page 108.

Recursive Event Loop
: SBCL provides a recursive event loop (`serve-event`) for doing non-blocking IO on multiple streams without using threads.

Metaobject Protocol
: `sb-mop` package provides a metaobject protocol for the Common Lisp Object System as described in *Art of Metaobject Protocol*.

Native Threads
: SBCL has native threads on x86/Linux, capable of taking advantage of SMP on multiprocessor machines. See Chapter 11 [Threading], page 87.

Network Interface
: `sb-bsd-sockets` is a low-level networking interface, providing both TCP and UDP sockets. See Chapter 13 [Networking], page 95.

Introspective Facilities
: `sb-introspect` module offers numerous introspective extensions, including access to function lambda-lists.

Operating System Interface
: `sb-ext` contains a number of functions for running external processes, accessing environment variables, etc.

  `sb-posix` module provides a lispy interface to standard POSIX facilities.

Extensible Streams
: `sb-gray` is an implementation of *Gray Streams*. See Section 9.2 [Gray Streams], page 71.

  `sb-simple-streams` is an implementation of the *simple streams* API proposed by Franz Inc. See Section 9.3 [Simple Streams], page 79.

Profiling
: `sb-profile` is a exact per-function profiler. See Section 14.1 [Deterministic Profiler], page 100.

  `sb-sprof` is a statistical profiler, capable of call-graph generation and instruction level profiling. See Section 14.2 [Statistical Profiler], page 100.

Customization Hooks
: SBCL contains a number of extra-standard customization hooks that can be used to tweak the behaviour of the system. See Section 6.4 [Customization Hooks for Users], page 52.

  `sb-aclrepl` provides an Allegro CL -style toplevel for SBCL, as an alternative to the classic CMUCL-style one. See Section 15.1 [sb-aclrepl], page 107.

CLTL2 Compatility Layer
: `sb-cltl2` module provides `compiler-let` and environment access functionality described in *Common Lisp The Language, 2nd Edition* which were removed from the language during the ANSI standardization process.

Executable Fasl Packaging
: `sb-executable` can be used to concatenate multiple fasls into a single executable (though the presence of an SBCL runtime and core image is still required to run it).

  The `:executable` argument to [Function sb-ext:save-lisp-and-die], page 10 can produce a 'standalone' executable containing both an image of the current Lisp session and an SBCL runtime.

Bitwise Rotation
: `sb-rotate-byte` provides an efficient primitive for bitwise rotation of integers, an operation required by eg. numerous cryptographic algorithms, but not available as a primitive in ANSI Common Lisp. See Section 15.5 [sb-rotate-byte], page 118.

Test Harness
: `sb-rt` module is a simple yet attractive regression and unit-test framework.

MD5 Sums
: `sb-md5` is an implementation of the MD5 message digest algorithm for Common Lisp, using the modular arithmetic optimizations provided by SBCL. See Section 15.4 [sb-md5], page 117.

1.3 Idiosyncrasies

The information in this section describes some of the ways that SBCL deals with choices that the ANSI standard leaves to the implementation.

1.3.1 Declarations

Declarations are generally treated as assertions. This general principle, and its implications, and the bugs which still keep the compiler from quite satisfying this principle, are discussed in Section 3.2.1 [Declarations as Assertions], page 21.

1.3.2 Compiler-only Implementation

SBCL is essentially a compiler-only implementation of Common Lisp. That is, for all but a few special cases, `eval` creates a lambda expression, calls `compile` on the lambda expression to create a compiled function, and then calls `funcall` on the resulting function object. This is explicitly allowed by the ANSI standard, but leads to some oddities, e.g. collapsing `functionp` and `compiled-function-p` into the same predicate.

1.3.3 Defining Constants

SBCL is quite strict about ANSI's definition of `defconstant`. ANSI says that doing `defconstant` of the same symbol more than once is undefined unless the new value is `eql` to the old value. Conforming to this specification is a nuisance when the "constant" value is only constant under some weaker test like `string=` or `equal`.

It's especially annoying because, in SBCL, `defconstant` takes effect not only at load time but also at compile time, so that just compiling and loading reasonable code like

```
(defconstant +foobyte+ '(1 4))
```

runs into this undefined behavior. Many implementations of Common Lisp try to help the programmer around this annoyance by silently accepting the undefined code and trying to do what the programmer probably meant.

SBCL instead treats the undefined behavior as an error. Often such code can be rewritten in portable ANSI Common Lisp which has the desired behavior. E.g., the code above can be given an exactly defined meaning by replacing `defconstant` either with `defparameter` or with a customized macro which does the right thing, eg.

```
(defmacro define-constant (name value &optional doc)
  `(defconstant ,name (if (boundp ',name) (symbol-value ',name) ,value)
                ,@(when doc (list doc))))
```

or possibly along the lines of the `defconstant-eqx` macro used internally in the implementation of SBCL itself. In circumstances where this is not appropriate, the programmer can handle the condition type `sb-ext:defconstant-uneql`, and choose either the `continue` or `abort` restart as appropriate.

1.3.4 Style Warnings

SBCL gives style warnings about various kinds of perfectly legal code, e.g.

- `defmethod` without a preceding `defgeneric`;
- multiple `defun`s of the same symbol in different units;

- special variables not named in the conventional `*foo*` style, and lexical variables unconventionally named in the `*foo*` style

This causes friction with people who point out that other ways of organizing code (especially avoiding the use of `defgeneric`) are just as aesthetically stylish. However, these warnings should be read not as "warning, bad aesthetics detected, you have no style" but "warning, this style keeps the compiler from understanding the code as well as you might like." That is, unless the compiler warns about such conditions, there's no way for the compiler to warn about some programming errors which would otherwise be easy to overlook. (Related bug: The warning about multiple `defun`s is pointlessly annoying when you compile and then load a function containing `defun` wrapped in `eval-when`, and ideally should be suppressed in that case, but still isn't as of SBCL 0.7.6.)

1.4 Development Tools

1.4.1 Editor Integration

Though SBCL can be used running "bare", the recommended mode of development is with an editor connected to SBCL, supporting not only basic lisp editing (paren-matching, etc), but providing among other features an integrated debugger, interactive compilation, and automated documentation lookup.

Currently *SLIME*[1] (Superior Lisp Interaction Mode for Emacs) together with Emacs is recommended for use with SBCL, though other options exist as well.

SLIME can be downloaded from http://www.common-lisp.net/project/slime/.

1.4.2 Language Reference

CLHS (Common Lisp Hyperspec) is a hypertext version of the ANSI standard, made freely available by *LispWorks* – an invaluable reference.

See: http://www.lispworks.com/reference/HyperSpec/index.html

1.4.3 Generating Executables

SBCL can generate stand-alone executables. The generated executables include the SBCL runtime itself, so no restrictions are placed on program functionality. For example, a deployed program can call `compile` and `load`, which requires the compiler to be present in the executable. For further information, See [Function sb-ext:save-lisp-and-die], page 10.

1.5 More SBCL Information

1.5.1 SBCL Homepage

The SBCL website at http://www.sbcl.org/ has some general information, plus links to mailing lists devoted to SBCL, and to archives of these mailing lists. Subscribing to the mailing lists *sbcl-help* and *sbcl-announce* is recommended: both are fairly low-volume, and help you keep abrest with SBCL development.

[1] Historically, the ILISP package at http://ilisp.cons.org/ provided similar functionality, but it does not support modern SBCL versions.

1.5.2 Additional Distributed Documentation

Besides this user manual both SBCL source and binary distributions include some other SBCL-specific documentation files, which should be installed along with this manual in on your system, eg. in '`/usr/local/share/doc/sbcl/`'.

'`BUGS`' Lists known bugs in the distribution.

'`COPYING`' Licence and copyright summary.

'`CREDITS`' Authorship information on various parts of SBCL.

'`INSTALL`' Covers installing SBCL from both source and binary distributions on your system, and also has some installation related troubleshooting information.

'`NEWS`' Summarizes changes between various SBCL versions.

'`SUPPORT`' Lists SBCL developers available for-pay development of SBCL.

1.5.3 Online Documentation

Documentation for non-ANSI extensions for various commands is available online from the SBCL executable itself. The extensions for functions which have their own command prompts (e.g. the debugger, and `inspect`) are documented in text available by typing `help` at their command prompts. The extensions for functions which don't have their own command prompt (such as `trace`) are described in their documentation strings, unless your SBCL was compiled with an option not to include documentation strings, in which case the documentation strings are only readable in the source code.

1.5.4 Internals Documentation

If you're interested in the development of the SBCL system itself, then subscribing to *sbcl-devel* is a good idea.

SBCL internals documentation – besides comments in the source – is currently maintained as a *wiki-like* website: http://sbcl-internals.cliki.net/.

Some low-level information describing the programming details of the conversion from CMUCL to SBCL is available in the '`doc/FOR-CMUCL-DEVELOPERS`' file in the SBCL distribution, though it is not installed by default.

1.6 More Common Lisp Information

1.6.1 Internet Community

The Common Lisp internet community is fairly diverse: news://comp.lang.lisp is fairly high volume newsgroup, but has a rather poor signal/noise ratio. Various special interest mailing lists and IRC tend to provide more content and less flames. http://www.lisp.org and http://www.cliki.net contain numerous pointers places in the net where lispers talks shop.

1.6.2 Third-party Libraries

For a wealth of information about free Common Lisp libraries and tools we recommend checking out *CLiki*: http://www.cliki.net/.

1.6.3 Common Lisp Books

If you're not a programmer and you're trying to learn, many introductory Lisp books are available. However, we don't have any standout favorites. If you can't decide, try checking the Usenet news://comp.lang.lisp FAQ for recent recommendations.

If you are an experienced programmer in other languages but need to learn about Common Lisp, some books stand out:

ANSI Common Lisp, by Paul Graham
: Introduces most of the language, though some parts (eg. CLOS) are covered only lightly.

On Lisp, by Paul Graham
: An in-depth treatment of macros, but not recommended as a first Common Lisp book, since it is slightly pre-ANSI so you need to be on your guard against non-standard usages, and since it doesn't really even try to cover the language as a whole, focusing solely on macros. Downloadable from http://www.paulgraham.com/onlisp.html.

Paradigms Of Artificial Intelligence Programming, by Peter Norvig
: Good information on general Common Lisp programming, and many nontrivial examples. Whether or not your work is AI, it's a very good book to look at.

Object-Oriented Programming In Common Lisp, by Sonya Keene
: None the books above emphasize CLOS, but this one does. Even if you're very knowledgeable about object oriented programming in the abstract, it's worth looking at this book if you want to do any OO in Common Lisp. Some abstractions in CLOS (especially multiple dispatch) go beyond anything you'll see in most OO systems, and there are a number of lesser differences as well. This book tends to help with the culture shock.

Art Of Metaobject Programming, by Gregor Kiczales et al.
: Currently to prime source of information on the Common Lisp Metaobject Protocol, which is supported by SBCL. Section 2 (Chapers 5 and 6) are freely available at http://www.lisp.org/mop/.

1.7 History and Implementation of SBCL

You can work productively with SBCL without knowing anything understanding anything about where it came from, how it is implemented, or how it extends the ANSI Common Lisp standard. However, a little knowledge can be helpful in order to understand error messages, to troubleshoot problems, to understand why some parts of the system are better debugged than others, and to anticipate which known bugs, known performance problems, and missing extensions are likely to be fixed, tuned, or added.

SBCL is descended from CMUCL, which is itself descended from Spice Lisp, including early implementations for the Mach operating system on the IBM RT, back in the 1980s. Some design decisions from that time are still reflected in the current implementation:

- The system expects to be loaded into a fixed-at-compile-time location in virtual memory, and also expects the location of all of its heap storage to be specified at compile time.

- The system overcommits memory, allocating large amounts of address space from the system (often more than the amount of virtual memory available) and then failing if ends up using too much of the allocated storage.
- The system is implemented as a C program which is responsible for supplying low-level services and loading a Lisp '`.core`' file.

SBCL also inherited some newer architectural features from CMUCL. The most important is that on some architectures it has a generational garbage collector ("GC"), which has various implications (mostly good) for performance. These are discussed in another chapter, Chapter 5 [Efficiency], page 42.

SBCL has diverged from CMUCL in that SBCL is now essentially a "compiler-only implementation" of Common Lisp. This is a change in implementation strategy, taking advantage of the freedom "any of these facilities might share the same execution strategy" guaranteed in the ANSI specification section 3.1 ("Evaluation"). It does not mean SBCL can't be used interactively, and in fact the change is largely invisible to the casual user, since SBCL still can and does execute code interactively by compiling it on the fly. (It is visible if you know how to look, like using `compiled-function-p`; and it is visible in the way that that SBCL doesn't have many bugs which behave differently in interpreted code than in compiled code.) What it means is that in SBCL, the `eval` function only truly "interprets" a few easy kinds of forms, such as symbols which are `boundp`. More complicated forms are evaluated by calling `compile` and then calling `funcall` on the returned result.

The direct ancestor of SBCL is the x86 port of CMUCL. This port was in some ways the most cobbled-together of all the CMUCL ports, since a number of strange changes had to be made to support the register-poor x86 architecture. Some things (like tracing and debugging) do not work particularly well there. SBCL should be able to improve in these areas (and has already improved in some other areas), but it takes a while.

On the x86 SBCL – like the x86 port of CMUCL – uses a *conservative* GC. This means that it doesn't maintain a strict separation between tagged and untagged data, instead treating some untagged data (e.g. raw floating point numbers) as possibly-tagged data and so not collecting any Lisp objects that they point to. This has some negative consequences for average time efficiency (though possibly no worse than the negative consequences of trying to implement an exact GC on a processor architecture as register-poor as the X86) and also has potentially unlimited consequences for worst-case memory efficiency. In practice, conservative garbage collectors work reasonably well, not getting anywhere near the worst case. But they can occasionally cause odd patterns of memory usage.

The fork from CMUCL was based on a major rewrite of the system bootstrap process. CMUCL has for many years tolerated a very unusual "build" procedure which doesn't actually build the complete system from scratch, but instead progressively overwrites parts of a running system with new versions. This quasi-build procedure can cause various bizarre bootstrapping hangups, especially when a major change is made to the system. It also makes the connection between the current source code and the current executable more tenuous than in other software systems – it's easy to accidentally "build" a CMUCL system containing characteristics not reflected in the current version of the source code.

Other major changes since the fork from CMUCL include

- SBCL has removed many CMUCL extensions, (e.g. IP networking, remote procedure call, Unix system interface, and X11 interface) from the core system. Most of these

are available as contributed modules (distributed with sbcl) or third-party modules instead.

- SBCL has deleted or deprecated some nonstandard features and code complexity which helped efficiency at the price of maintainability. For example, the SBCL compiler no longer implements memory pooling internally (and so is simpler and more maintainable, but generates more garbage and runs more slowly), and various block-compilation efficiency-increasing extensions to the language have been deleted or are no longer used in the implementation of SBCL itself.

2 Starting and Stopping

2.1 Starting SBCL

2.1.1 From Shell to Lisp

To run SBCL type `sbcl` at the command line.

You should end up in the toplevel *REPL* (read, eval, print -loop), where you can interact with SBCL by typing expressions.

```
$ sbcl
This is SBCL 0.8.13.60, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* (+ 2 2)

4
* (quit)
$
```

See also Section 2.3 [Command Line Options], page 11 and Section 2.2 [Stopping SBCL], page 9.

2.1.2 Running from Emacs

To run SBCL as an inferior-lisp from Emacs in your '`.emacs`' do something like:

```
;;; The SBCL binary and command-line arguments
(setq inferior-lisp-program "/usr/local/bin/sbcl --noinform")
```

For more information on using SBCL with Emacs, see Section 1.4.1 [Editor Integration], page 4.

2.1.3 Shebang Scripts

SBCL doesn't come with built-in support for shebang-line execution, but this can be provided with a shell trampoline, or by dispatching from initialization files (see Section 2.4.4.1 [Unix-style Command Line Protocol], page 14 for an example.)

2.2 Stopping SBCL

2.2.1 Quit

SBCL can be stopped at any time by calling `sb-ext:quit`, optionally returning a specified numeric value to the calling process. See notes in Chapter 11 [Threading], page 87 about the interaction between this feature and sessions.

`sb-ext:quit` **&key** *recklessly-p unix-status* [Function]

Terminate the current Lisp. Things are cleaned up (with `unwind-protect` and so forth) unless `recklessly-p` is non-NIL. On UNIX-like systems, `unix-status` is used as the status code.

2.2.2 End of File

By default SBCL also exits on end of input, caused either by user pressing *Control-D* on an attached terminal, or end of input when using SBCL as part of a shell pipeline.

2.2.3 Saving a Core Image

SBCL has the ability to save its state as a file for later execution. This functionality is important for its bootstrapping process, and is also provided as an extension to the user.

`sb-ext:save-lisp-and-die` *core-file-name* **&key** *toplevel purify root-structures environment-name executable* [Function]

Save a "core image", i.e. enough information to restart a Lisp process later in the same state, in the file of the specified name. Only global state is preserved: the stack is unwound in the process.

The following `&key` arguments are defined:

`:toplevel`
: The function to run when the created core file is resumed. The default function handles command line toplevel option processing and runs the top level read-eval-print loop. This function should not return.

`:executable`
: If true, arrange to combine the `sbcl` runtime and the core image to create a standalone executable. If false (the default), the core image will not be executable on its own.

`:purify`
: If true (the default on cheneygc), do a purifying `gc` which moves all dynamically allocated objects into static space. This takes somewhat longer than the normal `gc` which is otherwise done, but it's only done once, and subsequent GC's will be done less often and will take less time in the resulting core file. See the `purify` function. This parameter has no effect on platforms using the generational garbage collector.

`:root-structures`
: This should be a list of the main entry points in any newly loaded systems. This need not be supplied, but locality and/or `gc` performance may be better if they are. Meaningless if `:purify` is `nil`. See the `purify` function.

`:environment-name`
: This is also passed to the `purify` function when `:purify` is `t`. (rarely used)

The save/load process changes the values of some global variables:

`*standard-output*`, `*debug-io*`, *etc.*
: Everything related to open streams is necessarily changed, since the `os` won't let us preserve a stream across save and load.

`*default-pathname-defaults*`
: This is reinitialized to reflect the working directory where the saved core is loaded.

Foreign objects loaded with `sb-alien:load-shared-object` are automatically reloaded on startup, but references to foreign symbols do not survive intact on all platforms: in this case a `warning` is signalled when saving the core. If no warning is signalled, then the foreign symbol references will remain intact. Platforms where this is currently the case are x86/FreeBSD, x86/Linux, x86/NetBSD, sparc/Linux, sparc/SunOS, and ppc/Darwin.

On threaded platforms only a single thread may remain running after `sb-ext:*save-hooks*` have run. Applications using multiple threads can be `save-lisp-and-die` friendly by registering a save-hook that quits any additional threads, and an init-hook that restarts them.

This implementation is not as polished and painless as you might like:

- It corrupts the current Lisp image enough that the current process needs to be killed afterwards. This can be worked around by forking another process that saves the core.
- There is absolutely no binary compatibility of core images between different runtime support programs. Even runtimes built from the same sources at different times are treated as incompatible for this purpose.

This isn't because we like it this way, but just because there don't seem to be good quick fixes for either limitation and no one has been sufficiently motivated to do lengthy fixes.

To facilitate distribution of SBCL applications using external resources, the filesystem location of the SBCL core file being used is available from Lisp.

`sb-ext:*core-pathname*` [Variable]
The absolute pathname of the running `sbcl` core.

2.2.4 Exit on Errors

SBCL can also be configured to exit if an unhandled error occurs, which is mainly useful for acting as part of a shell pipeline; doing so under most other circumstances would mean giving up large parts of the flexibility and robustness of Common Lisp. See Section 4.1 [Debugger Entry], page 28.

2.3 Command Line Options

Command line options can be considered an advanced topic; for ordinary interactive use, no command line arguments should be necessary.

In order to understand the command line argument syntax for SBCL, it is helpful to understand that the SBCL system is implemented as two components, a low-level runtime environment written in C and a higher-level system written in Common Lisp itself. Some command line arguments are processed during the initialization of the low-level runtime environment, some command line arguments are processed during the initialization of the Common Lisp system, and any remaining command line arguments are passed on to user code.

The full, unambiguous syntax for invoking SBCL at the command line is:

`sbcl` *runtime-option** `--end-runtime-options` *toplevel-option** `--end-toplevel-options` *user-options**

For convenience, the `--end-runtime-options` and `--end-toplevel-options` elements can be omitted. Omitting these elements can be convenient when you are running the program interactively, and you can see that no ambiguities are possible with the option values you are using. Omitting these elements is probably a bad idea for any batch file where any of the options are under user control, since it makes it impossible for SBCL to detect erroneous command line input, so that erroneous command line arguments will be passed on to the user program even if they was intended for the runtime system or the Lisp system.

2.3.1 Runtime Options

`--core` *corefilename*
: Run the specified Lisp core file instead of the default. Note that if the Lisp core file is a user-created core file, it may run a nonstandard toplevel which does not recognize the standard toplevel options.

`--dynamic-space-size` *megabytes*
: Size of the dynamic space reserved on startup in megabytes. Default value is platform dependent.

`--noinform`
: Suppress the printing of any banner or other informational message at startup. This makes it easier to write Lisp programs which work cleanly in Unix pipelines. See also the `--noprint` and `--disable-debugger` options.

`--help` Print some basic information about SBCL, then exit.

`--version`
: Print SBCL's version information, then exit.

In the future, runtime options may be added to control behaviour such as lazy allocation of memory.

Runtime options, including any –end-runtime-options option, are stripped out of the command line before the Lisp toplevel logic gets a chance to see it.

2.3.2 Toplevel Options

`--sysinit` *filename*
: Load filename instead of the default system initialization file (see Section 2.4.1 [System Initialization File], page 13.)

`--no-sysinit`
: Don't load a system-wide initialization file. If this option is given, the `--sysinit` option is ignored.

`--userinit` *filename*
: Load filename instead of the default user initialization file (see Section 2.4.2 [User Initialization File], page 13.)

`--no-userinit`
: Don't load a user initialization file. If this option is given, the `--userinit` option is ignored.

`--eval` *command*
: After executing any initialization file, but before starting the read-eval-print loop on standard input, read and evaluate the com- mand given. More than one `--eval` option can be used, and all will be read and executed, in the order they appear on the command line.

`--load` *filename*
: This is equivalent to `--eval '(load "`*filename*`")'`. The special syntax is intended to reduce quoting headaches when invoking SBCL from shell scripts.

`--noprint`
: When ordinarily the toplevel "read-eval-print loop" would be exe- cuted, execute a "read-eval loop" instead, i.e. don't print a prompt and don't echo results. Combined with the `--noinform` runtime option, this makes it easier to write Lisp "scripts" which work cleanly in Unix pipelines.

`--disable-debugger`
: This is equivalent to `--eval '(sb-ext:disable-debugger)'`. See Section 4.1 [Debugger Entry], page 28.

2.4 Initialization Files

This section covers initialization files processed at startup, which can be used to customize the lisp environment.

2.4.1 System Initialization File

Site-wide startup script. Unless overridden with the command line option `--sysinit` defaults to '`SBCL_HOME/sbclrc`', or if that doesn't exist to '`/etc/sbclrc`'.

No system initialization file is required.

2.4.2 User Initialization File

Per-user startup script. Unless overridden with the command line option `--userinit` defaults to '`HOME/.sbclrc`'.

No user initialization file is required.

2.4.3 Initialization File Semantics

SBCL processes initialization files with `read` and `eval`, not `load`; hence initialization files can be used to set startup `*package*` and `*readtable*`, and for proclaiming a global optimization policy.

2.4.4 Initialization Examples

Some examples of what you may consider doing in the initialization files follow.

2.4.4.1 Unix-style Command Line Protocol

Standard Unix tools that are interpreters follow a common command line protocol that is necessary to work with "shebang scripts". SBCL doesn't do this by default, but adding the following snippet to an initialization file does the trick:

```
;;; If the first user-processable command-line argument is a filename,
;;; disable the debugger, load the file handling shebang-line and quit.
(let ((script (and (second *posix-argv*)
                   (probe-file (second *posix-argv*)))))
   (when script
      ;; Handle shebang-line
      (set-dispatch-macro-character #\# #\!
                                    (lambda (stream char arg)
                                       (declare (ignore char arg))
                                       (read-line stream)))
      ;; Disable debugger
      (setf *invoke-debugger-hook*
            (lambda (condition hook)
              (declare (ignore hook))
              ;; Uncomment to get backtraces on errors
              ;; (sb-debug:backtrace 20)
              (format *error-output* "Error: ~A~%" condition)
              (quit)))
      (load script)
      (quit)))
```

Example file ('`hello.lisp`'):

```
#!/usr/local/bin/sbcl --noinform
(write-line "Hello, World!")
```

Usage examples:

```
$ ./hello.lisp
Hello, World!
```

```
$ sbcl hello.lisp
This is SBCL 0.8.13.70, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
Hello, World!
```

2.4.4.2 Automatic Recompilation of Stale Fasls

SBCL fasl-format is at current stage of development undergoing non-backwards compatible changes fairly often. The following snippet handles recompilation automatically for ASDF-based systems.

```
(require :asdf)

;;; If a fasl was stale, try to recompile and load (once).
```

```
(defmethod asdf:perform :around ((o asdf:load-op)
                                 (c asdf:cl-source-file))
   (handler-case (call-next-method o c)
      ;; If a fasl was stale, try to recompile and load (once).
      (sb-ext:invalid-fasl ()
         (asdf:perform (make-instance 'asdf:compile-op) c)
         (call-next-method))))
```

3 Compiler

This chapter will discuss most compiler issues other than efficiency, including compiler error messages, the SBCL compiler's unusual approach to type safety in the presence of type declarations, the effects of various compiler optimization policies, and the way that inlining and open coding may cause optimized code to differ from a naive translation. Efficiency issues are sufficiently varied and separate that they have their own chapter, Chapter 5 [Efficiency], page 42.

3.1 Diagnostic Messages

3.1.1 Controlling Verbosity

The compiler can be quite verbose in its diagnostic reporting, rather more then some users would prefer – the amount of noise emitted can be controlled, however.

To control emission of compiler diagnostics (of any severity other than `error`: see Section 3.1.2 [Diagnostic Severity], page 17) use the `sb-ext:muffle-conditions` and `sb-ext:unmuffle-conditions` declarations, specifying the type of condition that is to be muffled (the muffling is done using an associated `muffle-warning` restart).

Global control:

```
;;; Muffle compiler-notes globally
(declaim (sb-ext:muffle-conditions sb-ext:compiler-note))
```

Local control:

```
;;; Muffle compiler-notes based on lexical scope
(defun foo (x)
  (declare (optimize speed) (fixnum x)
           (sb-ext:muffle-conditions sb-ext:compiler-note))
  (values (* x 5) ; no compiler note from this
    (locally
      (declare (sb-ext:unmuffle-conditions sb-ext:compiler-note))
      ;; this one gives a compiler note
      (* x -5))))
```

`sb-ext:muffle-conditions` [Declaration]

Syntax: type*

Muffles the diagnostic messages that would be caused by compile-time signals of given types.

