

Debugging

This document contains the following sections:

[1.0 Introduction](#)

[2.0 Getting out of the debugger](#)

[3.0 Internal functions](#)

[4.0 Debugging background processes](#)

[5.0 Stack commands](#)

[5.1 :zoom](#)

[5.2 :brief, :moderate, and :verbose modes of :zoom](#)

[5.3 :all t and :all nil modes of :zoom](#)

[5.4 :function t and :function nil modes of :zoom](#)

[5.5 :specials t and :specials nil modes of :zoom](#)

[5.6 :relative t and :relative nil modes of :zoom](#)

[5.7 The :bt command for a quick look at the stack](#)

[5.8 Variables that affect the behavior of :zoom](#)

[5.9 Special handling of certain errors by :zoom](#)

[5.10 :zoom analogs and stack movement commands](#)

[5.11 Commands that hide frames](#)

[5.12 Frame information commands](#)

[5.13 Local variables and evaluation](#)

[5.14 Getting a backtrace programmatically](#)

[6.0 Local variables and the debugger](#)

[6.1 Discard local variable information before dumplisp](#)

[6.2 Summary of the discussion of locals](#)

[6.3 What are local variables?](#)

[6.4 How does the compiler treat local variables?](#)

[6.5 What is the difference between using registers and using the stack?](#)

[6.6 Live and dead ranges of local variables](#)

[6.7 Locals and functions in the tail position](#)

[6.8 Example showing live range](#)

[6.9 The debug=1 behavior with locals](#)

[6.10 The behavior with debug=2](#)

[6.11 Problem with debug=1 and debug=2 behavior before a local has a value](#)

[6.12 Why only have good behavior of locals at debug=3?](#)

[6.13 The behavior with debug=3 \(and speed < 3\)](#)

[6.14 The behavior with debug=3 and speed=3](#)

[6.15 I compiled with debug=3 but I want to see dead locals anyway](#)

[7.0 Break on exit](#)

[8.0 :return and :restart](#)

[9.0 Ghost frames in backtraces](#)

[9.1 Summary of the ghost frames section](#)

[9.2 What is a ghost frame?](#)

[9.3 What kinds of optimizations cause ghost frames?](#)

[9.4 How does the debugger know about ghost frames?](#)

[9.5 When will the debugger display ghost frames?](#)

[9.6 Can I return from or restart a ghost frame?](#)

[9.7 What do the suspension points \(^...'\) mean in a ghost frame?](#)

[9.8 The ghost frame has no `...'s; are all possible frames displayed?](#)

[9.9 No ghost frames are displayed. Do all functions appear on the stack?](#)

[9.10 Ghost frames in a brief backtrace](#)

[9.11 Can I turn off printing of ghost frames?](#)

[9.12 Can backtraces involving system functions have ghost frames?](#)

[9.13 Ghost frames and Allegro Composer](#)

[10.0 The tracer](#)

[10.1 Tracing function objects](#)

[10.2 Trace example](#)

[10.3 Tracing setf, :before, and :after methods and internal functions](#)

[11.0 The stepper](#)

[11.1 Turning stepping off](#)

[11.2 Other stepping commands and variables](#)

[11.3 Stepping example](#)

[12.0 The Lisp DeBug \(ldb\) stepper](#)

[12.1 Entering and Exiting the ldb stepper](#)

[12.2 Ldb stepper functional interface](#)

[12.3 Ldb stepping example run](#)

1.0 Introduction

Allegro CL provides a number of facilities for debugging and correcting Lisp code. Among these facilities are a set of commands to view and manipulate the runtime stack, a tracer, a stepper and an inspector. All are documented in this document except the inspector, which is documented in *inspector.htm*.

Debugger functionality is included by default in development images (those built with either the *include-debugger* or *include-devel-env* arguments to **build-lisp-image** specified true). If you are building a runtime image and want debugging capability, be sure to specify *include-debugger* true (you cannot specify *include-devel-env* true when building a runtime image). See **build-lisp-image** and *building-images.htm* for information on arguments to **build-lisp-image**. Note that the stepper is not permitted in runtime images.

A feature in Allegro CL is the ability to debug background processes in separate Lisp listeners. See **use-background-streams** and [Section 4.0 Debugging background processes](#) for more information. If you are unable to use the facility described there, note the following. When multiprocessing is enabled, the user must be careful to ensure that he or she is debugging the correct process. The user types to the *listener* process, and debugs the *focused* process. Initially, they are the same process. The focus is changed only by user action. But if the focus is changed, the processes will not be the same, and it is important that the user keep track of which process is being debugged. Where relevant, the description of a debugging command specifies how it is used with multiprocessing. Users who are not using the multiprocessing facility need not concern themselves with these distinctions.

Note too that debugging compiled code is affected by what optimizations were in force when the code was compiled. These optimizations are discussed in *compiling.htm*. Suffice it to say here that functions compiled for speed over safety may be harder to debug for the following reasons.

1. Functions may be compiled inline, so it may be hard to figure out where you are, comparing what you expect from looking at the source to what you see on the stack.
2. The real error may have gone unnoticed and the reported error may occur much later, perhaps in a different function. Thus you may notice unexpected arguments to a function (resulting in an error) but it may be difficult to discover where these arguments came from.
3. The error messages themselves may be quite uninformative. This is because errors are not caught by type or argument checking, but rather later, system problems.

One more point: if an error occurs in code compiled for speed over safety which is hard to debug, one strategy is to recompile the code for safety over speed. It should then be easier to correct.

The debugger in an application

You can include much of the debugger (except the stepper) in a runtime application (see *runtime.htm* for details). Further, the document *debugger-api.htm* provides information about the internals of the debugger and using it, it is possible to provide a customized debugging interface. Note that *debugger-api.htm* does not describe things needed in ordinary use of Allegro CL.

2.0 Getting out of the debugger

Debugger commands are available all the time, so you are never really in or out of the debugger. When an error occurs, you are put into a break loop (as indicated by the number in brackets in the prompt). See *top-level.htm* for complete details, but briefly, the top-level command **:pop** will pop up one break level and the top-level command **:reset** will return you to the top-level, out of all break levels, as the following example shows:

```
USER(1): (car 1)
Error: Attempt to take the car of 1 which is not listp.
[condition type: SIMPLE-ERROR]
```

```
[1] USER(2): :reset
USER(3):
```

3.0 Internal functions

Users trying to debug code will often have occasion to look at the stack for a list of recent function calls. Included in the list will be functions with names like: *operator_2op* and *operator_3op* where *operator* is a numeric function such as *, +, <, etc. These functions are used in place of the expected functions (*, +, <, etc.) for compiler efficiency. They should be interpreted as the functions named by operator. Thus, for example, *<_2op* should be interpreted < (i.e. the less-than predicate).

4.0 Debugging background processes

The background-streams facilities described in this section only work when Allegro CL has been started with the **fi:common-lisp** command to Emacs. The Emacs-Lisp interface is fully described in *eli.htm*. If you are not running under Emacs (or if the backdoor connection between Lisp and Emacs has not been established), you will not get the behavior described here. Note that it is not an error to use background streams when not running Lisp under Emacs. The described effect (creating a new listener in another Emacs buffer) will not happen. But if you later establish the connection, background streams will be used. The function **use-background-streams** is called automatically when Lisp starts up so it is not necessary to call it explicitly. However, you should call it if you do not get the expected behavior.

If an error occurs in a background process while Allegro Composer is not running, another listener buffer is created at once and output from the error is printed in the new listener buffer. To test the effect of background streams, evaluate the following forms.

```
(fmakunbound 'foo)
  (mp:process-run-function "bad process" 'foo)
```

These forms start a background process that will break and cause a new listener to be created. A message will be printed to the new listener saying that a background process has broken due to foo being undefined.

Background streams work by setting the global values of the following streams to the background stream:

```
*terminal-io*
*debug-io*
*standard-input*
*query-io*
```

Note that setting the value of those streams has the following consequences:

- The debugger checks to see if `*debug-io*` is a background stream and if so, it causes a listener to be created.
- Any attempt to do I/O on those streams results in an error.

The second effect is less drastic than it sounds. The existing Lisp listener (what you see when Lisp starts up) has already bound all the listed streams so it does not see the global value at all. There may be a problem if you start your own processes (with, e.g., **process-run-function**), however. You should (and all applications should) set up your own streams. Consider the following two examples. In the first, the function run by **process-run-function** takes a stream argument and is passed `*terminal-io*`. This will work without error. In the second, the function itself contains a reference to `*terminal-io*`. It will signal an error if background streams are used.

```
;;; This will work:
(mp:process-run-function
 "foo"
 #'(lambda (stream)
      (format stream "This is from process ~a~%"
              mp:*current-process*)))
*terminal-io*)
;;; This will fail:
(mp:process-run-function
 "foo"
 #'(lambda ()
      (format *terminal-io* "This is from process ~a~%"
              mp:*current-process*)))
```

The variable `*initial-terminal-io*` holds the original `*terminal-io*` stream when Lisp starts up. It may be useful for processes that aren't connected to a usable `*terminal-io*` but wish to produce some output, for example for debugging.

5.0 Stack commands

The *runtime stack* is the entity where arguments and variables local to Lisp functions are stored. When a function is called, the calling function evaluates and *pushes* the arguments to the called function onto the stack. The called function then references the stack when accessing its arguments. It also allocates space for its local variables. A *stack frame* is the area on the stack where the arguments to one function call and also its local variables reside. If **foo** calls **bar**, which in turns calls **yaf**, then (at least, and typically) three stack frames are *active* when **yaf** is entered. The frame for the most recently called function is on the *top* of the stack. The commands described in the following subsections access and display the stack. After a frame is examined, it normally becomes the *current stack frame*. Further reference to the stack will, by default, operate on the current stack frame. When a break level is entered, the current frame pointer is typically the first interesting frame.

A *frame object* is a type of Lisp object. A *frame expression* is the printed representation of that object. We are somewhat imprecise in this document in distinguishing between frame objects (the internal Lisp object) and frame expressions (what you see printed) because in most cases, the distinction is not important. Where the distinction is important, we say *frame object* or *frame expression*.

5.1 :zoom

The **:zoom** top-level command prints the evaluation stack. It uses the current stack frame as the center of attention, and prints some number of frames on either side of the current frame. The value of the variable `*zoom-display*` is the total number of frames to display, and an equal number of frames are printed above and below the current stack frame, if possible. The arguments to the **:zoom** command control the type and quantity of the displayed stack.

After a **:zoom** or any of its analogs (such as **:top** or **:bottom**) the special variable `cl:*` contains the lisp expression representing the current frame. That expression is approximately what is shown in a moderate display with `:function nil`, regardless of the mode in which **:zoom** itself displays.

Here are some examples of **:zoom** command calls. We cause an error by trying to evaluate `foo`, which has no value.

```
USER(1): foo
Error: Attempt to take the value of the unbound variable `FOO'.
  [condition type: UNBOUND-VARIABLE]
[1] USER(2):
```

5.2 :brief, :moderate, and :verbose modes of :zoom

These arguments to **:zoom** (only one can be specified true and that value controls further **:zoom** commands until a new value is specified) control the amount of information printed.

In **:brief** mode, only the function name is displayed for each frame and more than one frame is displayed on a single line. The current frame (EVAL) is displayed on its own line.

```
[1] USER(2): :zoom :brief t
Evaluation stack:

  ERROR <-
EVAL <-
  TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP <- TPL:START-INTERACTIVE-TOP-LEVEL <-
```

In :moderate mode, each frame is on its own line and the function name and arguments appear.

```
[1] USER(3): :zoom :moderate t
Evaluation stack:
```

```
(ERROR UNBOUND-VARIABLE :NAME ...)
->(EVAL FOO)
(TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
(TPL:START-INTERACTIVE-TOP-LEVEL
#<EXCL::BIDIRECTIONAL-TERMINAL-STREAM @ #x18ad06>
#<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP @ #x2df9de> ...)
```

In :verbose mode, several lines are used per frame and much more information about arguments is provided.

```
[1] USER(4): :zoom :verbose t
Evaluation stack:
```

```
call to ERROR
required arg: EXCL::DATUM = UNBOUND-VARIABLE
&rest EXCL::ARGUMENTS = (:NAME FOO :FORMAT-CONTROL ...)
function suspended at relative address 600
```

```
->call to EVAL
required arg: EXP = FOO
function suspended at relative address 212
```

```
call to TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP
function suspended at relative address 692
```

```
call to TPL:START-INTERACTIVE-TOP-LEVEL
required arg: *TERMINAL-IO* = #<EXCL::BIDIRECTIONAL-TERMINAL-STREAM @
#x18ad06>
required arg: FUNCTION = #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP @
#x2df9de>
required arg: TPL::ARGS = NIL
&key TPL::INITIAL-BINDINGS = :UNSUPPLIED
function suspended at relative address 468
```

```
[1] USER(5):
```

Note that in verbose mode, each frame specifies the location at which the function was suspended:

```
call to ERROR
required arg: EXCL::DATUM = UNBOUND-VARIABLE
&rest EXCL::ARGUMENTS = (:NAME FOO :FORMAT-CONTROL ...)
function suspended at relative address 600
```

The suspension location may be relative or absolute (it is relative in the example just shown). It represents the value of the program counter at the time the function was suspended (usually because it called another function). The suspension location may be the instruction that caused the suspension (a call instruction or an instruction that caused trapping) or it may be the next instruction to be executed when the function is reactivated.

