Allegro CL のデバッグ機能

茂野 真弓

2009 年 8 月 31 日

目 次

1　はじめに

プログラムの開発中にエラーが発生したら、まずはエラー発生箇所を確認し、それからエラーの原因を調査して修正する必要がある。場合にもよるが、エラー発生箇所が特定できても、エラーになった原因であるバグはそれよりも前の段階にあることがある。
例えば、ある関数から他の関数を呼び出してエラーが起こった場合、呼び出した側の関数で引数として渡した変数にエラーとなるような値をバインドしてしまっていることなどがある。
このような場合はエラーを起こした関数に渡った値の確認はもちろん、エラーを起こした関数がどの関数から呼ばれたかという関数の呼び出し関係や、呼び出した側の関数内部でどの変数にどういう値をバインドしたかを調査できるとデバッグには大変有効である。
もちろん、エラーが発生しそうな箇所にデバッグプリントを入れていくというのもデバッグには有効な手段である。ただし、大規模なプログラムだと何箇所にもデバッグプリントを入れる必要があるなど、それだけで手間がかかってしまう。

本レポートでは Allegro CL のデバッグ機能のうち、関数の呼び出しなどの評価がどの順で行われたか、また、関数内部で変数のバインドがどうなっているのかを調査するためのスタックコマンドを説明する。また、特定の関数やメソッドについて、呼び出されたときの引数と終了時の返り値を呼出しごとに表示する tracer について説明する。

2 スタックコマンド

まず、用語とスタックの見方を説明する。

runtime stack

Lisp 関数の引数とローカル変数が保存される場所の実体。

関数呼び出しがあると、呼び出された関数のために runtime stack 上の領域が確保される。また、確保された領域の一部に、引数が評価されて置かれる。呼び出された関数が引数へアクセスする場合は、このスタックを参照する。また、関数内のローカル変数についてもスタック上に確保される。単に スタックという場合は、runtime stack を指す。

stack frame

ひとつの関数呼び出しの引数とローカル変数が置かれるスタック上の領域。単に frame という場合は、stack frame のことを指す。例えば、`foo -> bar -> yaf` という順で関数が呼ばれると、foo の実行に入るときに foo の frame が、bar の実行に入るときに bar の frame が、yaf の実行に入るときに yaf の frame がそれぞれ作られる。つまり、yaf の実行に入った時点ではこれら 3 つの関数に関する frame が作られる。

実際に stack 上に作られる frame は、frame object と呼ばれる Lisp object である。デバッグのときなどは frame object そのものではなく、ユーザにわかりやすい形式 (リスト形式など。ただし表示モードによって異なる) に変換して表示される。本レポートでは frame object そのものの表示や直接の操作はしないが、frame は実際には Lisp object である。リストなどではない。

次のスタックの表示例では、関数 foo, bar, baz についての frame が表示されている。

スタック表示例

```
Evaluation stack:

   (BREAK "baz")
 ->(BAZ 3)
   (BAR 2)
   (FOO 1)
   (EVAL (FOO 1))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<MULTIVALENT stream socket connected from localhost/9916 to ....
      TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP ...)
```

frame は上に表示されるものほど実行順序が新しく、下に表示されるものほど実行順序が古い。この例では foo が bar を呼び、bar が baz を呼び、baz の実行中に break したことを意味している。また実行中の frame だけが表示されるので、foo の内部で bar を呼び出す前に他の関数を呼んでいたとしても、それが既に終了している場合はその関数の frame は表示されない。

次の節から説明するスタックコマンドは、スタックのアクセスと表示を行うためのものである。スタック中のある frame を調べるなどした後は、その frame は current stack frame または current

frame と呼ばれ、他の frame と区別がつくように、矢印-> などのポインタがついた状態で表示される。上の例のように、普通は break した直後にスタックを表示すると、break の原因に一番近い frame が current frame となっている。

2.1 :zoom

トップレベルコマンド :zoom **はスタックを表示する。エラーが発生した直後に** :zoom **を実行すると、エラーが発生した** frame **に矢印** -> **などのポインタ** (**表示モードによって異なる**) **がついて表示される。**

次は、未定義の変数 foo **を評価したときのエラーである。このエラーを例に、次の項から** :zoom **コマンドのモードを説明する。**

エラーサンプル

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

Restart actions (select using :continue):
 0: Try evaluating FOO again.
 1: Set the symbol-value of FOO and use its value.
 2: Use a value without setting FOO.
 3: Return to Top Level (an "abort" restart).
 4: Abort entirely from this (lisp) process.
[1] CL-USER(3):
```

2.1.1 :brief, :moderate, :verbose モード

ここで紹介する :brief, :moderate, :verbose **の** 3 **つのモードは、**:zoom **コマンドで表示される** frame **の表示形式を指定するものである。これらのモードは一度にひとつしかオンにできない。オンにするときは** nil **以外の値を指定する。また、次に** :zoom **コマンドで他のモードをオンにするまでは、同じモードで表示される。**

:brief

frame **ごとに関数名だけが表示される。一行につき一つ以上の関数名が表示される。**current frame **が表示される行は、**current frame **の関数名だけが表示される。また、**current frame **の行だけインデントが他の行とは異なる。**

:brief **モードの場合**

```
[1] CL-USER(4): :zoom :brief t
Evaluation stack:

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

:moderate

関数名と引数が表示される。また、各 frame **がそれぞれ別の行に表示される。**

:moderate **モードの場合**

```
[1] CL-USER(5): :zoom :moderate t
Evaluation stack:

   (ERROR #<UNBOUND-VARIABLE @ #x4c4e1e2>)
 ->(EVAL FOO)
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
```

:verbose

関数名と引数が表示される。また、各 frame **についての情報が複数行にまたがって表示される。**

:verbose モードの場合

```
[1] CL-USER(6): :zoom :verbose t
Evaluation stack:

   call to ERROR
required arg: EXCL::DATUM = #<UNBOUND-VARIABLE @ #x4c4e1e2>
&rest EXCL::ARGUMENTS = NIL
function suspended at relative address 312

----------------------------
 ->call to EVAL
required arg: EXP = FOO
function suspended at relative address 68

----------------------------
   call to TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP
function suspended at relative address 300

----------------------------
   call to TPL:START-INTERACTIVE-TOP-LEVEL
required arg: *TERMINAL-IO* = #<TERMINAL-SIMPLE-STREAM
                                [initial terminal io] fd 0/1 @ #x4115742>
required arg: FUNCTION = #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP>
required arg: TPL::ARGS = NIL
&key TPL::INITIAL-BINDINGS = :UNSUPPLIED
function suspended at relative address 432

----------------------------
[1] CL-USER(7):
```

2.1.2 :all t, :all nil モード

一部の frame は隠されていて、表示されないことがある。このような場合、:all nil が設定されている。表示するには、:all t を設定する。この 2 つのモードの設定は以降の :zoom の実行時にもそのまま使われる。切り替えたければ、:zoom 実行時に明示的にモードを設定しなおす必要がある。