`sb-ext:unmuffle-conditions` [Declaration]

Syntax: type*

Cancels the effect of a previous `sb-ext:muffle-condition` declaration.

Various details of *how* the compiler messages are printed can be controlled via the alist `sb-ext:*compiler-print-variable-alist*`.

`sb-ext:*compiler-print-variable-alist*` [Variable]

an association list describing new bindings for special variables to be used by the compiler for error-reporting, etc. Eg.

```
((*PRINT-LENGTH* . 10) (*PRINT-LEVEL* . 6) (*PRINT-PRETTY* . NIL))
```

The variables in the `car` positions are bound to the values in the `cdr` during the execution of some debug commands. When evaluating arbitrary expressions in the debugger, the normal values of the printer control variables are in effect.

Initially empty, `*compiler-print-variable-alist*` is Typically used to specify bindings for printer control variables.

3.1.2 Diagnostic Severity

There are four levels of compiler diagnostic severity:

1. error
2. warning
3. style warning
4. note

The first three levels correspond to condition classes which are defined in the ANSI standard for Common Lisp and which have special significance to the `compile` and `compile-file` functions. These levels of compiler error severity occur when the compiler handles conditions of these classes.

The fourth level of compiler error severity, *note*, corresponds to the `sb-ext:compiler-note`, and is used for problems which are too mild for the standard condition classes, typically hints about how efficiency might be improved. The `sb-ext:code-deletion-note`, a subtype of `compiler-note`, is signalled when the compiler deletes user-supplied code, usually after proving that the code in question is unreachable.

Future work for SBCL includes expanding this hierarchy of types to allow more fine-grained control over emission of diagnostic messages.

`sb-ext:compiler-note` [Condition]

Class precedence list: `compiler-note, condition, t`

Root of the hierarchy of conditions representing information discovered by the compiler that the user might wish to know, but which does not merit a `style-warning` (or any more serious condition).

`sb-ext:code-deletion-note` [Condition]

Class precedence list: `code-deletion-note, compiler-note, condition, t`

A condition type signalled when the compiler deletes code that the user has written, having proved that it is unreachable.

3.1.3 Understanding Compile Diagnostics

The messages emitted by the compiler contain a lot of detail in a terse format, so they may be confusing at first. The messages will be illustrated using this example program:

```
(defmacro zoq (x)
  `(roq (ploq (+ ,x 3))))

(defun foo (y)
  (declare (symbol y))
```

```
(zoq y))
```

The main problem with this program is that it is trying to add `3` to a symbol. Note also that the functions `roq` and `ploq` aren't defined anywhere.

3.1.3.1 The Parts of a Compiler Diagnostic

When processing this program, the compiler will produce this warning:

```
; file: /tmp/foo.lisp
; in: DEFUN FOO
;     (ZOQ Y)
; --> ROQ PLOQ
; ==>
;   (+ Y 3)
;
; caught WARNING:
;   Asserted type NUMBER conflicts with derived type (VALUES SYMBOL &OPTIONAL).
```

In this example we see each of the six possible parts of a compiler diagnostic:

1. '`file: /tmp/foo.lisp`' This is the name of the file that the compiler read the relevant code from. The file name is displayed because it may not be immediately obvious when there is an error during compilation of a large system, especially when `with-compilation-unit` is used to delay undefined warnings.
2. '`in: DEFUN FOO`' This is the definition top level form responsible for the diagnostic. It is obtained by taking the first two elements of the enclosing form whose first element is a symbol beginning with "'`def`"'. If there is no such enclosing "'`def`"' form, then the outermost form is used. If there are multiple '`def`' forms, then they are all printed from the outside in, separated by '`=>`''s. In this example, the problem was in the `defun` for `foo`.
3. '`(ZOQ Y)`' This is the *original source* form responsible for the diagnostic. Original source means that the form directly appeared in the original input to the compiler, i.e. in the lambda passed to `compile` or in the top level form read from the source file. In this example, the expansion of the `zoq` macro was responsible for the message.
4. '`--> ROQ PLOQ`' This is the *processing path* that the compiler used to produce the code that caused the message to be emitted. The processing path is a representation of the evaluated forms enclosing the actual source that the compiler encountered when processing the original source. The path is the first element of each form, or the form itself if the form is not a list. These forms result from the expansion of macros or source-to-source transformation done by the compiler. In this example, the enclosing evaluated forms are the calls to `roq` and `ploq`. These calls resulted from the expansion of the `zoq` macro.
5. '`==> (+ Y 3)`' This is the *actual source* responsible for the diagnostic. If the actual source appears in the explanation, then we print the next enclosing evaluated form, instead of printing the actual source twice. (This is the form that would otherwise have been the last form of the processing path.) In this example, the problem is with the evaluation of the reference to the variable `y`.
6. '`caught WARNING: Asserted type NUMBER conflicts with derived type (VALUES SYMBOL &OPTIONAL).`' This is the *explanation* of the problem. In this example, the

problem is that, while the call to `+` requires that its arguments are all of type `number`, the compiler has derived that `y` will evaluate to a `symbol`. Note that '`(VALUES SYMBOL &OPTIONAL)`' expresses that `y` evaluates to precisely one value.

Note that each part of the message is distinctively marked:

- '`file:`' and '`in:`' mark the file and definition, respectively.
- The original source is an indented form with no prefix.
- Each line of the processing path is prefixed with '`-->`'
- The actual source form is indented like the original source, but is marked by a preceding '`==>`' line.
- The explanation is prefixed with the diagnostic severity, which can be '`caught ERROR:`', '`caught WARNING:`', '`caught STYLE-WARNING:`', or '`note:`'.

Each part of the message is more specific than the preceding one. If consecutive messages are for nearby locations, then the front part of the messages would be the same. In this case, the compiler omits as much of the second message as in common with the first. For example:

```
; file: /tmp/foo.lisp
; in: DEFUN FOO
;     (ZOQ Y)
; --> ROQ
; ==>
;   (PLOQ (+ Y 3))
;
; caught STYLE-WARNING:
;   undefined function: PLOQ

; ==>
;   (ROQ (PLOQ (+ Y 3)))
;
; caught STYLE-WARNING:
;   undefined function: ROQ
```

In this example, the file, definition and original source are identical for the two messages, so the compiler omits them in the second message. If consecutive messages are entirely identical, then the compiler prints only the first message, followed by: '`[Last message occurs` *`repeats`* `times]`' where *repeats* is the number of times the message was given.

If the source was not from a file, then no file line is printed. If the actual source is the same as the original source, then the processing path and actual source will be omitted. If no forms intervene between the original source and the actual source, then the processing path will also be omitted.

3.1.3.2 The Original and Actual Source

The *original source* displayed will almost always be a list. If the actual source for an message is a symbol, the original source will be the immediately enclosing evaluated list form. So even if the offending symbol does appear in the original source, the compiler will print the

enclosing list and then print the symbol as the actual source (as though the symbol were introduced by a macro.)

When the *actual source* is displayed (and is not a symbol), it will always be code that resulted from the expansion of a macro or a source-to-source compiler optimization. This is code that did not appear in the original source program; it was introduced by the compiler.

Keep in mind that when the compiler displays a source form in an diagnostic message, it always displays the most specific (innermost) responsible form. For example, compiling this function

```
(defun bar (x)
  (let (a)
    (declare (fixnum a))
    (setq a (foo x))
    a))
```

gives this error message

```
; file: /tmp/foo.lisp
; in: DEFUN BAR
;     (LET (A)
;     (DECLARE (FIXNUM A))
;     (SETQ A (FOO X))
;     A)
;
; caught WARNING:
;   Asserted type FIXNUM conflicts with derived type (VALUES NULL &OPTIONAL).
```

This message is not saying "there is a problem somewhere in this `let`" – it is saying that there is a problem with the `let` itself. In this example, the problem is that `a`'s `nil` initial value is not a `fixnum`.

3.1.3.3 The Processing Path

The processing path is mainly useful for debugging macros, so if you don't write macros, you can probably ignore it. Consider this example:

```
(defun foo (n)
  (dotimes (i n *undefined*)))
```

Compiling results in this error message:

```
; in: DEFUN FOO
;     (DOTIMES (I N *UNDEFINED*))
; --> DO BLOCK LET TAGBODY RETURN-FROM
; ==>
;   (PROGN *UNDEFINED*)
;
; caught WARNING:
;   undefined variable: *UNDEFINED*
```

Note that `do` appears in the processing path. This is because `dotimes` expands into:

```
(do ((i 0 (1+ i)) (#:g1 n))
    ((>= i #:g1) *undefined*)
```

```
  (declare (type unsigned-byte i)))
```

The rest of the processing path results from the expansion of `do`:

```
(block nil
  (let ((i 0) (#:g1 n))
    (declare (type unsigned-byte i))
    (tagbody (go #:g3)
      #:g2    (psetq i (1+ i))
      #:g3    (unless (>= i #:g1) (go #:g2))
      (return-from nil (progn *undefined*)))))
```

In this example, the compiler descended into the `block`, `let`, `tagbody` and `return-from` to reach the `progn` printed as the actual source. This is a place where the "actual source appears in explanation" rule was applied. The innermost actual source form was the symbol `*undefined*` itself, but that also appeared in the explanation, so the compiler backed out one level.

3.2 Handling of Types

The most unusual features of the SBCL compiler (which is very similar to the original CMUCL compiler, also known as *Python*) is its unusually sophisticated understanding of the Common Lisp type system and its unusually conservative approach to the implementation of type declarations.

These two features reward the use of type declarations throughout development, even when high performance is not a concern. Also, as discussed in the chapter on performance (see Chapter 5 [Efficiency], page 42), the use of appropriate type declarations can be very important for performance as well.

The SBCL compiler also has a greater knowledge of the Common Lisp type system than other compilers. Support is incomplete only for types involving the `satisfies` type specifier.

3.2.1 Declarations as Assertions

The SBCL compiler treats type declarations differently from most other Lisp compilers. Under default compilation policy the compiler doesn't blindly believe type declarations, but considers them assertions about the program that should be checked: all type declarations that have not been proven to always hold are asserted at runtime.

> *Remaining bugs in the compiler's handling of types unfortunately provide some exceptions to this rule, see Section 3.2.4 [Implementation Limitations], page 24).*

There are three type checking policies available in SBCL, selectable via `optimize` declarations.

Full Type Checks

> All declarations are considered assertions to be checked at runtime, and all type checks are precise.
>
> Used when `(and (< 0 safety) (or (>= safety 2) (>= safety speed)))`. The default compilation policy provides full type checks.

Weak Type Checks
: Any or all type declarations may be believed without runtime assertions, and assertions that are done may be imprecise. It should be noted that it is relatively easy to corrupt the heap when weak type checks are used, and type-errors are introduced into the program.

  Used when `(and (< safety 2) (< safety speed))`

No Type Checks
: All declarations are believed without assertions. Also disables argument count and array bounds checking.

  Used when `(= safety 0)`.

3.2.2 Precise Type Checking

Precise checking means that the check is done as though `typep` had been called with the exact type specifier that appeared in the declaration.

If a variable is declared to be `(integer 3 17)` then its value must always be an integer between `3` and `17`. If multiple type declarations apply to a single variable, then all the declarations must be correct; it is as though all the types were intersected producing a single `and` type specifier.

To gain maximum benefit from the compiler's type checking, you should always declare the types of function arguments and structure slots as precisely as possible. This often involves the use of `or`, `member`, and other list-style type specifiers.

3.2.3 Getting Existing Programs to Run

Since SBCL's compiler does much more comprehensive type checking than most Lisp compilers, SBCL may detect type errors in programs that have been debugged using other compilers. These errors are mostly incorrect declarations, although compile-time type errors can find actual bugs if parts of the program have never been tested.

Some incorrect declarations can only be detected by run-time type checking. It is very important to initially compile a program with full type checks (high `safety` optimization) and then test this safe version. After the checking version has been tested, then you can consider weakening or eliminating type checks. *This applies even to previously debugged programs,* because the SBCL compiler does much more type inference than other Common Lisp compilers, so an incorrect declaration can do more damage.

The most common problem is with variables whose constant initial value doesn't match the type declaration. Incorrect constant initial values will always be flagged by a compile-time type error, and they are simple to fix once located. Consider this code fragment:

```
(prog (foo)
  (declare (fixnum foo))
  (setq foo ...)
  ...)
```

Here `foo` is given an initial value of `nil`, but is declared to be a `fixnum`. Even if it is never read, the initial value of a variable must match the declared type. There are two ways to fix this problem. Change the declaration

```
(prog (foo)
```

```
(declare (type (or fixnum null) foo))
(setq foo ...)
...)
```

or change the initial value

```
(prog ((foo 0))
  (declare (fixnum foo))
  (setq foo ...)
  ...)
```

It is generally preferable to change to a legal initial value rather than to weaken the declaration, but sometimes it is simpler to weaken the declaration than to try to make an initial value of the appropriate type.

Another declaration problem occasionally encountered is incorrect declarations on `defmacro` arguments. This can happen when a function is converted into a macro. Consider this macro:

```
(defmacro my-1+ (x)
  (declare (fixnum x))
  `(the fixnum (1+ ,x)))
```

Although legal and well-defined Common Lisp code, this meaning of this definition is almost certainly not what the writer intended. For example, this call is illegal:

```
(my-1+ (+ 4 5))
```

This call is illegal because the argument to the macro is `(+ 4 5)`, which is a `list`, not a `fixnum`. Because of macro semantics, it is hardly ever useful to declare the types of macro arguments. If you really want to assert something about the type of the result of evaluating a macro argument, then put a `the` in the expansion:

```
(defmacro my-1+ (x)
  `(the fixnum (1+ (the fixnum ,x))))
```

In this case, it would be stylistically preferable to change this macro back to a function and declare it inline.

Some more subtle problems are caused by incorrect declarations that can't be detected at compile time. Consider this code:

```
(do ((pos 0 (position #\a string :start (1+ pos))))
  ((null pos))
  (declare (fixnum pos))
  ...)
```

Although `pos` is almost always a `fixnum`, it is `nil` at the end of the loop. If this example is compiled with full type checks (the default), then running it will signal a type error at the end of the loop. If compiled without type checks, the program will go into an infinite loop (or perhaps `position` will complain because `(1+ nil)` isn't a sensible start.) Why? Because if you compile without type checks, the compiler just quietly believes the type declaration. Since the compiler believes that `pos` is always a `fixnum`, it believes that `pos` is never `nil`, so `(null pos)` is never true, and the loop exit test is optimized away. Such errors are sometimes flagged by unreachable code notes, but it is still important to initially compile and test any system with full type checks, even if the system works fine when compiled using other compilers.

In this case, the fix is to weaken the type declaration to `(or fixnum null)`[1].

Note that there is usually little performance penalty for weakening a declaration in this way. Any numeric operations in the body can still assume that the variable is a `fixnum`, since `nil` is not a legal numeric argument. Another possible fix would be to say:

```
(do ((pos 0 (position #\a string :start (1+ pos))))
    ((null pos))
  (let ((pos pos))
    (declare (fixnum pos))
    ...))
```

This would be preferable in some circumstances, since it would allow a non-standard representation to be used for the local `pos` variable in the loop body.

3.2.4 Implementation Limitations

Ideally, the compiler would consider *all* type declarations to be assertions, so that adding type declarations to a program, no matter how incorrect they might be, would *never* cause undefined behavior. However, the compiler is known to fall short of this goal in two areas:

- *Proclaimed* constraints on argument and result types of a function are supposed to be checked by the function. If the function type is proclaimed before function definition, type checks are inserted by the compiler, but the standard allows the reversed order, in which case the compiler will trust the declaration.
- The compiler cannot check types of an unknown number of values; if the number of generated values is unknown, but the number of consumed is known, only consumed values are checked.

  For example,

  ```
  (defun foo (x)
    (the integer (bar x)))
  ```

  causes the following compiler diagnostic to be emitted:

  ```
  ; note: type assertion too complex to check:
  ;  (VALUES INTEGER &REST T).
  ```

  A partial workaround is instead write:

  ```
  (defun foo (x)
    (the (values integer &optional) (bar x)))
  ```

These are important issues, but are not necessarily easy to fix, so they may, alas, remain in the system for a while.

3.3 Compiler Policy

Compiler policy is controlled by the `optimize` declaration, supporting all ANSI optimization qualities (`debug`, `safety`, `space`, and `speed`).[2]

---

[1] Actually, this declaration is unnecessary in SBCL, since it already knows that `position` returns a non-negative `fixnum` or `nil`.

[2] A deprecated extension `sb-ext:inhibit-warnings` is still supported, but liable to go away at any time.

For effects of various optimization qualities on type-safety and debuggability see Section 3.2.1 [Declarations as Assertions], page 21 and Section 4.6 [Debugger Policy Control], page 37.

Ordinarily, when the `speed` quality is high, the compiler emits notes to notify the programmer about its inability to apply various optimizations. For selective muffling of these notes See Section 3.1.1 [Controlling Verbosity], page 16.

The value of `space` mostly influences the compiler's decision whether to inline operations, which tend to increase the size of programs. Use the value `0` with caution, since it can cause the compiler to inline operations so indiscriminately that the net effect is to slow the program by causing cache misses or even swapping.

3.4 Compiler Errors

3.4.1 Type Errors at Compile Time

If the compiler can prove at compile time that some portion of the program cannot be executed without a type error, then it will give a warning at compile time.

It is possible that the offending code would never actually be executed at run-time due to some higher level consistency constraint unknown to the compiler, so a type warning doesn't always indicate an incorrect program.

For example, consider this code fragment:

```
(defun raz (foo)
  (let ((x (case foo
             (:this 13)
             (:that 9)
             (:the-other 42))))
    (declare (fixnum x))
    (foo x)))
```

Compilation produces this warning:

```
; in: DEFUN RAZ
;     (CASE FOO (:THIS 13) (:THAT 9) (:THE-OTHER 42))
; --> LET COND IF COND IF COND IF
; ==>
;   (COND)
;
; caught WARNING:
;   This is not a FIXNUM:
;   NIL
```

In this case, the warning means that if `foo` isn't any of `:this`, `:that` or `:the-other`, then `x` will be initialized to `nil`, which the `fixnum` declaration makes illegal. The warning will go away if `ecase` is used instead of `case`, or if `:the-other` is changed to `t`.

This sort of spurious type warning happens moderately often in the expansion of complex macros and in inline functions. In such cases, there may be dead code that is impossible to correctly execute. The compiler can't always prove this code is dead (could never be executed), so it compiles the erroneous code (which will always signal an error if it is executed) and gives a warning.

3.4.2 Errors During Macroexpansion

The compiler handles errors that happen during macroexpansion, turning them into compiler errors. If you want to debug the error (to debug a macro), you can set `*break-on-signals*` to `error`. For example, this definition:

```
(defun foo (e l)
  (do ((current l (cdr current))
       ((atom current) nil))
      (when (eq (car current) e) (return current))))
```

gives this error:

```
; in: DEFUN FOO
;     (DO ((CURRENT L (CDR CURRENT))
;          ((ATOM CURRENT) NIL))
;         (WHEN (EQ (CAR CURRENT) E) (RETURN CURRENT)))
;
; caught ERROR:
;   (in macroexpansion of (DO # #))
;   (hint: For more precise location, try *BREAK-ON-SIGNALS*.)
;   DO step variable is not a symbol: (ATOM CURRENT)
```

3.4.3 Read Errors

SBCL's compiler does not attempt to recover from read errors when reading a source file, but instead just reports the offending character position and gives up on the entire source file.

3.5 Open Coding and Inline Expansion

Since Common Lisp forbids the redefinition of standard functions, the compiler can have special knowledge of these standard functions embedded in it. This special knowledge is used in various ways (open coding, inline expansion, source transformation), but the implications to the user are basically the same:

- Attempts to redefine standard functions may be frustrated, since the function may never be called. Although it is technically illegal to redefine standard functions, users sometimes want to implicitly redefine these functions when they are debugging using the `trace` macro. Special-casing of standard functions can be inhibited using the `notinline` declaration, but even then some phases of analysis such as type inferencing are applied by the compiler.
- The compiler can have multiple alternate implementations of standard functions that implement different trade-offs of speed, space and safety. This selection is based on the compiler policy, Section 3.3 [Compiler Policy], page 24.

When a function call is *open coded*, inline code whose effect is equivalent to the function call is substituted for that function call. When a function call is *closed coded*, it is usually left as is, although it might be turned into a call to a different function with different arguments. As an example, if `nthcdr` were to be open coded, then

```
(nthcdr 4 foobar)
```

might turn into

```
(cdr (cdr (cdr (cdr foobar))))
```

or even

```
(do ((i 0 (1+ i))
  (list foobar (cdr foobar)))
  ((= i 4) list))
```

If `nth` is closed coded, then

```
(nth x l)
```

might stay the same, or turn into something like

```
(car (nthcdr x l))
```

In general, open coding sacrifices space for speed, but some functions (such as `car`) are so simple that they are always open-coded. Even when not open-coded, a call to a standard function may be transformed into a different function call (as in the last example) or compiled as *static call*. Static function call uses a more efficient calling convention that forbids redefinition.

3.6 Interpreter

By default SBCL implements `eval` by calling the native code compiler. SBCL also includes an interpreter for use in special cases where using the compiler is undesirable, for example due to compilation overhead. Unlike in some other Lisp implementations, in SBCL interpreted code is not safer or more debuggable than compiled code.

Switching between the compiler and the interpreter is done using the special variable `sb-ext:*evaluator-mode*`. As of 0.9.17, valid values for `sb-ext:*evaluator-mode*` are `:compile` and `:interpret`.

4 Debugger

This chapter documents the debugging facilities of SBCL, including the debugger, single-stepper and `trace`, and the effect of `(optimize debug)` declarations.

4.1 Debugger Entry

4.1.1 Debugger Banner

When you enter the debugger, it looks something like this:

```
debugger invoked on a TYPE-ERROR in thread 11184:
  The value 3 is not of type LIST.

You can type HELP for debugger help, or (SB-EXT:QUIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT   ] Reduce debugger level (leaving debugger, returning to toplevel).
  1: [TOPLEVEL] Restart at toplevel READ/EVAL/PRINT loop.
(CAR 1 3)
0]
```

The first group of lines describe what the error was that put us in the debugger. In this case `car` was called on `3`, causing a `type-error`.

This is followed by the "beginner help line", which appears only if `sb-ext:*debugger-beginner-help*` is true (default).

Next comes a listing of the active restart names, along with their descriptions – the ways we can restart execution after this error. In this case, both options return to top-level. Restarts can be selected by entering the corresponding number or name.

The current frame appears right underneath the restarts, immediately followed by the debugger prompt.

4.1.2 Debugger Invocation

The debugger is invoked when:

- `error` is called, and the condition it signals is not handled.
- `break` is called, or `signal` is called with a condition that matches the current `*break-on-signals*`.
- the debugger is explicitly entered with the `invoke-debugger` function.

When the debugger is invoked by a condition, ANSI mandates that the value of `*debugger-hook*`, if any, be called with two arguments: the condition that caused the debugger to be invoked and the previous value of `*debugger-hook*`. When this happens, `*debugger-hook*` is bound to NIL to prevent recursive errors. However, ANSI also mandates that `*debugger-hook*` not be invoked when the debugger is to be entered by the `break` function. For users who wish to provide an alternate debugger interface (and thus catch `break` entries into the debugger), SBCL provides `sb-ext:*invoke-debugger-hook*`, which is invoked during any entry into the debugger.

`sb-ext:*invoke-debugger-hook*` [Variable]

This is either `nil` or a designator for a function of two arguments, to be run when the debugger is about to be entered. The function is run with `*invoke-debugger-hook*` bound to `nil` to minimize recursive errors, and receives as arguments the condition that triggered debugger entry and the previous value of `*invoke-debugger-hook*`

This mechanism is an `sbcl` extension similar to the standard `*debugger-hook*`. In contrast to `*debugger-hook*`, it is observed by `invoke-debugger` even when called by `break`.

4.2 Debugger Command Loop

The debugger is an interactive read-eval-print loop much like the normal top level, but some symbols are interpreted as debugger commands instead of being evaluated. A debugger command starts with the symbol name of the command, possibly followed by some arguments on the same line. Some commands prompt for additional input. Debugger commands can be abbreviated by any unambiguous prefix: `help` can be typed as '`h`', '`he`', etc.

The package is not significant in debugger commands; any symbol with the name of a debugger command will work. If you want to show the value of a variable that happens also to be the name of a debugger command you can wrap the variable in a `progn` to hide it from the command loop.

The debugger prompt is "*`frame`*`]`", where *frame* is the number of the current frame. Frames are numbered starting from zero at the top (most recent call), increasing down to the bottom. The current frame is the frame that commands refer to.

It is possible to override the normal printing behaviour in the debugger by using the `sb-ext:*debug-print-variable-alist*`.

`sb-ext:*debug-print-variable-alist*` [Variable]

an association list describing new bindings for special variables to be used within the debugger. Eg.

```
((*PRINT-LENGTH* . 10) (*PRINT-LEVEL* . 6) (*PRINT-PRETTY* . NIL))
```

The variables in the `car` positions are bound to the values in the `cdr` during the execution of some debug commands. When evaluating arbitrary expressions in the debugger, the normal values of the printer control variables are in effect.

Initially empty, `*debug-print-variable-alist*` is typically used to provide bindings for printer control variables.

4.3 Stack Frames

A *stack frame* is the run-time representation of a call to a function; the frame stores the state that a function needs to remember what it is doing. Frames have:

- *variables* (see Section 4.4 [Variable Access], page 32), which are the values being operated on.
- *arguments* to the call (which are really just particularly interesting variables).
- a current source location (see Section 4.5 [Source Location Printing], page 34), which is the place in the program where the function was running when it stopped to call another function, or because of an interrupt or error.

4.3.1 Stack Motion

These commands move to a new stack frame and print the name of the function and the values of its arguments in the style of a Lisp function call:

`up` [Debugger Command]
: Move up to the next higher frame. More recent function calls are considered to be higher on the stack.

`down` [Debugger Command]
: Move down to the next lower frame.

`top` [Debugger Command]
: Move to the highest frame, that is, the frame where the debugger was entered.

`bottom` [Debugger Command]
: Move to the lowest frame.

`frame` [*n*] [Debugger Command]
: Move to the frame with the specified number. Prompts for the number if not supplied. The frame with number 0 is the frame where the debugger was entered.

4.3.2 How Arguments are Printed

A frame is printed to look like a function call, but with the actual argument values in the argument positions. So the frame for this call in the source:

```
(myfun (+ 3 4) 'a)
```

would look like this:

```
(MYFUN 7 A)
```

All keyword and optional arguments are displayed with their actual values; if the corresponding argument was not supplied, the value will be the default. So this call:

```
(subseq "foo" 1)
```

would look like this:

```
(SUBSEQ "foo" 1 3)
```

And this call:

```
(string-upcase "test case")
```

would look like this:

```
(STRING-UPCASE "test case" :START 0 :END NIL)
```

The arguments to a function call are displayed by accessing the argument variables. Although those variables are initialized to the actual argument values, they can be set inside the function; in this case the new value will be displayed.