Suspension locations are interpreted as follows:

- **Relative:** a relative address is the distance from the start of the function's codevector. It can be seen by disassembling the function with **disassemble** with the absolute keyword argument `nil` (see *implementation.htm* for information on **disassemble**). The leftmost column of the disassembly shows the relative address of each instruction in bytes. Relative addresses are needed for Lisp functions because their codevectors may move so the actual address of a particular function may change after a garbage collection.
- **Absolute:** an absolute address is an address that will not change within the current Lisp process execution. Absolute addresses will be given when a frame represents a runtime operation. Examples of runtime operations are calls to **apply** and **funcall**, which are written in low-level lisp-assembler code.

When a runtime operation is encountered in `:verbose` mode, a symbol-table is built (if an up-to-date one does not already exist) if possible. This build may take some time and trigger several garbage collections. After the symbol table is built, the address is associated with a name in the table and printed as its offset.

If the symbol table cannot be built, then no interpretation is given to the suspension location, only the absolute address is shown. Here is an example from the verbose backtrace following the attempt to evaluate an unbound variable:

```
-----
->call to SYS::..CONTEXT-SAVING-RUNTIME-OPERATION with 0 arguments.
function suspended at address #x6fa1b1d0 (unbound+404)
```

Note that any non-lisp function can be disassembled (with **disassemble**) if it is represented in the symbol table by the string associated with its name. Thus

```
(cl:disassemble "qcons")
```

will print the disassembly of the runtime operation "qcons". If the name is not in the symbol table, `cl:disassemble` will complain that the argument is invalid.

5.3 `:all t` and `:all nil` modes of `:zoom`

Using the same error as above (trying to evaluate `f oo`, which is unbound), here is the difference between specifying `:all t` and `:all nil` as arguments to **zoom**. The frames hidden by default are those specified by default by **hide**.

```
[1] USER(5): :zoom :all nil :moderate t
Evaluation stack:

  (ERROR UNBOUND-VARIABLE :NAME ...)
->(EVAL FOO)
  (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
  (TPL:START-INTERACTIVE-TOP-LEVEL
   #<EXCL::BIDIRECTIONAL-TERMINAL-STREAM @ #x18ad06>
   #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP @ #x2df9de> ...)
[1] USER(6): :zoom :all t
Evaluation stack:

... 1 more newer frame ...

  (ERROR UNBOUND-VARIABLE :NAME ...)
  (EXCL::UNBOUND-VARIABLE-HANDLER FOO)
  (EXCL::ER-GENERAL-ERROR-HANDLER-ONE 5 FOO)
  (EXCL::%EVAL FOO)
->(EVAL FOO)
  (TPL::READ-EVAL-PRINT-ONE-COMMAND NIL NIL)
  (EXCL::READ-EVAL-PRINT-LOOP :LEVEL 0)
  (TPL::TOP-LEVEL-READ-EVAL-PRINT-LOOP1)
  (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)

... more older frames ...
[1] USER(7):
```

5.4 :function t and :function nil modes of :zoom

The difference between specifying the `:function` argument to `:zoom` `t` or `nil` can be shown by one frame, so we have removed all but one displayed frame. With `:function t`, the exact function object is identified.

```
[1] USER(7): :zoom :function t
Evaluation stack: [note: some lines removed]

->(FUNCALL #<Function EVAL @ #x1216d6> FOO)

[1] USER(8): :zoom :function nil
Evaluation stack: [note: some lines removed]

->(EVAL FOO)
```

5.5 :specials t and :specials nil modes of :zoom

The difference between specifying the `:specials` argument to `:zoom` `t` or `nil` can be shown with one frame so some printed material is deleted. With `:specials t`, the names and values of specials associated with the frame are shown.

```
[1] USER(10): :zoom :specials t
Evaluation stack: [note: some lines removed]
```

```
->(EVAL FOO)
  EXCL::%VENV% = NIL
  EXCL::%FENV% = NIL
  EXCL::%BENV% = NIL
  EXCL::%GENV% = NIL
  EXCL::%FUNCTION-SPEC% = NIL
```

The stack with `:specials nil` only shows the stack frame:

```
[1] USER(11): :zoom :specials nil
Evaluation stack: [note: some lines removed]
```

```
->(EVAL FOO)
```

5.6 :relative t and :relative nil modes of :zoom

The `:relative` argument causes `:zoom` to identify a frame with respect to the current frame. The identification is done with a number and either ``u'` (meaning up or newer than the current frame) or ``d'` (meaning down or older than the current frame).

```
[1] USER(18): :zoom :relative t
Evaluation stack:

1u: (ERROR UNBOUND-VARIABLE :NAME ...)
  ->(EVAL FOO)
1d: (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
2d: (TPL:START-INTERACTIVE-TOP-LEVEL
     #<EXCL::BIDIRECTIONAL-TERMINAL-STREAM @
     #x18ad06>
  ...)
  ...)
```

The stack with `:relative nil` does not display numbers next to frames.

```
[1] USER(18): :zoom :relative nil
```

```
Evaluation stack:
```

```
(ERROR UNBOUND-VARIABLE :NAME ...)
->(EVAL FOO)
(TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
(TPL:START-INTERACTIVE-TOP-LEVEL
 #<EXCL::BIDIRECTIONAL-TERMINAL-STREAM @ #x18ad06> ...)
```

5.7 The `:bt` command for a quick look at the stack

The `:bt` command provides a quick way to scan the stack.

Here is the effect of the `:bt` command on the example above (attempt to evaluate `foo` which has no value). In this case, `:zoom` is printing all frames (`:all t`) so `:bt` does as well.

```
[1] USER(24): :bt
```

```
Evaluation stack:
```

```
EVAL <-
  TPL::READ-EVAL-PRINT-ONE-COMMAND <-
  EXCL::READ-EVAL-PRINT-LOOP <-
  TPL::TOP-LEVEL-READ-EVAL-PRINT-LOOP1 <-
  TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP <- APPLY <-
  TPL:START-INTERACTIVE-TOP-LEVEL <- TPL::START-TOP-LEVEL <-
  EXCL::START-REBORN-LISP
```

5.8 Variables that affect the behavior of `:zoom`

The following variables affect the behavior of `:zoom`, usually by providing a default for an argument to `:zoom`. Note that the values of some of these variables are changed when `:zoom` is called with the argument associated with the variable specified. All symbols naming the variables are in the `top-level` package (nicknamed `tpl`).

<code>*zoom-display*</code>	Controls the maximum number of stack frames displayed by a call to <code>:zoom</code> . Default for the <code>:</code> count argument for <code>:zoom</code> .
<code>*zoom-print-circle*</code>	<code>cl:*print-circle*</code> is bound to this during <code>:zoom</code> output.

<code>*zoom-print-level*</code>	<code>cl:*</code> print-level* is bound to this during :zoom output.
<code>*zoom-print-length*</code>	<code>cl:*</code> print-length* is bound to this during :zoom output.
<code>*zoom-print-special-bindings*</code>	Default for the <code>:specials</code> argument to :zoom . Value reset if :zoom is called with <code>:specials</code> specified.
<code>*zoom-show-newer-frames*</code>	If true, :zoom output shows frames newer than the current frame.
<code>*auto-zoom*</code>	Controls whether frames are printed after moving up or down a frame or displaying the current frame (see <code>:up</code> , <code>:dn</code> , and <code>:current</code>).

5.9 Special handling of certain errors by `:zoom`

Here is how calls to undefined functions and calls with the wrong number of arguments, both in compiled code, are handled.

If, in compiled code, a call is made to an undefined function **foo**, a frame on the stack will be created to represent that call and it will show all of the arguments passed to **foo**.

For example:

```
user(4): (defun bar (x) (foo 1 2 3 4 x))
bar
user(5): (compile 'bar)
compiling bar
user(6): (bar 222)
Error: the function foo is undefined.
[condition type: undefined-function]
```

```
Restart actions (select using :continue):
0: Try calling foo again
1: Return a value instead of calling foo
2: Try calling a function other than foo
3: Setf the symbol-function of foo and call it again
[1] user(7): :zoom
Evaluation stack:
```

```

(error #<undefined-function @ #x10b9a86>)
->(foo 1 2 ...)
(bar 222)
(eval (bar 222))
(tpl:top-level-read-eval-print-loop)
(tpl:start-interactive-top-level
 #<excl::bidirectional-terminal-stream @ #x2aa24e>
 #<Function top-level-read-eval-print-loop @ #x2d96de>
...)
[1] user(8): :cur
(foo 1 2 3 4 222)
[1] user(9):

```

This frame can be restarted after defining foo:

```

[1] user(10): (defun foo (&rest x) x)
foo
[1] user(11): :cur
(foo 1 2 3 4 222)
[1] user(12): :restart
(1 2 3 4 222)
user(13):

```

In the wrong number of arguments situation (in our case, too few arguments) we again get an ordinary frame with the unsupplied arguments identified as :unknown:

```

USER(14): (defun baz (a b c)
           (+ a b c))
BAZ
USER(15): (compile 'baz)
BAZ
NIL
NIL
USER(16): (baz 1 2)
Error: BAZ got 2 args, wanted 3 args.
[condition type: PROGRAM-ERROR]
[1] USER(17): :zo
Evaluation stack:

(ERROR PROGRAM-ERROR :FORMAT-CONTROL ...)
->(BAZ 1 2 :UNKNOWN)
(EVAL (BAZ 1 2))
(TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
(TPL:START-INTERACTIVE-TOP-LEVEL
 #<EXCL::BIDIRECTIONAL-TERMINAL-STREAM @
#x487586>

```

```
#<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP @
#x49114e> ...)
[1] USER(18):
```

5.10 :zoom analogs and stack movement commands

The following commands are used to move the current frame pointer around the stack.

:dn	Move down the stack
:up	Move up the stack.
:bottom	Move to the bottom (oldest frame) of the stack.
:top	Move to the top (newest frame) of the stack.
:find	Move to the frame associated with the <i>function-name</i> argument to this command.

5.11 Commands that hide frames

These commands control which frames are displayed. Note that their effect can be overridden by calling **:zoom** with the `:all` argument specified `t`.

:hide	Hide (i.e. :zoom should not display unless <code>:all</code> is specified <code>t</code>) things specified by the arguments.
:unhide	Unhide (i.e. :zoom with <code>:all nil</code> should display) things specified by the arguments. :unhide called with no arguments causes the list of hidden objects to revert to the default set of hidden objects. Note that if you want to see all frames in the stack, you should call :zoom with a true value for the <i>all</i> argument. :unhide itself is not designed to unhide all frames.

```
;; In this example we hide all frames in the tpl package.
```

```
USER(66): :unhide
```

```
Reverting hidden objects to initial state.
```

```
USER(67): :hide
```

```
hidden packages: LEP EXCL SYSTEM CLOS DEBUGGER
```

```

hidden packages internals: TOP-LEVEL
hidden functions: BLOCK APPLY
hidden frames: :INTERPRETER :EVAL :INTERNAL
;;
;; We cause a BREAK and do a :zoom.
USER(68): (break)
Break: call to the `break' function.

Restart actions (select using :continue):
  0: return from break.
[1c] USER(69): :zoom :moderate t
Evaluation stack:

  (BREAK "call to the `break' function.")
->(EVAL (BREAK))
  (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
  (TPL:START-INTERACTIVE-TOP-LEVEL
   #<EXCL::BIDIRECTIONAL-TERMINAL-STREAM @ #x19e67e>
   #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP @ #x1c7ad6>
  ...)
;;
;; We hide all symbols in the TOP-LEVEL package.
[1c] USER(70): :hide :package tpl
[1c] USER(71): :zoom
Evaluation stack:

  (BREAK "call to the `break' function.")
->(EVAL (BREAK))
[1c] USER(72):

```

5.12 Frame information commands

The following commands identify the current frame and the function called by the current frame.