以下は前項と同様、未定義の変数 foo を評価したときに:all t と :all nil を実行した例である。:all nil のときは表示されなかった frame が、:all t のときは表示されることがわかる。

:all nil, :all t

```
[1] CL-USER(7): :zoom :all nil :moderate t
Evaluation stack:

   (ERROR #<UNBOUND-VARIABLE @ #x20d22112>)
 ->(EVAL FOO)
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<MULTIVALENT stream socket connected from localhost/3872 to
        localhost/3874 @ #x20a321e2>
      TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP ...)
[1] CL-USER(8): :zoom :all t
Evaluation stack:

... 3 more newer frames ...

   (EXCL::GENERAL-ERROR-HANDLER 5 FOO ...)
   (EXCL::CER-GENERAL-ERROR-HANDLER-ONE 5 FOO)
   (SYS::..CONTEXT-SAVING-RUNTIME-OPERATION)
   (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] CL-USER(9):
```

:all nil モードで表示されない (隠される) ものは、後述する :hide コマンドにより、隠すように指定された関数の frame や、同じく隠すように指定されたパッケージの関数の frame や symbol などである。Allegro CL 起動直後のデフォルトの状態でも、既に隠されている frame や symbol がある。また、:hide とは逆のコマンドとして :unhide がある。:hide, :unhide コマンドを利用して設定を変更すれば、:all nil モードで表示/非表示にする frame を変更できる。

2.1.3 :function t, :function nil モード

:function t モードでは、frame 中の関数が関数オブジェクトの形式で表示される。:all と同様に、次に :function の設定をし直すまでは同じ設定が続く。

この 2 つのモードの違いは、ひとつの frame を見ればわかるので、次の例では EVAL の frame だけを表示する。

:function t, function nil

```
[1] CL-USER(26): :zoom :function t
Evaluation stack: ;; EVAL の行以外は除去している
... 略 ...
 ->(FUNCALL #<Function EVAL> FOO)
... 略 ...
[1] CL-USER(27): :zoom :function nil
Evaluation stack:  ;; EVAL の行以外は除去している
... 略 ...
 ->(EVAL FOO)
... 略 ...
```

:function nil モードのときは、実行した関数がコンパイルされたものか、それとも interpret したものなのかはわからない。(関数の情報として関数名しか表示されないため。) :function t では、関数オブジェクトが表示されるので、コンパイルされているものについては #<Function FOO> という形式で表示され、interpret しただけのものだと #<Interpreted Function FOO> と表示される。このため、:function t モードでは、関数がコンパイルされたものなのか、interpret されたものなのかがわかる。

2.1.4 :specials t, :specials nil モード

:specials t モードでは、関数の実行中にバインドされたスペシャル変数を出力する。:specials に t または nil を指定したときの違いは一部の frame を見ればわかるので、以下では表示される一部分だけを表示する。

:specials t, :specials nil

```
[1] CL-USER(30): :zoom :specials t
Evaluation stack:
... 略 ...
 ->(EVAL FOO)
new special binding for SYS:*INTERPRETER-ENVIRONMENT*; old value was NIL
new special binding for EXCL::%FUNCTION-SPEC%; old value was NIL
   (TPL::READ-EVAL-PRINT-ONE-COMMAND NIL NIL)
new special binding for TPL::*TPL-TIME*; old value was NIL
new special binding for EXCL::*RESTART-CLUSTERS*; old value was ((#))
...略...
[1] CL-USER(31): :zoom :specials nil
Evaluation stack:
... 略 ...
 ->(EVAL FOO)
... 略 ...
```

2.1.5 :relative t, :relative nil モード

:relative t モードでは current frame 以外の各 frame が current frame から見て何番目の frame にあたるかを表示する。'u' は current frame の上である (つまり current frame より実行順序が新しい) ことを表し、'd' は current frame の下である (つまり current frame より実行順序が古い) ことを表す。

:relative t, :relative nil

```
[1] CL-USER(3): :zoom :relative t
Evaluation stack:

 1u:    (ERROR #<UNBOUND-VARIABLE @ #x4c4e19a>)
      ->(EVAL FOO)
 1d:    (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
 2d:    (TPL:START-INTERACTIVE-TOP-LEVEL
           #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
           #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
[1] CL-USER(4): :zoom :relative nil
Evaluation stack:

   (ERROR #<UNBOUND-VARIABLE @ #x4c4e19a>)
 ->(EVAL FOO)
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
[1] CL-USER(5):
```

2.1.6 :zoom の動作に関係する変数

ここで紹介する変数は、全て TOP-LEVEL パッケージで定義、export されている。

zoom-display
: 表示される stack frame の最大個数。:zoom コマンドの:count 引数のデフォルトに使われる。

zoom-print-circle
: :zoom 出力のときに cl:*print-circle* にこの値がバインドされる。よって t にしておくと、:zoom 出力時に構造が輪になっているオブジェクト (例えばリスト) の出力を省略した形で表示する。

zoom-print-level
: :zoom 出力のときに cl:*print-level* にこの値がバインドされる。つまり、:zoom 出力時に出力されるオブジェクトの階層を指定することができる。指定よりも下の階層は省略される。

zoom-print-length
: :zoom 出力のときに cl:*print-length* にこの値がバインドされる。つまり、:zoom 出力時に出力されるオブジェクトの各階層で表示される要素の個数を指定することができる。指定よりも要素の個数が多い場合は、その分が省略される。

zoom-print-special-bindings
: :specials モードのデフォルト値。:zoom 実行時に :specials t/nil を指定すればリセットされる。

zoom-show-newer-frames

nil でなければ、:zoom 出力の際に current frame より新しい frame を表示する。

auto-zoom

current frame を上または下の frame に変更したあと (:up, dn などの実行後) で自動的に :zoom を実行するかどうかを表す。

2.1.7 :zoom の使用例

未定義の関数を呼び出したときに発生したエラーのデバッグに、:zoom を使ってみる。次の関数 bar は、未定義の関数 foo を呼んでいる。

```
(defun bar (x) (foo 1 2 3 4 x))
```

次の例では、この bar をコンパイルした後に呼び出している。:zoom で確認すると、bar の frame の上に foo の frame が作られ、その中に渡した引数があることがわかる。

bar の実行

```
CL-USER(4): (bar 222)
Error: attempt to call 'FOO' which is an undefined function.
  [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.
 4: Return to Top Level (an "abort" restart).
 5: Abort entirely from this (lisp) process.
[1] CL-USER(5): :zoom
Evaluation stack:

   (ERROR #<UNDEFINED-FUNCTION @ #x4c8d642>)
 ->(FOO 1 2 ...)
   (BAR 222)
   [... EXCL::%EVAL ]
   (EVAL (BAR 222))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
[1] CL-USER(6): :current     ;; current frame を表示
(FOO 1 2 3 4 222)
```