`&rest` arguments are handled somewhat differently. The value of the rest argument variable is displayed as the spread-out arguments to the call, so:

```
(format t "~A is a ~A." "This" 'test)
```

would look like this:

```
(FORMAT T "~A is a ~A." "This" 'TEST)
```

Rest arguments cause an exception to the normal display of keyword arguments in functions that have both `&rest` and `&key` arguments. In this case, the keyword argument variables are not displayed at all; the rest arg is displayed instead. So for these functions, only the keywords actually supplied will be shown, and the values displayed will be the argument values, not values of the (possibly modified) variables.

If the variable for an argument is never referenced by the function, it will be deleted. The variable value is then unavailable, so the debugger prints '`#<unused-arg>`' instead of the value. Similarly, if for any of a number of reasons the value of the variable is unavailable or not known to be available (see Section 4.4 [Variable Access], page 32), then '`#<unavailable-arg>`' will be printed instead of the argument value.

Note that inline expansion and open-coding affect what frames are present in the debugger, see Section 4.6 [Debugger Policy Control], page 37.

4.3.3 Function Names

If a function is defined by `defun` it will appear in backtrace by that name. Functions defined by `labels` and `flet` will appear as `(FLET <name>)` and `(LABELS <name>)` respectively. Anonymous lambdas will appear as `(LAMDBA <lambda-list>)`.

4.3.3.1 Entry Point Details

Sometimes the compiler introduces new functions that are used to implement a user function, but are not directly specified in the source. This is mostly done for argument type and count checking.

The debugger will normally show these entry point functions as if they were the normal main entry point, but more detail can be obtained by setting `sb-debug:*show-entry-point-details*` to true; this is primarily useful for debugging SBCL itself, but may help pinpoint problems that occur during lambda-list processing.

With recursive functions, an additional `:EXTERNAL` frame may appear before the frame representing the first call to the recursive function. This is a consequence of the way the compiler works: there is nothing odd with your program. You will also see `:CLEANUP` frames during the execution of `unwind-protect` cleanup code. The `:EXTERNAL` and `:CLEANUP` above are entry-point types, visible only if `sb-debug:*show-entry-point-details*` os true.

4.3.4 Debug Tail Recursion

The compiler is "properly tail recursive." If a function call is in a tail-recursive position, the stack frame will be deallocated *at the time of the call*, rather than after the call returns. Consider this backtrace:

```
(BAR ...)
(FOO ...)
```

Because of tail recursion, it is not necessarily the case that `FOO` directly called `BAR`. It may be that `FOO` called some other function `FOO2` which then called `BAR` tail-recursively, as in this example:

```
(defun foo ()
  ...
  (foo2 ...)
```

```
  ...)

(defun foo2 (...)
  ...
  (bar ...))

(defun bar (...)
  ...)
```

Usually the elimination of tail-recursive frames makes debugging more pleasant, since these frames are mostly uninformative. If there is any doubt about how one function called another, it can usually be eliminated by finding the source location in the calling frame. See Section 4.5 [Source Location Printing], page 34.

The elimination of tail-recursive frames can be prevented by disabling tail-recursion optimization, which happens when the `debug` optimization quality is greater than `2`. See Section 4.6 [Debugger Policy Control], page 37.

4.3.5 Unknown Locations and Interrupts

The debugger operates using special debugging information attached to the compiled code. This debug information tells the debugger what it needs to know about the locations in the code where the debugger can be invoked. If the debugger somehow encounters a location not described in the debug information, then it is said to be *unknown*. If the code location for a frame is unknown, then some variables may be inaccessible, and the source location cannot be precisely displayed.

There are three reasons why a code location could be unknown:

- There is inadequate debug information due to the value of the `debug` optimization quality. See Section 4.6 [Debugger Policy Control], page 37.
- The debugger was entered because of an interrupt such as ⟨C-c⟩.
- A hardware error such as "`bus error`" occurred in code that was compiled unsafely due to the value of the `safety` optimization quality.

In the last two cases, the values of argument variables are accessible, but may be incorrect. For more details on when variable values are accessible, Section 4.4.1 [Variable Value Availability], page 33.

It is possible for an interrupt to happen when a function call or return is in progress. The debugger may then flame out with some obscure error or insist that the bottom of the stack has been reached, when the real problem is that the current stack frame can't be located. If this happens, return from the interrupt and try again.

4.4 Variable Access

There are two ways to access the current frame's local variables in the debugger: `list-locals` and `sb-debug:var`.

The debugger doesn't really understand lexical scoping; it has just one namespace for all the variables in the current stack frame. If a symbol is the name of multiple variables in the same function, then the reference appears ambiguous, even though lexical scoping specifies which value is visible at any given source location. If the scopes of the two variables are not

nested, then the debugger can resolve the ambiguity by observing that only one variable is accessible.

When there are ambiguous variables, the evaluator assigns each one a small integer identifier. The `sb-debug:var` function uses this identifier to distinguish between ambiguous variables. The `list-locals` command prints the identifier. In the following example, there are two variables named `X`. The first one has identifier 0 (which is not printed), the second one has identifier 1.

```
X  =  1
X#1  =  2
```

`list-locals` [*prefix*] [Debugger Command]
: This command prints the name and value of all variables in the current frame whose name has the specified *prefix*. *prefix* may be a string or a symbol. If no *prefix* is given, then all available variables are printed. If a variable has a potentially ambiguous name, then the name is printed with a "`#`*identifier*" suffix, where *identifier* is the small integer used to make the name unique.

`sb-debug:var` *name* **&optional** *identifier* [Function]
: This function returns the value of the variable in the current frame with the specified *name*. If supplied, *identifier* determines which value to return when there are ambiguous variables.

  When *name* is a symbol, it is interpreted as the symbol name of the variable, i.e. the package is significant. If *name* is an uninterned symbol (gensym), then return the value of the uninterned variable with the same name. If *name* is a string, `sb-debug:var` interprets it as the prefix of a variable name that must unambiguously complete to the name of a valid variable.

  identifier is used to disambiguate the variable name; use `list-locals` to find out the identifiers.

4.4.1 Variable Value Availability

The value of a variable may be unavailable to the debugger in portions of the program where Lisp says that the variable is defined. If a variable value is not available, the debugger will not let you read or write that variable. With one exception, the debugger will never display an incorrect value for a variable. Rather than displaying incorrect values, the debugger tells you the value is unavailable.

The one exception is this: if you interrupt (e.g., with ⟨C-c⟩) or if there is an unexpected hardware error such as "'`bus error`'" (which should only happen in unsafe code), then the values displayed for arguments to the interrupted frame might be incorrect.[1] This exception applies only to the interrupted frame: any frame farther down the stack will be fine.

The value of a variable may be unavailable for these reasons:

- The value of the `debug` optimization quality may have omitted debug information needed to determine whether the variable is available. Unless a variable is an argument, its value will only be available when `debug` is at least `2`.

[1] Since the location of an interrupt or hardware error will always be an unknown location, non-argument variable values will never be available in the interrupted frame. See Section 4.3.5 [Unknown Locations and Interrupts], page 32.

- The compiler did lifetime analysis and determined that the value was no longer needed, even though its scope had not been exited. Lifetime analysis is inhibited when the `debug` optimization quality is `3`.
- The variable's name is an uninterned symbol (gensym). To save space, the compiler only dumps debug information about uninterned variables when the `debug` optimization quality is `3`.
- The frame's location is unknown (see Section 4.3.5 [Unknown Locations and Interrupts], page 32) because the debugger was entered due to an interrupt or unexpected hardware error. Under these conditions the values of arguments will be available, but might be incorrect. This is the exception mentioned above.
- The variable (or the code referencing it) was optimized out of existence. Variables with no reads are always optimized away. The degree to which the compiler deletes variables will depend on the value of the `compilation-speed` optimization quality, but most source-level optimizations are done under all compilation policies.
- The variable is never set and its definition looks like

```
(LET ((var1 var2))
  ...)
```

  In this case, `var1` is substituted with `var2`.
- The variable is never set and is referenced exactly once. In this case, the reference is substituted with the variable initial value.

Since it is especially useful to be able to get the arguments to a function, argument variables are treated specially when the `speed` optimization quality is less than `3` and the `debug` quality is at least `1`. With this compilation policy, the values of argument variables are almost always available everywhere in the function, even at unknown locations. For non-argument variables, `debug` must be at least `2` for values to be available, and even then, values are only available at known locations.

4.4.2 Note On Lexical Variable Access

When the debugger command loop establishes variable bindings for available variables, these variable bindings have lexical scope and dynamic extent.[2] You can close over them, but such closures can't be used as upward funargs.

You can also set local variables using `setq`, but if the variable was closed over in the original source and never set, then setting the variable in the debugger may not change the value in all the functions the variable is defined in. Another risk of setting variables is that you may assign a value of a type that the compiler proved the variable could never take on. This may result in bad things happening.

4.5 Source Location Printing

One of the debugger's capabilities is source level debugging of compiled code. These commands display the source location for the current frame:

`source` [*context*] [Debugger Command]

This command displays the file that the current frame's function was defined from (if it was defined from a file), and then the source form responsible for generating

[2] The variable bindings are actually created using the Lisp `symbol-macrolet` special form.

the code that the current frame was executing. If *context* is specified, then it is an integer specifying the number of enclosing levels of list structure to print.

The source form for a location in the code is the innermost list present in the original source that encloses the form responsible for generating that code. If the actual source form is not a list, then some enclosing list will be printed. For example, if the source form was a reference to the variable `*some-random-special*`, then the innermost enclosing evaluated form will be printed. Here are some possible enclosing forms:

```
(let ((a *some-random-special*))
  ...)

(+ *some-random-special* ...)
```

If the code at a location was generated from the expansion of a macro or a source-level compiler optimization, then the form in the original source that expanded into that code will be printed. Suppose the file '`/usr/me/mystuff.lisp`' looked like this:

```
(defmacro mymac ()
  '(myfun))

(defun foo ()
  (mymac)
  ...)
```

If `foo` has called `myfun`, and is waiting for it to return, then the `source` command would print:

```
; File: /usr/me/mystuff.lisp

(MYMAC)
```

Note that the macro use was printed, not the actual function call form, `(myfun)`.

If enclosing source is printed by giving an argument to `source` or `vsource`, then the actual source form is marked by wrapping it in a list whose first element is '`#:***HERE***`'. In the previous example, `source 1` would print:

```
; File: /usr/me/mystuff.lisp

(DEFUN FOO ()
  (#:***HERE***
   (MYMAC))
  ...)
```

4.5.1 How the Source is Found

If the code was defined from Lisp by `compile` or `eval`, then the source can always be reliably located. If the code was defined from a '`fasl`' file created by `compile-file`, then the debugger gets the source forms it prints by reading them from the original source file. This is a potential problem, since the source file might have moved or changed since the time it was compiled.

The source file is opened using the `truename` of the source file pathname originally given to the compiler. This is an absolute pathname with all logical names and symbolic links

expanded. If the file can't be located using this name, then the debugger gives up and signals an error.

If the source file can be found, but has been modified since the time it was compiled, the debugger prints this warning:

```
; File has been modified since compilation:
;   filename
; Using form offset instead of character position.
```

where *filename* is the name of the source file. It then proceeds using a robust but not foolproof heuristic for locating the source. This heuristic works if:

- No top-level forms before the top-level form containing the source have been added or deleted, and
- The top-level form containing the source has not been modified much. (More precisely, none of the list forms beginning before the source form have been added or deleted.)

If the heuristic doesn't work, the displayed source will be wrong, but will probably be near the actual source. If the "shape" of the top-level form in the source file is too different from the original form, then an error will be signaled. When the heuristic is used, the source location commands are noticeably slowed.

Source location printing can also be confused if (after the source was compiled) a read-macro you used in the code was redefined to expand into something different, or if a read-macro ever returns the same `eq` list twice. If you don't define read macros and don't use `##` in perverted ways, you don't need to worry about this.

4.5.2 Source Location Availability

Source location information is only available when the `debug` optimization quality is at least `2`. If source location information is unavailable, the source commands will give an error message.

If source location information is available, but the source location is unknown because of an interrupt or unexpected hardware error (see Section 4.3.5 [Unknown Locations and Interrupts], page 32), then the command will print:

```
Unknown location: using block start.
```

and then proceed to print the source location for the start of the *basic block* enclosing the code location. It's a bit complicated to explain exactly what a basic block is, but here are some properties of the block start location:

- The block start location may be the same as the true location.
- The block start location will never be later in the program's flow of control than the true location.
- No conditional control structures (such as `if`, `cond`, `or`) will intervene between the block start and the true location (but note that some conditionals present in the original source could be optimized away.) Function calls *do not* end basic blocks.
- The head of a loop will be the start of a block.
- The programming language concept of "block structure" and the Lisp `block` special form are totally unrelated to the compiler's basic block.

In other words, the true location lies between the printed location and the next conditional (but watch out because the compiler may have changed the program on you.)

4.6 Debugger Policy Control

The compilation policy specified by `optimize` declarations affects the behavior seen in the debugger. The `debug` quality directly affects the debugger by controlling the amount of debugger information dumped. Other optimization qualities have indirect but observable effects due to changes in the way compilation is done.

Unlike the other optimization qualities (which are compared in relative value to evaluate tradeoffs), the `debug` optimization quality is directly translated to a level of debug information. This absolute interpretation allows the user to count on a particular amount of debug information being available even when the values of the other qualities are changed during compilation. These are the levels of debug information that correspond to the values of the `debug` quality:

`0` Only the function name and enough information to allow the stack to be parsed.

`> 0` Any level greater than `0` gives level `0` plus all argument variables. Values will only be accessible if the argument variable is never set and `speed` is not `3`. SBCL allows any real value for optimization qualities. It may be useful to specify `0.5` to get backtrace argument display without argument documentation.

`1` Level `1` provides argument documentation (printed arglists) and derived argument/result type information. This makes `describe` more informative, and allows the compiler to do compile-time argument count and type checking for any calls compiled at run-time. This is the default.

`2` Level `1` plus all interned local variables, source location information, and lifetime information that tells the debugger when arguments are available (even when `speed` is `3` or the argument is set).

`> 2` Any level greater than `2` gives level `2` and in addition disables tail-call optimization, so that the backtrace will contain frames for all invoked functions, even those in tail positions.

`3` Level `2` plus all uninterned variables. In addition, lifetime analysis is disabled (even when `speed` is `3`), ensuring that all variable values are available at any known location within the scope of the binding. This has a speed penalty in addition to the obvious space penalty.

`> (max speed space)`
If `debug` is greater than both `speed` and `space`, the command `return` can be used to continue execution by returning a value from the current stack frame.

`> (max speed space compilation-speed)`
If `debug` is greater than all of `speed`, `space` and `compilation-speed` the code will be steppable (see Section 4.10 [Single Stepping], page 41).

As you can see, if the `speed` quality is `3`, debugger performance is degraded. This effect comes from the elimination of argument variable special-casing (see Section 4.4.1 [Variable Value Availability], page 33). Some degree of speed/debuggability tradeoff is unavoidable, but the effect is not too drastic when `debug` is at least `2`.

In addition to `inline` and `notinline` declarations, the relative values of the `speed` and `space` qualities also change whether functions are inline expanded. If a function is inline

expanded, then there will be no frame to represent the call, and the arguments will be treated like any other local variable. Functions may also be "semi-inline", in which case there is a frame to represent the call, but the call is to an optimized local version of the function, not to the original function.

4.7 Exiting Commands

These commands get you out of the debugger.

`toplevel` [Debugger Command]
: Throw to top level.

`restart` [*n*] [Debugger Command]
: Invokes the *n*th restart case as displayed by the `error` command. If *n* is not specified, the available restart cases are reported.

`continue` [Debugger Command]
: Calls `continue` on the condition given to `debug`. If there is no restart case named *continue*, then an error is signaled.

`abort` [Debugger Command]
: Calls `abort` on the condition given to `debug`. This is useful for popping debug command loop levels or aborting to top level, as the case may be.

`return` *value* [Debugger Command]
: Returns *value* from the current stack frame. This command is available when the `debug` optimization quality is greater than both `speed` and `space`. Care must be taken that the value is of the same type as SBCL expects the stack frame to return.

`restart-frame` [Debugger Command]
: Restarts execution of the current stack frame. This command is available when the `debug` optimization quality is greater than both `speed` and `space` and when the frame is for is a global function. If the function is redefined in the debugger before the frame is restarted, the new function will be used.

4.8 Information Commands

Most of these commands print information about the current frame or function, but a few show general information.

`help` [Debugger Command]
`?` [Debugger Command]
: Displays a synopsis of debugger commands.

`describe` [Debugger Command]
: Calls `describe` on the current function and displays the number of local variables.

`print` [Debugger Command]
: Displays the current function call as it would be displayed by moving to this frame.

`error` [Debugger Command]
: Prints the condition given to `invoke-debugger` and the active proceed cases.

`backtrace` [*n*] [Debugger Command]
: Displays all the frames from the current to the bottom. Only shows *n* frames if specified. The printing is controlled by `*debug-print-variable-alist*`.

4.9 Function Tracing

The tracer causes selected functions to print their arguments and their results whenever they are called. Options allow conditional printing of the trace information and conditional breakpoints on function entry or exit.

`common-lisp:trace` **&rest** *specs* [Macro]
: `trace` {Option Global-Value}* {Name {Option Value}*}*

`trace` is a debugging tool that provides information when specified functions are called. In its simplest form:

```
(TRACE NAME-1 NAME-2 ...)
```

The NAMEs are not evaluated. Each may be a symbol, denoting an individual function, or a string, denoting all functions fbound to symbols whose home package is the package with the given name.

Options allow modification of the default behavior. Each option is a pair of an option keyword and a value form. Global options are specified before the first name, and affect all functions traced by a given use of `trace`. Options may also be interspersed with function names, in which case they act as local options, only affecting tracing of the immediately preceding function name. Local options override global options.

By default, `trace` causes a printout on `*trace-output*` each time that one of the named functions is entered or returns. (This is the basic, `ansi` Common Lisp behavior of `trace`.) As an `sbcl` extension, the `:report sb-ext:profile` option can be used to instead cause information to be silently recorded to be inspected later using the `sb-ext:profile` function.

The following options are defined:

`:report` *Report-Type*
: If Report-Type is `trace` (the default) then information is reported by printing immediately. If Report-Type is `sb-ext:profile`, information is recorded for later summary by calls to `sb-ext:profile`. If Report-Type is `nil`, then the only effect of the trace is to execute other options (e.g. `print` or BREAK).

`:condition` *Form*
`:condition-after` *Form*
`:condition-all` *Form*
: If `:condition` is specified, then `trace` does nothing unless Form evaluates to true at the time of the call. `:condition-after` is similar, but suppresses the initial printout, and is tested when the function returns. `:condition-all` tries both before and after. This option is not supported with `:report profile`.

`:break` *Form*
`:break-after` *Form*
`:break-all` *Form*
: If specified, and Form evaluates to true, then the debugger is invoked at the start of the function, at the end of the function, or both, according to the respective option.

`:print` *Form*
`:print-after` *Form*
`:print-all` *Form*
: In addition to the usual printout, the result of evaluating Form is printed at the start of the function, at the end of the function, or both, according to the respective option. Multiple print options cause multiple values to be printed.

`:wherein` *Names*
: If specified, Names is a function name or list of names. `trace` does nothing unless a call to one of those functions encloses the call to this function (i.e. it would appear in a backtrace.) Anonymous functions have string names like "DEFUN FOO". This option is not supported with `:report profile`.

`:encapsulate` {*:DEFAULT* | `t` | *NIL*}
: If `t`, the tracing is done via encapsulation (redefining the function name) rather than by modifying the function. `:default` is the default, and means to use encapsulation for interpreted functions and funcallable instances, breakpoints otherwise. When encapsulation is used, forms are *not* evaluated in the function's lexical environment, but `sb-debug:arg` can still be used.

`:methods` {*T* | *NIL*}
: If `t`, any function argument naming a generic function will have its methods traced in addition to the generic function itself.

`:function` *Function-Form*
: This is a not really an option, but rather another way of specifying what function to trace. The Function-Form is evaluated immediately, and the resulting function is instrumented, i.e. traced or profiled as specified in `report`.

`:condition`, `:break` and `:print` forms are evaluated in a context which mocks up the lexical environment of the called function, so that `sb-debug:var` and `sb-debug:arg` can be used. The `-after` and `-all` forms are evaluated in the null environment.

`common-lisp:untrace` **&rest** *specs* [Macro]
Remove tracing from the specified functions. With no args, untrace all functions.

`sb-debug:*trace-indentation-step*` [Variable]
the increase in trace indentation at each call level

`sb-debug:*max-trace-indentation*` [Variable]
If the trace indentation exceeds this value, then indentation restarts at 0.

`sb-debug:*trace-encapsulate-default*` [Variable]
the default value for the `:encapsulate` option to `trace`

`sb-debug:*trace-values*` [Variable]
This is bound to the returned values when evaluating `:break-after` and `:print-after` forms.

4.10 Single Stepping

SBCL includes an instrumentation based single-stepper for compiled code, that can be invoked via the `step` macro, or from within the debugger. See Section 4.6 [Debugger Policy Control], page 37, for details on enabling stepping for compiled code.

The following debugger commands are used for controlling single stepping.

`start` [Debugger Command]
Selects the `continue` restart if one exists and starts single stepping. None of the other single stepping commands can be used before stepping has been started either by using `start` or by using the standard `step` macro.

`step` [Debugger Command]
Steps into the current form. Stepping will be resumed when the next form that has been compiled with stepper instrumentation is evaluated.

`next` [Debugger Command]
Steps over the current form. Stepping will be disabled until evaluation of the form is complete.

`out` [Debugger Command]
Steps out of the current frame. Stepping will be disabled until the topmost stack frame that had been stepped into returns.

`stop` [Debugger Command]
Stops the single stepper and resumes normal execution.

`common-lisp:step` *form* [Macro]
The form is evaluated with single stepping enabled. Function calls outside the lexical scope of the form can be stepped into only if the functions in question have been compiled with sufficient `debug` policy to be at least partially steppable.

5 Efficiency

FIXME: The material in the CMUCL manual about getting good performance from the compiler should be reviewed, reformatted in Texinfo, lightly edited for SBCL, and substituted into this manual. In the meantime, the original CMUCL manual is still 95+% correct for the SBCL version of the Python compiler. See the sections

- Advanced Compiler Use and Efficiency Hints
- Advanced Compiler Introduction
- More About Types in Python
- Type Inference
- Source Optimization
- Tail Recursion
- Local Call
- Block Compilation
- Inline Expansion
- Object Representation
- Numbers
- General Efficiency Hints
- Efficiency Notes

Besides this information from the CMUCL manual, there are a few other points to keep in mind.

- The CMUCL manual doesn't seem to state it explicitly, but Python has a mental block about type inference when assignment is involved. Python is very aggressive and clever about inferring the types of values bound with `let`, `let*`, inline function call, and so forth. However, it's much more passive and dumb about inferring the types of values assigned with `setq`, `setf`, and friends. It would be nice to fix this, but in the meantime don't expect that just because it's very smart about types in most respects it will be smart about types involved in assignments. (This doesn't affect its ability to benefit from explicit type declarations involving the assigned variables, only its ability to get by without explicit type declarations.)
- Since the time the CMUCL manual was written, CMUCL (and thus SBCL) has gotten a generational garbage collector. This means that there are some efficiency implications of various patterns of memory usage which aren't discussed in the CMUCL manual. (Some new material should be written about this.)
- SBCL has some important known efficiency problems. Perhaps the most important are
  - There is only limited support for the ANSI `dynamic-extent` declaration. See Section 5.1 [Dynamic-extent allocation], page 43.
  - The garbage collector is not particularly efficient, at least on platforms without the generational collector (as of SBCL 0.8.9, all except x86).
  - Various aspects of the PCL implementation of CLOS are more inefficient than necessary.

Finally, note that Common Lisp defines many constructs which, in the infamous phrase, "could be compiled efficiently by a sufficiently smart compiler". The phrase is infamous because making a compiler which actually is sufficiently smart to find all these optimizations systematically is well beyond the state of the art of current compiler technology. Instead, they're optimized on a case-by-case basis by hand-written code, or not optimized at all if the appropriate case hasn't been hand-coded. Some cases where no such hand-coding has been done as of SBCL version 0.6.3 include

- `(reduce #'f x)` where the type of `x` is known at compile time
- various bit vector operations, e.g. `(position 0 some-bit-vector)`
- specialized sequence idioms, e.g. `(remove item list :count 1)`
- cases where local compilation policy does not require excessive type checking, e.g. `(locally (declare (safety 1)) (assoc item list))` (which currently performs safe `endp` checking internal to assoc).

If your system's performance is suffering because of some construct which could in principle be compiled efficiently, but which the SBCL compiler can't in practice compile efficiently, consider writing a patch to the compiler and submitting it for inclusion in the main sources. Such code is often reasonably straightforward to write; search the sources for the string "`deftransform`" to find many examples (some straightforward, some less so).

5.1 Dynamic-extent allocation

SBCL has limited support for performing allocation on the stack when a variable is declared `dynamic-extent`. The `dynamic-extent` declarations are not verified, but are simply trusted; if the constraints in the Common Lisp standard are violated, the best that can happen is for the program to have garbage in variables and return values; more commonly, the system will crash.

As a consequence of this, the condition for performing stack allocation is stringent: either of the `speed` or `space` optimization qualities must be higher than the maximum of `safety` and `debug` at the point of the allocation. For example:

```
(locally
  (declare (optimize speed (safety 1) (debug 1)))
  (defun foo (&rest rest)
    (declare (dynamic-extent rest))
    (length rest)))
```

Here the `&rest` list will be allocated on the stack. Note that it would not be in the following situation:

```
(defun foo (&rest rest)
  (declare (optimize speed (safety 1) (debug 1)))
  (declare (dynamic-extent rest))
  (length rest))
```

because both the allocation of the `&rest` list and the variable binding are outside the scope of the `optimize` declaration.

There are many cases when `dynamic-extent` declarations could be useful. At present, SBCL implements

- Stack allocation of `&rest` lists, where these are declared `dynamic-extent`.
- Stack allocation of `list` and `list*`, whose result is bound to a variable, declared `dynamic-extent`, such as

```
(let ((list (list 1 2 3)))
  (declare (dynamic-extent list)
  ...))
```

  or

```
(flet ((f (x)
         (declare (dynamic-extent x))
         ...))
  ...
  (f (list 1 2 3))
  ...)
```

- Stack allocation of simple forms of `make-array`, whose result is bound to a variable, declared `dynamic-extent`. The resulting array should be one-dimensional, the only allowed keyword argument is `:element-type`.