:current	Print the current stack frame and set the value of <code>c1:*</code> to be the frame expression.
:function	Print the function object called in the current stack frame and set the value of <code>c1:*</code> to that object.
:frame	Print out information about the current frame.

```

;; In the example below, we use :current, :function and :frame.
;; Note that we use :hide :binding to limit the amount of output
;; printed in this example.
user(47): (defun foo (x)
           (let ((a (1+ x)))
             (let ((b
(1+ a)))

(progn (break "foo: ~a" b)

(foo 1 2 3 4 5 x a b))))))
foo

;; :hide :binding causes binding frames to be hidden.
user(48): :hide :binding
user(49): (foo 10)
Break: foo: 12

Restart actions (select using :continue):
  0: return from break.
[1c] user(50): :zoom
Evaluation stack:

  (break "foo: ~a" 12)
->(foo 10)
  (eval (foo 10))
  (tpl:top-level-read-eval-print-loop)
  (tpl:start-interactive-top-level
    #<excl::bidirectional-terminal-stream @ #x280886>
    #<Function top-level-read-eval-print-loop @ #x2ce19e>
  ...)
;;
;; Print out the current frame.
[1c] user(51): :current
(foo 10)

;; Print out the function object associated with the
;; current frame.
[1c] user(52): :function
#<Interpreted Function foo @ #xaf983e>

;; Print out interesting information about the frame. Note
;; that the information printed included frames hidden by the
;; :hide :binding command
[1c] user(53): :frame
Expression: (foo 10)
Source code: (let ((a (1+ x))) ..)

```

```

Local x: 10
Source code: (let ((b (1+ a))) ..)
Local a: 11
Source code: (progn (break "foo: ~a" b) (foo 1 2 3 4 5 x a b))
Local b: 12
[1c] user(54): :res
user(55):

```

5.13 Local variables and evaluation

The commands described in this section operate in some fashion on frames. Note that local name information (which is very useful for debugging) is only loaded from a compiled file if the variable `*load-local-names-info*` is non-`nil` when the compiled (fasl) file is loaded.

:local	This command causes the values of local variables for the current stack frame to be printed. The handling of local variables by the debugger is discussed under the heading Section 6.0 Local variables and the debugger below for complete information and examples.
:set-local	This command sets the value of the specified (as an argument) local variable to value, which is evaluated. The local variable is identified by name if interpreted, or name or index if compiled.
:evalmode	This command allows evaluation in context, that is local variables can be used by name in forms typed to the top level. Setting such a local variable (with setq or setf) at the top level will change the value in the frame. Evaluation in context does not always work as desired (unavoidably so because of the lexically-scoped design of Lisp). It should only be used temporarily while debugging.

5.14 Getting a backtrace programmatically

Sometimes, particularly when a program is being run in batch mode (rather than interactively) or when the user of the program is not familiar with Lisp internals, you may want to generate a backtrace (as produced by **:zoom**) programmatically, perhaps writing it to a file as part of a problem report.

For a variety of reasons, doing this is not trivial. However, the macro **with-auto-zoom-and-exit**, defined in the file `[Allegro directory]/src/autozoom.cl` (and not part of the standard Lisp image) provides the functionality for producing such a backtrace. That macro takes a *place* argument, which is typically a filename, to which the backtrace will be written, and optionally exits the running Lisp afterwards. If the **with-auto-zoom-and-exit** functionality is not exactly what you want, you can use the guts of the macro to define your own customized

functionality.

6.0 Local variables and the debugger

Allegro CL provides for examining and modifying the values of local variables in compiled code but this facility comes at a fairly significant space cost (since extra information is needed in compiled code to provide the debugger with this information). Therefore, the enhancement is only available in code compiled with the debug optimization quality set to 3 (assuming compiler switches have their default values). Further, local name information (which is very useful for debugging) is only loaded from a compiled file if the variable `*load-local-names-info*` is non-`nil` when the compiled (fasl) file is loaded.

6.1 Discard local variable information before `dumplisp`

Before we discuss local variables in detail, we should remark that the information necessary to provide information on local variables for debugging takes up a fairly significant amount of space in the image. If you have completed debugging and you want to dump an image (with `dumplisp`), you may want to discard this information before dumping (to save space in the dumped image). The function `discard-local-name-info` will do that.

6.2 Summary of the discussion of locals

- In code compiled with `debug=3` (so `save-local-scopes-switch` is true), information on the live ranges of local variables is stored. The debugger uses this information when the `:local` command is executed to print out just the locals which have valid values and are still in use. (The live range of a local is defined below. It is basically the time after a local is first assigned a value until the local is last used.) The variable `*load-local-names-info*` should be non-`nil` when compiled (fasl) files created with local variable information are loaded.
 - There is a significant space cost to storing information on the live range of locals. For this reason, it is only done (by default) when `debug=3`. We do not recommend compiling code with `debug=3` except for purposes of debugging.
 - In code compiled with `debug=1` or `debug=2`, live ranges of locals are not stored. The debugger prints the contents of all locations assigned to locals when the `:local` command is executed. The value in a location of a local not yet assigned a value may cause the debugger to print a great deal of garbage.
-

6.3 What are local variables?

The local variables in a function include the arguments of the function and variables bound in a let or similar form within the function. Arguments and other local variables are handled differently by the system, as we describe below.

6.4 How does the compiler treat local variables?

The names of local variables are not necessary in compiled code (the locations are sufficient). Saving names takes space. The compiler will only save names when the speed, safety, and debug compiler optimization qualities are such that the compiler switch `save-local-names-switch` is true. It is usual when developing code (so debugging is likely) to compile with that switch true, so names are available. In production code, it is usual to compile so names are not saved, to save space. When debugging code where names are not saved, locals are identified by a numerical index.

Further, local name information, if stored in a compiled file, is only loaded if the value of the variable `*load-local-names-info*` is non-`nil` when compiled (fasl) files created with local variable information are loaded.

Locals may be stored on the stack or in registers. Which method is used depends on the type of processor. When there is hardware support for using registers, the compiler stores local variables in registers. The processors that provide that support include the Sparc and the IBM RS/6000 processor. Where the necessary hardware support does not exist, locals are stored on the stack. (Some locals may be stored on the stack even in machines with the necessary hardware support. This happens when the number of live locals at some point exceeds the number of available registers -- eight on the Sparc and ten on the RS/6000.)

6.5 What is the difference between using registers and using the stack?

The advantage of register access is that it is much faster than stack access. But there is also a downside: there may not be a variable name associated with the register, because it is transient (because of efficient compilation) and never stored on the stack. This means that debugging will be harder.

We will get back to the advantages of placing values only in registers below, but we want to emphasize the debugging costs so that they are clear. Consider this example:

```
(defun blah (&rest format-args)
  (let ((result (apply #'format nil format-args)))
    (read-from-string result)))

(compile 'blah)
```

```
(blah "~a/0" 42)
```

Evaluating the last form will cause a divide by zero (`result` is the string "42/0", which is then read). But the variable named `result` never shows up on the stack because it is transiently used as the return value from an **apply** and as the first argument to **read-from-string**, and thus does not even move from its register location - in all Allegro CL implementations, the first return-value register is also the first argument register; this provides for extremely efficient code. Therefore, when debugging (after the divide by zero error is signaled), you want to look at the value of `result` but will not be able to see it under that name.

If you want to force storing values on the stack, you can trace **read-from-string** with a form like

```
(trace (read-from-string :inside blah))
```

Or you can use the **:break** top-level command to set an instruction-level breakpoint and setting `:ldb` mode on will allow single-stepping to track the variables (including the `arg0/result` register) if the user knows how to look at the disassembler output and interpret assembler code.

Or you can redefine `blah` to force `result` onto the stack so that it will show up as a local variable:

```
(defun blarg () nil)

(defun blah (&rest format-args)
  (let ((result (apply #'format nil format-args)))
    (blarg)
    (read-from-string result)))
```

Inserting the call to `blarg` forces `result` onto the stack (because the register holding its value cannot be protected during a function call). `result` will thus show up in debugger when the **read-from-string** errors. Note, however, that if local-scopes are compiled in, then the debugger will not show the variable named `result`, because it is a dead local (its last usage is as an argument to `read-from-string`). To see the name in that situation, `tpl::*print-dead-locals*` must be set to `t` (see [Section 6.15 I compiled with debug=3 but I want to see dead locals anyway](#)).

Now back to the advantages of using registers to hold values. Because of the very significant performance boost, registers are used whenever possible. Further, when registers are used, the compiler will analyze the code in order to determine the live ranges of locals and have locals with disjoint live ranges share the same register (*live range* is defined just [below](#)). This allows maximum use of registers.

Locals stored on the stack do not share locations -- that is, each local stored on the stack is assigned its own location. This is true on machines where all locals are stored on the stack and on machines where some locals are stored in registers and others are stored on the stack. It may be that sharing locations on the stack would be a desirable optimization (because of reduced stack growth). However, it has not been implemented.

The examples in the rest of this section are taken from a Sparc machine and therefore registers are used for

locals. All the behavior described applies to any machine (whether or not registers are used to store locals) except maybe the concept of register sharing.

6.6 Live and dead ranges of local variables

When compiling a function, the compiler tries to determine when a local variable is first used (typically, when it is bound) and when a local variable is last used. The time that the local is used is called the *live range* of the local.

When locals can be stored in registers, if two locals have disjoint live ranges, the compiler may use the same register to store the values of both variables.

6.7 Locals and functions in the tail position

Consider this definition:

```
(defun foo (lis)
  (let ((a 10) (b 9))
    (pprint (list a b lis))
    (list-length lis)))
```

The function **list-length** is in the tail position, which means that what it returns will be returned by **foo**, and so no information about **foo** is needed on the stack. The compiler will always arrange that the stack is cleared of all but the tail position function when possible (regardless of whether it tail merges or not). Therefore, the form `(foo 10)` will cause an error when **list-length** is applied to 10 -- which is not a list -- but the values of locals *a* and *b* will not be available.

6.8 Example showing live range

The live and dead ranges are best shown by example. Consider the following function definition:

```
(defun loc-fun (arg)
  (let ((a (+ arg 1)) ;; a alive but see text
        (b (+ arg 2)) ;; b alive but see text
        (c 3)) ;; c alive
    (break "break1")
    (setq c (+ a b c arg))))
```

```
;; a and b are now dead
  (let ((d 4) ;; d alive
        (e 5)) ;; e alive
    (break "break2")
    (print (* d e c))))

;; c, d, and e are now dead
(let* ((x 10) ;; x alive
      (y 11) ;; y alive
      (z 12)) ;; z alive
  (break "break3")
  (print (+ x y z)))
```

The comments show where the various locals become alive and when they die. There is an issue about where *a* and *b* become alive. Strictly speaking, *a*, *b*, and *c* all become alive when the binding form completes. That is guaranteed in **let** so if *a* and *b* had lexically apparent values from above the **let** form, they could be used later in the binding form. In fact that does not happen, so the compiler may make the assignment at the position indicated rather than later. This point is illustrated under the heading [Section 6.14 The behavior with debug=3 and speed=3](#) below.

The **breaks** are inserted to allow us to examine what is happening at various points.

Compiling this function with debug 3 ensures that local names are saved. Below we inspect the function object #'loc-fun. This inspection was done on a Sparc, a machine that uses registers to store locals and hence local locations are shared. Things will likely look different on other types of machines.

As the inspection shows, information about locals is stored in slot 9.

```
user(46): (inspect #'loc-fun)
#<Function loc-fun @ #x6ccd12>
  lambda-list: (arg)
  0 excl-type ----> Bit field: #x08
  1 flags -----> Bit field: #x88
  2 start -----> Bit field: #x006ccd54
  3 hash -----> Bit field: #x00004e9d
  4 symdef -----> The symbol loc-fun
  5 code -----> simple code vector (288) = #(40419 49048 ...)
  6 formals -----> (arg), a proper list with 1 element
  7 cframe-size --> fixnum 0 [#x00000000]
  8 call-count ---> fixnum 0 [#x00000000]
  9 locals -----> simple t vector (4) = #(8 ...) ...
  ...
  14 <constant> ---> A simple-string (6) "break3"
```

If we further inspect slot 9, we see how the compiler assigned locals to registers:

```
[1i] user(47): :i 9
A simple t vector (4) @ #x6cd032
  0-> simple t vector (15) = #(8 ...)
  1-> (y c), a proper list with 2 elements
  2-> (x d b), a proper list with 3 elements
  3-> (e a), a proper list with 2 elements
[1i] user(48):
```

Three registers are used. *y* and *c* share the first. *x*, *d*, and *b* share the second. *e* and *a* share the third. If you examine the code above, you will see that the live ranges of the locals sharing a register are disjoint (that is what allows them to share the register). The important result of this is that at any particular time, only the values of live locals are meaningful. On machines that do not use registers to store locals, no sharing would take place. Each local would have its own location on the stack.

6.9 The debug=1 behavior with locals

The **:local** debugging command with the debug optimization quality less than 3 simply prints out the information it has without any analysis. This usually means simply printing the contents of the locations where the locals are stored, regardless of whether the value in the location was the actual value of a local.

```
user(53): (loc-fun 22)
Break: break1

Restart actions (select using :continue):
  0: return from break.
[1c] user(54): :loc
Compiled lexical environment:
0(required): arg: 22
1(local): :unknown: 3
2(local): :unknown: 23
3(local): :unknown: 24
4(local): :unknown: 3
5(local): :unknown: #<non-lisp object @ #x11>
6(local): :unknown: 0
7(local): :unknown: 0
8(local): :unknown: 0
[1c] user(55):
```

What does this information tell us? The system has not stored any information about locals (including their names), so it simply prints the contents of the eight registers (because this is a Sparc, locals are saved in registers and all eight possible registers are displayed) that might contain locals. Knowing the function and how it was called, we can see that *a* is in 2 (the argument 22, plus 1) and *b* is in 3 (arg plus 2). *c* might be in 1 or 4 (later, we can determine it is in 1) but it is often hard to figure out what is going on. Further, some of the registers contain

garbage and the printed representation may not possibly represent a Lisp object (5, e.g.) or may be a Lisp object but simply left over from some earlier call (the rest).