この状態で foo を定義し、この frame から計算を再開することができる。

デバッグ例

```
[1] CL-USER(7): (defun foo (&rest x) x)
FOO
[1] CL-USER(8): :current
(FOO 1 2 3 4 222)
[1] CL-USER(9): :restart     ;; current frame から計算を再開
(1 2 3 4 222)
CL-USER(10):
```

次の関数 baz を定義し、わざと定義よりも少ない個数の引数を与えて呼び出した場合を見てみる。

```
(defun baz (a b c)
  (+ a b c))
```

この baz に引数を 2 つしか渡さずに呼び出してみる。以下は、baz をコンパイルした後で呼び出している。:current で current frame を表示すると、与えられなかった引数が :unknown と表現されていることがわかる。

引数の確認

```
CL-USER(84): (baz 1 2)
Error: BAZ got 2 args, wanted 3 args.
  [condition type: PROGRAM-ERROR]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] CL-USER(85): :zoom
Evaluation stack:

   (ERROR PROGRAM-ERROR :FORMAT-CONTROL ...)
 ->(BAZ 1 2 ...)
   [... EXCL::%EVAL ]
   (EVAL (BAZ 1 2))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<MULTIVALENT stream socket connected from localhost/3872 to
        localhost/3874 @ #x205457ba>
      TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP ...)
[1] CL-USER(86): :current
(BAZ 1 2 :UNKNOWN)
[1] CL-USER(87):
```

2.2 デバッガを終了するコマンド

エラーが発生していないときでも :zoom コマンドなどは使えるので、デバッガを完全に終了するということはできない。ただし、エラーが発生して break loop に入ったときに break を抜けることはできる。次のコマンドを利用する。

:pop

指定した数だけ、break を抜けることができる。数を省略した場合は 1 を指定した場合と同様に、1 つだけ break を抜けることができる。

:reset

全ての break を抜けることができる。

また、break したときは実行可能なアクションが表示される。表示されたアクションを実行するコマンドに次の :continue がある。

:continue

エラー発生時に表示される実行可能なアクションの番号をを指定すると、そのアクションが実行される。番号を指定しないときは 0 番目のアクションが実行される。

2.3 ポインタの移動コマンド

current frame であることを表すポインタを他の frame に移動させれば、移動先の frame を current frame にすることができる。移動コマンドは次のとおり。

:dn

ポインタを下方向に移動する。数を指定すると、その分だけ下方向に移動する。指定しない場合は 1 つだけ移動する。また、関数名を指定すると、指定した関数の frame が下方向に見つかればそこに移動する。

:up

:dn の逆でポインタを上方向に移動する。数を指定すると、その分移動する。指定しない場合は 1 つだけ移動する。:dn と同様に、関数名を指定すると、指定した関数の frame が上方向に見つかればそこに移動する。

:bottom

一番下の frame までポインタを移動する。

:top

一番上の frame までポインタを移動する。

:find

指定した関数名の frame を探し、ポインタを移動する。:up キーワードに t を指定すると、上方向に探すことができる。

2.4 :hide, :unhide

:hide, :unhide は指定した関数の frame などを表示/非表示にすることができる。ただし、:hide で隠すように指定しても、:all t モードでの :zoom 実行時は全ての frame が表示される。

:hide

引数を与えると、条件に合う frame などが表示されなくなる。引数なしの場合は、その時点で :hide により隠されているものが表示される。

:unhide

:hide の逆で、引数を与えると、条件に合う frame などが表示されるようになる。引数なしの場合は、:hide, :unhide で変更した設定を、デフォルトの状態に戻す。

:hide, :unhide のオプション引数には、関数名と frame-type の 2 種類がある。関数名の場合は、その関数の frame の表示/非表示を変更できる。frame-type も同様に、指定した種類の frame の表示/非表示を変更できる。指定できる frame-type の種類は次のとおり。

:binding

変数をバインドする frame (`let, let*`など)

:eval

S 式を評価する frame (`excl::%eval` など)

:interpreter

interpreter の内部メカニズムによって作られる frame

また、:hide, :unhide のキーワード引数には :package と :package-internals がある。:package キーワード引数では、指定したパッケージに含まれる全て関数の frame や全てのシンボルの表示/非表示を変更できる。:package-internals キーワード引数では、指定したパッケージの内部関数、内部シンボルの表示/非表示を変更できる。
あるパッケージの frame やシンボルは非表示にしたいが、そのパッケージの特定の関数の frame のみ表示したい場合は、:hide でパッケージを非表示にしたあとで、:unhide でその関数のみ表示するように設定すればいい。

:hide, :unhide

```
CL-USER(17): :unhide
Reverting hidden objects to initial state.
CL-USER(18): :hide
hidden packages: LEP DEBUGGER
hidden packages internals: TOP-LEVEL CLOS SYSTEM MULTIPROCESSING EXCL
excepted functions: SYSTEM::..LISP-BREAKPOINT-RUNTIME-HANDLER
hidden functions: BLOCK APPLY SYSTEM::INVALID-RT-FUNCTION
hidden frames: :INTERPRETER :EVAL :INTERNAL
CL-USER(19): (break)
Break: call to the ‘break’ function.

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(20): :zoom :moderate t
Evaluation stack:

   (BREAK)
   [... EXCL::%EVAL ]
 ->(EVAL (BREAK))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
[1c] CL-USER(21): :hide :package tpl
[1c] CL-USER(22): :zoom
Evaluation stack:

   (BREAK)
   [... EXCL::%EVAL ]
 ->(EVAL (BREAK))
[1c] CL-USER(23):
```

2.5 frame に関するコマンド

次にあげるコマンドは、current frame や current frame で呼ばれている関数を表示する。

:current

current frame だけを表示する。また、cl:* に表示した frame をバインドする。ただし、このコマンドで表示され、cl:* にバインドされる frame は、:zoom 実行時の表示モードに関係なく、(関数名 引数 1 ...) という形式である。(例えば、:brief t かつ :function t モードで :zoom を実行すると、各 frame は関数オブジェクトだけが表示される。しかし、:current で表示される frame は、常に (関数名 引数 1 ...) という形である。)

:function

current frame の関数オブジェクトを表示し、cl:* にバインドする。ただし、内部関数 (flet などで定義したもの) で関数オブジェクトを持たないものについては nil を表示する。

:frame

current frame のローカル変数や、ローカル変数がバインドされる部分のソースコードなどを含めた情報を表示する。

次の例で、これらのコマンドの動作を確認する。また、特に :frame コマンドで表示されるソースコードとローカル変数の情報を見るために、関数 foo はコンパイルせずに interpret するだけにする。interpret するだけだと、foo の frame のほかに foo の定義内にある let の frame が、foo の frame の上に作られる。(コンパイルすると作られない。)
さらに、:hide :binding を実行することで、foo の上に作られる let の frame が隠される。この状態で foo の frame について :frame コマンドを実行すると、foo のソースコードや、ローカル変数の情報が表示される。:hide :binding を実行しないと、foo の frame に対して :frame コマンドを実行しても、これらの情報が表示されない。(一行目の Expression: の行だけは表示される。)