  Notice, that stack space is limited, so allocation of a large vector may cause stack overflow and abnormal termination of the SBCL process.
- Stack allocation of closures, defined with `flet` or `labels` with a bound declaration `dynamic-extent`. Closed-over variables, which are assigned (either inside or outside the closure) are still allocated on the heap. Blocks and tags are also allocated on the heap, unless all non-local control transfers to them are compiled with zero `safety`.

Future plans include

- Stack allocation of closures, where these are declared `dynamic-extent`;
- Stack allocation of `list`, `list*` and `cons` (including following chains during initialization, and also for binding mutation), where the allocation is declared `dynamic-extent`;
- Automatic detection of the common idiom of applying a function to some defaults and a `&rest` list, even when this is not declared `dynamic-extent`;
- Automatic detection of the common idiom of calling quantifiers with a closure, even when the closure is not declared `dynamic-extent`.

5.2 Modular arithmetic

Some numeric functions have a property: N lower bits of the result depend only on N lower bits of (all or some) arguments. If the compiler sees an expression of form `(logand` *`exp mask`*`)`, where *exp* is a tree of such "good" functions and *mask* is known to be of type `(unsigned-byte w)`, where w is a "good" width, all intermediate results will be cut to w bits (but it is not done for variables and constants!). This often results in an ability to use simple machine instructions for the functions.

Consider an example.

```
(defun i (x y)
  (declare (type (unsigned-byte 32) x y))
  (ldb (byte 32 0) (logxor x (lognot y))))
```

The result of `(lognot y)` will be negative and of type `(signed-byte 33)`, so a naive implementation on a 32-bit platform is unable to use 32-bit arithmetic here. But modular arithmetic optimizer is able to do it: because the result is cut down to 32 bits, the compiler will replace `logxor` and `lognot` with versions cutting results to 32 bits, and because terminals (here—expressions `x` and `y`) are also of type `(unsigned-byte 32)`, 32-bit machine arithmetic can be used.

As of SBCL 0.8.5 "good" functions are `+`, `-`; `logand`, `logior`, `logxor`, `lognot` and their combinations; and `ash` with the positive second argument. "Good" widths are 32 on HPPA, MIPS, PPC, Sparc and x86 and 64 on Alpha. While it is possible to support smaller widths as well, currently this is not implemented.

6 Beyond the ANSI Standard

SBCL is derived from CMUCL, which implements many extensions to the ANSI standard. SBCL doesn't support as many extensions as CMUCL, but it still has quite a few. See Chapter 15 [Contributed Modules], page 106.

6.1 Garbage Collection

SBCL provides additional garbage collection functionality not specified by ANSI. Weak pointers allow references to objects to be maintained without keeping them from being garbage collected, and "finalization" hooks are available to cause code to be executed when an object has been garbage collected. Additionally users can specify their own cleanup actions to be executed with garbage collection.

`sb-ext:finalize` *object function* **&key** *dont-save* [Function]

Arrange for the designated `function` to be called when there are no more references to `object`, including references in `function` itself.

If `dont-save` is true, the finalizer will be cancelled when `save-lisp-and-die` is called: this is useful for finalizers deallocating system memory, which might otherwise be called with addresses from the old image.

In a multithreaded environment `function` may be called in any thread. In both single and multithreaded environments `function` may be called in any dynamic scope: consequences are unspecified if `function` is not fully re-entrant.

Errors from `function` are handled and cause a `warning` to be signalled in whichever thread the `function` was called in.

Examples:

```
;;; good (assumes RELEASE-HANDLE is re-entrant)
(let* ((handle (get-handle))
       (object (make-object handle)))
 (finalize object (lambda () (release-handle handle)))
 object)

;;; bad, finalizer refers to object being finalized, causing
;;; it to be retained indefinitely
(let* ((handle (get-handle))
       (object (make-object handle)))
  (finalize object (lambda () (release-handle (object-handle object)))))

;;; bad, not re-entrant
(defvar *rec* nil)

(defun oops ()
 (when *rec*
   (error "recursive OOPS"))
 (let ((*rec* t))
   (gc))) ; or just cons enough to cause one

(progn
  (finalize "oops" #'oops)
```

```
        (oops)) ; causes GC and re-entry to #'oops due to the finalizer
                ; -> ERROR, caught, WARNING signalled
```

`sb-ext:cancel-finalization` *object* [Function]
Cancel any finalization for `object`.

`sb-ext:make-weak-pointer` *object* [Function]
Allocate and return a weak pointer which points to `object`.

`sb-ext:weak-pointer-value` *weak-pointer* [Function]
If `weak-pointer` is valid, return the value of `weak-pointer` and `t`. If the referent of `weak-pointer` has been garbage collected, returns the values `nil` and `nil`.

`sb-ext:*after-gc-hooks*` [Variable]
Called after each garbage collection, except for garbage collections triggered during thread exits. In a multithreaded environment these hooks may run in any thread.

6.2 Metaobject Protocol

SBCL supports a metaobject protocol which is intended to be compatible with AMOP; present exceptions to this (as distinct from current bugs) are:

- `compute-effective-method` only returns one value, not two.

  There is no record of what the second return value was meant to indicate, and apparently no clients for it.

- The direct superclasses of `sb-mop:funcallable-standard-object` are `(function standard-object)`, not `(standard-object function)`.

  This is to ensure that the `standard-object` class is the last of the standardized classes before `t` appearing in the class precedence list of `generic-function` and `standard-generic-function`, as required by section 1.4.4.5 of the ANSI specification.

- the arguments `:declare` and `:declarations` to `ensure-generic-function` are both accepted, with the leftmost argument defining the declarations to be stored and returned by `generic-function-declarations`.

  Where AMOP specifies `:declarations` as the keyword argument to `ensure-generic-function`, the Common Lisp standard specifies `:declare`. Portable code should use `:declare`.

- although SBCL obeys the requirement in AMOP that `validate-superclass` should treat `standard-class` and `funcallable-standard-class` as compatible metaclasses, we impose an additional requirement at class finalization time: a class of metaclass `funcallable-standard-class` must have `function` in its superclasses, and a class of metaclass `standard-class` must not.

  After a class has been finalized, it is associated with a class prototype which is accessible by a standard mop function `sb-mop:class-prototype`. The user can then ask whether this object is a `function` or not in several different ways: whether it is a function according to `typep`; whether its `class-of` is `subtypep` `function`, or whether `function`

appears in the superclasses of the class. The additional consistency requirement comes from the desire to make all of these answers the same.

The following class definitions are bad, and will lead to errors either immediately or if an instance is created:

```
(defclass bad-object (funcallable-standard-object)
  ()
  (:metaclass standard-class))
(defclass bad-funcallable-object (standard-object)
  ()
  (:metaclass funcallable-standard-class))
```

The following definition is acceptable:

```
(defclass mixin ()
  ((slot :initarg slot)))
(defclass funcallable-object (funcallable-standard-object mixin)
  ()
  (:metaclass funcallable-standard-class))
```

and leads to a class whose instances are funcallable and have one slot.

Note that this requirement also applies to the class `sb-mop:funcallable-standard-object`, which has metaclass `sb-mop:funcallable-standard-class` rather than `standard-class` as AMOP specifies.

- the requirement that "No portable class C_p may inherit, by virtue of being a direct or indirect subclass of a specified class, any slot for which the name is a symbol accessible in the `common-lisp-user` package or exported by any package defined in the ANSI Common Lisp standard." is interpreted to mean that the standardized classes themselves should not have slots named by external symbols of public packages.

  The rationale behind the restriction is likely to be similar to the ANSI Common Lisp restriction on defining functions, variables and types named by symbols in the Common Lisp package: preventing two independent pieces of software from colliding with each other.

- specializations of the `new-value` argument to `(setf sb-mop:slot-value-using-class)` are not allowed: all user-defined methods must have a specializer of the class `t`.

  This prohibition is motivated by a separation of layers: the `slot-value-using-class` family of functions is intended for use in implementing different and new slot allocation strategies, rather than in performing application-level dispatching. Additionally, with this requirement, there is a one-to-one mapping between metaclass, class and slot-definition-class tuples and effective methods of `(setf slot-value-using-class)`, which permits optimization of `(setf slot-value-using-class)`'s discriminating function in the same manner as for `slot-value-using-class` and `slot-boundp-using-class`.

  Note that application code may specialize on the `new-value` argument of slot accessors.

- the class named by the `name` argument to `ensure-class`, if any, is only redefined if it is the proper name of that class; otherwise, a new class is created.

This is consistent with the description of `ensure-class` in AMOP as the functional version of `defclass`, which has this behaviour; however, it is not consistent with the weaker requirement in AMOP, which states that any class found by `find-class`, no matter what its `class-name`, is redefined.

In addition, SBCL supports extensions to the Metaobject protocol from AMOP; at present, they are:

- compile-time support for generating specializer metaobjects from specializer names in `defmethod` forms is provided by the `make-method-specializers-form` function, which returns a form which, when evaluated in the lexical environment of the `defmethod`, returns a list of specializer metaobjects. This operator suffers from similar restrictions to those affecting `make-method-lambda`, namely that the generic function must be defined when the `defmethod` form is expanded, so that the correct method of `make-method-specializers-form` is invoked. The system-provided method on `make-method-specializers-form` generates a call to `find-class` for each symbol specializer name, and a call to `intern-eql-specializer` for each `(eql x)` specializer name.
- run-time support for converting between specializer names and specializer metaobjects, mostly for the purposes of `find-method`, is provided by `parse-specializer-using-class` and `unparse-specializer-using-class`, which dispatch on their first argument, the generic function associated with a method with the given specializer. The system-provided methods on those methods convert between classes and proper names and between lists of the form `(eql x)` and interned eql specializer objects.

6.3 Support For Unix

6.3.1 Command-line arguments

The UNIX command line can be read from the variable `sb-ext:*posix-argv*`.

6.3.2 Querying the process environment

The UNIX environment can be queried with the `sb-ext:posix-getenv` function.

`sb-ext:posix-getenv` *name* [Function]

Return the "value" part of the environment string "name=value" which corresponds to `name`, or `nil` if there is none.

6.3.3 Running external programs

External programs can be run with `sb-ext:run-program`.[1]

[1] In SBCL versions prior to 1.0.13, `sb-ext:run-program` searched for executables in a manner somewhat incompatible with other languages. As of this version, SBCL uses the system library routine `execvp(3)`, and no longer contains the function, `find-executable-in-search-path`, which implemented the old search. Users who need this function may find it in '`run-program.lisp`' versions 1.67 and earlier in SBCL's CVS repository here http://sbcl.cvs.sourceforge.net/sbcl/sbcl/src/code/run-program.lisp?view=log. However, we caution such users that this search routine finds executables that system library routines do not.

`sb-ext:run-program` *program args* **&key** *env environment wait search pty input if-input-does-not-exist output if-output-exists error if-error-exists status-hook* [Function]

`run-program` creates a new process specified by the `program` argument. `args` are the standard arguments that can be passed to a program. For no arguments, use `nil` (which means that just the name of the program is passed as arg 0).

The program arguments and the environment are encoded using the default external format for streams.

`run-program` will return a `process` structure. See the `cmu` Common Lisp Users Manual for details about the `process` structure.

Notes about Unix environments (as in the `:environment` and `:env` args):

- The `sbcl` implementation of `run-program`, like Perl and many other programs, but unlike the original `cmu cl` implementation, copies the Unix environment by default.
- Running Unix programs from a setuid process, or in any other situation where the Unix environment is under the control of someone else, is a mother lode of security problems. If you are contemplating doing this, read about it first. (The Perl community has a lot of good documentation about this and other security issues in script-like programs.)

The `&key` arguments have the following meanings:

`:environment`
: a list of STRINGs describing the new Unix environment (as in `"man environ"`). The default is to copy the environment of the current process.

`:env`
: an alternative lossy representation of the new Unix environment, for compatibility with `cmu cl`

`:search`
: Look for `program` in each of the directories in the child's $PATH environment variable. Otherwise an absolute pathname is required.

`:wait`
: If non-NIL (default), wait until the created process finishes. If `nil`, continue running Lisp until the program finishes.

`:pty`
: Either `t`, `nil`, or a stream. Unless `nil`, the subprocess is established under a `pty`. If :pty is a stream, all output to this pty is sent to this stream, otherwise the `process-pty` slot is filled in with a stream connected to pty that can read output and write input.

`:input`
: Either `t`, `nil`, a pathname, a stream, or `:stream`. If `t`, the standard input for the current process is inherited. If `nil`, /dev/null is used. If a pathname, the file so specified is used. If a stream, all the input is read from that stream and sent to the subprocess. If `:stream`, the `process-input` slot is filled in with a stream that sends its output to the process. Defaults to `nil`.

`:if-input-does-not-exist` *(when* `:input` *is the name of a file)*
: can be one of: `:error` to generate an error `:create` to create an empty file `nil` (the default) to return `nil` from `run-program`

`:output` Either `t`, `nil`, a pathname, a stream, or `:stream`. If `t`, the standard output for the current process is inherited. If `nil`, /dev/null is used. If a pathname, the file so specified is used. If a stream, all the output from the process is written to this stream. If `:stream`, the `process-output` slot is filled in with a stream that can be read to get the output. Defaults to `nil`.

`:if-output-exists` *(when* `:output` *is the name of a file)*
can be one of: `:error` (the default) to generate an error `:supersede` to supersede the file with output from the program `:append` to append output from the program to the file `nil` to return `nil` from `run-program`, without doing anything

`:error` *and* `:if-error-exists`
Same as `:output` and `:if-output-exists`, except that `:error` can also be specified as `:output` in which case all error output is routed to the same place as normal output.

`:status-hook`
This is a function the system calls whenever the status of the process changes. The function takes the process as an argument.

When `sb-ext:run-program` is called with `wait` equal to NIL, an instance of class *sb-ext:process* is returned. The following functions are available for use with processes:

`sb-ext:process-p` *object* [Function]
`t` if `object` is a `process`, `nil` otherwise.

`sb-ext:process-input` *instance* [Function]
The input stream of the process or `nil`.

`sb-ext:process-output` *instance* [Function]
The output stream of the process or `nil`.

`sb-ext:process-error` *instance* [Function]
The error stream of the process or `nil`.

`sb-ext:process-alive-p` *process* [Function]
Return `t` if `process` is still alive, `nil` otherwise.

`sb-ext:process-status` *process* [Function]
Return the current status of `process`. The result is one of `:running`, `:stopped`, `:exited`, or `:signaled`.

`sb-ext:process-wait` *process* **&optional** *check-for-stopped* [Function]
Wait for `process` to quit running for some reason. When `check-for-stopped` is `t`, also returns when `process` is stopped. Returns `process`.

`sb-ext:process-exit-code` *instance* [Function]
The exit code or the signal of a stopped process.

`sb-ext:process-core-dumped` *instance* [Function]
`t` if a core image was dumped by the process.

`sb-ext:process-close` *process* [Function]
Close all streams connected to `process` and stop maintaining the status slot.

`sb-ext:process-kill` *process signal* **&optional** *whom* [Function]
Hand `signal` to `process`. If `whom` is `:pid`, use the kill Unix system call. If `whom` is `:process-group`, use the killpg Unix system call. If `whom` is `:pty-process-group` deliver the signal to whichever process group is currently in the foreground.

6.4 Customization Hooks for Users

The toplevel repl prompt may be customized, and the function that reads user input may be replaced completely.

The behaviour of `require` when called with only one argument is implementation-defined. In SBCL, `require` behaves in the following way:

`common-lisp:require` *module-name* **&optional** *pathnames* [Function]
Loads a module, unless it already has been loaded. `pathnames`, if supplied, is a designator for a list of pathnames to be loaded if the module needs to be. If `pathnames` is not supplied, functions from the list `*module-provider-functions*` are called in order with `module-name` as an argument, until one of them returns non-NIL. User code is responsible for calling `provide` to indicate a successful load of the module.

`sb-ext:*module-provider-functions*` [Variable]
See function documentation for `require`.

Although SBCL does not provide a resident editor, the `ed` function can be customized to hook into user-provided editing mechanisms as follows:

`common-lisp:ed` **&optional** *x* [Function]
Starts the editor (on a file or a function if named). Functions from the list `*ed-functions*` are called in order with `x` as an argument until one of them returns non-NIL; these functions are responsible for signalling a `file-error` to indicate failure to perform an operation on the file system.

`sb-ext:*ed-functions*` [Variable]
See function documentation for `ed`.

6.5 Tools To Help Developers

SBCL provides a profiler and other extensions to the ANSI `trace` facility. For more information, see [Macro common-lisp:trace], page 39.

The debugger supports a number of options. Its documentation is accessed by typing `help` at the debugger prompt. See Chapter 4 [Debugger], page 28.

Documentation for `inspect` is accessed by typing `help` at the `inspect` prompt.

6.6 Resolution of Name Conflicts

The ANSI standard (section 11.1.1.2.5) requires that name conflicts in packages be resolvable in favour of any of the conflicting symbols. In the interactive debugger, this is achieved by prompting for the symbol in whose favour the conflict should be resolved; for programmatic use, the `sb-ext:resolve-conflict` restart should be invoked with one argument, which should be a member of the list returned by the condition accessor `sb-ext:name-conflict-symbols`.

6.7 Stale Extensions

SBCL has inherited from CMUCL various hooks to allow the user to tweak and monitor the garbage collection process. These are somewhat stale code, and their interface might need to be cleaned up. If you have urgent need of them, look at the code in '`src/code/gc.lisp`' and bring it up on the developers' mailing list.

SBCL has various hooks inherited from CMUCL, like `sb-ext:float-denormalized-p`, to allow a program to take advantage of IEEE floating point arithmetic properties which aren't conveniently or efficiently expressible using the ANSI standard. These look good, and their interface looks good, but IEEE support is slightly broken due to a stupid decision to remove some support for infinities (because it wasn't in the ANSI spec and it didn't occur to me that it was in the IEEE spec). If you need this stuff, take a look at the code and bring it up on the developers' mailing list.

6.8 Efficiency Hacks

The `sb-ext:purify` function causes SBCL first to collect all garbage, then to mark all uncollected objects as permanent, never again attempting to collect them as garbage. This can cause a large increase in efficiency when using a primitive garbage collector, or a more moderate increase in efficiency when using a more sophisticated garbage collector which is well suited to the program's memory usage pattern. It also allows permanent code to be frozen at fixed addresses, a precondition for using copy-on-write to share code between multiple Lisp processes. This is less important with modern generational garbage collectors, but not all SBCL platforms use such a garbage collector.

`sb-ext:purify` **&key** *root-structures environment-name* [Function]

This function optimizes garbage collection by moving all currently live objects into non-collected storage. `root-structures` is an optional list of objects which should be copied first to maximize locality.

`defstruct` structures defined with the (:PURE T) option are moved into read-only storage, further reducing `gc` cost. List and vector slots of pure structures are also moved into read-only storage.

`environment-name` is gratuitous documentation for compacted version of the current global environment (as seen in `sb!c::*info-environment*`.) If `nil` is supplied, then environment compaction is inhibited.

This function is a no-op on platforms using the generational garbage collector (x86, x86-64, ppc).

The `sb-ext:truly-the` special form declares the type of the result of the operations, producing its argument; the declaration is not checked. In short: don't use it.

`sb-ext:truly-the` *type value* [Special Operator]

The `sb-ext:freeze-type` declaration declares that a type will never change, which can make type testing (`typep`, etc.) more efficient for structure types.

The `sb-ext:constant-function` declaration specifies that a function will always return the same value for the same arguments, which may allow the compiler to optimize calls to it. This is appropriate for functions like `sqrt`, but is *not* appropriate for functions like `aref`, which can change their return values when the underlying data are changed.

7 Foreign Function Interface

This chapter describes SBCL's interface to C programs and libraries (and, since C interfaces are a sort of *lingua franca* of the Unix world, to other programs and libraries in general.)

> Note: In the modern Lisp world, the usual term for this functionality is Foreign Function Interface, or FFI, where despite the mention of "function" in this term, FFI also refers to direct manipulation of C data structures as well as functions. The traditional CMUCL terminology is Alien Interface, and while that older terminology is no longer used much in the system documentation, it still reflected in names in the implementation, notably in the name of the `SB-ALIEN` package.

7.1 Introduction to the Foreign Function Interface

Because of Lisp's emphasis on dynamic memory allocation and garbage collection, Lisp implementations use non-C-like memory representations for objects. This representation mismatch creates friction when a Lisp program must share objects with programs which expect C data. There are three common approaches to establishing communication:

- The burden can be placed on the foreign program (and programmer) by requiring the knowledge and use of the representations used internally by the Lisp implementation. This can require a considerable amount of "glue" code on the C side, and that code tends to be sensitively dependent on the internal implementation details of the Lisp system.
- The Lisp system can automatically convert objects back and forth between the Lisp and foreign representations. This is convenient, but translation becomes prohibitively slow when large or complex data structures must be shared. This approach is supported by the SBCL FFI, and used automatically by the when passing integers and strings.
- The Lisp program can directly manipulate foreign objects through the use of extensions to the Lisp language.

SBCL, like CMUCL before it, relies primarily on the automatic conversion and direct manipulation approaches. The `SB-ALIEN` package provides a facility wherein foreign values of simple scalar types are automatically converted and complex types are directly manipulated in their foreign representation. Additionally the lower-level System Area Pointers (or SAPs) can be used where necessary to provide untyped access to foreign memory.

Any foreign objects that can't automatically be converted into Lisp values are represented by objects of type `alien-value`. Since Lisp is a dynamically typed language, even foreign objects must have a run-time type; this type information is provided by encapsulating the raw pointer to the foreign data within an `alien-value` object.

The type language and operations on foreign types are intentionally similar to those of the C language.

7.2 Foreign Types

Alien types have a description language based on nested list structure. For example the C type

```
struct foo {
    int a;
    struct foo *b[100];
};
```

has the corresponding SBCL FFI type

```
(struct foo
  (a int)
  (b (array (* (struct foo)) 100)))
```

7.2.1 Defining Foreign Types

Types may be either named or anonymous. With structure and union types, the name is part of the type specifier, allowing recursively defined types such as:

```
(struct foo (a (* (struct foo))))
```

An anonymous structure or union type is specified by using the name `nil`. The `with-alien` macro defines a local scope which "captures" any named type definitions. Other types are not inherently named, but can be given named abbreviations using the `define-alien-type` macro.

7.2.2 Foreign Types and Lisp Types

The foreign types form a subsystem of the SBCL type system. An `alien` type specifier provides a way to use any foreign type as a Lisp type specifier. For example,

```
(typep foo '(alien (* int)))
```

can be used to determine whether *foo* is a pointer to a foreign `int`. `alien` type specifiers can be used in the same ways as ordinary Lisp type specifiers (like `string`.) Alien type declarations are subject to the same precise type checking as any other declaration. See Section 3.2.2 [Precise Type Checking], page 22.

Note that the type identifiers used in the foreign type system overlap with native Lisp type specifiers in some cases. For example, the type specifier `(alien single-float)` is identical to `single-float`, since foreign floats are automatically converted to Lisp floats. When `type-of` is called on an alien value that is not automatically converted to a Lisp value, then it will return an `alien` type specifier.

7.2.3 Foreign Type Specifiers

Note: All foreign type names are exported from the `sb-alien` package. Some foreign type names are also symbols in the `common-lisp` package, in which case they are reexported from the `sb-alien` package, so that e.g. it is legal to refer to `sb-alien:single-float`.

These are the basic foreign type specifiers:

- The foreign type specifier `(* foo)` describes a pointer to an object of type *foo*. A pointed-to type *foo* of `t` indicates a pointer to anything, similar to `void *` in ANSI C. A null alien pointer can be detected with the `sb-alien:null-alien` function.
- The foreign type specifier `(array foo &rest dimensions)` describes array of the specified `dimensions`, holding elements of type *foo*. Note that (unlike in C) `(* foo)` and `(array foo)` are considered to be different types when type checking is done. If equivalence of pointer and array types is desired, it may be explicitly coerced using `sb-alien:cast`.

Arrays are accessed using `sb-alien:deref`, passing the indices as additional arguments. Elements are stored in column-major order (as in C), so the first dimension determines only the size of the memory block, and not the layout of the higher dimensions. An array whose first dimension is variable may be specified by using `nil` as the first dimension. Fixed-size arrays can be allocated as array elements, structure slots or `sb-alien:with-alien` variables. Dynamic arrays can only be allocated using `sb-alien:make-alien`.

- The foreign type specifier `(sb-alien:struct` *name* `&rest` *fields*`)` describes a structure type with the specified *name* and *fields*. Fields are allocated at the same offsets used by the implementation's C compiler, as guessed by the SBCL internals. An optional `:alignment` keyword argument can be specified for each field to explicitly control the alignment of a field. If *name* is `nil` then the structure is anonymous.

  If a named foreign `struct` specifier is passed to `define-alien-type` or `with-alien`, then this defines, respectively, a new global or local foreign structure type. If no *fields* are specified, then the fields are taken from the current (local or global) alien structure type definition of *name*.
- The foreign type specifier `(sb-alien:union` *name* `&rest` *fields*`)` is similar to `sb-alien:struct`, but describes a union type. All fields are allocated at the same offset, and the size of the union is the size of the largest field. The programmer must determine which field is active from context.
- The foreign type specifier `(sb-alien:enum` *name* `&rest` *specs*`)` describes an enumeration type that maps between integer values and symbols. If *name* is `nil`, then the type is anonymous. Each element of the *specs* list is either a Lisp symbol, or a list `(`*symbol value*`)`. *value* is an integer. If *value* is not supplied, then it defaults to one greater than the value for the preceding spec (or to zero if it is the first spec).
- The foreign type specifier `(sb-alien:signed &optional` *bits*`)` specifies a signed integer with the specified number of *bits* precision. The upper limit on integer precision is determined by the machine's word size. If *bits* is not specified, the maximum size will be used.
- The foreign type specifier `(integer &optional` *bits*`)` is equivalent to the corresponding type specifier using `sb-alien:signed` instead of `integer`.
- The foreign type specifier `(sb-alien:unsigned &optional` *bits*`)` is like corresponding type specifier using `sb-alien:signed` except that the variable is treated as an unsigned integer.
- The foreign type specifier `(boolean &optional` *bits*`)` is similar to an enumeration type, but maps from Lisp `nil` and `t` to C `0` and `1` respectively. *bits* determines the amount of storage allocated to hold the truth value.
- The foreign type specifier `single-float` describes a floating-point number in IEEE single-precision format.
- The foreign type specifier `double-float` describes a floating-point number in IEEE double-precision format.
- The foreign type specifier `(function` *result-type* `&rest` *arg-types*`)` describes a foreign function that takes arguments of the specified *arg-types* and returns a result of type *result-type*. Note that the only context where a foreign `function` type is directly

specified is in the argument to `sb-alien:alien-funcall`. In all other contexts, foreign functions are represented by foreign function pointer types: `(* (function ...))`.