In any case, it is not very useful for debugging but remember, that is how the function was compiled. Debugging information takes up space and reduces the efficiency of the code.

6.10 The behavior with debug=2

When debug=2, local names are saved. If we recompile **loc-fun** with debug=2, more information is provided:

```
USER(70): (loc-fun 22)
Break: break1
```

```
Restart actions (select using :continue):
```

```
  0: return from break.
```

```
[1c] USER(71): :loc
```

```
Compiled lexical environment:
```

```
0(REQUIRED): ARG: 22
```

```
1(LOCAL): (X C): 3
```

```
2(LOCAL): (Y D A): 23
```

```
3(LOCAL): (E B): 24
```

```
[1c] USER(72):
```

All the local names are printed but the system has no knowledge of which locals are alive. So it shows which locals share which registers and gives the current value in each register used by locals. This information is often helpful. There is a problem with this method. It is illustrated under the next heading.

This example is from a Sparc machine, so locals are stored in registers. On a machine where locals are stored on the stack, there would be no sharing. More information would be available, therefore, but bogus information would still be displayed for locals that had not yet been set.

6.11 Problem with debug=1 and debug=2 behavior before a local has a value

In the examples above, there was less information than desirable but otherwise things looked okay. This is particularly true in the debug=2 case just above. At the point of the break, all the locations used for locals had valid values. If, however, you examine the locals before all the locations have valid values, you may get a lot of garbage. Let us rewrite **loc-fun** slightly by putting a break before the let form -- here are the revised first 3 lines:

```
(defun loc-fun (arg)
```


6.12 Why only have good behavior of locals at debug=3?

As we will show, examining locals in code compiled at debug=3 (so the compiler switch `save-local-scopes-switch` is true) is cleaner and easier than in code compiled so the switch is false. The improvement is achieved by only printing the values of locals which are alive at the time of examination (equivalently, by suppressing the printing of dead locals). Why not have the clean behavior at all settings?

The reason is that there is a downside in suppressing dead locals: a fairly significant increase in the size of compiled code. This increase comes about because a lot of information about when locals become alive and become dead is saved.

The space hit is why the switch is only true (in the default) when debug is 3.

We do not recommend that you compile large amounts of code with debug 3. Instead, compile the code that you are currently working on.

6.13 The behavior with debug=3 (and speed < 3)

We recompile `local-fun` with debug 3 and call it again with the same argument, 10.

```
USER(10): (loc-fun 10)
Break: break1
```

```
Restart actions (select using :continue):
```

```
  0: return from break.
```

```
[1c] USER(11): :local
```

```
Compiled lexical environment:
```

```
0(REQUIRED): ARG: 10
```

```
1(LOCAL): C: 3
```

```
2(LOCAL): B: 12
```

```
3(LOCAL): A: 11
```

```
[1c] USER(12):
```

The live locals are identified and displayed. The yet-to-be-born locals are not shown. If we continue to the next **break**, we get the following:

```
[1c] USER(12): :cont
```

```
Break: break2
```

```
Restart actions (select using :continue):
```

```
  0: return from break.
```

```
[1c] USER(13): :local
```

```
Compiled lexical environment:
0(REQUIRED): ARG: 10
1(LOCAL): C: 36
2(LOCAL): D: 4
3(LOCAL): E: 5
[1c] USER(14):
```

a and *b* have died and *d* and *e* have appeared. Continuing to the next **break**:

```
[1c] USER(14): :cont

720
Break: break3
```

Restart actions (select using `:continue`):

```
  0: return from break.
[1c] USER(15): :local
Compiled lexical environment:
0(REQUIRED): ARG: 10
1(LOCAL): Y: 11
2(LOCAL): X: 10
[1c] USER(16):
```

Note that the argument `arg` is displayed in all cases. Arguments live throughout the function call, whether or not they are still needed. (Arguments are also stored in a different location than locals.)

6.14 The behavior with `debug=3` and `speed=3`

Recall [above](#) when we defined `loc-fun`, we pointed out that the live ranges of variables bound in a `let` form may start sooner than expected:

```
(defun loc-fun (arg)
  (let ((a (+ arg 1)) ;; a alive but see text
        (b (+ arg 2)) ;; b alive but see text
        (c 3)) ;; c alive
    [...])
```

As we said above, *a* and *b* should not be live (as locals) until the end of the binding form. The compiler can, however, see that they are not used and so may actually make the assignments as it processes each form in the binding form. The compiler will do this when the compiler switch `internal-optimize-switch` is true. This happens (by default) when `debug` is less than 3 or when `debug` is 3 and `speed` is 3. To illustrate this, let us define a new function that places breaks in the binding forms:

```
(defun loc-fun-2 (arg)
  (let ((a (+ 2 (bar arg)))
        (b (+ 3 (baz arg))))
    (break "break3")
    (print (+ a b))))
(defun bar (x) (break "break1") x)
(defun baz (x) (break "break2") x)
```

bar and **baz** just break and return their argument, allowing us to break while the binding form is being evaluated. The question is, when do *a* and *b* become alive? If **loc-fun-2** is compiled with `speed=3 debug=3`, *a* becomes alive before the *b* binding form completes (that is, before `break2` in **baz** is reached). If **loc-fun-2** is compiled with `speed=1 debug=3` (or `speed=2 debug=3`), *a* becomes alive when the whole binding form completes. This is illustrated by the transcripts below. Since the transcripts are the same except for the locals printed, we simply have one transcript with two columns when locals are shown. On the left, **loc-fun-2** is compiled with `speed=3 debug=3`; on the right, with `speed=1 debug=3`:

```
USER(28): (loc-fun-2 24)
Break: break1
```

```
Restart actions (select using :continue):
```

```
0: return from break.
```

```
[1c] USER(29): :dn
```

```
Evaluation stack:
```

```
(BREAK "break1")
```

```
(BAR 24)
```

```
->(LOC-FUN-2 24)
```

```
(EVAL (LOC-FUN-2 24))
```

```
[1c] USER(30): :local
```

-----speed=3 debug=3-----	-----speed=1 debug=3
Compiled lexical environment:	Compiled lexical environment:
0(REQUIRED): ARG: 24	0(REQUIRED): ARG: 24

```
[1c] USER(31): :cont
```

```
Break: break2
```

```
Restart actions (select using :continue):
```

```
0: return from break.
```

```
[1c] USER(32): :dn
```

```
Evaluation stack:
```

```
(BREAK "break2")
```

```
(BAZ 24)
```

```
->(LOC-FUN-2 24)
```

```
(EVAL (LOC-FUN-2 24))
```

```
[1c] USER(33): :local
-----speed=3 debug=3-----speed=1 debug=3
Compiled lexical environment:      Compiled lexical environment:
0(REQUIRED): ARG: 24              0(REQUIRED): ARG: 24
1(LOCAL): A: 26
```

```
[1c] USER(34): :cont
Break: break3
```

Restart actions (select using :continue):

0: return from break.

```
[1c] USER(35): :local
-----speed=3 debug=3-----speed=1 debug=3
Compiled lexical environment:      Compiled lexical environment:
0(REQUIRED): ARG: 26              0(REQUIRED): ARG: 24
1(LOCAL): A: 26                   1(LOCAL): B: 27
2(LOCAL): B: 27                   2(LOCAL): A: 26
```

```
[1c] USER(35):
```

The difference appears at break2 (while in **baz**). a is alive on the left and not on the right.

6.15 I compiled with debug=3 but I want to see dead locals anyway

This is possible but the interface is not optimal. There is an internal (i.e. not exported) variable which we are about to define that causes the debugger to print dead locals, either those that are not yet alive and those that were alive but are now dead.

[Variable]

```
tpl::*print-dead-locals*
```

When `nil` (the initial value), the debugger will suppress the printing of dead local variables. This suppression will only happen in functions compiled with the switch `save-local-scopes-switch true`, in the default when and only when `debug = 3`.

When true, dead locals are printed, regardless of how the function was compiled.

As we said at the beginning, one problem with printing locals that have never been alive is that wholly bogus and sometimes lengthy values may be printed because such a value happens to be in the location that will later hold the value of the live local. We are not going to repeat that example (see the discussion above under the heading [Section 6.11 Problem with debug=1 and debug=2 behavior before a local has a value](#)).

More interesting than bogus values is what happens when all the locals associated with a register (on machines where registers are used to store locals) that is used for locals are dead. In our **loc-fun** example, this happens at `break3`. *a*, *b*, *c*, *d*, and *e* are all dead and *x* and *y* are alive. *x* uses register 1 and *y* uses register 2 but both values that use register 3 (*a* and *e*) are dead. We compile **loc-fun** at `debug=3` and move to `break3`:

```
[1c] USER(93): :cont
1560
Break: break3

Restart actions (select using :continue):
  0: return from break.
[1c] USER(94): :local
Compiled lexical environment:
0(REQUIRED): ARG: 24
1(LOCAL): Y: 11
2(LOCAL): X: 10
[1c] USER(95): (setq tpl::*print-dead-locals* t)
T
[1c] USER(96): :local
Compiled lexical environment:
0(REQUIRED): ARG: 24
1(LOCAL): Y: 11
2(LOCAL): X: 10
3(LOCAL): (:DEAD (E A)): 5
[1c] USER(97):
```

For registers that hold live locals, the live local name and value are printed. For registers that hold only dead locals, the list of possible locals is printed, along with the keyword `:dead`, and the value in the register. The system has no way of knowing which of the identified dead locals had the indicated value (we know it is *e* by examining the code). The value is always the value of the last local to have died (assuming they were ever alive).

On machines where locals are stored on the stack and locations are not shared, each local has a location and the value associated with a local that was alive but is now dead is the last value of that local.

Note that you can set the value of `tpl::*print-dead-locals*` at any time and the behavior will change accordingly.

7.0 Break on exit

The `:boe` command allows you to break on exit, that is to stop execution when a frame is returned from.

Here is an example of using the **:boe** command.

```
user(2): (defun foo (a b) (let ((c (bar a b))) c))
foo
user(3): (defun bar (a b) (break) (list a b))
bar
user(4): (compile 'foo)
foo
nil
nil
user(5): (compile 'bar)
bar
nil
nil
user(6): (foo 1 2)
Break: call to the `break' function.
```

```
Restart actions (select using :continue):
  0: return from break.
[1c] user(7): :zo
```

Evaluation stack:

```
(break "call to the `break' function.")
->(bar 1 2)
  (foo 1 2)
  (eval (foo 1 2))
  (tpl:top-level-read-eval-print-loop)
  (tpl:start-interactive-top-level
    #<excl::bidirectional-terminal-stream @ #x2a1f5e>
    #<Function top-level-read-eval-print-loop @ #x2d437e>
    ...)
[1c] user(8): :cont
(1 2)
user(9): (foo 1 2)
Break: call to the `break' function.
```

```
Restart actions (select using :continue):
  0: return from break.
[1c] user(10): :dn
```

Evaluation stack:

```
(break "call to the `break' function.")
(bar 1 2)
->(foo 1 2)
```

```

(eval (foo 1 2))
(tpl:top-level-read-eval-print-loop)
(tpl:start-interactive-top-level
  #<excl::bidirectional-terminal-stream @
#x2a1f5e>
  #<Function top-level-read-eval-print-loop @
#x2d437e> ...)
[1c] user(11): :boe
[1c] user(12): :zo
Evaluation stack:

  (break "call to the `break' function.")
  (bar 1 2)
*->(foo 1 2)
  (eval (foo 1 2))
  (tpl:top-level-read-eval-print-loop)
  (tpl:start-interactive-top-level
    #<excl::bidirectional-terminal-stream
@ #x2a1f5e>
    #<Function
top-level-read-eval-print-loop @ #x2d437e> ...)
[1c] user(13): :cont
Break: just after returning from function

Restart actions (select using :continue):
  0: return from break.
[1c] user(14): :zo
Evaluation stack:

  (break "just after returning from function")
->(foo (1 2) 2)
  (eval (foo 1 2))
  (tpl:top-level-read-eval-print-loop)
  (tpl:start-interactive-top-level
    #<excl::bidirectional-terminal-stream @
#x2a1f5e>
    #<Function top-level-read-eval-print-loop @
#x2d437e> ...)
[1c] user(15): :cont
(1 2)

```

8.0 :return and :restart

The top-level commands **:return** and **:restart** both attempt to restart evaluation of a broken process from the current frame (see **:current**). **:return** sequentially evaluates the arguments provided and returns their values from the current stack frame. The function associated with the current stack frame is not executed, and its arguments are not again evaluated.