:current, :function, :frame

```
CL-USER(150): (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
CL-USER(151): :hide :binding ;; let などの frame を非表示にしておく。
CL-USER(152): (foo 10)
Break: foo: 12

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(153): :zoom
Evaluation stack:

   (BREAK "foo: ~a" 12)
 ->(FOO 10)
   (EVAL (FOO 10))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<MULTIVALENT stream socket connected from localhost/3872 to
        localhost/3874 @ #x205457ba>
      TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP ...)
[1c] CL-USER(154): :current
(FOO 10)
[1c] CL-USER(155): :function
#<Interpreted Function FOO>
[1c] CL-USER(156): *
#<Interpreted Function FOO>
[1c] CL-USER(157): :frame  ;; :hide :binding を実行していないと、
Expression: (FOO 10)       ;; Source code などの情報が表示されない
  Source code: (LET ((A (1+ X))) ..)
  Name: X (:LEXICAL): 10  Source code: (LET ..)
  Name: X (:LEXICAL): 10  Source code: (PROGN ..)
  Name: X (:LEXICAL): 10  Name: A (:LEXICAL): 11
[1c] CL-USER(158):
```

2.6 ローカル変数に関するコマンド

この節では frame 中のローカル変数を操作する方法を説明する。ただし、ローカル変数の名前の情報を利用するには、*load-local-names-info* の値を non-nil にして関数をコンパイルしておく必要がある。

:local

current frame 中のローカル変数の値を表示する。

:set-local

引数として指定したローカル変数に評価した値をセットする。

:evalmode

指定できる引数は :context キーワード引数のみである。:context キーワードに t を与えて実行すると、デバッグの間だけ setq や setf でローカル変数の値をセットすることができるようになる。つまり、トップレベルでローカル変数の名前を指定して、frame 中の変数に値をセットすることができる。:context キーワード引数を与えない場合は、その時点で context の設定 (t/nil) に応じたメッセージが表示される。

これらのコマンドを試してみる。次の関数をコンパイルしておく。

```
(defun lv-fun (arg)
  (let ((a (+ arg 1))
        (b (+ arg 2))
        (c 3))
    (break "break")
    (setq c (+ a b c arg))
    (print `(:a ,a :b ,b :c ,c))))
```

lv-fun を引数 10 を指定して呼び出す。

:local, :evalmode, :set-local **使用例**

```
CL-USER(2): (lv-fun 10)
Break: break

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(3): :local
Compiled lexical environment:
0(REQUIRED): ARG: 10
1(LOCAL): A: 11
2(LOCAL): B: 12
3(LOCAL): C: 3
[1c] CL-USER(4): :set-local c 13 ;; c の値を変更
[1c] CL-USER(5): :local
Compiled lexical environment:
0(REQUIRED): ARG: 10
1(LOCAL): A: 11
2(LOCAL): B: 12
3(LOCAL): C: 13                  ;; c の値が変わっていることを確認
[1c] CL-USER(6): :evalmode :context t ;; evalmode 開始
[1c] CL-USER(7): (setq c 130)    ;; c の値を変更
130
[1c] CL-USER(8): :local
Compiled lexical environment:
0(REQUIRED): ARG: 10
1(LOCAL): A: 11
2(LOCAL): B: 12
3(LOCAL): C: 130                 ;; c の値が変わっていることを確認
[1c] CL-USER(9): :cont

(:A 11 :B 12 :C 163)
(:A 11 :B 12 :C 163)
CL-USER(10):
```

2.7 自動的にバックトレースを取得する

バッチモードでプログラムを実行しているときなど、ユーザが自分で :zoom **コマンドを実行できない場合はプログラムで自動的に** :zoom **で生成されるバックトレースを取得できるようにしておくと便利である。**

Allegro CL 8.1 では、zoom 関数と with-auto-zoom-and-exit マクロがあり、これらを使うとエラー発生時に :zoom を実行し、結果を stream やファイルに出力することができる。zoom 関数と with-auto-zoom-and-exit マクロの定義は、Allegro CL のインストールディレクトリ以下の src/autozoom.cl にある。

autozoom モジュールを使うには、次のようにしてモジュールをロードする必要がある。

```
(require :autozoom)
```

zoom 関数、with-auto-zoom-and-exit マクロは :zoom コマンドの wrapper であり、引数で与えた出力先 (stream や pathname) に結果を出力する。また、 :zoom 実行時の :count や :all を指定することができる。
zoom 関数は、次のように handler-bind のハンドラーの中で呼び出すことで、エラー発生時の :zoom を取得することができる。

```
(handler-bind
   ((error (lambda (condition)
           ;; write info about CONDITION to a log file...
             (format *log-stream* \"Error in app: ~a~%\" condition)

           ;; send a zoom to the log file, too
           (top-level.debug:zoom *log-stream*))))
 (application-init-function))
```

アプリケーションの実行中にエラーが発生した場合、アプリケーションを停止したままにしておくより、ログを出力して終了してしまった方が、ユーザにとっては親切である。with-auto-zoom-and-exit マクロは名前のとおり、:zoom を実行した後にプログラムを終了させることができる。このマクロも body 部を上記のような handler-bind で囲っていて、ハンドラーの中で :zoom を実行し、さらに exit でプログラムを終了する。

2.8 :return, :restart コマンド

:return と :restart はどちらも current frame から計算を再開するときに使う。

:return は与えられた引数を評価し、その値を current frame の返り値とする。この場合、current frame 自体は再評価されない。例えば、

```
:return (exp 3)
```

とすると、current frame の結果を 8 (= (exp 3)) とし、続けて残りの frame を計算していく。引数を与えない場合は current frame の結果が nil となり、同様に残りの計算をしていく。

:restart の引数は nil またはリストである。リストの場合は第１要素が関数名で、残りの要素がその引数である。:restart の場合は、current frame を引数であるリストに置き換え、置き換えた current frame の計算から再開する。nil を引数として与える場合や、引数を与えない場合は、current frame がそのまま指定されたとして動作する。

次の例で:return, :restart の動作の違いを見る。次の関数 foo, bar を定義し、コンパイルしておく。

```
(defun foo (l)
  (print "foo start")
  `(:bar ,(bar l)))
(defun bar (l)
  (print "bar start")
  (let ((a (first l))
        (b (second l)))
    (print `(:a ,a :b ,b))
    (+ a b)))
```

foo を呼び出す。１回目は問題ないが、２回目はエラーが起こる。

```
CL-USER(267): (foo '(1 2)) ;; 1 回目

"foo start"
"bar start"
(:A 1 :B 2)
(:BAR 3)
CL-USER(268): (foo 1)      ;; 2 回目

"foo start"
"bar start"
Error: Attempt to take the car of 1 which is not listp.
  [condition type: TYPE-ERROR]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] CL-USER(269): :zoom
Evaluation stack:

   (ERROR TYPE-ERROR :DATUM ...)
 ->(BAR 1)
   (FOO 1)
   [... EXCL::%EVAL ]
   (EVAL (FOO 1))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
```

```
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
```