- The foreign type specifier `sb-alien:system-area-pointer` describes a pointer which is represented in Lisp as a `system-area-pointer` object. SBCL exports this type from `sb-alien` because CMUCL did, but tentatively (as of the first draft of this section of the manual, SBCL 0.7.6) it is deprecated, since it doesn't seem to be required by user code.
- The foreign type specifier `sb-alien:void` is used in function types to declare that no useful value is returned. Using `alien-funcall` to call a `void` foreign function will return zero values.
- The foreign type specifier `(sb-alien:c-string &key external-format element-type)` is similar to `(* char)`, but is interpreted as a null-terminated string, and is automatically converted into a Lisp string when accessed; or if the pointer is C `NULL` or `0`, then accessing it gives Lisp `nil`.

  External format conversion is automatically done when Lisp strings are passed to foreign code, or when foreign strings are passed to Lisp code. If the type specifier has an explicit `external-format`, that external format will be used. Otherwise a default external format that has been determined at SBCL startup time based on the current locale settings will be used. For example, when the following alien routine is called, the Lisp string given as argument is converted to an `ebcdic` octet representation.

  ```
  (define-alien-routine test int (str (c-string :external-format :ebcdic-us)))
  ```

  Lisp strings of type `base-string` are stored with a trailing NUL termination, so no copying (either by the user or the implementation) is necessary when passing them to foreign code, assuming that the `external-format` and `element-type` of the `c-string` type are compatible with the internal representation of the string. For an SBCL built with Unicode support that means an `external-format` of `:ascii` and an `element-type` of `base-char`. Without Unicode support the `external-format` can also be `:iso-8859-1`, and the `element-type` can also be `character`. If the `external-format` or `element-type` is not compatible, or the string is a `(simple-array character (*))`, this data is copied by the implementation as required.

  Assigning a Lisp string to a `c-string` structure field or variable stores the contents of the string to the memory already pointed to by that variable. When a foreign object of type `(* char)` is assigned to a `c-string`, then the `c-string` pointer is assigned to. This allows `c-string` pointers to be initialized. For example:

  ```
  (cl:in-package "CL-USER") ; which USEs package "SB-ALIEN"

  (define-alien-type nil (struct foo (str c-string)))

  (defun make-foo (str)
    (let ((my-foo (make-alien (struct foo))))
      (setf (slot my-foo 'str) (make-alien char (length str))
            (slot my-foo 'str) str)
      my-foo))
  ```

  Storing Lisp `NIL` in a `c-string` writes C `NULL` to the variable.

- `sb-alien` also exports translations of these C type specifiers as foreign type specifiers: `sb-alien:char`, `sb-alien:short`, `sb-alien:int`, `sb-alien:long`, `sb-alien:unsigned-char`, `sb-alien:unsigned-short`, `sb-alien:unsigned-int`, `sb-alien:unsigned-long`, `sb-alien:float`, and `sb-alien:double`.

7.3 Operations On Foreign Values

This section describes how to read foreign values as Lisp values, how to coerce foreign values to different kinds of foreign values, and how to dynamically allocate and free foreign variables.

7.3.1 Accessing Foreign Values

`sb-alien:deref` *pointer-or-array* **&rest** *indices* [Function]

The `sb-alien:deref` function returns the value pointed to by a foreign pointer, or the value of a foreign array element. When dereferencing a pointer, an optional single index can be specified to give the equivalent of C pointer arithmetic; this index is scaled by the size of the type pointed to. When dereferencing an array, the number of indices must be the same as the number of dimensions in the array type. `deref` can be set with `setf` to assign a new value.

`sb-alien:slot` *struct-or-union slot-name* [Function]

The `sb-alien:slot` function extracts the value of the slot named *slot-name* from a foreign `struct` or `union`. If *struct-or-union* is a pointer to a structure or union, then it is automatically dereferenced. `sb-alien:slot` can be set with `setf` to assign a new value. Note that *slot-name* is evaluated, and need not be a compile-time constant (but only constant slot accesses are efficiently compiled).

7.3.1.1 Untyped memory

As noted at the beginning of the chapter, the System Area Pointer facilities allow untyped access to foreign memory. SAPs can be converted to and from the usual typed foreign values using `sap-alien` and `alien-sap` (described elsewhere), and also to and from integers - raw machine addresses. They should thus be used with caution; corrupting the Lisp heap or other memory with SAPs is trivial.

`sb-sys:int-sap` *machine-address* [Function]

Creates a SAP pointing at the virtual address *machine-address*.

`sb-sys:sap-ref-32` *sap offset* [Function]

Access the value of the memory location at *offset* bytes from *sap*. This form may also be used with `setf` to alter the memory at that location.

`sb-sys:sap=` *sap1 sap2* [Function]

Compare *sap1* and *sap2* for equality.

Similarly named functions exist for accessing other sizes of word, other comparisons, and other conversions. The reader is invited to use `apropos` and `describe` for more details

```
(apropos "sap" :sb-sys)
```

7.3.2 Coercing Foreign Values

`sb-alien:addr` *alien-expr* [Function]

The `sb-alien:addr` macro returns a pointer to the location specified by *alien-expr*, which must be either a foreign variable, a use of `sb-alien:deref`, a use of `sb-alien:slot`, or a use of `sb-alien:extern-alien`.

`sb-alien:cast` *foreign-value new-type* [Function]

The `sb-alien:cast` macro converts *foreign-value* to a new foreign value with the specified *new-type*. Both types, old and new, must be foreign pointer, array or function types. Note that the resulting Lisp foreign variable object is not `eq` to the argument, but it does refer to the same foreign data bits.

`sb-alien:sap-alien` *sap type* [Function]

The `sb-alien:sap-alien` function converts *sap* (a system area pointer) to a foreign value with the specified *type*. *type* is not evaluated.

The *type* must be some foreign pointer, array, or record type.

`sb-alien:alien-sap` *foreign-value type* [Function]

The `sb-alien:alien-sap` function returns the SAP which points to *alien-value*'s data.

The *foreign-value* must be of some foreign pointer, array, or record type.

7.3.3 Foreign Dynamic Allocation

Lisp code can call the C standard library functions `malloc` and `free` to dynamically allocate and deallocate foreign variables. The Lisp code shares the same allocator with foreign C code, so it's OK for foreign code to call `free` on the result of Lisp `sb-alien:make-alien`, or for Lisp code to call `sb-alien:free-alien` on foreign objects allocated by C code.

`sb-alien:make-alien` *type size* [Macro]

The `sb-alien:make-alien` macro returns a dynamically allocated foreign value of the specified *type* (which is not evaluated.) The allocated memory is not initialized, and may contain arbitrary junk. If supplied, *size* is an expression to evaluate to compute the size of the allocated object. There are two major cases:

- When *type* is a foreign array type, an array of that type is allocated and a pointer to it is returned. Note that you must use `deref` to change the result to an array before you can use `deref` to read or write elements:

  ```
  (cl:in-package "CL-USER") ; which USEs package "SB-ALIEN"
  (defvar *foo* (make-alien (array char 10)))
  (type-of *foo*) ⇒ (alien (* (array (signed 8) 10)))
  (setf (deref (deref foo) 0) 10) ⇒ 10
  ```

  If supplied, *size* is used as the first dimension for the array.

- When *type* is any other foreign type, then an object for that type is allocated, and a pointer to it is returned. So `(make-alien int)` returns a `(* int)`. If *size* is specified, then a block of that many objects is allocated, with the result pointing to the first one.

`sb-alien:free-alien` *foreign-value* [Function]

The `sb-alien:free-alien` function frees the storage for *foreign-value*, which must have been allocated with Lisp `make-alien` or C `malloc`.

See also the `sb-alien:with-alien` macro, which allocates foreign values on the stack.

7.4 Foreign Variables

Both local (stack allocated) and external (C global) foreign variables are supported.

7.4.1 Local Foreign Variables

`sb-alien:with-alien` *var-definitions* **&body** *body* [Macro]

The `with-alien` macro establishes local foreign variables with the specified alien types and names. This form is analogous to defining a local variable in C: additional storage is allocated, and the initial value is copied. This form is less analogous to `LET`-allocated Lisp variables, since the variables can't be captured in closures: they live only for the dynamic extent of the body, and referring to them outside is a gruesome error.

The *var-definitions* argument is a list of variable definitions, each of the form

```
(name type &optional initial-value)
```

The names of the variables are established as symbol-macros; the bindings have lexical scope, and may be assigned with `setq` or `setf`.

The `with-alien` macro also establishes a new scope for named structures and unions. Any *type* specified for a variable may contain named structure or union types with the slots specified. Within the lexical scope of the binding specifiers and body, a locally defined foreign structure type *foo* can be referenced by its name using `(struct foo)`.

7.4.2 External Foreign Variables

External foreign names are strings, and Lisp names are symbols. When an external foreign value is represented using a Lisp variable, there must be a way to convert from one name syntax into the other. The macros `extern-alien`, `define-alien-variable` and `define-alien-routine` use this conversion heuristic:

- Alien names are converted to Lisp names by uppercasing and replacing underscores with hyphens.
- Conversely, Lisp names are converted to alien names by lowercasing and replacing hyphens with underscores.
- Both the Lisp symbol and alien string names may be separately specified by using a list of the form

  ```
  (alien-string lisp-symbol)
  ```

`sb-alien:define-alien-variable` *name type* [Macro]

The `define-alien-variable` macro defines *name* as an external foreign variable of the specified foreign `type`. *name* and `type` are not evaluated. The Lisp name of the variable (see above) becomes a global alien variable. Global alien variables are effectively "global symbol macros"; a reference to the variable fetches the contents of the external variable. Similarly, setting the variable stores new contents – the

new contents must be of the declared `type`. Someday, they may well be implemented using the ANSI `define-symbol-macro` mechanism, but as of SBCL 0.7.5, they are still implemented using an older more-or-less parallel mechanism inherited from CMUCL.

For example, to access a C-level counter *foo*, one could write

```
(define-alien-variable "foo" int)
;; Now it is possible to get the value of the C variable foo simply by
;; referencing that Lisp variable:
(print foo)
(setf foo 14)
(incf foo)
```

`sb-alien:get-errno` [Function]

Since in modern C libraries, the `errno` "variable" is typically no longer a variable, but some bizarre artificial construct which behaves superficially like a variable within a given thread, it can no longer reliably be accessed through the ordinary `define-alien-variable` mechanism. Instead, SBCL provides the operator `sb-alien:get-errno` to allow Lisp code to read it.

`sb-alien:extern-alien` *name type* [Macro]

The `extern-alien` macro returns an alien with the specified *type* which points to an externally defined value. *name* is not evaluated, and may be either a string or a symbol. *type* is an unevaluated alien type specifier.

7.5 Foreign Data Structure Examples

Now that we have alien types, operations and variables, we can manipulate foreign data structures. This C declaration

```
struct foo {
    int a;
    struct foo *b[100];
};
```

can be translated into the following alien type:

```
(define-alien-type nil
  (struct foo
    (a int)
    (b (array (* (struct foo)) 100))))
```

Once the `foo` alien type has been defined as above, the C expression

```
struct foo f;
f.b[7].a;
```

can be translated in this way:

```
(with-alien ((f (struct foo)))
  (slot (deref (slot f 'b) 7) 'a)
  ;;
  ;; Do something with f...
  )
```

Or consider this example of an external C variable and some accesses:

```
struct c_struct {
        short x, y;
        char a, b;
        int z;
        c_struct *n;
};
extern struct c_struct *my_struct;
my_struct->x++;
my_struct->a = 5;
my_struct = my_struct->n;
```

which can be manipulated in Lisp like this:

```
(define-alien-type nil
  (struct c-struct
          (x short)
          (y short)
          (a char)
          (b char)
          (z int)
          (n (* c-struct))))
(define-alien-variable "my_struct" (* c-struct))
(incf (slot my-struct 'x))
(setf (slot my-struct 'a) 5)
(setq my-struct (slot my-struct 'n))
```

7.6 Loading Shared Object Files

Foreign object files can be loaded into the running Lisp process by calling `load-shared-object`.

`sb-alien:load-shared-object` *file* [Function]

Load a shared library/dynamic shared object file/general dlopenable alien container, such as a .so on an `elf` platform.

Reloading the same shared object will replace the old definitions; if a symbol was previously referenced thru the object and is not present in the reloaded version an error will be signalled. Sameness is determined using the library filename. Reloading may not work as expected if user or library-code has called dlopen on `file`.

References to foreign symbols in loaded shared objects do not survive intact through `sb-ext:save-lisp-and-die` on all platforms. See `sb-ext:save-lisp-and-die` for details.

7.7 Foreign Function Calls

The foreign function call interface allows a Lisp program to call many functions written in languages that use the C calling convention.

Lisp sets up various signal handling routines and other environment information when it first starts up, and expects these to be in place at all times. The C functions called by Lisp should not change the environment, especially the signal handlers: the signal handlers

installed by Lisp typically have interesting flags set (e.g to request machine context information, or for signal delivery on an alternate stack) which the Lisp runtime relies on for correct operation. Precise details of how this works may change without notice between versions; the source, or the brain of a friendly SBCL developer, is the only documentation. Users of a Lisp built with the `:sb-thread` feature should also read the section about threads, Chapter 11 [Threading], page 87.

7.7.1 The `alien-funcall` Primitive

`sb-alien:alien-funcall` *alien-function* **&rest** *arguments* [Function]

The `alien-funcall` function is the foreign function call primitive: *alien-function* is called with the supplied *arguments* and its C return value is returned as a Lisp value. The *alien-function* is an arbitrary run-time expression; to refer to a constant function, use `extern-alien` or a value defined by `define-alien-routine`.

The type of `alien-function` must be `(alien (function ...))` or `(alien (* (function ...)))`. The function type is used to determine how to call the function (as though it was declared with a prototype.) The type need not be known at compile time, but only known-type calls are efficiently compiled. Limitations:

- Structure type return values are not implemented.
- Passing of structures by value is not implemented.

Here is an example which allocates a `(struct foo)`, calls a foreign function to initialize it, then returns a Lisp vector of all the `(* (struct foo))` objects filled in by the foreign call:

```
;; Allocate a foo on the stack.
(with-alien ((f (struct foo)))
  ;; Call some C function to fill in foo fields.
  (alien-funcall (extern-alien "mangle_foo" (function void (* foo)))
                 (addr f))
  ;; Find how many foos to use by getting the A field.
  (let* ((num (slot f 'a))
         (result (make-array num)))
    ;; Get a pointer to the array so that we don't have to keep extracting it:
    (with-alien ((a (* (array (* (struct foo)) 100)) (addr (slot f 'b))))
      ;; Loop over the first N elements and stash them in the result vector.
      (dotimes (i num)
        (setf (svref result i) (deref (deref a) i)))
      ;; Voila.
      result)))
```

7.7.2 The `define-alien-routine` Macro

`sb-alien:define-alien-routine` *name result-type* **&rest** *arg-specifiers* [Macro]

The `define-alien-routine` macro is a convenience for automatically generating Lisp interfaces to simple foreign functions. The primary feature is the parameter style specification, which translates the C pass-by-reference idiom into additional return values.

name is usually a string external symbol, but may also be a symbol Lisp name or a list of the foreign name and the Lisp name. If only one name is specified, the other is automatically derived as for `extern-alien`. *result-type* is the alien type of the return value.

Each element of the *arg-specifiers* list specifies an argument to the foreign function, and is of the form

```
(aname atype &amp;optional style)
```

aname is the symbol name of the argument to the constructed function (for documentation). *atype* is the alien type of corresponding foreign argument. The semantics of the actual call are the same as for `alien-funcall`. *style* specifies how this argument should be handled at call and return time, and should be one of the following:

- `:in` specifies that the argument is passed by value. This is the default. `:in` arguments have no corresponding return value from the Lisp function.
- `:copy` is similar to `:in`, but the argument is copied to a pre-allocated object and a pointer to this object is passed to the foreign routine.
- `:out` specifies a pass-by-reference output value. The type of the argument must be a pointer to a fixed-sized object (such as an integer or pointer). `:out` and `:in-out` style cannot be used with pointers to arrays, records or functions. An object of the correct size is allocated on the stack, and its address is passed to the foreign function. When the function returns, the contents of this location are returned as one of the values of the Lisp function (and the location is automatically deallocated).
- `:in-out` is a combination of `:copy` and `:out`. The argument is copied to a pre-allocated object and a pointer to this object is passed to the foreign routine. On return, the contents of this location is returned as an additional value.

  Note: Any efficiency-critical foreign interface function should be inline expanded, which can be done by preceding the `define-alien-routine` call with:

  ```
  (declaim (inline lisp-name))
  ```

  In addition to avoiding the Lisp call overhead, this allows pointers, word-integers and floats to be passed using non-descriptor representations, avoiding consing.)

7.7.3 `define-alien-routine` Example

Consider the C function `cfoo` with the following calling convention:

```
void
cfoo (str, a, i)
    char *str;
    char *a; /* update */
    int *i; /* out */
{
  /* body of cfoo(...) */
}
```

This can be described by the following call to `define-alien-routine`:

```
(define-alien-routine "cfoo" void
  (str c-string)
  (a char :in-out)
  (i int :out))
```

The Lisp function `cfoo` will have two arguments (*str* and *a*) and two return values (*a* and *i*).

7.7.4 Calling Lisp From C

Calling Lisp functions from C is sometimes possible, but is extremely hackish and poorly supported as of SBCL 0.7.5. See `funcall0` ... `funcall3` in the runtime system. The arguments must be valid SBCL object descriptors (so that e.g. fixnums must be left-shifted by 2.) As of SBCL 0.7.5, the format of object descriptors is documented only by the source code and, in parts, by the old CMUCL '`INTERNALS`' documentation.

Note that the garbage collector moves objects, and won't be able to fix up any references in C variables. There are three mechanisms for coping with this:

1. The `sb-ext:purify` moves all live Lisp data into static or read-only areas such that it will never be moved (or freed) again in the life of the Lisp session
2. `sb-sys:with-pinned-objects` is a macro which arranges for some set of objects to be pinned in memory for the dynamic extent of its body forms. On ports which use the generational garbage collector (as of SBCL 0.8.3, only the x86) this has a page granularity - i.e. the entire 4k page or pages containing the objects will be locked down. On other ports it is implemented by turning off GC for the duration (so could be said to have a whole-world granularity).
3. Disable GC, using the `without-gcing` macro or `gc-off` call.

7.8 Step-By-Step Example of the Foreign Function Interface

This section presents a complete example of an interface to a somewhat complicated C function.

Suppose you have the following C function which you want to be able to call from Lisp in the file '`test.c`'

```
struct c_struct
{
  int x;
  char *s;
};

struct c_struct *c_function (i, s, r, a)
    int i;
    char *s;
    struct c_struct *r;
    int a[10];
{
  int j;
  struct c_struct *r2;
```

```
  printf("i = %d\n", i);
  printf("s = %s\n", s);
  printf("r->x = %d\n", r->x);
  printf("r->s = %s\n", r->s);
  for (j = 0; j < 10; j++) printf("a[%d] = %d.\n", j, a[j]);
  r2 = (struct c_struct *) malloc (sizeof(struct c_struct));
  r2->x = i + 5;
  r2->s = "a C string";
  return(r2);
};
```

It is possible to call this C function from Lisp using the file '`test.lisp`' containing

```
(cl:defpackage "TEST-C-CALL" (:use "CL" "SB-ALIEN" "SB-C-CALL"))
(cl:in-package "TEST-C-CALL")

;;; Define the record C-STRUCT in Lisp.
(define-alien-type nil
    (struct c-struct
            (x int)
            (s c-string)))

;;; Define the Lisp function interface to the C routine.  It returns a
;;; pointer to a record of type C-STRUCT.  It accepts four parameters:
;;; I, an int; S, a pointer to a string; R, a pointer to a C-STRUCT
;;; record; and A, a pointer to the array of 10 ints.
;;;
;;; The INLINE declaration eliminates some efficiency notes about heap
;;; allocation of alien values.
(declaim (inline c-function))
(define-alien-routine c-function
    (* (struct c-struct))
  (i int)
  (s c-string)
  (r (* (struct c-struct)))
  (a (array int 10)))

;;; a function which sets up the parameters to the C function and
;;; actually calls it
(defun call-cfun ()
  (with-alien ((ar (array int 10))
               (c-struct (struct c-struct)))
    (dotimes (i 10)                     ; Fill array.
      (setf (deref ar i) i))
    (setf (slot c-struct 'x) 20)
    (setf (slot c-struct 's) "a Lisp string")

    (with-alien ((res (* (struct c-struct))
```

```
                    (c-function 5 "another Lisp string" (addr c-struct) ar)))
(format t "~&amp;back from C function~%")
(multiple-value-prog1
    (values (slot res 'x)
            (slot res 's))

  ;; Deallocate result. (after we are done referring to it:
  ;; "Pillage, *then* burn.")
  (free-alien res)))))
```

To execute the above example, it is necessary to compile the C routine, e.g.: '`cc -c test.c && ld -shared -o test.so test.o`' (In order to enable incremental loading with some linkers, you may need to say '`cc -G 0 -c test.c`')

Once the C code has been compiled, you can start up Lisp and load it in: '`sbcl`'. Lisp should start up with its normal prompt.

Within Lisp, compile the Lisp file. (This step can be done separately. You don't have to recompile every time.) '`(compile-file "test.lisp")`'

Within Lisp, load the foreign object file to define the necessary symbols: '`(load-shared-object "test.so")`'.

Now you can load the compiled Lisp ("fasl") file into Lisp: '`(load "test.fasl")`' And once the Lisp file is loaded, you can call the Lisp routine that sets up the parameters and calls the C function: '`(test-c-call::call-cfun)`'

The C routine should print the following information to standard output:

```
i = 5
s = another Lisp string
r->x = 20
r->s = a Lisp string
a[0] = 0.
a[1] = 1.
a[2] = 2.
a[3] = 3.
a[4] = 4.
a[5] = 5.
a[6] = 6.
a[7] = 7.
a[8] = 8.
a[9] = 9.
```

After return from the C function, the Lisp wrapper function should print the following output:

```
back from C function
```

And upon return from the Lisp wrapper function, before the next prompt is printed, the Lisp read-eval-print loop should print the following return values:

```
10
"a C string"
```

8 Pathnames

8.1 Lisp Pathnames

There are many aspects of ANSI Common Lisp's pathname support which are implementation-defined and so need documentation.

8.1.1 The SYS Logical Pathname Host

The logical pathname host named by `"SYS"` exists in SBCL. Its `logical-pathname-translations` may be set by the site or the user applicable to point to the locations of the system's sources; in particular, the core system's source files match the logical pathname `"SYS:SRC;**;*.*.*"`, and the contributed modules' source files match `"SYS:CONTRIB;**;*.*.*"`.

8.2 Native Filenames

In some circumstances, what is wanted is a Lisp pathname object which corresponds to a string produced by the Operating System. In this case, some of the default parsing rules are inappropriate: most filesystems do not have a native understanding of wild pathnames; such functionality is often provided by shells above the OS, often in mutually-incompatible ways.

To allow the user to deal with this, the following functions are provided: `parse-native-namestring` and `native-pathname` return the closest equivalent Lisp pathname to a given string (appropriate for the Operating System), while `native-namestring` converts a non-wild pathname designator to the equivalent native namestring, if possible. Some Lisp pathname concepts (such as the `:back` directory component) have no direct equivalents in most Operating Systems; the behaviour of `native-namestring` is unspecified if an inappropriate pathname designator is passed to it. Additionally, note that conversion from pathname to native filename and back to pathname should not be expected to preserve equivalence under `equal`.

`sb-ext:parse-native-namestring` *thing* **&optional** *host defaults* **&key** *start end junk-allowed as-directory* [Function]

Convert `thing` into a pathname, using the native conventions appropriate for the pathname host `host`, or if not specified the host of `defaults`. If `thing` is a string, the parse is bounded by `start` and `end`, and error behaviour is controlled by `junk-allowed`, as with `parse-namestring`. For file systems whose native conventions allow directories to be indicated as files, if `as-directory` is true, return a pathname denoting `thing` as a directory.

`sb-ext:native-pathname` *pathspec* [Function]

Convert `pathspec` (a pathname designator) into a pathname, assuming the operating system native pathname conventions.

`sb-ext:native-namestring` *pathname* **&key** *as-file* [Function]

Construct the full native (name)string form of `pathname`. For file systems whose native conventions allow directories to be indicated as files, if `as-file` is true and

the name, type, and version components of `pathname` are all `nil` or `:unspecific`, construct a string that names the directory according to the file system's syntax for files.

Because some file systems permit the names of directories to be expressed in multiple ways, it is occasionally necessary to parse a native file name "as a directory name" or to produce a native file name that names a directory "as a file". For these cases, `parse-native-namestring` accepts the keyword argument `as-directory` to force a filename to parse as a directory, and `native-namestring` accepts the keyword argument `as-file` to force a pathname to unparse as a file. For example,

```
; On Unix, the directory "/tmp/" can be denoted by "/tmp/" or "/tmp".
; Under the default rules for native filenames, these parse and
; unparse differently.
(defvar *p*)
(setf *p* (parse-native-namestring "/tmp/")) ⇒ #P"/tmp/"
(pathname-name *p*) ⇒ NIL
(pathname-directory *p*) ⇒ (:ABSOLUTE "tmp")
(native-namestring *p*) ⇒ "/tmp/"

(setf *p* (parse-native-namestring "/tmp")) ⇒ #P"/tmp"
(pathname-name *p*) ⇒ "tmp"
(pathname-directory *p*) ⇒ (:ABSOLUTE)
(native-namestring *p*) ⇒ "/tmp"

; A non-NIL AS-DIRECTORY argument to PARSE-NATIVE-NAMESTRING forces
; both the second string to parse the way the first does.
(setf *p* (parse-native-namestring "/tmp"
                                   nil *default-pathname-defaults*
                                   :as-directory t)) ⇒ #P"/tmp/"
(pathname-name *p*) ⇒ NIL
(pathname-directory *p*) ⇒ (:ABSOLUTE "tmp")

; A non-NIL AS-FILE argument to NATIVE-NAMESTRING forces the pathname
; parsed from the first string to unparse as the second string.
(setf *p* (parse-native-namestring "/tmp/")) ⇒ #P"/tmp/"
(native-namestring *p* :as-file t) ⇒ "/tmp"
```

9 Extensible Streams

SBCL supports various extensions of ANSI Common Lisp streams.

Bivalent Streams
: A type of stream that can read and write both `character` and `(unsigned-byte 8)` values.

Gray Streams
: User-overloadable CLOS classes whose instances can be used as Lisp streams (e.g. passed as the first argument to `format`).

Simple Streams
: The bundled contrib module *sb-simple-streams* implements a subset of the Franz Allegro simple-streams proposal.

9.1 Bivalent Streams

A *bivalent stream* can be used to read and write both `character` and `(unsigned-byte 8)` values. A bivalent stream is created by calling `open` with the argument `:element-type :default`. On such a stream, both binary and character data can be read and written with the usual input and output functions.

> Streams are *not* created bivalent by default for performance reasons. Bivalent streams are incompatible with `fast-read-char`, an internal optimization in sbcl's stream machinery that bulk-converts octets to characters and implements a fast path through `read-char`.