The **:return** command is useful in situations similar to the following. Suppose that in your code you had taken the **log** of a value when you meant to take the **exp** of the value. If you make the call to **log** be the current stack frame, then typing

```
:return (exp value)
```

will cause the computation to continue as if the code were correct to begin with, that is, **exp** of value will be calculated and returned from the current frame. The **log** function will not be re-executed.

:restart works on frame objects. A frame object is a list whose first element is a function and whose remaining elements are the *evaluated* arguments to the function. The top-level command **:zoom** prints a backtrace of the current execution stack. If **:zoom** is printing in moderate mode, it prints frame objects. (In brief mode, it simply prints the function name. In verbose mode, more than the frame object is printed.) **:zoom** and related commands (**:find**, **:dn**, **:up**, **:top**, etc.) all display the stack and identify one stack frame as the current stack frame. The value of the lisp variable ***** is set to the identified frame as part of the action of **:zoom** or a related command. Thus, if **:zoom** is printing in moderate mode, ***** will be set to a frame object after **:zoom** or a related command completes.

The argument to the **:restart** command must evaluate to a frame object (or `nil`, indicating that the current frame object should be used unchanged). Calling **:restart** restarts computation, replacing the current frame with the argument frame and continuing from there. Contrast this with **:return** which returns from a frame with a specified value (but does not re-evaluate the current frame). The two commands, **:return** and **:restart** can behave quite differently. We will give an example below.

As an example of **:restart** called with no arguments, suppose you define the following functions:

```
(defun baz (n) (bar (+ n 3)))
  (defun bar (n) (+ (goo n) 5))
  (defun foo (x) (* x x))
```

We have misspelled *foo* as *goo* in the definition of *bar*. If we evaluate

```
(baz 7)
```

we get an error since **goo** is undefined. The stack as printed by **:zoom** contains the frame

```
(bar 10)
```

We can then correct the definition of **bar** and load in just the new definition using the Emacs-Lisp interface. Next, if we make

```
(bar 10)
```

the current frame (using **:dn** or **:find**), we can call **:restart** without arguments (or with `nil` as an argument) and the computation will continue, returning the correct answer (105).

As an example of calling **:restart** with an argument, consider the following. Suppose **bar** and **foo** are defined as:

```
(defun bar () (+ (hoo 1 2 3) (hoo 4 5 6)))
  (defun foo (a b c) (+ a b c))
```

Here, the call to **hoo** in **bar** is a misprint: it should be a call to **foo**. If we evaluate `(bar)`, we again get an undefined function error. If we call **:zoom** (with its `:all` argument specified as `t` so all frames are printed), we see the following frame:

```
(excl::%eval (+ (hoo 1 2 3) (hoo 4 5 6)))
```

(You may have to set the values of `*zoom-print-level*` and `*zoom-print-length*` to `nil` to get the entire frame without `#` marks or suspension points.) If we make that frame the current frame (so it is the value of `cl:*`) and then evaluate

```
:restart (subst 'foo 'hoo *)
```

the frame will be re-evaluated so that **foo** is called rather than **hoo** and the correct result (21) will be returned.

The argument to **:restart** must evaluate to a frame object and the ``arguments'` in a frame object (the elements of the `cdr`) will not be further evaluated. Thus the following are not equivalent:

```
(+ 1 2 3 5)
  (+ 1 2 3 (- 6 1))
```

The first will evaluate correctly. The second will bomb since a list is not a legal argument to `+`.

One might think at first that **:restart** called with a frame object and **:return** called with a form which, when the arguments are evaluated, is the same since the frame object will have the same result. This is not, however, correct.

The bindings set up on the stack are (in some cases) handled differently by the two **:restart** and **:return**. Like many binding issues in Lisp, the example is not immediately intuitive (at least to Lisp beginners). Let us give an example and then discuss it. Consider the following functions:

```
(defun bar ()
  (let ((*special* 1))
    (declare (special *special*))
    (foo 5)))
(defun foo (x)
```

```
(let ((*special* (1+ *special*)))
  (declare (special *special*))
  (if (eq x 5) (break "sorry, bad number") *special*)))
```

When we evaluate `(bar)`, **foo** will be called with argument 5 and `break` will put Lisp into a break loop. We decide to go to the frame where **foo** is called and recall **foo** with a different argument. Here is the stack after the break. We go down two frames to reach the frame where **foo** is called.

```
;; Note: this manual is generic over several implementations.
;; The stack may differ from one implementation to another
;; so the stack you see may differ in detail from what is
;; reproduced here.
```

```
USER(4): (bar)
Break: sorry, bad number
```

```
Restart actions (select using :continue):
```

```
  0: return from break.
```

```
[1c] USER(5): :zoom
```

```
Evaluation stack:
```

```
(BREAK "sorry, bad number")
->(IF (EQ X 5) (BREAK "sorry, bad number") ...)
  (LET (#) (DECLARE #) ...)
  (FOO 5)
  (LET (#) (DECLARE #) ...)
  (BAR)
  (EVAL (BAR))
  (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
  (TPL:START-INTERACTIVE-TOP-LEVEL
   #<EXCL::BIDIRECTIONAL-TERMINAL-STREAM @ #x162f5e>
   #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP @ #x282156>
   ...)
```

```
... more older frames ...
```

```
[1c] USER(6): :dn 2
```

```
Evaluation stack:
```

```
(BREAK "sorry, bad number")
(IF (EQ X 5) (BREAK "sorry, bad number") ...)
(LET (#) (DECLARE #) ...)
->(FOO 5)
  (LET (#) (DECLARE #) ...)
  (BAR)
  (EVAL (BAR))
  (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
```

```
(TPL:START-INTERACTIVE-TOP-LEVEL
  #<EXCL::BIDIRECTIONAL-TERMINAL-STREAM @ #x162f5e>
  #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP @ #x282156>
...)
```

... more older frames ...

```
[1c] USER(7):
```

Let us compare the following two commands that could be entered at this point. We show the commands and what is returned and then we explain why the returned values are different.

```
:return (foo 6) -> 3
:restart '(foo 6) -> 2
```

Both avoid the call to **break** since the argument to **foo** is not 5. However, the **:return** command causes **bar** to return the value 3 while the **:restart** command causes **bar** to return the value 2. The binding of **special** which took place in **foo** before the call to **break** is not undone by **:return**. It is undone by **:restart**.

The moral is that you can use **:return** with a form argument when you know what value is expected and there are *no* side effects that you care or are concerned about. You should use **:restart** when side effects are important.

In the following example, we show **:zoom** displaying the evaluation stack, and show how to use some of the other stack manipulation commands.

```
;; Note: this manual is generic over several implementations.
;; The stack may differ from one implementation to another
;; so the stack you see may differ in detail from what is
;; reproduced here.
```

```
user(2): (defun func (x) (* x (bar x)))
```

```
func
```

```
user(3): (defun bar (x) (car x))
```

```
bar
```

```
user(4): (func 10)
```

```
Error: Attempt to take the car of 10 which is not a cons.
```

```
[condition type: simple-error]
```

```
[1] user(5): :zo
```

```
Evaluation stack:
```

```
(error simple-error :format-control ...)
->(car 10)
(bar 10)
(func 10)
(eval (func 10))
(tpl:top-level-read-eval-print-loop)
(tpl:start-interactive-top-level
  #<excl::bidirectional-terminal-stream @ #x2a21de>
```

```

      #<Function top-level-read-eval-print-loop @ #x2d59c6>
...))
[1] user(6): :find bar
Evaluation stack:

  (error simple-error :format-control ...)
  (car 10)
->(bar 10)
  (func 10)
  (eval (func 10))
  (tpl:top-level-read-eval-print-loop)
  (tpl:start-interactive-top-level
    #<excl::bidirectional-terminal-stream @ #x2a21de>
    #<Function top-level-read-eval-print-loop @ #x2d59c6>
...))
[1] user(7): :return 5
;; :RETURN simply returns 5 from the frame
;; resulting in FUNC returning 50.
50
user(8): (func 10)
Error: Attempt to take the car of 10 which is not a cons.
  [condition type: simple-error]
[1] user(9): :find func
Evaluation stack:

  (error simple-error :format-control ...)
  (car 10)
  (bar 10)
->(func 10)
  (eval (func 10))
  (tpl:top-level-read-eval-print-loop)
  (tpl:start-interactive-top-level
    #<excl::bidirectional-terminal-stream @ #x2a21de>
    #<Function top-level-read-eval-print-loop @ #x2d59c6>
...))
[1] user(10): (defun bar (x) (car (list x)))
;; BAR is now just the identity function, but it will work
bar
[1] user(11): :error
Attempt to take the car of 10 which is not a cons.
[1] user(12): :current
(func 10)
[1] user(13): :restart
100
user(14): (defun fact (n)
          (cond ((= 1 n) 1) (t (* n
                                (fact (1- n))))))

```

```

fact
user(15): (fact nil)
Error: nil is an illegal argument to =
  [condition type: type-error]
[1] user(16): :zo
Evaluation stack:

  (error type-error :datum ...)
->(= 1 nil)
  (cond (# 1) (t #))
  (fact nil)
  (eval (fact nil))
  (tpl:top-level-read-eval-print-loop)
  (tpl:start-interactive-top-level
   #<excl::bidirectional-terminal-stream @ #x2a21de>
   #<Function top-level-read-eval-print-loop @ #x2d59c6>
  ...)
[1] user(17): :pop
user(18):

```

9.0 Ghost frames in backtraces

9.1 Summary of the ghost frames section

- When code compiled so that the tail-merge optimization is performed, frames that should (according to the source code) appear on the stack may not be there. Although such optimized code may run faster and use less stack, debugging is made harder.
- The debugger will, where possible, display such missing frames by identifying the function called within brackets ([]).
- Ghost frames do not correspond to any actual frames on the stack, and therefore, they cannot become the current frame, or be restarted or returned from.
- The disassembler must be loaded into the system for ghost frames to be displayed. When the disassembler is loaded, the string "disasm" appears on the `cl:*modules*` list. The debugger will not load the disassembler if it is not present. It will print a message at the end of the backtrace saying that ghost frames are not displayed because the disassembler is not loaded. The disassembler can be loaded by calling the function `disassemble` function or by evaluating `(require :disasm)`. It is not an error to evaluate that form when the disassembler is already loaded.
- If a ghost frame includes suspension points (``...'`), the debugger has determined that additional frames have been optimized off the stack by tail merging but that the functions called in these missing frames cannot be determined.

- The debugger may or may not be able to discover all ghost frames. The fact that no ghost frames appear does not mean that frames have not been optimized off the stack. Similarly, the fact that suspension points do not appear in a ghost frame does not mean that the function in the ghost frame called the next frame up the stack.

In the following backtrace, the frame `[... foo]' is a ghost frame. It indicates that **start-fun** called **foo** and that **foo** did not itself call **list-length**, so **foo** must have called something else before a call to **list-length**:

```
(error type-error :datum ...)
->(list-length 1)
  [... foo]
  (start-fun 1)
  (eval (start-fun 1))
```

The example that generated this backtrace is given in [Section 9.3 What kinds of optimizations cause ghost frames?](#) below. From the example, we know that **start-fun** called **foo**, which called **bar**, which called **baz**, which called **list-length**, and looking at the backtrace, we know that calls to **foo**, **bar**, and **baz** have been optimized out, but the system can still detect that **foo** must have been called.

The remainder of this section contains the following headings:

[Section 9.2 What is a ghost frame?](#)

[Section 9.3 What kinds of optimizations cause ghost frames?](#)

[Section 9.4 How does the debugger know about ghost frames?](#)

[Section 9.5 When will the debugger display ghost frames?](#)

[Section 9.6 Can I return from or restart a ghost frame?](#)

[Section 9.7 What do the suspension points \(^...'\) mean in a ghost frame?](#)

[Section 9.8 The ghost frame has no `...'s; are all possible frames displayed?](#)

[Section 9.9 No ghost frames are displayed. Do all functions appear on the stack?](#)

[Section 9.10 Ghost frames in a brief backtrace](#)

[Section 9.11 Can I turn off printing of ghost frames?](#)

[Section 9.12 Can backtraces involving system functions have ghost frames?](#)

[Section 9.13 Ghost frames and Allegro Composer](#)

9.2 What is a ghost frame?

Typically, Lisp executes a series of function calls. It maintains a stack of function call frames that show the sequence of functions, each calling the next, that resulted in the current state of Lisp. However, compiled code may cause the frames associated with functions actually called to be removed from the stack. Doing so makes code run faster but means that a backtrace (displayed by the top-level commands **:zoom** and **:bt**) does not show calls to some functions actually called, thus making debugging more difficult.