ここで、:return 3 **を実行すると、エラーが起こった** bar **の** frame **から** 3 **が返されたことがわかる。**

```
[1] CL-USER(270): :return 3
(:BAR 3)
```

同じエラーを発生させ、:restart '(bar (1 2)) **を実行する。すると、**:restart **で指定した** (bar (1 2)) **が** current frame **に置き換わり、実行されたことがわかる。**

```
[1] CL-USER(272): :restart '(bar (1 2))

"bar start"
(:A 1 :B 2)
(:BAR 3)
CL-USER(273):
```

上記の例で :return 3 **と** :restart '(bar (1 2)) **で計算を再開すると、同じ結果** (:BAR 3) **を返すので、**:return **と** :restart **の違いを** current frame **を再評価するか否かだけと勘違いしやすいが、実際にはそうではない。次の例を見てみる。**

```
(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*)))
```

上記の関数 foo **は引数** x **の値が** 5 **であれば** break **する。**bar **は** (foo 5) **を実行するので、**bar **を呼び出すと** break **する。**

```
CL-USER(278): (bar)
Break: sorry, bad number

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
```

```
[1c] CL-USER(279): :zoom
Evaluation stack:

   (BREAK "sorry, bad number")
 ->(FOO 5)
   (BAR)
   [... EXCL::%EVAL ]
   (EVAL (BAR))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
```

この段階で、:return, :restart **をそれぞれ次のように実行すると、結果が違うことがわかる。**

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

実は、:restart **はエラーが発生した** foo **が呼ばれる前まで計算の状態を戻し、そこから** `(foo 6)` **の評価を再開する。それに対し、**:return **は状態を戻さないまま** `(foo 6)` **を評価する。このため、***special* **のインクリメントの回数が**:return **のほうが１回多い。**

このような副作用がないプログラムでは :return **と** :restart **の違いを意識せずに使用できるが、副作用がある場合は** :restart **を使用するべきである。**

3 ローカル変数とデバッガ

3.1 ローカル変数とは

ローカル変数とは、関数の引数と、関数中で let **などで作られた変数のことである。**

3.2 ローカル変数とデバッガ

エラーが発生した時点でのローカル変数とその値の情報がわかると、デバッグがしやすい。Allegro CL **にはコンパイルされたコード中のローカル変数の値を調べて変更する方法がある。ただしこの機能を利用すると、より多くのスペースコストが必要となる。(デバッガに余分な情報を提供するために、コンパイルされたコードに余分な情報が必要となるため。) プログラムをファイルに保存して、その** fasl **を読み込む場合は、***load-local-names-info* **を** non-nil **にした状態でプログラムファイルをコンパイルし、その** fasl **をロードした場合のみ、ローカル変数の名前の情報がロードされる。**

ローカル変数の保存には、レジスタまたはスタックを使う。ハードウェアに充分な個数のレジスタがあればローカル変数はレジスタに保存され、そうでなければスタックに置かれる。このように、レジスタの個数とローカル変数の個数によって、レジスタとスタックのどちらを使うかが決ま

る。また、使用できるレジスタの個数はプロセッサによって異なる。例えば Sparc では 8 個、IBM RS/6000 では 10 個のレジスタが使用可能である。

3.3 レジスタとスタックの違い

レジスタを使う利点は、アクセスがスタックよりも早いことにある。つまり、スタックを使うよりも実行速度があがる。ただし、コンパイル時に行われる最適化によっては、デバッグする時点でレジスタにもスタックにも値が保存されない変数が発生することがあり、デバッグが難しくなるという問題もある。
実行速度をあげるために、レジスタを使える場合はレジスタをできるだけ利用するようになっている。レジスタの有効利用の方法として、ローカル変数が無効になると、そのローカル変数のために使っていたレジスタを他のローカル変数のために使えるように、ひとつのレジスタを複数のローカル変数で共有するという方法を取っている。ひとつのレジスタを共有する複数のローカル変数が同時に有効になることはなく、その時点で有効なひとつのローカル変数の値だけが保存される。
スタックの場合は、同じスタックを使いまわすことはしない。つまり、各ローカル変数ごとに別のスタックが作られることになり、場合によってはスタックが巨大になることがある。
残りの節では、Sparc マシンを利用して、ローカル変数とレジスタの関係を見ていく。

3.4 末尾呼び出しの関数とローカル変数

次の foo は末尾の list-length の返り値をそのまま返す。つまり、list-length を実行するときは foo のそれより前の情報は必要なくなる。

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

この foo のように末尾で呼び出した関数の返り値をそのまま自身の返り値とするような関数を、compiler:tail-call-non-self-merge-switch が t の状態でコンパイルすると、list-length の実行時に、必要なくなった foo の frame は runtime stack からクリアされる。よって、list-length の実行でエラーになっても、foo の frame はデバッグ時に表示されないので、a, b の値を確認することができなくなる。

3.5 ローカル変数の有効範囲

関数のコンパイル時、コンパイラはあるローカル変数がいつ初めて使われるか (通常は値がバインドされたとき) と、最後に使われるのはいつなのかを決定する。この区間をローカル変数の有効範囲と呼ぶ。

次の関数 loc-fun では主要なところで break を入れているので、各場所でのスタックの利用状況を調査しやすい。
また、コメントに有効範囲を記述してある。

```
(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
                (break "break3")
                (print (+ x y))))
```

コメントを見ると、let **でローカル変数** a, b, c **を作っているのに、この** let **が終る前に** a, b **が無効になったというコメントが入っている。これは、コンパイラは最後にローカル変数が使われた位置までを有効範囲としているためで、**let **の途中でも** a, b **が無効になるからである。**

上記の loc-fun **をコンパイルして、ローカル変数がどのようにレジスタを共有しているか見てみる。次のとおり、ローカル変数はスロット** 9 **に保存されている。**

```
CL-USER(2): (inspect #'loc-fun)
A NEW #<Function LOC-FUN>
  lambda-list: (ARG)
   0 excl-type ----> Bit field: #x08
   1 flags --------> Bit field: #x88
   2 start --------> Bit field: #x04ccb56c
   3 hash ---------> Bit field: #x00008213
   4 symdef -------> The symbol LOC-FUN
   5 code ---------> short simple CODE vector (192) = #(40419 49048 32928 ...)
   6 formals ------> (ARG), a proper list with 1 element
   7 cframe-size --> fixnum 0 [#x00000000]
   8 immed-args ---> fixnum 0 [#x00000000]
   9 locals -------> simple T vector (4) = #(8 (X C) (Y D A) (E B))
  10 <constant> ---> A simple-string (6) "break1"
  11 <constant> ---> The symbol BREAK
  12 <constant> ---> A simple-string (6) "break2"
  13 <constant> ---> The symbol PRINT
  14 <constant> ---> A simple-string (6) "break3"
```