9.2 Gray Streams

The Gray Streams interface is a widely supported extension that provides for definition of CLOS-extensible stream classes. Gray stream classes are implemented by adding methods to generic functions analogous to Common Lisp's standard I/O functions. Instances of Gray stream classes may be used with any I/O operation where a non-Gray stream can, provided that all required methods have been implemented suitably.

9.2.1 Gray Streams classes

The defined Gray Stream classes are these:

`sb-gray:fundamental-stream` [Class]
: Class precedence list: `fundamental-stream, standard-object, stream, t`

: the base class for all Gray streams

`sb-gray:fundamental-input-stream` [Class]
: Class precedence list: `fundamental-input-stream, fundamental-stream, standard-object, stream, t`

: a superclass of all Gray input streams

The function input-stream-p will return true of any generalized instance of fundamental-input-stream.

`sb-gray:fundamental-output-stream` [Class]

Class precedence list: `fundamental-output-stream`, `fundamental-stream`, `standard-object`, `stream`, `t`

a superclass of all Gray output streams

The function output-stream-p will return true of any generalized instance of fundamental-output-stream.

`sb-gray:fundamental-binary-stream` [Class]

Class precedence list: `fundamental-binary-stream`, `fundamental-stream`, `standard-object`, `stream`, `t`

a superclass of all Gray streams whose element-type is a subtype of unsigned-byte or signed-byte

Note that instantiable subclasses of fundamental-binary-stream should provide (or inherit) an applicable method for the generic function stream-element-type.

`sb-gray:fundamental-character-stream` [Class]

Class precedence list: `fundamental-character-stream`, `fundamental-stream`, `standard-object`, `stream`, `t`

a superclass of all Gray streams whose element-type is a subtype of character

`sb-gray:fundamental-binary-input-stream` [Class]

Class precedence list: `fundamental-binary-input-stream`, `fundamental-input-stream`, `fundamental-binary-stream`, `fundamental-stream`, `standard-object`, `stream`, `t`

a superclass of all Gray input streams whose element-type is a subtype of unsigned-byte or signed-byte

`sb-gray:fundamental-binary-output-stream` [Class]

Class precedence list: `fundamental-binary-output-stream`, `fundamental-output-stream`, `fundamental-binary-stream`, `fundamental-stream`, `standard-object`, `stream`, `t`

a superclass of all Gray output streams whose element-type is a subtype of unsigned-byte or signed-byte

`sb-gray:fundamental-character-input-stream` [Class]

Class precedence list: `fundamental-character-input-stream`, `fundamental-input-stream`, `fundamental-character-stream`, `fundamental-stream`, `standard-object`, `stream`, `t`

a superclass of all Gray input streams whose element-type is a subtype of character

`sb-gray:fundamental-character-output-stream` [Class]

Class precedence list: `fundamental-character-output-stream`, `fundamental-output-stream`, `fundamental-character-stream`, `fundamental-stream`, `standard-object`, `stream`, `t`

a superclass of all Gray output streams whose element-type is a subtype of character

9.2.2 Methods common to all streams

These generic functions can be specialized on any generalized instance of fundamental-stream.

`common-lisp:stream-element-type` *stream* [Generic Function]
Return a type specifier for the kind of object returned by the `stream`. The class `fundamental-character-stream` provides a default method which returns `character`.

`common-lisp:close` *stream* **&key** *abort* [Generic Function]
Close the given `stream`. No more I/O may be performed, but inquiries may still be made. If `:abort` is true, an attempt is made to clean up the side effects of having created the stream.

`sb-gray:stream-file-position` *stream* **&optional** *position-spec* [Generic Function]
Used by `file-position`. Returns or changes the current position within `stream`.

9.2.3 Input stream methods

These generic functions may be specialized on any generalized instance of fundamental-input-stream.

`sb-gray:stream-clear-input` *stream* [Generic Function]
This is like `cl:clear-input`, but for Gray streams, returning `nil`. The default method does nothing.

`sb-gray:stream-read-sequence` *stream seq* **&optional** *start end* [Generic Function]
This is like `cl:read-sequence`, but for Gray streams.

9.2.4 Character input stream methods

These generic functions are used to implement subclasses of fundamental-input-stream:

`sb-gray:stream-peek-char` *stream* [Generic Function]
This is used to implement `peek-char`; this corresponds to `peek-type` of `nil`. It returns either a character or `:eof`. The default method calls `stream-read-char` and `stream-unread-char`.

`sb-gray:stream-read-char-no-hang` *stream* [Generic Function]
This is used to implement `read-char-no-hang`. It returns either a character, or `nil` if no input is currently available, or `:eof` if end-of-file is reached. The default method provided by `fundamental-character-input-stream` simply calls `stream-read-char`; this is sufficient for file streams, but interactive streams should define their own method.

`sb-gray:stream-read-char` *stream* [Generic Function]
Read one character from the stream. Return either a character object, or the symbol `:eof` if the stream is at end-of-file. Every subclass of `fundamental-character-input-stream` must define a method for this function.

`sb-gray:stream-read-line` *stream* [Generic Function]
This is used by `read-line`. A string is returned as the first value. The second value is true if the string was terminated by end-of-file instead of the end of a line. The default method uses repeated calls to `stream-read-char`.

`sb-gray:stream-listen` *stream* [Generic Function]
This is used by `listen`. It returns true or false. The default method uses `stream-read-char-no-hang` and `stream-unread-char`. Most streams should define their own method since it will usually be trivial and will always be more efficient than the default method.

`sb-gray:stream-unread-char` *stream character* [Generic Function]
Un-do the last call to `stream-read-char`, as in `unread-char`. Return `nil`. Every subclass of `fundamental-character-input-stream` must define a method for this function.

9.2.5 Output stream methods

These generic functions are used to implement subclasses of fundamental-output-stream:

`sb-gray:stream-clear-output` *stream* [Generic Function]
This is like `cl:clear-output`, but for Gray streams: clear the given output `stream`. The default method does nothing.

`sb-gray:stream-finish-output` *stream* [Generic Function]
Attempts to ensure that all output sent to the Stream has reached its destination, and only then returns false. Implements `finish-output`. The default method does nothing.

`sb-gray:stream-force-output` *stream* [Generic Function]
Attempts to force any buffered output to be sent. Implements `force-output`. The default method does nothing.

`sb-gray:stream-write-sequence` *stream seq* **&optional** *start end* [Generic Function]
This is like `cl:write-sequence`, but for Gray streams.

9.2.6 Binary stream methods

The following generic functions are available for subclasses of fundamental-binary-stream:

`sb-gray:stream-read-byte` *stream* [Generic Function]
Used by `read-byte`; returns either an integer, or the symbol `:eof` if the stream is at end-of-file.

`sb-gray:stream-write-byte` *stream integer* [Generic Function]
Implements `write-byte`; writes the integer to the stream and returns the integer as the result.

9.2.7 Character output stream methods

These generic functions are used to implement subclasses of fundamental-character-output-stream:

`sb-gray:stream-advance-to-column` *stream column* [Generic Function]
Write enough blank space so that the next character will be written at the specified column. Returns true if the operation is successful, or `nil` if it is not supported for this stream. This is intended for use by by `pprint` and `format ~T`. The default method uses `stream-line-column` and repeated calls to `stream-write-char` with a #SPACE character; it returns `nil` if `stream-line-column` returns `nil`.

`sb-gray:stream-fresh-line` *stream* [Generic Function]
Outputs a new line to the Stream if it is not positioned at the begining of a line. Returns `t` if it output a new line, nil otherwise. Used by `fresh-line`. The default method uses `stream-start-line-p` and `stream-terpri`.

`sb-gray:stream-line-column` *stream* [Generic Function]
Return the column number where the next character will be written, or `nil` if that is not meaningful for this stream. The first column on a line is numbered 0. This function is used in the implementation of `pprint` and the `format ~T` directive. For every character output stream class that is defined, a method must be defined for this function, although it is permissible for it to always return `nil`.

`sb-gray:stream-line-length` *stream* [Generic Function]
Return the stream line length or `nil`.

`sb-gray:stream-start-line-p` *stream* [Generic Function]
Is `stream` known to be positioned at the beginning of a line? It is permissible for an implementation to always return `nil`. This is used in the implementation of `fresh-line`. Note that while a value of 0 from `stream-line-column` also indicates the beginning of a line, there are cases where `stream-start-line-p` can be meaningfully implemented although `stream-line-column` can't be. For example, for a window using variable-width characters, the column number isn't very meaningful, but the beginning of the line does have a clear meaning. The default method for `stream-start-line-p` on class `fundamental-character-output-stream` uses `stream-line-column`, so if that is defined to return `nil`, then a method should be provided for either `stream-start-line-p` or `stream-fresh-line`.

`sb-gray:stream-terpri` *stream* [Generic Function]
Writes an end of line, as for `terpri`. Returns `nil`. The default method does (STREAM-WRITE-CHAR stream #NEWLINE).

`sb-gray:stream-write-char` *stream character* [Generic Function]
Write `character` to `stream` and return `character`. Every subclass of `fundamental-character-output-stream` must have a method defined for this function.

`sb-gray:stream-write-string` *stream string* **&optional** *start end* [Generic Function]
This is used by `write-string`. It writes the string to the stream, optionally delimited by start and end, which default to 0 and `nil`. The string argument is returned. The default method provided by `fundamental-character-output-stream` uses repeated calls to `stream-write-char`.

9.2.8 Gray Streams examples

Below are two classes of stream that can be conveniently defined as wrappers for Common Lisp streams. These are meant to serve as examples of minimal implementations of the protocols that must be followed when defining Gray streams. Realistic uses of the Gray Streams API would implement the various methods that can do I/O in batches, such as `stream-read-line`, `stream-write-string`, `stream-read-sequence`, and `stream-write-sequence`.

9.2.8.1 Character counting input stream

It is occasionally handy for programs that process input files to count the number of characters and lines seen so far, and the number of characters seen on the current line, so that useful messages may be reported in case of parsing errors, etc. Here is a character input stream class that keeps track of these counts. Note that all character input streams must implement `stream-read-char` and `stream-unread-char`.

```
(defclass wrapped-stream (fundamental-stream)
  ((stream :initarg :stream :reader stream-of)))

(defmethod stream-element-type ((stream wrapped-stream))
  (stream-element-type (stream-of stream)))

(defmethod close ((stream wrapped-stream) &key abort)
  (close (stream-of stream) :abort abort))

(defclass wrapped-character-input-stream
    (wrapped-stream fundamental-character-input-stream)
  ())

(defmethod stream-read-char ((stream wrapped-character-input-stream))
  (read-char (stream-of stream) nil :eof))

(defmethod stream-unread-char ((stream wrapped-character-input-stream)
                               char)
  (unread-char char (stream-of stream)))
```

```
(defclass counting-character-input-stream
    (wrapped-character-input-stream)
  ((char-count :initform 1 :accessor char-count-of)
   (line-count :initform 1 :accessor line-count-of)
   (col-count :initform 1 :accessor col-count-of)
   (prev-col-count :initform 1 :accessor prev-col-count-of)))

(defmethod stream-read-char ((stream counting-character-input-stream))
  (with-accessors ((inner-stream stream-of) (chars char-count-of)
                   (lines line-count-of) (cols col-count-of)
                   (prev prev-col-count-of)) stream
      (let ((char (call-next-method)))
        (cond ((eql char :eof)
               :eof)
              ((char= char #\Newline)
               (incf lines)
               (incf chars)
               (setf prev cols)
               (setf cols 1)
               char)
              (t
               (incf chars)
               (incf cols)
               char)))))

(defmethod stream-unread-char ((stream counting-character-input-stream)
                               char)
  (with-accessors ((inner-stream stream-of) (chars char-count-of)
                   (lines line-count-of) (cols col-count-of)
                   (prev prev-col-count-of)) stream
      (cond ((char= char #\Newline)
             (decf lines)
             (decf chars)
             (setf cols prev))
            (t
             (decf chars)
             (decf cols)
             char))
      (call-next-method)))
```

The default methods for `stream-read-char-no-hang`, `stream-peek-char`, `stream-listen`, `stream-clear-input`, `stream-read-line`, and `stream-read-sequence` should be sufficient (though the last two will probably be slower than methods that forwarded directly).

Here's a sample use of this class:

```
(with-input-from-string (input "1 2
 3 :foo  ")
  (let ((counted-stream (make-instance 'counting-character-input-stream
                          :stream input)))
    (loop for thing = (read counted-stream) while thing
       unless (numberp thing) do
         (error "Non-number ~S (line ~D, column ~D)" thing
                (line-count-of counted-stream)
                (- (col-count-of counted-stream)
                   (length (format nil "~S" thing))))
       end
       do (print thing))))
1
2
3
Non-number :FOO (line 2, column 5)
  [Condition of type SIMPLE-ERROR]
```

9.2.8.2 Output prefixing character stream

One use for a wrapped output stream might be to prefix each line of text with a time-stamp, e.g., for a logging stream. Here's a simple stream that does this, though without any fancy line-wrapping. Note that all character output stream classes must implement `stream-write-char` and `stream-line-column`.

```
(defclass wrapped-stream (fundamental-stream)
  ((stream :initarg :stream :reader stream-of)))

(defmethod stream-element-type ((stream wrapped-stream))
  (stream-element-type (stream-of stream)))

(defmethod close ((stream wrapped-stream) &key abort)
  (close (stream-of stream) :abort abort))

(defclass wrapped-character-output-stream
    (wrapped-stream fundamental-character-output-stream)
  ((col-index :initform 0 :accessor col-index-of)))

(defmethod stream-line-column ((stream wrapped-character-output-stream))
  (col-index-of stream))

(defmethod stream-write-char ((stream wrapped-character-output-stream)
                              char)
  (with-accessors ((inner-stream stream-of) (cols col-index-of)) stream
    (write-char char inner-stream)
    (if (char= char #\Newline)
        (setf cols 0)
        (incf cols))))
```

```
(defclass prefixed-character-output-stream
    (wrapped-character-output-stream)
  ((prefix :initarg :prefix :reader prefix-of)))

(defgeneric write-prefix (prefix stream)
  (:method ((prefix string) stream) (write-string prefix stream))
  (:method ((prefix function) stream) (funcall prefix stream)))

(defmethod stream-write-char ((stream prefixed-character-output-stream)
                              char)
  (with-accessors ((inner-stream stream-of) (cols col-index-of)
                   (prefix prefix-of)) stream
    (when (zerop cols)
      (write-prefix prefix inner-stream))
    (call-next-method)))
```

As with the example input stream, this implements only the minimal protocol. A production implementation should also provide methods for at least `stream-write-line`, `stream-write-sequence`.

And here's a sample use of this class:

```
(flet ((format-timestamp (stream)
         (apply #'format stream "[~2@*~2,' D:~1@*~2,'0D:~0@*~2,'0D] "
                (multiple-value-list (get-decoded-time)))))
  (let ((output (make-instance 'prefixed-character-output-stream
                               :stream *standard-output*
                               :prefix #'format-timestamp)))
    (loop for string in '("abc" "def" "ghi") do
         (write-line string output)
         (sleep 1))))
[ 0:30:05] abc
[ 0:30:06] def
[ 0:30:07] ghi
NIL
```

9.3 Simple Streams

Simple streams are an extensible streams protocol that avoids some problems with Gray streams.

Documentation about simple streams is available at:

http://www.franz.com/support/documentation/6.2/doc/streams.htm

The implementation should be considered Alpha-quality; the basic framework is there, but many classes are just stubs at the moment.

See '`SYS:CONTRIB;SB-SIMPLE-STREAMS;SIMPLE-STREAM-TEST.LISP`' for things that should work.

Known differences to the ACL behaviour:

- `open` not return a simple-stream by default. This can be adjusted; see default-open-class in the file cl.lisp
- `write-vector` is unimplemented.

10 Package Locks

None of the following sections apply to SBCL built without package locking support.

The interface described here is experimental: incompatible changes in future SBCL releases are possible, even expected: the concept of "implementation packages" and the associated operators may be renamed; more operations (such as naming restarts or catch tags) may be added to the list of operations violating package locks.

10.1 Package Lock Concepts

10.1.1 Package Locking Overview

Package locks protect against unintentional modifications of a package: they provide similar protection to user packages as is mandated to `common-lisp` package by the ANSI specification. They are not, and should not be used as, a security measure.

Newly created packages are by default unlocked (see the `:lock` option to `defpackage`).

The package `common-lisp` and SBCL internal implementation packages are locked by default, including `sb-ext`.

It may be beneficial to lock `common-lisp-user` as well, to ensure that various libraries don't pollute it without asking, but this is not currently done by default.

10.1.2 Implementation Packages

Each package has a list of associated implementation packages. A locked package, and the symbols whose home package it is, can be modified without violating package locks only when `*package*` is bound to one of the implementation packages of the locked package.

Unless explicitly altered by `defpackage`, `sb-ext:add-implementation-package`, or `sb-ext:remove-implementation-package` each package is its own (only) implementation package.

10.1.3 Package Lock Violations

10.1.3.1 Lexical Bindings and Declarations

Lexical bindings or declarations that violate package locks cause result in a `program-error` being signalled at when the form that violates package locks would be executed.

A complete listing of operators affect by this is: `let`, `let*`, `flet`, `labels`, `macrolet`, and `symbol-macrolet`, `declare`.

Package locks affecting both lexical bindings and declarations can be disabled locally with `sb-ext:disable-package-locks` declaration, and re-enabled with `sb-ext:enable-package-locks` declaration.

Example:

```
(in-package :locked)

(defun foo () ...)

(defmacro with-foo (&body body)
```

```
`(locally (declare (disable-package-locks locked:foo))
   (flet ((foo () ...))
     (declare (enable-package-locks locked:foo)) ; re-enable for body
     ,@body)))
```

10.1.3.2 Other Operations

If an non-lexical operation violates a package lock, a continuable error that is of a subtype of `sb-ext:package-lock-violation` (subtype of `package-error`) is signalled when the operation is attempted.

Additional restarts may be established for continuable package lock violations for interactive use.

The actual type of the error depends on circumstances that caused the violation: operations on packages signal errors of type `sb-ext:package-locked-error`, and operations on symbols signal errors of type `sb-ext:symbol-package-locked-error`.

10.1.4 Package Locks in Compiled Code

10.1.4.1 Interned Symbols

If file-compiled code contains interned symbols, then loading that code into an image without the said symbols will not cause a package lock violation, even if the packages in question are locked.

10.1.4.2 Other Limitations on Compiled Code

With the exception of interned symbols, behaviour is unspecified if package locks affecting compiled code are not the same during loading of the code or execution.

Specifically, code compiled with packages unlocked may or may not fail to signal package-lock-violations even if the packages are locked at runtime, and code compiled with packages locked may or may not signal spurious package-lock-violations at runtime even if the packages are unlocked.

In practice all this means that package-locks have a negligible performance penalty in compiled code as long as they are not violated.

10.1.5 Operations Violating Package Locks

10.1.5.1 Operations on Packages

The following actions cause a package lock violation if the package operated on is locked, and `*package*` is not an implementation package of that package, and the action would cause a change in the state of the package (so e.g. exporting already external symbols is never a violation). Package lock violations caused by these operations signal errors of type `sb-ext:package-locked-error`.

1. Shadowing a symbol in a package.
2. Importing a symbol to a package.
3. Uninterning a symbol from a package.
4. Exporting a symbol from a package.
5. Unexporting a symbol from a package.

6. Changing the packages used by a package.
7. Renaming a package.
8. Deleting a package.

10.1.5.2 Operations on Symbols

Following actions cause a package lock violation if the home package of the symbol operated on is locked, and `*package*` is not an implementation package of that package. Package lock violations caused by these action signal errors of type `sb-ext:symbol-package-locked-error`.

These actions cause only one package lock violation per lexically apparent violated package.

Example:

```
;;; Packages FOO and BAR are locked.
;;;
;;; Two lexically apparent violated packages: exactly two
;;; package-locked-errors will be signalled.

(defclass foo:point ()
  ((x :accessor bar:x)
   (y :accessor bar:y)))
```

1. Binding or altering its value lexically or dynamically, or establishing it as a symbol-macro.

   Exceptions:

   – If the symbol is not defined as a constant, global symbol-macro or a global dynamic variable, it may be lexically bound or established as a local symbol macro.
   – If the symbol is defined as a global dynamic variable, it may be assigned or bound.
2. Defining, undefining, or binding it, or its setf name as a function.

   Exceptions:

   – If the symbol is not defined as a function, macro, or special operator it and its setf name may be lexically bound as a function.
3. Defining, undefining, or binding it as a macro or compiler macro.

   Exceptions:

   – If the symbol is not defined as a function, macro, or special operator it may be lexically bound as a macro.
4. Defining it as a type specifier or structure.
5. Defining it as a declaration with a declaration proclamation.
6. Declaring or proclaiming it special.
7. Declaring or proclaiming its type or ftype.

   Exceptions:

   – If the symbol may be lexically bound, the type of that binding may be declared.
   – If the symbol may be lexically bound as a function, the ftype of that binding may be declared.

8. Defining a setf expander for it.
9. Defining it as a method combination type.
10. Using it as the class-name argument to setf of find-class.

10.2 Package Lock Dictionary

`sb-ext:disable-package-locks` [Declaration]

Syntax: `(sb-ext:disable-package-locks symbol*)`

Disables package locks affecting the named symbols during compilation in the lexical scope of the declaration. Disabling locks on symbols whose home package is unlocked, or disabling an already disabled lock, has no effect.

`sb-ext:enable-package-locks` [Declaration]

Syntax: `(sb-ext:enable-package-locks symbol*)`

Re-enables package locks affecting the named symbols during compilation in the lexical scope of the declaration. Enabling locks that were not first disabled with `sb-ext:disable-package-locks` declaration, or enabling locks that are already enabled has no effect.

`sb-ext:package-lock-violation` [Condition]

Class precedence list: `package-lock-violation`, `package-error`, `error`, `serious-condition`, `condition`, `t`

Subtype of `cl:package-error`. A subtype of this error is signalled when a package-lock is violated.

`sb-ext:package-locked-error` [Condition]

Class precedence list: `package-locked-error`, `package-lock-violation`, `package-error`, `error`, `serious-condition`, `condition`, `t`

Subtype of `sb-ext:package-lock-violation`. An error of this type is signalled when an operation on a package violates a package lock.

`sb-ext:symbol-package-locked-error` [Condition]

Class precedence list: `symbol-package-locked-error`, `package-lock-violation`, `package-error`, `error`, `serious-condition`, `condition`, `t`

Subtype of `sb-ext:package-lock-violation`. An error of this type is signalled when an operation on a symbol violates a package lock. The symbol that caused the violation is accessed by the function `sb-ext:package-locked-error-symbol`.

`sb-ext:package-locked-error-symbol` *symbol-package-locked-error* [Function]

Returns the symbol that caused the `symbol-package-locked-error` condition.

`sb-ext:package-locked-p` *package* [Function]

Returns `t` when `package` is locked, `nil` otherwise. Signals an error if `package` doesn't designate a valid package.

`sb-ext:lock-package` *package* [Function]
Locks `package` and returns `t`. Has no effect if `package` was already locked. Signals an error if `package` is not a valid package designator

`sb-ext:unlock-package` *package* [Function]
Unlocks `package` and returns `t`. Has no effect if `package` was already unlocked. Signals an error if `package` is not a valid package designator.

`sb-ext:package-implemented-by-list` *package* [Function]
Returns a list containing the implementation packages of `package`. Signals an error if `package` is not a valid package designator.

`sb-ext:package-implements-list` *package* [Function]
Returns the packages that `package` is an implementation package of. Signals an error if `package` is not a valid package designator.

`sb-ext:add-implementation-package` *packages-to-add* **&optional** *package* [Function]
Adds `packages-to-add` as implementation packages of `package`. Signals an error if `package` or any of the `packages-to-add` is not a valid package designator.

`sb-ext:remove-implementation-package` *packages-to-remove* **&optional** *package* [Function]
Removes `packages-to-remove` from the implementation packages of `package`. Signals an error if `package` or any of the `packages-to-remove` is not a valid package designator.

`sb-ext:without-package-locks` **&body** *body* [Macro]
Ignores all runtime package lock violations during the execution of body. Body can begin with declarations.

`sb-ext:with-unlocked-packages` (**&rest** *packages*) **&body** *forms* [Macro]
Unlocks `packages` for the dynamic scope of the body. Signals an error if any of `packages` is not a valid package designator.

`defpackage` *name* [[*option*]]* ⇒ *package* [Macro]
Options are extended to include the following:

- `:lock` *boolean*

  If the argument to `:lock` is `t`, the package is initially locked. If `:lock` is not provided it defaults to `nil`.

- `:implement` *package-designator**

  The package is added as an implementation package to the packages named. If `:implement` is not provided, it defaults to the package itself.

Example:

```
(defpackage "FOO" (:export "BAR") (:lock t) (:implement))
(defpackage "FOO-INT" (:use "FOO") (:implement "FOO" "FOO-INT"))

;;; is equivalent to

(defpackage "FOO") (:export "BAR"))
(lock-package "FOO")
(remove-implementation-package "FOO" "FOO")
(defpackage "FOO-INT" (:use "BAR"))
(add-implementation-package "FOO-INT" "FOO")
```

11 Threading

SBCL supports a fairly low-level threading interface that maps onto the host operating system's concept of threads or lightweight processes. This means that threads may take advantage of hardware multiprocessing on machines that have more than one CPU, but it does not allow Lisp control of the scheduler. This is found in the SB-THREAD package.

This requires Linux (2.6+ or systems with NPTL backports) running on the x86 or x86-64 architecture, or SunOS (Solaris) on the x86. Support for threading on Darwin (Mac OS X) and FreeBSD on the x86 is experimental.

11.1 Threading basics

```
(make-thread (lambda () (write-line "Hello, world")))
```

`sb-thread:thread` [Structure]

Class precedence list: `thread, structure-object, t`

Thread type. Do not rely on threads being structs as it may change in future versions.

`sb-thread:*current-thread*` [Variable]

Bound in each thread to the thread itself.

`sb-thread:make-thread` *function* **&key** *name* [Function]

Create a new thread of `name` that runs `function`. When the function returns the thread exits. The return values of `function` are kept around and can be retrieved by `join-thread`.

`sb-thread:join-thread` *thread* **&key** *default* [Function]

Suspend current thread until `thread` exits. Returns the result values of the thread function. If the thread does not exit normally, return `default` if given or else signal `join-thread-error`.

`sb-thread:join-thread-error` [Condition]

Class precedence list: `join-thread-error, error, serious-condition, condition, t`

Joining thread failed.

`sb-thread:join-thread-error-thread` *condition* [Function]

The thread that we failed to join.

`sb-thread:thread-alive-p` *thread* [Function]

Check if `thread` is running.

`sb-thread:list-all-threads` [Function]

Return a list of the live threads.

`sb-thread:interrupt-thread-error` [Condition]
Class precedence list: `interrupt-thread-error, error, serious-condition, condition, t`

Interrupting thread failed.

`sb-thread:interrupt-thread-error-thread` *condition* [Function]
The thread that was not interrupted.