The debugger will where possible insert frames into a backtrace to indicate either that a specific function was called but optimization has removed the associated frame from the stack or that some function was called and removed from the stack by optimization although the debugger cannot determine exactly which function.

These inserted frames are called *ghost frames*. They are indicated in moderate or verbose printed backtraces by being surrounded with brackets ([]). Consider the following backtraces. The first shows the interpreted code. The second shows compiled code at maximum optimization. The `[... foo]` frame in the second backtrace is a ghost frame, indicating that there was a call to **foo** whose associated frame is no longer on the stack. The suspension points (``...'`) further indicate that other functions were called whose associated frames have been removed from the stack, although the debugger cannot determine which ones (we of course know from the interpreted backtrace that the missing functions are **bar** and **baz**):

```
; Interpreted code backtrace:
  (error type-error :datum ...)
->(list-length 1)
   (baz 1)
   (bar 1)
   (foo 1)
   (start-fun 1)
   (eval (start-fun 1))
```

```
; Compiled code backtrace:
  (error type-error :datum ...)
->(list-length 1)
   [... foo]
   (start-fun 1)
   (eval (start-fun 1))
```

9.3 What kinds of optimizations cause ghost frames?

Frames get removed from the stack by a compiler optimization called non-self tail merging. Consider the following example (which generated the backtraces shown above when compiled with speed 2, debug 1):

```
(defun start-fun (x) (foo x) nil)
(defun foo (x) (print x) (bar x))
(defun bar (x) (princ x) (baz x))
(defun baz (x) (pprint x) (list-length x))
```

The error for which the backtrace is displayed above occurs when the form `(start-fun 1)` is evaluated. `1` is not a legal argument to **list-length** (which expects a list) and so an error is signaled.

This is a pretty trivial example but it is not uncharacteristic of real code. **foo**, **bar**, and **baz** each accepts an argument, does something with the argument, and then passes the argument to the next function. **foo**, **bar**, and

baz each has the property that once the system has reached the last form, nothing further is required of the function.

Thus, **foo** calls **print** and then calls **bar**. **foo** is going to return whatever **bar** returns and there are no possible side effects that are not caused by **bar**. At the point where **foo** calls **bar**, there is no reason for a frame associated with **foo** to be on the stack, except for debugging. Indeed, keeping the frame on the stack causes the stack to grow more than is absolutely necessary and, when **bar** returns, causes extra instructions (to exit from **foo**) to be executed. Things would run faster if the frame associated with **foo** were simply removed from the stack and the system acted as if **start-fun** had called **bar** directly (the **print** called for in **foo** has already been completed by the time **bar** is called).

The compiler will optimize this code to remove frames as described as long as the compiler optimization qualities `safety`, `space`, `speed`, and `debug` have values such that the switch `tail-call-non-self-merge-switch` is true. In that case, the compiler will generate code which will make the stack look as if the functions were defined as follows:

```
(defun start-fun (x)
  (foo x)
  (bar x)
  (baz x)
  (list-length x))
(defun foo (x) (print x))
(defun bar (x) (princ x))
(defun baz (x) (pprint x))
```

This code will run faster and suppress unnecessary stack growth but the stack will not reflect the actual sequence of function calls. Thus, when **start-fun** is called with the argument 1, **list-length** will signal an error because its argument is not a list and the backtraces shown above will result.

Compiler optimization switches in general and `tail-call-non-self-merge-switch` in particular are described in *compiling.htm*.

9.4 How does the debugger know about ghost frames?

Since all references to **foo**, **bar**, and **baz** in our example have disappeared from the stack by the time Lisp enters a break loop (when **list-length** signals an error), the information on the stack is insufficient to print ghost frames. One piece of information, is available, however: the location to which control should return in **start-fun** when **list-length** has completed. The debugger can access that return location and can (assuming the disassembler has been loaded) disassemble the code for the function named by **start-fun**. Looking at that disassembly, the debugger can determine that the return location is just after a call to **foo**.

Therefore, it knows that **foo** was called but the **foo** frame has been optimized off the stack. From that information, the debugger generates the **foo** ghost frame.

The debugger also knows that some function must have called **list-length** (since the **list-length** frame is still on the stack). Examining the disassembly of **foo** shows that **foo** could not have called **list-length** itself and so the debugger also knows that at least one other function must have been called between **foo** and **list-length** (we know that **bar** and **baz** were called). Therefore, the debugger puts the suspension points in the ghost frame to indicate the missing ghost frames ([... foo]). However, the debugger is not able to tell which function called by **foo** called **list-length**.

9.5 When will the debugger display ghost frames?

In order for the debugger to determine that frames have been optimized off the stack, it must have access to the disassembler. The disassembler in Allegro CL is not included as part of the base system. Instead, it is loaded when needed, typically when the function **disassemble** is called.

When the disassembler is loaded, the system adds the string "disasm" to the list that is the value of `*modules*`. The debugger will not load the disassembler just to print ghost frames. So, when the disassembler is loaded, the debugger will print ghost frames in backtraces. When the disassembler is not loaded, the debugger will not print such frames. In that case, the following message will be printed at the end of the backtrace:

```
(to see any ghost frames, the disassembler must be loaded)
```

9.6 Can I return from or restart a ghost frame?

The short answer is no. Various debugger commands in Allegro CL operate on the current frame. These include **:restart**, **:return**, **:current**, **:local**, etc. Since ghost frames are not really on the stack, applying these commands to a ghost frame does not make sense. In fact, it is not possible to make a ghost frame the current frame. Here is the backtrace shown above:

```
(error type-error :datum ...)
->(list-length 1)
  [... foo]
  (start-fun 1)
  (eval (start-fun 1))
```

The arrow (->) indicates the current frame is the **list-length** frame. If you call the command **:dn** (which moves the current frame down one frame on the stack), the display changes as follows:

```
(error type-error :datum ...)
(list-length 1)
  [... foo]
->(start-fun 1)
```

```
(eval (start-fun 1))
```

That is, the ghost frame is skipped and the next real frame, the **start-fun** frame, becomes the current frame.

9.7 What do the suspension points (^ ...) mean in a ghost frame?

The backtrace shown just above indicates the ghost frame [. . . foo]. The suspension points indicate that other functions were called between **foo** and **list-length**, although the debugger does not know what these functions are.

As described under the heading [Section 9.4 How does the debugger know about ghost frames?](#) above, the debugger can sometimes tell that a ghost frame belongs in the stack and that the function associated with the ghost frame cannot have called the function associated with the next real frame on the stack. Thus, in our example, the debugger can tell that a **foo** frame must have been on the stack (so it adds a ghost frame) and can further tell **foo** did not call **list-length** (and so adds suspension points to the ghost frame).

9.8 The ghost frame has no `...`s; are all possible frames displayed?

We have already said that suspension points (^ ...) indicate that other functions were called between the ghost frame and the next real frame. However, if there are no suspension points, you cannot immediately conclude that no other functions were called.

Consider the following modification of our example above. Instead of defining **foo** this way:

```
(defun foo (x) (print x) (bar x))
```

we define it as follows:

```
(defun foo (x) (let ((y (list-length (list x)))) (print y)) (bar x))
```

(Again, a trivial example since *y* is obviously 1.) We compile **foo** again and again evaluate `(start-fun 1)`. Again, we get an error (from the call to **list-length** in **baz**) and the following backtrace is displayed:

```
(error type-error :datum ...)
->(list-length 1)
 [foo]
 (start-fun 1)
 (eval (start-fun 1))
```

Here, the ghost frame for **foo** has no suspension points even though we know that **foo** called **bar** called **baz**

called **list-length**, where the error occurred. However, because **foo** itself calls **list-length** (for a different purpose), the debugger could not conclude that **foo** did not call **list-length** with an argument which resulted in an error.

Therefore, suspension points indicate missing ghost frames for sure. No suspension points does not by itself mean there are no missing ghost frames.

9.9 No ghost frames are displayed. Do all functions appear on the stack?

First, some functions are hidden by the system from backtraces to avoid unnecessary clutter. These functions are printed when **:zoom** is called with arguments `:all t`.

However, ghost frames may also be missing in cases similar to our example. Suppose **start-fun**, which was originally defined (in [Section 9.8 The ghost frame has no `...':s; are all possible frames displayed?](#)) above:

```
(defun start-fun (x) (foo x) nil)
```

was instead defined as follows:

```
(defun start-fun (x) (funcall 'foo x) nil)
```

Again, we evaluate `(start-fun 1)` and do a **:zoom**. The following backtrace is printed:

```
(error type-error :datum ...)
->(list-length 1)
    (start-fun 1)
    (eval (start-fun 1))
```

The moral of this and the information under the previous heading, therefore, is that printed information is always guaranteed but missing information should not be depended upon to accurately reflect the situation.

9.10 Ghost frames in a brief backtrace

Calling **:zoom** with arguments `:brief t` or calling the **:bt** command prints an abbreviated backtrace. Here is roughly what is printed when **:bt** is called after evaluating `(start-fun 1)`. The original definitions of **foo** and **start-fun** are in [Section 9.3 What kinds of optimizations cause ghost frames?](#) above. Note that you may see additional frames.

```
list-length <-
[... foo] <- start-fun <- eval <- tpl:top-level-read-eval-print-loop <-
tpl:start-interactive-top-level
```

(This backtrace includes frames for functions below **start-fun** in the stack (such as **eval**) that were left out in the backtraces printed above.)

9.11 Can I turn off printing of ghost frames?

If the disassembler is not loaded and so "disasm" is not on the `cl: *modules*` list, ghost frames will not be printed. However, if that string is on `*modules*`, ghost frames will be printed. The debugger will always print ghost frames if it can. Load the disassembler with

```
(require :disasm)
```

There is no way to unload the disassembler. It is typically already loaded at startup in development images. (See the discussion of the *include-devel-env* keyword argument to **build-lisp-image** in *Arguments to build-lisp-image I* in *building-images.htm*.)

9.12 Can backtraces involving system functions have ghost frames?

Yes, they can. Many system (i.e. predefined) functions in Allegro CL are compiled at a settings of speed and debug which cause `tail-call-non-self-merge-switch` to be true. Therefore, it is possible that ghost frames can appear identified with symbols naming system functions.

9.13 Ghost frames and Allegro Composer

Debugger windows in Allegro Composer do not display ghost frames.

10.0 The tracer

The tracer provides a way to track or trace when functions are called. For example, when tracing a function, a message is printed upon entering and exiting the function.

The tracer is invoked at the top level using **:trace** and turned off using **:untrace**. The tracer can also be invoked and exited using the macros **trace** and **untrace**, which have the same argument syntax as their top-level command counterparts.

The output from **:trace** is designed to be readable - a function being traced may be called many times, and the entrance and exit from each instance should be obvious, by the numbers at the beginning of the lines and the indentation of the lines printed by the traced function. Also printed at the beginning is a number in brackets, such as [1]. This number indicates the process and can be associated with an actual process by looking at the information printed by the **:processes** top-level command, in the Bix (Bindstack Index) column.

A couple of notes on tracing in the IDE

- **Output normally goes to the trace dialog:** if you are using the Integrated Development Environment (IDE) with Allegro CL on Windows, trace output will go to the Trace dialog if that dialog is available, whether or not it is visible or iconified. (Output will go to that dialog if it exists and its state, as returned by **state**, is `:normal`, `:maximized`, or `:icon`. Output will go to the Debug window if the Trace dialog does not exist or has state `:shrunk`. Click on the close button of the Trace dialog if you want output to go to the Debug window.
- **Tracing in event code can be suppressed for a specific window.** If the generic function **inhibit-trace-for-object** returns true for a particular window, then tracing will be suppressed for code that runs in event-handling callbacks for that window. This could be useful for eliminating extraneous trace output that would otherwise be generated for tool windows that you are not testing.

:trace prints function names with a package qualifier for functions on symbols not accessible in the current package. The full package name will be printed unless the variable `*print-nickname*` is `t`, in which case the package nickname is printed.

The following are valid options to **:trace**:

Option	Arguments	Description
<code>:condition</code>	<i>expr</i>	Trace this function if <i>expr</i> evaluates to a true value.

:break-before	<i>val</i>	The expression <i>val</i> is evaluated just before entering a function. If <i>val</i> evaluates to a non- <code>nil</code> value, then a new break level is entered. Otherwise, execution continues. When used in combination with <code>:inside</code> and <code>:not-inside</code> , breaks only occur when the <code>:inside</code> and <code>:not-inside</code> conditions are satisfied.
:break-after	<i>val</i>	The expression <i>val</i> is evaluated just after exiting a function. If <i>val</i> is <code>t</code> , then a new break level is entered. Otherwise, execution continues. When used in combination with <code>:inside</code> and <code>:not-inside</code> , breaks only occur when the <code>:inside</code> and <code>:not-inside</code> conditions are satisfied.
:break-all	<i>val</i>	The expression <i>val</i> is evaluated just before entering a function and just after exiting a function. If <i>val</i> is <code>t</code> , then a new break level is entered. Otherwise, execution continues. When used in combination with <code>:inside</code> and <code>:not-inside</code> , breaks only occur when the <code>:inside</code> and <code>:not-inside</code> conditions are satisfied.
:inside	func	Trace this function if Lisp is currently <i>inside</i> a call of the function <i>func</i> . <i>func</i> may also be a list of functions, optionally starting with the symbols cl:and or cl:or . If neither symbol or cl:and starts the list (e.g. <code>(cl:and foo1 bar2)</code> or <code>(foo1 bar2)</code>), tracing is done when the function to be traced has been called directly or indirectly by all the functions in the list (by foo1 and

bar2 in the example). If **cl:or** starts the list (e.g. (`cl:or foo1 bar2`)) tracing is done when the function to be traced has been called directly or indirectly by **any** of the functions in the list (by **foo1** or **bar2** in the example).