スロット 9 **についてさらに詳しく調べると、変数がレジスタを共有していることがわかる。**

```
[1i] CL-USER(3): :i 9
A NEW simple T vector (4) @ #x458001a
   0-> fixnum 8 [#x00000020]
   1-> (X C), a proper list with 2 elements
   2-> (Y D A), a proper list with 3 elements
   3-> (E B), a proper list with 2 elements
[1i] CL-USER(4): :res
```

この例では 3 **つのレジスタが使われ、**1 **つ目を** x, c**、**2 **つ目を** y, d, a, 3 **つ目を** e, b **が共有していることがわかる。ただし、**loc-fun **の実行時にこれらのレジスタを見ても、レジスタにはその時点で有効なローカル変数だけ、値が保存されているため、有効でないローカル変数の値は基本的に見ることができない。**

3.6 コンパイル方法による違い

コンパイラの safety, space, speed, debug **の設定はコンパイル時の様々な** switch **変数の値に作用し、その値が** t(**オン**) **となるか** nil(**オフ**) **となるかでコンパイルされたコードに保存される情報が違ってくる。**
ローカル変数の情報に関する switch **変数と、オンになる条件、オンになったときの効果は以下のとおりである。**(**これらの変数は** compiler **パッケージで定義されている。**)

save-arglist-switch
　　条件: `debug > 0`
　　効果: **引数の名前を保存する**

save-local-names-switch
　　条件: `debug > 1`
　　効果: **ローカル変数の名前を保存する**

save-local-scopes-switch
　　条件: `debug = 3`
　　効果: **ローカル変数の有効範囲に関する情報を保存する**

これらの変数に直接 t **や** nil **をバインドすることもでき、その場合はコンパイラの** safety, space, speed, debug **の設定に関係なく、常にオンまたはオフにできる。**
safety, space, speed, debug **の設定や、**switch **変数は、**(`explain-compiler-settings`)**で表示して確認することができる。**
safety, space, speed, debug **の設定を変更する場合は、次のように** proclaim **を利用する。**

```
(proclaim '(optimize (speed 1) (safety 1) (space 1) (debug 3)))
```

また、ローカル変数の名前や有効範囲の情報とは直接の関係はないが、ローカル変数がレジスタを共有するか否かを決める switch **変数がある。**

internal-optimize-switch

条件: `speed > 2 or debug < 3`

効果: **最適化により、ローカル変数がレジスタを共有できる**

実は、前節で関数 loc-fun **をコンパイルしたときの設定は、**speed=1 **かつ** debug=2 **だった。このため** internal-optimize-switch **がオン** (t) **となり、ローカル変数がレジスタを共有するようになっていた。**
この節では、proclaim **で** debug **の設定を変更しながら** loc-fun **をコンパイルし、保存される情報の違いを見ていく。本来なら** debug **の設定を変更することで** internal-optimize-switch **のオン/オフも切り替わるのだが、今回は保存されるローカル変数の情報の違いだけを見るために、**internal-optimize-switch **に** t **をバインドし、**debug **の設定に関係なく常にローカル変数がレジスタを共有するようにしておく。**

3.6.1 debug=0

引数の名前もローカル変数の名前も保存されない。また、有効範囲も保存されず、どのレジスタに有効な変数があるかわからないので、使用可能なすべてのレジスタが表示される。この例では Sparc **マシンを利用しているので、**8 **個のレジスタが表示される。**

```
CL-USER(68): (loc-fun 22)
Break: break1

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(69): :local
Compiled lexical environment:
0(EXTRA-ARG): NIL: 22
1(LOCAL): EXCL::LOCAL-0: 3
2(LOCAL): EXCL::LOCAL-1: 23
3(LOCAL): EXCL::LOCAL-2: 24
4(LOCAL): EXCL::LOCAL-3: NIL
5(LOCAL): EXCL::LOCAL-4: 38
6(LOCAL): EXCL::LOCAL-5: 16809641
7(LOCAL): EXCL::LOCAL-6: 352323838
8(LOCAL): EXCL::LOCAL-7: 0
[1c] CL-USER(70):
```

loc-fun **に引数** 22 **を与えているので、**loc-fun **の定義と合わせて考えると、**a **が** 2 **番目のレジスタに、**b **が** 3 **番目のレジスタにあることがわかる。**c **は** 1 **番目である。ただし、ここに表示されている情報からは、何が起こっているのかわかりづらい。また、残りのレジスタには、ゴミの値や、以前そのレジスタを使ったときに入っていた値が残っているだけである。**

3.6.2 debug=1

debug=1 **では、引数の名前は保存されるが、その他のローカル変数の名前や有効範囲の情報は保存されない。**

```
CL-USER(6): (loc-fun 22)
Break: break1

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(7): :local
Compiled lexical environment:
0(REQUIRED): ARG: 22
1(LOCAL): EXCL::LOCAL-0: 3
2(LOCAL): EXCL::LOCAL-1: 23
3(LOCAL): EXCL::LOCAL-2: 24
4(LOCAL): EXCL::LOCAL-3: NIL
5(LOCAL): EXCL::LOCAL-4: 38
6(LOCAL): EXCL::LOCAL-5: 16809641
7(LOCAL): EXCL::LOCAL-6: 352323838
8(LOCAL): EXCL::LOCAL-7: 0
[1c] CL-USER(8): :res
```

3.6.3 debug=2

ローカル変数の名前が保存される。ただし、有効範囲の情報が保存されない。

```
CL-USER(10): (loc-fun 22)
Break: break1

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(11): :local
Compiled lexical environment:
0(REQUIRED): ARG: 22
1(LOCAL): (X C): 3
2(LOCAL): (Y D A): 23
3(LOCAL): (E B): 24
[1c] CL-USER(12): :res
```

今度はどのローカル変数がどのレジスタ使うかわかるので、使われないレジスタが表示されなくなる。
ただし、有効範囲の情報が保存されていないため、break した時点でどのローカル変数が有効であるかはわからない。この情報からは、どのローカル変数がどのレジスタを共有しているかと、ローカル変数によって使用されているレジスタにある値がわかる。

3.6.4 debug=3

ローカル変数の名前と有効範囲が保存される。

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

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(23): :local
Compiled lexical environment:
0(REQUIRED): ARG: 22
1(LOCAL): C: 3
2(LOCAL): A: 23
3(LOCAL): B: 24
[1c] CL-USER(24):
```

有効範囲も保存されているので、break した時点でどのローカル変数が有効なのかわかり、有効なローカル変数のみを表示することができている。つまり、まだ有効になっていないものや、一度は有効になったが既に有効でなくなったものについては表示されない。
さらに次の break まで計算を続ける。