`sb-thread:interrupt-thread` *thread function* [Function]
Interrupt the live `thread` and make it run `function`. A moderate degree of care is expected for use of `interrupt-thread`, due to its nature: if you interrupt a thread that was holding important locks then do something that turns out to need those locks, you probably won't like the effect.

`sb-thread:terminate-thread` *thread* [Function]
Terminate the thread identified by `thread`, by causing it to run `sb-ext:quit` - the usual cleanup forms will be evaluated

`sb-thread:thread-yield` [Function]
Yield the processor to other threads.

11.2 Special Variables

The interaction of special variables with multiple threads is mostly as one would expect, with behaviour very similar to other implementations.

- global special values are visible across all threads;
- bindings (e.g. using LET) are local to the thread;
- threads do not inherit dynamic bindings from the parent thread

The last point means that

```
(defparameter *x* 0)
(let ((*x* 1))
  (sb-thread:make-thread (lambda () (print *x*))))
```

prints `0` and not `1` as of 0.9.6.

11.3 Mutex Support

Mutexes are used for controlling access to a shared resource. One thread is allowed to hold the mutex, others which attempt to take it will be made to wait until it's free. Threads are woken in the order that they go to sleep.

There isn't a timeout on mutex acquisition, but the usual WITH-TIMEOUT macro (which throws a TIMEOUT condition after n seconds) can be used if you want a bounded wait.

```
(defpackage :demo (:use "CL" "SB-THREAD" "SB-EXT"))

(in-package :demo)
```

```
(defvar *a-mutex* (make-mutex :name "my lock"))

(defun thread-fn ()
  (format t "Thread ~A running ~%" *current-thread*)
  (with-mutex (*a-mutex*)
    (format t "Thread ~A got the lock~%" *current-thread*)
    (sleep (random 5)))
  (format t "Thread ~A dropped lock, dying now~%" *current-thread*))

(make-thread #'thread-fn)
(make-thread #'thread-fn)
```

`sb-thread:mutex` [Structure]

Class precedence list: `mutex, structure-object, t`

Mutex type.

`sb-thread:make-mutex` **&key** *name %owner* [Function]

Create a mutex.

`sb-thread:mutex-name` *instance* [Function]

The name of the mutex. Setfable.

`sb-thread:mutex-value` *mutex* [Function]

Current owner of the mutex, `nil` if the mutex is free.

`sb-thread:get-mutex` *mutex* **&optional** *new-owner waitp* [Function]

Acquire `mutex` for `new-owner`, which must be a thread or `nil`. If `new-owner` is `nil`, it defaults to the current thread. If `waitp` is non-NIL and the mutex is in use, sleep until it is available.

Note: using `get-mutex` to assign a `mutex` to another thread then the current one is not recommended, and liable to be deprecated.

`get-mutex` is not interrupt safe. The correct way to call it is:

```
(WITHOUT-INTERRUPTS
  ...
  (ALLOW-WITH-INTERRUPTS (GET-MUTEX ...))
  ...)
```

`without-interrupts` is necessary to avoid an interrupt unwinding the call while the mutex is in an inconsistent state while `allow-with-interrupts` allows the call to be interrupted from sleep.

It is recommended that you use `with-mutex` instead of calling `get-mutex` directly.

`sb-thread:release-mutex` *mutex* [Function]

Release `mutex` by setting it to `nil`. Wake up threads waiting for this mutex.

`release-mutex` is not interrupt safe: interrupts should be disabled around calls to it.

Signals a `warning` is current thread is not the current owner of the mutex.

`sb-thread:with-mutex` (*mutex* **&key** *value wait-p*) **&body** *body* [Macro]
Acquire `mutex` for the dynamic scope of `body`, setting it to `new-value` or some suitable default value if `nil`. If `wait-p` is non-NIL and the mutex is in use, sleep until it is available

`sb-thread:with-recursive-lock` (*mutex*) **&body** *body* [Macro]
Acquires `mutex` for the dynamic scope of `body`. Within that scope further recursive lock attempts for the same mutex succeed. It is allowed to mix `with-mutex` and `with-recursive-lock` for the same mutex provided the default value is used for the mutex.

11.4 Semaphores

described here should be considered experimental, subject to API changes without notice.

`sb-thread:semaphore` [Structure]
Class precedence list: `semaphore, structure-object, t`

Semaphore type. The fact that a `semaphore` is a `structure-object` should be considered an implementation detail, and may change in the future.

`sb-thread:make-semaphore` **&key** *name count* [Function]
Create a semaphore with the supplied `count` and `name`.

`sb-thread:semaphore-count` *instance* [Function]
Returns the current count of the semaphore `instance`.

`sb-thread:semaphore-name` *instance* [Function]
The name of the semaphore `instance`. Setfable.

`sb-thread:signal-semaphore` *semaphore* **&optional** *n* [Function]
Increment the count of `semaphore` by `n`. If there are threads waiting on this semaphore, then `n` of them is woken up.

`sb-thread:wait-on-semaphore` *semaphore* [Function]
Decrement the count of `semaphore` if the count would not be negative. Else blocks until the semaphore can be decremented.

11.5 Waitqueue/condition variables

These are based on the POSIX condition variable design, hence the annoyingly CL-conflicting name. For use when you want to check a condition and sleep until it's true. For example: you have a shared queue, a writer process checking "queue is empty" and one or more readers that need to know when "queue is not empty". It sounds simple, but is astonishingly easy to deadlock if another process runs when you weren't expecting it to.

There are three components:

- the condition itself (not represented in code)

- the condition variable (a.k.a waitqueue) which proxies for it
- a lock to hold while testing the condition

Important stuff to be aware of:

- when calling condition-wait, you must hold the mutex. condition-wait will drop the mutex while it waits, and obtain it again before returning for whatever reason;
- likewise, you must be holding the mutex around calls to condition-notify;
- a process may return from condition-wait in several circumstances: it is not guaranteed that the underlying condition has become true. You must check that the resource is ready for whatever you want to do to it.

```
(defvar *buffer-queue* (make-waitqueue))
(defvar *buffer-lock* (make-mutex :name "buffer lock"))

(defvar *buffer* (list nil))

(defun reader ()
  (with-mutex (*buffer-lock*)
    (loop
     (condition-wait *buffer-queue* *buffer-lock*)
     (loop
      (unless *buffer* (return))
      (let ((head (car *buffer*)))
        (setf *buffer* (cdr *buffer*))
        (format t "reader ~A woke, read ~A~%"
                *current-thread* head))))))

(defun writer ()
  (loop
   (sleep (random 5))
   (with-mutex (*buffer-lock*)
     (let ((el (intern
                (string (code-char
                         (+ (char-code #\A) (random 26)))))))
       (setf *buffer* (cons el *buffer*)))
     (condition-notify *buffer-queue*))))

(make-thread #'writer)
(make-thread #'reader)
(make-thread #'reader)
```

`sb-thread:waitqueue` [Structure]

Class precedence list: `waitqueue, structure-object, t`

Waitqueue type.

`sb-thread:make-waitqueue` **&key** *name* [Function]

Create a waitqueue.

`sb-thread:waitqueue-name` *instance* [Function]
The name of the waitqueue. Setfable.

`sb-thread:condition-wait` *queue mutex* [Function]
Atomically release `mutex` and enqueue ourselves on `queue`. Another thread may subsequently notify us using `condition-notify`, at which time we reacquire `mutex` and return to the caller.

`sb-thread:condition-notify` *queue* **&optional** *n* [Function]
Notify `n` threads waiting on `queue`.

`sb-thread:condition-broadcast` *queue* [Function]
Notify all threads waiting on `queue`.

11.6 Sessions/Debugging

If the user has multiple views onto the same Lisp image (for example, using multiple terminals, or a windowing system, or network access) they are typically set up as multiple *sessions* such that each view has its own collection of foreground/background/stopped threads. A thread which wishes to create a new session can use `sb-thread:with-new-session` to remove itself from the current session (which it shares with its parent and siblings) and create a fresh one. # See also `sb-thread:make-listener-thread`.

Within a single session, threads arbitrate between themselves for the user's attention. A thread may be in one of three notional states: foreground, background, or stopped. When a background process attempts to print a repl prompt or to enter the debugger, it will stop and print a message saying that it has stopped. The user at his leisure may switch to that thread to find out what it needs. If a background thread enters the debugger, selecting any restart will put it back into the background before it resumes. Arbitration for the input stream is managed by calls to `sb-thread:get-foreground` (which may block) and `sb-thread:release-foreground`.

`sb-ext:quit` terminates all threads in the current session, but leaves other sessions running.

11.7 Implementation (Linux x86/x86-64)

Threading is implemented using pthreads and some Linux specific bits like futexes.

On x86 the per-thread local bindings for special variables is achieved using the %fs segment register to point to a per-thread storage area. This may cause interesting results if you link to foreign code that expects threading or creates new threads, and the thread library in question uses %fs in an incompatible way. On x86-64 the r12 register has a similar role.

Queues require the `sys_futex()` system call to be available: this is the reason for the NPTL requirement. We test at runtime that this system call exists.

Garbage collection is done with the existing Conservative Generational GC. Allocation is done in small (typically 8k) regions: each thread has its own region so this involves no stopping. However, when a region fills, a lock must be obtained while another is allocated,

and when a collection is required, all processes are stopped. This is achieved by sending them signals, which may make for interesting behaviour if they are interrupted in system calls. The streams interface is believed to handle the required system call restarting correctly, but this may be a consideration when making other blocking calls e.g. from foreign library code.

Large amounts of the SBCL library have not been inspected for thread-safety. Some of the obviously unsafe areas have large locks around them, so compilation and fasl loading, for example, cannot be parallelized. Work is ongoing in this area.

A new thread by default is created in the same POSIX process group and session as the thread it was created by. This has an impact on keyboard interrupt handling: pressing your terminal's intr key (typically *Control-C*) will interrupt all processes in the foreground process group, including Lisp threads that SBCL considers to be notionally 'background'. This is undesirable, so background threads are set to ignore the SIGINT signal.

`sb-thread:make-listener-thread` in addition to creating a new Lisp session makes a new POSIX session, so that pressing *Control-C* in one window will not interrupt another listener - this has been found to be embarrassing.

12 Timers

SBCL supports a system-wide scheduler implemented on top of `setitimer` that also works with threads but does not require a separate scheduler thread.

```
(schedule-timer (make-timer (lambda ()
                              (write-line "Hello, world")
                              (force-output)))
                2)
```

`sb-ext:timer` [Structure]

Class precedence list: `timer, structure-object, t`

Timer type. Do not rely on timers being structs as it may change in future versions.

`sb-ext:make-timer` *function* **&key** *name thread* [Function]

Create a timer object that's when scheduled runs `function`. If `thread` is a thread then that thread is to be interrupted with `function`. If `thread` is `t` then a new thread is created each timer `function` is run. If `thread` is `nil` then `function` can be run in any thread.

`sb-ext:timer-name` *timer* [Function]

Return the name of `timer`.

`sb-ext:timer-scheduled-p` *timer* **&key** *delta* [Function]

See if `timer` will still need to be triggered after `delta` seconds from now. For timers with a repeat interval it returns true.

`sb-ext:schedule-timer` *timer time* **&key** *repeat-interval absolute-p* [Function]

Schedule `timer` to be triggered at `time`. If `absolute-p` then `time` is universal time, but non-integral values are also allowed, else `time` is measured as the number of seconds from the current time. If `repeat-interval` is given, `timer` is automatically rescheduled upon expiry.

`sb-ext:unschedule-timer` *timer* [Function]

Cancel `timer`. Once this function returns it is guaranteed that `timer` shall not be triggered again and there are no unfinished triggers.

`sb-ext:list-all-timers` [Function]

Return a list of all timers in the system.

13 Networking

The `sb-bsd-sockets` module provides a thinly disguised BSD socket API for SBCL. Ideas have been stolen from the BSD socket API for C and Graham Barr's IO::Socket classes for Perl.

Sockets are represented as CLOS objects, and the API naming conventions attempt to balance between the BSD names and good lisp style.

13.1 Sockets Overview

Most of the functions are modelled on the BSD socket API. BSD sockets are widely supported, portably *("portable" by Unix standards, at least)* available on a variety of systems, and documented. There are some differences in approach where we have taken advantage of some of the more useful features of Common Lisp - briefly:

- Where the C API would typically return -1 and set `errno`, `sb-bsd-sockets` signals an error. All the errors are subclasses of `sb-bsd-sockets:socket-condition` and generally correspond one for one with possible `errno` values.
- We use multiple return values in many places where the C API would use pass-by-reference values.
- We can often avoid supplying an explicit *length* argument to functions because we already know how long the argument is.
- IP addresses and ports are represented in slightly friendlier fashion than "network-endian integers".

13.2 General Sockets

`sb-bsd-sockets:socket` [Class]

Class precedence list: `socket, standard-object, t`

Slots:

- `protocol` — initarg: `:protocol`; reader: `sb-bsd-sockets:socket-protocol`
  Protocol used by the socket. If a keyword, the symbol-name of the keyword will be passed to `get-protocol-by-name` downcased, and the returned value used as protocol. Other values are used as-is.
- `type` — initarg: `:type`; reader: `sb-bsd-sockets:socket-type`
  Type of the socket: `:stream` or `:datagram`.

Common base class of all sockets, not meant to be directly instantiated.

`sb-bsd-sockets:socket-bind` *socket* **&rest** *address* [Generic Function]

Bind `socket` to `address`, which may vary according to socket family. For the `inet` family, pass `address` and `port` as two arguments; for `file` address family sockets, pass the filename string. See also bind(2)

`sb-bsd-sockets:socket-accept` *socket* [Generic Function]

Perform the accept(2) call, returning a newly-created connected socket and the peer address as multiple values

`sb-bsd-sockets:socket-connect` *socket* **&rest** *address* [Generic Function]
Perform the connect(2) call to connect `socket` to a remote `peer`. No useful return value.

`sb-bsd-sockets:socket-peername` *socket* [Generic Function]
Return the socket's peer; depending on the address family this may return multiple values

`sb-bsd-sockets:socket-name` *socket* [Generic Function]
Return the address (as vector of bytes) and port that the socket is bound to, as multiple values.

`sb-bsd-sockets:socket-receive` *socket buffer length* **&key** [Generic Function]
oob peek waitall element-type element-type
Read `length` octets from `socket` into `buffer` (or a freshly-consed buffer if NIL), using recvfrom(2). If `length` is `nil`, the length of `buffer` is used, so at least one of these two arguments must be non-NIL. If `buffer` is supplied, it had better be of an element type one octet wide. Returns the buffer, its length, and the address of the peer that sent it, as multiple values. On datagram sockets, sets MSG_TRUNC so that the actual packet length is returned even if the buffer was too small

`sb-bsd-sockets:socket-send` *socket buffer length* **&key** [Generic Function]
address external-format oob eor dontroute dontwait nosignal confirm more external-format
Send `length` octets from `buffer` into `socket`, using sendto(2). If `buffer` is a string, it will converted to octets according to `external-format`. If `length` is `nil`, the length of the octet buffer is used. The format of `address` depends on the socket type (for example for `inet` domain sockets it would be a list of an `ip` address and a port). If no socket address is provided, send(2) will be called instead. Returns the number of octets written.

`sb-bsd-sockets:socket-listen` *socket backlog* [Generic Function]
Mark `socket` as willing to accept incoming connections. `backlog` defines the maximum length that the queue of pending connections may grow to before new connection attempts are refused. See also listen(2)

`sb-bsd-sockets:socket-open-p` *socket* [Generic Function]
Return true if `socket` is open; otherwise, return false.

`sb-bsd-sockets:socket-close` *socket* [Generic Function]
Close `socket`. May throw any kind of error that write(2) would have thrown. If `socket-make-stream` has been called, calls `close` on that stream instead

`sb-bsd-sockets:socket-make-stream` *socket* **&rest** *args* [Generic Function]
Find or create a `stream` that can be used for `io` on `socket` (which must be connected). `args` are passed onto `sb-sys:make-fd-stream`.

`sb-bsd-sockets:socket-error` *where* [Function]

`sb-bsd-sockets:non-blocking-mode` *socket* [Generic Function]
Is `socket` in non-blocking mode?

13.3 Socket Options

A subset of socket options are supported, using a fairly general framework which should make it simple to add more as required - see '`SYS:CONTRIB;SB-BSD-SOCKETS:SOCKOPT.LISP`' for details. The name mapping from C is fairly straightforward: `SO_RCVLOWAT` becomes `sockopt-receive-low-water` and `(setf sockopt-receive-low-water)`.

`sb-bsd-sockets:sockopt-reuse-address` *socket* [Function]
Return the value of the `so-reuseaddr` socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-keep-alive` *socket* [Function]
Return the value of the `so-keepalive` socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-oob-inline` *socket* [Function]
Return the value of the `so-oobinline` socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-bsd-compatible` *socket* [Function]
Return the value of the `so-bsdcompat` socket option for `socket`. This can also be updated with `setf`. Available only on Linux.

`sb-bsd-sockets:sockopt-pass-credentials` *socket* [Function]
Return the value of the `so-passcred` socket option for `socket`. This can also be updated with `setf`. Available only on Linux.

`sb-bsd-sockets:sockopt-debug` *socket* [Function]
Return the value of the `so-debug` socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-dont-route` *socket* [Function]
Return the value of the `so-dontroute` socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-broadcast` *socket* [Function]
Return the value of the `so-broadcast` socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-tcp-nodelay` *socket* [Function]
Return the value of the `tcp-nodelay` socket option for `socket`. This can also be updated with `setf`.

13.4 INET Domain Sockets

The TCP and UDP sockets that you know and love. Some representation issues:

- Internet addresses are represented by vectors of (unsigned-byte 8) - viz. #(127 0 0 1). Ports are just integers: 6010. No conversion between network- and host-order data is needed from the user of this package.
- Socket addresses are represented by the two values for address and port, so for example, (socket-connect s #(192 168 1 1) 80).

`sb-bsd-sockets:inet-socket` [Class]

Class precedence list: `inet-socket, socket, standard-object, t`

Class representing `tcp` and `udp` sockets.

Examples:

```
(make-instance 'inet-socket :type :stream :protocol :tcp)
(make-instance 'inet-socket :type :datagram :protocol :udp)
```

`sb-bsd-sockets:make-inet-address` *dotted-quads* [Function]

Return a vector of octets given a string `dotted-quads` in the format "127.0.0.1"

`sb-bsd-sockets:get-protocol-by-name` *name* [Function]

Returns the network protocol number associated with the string `name`, using getprotobyname(2) which typically looks in `nis` or /etc/protocols

13.5 Local (Unix) Domain Sockets

Local domain (`AF_LOCAL`) sockets are also known as Unix-domain sockets, but were renamed by POSIX presumably on the basis that they may be available on other systems too.

A local socket address is a string, which is used to create a node in the local filesystem. This means of course that they cannot be used across a network.

`sb-bsd-sockets:local-socket` [Class]

Class precedence list: `local-socket, socket, standard-object, t`

Class representing local domain (AF_LOCAL) sockets, also known as unix-domain sockets.

13.6 Name Service

Presently name service is implemented by calling out to the `getaddrinfo(3)` and `gethostinfo(3)`, or to `gethostbyname(3)` `gethostbyaddr(3)` on platforms where the preferred functions are not available. The exact details of the name resolving process (for example the choice of whether DNS or a hosts file is used for lookup) are platform dependent.

`sb-bsd-sockets:host-ent` [Class]

Class precedence list: `host-ent, standard-object, t`

Slots:

- `name` — initarg: `:name`; reader: `sb-bsd-sockets:host-ent-name`
  The name of the host
- `addresses` — initarg: `:addresses`; reader: `sb-bsd-sockets:host-ent-addresses`
  A list of addresses for this host.

This class represents the results of an address lookup.

`sb-bsd-sockets:get-host-by-name` *host-name* [Function]
Returns a `host-ent` instance for `host-name` or signals a `name-service-error`. `host-name` may also be an `ip` address in dotted quad notation or some other weird stuff - see gethostbyname(3) or getaddrinfo(3) for the details.

`sb-bsd-sockets:get-host-by-address` *address* [Function]
Returns a `host-ent` instance for `address`, which should be a vector of (integer 0 255), or signals a `name-service-error`. See gethostbyaddr(3) or gethostinfo(3) for details.

`sb-bsd-sockets:host-ent-address` *host-ent* [Generic Function]
Returns some valid address for `host-ent`.

14 Profiling

SBCL includes both a deterministic profiler, that can collect statistics on individual functions, and a more "modern" statistical profiler.

Inlined functions do not appear in the results reported by either.

14.1 Deterministic Profiler

The package `sb-profile` provides a classic, per-function-call profiler.

`sb-profile:profile` **&rest** *names* [Macro]

`profile` Name*

If no names are supplied, return the list of profiled functions.

If names are supplied, wrap profiling code around the named functions. As in `trace`, the names are not evaluated. A symbol names a function. A string names all the functions named by symbols in the named package. If a function is already profiled, then unprofile and reprofile (useful to notice function redefinition.) If a name is undefined, then we give a warning and ignore it. See also `unprofile`, `report` and `reset`.

`sb-profile:unprofile` **&rest** *names* [Macro]

Unwrap any profiling code around the named functions, or if no names are given, unprofile all profiled functions. A symbol names a function. A string names all the functions named by symbols in the named package. `names` defaults to the list of names of all currently profiled functions.

`sb-profile:report` [Function]

Report results from profiling. The results are approximately adjusted for profiling overhead. The compensation may be rather inaccurate when bignums are involved in runtime calculation, as in a very-long-running Lisp process.

`sb-profile:reset` [Function]

Reset the counters for all profiled functions.

14.2 Statistical Profiler

The `sb-sprof` module, loadable by

```
(require :sb-sprof)
```

provides an alternate profiler which works by taking samples of the program execution at regular intervals, instead of instrumenting functions like `sb-profile:profile` does. You might find `sb-sprof` more useful than the deterministic profiler when profiling functions in the `common-lisp`-package, SBCL internals, or code where the instrumenting overhead is excessive.

Additionally `sb-sprof` includes a limited deterministic profiler which can be used for reporting the amounts of calls to some functions during

14.2.1 Example Usage

```
(in-package :cl-user)

(require :sb-sprof)

(declaim (optimize speed))

(defun cpu-test-inner (a i)
  (logxor a
          (* i 5)
          (+ a i)))

(defun cpu-test (n)
  (let ((a 0))
    (dotimes (i (expt 2 n) a)
      (setf a (cpu-test-inner a i)))))

;;;; CPU profiling

;;; Take up to 1000 samples of running (CPU-TEST 26), and give a flat
;;; table report at the end. Profiling will end one the body has been
;;; evaluated once, whether or not 1000 samples have been taken.
(sb-sprof:with-profiling (:max-samples 1000
                          :report :flat
                          :loop nil)
  (cpu-test 26))

;;; Record call counts for functions defined on symbols in the CL-USER
;;; package.
(sb-sprof:profile-call-counts "CL-USER")

;;; Take 1000 samples of running (CPU-TEST 24), and give a flat
;;; table report at the end. The body will be re-evaluated in a loop
;;; until 1000 samples have been taken. A sample count will be printed
;;; after each iteration.
(sb-sprof:with-profiling (:max-samples 1000
                          :report :flat
                          :loop t
                          :show-progress t)
  (cpu-test 24))

;;;; Allocation profiling

(defun foo (&rest args)
  (mapcar (lambda (x) (float x 1d0)) args))
```

```
(defun bar (n)
  (declare (fixnum n))
  (apply #'foo (loop repeat n collect n)))

(sb-sprof:with-profiling (:max-samples 10000
                          :mode :alloc
                          :report :flat)
  (bar 1000))
```

14.2.2 Output

The flat report format will show a table of all functions that the profiler encountered on the call stack during sampling, ordered by the number of samples taken while executing that function.

```
           Self        Total        Cumul
  Nr  Count     %  Count     %  Count     %    Calls  Function
------------------------------------------------------------------------
   1     69  24.4     97  34.3     69  24.4 67108864  CPU-TEST-INNER
   2     64  22.6     64  22.6    133  47.0        -  SB-VM::GENERIC-+
   3     39  13.8    256  90.5    172  60.8        1  CPU-TEST
   4     31  11.0     31  11.0    203  71.7        -  SB-KERNEL:TWO-ARG-XOR
```

For each function, the table will show three absolute and relative sample counts. The Self column shows samples taken while directly executing that function. The Total column shows samples taken while executing that function or functions called from it (sampled to a platform-specific depth). The Cumul column shows the sum of all Self columns up to and including that line in the table.

Additionally the Calls column will record the amount of calls that were made to the function during the profiling run. This value will only be reported for functions that have been explicitly marked for call counting with `profile-call-counts`.

The profiler also hooks into the disassembler such that instructions which have been sampled are annotated with their relative frequency of sampling. This information is not stored across different sampling runs.

```
;      6CF:       702E            JO L4              ; 6/242 samples
;      6D1:       D1E3            SHL EBX, 1
;      6D3:       702A            JO L4
;      6D5: L2:   F6C303          TEST BL, 3         ; 2/242 samples
;      6D8:       756D            JNE L8
;      6DA:       8BC3            MOV EAX, EBX       ; 5/242 samples
;      6DC: L3:   83F900          CMP ECX, 0         ; 4/242 samples
```

14.2.3 Platform support

This module is known not to work consistently on the Alpha platform, for technical reasons related to the implementation of a machine language idiom for marking sections of code to be treated as atomic by the garbage collector; However, it should work on other platforms, and the deficiency on the Alpha will eventually be rectified.

Allocation profiling is only supported on SBCL builds that use the generational garbage collector. Tracking of call stacks at a depth of more than two levels is only supported on x86 and x86-64.

14.2.4 Macros

`sb-sprof:with-profiling` (**&key** *sample-interval alloc-interval max-samples reset mode loop max-depth show-progress report*) **&body** *body* [Macro]

Repeatedly evaluate `body` with statistical profiling turned on. In multi-threaded operation, only the thread in which `with-profiling` was evaluated will be profiled by default. If you want to profile multiple threads, invoke the profiler with `start-profiling`.

The following keyword args are recognized:

`:sample-interval` <*n*>
: Take a sample every <n> seconds. Default is `*sample-interval*`.

`:alloc-interval` <*n*>
: Take a sample every time <n> allocation regions (approximately 8kB) have been allocated since the last sample. Default is `*alloc-interval*`.

`:mode` <*mode*>
: If `:cpu`, run the profiler in `cpu` profiling mode. If `:alloc`, run the profiler in allocation profiling mode.

`:max-samples` <*max*>
: Repeat evaluating body until <max> samples are taken. Default is `*max-samples*`.

`:max-depth` <*max*>
: Maximum call stack depth that the profiler should consider. Only has an effect on x86 and x86-64.

`:report` <*type*>
: If specified, call `report` with `:type` <type> at the end.

`:reset` <*bool*>
: It true, call `reset` at the beginning.

e `:loop` <bool> If true (the default) repeatedly evaluate `body`. If false, evaluate if only once.