`:inside` works in combination with `:not-inside`, described next. Both must be satisfied for tracing to occur.

For example, (`trace (deeper :inside deep)`) would trace the function **deeper** only when called from within a call to **deep**. (`trace (deeper :inside deep :not-inside (cl:or foo1 bar2))`) would trace the function **deeper** only when called from within a call to **deep** but not within a call to **foo1** or **bar2**. See [Note on inside and not inside](#) for a fuller description of a call being inside another.

`:not-inside`

`func`

Trace this function if Lisp is not currently *inside* the evaluation of the function *func*.

func may also be a list of functions, optionally starting with the symbols **cl:and** or **cl:or**. If neither symbol or **cl:and** starts the list (e.g. (`cl:and foo1 bar2`) or (`foo1 bar2`)), tracing is done when the function to be traced has not been called directly or indirectly by **all** the functions in the list (by **foo1** and **bar2** in the example). If **cl:or** starts the list (e.g. (`cl:or foo1 bar2`)) tracing is done

		<p>when the function to be traced has not been called directly or indirectly by any of the functions in the list (by foo1 or bar2 in the example).</p> <p><code>:not-inside</code> works in combination with <code>:inside</code>, described above. Both must be satisfied for tracing to occur.</p> <p>For example, <code>(trace (deeper :not-inside deep))</code> would trace the function deeper except when called from within a call to deep. <code>(trace (deeper :not-inside deep :inside (cl:or foo1 bar2)))</code> would trace the function deeper except when called from within a call to deep and within a call to foo1 or bar2. See Note on inside and not inside for a fuller description of a call being inside another.</p>
<code>:print-before</code>	<i>expr</i>	<i>expr</i> should either be a single object or a list of objects which will be evaluated. The results will be printed before entering the function.
<code>:print-after</code>	<i>expr</i>	<i>expr</i> should either be a single object or a list of objects which will be evaluated. The results will be printed after leaving the function.
<code>:print-all</code>	<i>expr</i>	<i>expr</i> should either be a single object or a list of objects which will be evaluated. The results will be printed before entering and after leaving the function.

Note on inside and not inside

A call to function **foo** is *inside* a call to function **bar** if **bar** appears on the stack when **foo** is called. **bar** can call **foo** directly, meaning there is an explicit call to **foo** in the code defining **bar**, or indirectly, meaning **bar** calls another function which (perhaps calls more intermediate functions) which calls **foo** directly.

There are a few special cases. One is caused by tail-merging. Thus, if **bar** calls **baz** which does a tail-merged call to **foo**, then **foo** is considered inside **bar** but not inside **baz**.

The other special case is generic functions. If a method is on the stack, its generic function is also considered to be on the stack, even though it never will be seen there (meaning you can specify the generic function rather than the specific method). Thus the following all work for tracing **foo** inside a call to **device-read** called on a `terminal-simple-stream`, but the first will catch calls to **foo** inside any **device-read** method.

```
:trace (foo :inside device-read)
:trace (foo :inside #'(method device-read (terminal-simple-stream t t t
t)))
:trace (foo :inside ((method device-read (terminal-simple-stream t t t
t))))
```

Stopping tracing and trace output

Tracing of individual objects or all tracing can be stopped with the **:untrace** command.

The value of `cl:*trace-output*` is the where **:trace** sends output, which is normally `cl:*terminal-io*`. `cl:*print-level*` and `cl:*print-length*` are bound to `*trace-print-level*` and `*trace-print-length*` during trace output.

10.1 Tracing function objects

trace and **:trace** may be passed function names but not function objects. Allegro CL allows specifying function objects to be traced using the **ftrace** and **funtrace** functions.

Each function takes a function specification as an argument. The function specification can be a function name (as with **trace** and **untrace**) or a function object (which is not accepted by **trace** and **untrace**).

untrace and **:untrace** called with no arguments stop all tracing, including tracing started by **ftrace** with a function object as an argument, but you should use **funtrace** to stop tracing of a function object if you want some tracing to continue.

Here is an example:

```
cl-user(1): (defun foo () (break "hello"))
foo
```

```

cl-user(2): (foo)
Break: hello

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this process.
[1c] cl-user(3): :zo
Evaluation stack:

  (break "hello")
->(foo)
  (eval (foo))
  (tpl:top-level-read-eval-print-loop)
  (tpl:start-interactive-top-level
   #<terminal-simple-stream [initial terminal io] fd 0/1 @
    #x7115d9da>
   #<Function top-level-read-eval-print-loop> ...)
[1c] cl-user(4): :func
#<Interpreted Function foo>
[1c] cl-user(5): (ftrace *)
#<Interpreted Function foo>
[1c] cl-user(6): :prt
cl-user(7): (foo) ;; :prt evaluation
 0: (foo)
Break: hello

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this process.
[1c] cl-user(8): :cont
 0: returned nil
nil
cl-user(9): (funtrace #'foo)
#<Interpreted Function foo>
cl-user(10):

```

10.2 Trace example

In the following example, we trace the factorial function and then give an example of tracing one function inside another.

;; Note: this manual is generic over several implementations.

```
;; The stack may differ from one implementation to another
;; so the output you see may differ in detail from what is
;; reproduced here.
```

```
cl-user(38): (defun fact (n)
              (cond ((= n 1) 1)
                    (t (* n (fact (1- n))))))
```

```
fact
```

```
cl-user(39): (fact 4)
```

```
24
```

```
cl-user(40): :trace fact
```

```
(fact)
```

```
cl-user(41): (fact 4)
```

```
0[1]: (fact 4)
```

```
1[1]: (fact 3)
```

```
2[1]: (fact 2)
```

```
3[1]: (fact 1)
```

```
3[1]: returned 1
```

```
2[1]: returned 2
```

```
1[1]: returned 6
```

```
0[1]: returned 24
```

```
24
```

```
cl-user(42): (defun deep (x) (deeper (list x)))
```

```
deep
```

```
cl-user(43): (defun deeper (x) (format t "~&~s~%" x))
```

```
deeper
```

```
cl-user(44): (deep 10)
```

```
(10)
```

```
nil
```

```
cl-user(45): :tr (deeper :inside deep)
```

```
(deeper)
```

```
cl-user(46): (deeper 10)
```

```
10
```

```
nil
```

```
cl-user(47): (deep 10)
```

```
0[1]: (deeper (10))
```

```
(10)
```

```
0[1]: returned nil
```

```
nil
```

```
cl-user(48): :tr (deeper :break-before t)
```

```
(deeper)
```

```
cl-user(49): (deep 10)
```

```
0[1]: (deeper (10))
```

```
Break: traced call to deeper
```

```
Restart actions (select using :continue):
```

```
0: return from break.
```

```
1: Return to Top Level (an "abort" restart).
```

2: Abort entirely from this process.

```
[1c] cl-user(50): :zo
```

Evaluation stack:

```
(break "traced call to ~s" deeper)
```

```
->(deep 10)
```

```
(eval (deep 10))
```

```
(tpl:top-level-read-eval-print-loop)
```

```
(tpl:start-interactive-top-level
```

```
  #<terminal-simple-stream [initial terminal io] fd 0/1 @ #x4095faa>
```

```
  #<Function top-level-read-eval-print-loop> ...)
```

```
[1c] cl-user(51): :cont
```

```
(10)
```

```
0[1]: returned nil
```

```
nil
```

```
;; The [1] that appears in the trace output identified the process
;; being traced. It can be associated with an actual process
;; by looking at the output of the :processes top-level
;; command (nicknamed :proc), under the Bix (bindstack index)
;; column. We see from the output it is the Initial
;; Lisp Listener.
```

```
cl-user(52): :proc
```

```
P Bix Dis   Sec  dSec  Priority  State   Process Name, Whostate, Arrest
```

```
*   1  16   279   4.5         0 runnable Initial Lisp Listener
```

```
*   3   0    0    0.0         0 waiting  Connect to Emacs daemon,
                                     waiting for input
```

```
*   4   0    0    0.0         0 inactive Run Bar Process
```

```
*   6   0    2    0.0         0 waiting  Editor Server, waiting for input
```

```
cl-user(53):
```

```
;; In the next example, we define functions that call other
;; functions after printing messages about what they are calling
;; what called them. This illustrates the :inside and :not-inside
;; options.
```

```
cl-user(53): (defun foo ()
```

```
  (progn (format t "foo calling bar~%" ) (bar 'foo))
```

```
  (progn (format t "foo calling baz~%" ) (baz 'foo))
```

```
  (progn (format t "foo calling bzz~%" ) (bzz 'foo)))
```

```
foo
```

```
cl-user(54): (defun bar (from)
```

```
  (progn (format t "bar calling baz from ~S~%" from)
         (baz 'bar))
```

```
  (progn (format t "bar calling bzz from ~S~%" from)
         (bzz 'bar)))
```

```
bar
```

```
cl-user(55): (defun baz (from)
              (progn (format t "baz calling bzz from ~S~%" from)
                     (bzz 'baz)))
```

```
baz
```

```
cl-user(56): (defun bzz (from)
              (format t "bzz called from ~S~%" from))
```

```
bzz
```

```
;; Here is an untraced call to FOO:
```

```
cl-user(57): (foo)
foo calling bar
bar calling baz from foo
baz calling bzz from bar
bzz called from baz
bar calling bzz from foo
bzz called from bar
foo calling baz
baz calling bzz from foo
bzz called from baz
foo calling bzz
bzz called from foo
nil
```

```
;; Here we trace BZZ when called inside both FOO and BAZ (:inside
;; followed by a list contains an implicit AND, so the call
;; must be inside all listed functions):
```

```
cl-user(58): :trace (bzz :inside (foo baz))
(bzz)
```

```
cl-user(59): (foo)
foo calling bar
bar calling baz from foo
baz calling bzz from bar
0[1]: (bzz baz)
bzz called from baz
0[1]: returned nil
bar calling bzz from foo
bzz called from bar
foo calling baz
baz calling bzz from foo
0[1]: (bzz baz)
bzz called from baz
0[1]: returned nil
foo calling bzz
bzz called from foo
```

```
;; this call to BZZ is not traced since
;; it is not inside BAZ.
```

```
nil
```

```

cl-user(60): :untrace ;; We remove all tracing
nil

;; Now tracing will happen when inside FOO but not inside BAR:
cl-user(61): :trace (bzz :inside foo :not-inside bar)
(bzz)
cl-user(62): (foo)
foo calling bar
bar calling baz from foo
baz calling bzz from bar
bzz called from baz ;; inside BAR so not traced.
bar calling bzz from foo
bzz called from bar
foo calling baz
baz calling bzz from foo ;; not inside BAR
0[1]: (bzz baz)
bzz called from baz
0[1]: returned nil
foo calling bzz
0[1]: (bzz foo)
bzz called from foo ;; Again, not inside BAR
0[1]: returned nil
nil
cl-user(63):

```

CLOS methods can be traced by name. Here is an example.

```

user(16): (defmethod my-function ((x integer))
          (cons x :integer))
#<standard-method my-function (integer)>
user(17): (my-function 1)
(1 . :integer)
user(18): (trace ((method my-function (integer))))
(method my-function (integer))
user(19): (my-function 1)
0: ((method my-function (integer)) 1)
0: returned (1 . :integer)
(1 . :integer)
user(20): (untrace (method my-function (integer)))
((method my-function (integer)))
user(21): (my-function 1)
(1 . :integer)
user(22):

```

10.3 Tracing `setf`, `:before`, and `:after` methods and internal functions

Methods and internal functions are typically named by function specs (see *Function specs (fspecs)* in *implementation.htm*). This section describes how to trace such things, concentrating on `setf`, `:before` and `:after` methods and internal functions (such as those defined by **flet** or **labels**).

Here is how to trace **setf**, `:before` and `:after` methods. Note that the methods must be defined before they can be traced (we have not done that -- the examples simply show the format of the calls to **trace** and **untrace**):

```
(trace ((method (setf slot-1) (t baz))))
(trace ((method foo :before (integer))))
(trace ((method foo :after (integer))))
```

The extra set of parentheses is required to avoid confusion with specifying trace options (they are specified with a list whose car is the function to be traced and whose cdr is a possibly empty list of options). Note that the extra set of parentheses is not used with **untrace**:

```
(untrace (method (setf slot-1) (t baz)))
(untrace (method foo :before (integer)))
(untrace (method foo :after (integer)))
```

A generic function itself can be traced exactly like any other function.