```
[1c] CL-USER(24): :cont
Break: break2

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(25): :local
Compiled lexical environment:
0(REQUIRED): ARG: 22
1(LOCAL): C: 72
2(LOCAL): D: 4
3(LOCAL): E: 5
[1c] CL-USER(26):
```

a と b が有効ではなくなったため表示されなくなり、そのかわりに d と e が有効になったため表示される。さらに次の break まで計算を進める。

```
[1c] CL-USER(26): :cont

1440
Break: break3

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(27): :local
Compiled lexical environment:
0(REQUIRED): ARG: 22
1(LOCAL): X: 10
2(LOCAL): Y: 11
[1c] CL-USER(28):
```

3 つ目のレジスタを共有していたローカル変数 e, b はどちらも有効ではなくなっているので、3 つ目のレジスタが表示されなくなる。
このように普通は debug=3 でコンパイルすると有効でなくなったローカル変数は表示されないが、変数 tpl::*print-dead-locals* を t にすると表示することができる。

```
[1c] CL-USER(28): (setq tpl::*print-dead-locals* t)
T
[1c] CL-USER(29): :local
Compiled lexical environment:
0(REQUIRED): ARG: 22
1(LOCAL): X: 10
2(LOCAL): Y: 11
3(LOCAL): (:DEAD (E B)): 5
[1c] CL-USER(30):
```

先程は表示されなかった 3 つ目のレジスタが表示される。(:DEAD (E B)) は e と b がこのレジスタを共有しているが、この時点ではどちらも有効でないことを意味する。どちらが後で有効になったかなどはわからないが、有効だったときにレジスタに保存されていた値である 5 を表示している。

3.6.5 デフォルトの設定が debug=3 でない理由

前項までで見たように、debug=3 でコンパイルした関数はローカル変数の名前とその有効範囲の情報が保存されるため、デバッグがしやすい。

にも関わらず、デフォルトの設定では debug=3 になっていない。これは、fasl ファイルを作成する際にこれらの情報を保存すると、保存しない場合と比べてファイルサイズが大きくなるためである。(例えば loc-fun をファイルに記述してコンパイルした場合、debug=1 では 1874 バイト, debug=3 では 2081 バイトだった。)
開発中は debug=3 にしておくと便利だが、製品版のアプリケーションを作成する場合などは無駄にファイルが大きくなるのを避けるために、debug=3 にしないほうがよい。

4 Ghost frame

4.1 Ghost frame とは

通常 Lisp では、関数が関数を呼んでいくと、スタック上に frame が作られていく。ただし、コードをコンパイルしたときに最適化が行われると、本来ならスタック上に作られ、表示されるはずの frame がスタック上に作られないことがある。このように最適化されたコードはスタック上に作られる frame 数が少なく、最適化されていない場合より動作が速いが、デバッグをするという点では難しくなる。
デバッガは失われた frame を可能であれば [ ] 括弧で括って表示する。このような frame を Ghost frame と呼ぶ。

```
[1] CL-USER(18): :zoom
Evaluation stack:

   (ERROR TYPE-ERROR :DATUM ...)
 ->(LIST-LENGTH 1)
   [... FOO ]
   (START-FUN 1)
   [... EXCL::%EVAL ]
   (EVAL (START-FUN 1))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
```

Ghost frame である [... FOO] は、関数 foo が呼び出されたが、その frame がなくなったことを表している。また、省略記号 ... は、さらに他の関数の呼び出しの frame もなくなっているが、その関数が何かを特定できないことを表す。

Ghost frame はスタック中に実際にある frame ではなく、current frame にはできない。また、Ghost frame から restart や return コマンドを実行することもできない。

4.2 Ghost frame が作られる条件

次の 4 つの関数は、前節で見たエラーを発生したものである。

```
(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))
```

これらの関数をコンパイルせず (つまり interpret するだけ)、(start-fun 1) を実行すると、次のようなスタックが表示される。

```
[1] CL-USER(25): :zoom
Evaluation stack:

   (ERROR TYPE-ERROR :DATUM ...)
 ->(LIST-LENGTH 1)
   (BAZ 1)
   (BAR 1)
   (FOO 1)
   (START-FUN 1)
   (EVAL (START-FUN 1))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
[1] CL-USER(26):
```

コンパイラが行う最適化によってスタックから消される frame は、末尾で他の関数を呼び出している (non-self tail merging) ものであり、safety, space, speed, debug の設定により compilet:tail-call-non-self-merge-switch が true となる状態コンパイルされると、最適化によって消去される。その結果が、前節で見たスタックである。

```
[1] CL-USER(18): :zoom
Evaluation stack:

   (ERROR TYPE-ERROR :DATUM ...)
 ->(LIST-LENGTH 1)
   [... FOO ]
   (START-FUN 1)
   [... EXCL::%EVAL ]
   (EVAL (START-FUN 1))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
```

```
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
```

また、Ghost frame **を生成するには、逆アセンブラがロードされている必要がある。**Allegro CL **の逆アセンブラは必ず含まれているのではなく、必要に応じてロードすることになっている。普通は** disassemble **が呼ばれたときにロードされる。**
逆アセンブラをロードすると、*modules* **変数のリストに**"DISASM" **という文字列が含まれる。**

4.3 Ghost frame とデバッガ

前節の例では、foo, bar, baz **の** frame **がスタックから消えてしまってるにも関わらず、**Ghost frame **が表示される。**
これは、スタック上の情報だけでは Ghost frame **を表示するには不十分であるが、デバッガは逆アセンブラを使って** Ghost frame **を生成できるからである。**

エラーが起こった list-length **の実行が終ると、**start-fun **に制御が戻ってくる。デバッガはこの** start-fun **を逆アセンブルし、その結果を見て、**foo **の呼び出し直後が制御が戻る位置だとわかる。**

```
CL-USER(23): (disassemble 'start-fun)
;; disassembly of #<Function START-FUN>
;; formals: X
;; constant vector:
0: FOO

;; code start: #x479067c:
   0: 9de3bf98     save %o6, #x-68, %o6
   4: 80a0e001     cmp %g3, #x1
   8: 93d02010     tne %icc,%g0, #x10
  12: 81100001     taddcctv %g0, %g1, %g0
  16: c4076022     ld [%i5 + 34], %g2 ; FOO
  20: 90100018     mov %i0, %o0
  24: 9fc1200b     jmpl %g4 + 11, %o7
  28: 86182001     xor %g0, #x1, %g3
  32: 90100004     mov %g4, %o0
  36: 86102001     mov #x1, %g3
  40: 81c7e008     jmp %i7 + 8
  44: 91ea0000     restore %o0, %g0, %o0
CL-USER(24):
```

これにより、foo **が呼ばれたことがわかる。**foo **が呼ばれたことと、**foo **の** frame **がスタックにないことから、デバッガは** foo **の** Ghost frame **を生成する。**

さらにデバッガは foo を逆アセンブルし、その結果 foo が list-length を直接呼んでいないことから、少なくともあとひとつは他の関数が foo に呼ばれ、それが list-length を呼んでいることがわかる。これにより、デバッガは省略記号 ... を Ghost frame の中に表示し、foo 以外にも最適化により失われた frame があることを示す。

4.4 Ghost frame や省略記号が表示されない場合

Ghost frame が表示されないからといって、消去された frame がないとはいえない。その例を見るために、start-fun を次のように定義し直してみる。