`sb-sprof:with-sampling` (**&optional** *on*) **&body** *body* [Macro]

Evaluate body with statistical sampling turned on or off.

14.2.5 Functions

`sb-sprof:report` **&key** *type max min-percent call-graph stream show-progress* [Function]

Report statistical profiling results. The following keyword args are recognized:

`:type` <*type*>
: Specifies the type of report to generate. If `:flat`, show flat report, if `:graph` show a call graph and a flat report. If nil, don't print out a report.

`:stream` <*stream*>
: Specify a stream to print the report on. Default is `*standard-output*`.

`:max` <*max*>
: Don't show more than <max> entries in the flat report.

`:min-percent` <*min-percent*>
: Don't show functions taking less than <min-percent> of the total time in the flat report.

`:show-progress` <*bool*>
: If true, print progress messages while generating the call graph.

`:call-graph` <*graph*>
: Print a report from <graph> instead of the latest profiling results.

Value of this function is a `call-graph` object representing the resulting call-graph.

`sb-sprof:reset` [Function]

Reset the profiler.

`sb-sprof:start-profiling` **&key** *max-samples mode sample-interval alloc-interval max-depth sampling* [Function]

Start profiling statistically if not already profiling. The following keyword args are recognized:

`:sample-interval` <*n*>
: Take a sample every <n> seconds. Default is `*sample-interval*`.

`:alloc-interval` <*n*>
: Take a sample every time <n> allocation regions (approximately 8kB) have been allocated since the last sample. Default is `*alloc-interval*`.

`:mode` <*mode*>
: If `:cpu`, run the profiler in `cpu` profiling mode. If `:alloc`, run the profiler in allocation profiling mode.

`:max-samples` <*max*>
: Maximum number of samples. Default is `*max-samples*`.

`:max-depth` <*max*>
: Maximum call stack depth that the profiler should consider. Only has an effect on x86 and x86-64.

`:sampling` <*bool*>
: If true, the default, start sampling right away. If false, `start-sampling` can be used to turn sampling on.

`sb-sprof:stop-profiling` [Function]
Stop profiling if profiling.

`sb-sprof:profile-call-counts` **&rest** *names* [Function]
Mark the functions named by `names` as being subject to call counting during statistical profiling. If a string is used as a name, it will be interpreted as a package name. In this case call counting will be done for all functions with names like `x` or (SETF X), where `x` is a symbol with the package as its home package.

`sb-sprof:unprofile-call-counts` [Function]
Clear all call counting information. Call counting will be done for no functions during statistical profiling.

14.2.6 Variables

`sb-sprof:*max-samples*` [Variable]
Default number of traces taken. This variable is somewhat misnamed: each trace may actually consist of an arbitrary number of samples, depending on the depth of the call stack.

`sb-sprof:*sample-interval*` [Variable]
Default number of seconds between samples.

14.2.7 Credits

`sb-sprof` is an SBCL port, with enhancements, of Gerd Moellmann's statistical profiler for CMUCL.

15 Contributed Modules

SBCL comes with a number of modules that are not part of the core system. These are loaded via `(require :modulename)` (see Section 6.4 [Customization Hooks for Users], page 52). This section contains documentation (or pointers to documentation) for some of the contributed modules.

15.1 sb-aclrepl

The `sb-aclrepl` module offers an Allegro CL-style Read-Eval-Print Loop for SBCL, with integrated inspector. Adding a debugger interface is planned.

15.1.1 Usage

To start `sb-aclrepl` as your read-eval-print loop, put the form

```
(require 'sb-aclrepl)
```

in your '`~/.sbclrc`' initialization file.

15.1.2 Example Initialization

Here's a longer example of a '`~/.sbclrc`' file that shows off some of the features of `sb-aclrepl`:

```
(ignore-errors (require 'sb-aclrepl))

(when (find-package 'sb-aclrepl)
  (push :aclrepl cl:*features*))
#+aclrepl
(progn
  (setq sb-aclrepl:*max-history* 100)
  (setf (sb-aclrepl:alias "asdc")
        #'(lambda (sys) (asdf:operate 'asdf:compile-op sys)))
  (sb-aclrepl:alias "l" (sys) (asdf:operate 'asdf:load-op sys))
  (sb-aclrepl:alias "t" (sys) (asdf:operate 'asdf:test-op sys))
  ;; The 1 below means that two characaters ("up") are required
  (sb-aclrepl:alias ("up" 1 "Use package") (package) (use-package package))
  ;; The 0 below means only the first letter ("r") is required,
  ;; such as ":r base64"
  (sb-aclrepl:alias ("require" 0 "Require module") (sys) (require sys))
  (setq cl:*features* (delete :aclrepl cl:*features*)))
```

Questions, comments, or bug reports should be sent to Kevin Rosenberg (kevin@rosenberg.net).

15.1.3 Credits

Allegro CL is a registered trademark of Franz Inc.

15.2 sb-grovel

The `sb-grovel` module helps in generation of foreign function interfaces. It aids in extracting constants' values from the C compiler and in generating SB-ALIEN structure and union types, see Section 7.2.1 [Defining Foreign Types], page 56.

The ASDF(`http://www.cliki.net/ASDF`) component type GROVEL-CONSTANTS-FILE has its PERFORM operation defined to write out a C source file, compile it, and run it. The output from this program is Lisp, which is then itself compiled and loaded.

sb-grovel is used in a few contributed modules, and it is currently compatible only to SBCL. However, if you want to use it, here are a few directions.

15.2.1 Using sb-grovel in your own ASDF system

1. Create a Lisp package for the foreign constants/functions to go into.
2. Make your system depend on the 'sb-grovel system.
3. Create a grovel-constants data file - for an example, see example-constants.lisp in the contrib/sb-grovel/ directory in the SBCL source distribution.
4. Add it as a component in your system. e.g.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (require :sb-grovel))

(defpackage :example-package.system
            (:use :cl :asdf :sb-grovel :sb-alien))

(in-package :example-package.system)

(defsystem example-system
    :depends-on (sb-grovel)
    :components
    ((:module "sbcl"
              :components
              ((:file "defpackage")
               (grovel-constants-file "example-constants"
                                      :package :example-package)))))
```

   Make sure to specify the package you chose in step 1
5. Build stuff.

15.2.2 Contents of a grovel-constants-file

The grovel-constants-file, typically named `constants.lisp`, comprises lisp expressions describing the foreign things that you want to grovel for. A `constants.lisp` file contains two sections:

- a list of headers to include in the C program, for example:

```
("sys/types.h" "sys/socket.h" "sys/stat.h" "unistd.h" "sys/un.h"
 "netinet/in.h" "netinet/in_systm.h" "netinet/ip.h" "net/if.h"
 "netdb.h" "errno.h" "netinet/tcp.h" "fcntl.h" "signal.h" )
```

- A list of sb-grovel clauses describing the things you want to grovel from the C compiler, for example:

```
((:integer af-local
           #+(or sunos solaris) "AF_UNIX"
           #-(or sunos solaris) "AF_LOCAL"
           "Local to host (pipes and file-domain).")
 (:structure stat ("struct stat"
                   (integer dev "dev_t" "st_dev")
                   (integer atime "time_t" "st_atime")))
 (:function getpid ("getpid" int )))
```

There are two types of things that sb-grovel can sensibly extract from the C compiler: constant integers and structure layouts. It is also possible to define foreign functions in the constants.lisp file, but these definitions don't use any information from the C program; they expand directly to `sb-alien:define-alien-routine` (see Section 7.7.2 [The define-alien-routine Macro], page 64) forms.

Here's how to use the grovel clauses:

- `:integer` - constant expressions in C. Used in this form:

```
(:integer lisp-variable-name "C expression" &optional doc export)
```

  `"C expression"` will be typically be the name of a constant. But other forms are possible.

- `:enum`

```
(:enum lisp-type-name ((lisp-enumerated-name c-enumerated-name) ...))
```

  An `sb-alien:enum` type with name `lisp-type-name` will be defined. The symbols are the `lisp-enumerated-name`s, and the values are grovelled from the `c-enumerated-name`s.

- `:structure` - alien structure definitions look like this:

```
(:structure lisp-struct-name ("struct c_structure"
                              (type-designator lisp-element-name
                               "c_element_type" "c_element_name"
                               :distrust-length nil)
                              ; ...
                              ))
```

  `type-designator` is a reference to a type whose size (and type constraints) will be groveled for. sb-grovel accepts a form of type designator that doesn't quite conform to either lisp nor sb-alien's type specifiers. Here's a list of type designators that sb-grovel currently accepts:

  - `integer` - a C integral type; sb-grovel will infer the exact type from size information extracted from the C program. All common C integer types can be grovelled for with this type designator, but it is not possible to grovel for bit fields yet.
  - `(unsigned n)` - an unsigned integer variable that is `n` bytes long. No size information from the C program will be used.
  - `(signed n)` - an signed integer variable that is `n` bytes long. No size information from the C program will be used.

- `c-string` - an array of `char` in the structure. sb-grovel will use the array's length from the C program, unless you pass it the `:distrust-length` keyword argument with non-`nil` value (this might be required for structures such as solaris's `struct dirent`).
- `c-string-pointer` - a pointer to a C string, corresponding to the `sb-alien:c-string` type (see Section 7.2.3 [Foreign Type Specifiers], page 56).
- `(array alien-type)` - An array of the previously-declared alien type. The array's size will be determined from the output of the C program and the alien type's size.
- `(array alien-type n)` - An array of the previously-declared alien type. The array's size will be assumed as being `n`.

Note that `c-string` and `c-string-pointer` do not have the same meaning. If you declare that an element is of type `c-string`, it will be treated as if the string is a part of the structure, whereas if you declare that the element is of type `c-string-pointer`, a *pointer to a string* will be the structure member.

- `:function` - alien function definitions are similar to `define-alien-routine` definitions, because they expand to such forms when the lisp program is loaded. See Section 7.7 [Foreign Function Calls], page 63.

```
(:function lisp-function-name ("alien_function_name" alien-return-type
                                                     (argument alien-type)
                                                     (argument2 alien-type)))
```

15.2.3 Programming with sb-grovel's structure types

Let us assume that you have a grovelled structure definition:

```
(:structure mystruct ("struct my_structure"
                      (integer myint "int" "st_int")
                      (c-string mystring "char[]" "st_str")))
```

What can you do with it? Here's a short interface document:

- Creating and destroying objects:
  - Function `(allocate-mystruct)` - allocates an object of type `mystruct`and returns a system area pointer to it.
  - Function `(free-mystruct var)` - frees the alien object pointed to by *var*.
  - Macro `(with-mystruct var ((member init) [...]) &body body)` - allocates an object of type `mystruct` that is valid in *body*. If *body* terminates or control unwinds out of *body*, the object pointed to by *var* will be deallocated.
- Accessing structure members:
  - `(mystruct-myint var)` and `(mystruct-mystring var)` return the value of the respective fields in `mystruct`.
  - `(setf (mystruct-myint var) new-val)` and `(setf (mystruct-mystring var) new-val)` sets the value of the respective structure member to the value of *new-val*. Notice that in `(setf (mystruct-mystring var) new-val)`'s case, new-val is a lisp string.

15.2.3.1 Traps and Pitfalls

Basically, you can treat functions and data structure definitions that sb-grovel spits out as if they were alien routines and types. This has a few implications that might not be immediately obvious (especially if you have programmed in a previous version of sb-grovel that didn't use alien types):

- You must take care of grovel-allocated structures yourself. They are alien types, so the garbage collector will not collect them when you drop the last reference.
- If you use the `with-mystruct` macro, be sure that no references to the variable thus allocated leaks out. It will be deallocated when the block exits.

15.3 sb-posix

Sb-posix is the supported interface for calling out to the operating system.[1]

The scope of this interface is "operating system calls on a typical Unixlike platform". This is section 2 of the Unix manual, plus section 3 calls that are (a) typically found in libc, but (b) not part of the C standard. For example, we intend to provide support for `opendir()` and `readdir()`, but not for `printf()`. That said, if your favourite system call is not included yet, you are encouraged to submit a patch to the SBCL mailing list.

Some facilities are omitted where they offer absolutely no additional use over some portable function, or would be actively dangerous to the consistency of Lisp. Not all functions are available on all platforms.

15.3.1 Lisp names for C names

All symbols are in the `SB-POSIX` package. This package contains a Lisp function for each supported Unix system call or function, a variable or constant for each supported Unix constant, an object type for each supported Unix structure type, and a slot name for each supported Unix structure member. A symbol name is derived from the C binding's name, by (a) uppercasing, then (b) removing leading underscores (`#\_`) then replacing remaining underscore characters with the hyphen (`#\-`). The requirement to uppercase is so that in a standard upcasing reader the user may write `sb-posix:creat` instead of `sb-posix:|creat|` as would otherise be required.

No other changes to "Lispify" symbol names are made, so `creat()` becomes `CREAT`, not `CREATE`.

The user is encouraged not to `(USE-PACKAGE :SB-POSIX)` but instead to use the `SB-POSIX:` prefix on all references, as some of the symbols symbols contained in the SB-POSIX package have the same name as CL symbols (`OPEN`, `CLOSE`, `SIGNAL` etc).

15.3.2 Types

Generally, marshalling between Lisp and C data types is done using SBCL's FFI. See Chapter 7 [Foreign Function Interface], page 55.

Some functions accept objects such as filenames or file descriptors. In the C binding to POSIX these are represented as strings and small integers respectively. For the Lisp programmer's convenience we introduce designators such that CL pathnames or open streams can be passed to these functions. For example, `rename` accepts both pathnames and strings as its arguments.

15.3.2.1 File-descriptors

A file-descriptor is a non-negative small integer.

A file-stream is a designator for a file-descriptor: the stream's file descriptor is extracted. Note that mixing I/O operations on a stream with operations directly on its descriptor may produce unexpected results if the stream is buffered.

`SB-EXT:MAKE-FD-STREAM` can be used to construct a stream associated with a file descriptor.

[1] The functionality contained in the package `SB-UNIX` is for SBCL internal use only; its contents are likely to change from version to version.

15.3.2.2 Filenames

A filename is a string.

A pathname is a designator for a filename: the filename is computed using the same mechanism that SBCL uses to map pathnames to OS filenames internally.

Note that filename syntax is distinct from namestring syntax, and that `SB-EXT:PARSE-NATIVE-NAMESTRING` may be used to construct Lisp pathnames that denote POSIX filenames returned by system calls. See [Function sb-ext:parse-native-namestring], page 69. Additionally, notice that POSIX filename syntax does not distinguish the names of files from the names of directories. Consequently, in order to parse the name of a directory in POSIX filename syntax into a pathname `defaults` for which

```
(merge-pathnames (make-pathname :name "FOO" :case :common)
                 defaults)
```

returns a pathname that denotes a file in the directory, supply a true `AS-DIRECTORY` argument to `SB-EXT:PARSE-NATIVE-NAMESTRING`. Likewise, if it is necessary to supply the name of a directory to a POSIX function in non-directory syntax, supply a true `AS-FILE` argument to `SB-EXT:NATIVE-NAMESTRING`.

15.3.2.3 Type conversion functions

For each of these types there is a function of the same name that converts any valid designator for the type into an object of said type.

```
(with-open-file (o "/tmp/foo" :direction :output)
  (sb-posix:file-descriptor o))
=> 4
```

15.3.3 Function Parameters

The calling convention is modelled after that of CMUCL's `UNIX` package: in particular, it's like the C interface except that:

a. Length arguments are omitted or optional where the sensible value is obvious. For example, `read` would be defined this way:

```
(read fd buffer &optional (length (length buffer))) => bytes-read
```

b. Where C simulates "out" parameters using pointers (for instance, in `pipe()` or `socketpair()`) these may be optional or omitted in the Lisp interface: if not provided, appropriate objects will be allocated and returned (using multiple return values if necessary).

c. Some functions accept objects such as filenames or file descriptors. Wherever these are specified as such in the C bindings, the Lisp interface accepts designators for them as specified in the 'Types' section above.

d. A few functions have been included in sb-posix that do not correspond exactly with their C counterparts. These are described in See Section 15.3.6 [Functions with idiosyncratic bindings], page 116.

15.3.4 Function Return Values

The return value is usually the same as for the C binding, except in error cases: where the C function is defined as returning some sentinel value and setting `errno` on error, we

instead signal an error of type `SYSCALL-ERROR`. The actual error value (`errno`) is stored in this condition and can be accessed with `SYSCALL-ERRNO`.

We do not automatically translate the returned value into "Lispy" objects – for example, `SB-POSIX:OPEN` returns a small integer, not a stream. Exception: boolean-returning functions (or, more commonly, macros) do not return a C integer, but instead a Lisp boolean.

15.3.5 Lisp objects and C structures

Sb-posix provides various Lisp object types to stand in for C structures in the POSIX library. Lisp bindings to C functions that accept, manipulate, or return C structures accept, manipulate, or return instances of these Lisp types instead of instances of alien types.

The names of the Lisp types are chosen according to the general rules described above. For example Lisp objects of type `STAT` stand in for C structures of type `struct stat`.

Accessors are provided for each standard field in the structure. These are named *`structure-name-field-name`* where the two components are chosen according to the general name conversion rules, with the exception that in cases where all fields in a given structure have a common prefix, that prefix is omitted. For example, `stat.st_dev` in C becomes `STAT-DEV` in Lisp.

Because sb-posix might not support all semi-standard or implementation-dependent members of all structure types on your system (patches welcome), here is an enumeration of all supported Lisp objects corresponding to supported POSIX structures, and the supported slots for those structures.

- passwd

  `sb-posix:passwd` [Class]

  Class precedence list: `passwd, standard-object, t`

  Slots:

  - `name` — initarg: `:name`; reader: `sb-posix:passwd-name`; writer: `(setf sb-posix:passwd-name)`

    User's login name.
  - `passwd` — initarg: `:passwd`; reader: `sb-posix:passwd-passwd`; writer: `(setf sb-posix:passwd-passwd)`

    The account's encrypted password.
  - `uid` — initarg: `:uid`; reader: `sb-posix:passwd-uid`; writer: `(setf sb-posix:passwd-uid)`

    Numerical user `id`.
  - `gid` — initarg: `:gid`; reader: `sb-posix:passwd-gid`; writer: `(setf sb-posix:passwd-gid)`

    Numerical group `id`.
  - `gecos` — initarg: `:gecos`; reader: `sb-posix:passwd-gecos`; writer: `(setf sb-posix:passwd-gecos)`

    User's name or comment field.
  - `dir` — initarg: `:dir`; reader: `sb-posix:passwd-dir`; writer: `(setf sb-posix:passwd-dir)`

    Initial working directory.

- `shell` — initarg: `:shell`; reader: `sb-posix:passwd-shell`; writer: `(setf sb-posix:passwd-shell)`

  Program to use as shell.

Instances of this class represent entries in the system's user database.

- stat

`sb-posix:stat` [Class]

Class precedence list: `stat, standard-object, t`

Slots:

- `mode` — initarg: `:mode`; reader: `sb-posix:stat-mode`

  Mode of file.
- `ino` — initarg: `:ino`; reader: `sb-posix:stat-ino`

  File serial number.
- `dev` — initarg: `:dev`; reader: `sb-posix:stat-dev`

  Device `id` of device containing file.
- `nlink` — initarg: `:nlink`; reader: `sb-posix:stat-nlink`

  Number of hard links to the file.
- `uid` — initarg: `:uid`; reader: `sb-posix:stat-uid`

  User `id` of file.
- `gid` — initarg: `:gid`; reader: `sb-posix:stat-gid`

  Group `id` of file.
- `size` — initarg: `:size`; reader: `sb-posix:stat-size`

  For regular files, the file size in bytes. For symbolic links, the length in bytes of the filename contained in the symbolic link.
- `atime` — initarg: `:atime`; reader: `sb-posix:stat-atime`

  Time of last access.
- `mtime` — initarg: `:mtime`; reader: `sb-posix:stat-mtime`

  Time of last data modification.
- `ctime` — initarg: `:ctime`; reader: `sb-posix:stat-ctime`

  Time of last status change

Instances of this class represent Posix file metadata.

- termios

`sb-posix:termios` [Class]

Class precedence list: `termios, standard-object, t`

Slots:

- `iflag` — initarg: `:iflag`; reader: `sb-posix:termios-iflag`; writer: `(setf sb-posix:termios-iflag)`

  Input modes.

- `oflag` — initarg: `:oflag`; reader: `sb-posix:termios-oflag`; writer: `(setf sb-posix:termios-oflag)`

  Output modes.
- `cflag` — initarg: `:cflag`; reader: `sb-posix:termios-cflag`; writer: `(setf sb-posix:termios-cflag)`

  Control modes.
- `lflag` — initarg: `:lflag`; reader: `sb-posix:termios-lflag`; writer: `(setf sb-posix:termios-lflag)`

  Local modes.

Instances of this class represent I/O characteristics of the terminal.

- timeval

`sb-posix:timeval` [Class]

Class precedence list: `timeval, standard-object, t`

Slots:

- `sec` — initarg: `:tv-sec`; reader: `sb-posix:timeval-sec`; writer: `(setf sb-posix:timeval-sec)`

  Seconds.
- `usec` — initarg: `:tv-usec`; reader: `sb-posix:timeval-usec`; writer: `(setf sb-posix:timeval-usec)`

  Microseconds.

Instances of this class represent time values.

15.3.6 Functions with idiosyncratic bindings

A few functions in sb-posix don't correspond directly to their C counterparts.

- getcwd

`sb-posix:getcwd` [Function]

Returns the process's current working directory as a string.

- readlink

`sb-posix:readlink` *pathspec* [Function]

Returns the resolved target of a symbolic link as a string.

- syslog

`sb-posix:syslog` *priority format* **&rest** *args* [Function]

Send a message to the syslog facility, with severity level `priority`. The message will be formatted as by `cl:format` (rather than C's printf) with format string `format` and arguments `args`.

15.4 sb-md5

The `sb-md5` module implements the RFC1321 MD5 Message Digest Algorithm. [FIXME cite]

`sb-md5:md5sum-file` *pathname* [Function]
: Calculate the MD5 message-digest of the file designated by pathname.

`sb-md5:md5sum-sequence` *sequence* **&key** *start end* [Function]
: Calculate the MD5 message-digest of data bounded by `start` and `end` in `sequence` , which must be a vector with element-type (UNSIGNED-BYTE 8).

`sb-md5:md5sum-stream` *stream* [Function]
: Calculate an MD5 message-digest of the contents of `stream`, whose element-type has to be (UNSIGNED-BYTE 8).

`sb-md5:md5sum-string` *string* **&key** *external-format start end* [Function]
: Calculate the MD5 message-digest of the binary representation of `string` (as octets) in `external-format`. The boundaries `start` and `end` refer to character positions in the string, not to octets in the resulting binary representation.

15.4.1 Credits

The implementation for CMUCL was largely done by Pierre Mai, with help from members of the `cmucl-help` mailing list. Since CMUCL and SBCL are similar in many respects, it was not too difficult to extend the low-level implementation optimizations for CMUCL to SBCL. Following this, SBCL's compiler was extended to implement efficient compilation of modular arithmetic (see Section 5.2 [Modular arithmetic], page 44), which enabled the implementation to be expressed in portable arithmetical terms, apart from the use of `rotate-byte` for bitwise rotation.

15.5 sb-rotate-byte

The `sb-rotate-byte` module offers an interface to bitwise rotation, with an efficient implementation for operations which can be performed directly using the platform's arithmetic routines. It implements the specification at http://www.cliki.net/ROTATE-BYTE.

Bitwise rotation is a component of various cryptographic or hashing algorithms: MD5, SHA-1, etc.; often these algorithms are specified on 32-bit rings. [FIXME cite cite cite].

`sb-rotate-byte:rotate-byte` *count bytespec integer* [Function]

Rotates a field of bits within `integer`; specifically, returns an integer that contains the bits of `integer` rotated `count` times leftwards within the byte specified by `bytespec`, and elsewhere contains the bits of `integer`.

15.6 sb-cover

The `sb-cover` module provides a code coverage tool for SBCL. The tool has support for expression coverage, and for some branch coverage. Coverage reports are only generated for code compiled using `compile-file` with the value of the `sb-cover:store-coverage-data` optimization quality set to 3.

As of SBCL 1.0.6 `sb-cover` is still experimental, and the interfaces documented here might change in later versions.

15.6.1 Example Usage

```
;;; Load SB-COVER
(require :sb-cover)

;;; Turn on generation of code coverage instrumentation in the compiler
(declaim (optimize sb-cover:store-coverage-data))

;;; Load some code, ensuring that it's recompiled with the new optimization
;;; policy.
(asdf:oos 'asdf:load-op :cl-ppcre-test :force t)

;;; Run the test suite.
(cl-ppcre-test:test)

;;; Produce a coverage report
(sb-cover:report "/tmp/report/")

;;; Turn off instrumentation
(declaim (optimize (sb-cover:store-coverage-data 0)))
```

15.6.2 Functions

`sb-cover:report` *directory* **&key** *form-mode external-format* [Function]

Print a code coverage report of all instrumented files into `directory`. If `directory` does not exist, it will be created. The main report will be printed to the file cover-index.html. The external format of the source files can be specified with the `external-format` parameter.

If the keyword argument `form-mode` has the value `:car`, the annotations in the coverage report will be placed on the CARs of any cons-forms, while if it has the value `:whole` the whole form will be annotated (the default). The former mode shows explicitly which forms were instrumented, while the latter mode is generally easier to read.

`sb-cover:reset-coverage` [Function]

Reset all coverage data back to the 'Not executed' state.

`sb-cover:clear-coverage` [Function]
Clear all files from the coverage database. The files will be re-entered into the database when the `fasl` files (produced by compiling `store-coverage-data` optimization policy set to 3) are loaded again into the image.

`sb-cover:save-coverage` [Function]
Returns an opaque representation of the current code coverage state. The only operation that may be done on the state is passing it to `restore-coverage`. The representation is guaranteed to be readably printable. A representation that has been printed and read back will work identically in `restore-coverage`.

`sb-cover:save-coverage-in-file` *pathname* [Function]
Call `save-coverage` and write the results of that operation into the file designated by `pathname`.

`sb-cover:restore-coverage` *coverage-state* [Function]
Restore the code coverage data back to an earlier state produced by `save-coverage`.

`sb-cover:restore-coverage-from-file` *pathname* [Function]
`read` the contents of the file designated by `pathname` and pass the result to `restore-coverage`.

Appendix A Concept Index

T

U

V

Appendix B Function Index

P

R

S

T

U

V

W

Appendix C Variable Index

Appendix D Type Index

Colophon

This manual is maintained in Texinfo, and automatically translated into other forms (e.g. HTML or pdf). If you're *reading* this manual in one of these non-Texinfo translated forms, that's fine, but if you want to *modify* this manual, you are strongly advised to seek out a Texinfo version and modify that instead of modifying a translated version. Even better might be to seek out *the* Texinfo version (maintained at the time of this writing as part of the SBCL project at http://sbcl.sourceforge.net/) and submit a patch.