Here is an example tracing an internal function, defined in a **labels** or **flet**. Note that the form containing the call to **labels** or **flet** must be compiled.

```
cl-user(1): (defun myfunc (a b c)
             (labels ((cubit (arg) (expt arg 3)))
               (if (> a b)
                   (+ (cubit a) (cubit c))
                   (+ (cubit b) (cubit c)))))
```

myfunc

```
cl-user(2): (compile 'myfunc)
```

myfunc

nil

nil

```
cl-user(4): (trace ((labels myfunc cubit)))
```

```
((labels myfunc cubit))
```

```
cl-user(5): (myfunc 1 2 3)
```

```
0: ((labels myfunc cubit) 2)
```

```
0: returned 8
```

```
0: ((labels myfunc cubit) 3)
```

```
0: returned 27
```

35

```
cl-user(6):
```

Here is another example. Again, the form must be compiled.

```

;;; File foo.cl:

(in-package :cl-user)
(defclass thing ()
  ((s1 :initform 1 :initarg :s1 :accessor s1)
   (s2 :initform 2 :initarg :s2 :accessor s2)
   (s3 :initform 3 :initarg :s3 :accessor s3)))

(defmethod doit ((arg thing))
  (labels ((cubit (arg) (expt arg 3)))
    (if (> (s1 arg) (s2 arg))
        (+ (cubit (s1 arg)) (cubit (s3 arg)))
        (+ (cubit (s2 arg)) (cubit (s2 arg))))))

;;; End of file foo.cl:

cl-user(1): :ld foo.cl
; loading /home/user1/foo.cl
cl-user(2): (trace ((labels (method doit (thing)) cubit)))
Error: `(labels (method doit (thing)) cubit)' is not fbound
 [condition type: undefined-function]

restart actions (select using :continue):
0: return to Top Level (an "abort" restart).
1: Abort entirely from this process.

      ;;; The form must be compiled so that internal functions
      ;;; can be traced.

[1] CL-USER(3): :res
cl-user(4) :cload foo.cl
;;; Compiling file foo.cl
;;; Writing fasl file foo.fasl
;;; Fasl write complete
cl-user(5): (trace ((labels (method doit (thing)) cubit)))
((labels (method doit (thing)) cubit))
cl-user(6): (setf thing1 (make-instance 'thing))
#<thing @ #x7150fac2>
cl-user(7): (doit thing1)
0: ((labels (method doit (thing)) cubit) 2)
0: returned 8
0: ((labels (method doit (thing)) cubit) 2)
0: returned 8
16

```

```
cl-user(8):
```

11.0 The stepper

The stepper allows the user to watch and control the evaluation of Lisp expressions, either inside certain functions or over certain expressions. When stepping is turned on, evaluation of all expressions is done in single-step mode - after evaluating one form, a step read-eval-print loop is entered, from which the user may continue or abort.

Note: starting in release 6.0, it is only useful to step through **compiled code**. (In earlier release, it was only useful to step through interpreted code.) Because of the changes that allow stepping through, you cannot effectively step through interpreted code. (Doing so results in stepping through the interpreter itself, with hundreds of uninteresting steps). Further, see `verify-funcalls-switch`. **funcall**'ed functions in certain cases in compiled code will not be caught by the stepper if the call was compiled with that compiler switch false.

If you ever see the entry

```
<excl::interpreted-funcall ...>
```

immediately enter the command `:sover`.

If you instead take another step (by hitting Return, for example), get out of stepping by entering

```
[step] cl-user(20): :step nil
[step] cl-user(21): :sover
```

and then you can start stepping afresh.

The `:step` top-level command is similar to the standard Common Lisp macro `step`.

With no arguments or an argument of `nil`, `:step` turns off stepping. With an argument of `t`, stepping is turned on globally. Otherwise the arguments must be symbols naming functions, and stepping is done only when inside one of the functions given to `:step`.

Once stepping is turned on, the top level recognizes three more commands: `:scont`, `:sover`, and carriage return (which is a synonym for `:scont 1`). Also, the top-level prompt for read-eval-print loops when stepping is enabled is prefixed with `[step]`, as a reminder that the above step commands are available.

11.1 Turning stepping off

Entering `:step nil` turns off stepping. You may have to then enter `:sover` to end the current stepping through a function. Turning off stepping is very useful when you accidentally start stepping through an interpreted function.

```
user(45): (defun fact (n) (if (= n 1) 1 (* n (fact (1- n)))))
fact
user(46): :step fact
user(47): (fact 3)
1: (fact 3)

[step] user(48):
2: (excl::interpreted-funcall #<Interpreted Function fact> 1 32864546 0)

;; :sover here would get you out

[step] user(48):
3: (excl::extract-special-decls ((block fact (if # 1 #))))

;; it is now too late. Turn stepping off.

[step] user(48): :step nil
[step] user(49): :sover
result 3: nil ((block fact (if (= n 1) 1 (* n #))))
result 2: 6
result 1: 6
6
user(50):
```

11.2 Other stepping commands and variables

Command	Arguments	Description
:scont	<i>n</i>	Continue stepping, for <i>n</i> expressions, beginning with the evaluation of the last expression printed by the stepper. If <i>n</i> is not provided, it defaults to 1.
:sover		Evaluate the current expression in normal, non-stepping mode.

`cl:*print-level*` and `cl:*print-length*` are bound to `*step-print-level*` and `*step-print-length*` during stepper output.

11.3 Stepping example

The example below demonstrates the use of the stepper. In it, we define `fact`, the factorial function, and then step through it. We use carriage returns for each single step. Notice particularly the difference between stepping through `fact` when it runs interpreted and when it runs compiled.

```
;; Note: this manual is generic over several implementations.
;; The stack may differ from one implementation to another
;; so the output you see may differ in detail from what is
;; reproduced here.
```

```
user(3): (defun fact (n)
          (if (= n 1) 1 (* n (fact (1- n)))))
fact
user(4): (fact 3)
6
user(5): (compile 'fact)
fact
user(6): :step nil
          ;; we turn all stepping off
user(7): (step fact)
;; Here we use the function form of step.
;; It has the same effect as :step fact.

user(8): (fact 3)
1: (fact 3)

[step] user(9):
2: (excl::*_2op 2 1)

[step] user(9):
result 2: 2
2: (excl::*_2op 3 2)

[step] user(8):
result 2: 6
result 1: 6
6
user(9):
```

12.0 The Lisp DeBug (ldb) stepper

The `ldb` stepping facility allows the user to place breakpoints at various instruction-level locations. Note that the `ldb`-stepping functionality may not be included in the running image. Evaluate `(require :lldb)` to ensure that the functionality is loaded before trying to use the facility.

The `ldb` stepper utilizes several top-level commands to create the stepping environment.

They are:

:ldb	Controls the installing (enabling) and uninstalling (disabling) of breakpoints, and the establishment of <code>ldb</code> mode.
:break	Allows the user to add or delete breakpoints.
:step	(Modified from earlier UNIX releases.) When in <code>ldb</code> -step mode, the <code>step</code> command recognizes several sub-commands as its first argument. See the full description for more information.
:local	(Modified from earlier UNIX releases.) When a breakpoint is hit, the default current frame is the register context in which the breakpoint occurred (and thus <code>ldb</code> -step mode was entered). See the full description for more information.
[<code>return</code>]	(Modified from earlier UNIX releases.) When in <code>ldb</code> -step mode, causes the previous step subcommand to be re-executed. If there was no previous step subcommand, a :step over is assumed.

12.1 Entering and Exiting the ldb stepper

Before starting `ldb`-mode stepping, be sure that the functionality is loaded by evaluating:

```
(require :lldb)
```

Entering `ldb`-mode can then be done in one of two ways:

1. Use the `:ldb t` command to start debugging
2. Use the **with-breakpoints-installed** macro around a form.

Either method allows breakpoints to be hit by installing them. If the **:break** command is used after the **:ldb** command, any new breakpoints will be properly installed.

Once a break is hit, a message is printed out, and lldb-step mode is entered. In this mode, breakpoints are not installed, but the top-level is primed to do fast installation/deinstallation/stepping of the breakpoints with little keyboard input.

12.2 Ldb stepper functional interface

The lldb stepper provides the following functional interface.

add-breakpoint	Adds a breakpoint.
delete-breakpoint	Deletes an existing breakpoint.
with-breakpoints-installed	Evaluates a body with breakpoints installed during the evaluation.

12.3 Ldb stepping example run

This example run is on a linux x86 version. The output on different platforms will be different. Comments are added within ;[]:

```

cl-user(1): (require :lldb)
Fast loading lldb.fasl
cl-user(2): :br count 17 ;[add a breakpoint (but do not install)]
Adding #<Function count>: 17
17: 89 45 dc movl [ebp-36],eax ; item
cl-user(3): (count #\a "abacad")
3 ;[no breakpoints had been installed]
cl-user(4): :ldb t ;[start things out]
[ldb] cl-user(4): (count #\a "abacad")
Hit breakpoint at func = #<Function count>, pc=17
breakpoint-> 17: 89 45 dc movl [ebp-36],eax ; item
[ldb-step] cl-user(5):
Hit breakpoint at func = #<Function count>, pc=20
temp brkpt-> 20: 89 55 cc movl [ebp-52],edx ; sequence
[ldb-step] cl-user(6): :br ;[ask to show current breakpoints]
#<Function count>: (17 (20 :temp))
[ldb-step] cl-user(7): :loc :%eax ;[eax is the first arg on x86]
#\a
[ldb-step] cl-user(8): :loc :%edx ;[edx is the second arg on x86]
"abacad"
[ldb-step] cl-user(9): :step into ;[this will have no effect until later]

```

```

Hit breakpoint at func = #<Function count>, pc=23
temp brkpt-> 23: 8d 49 fe leal ecx,[ecx-2]
[ldb-step] cl-user(9):
Hit breakpoint at func = #<Function count>, pc=26
temp brkpt-> 26: 8d 45 10 leal eax,[ebp+16] ; (argument 2)
[ldb-step] cl-user(9):
Hit breakpoint at func = #<Function count>, pc=29
temp brkpt-> 29: 8d 95 5c ff leal edx,[ebp-164]
ff ff
[ldb-step] cl-user(9):
Hit breakpoint at func = #<Function count>, pc=35
temp brkpt-> 35: 33 db xorl ebx,ebx
[ldb-step] cl-user(9):
Hit breakpoint at func = #<Function count>, pc=37
temp brkpt-> 37: b3 c8 movb bl,$200
[ldb-step] cl-user(9):
Hit breakpoint at func = #<Function count>, pc=39
temp brkpt-> 39: ff 57 4f call *[edi+79] ; kwdschan
[ldb-step] cl-user(9): ;[note that primcalls are _not_ stepped into]
Hit breakpoint at func = #<Function count>, pc=42
temp brkpt-> 42: 8d 9d 5c ff leal ebx,[ebp-164]
ff ff
[ldb-step] cl-user(9):
[some stepping deleted here]
Hit breakpoint at func = #<Function count>, pc=155
temp brkpt-> 155: 3b 7d e0 cmpl edi,[ebp-32] ; end
[ldb-step] cl-user(9):
Hit breakpoint at func = #<Function count>, pc=158
temp brkpt-> 158: 0f 85 93 00 jnz 311
00 00
[ldb-step] cl-user(9):
Hit breakpoint at func = #<Function count>, pc=164
temp brkpt-> 164: 8b 45 cc movl eax,[ebp-52] ; sequence
[ldb-step] cl-user(9):
Hit breakpoint at func = #<Function count>, pc=167
temp brkpt-> 167: 8b 9f 9f fe movl ebx,[edi-353] ; length
ff ff
[ldb-step] cl-user(9):
Hit breakpoint at func = #<Function count>, pc=173
temp brkpt-> 173: b1 01 movb cl,$1
[ldb-step] cl-user(9): :loc :%eax
"abacad"
[ldb-step] cl-user(10): :step into
Hit breakpoint at func = #<Function count>, pc=175
temp brkpt-> 175: ff d7 call *edi
[ldb-step] cl-user(11):
Hit breakpoint at func = #<Function length>, pc=0

```

```
temp brkpt-> 0: 8b c8 movl ecx,eax
[ldb-step] cl-user(11):
Hit breakpoint at func = #<Function length>, pc=2
temp brkpt-> 2: 80 e1 03 andb cl,$3
[ldb-step] cl-user(11): :br ;[check breakpoints again]
#<Function count>: (17)
#<Function length>: ((2 :temp))
[ldb-step] cl-user(12): :br nil ;[turn off all breakpoints]
[ldb-step] cl-user(13):
3 ;[get right answer]
[ldb] cl-user(13):
```