```
(defun start-fun (x) (foo x) nil)
;; => 定義しなおしたもの
(defun start-fun (x) (funcall 'foo x) nil)
```

`(list-length 1)` を実行し、スタックを確認すると、foo の Ghost frame が表示されていない。

```
[1] CL-USER(30): :zoom
Evaluation stack:

   (ERROR TYPE-ERROR :DATUM ...)
 ->(LIST-LENGTH 1)
   (START-FUN 1)
   [... EXCL::%EVAL ]
   (EVAL (START-FUN 1))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
```

このように foo の frame がないにも関わらず、Ghost frame が表示されないことがある。

同様に、frame がないにも関わらず、Ghost frame 中に省略記号 ... が表示されないことがある。次のように foo を再定義してコンパイルし、同様に `(start-fun 1)` を実行してスタックを表示してみると、Ghost frame `[ FOO ]` が表示されるが、省略記号 ... が表示されないことがわかる。

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

```
[1] CL-USER(7): :zoom
Evaluation stack:
```

```
   (ERROR TYPE-ERROR :DATUM ...)
 ->(LIST-LENGTH 1)
   [ FOO ]
   (START-FUN 1)
   [... EXCL::%EVAL ]
   (EVAL (START-FUN 1))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x4115742>
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
[1] CL-USER(8):
```

これは前の定義とは違い、foo **の中で直接** list-length **を呼び出しているので、**disassemble **してもエラーになった** list-length **が** foo **で直接呼ばれたものなのか、**foo **が呼んだ別の関数から呼ばれたものなのかわからず、省略記号を入れることができないためである。**

これらのことから、Ghost frame **や省略記号が表示される場合は** frame **が消去されたという正しい情報が表示されていると言えるが、表示されないからと言って、消去された** frame **がないとは言えないことがわかる。**

5 tracer

tracer **を利用すると、関数のトレースができる。関数をトレースすると、関数の開始時と終了時にメッセージが表示される。メッセージには開始時には引数が、終了時には返り値が含まれる。関数をトレースすると、何度も呼ばれる関数の場合、何回目でエラーが起こったかがわかる。また、複数の関数を** trace **しておけば、どの関数から呼ばれたときにエラーになったかがわかる。さらに、引数の値が何のときにエラーになったかもわかる。**
tracer **はトップレベルで** :trace **コマンドでトレースする関数を指定すると起動する。関数を指定しない場合は、現在トレース中の関数を表示する。**:untrace **ではトレースを停止することができる。関数を指定した場合は、指定した関数だけトレースを停止し、指定しなかった場合は、全てのトレースを停止する。**

また、マクロ trace, untrace **もトップレベルコマンド** :trace, :untrace **と同様に使うことができる。**

次は、階乗を計算する関数 fact **を定義し、トレースした結果である。**

```
CL-USER(11): (defun fact (x)
               (cond ((= 1 x) 1)
                     (t (* x (fact (1- x))))))
FACT
CL-USER(12): (trace fact)
```

```
(FACT)
CL-USER(13): (fact 5)
 0[2]: (FACT 5)
   1[2]: (FACT 4)
     2[2]: (FACT 3)
       3[2]: (FACT 2)
         4[2]: (FACT 1)
         4[2]: returned 1
       3[2]: returned 2
     2[2]: returned 6
   1[2]: returned 24
 0[2]: returned 120
120
CL-USER(14):
```

このように tracer **による出力は行の先頭に階層を表す数字が付き、インデントされているため、ユーザに読みやすいようになっている。**

階層の次に [] **括弧つきで表示される数字は**:process **コマンドで表示されるプロセス番号である**(Bix **カラムの値**)**。**

現在のパッケージからアクセスできない関数についてはパッケージ名をつけて出力される。

関数 ftrace **は、関数オブジェクトを指定してトレースを行う。**funtrace **も同様に、関数オブジェクトを指定してトレースの取り消しを行う。**

5.1 :trace のオプション

:condition

引数: expr

expr **が** nil **以外を返すなら、関数をトレースする**

:break-before

引数: val

関数の開始直前に val **を評価する。**val **が** nil **でなければ** break **する。**val **が** nil **なら** break **せずに実行を継続する。このオプションを** :inside **や** :not-inside **と組み合わせて使うと、**:inside **または** :not-inside **の状態を満たした場合だけ** break **する。**

:break-after

引数: val

引数 val **を関数の終了直後に評価し、**nil **でなければ、**break **する。**nil **なら** break **せずに実行を継続する。このオプションを** :inside **や** :not-inside **と組み合わせて使うと、**:inside **または** :not-inside **の状態を満たした場合だけ** break **する。**

:break-all

引数: val

val **は関数の開始直前と終了直後に評価され、その値が** nil **でなければ** break **する。このオ**

プションを :inside や :not-inside と組み合わせて使うと、:inside または :not-inside の状態を満たした場合だけ break する。

:inside

引数: func

指定した func の呼び出し中に関数をトレースする。例えば、`(trace (foo :inside bar))` は bar の呼び出し中にだけ、 foo をトレースする。

:inside で指定する func は関数のリストでもよい。先頭には cl:and または cl:or を指定することができる。リストの場合、`(foo1 bar2)` または `(cl:and foo1 bar2)` を指定すると foo1 と bar2 の両方が実行中である場合のみ、トレースが行われる。`(cl:or foo1 bar2)` の場合、foo1 と bar2 の少なくともどちらか一方が実行中の場合にトレースが行われる。

次で述べる :not-inside と組み合わせても使える。その場合は両方の条件を満たした場合にトレースが行われる。

例えば、`(trace (deeper :inside deep))` は deep の呼び出し中にだけ、deeper をトレースする。これに対して `(trace (deeper :inside deep :not-inside (cl:or foo1 bar2)))` は、deep の呼び出し中かつ foo1 または bar2 の呼び出し中でない場合に deeper をトレースする。

:not-inside

引数: func

:inside の逆で、指定した func の呼び出し中は関数をトレースしない。func の指定方法も同様で、cl:and または cl:or で始まるリストでもよい。`(cl:and foo1 bar2)` または `(foo1 bar2)` を指定する場合、foo1, bar2 が同時に実行中である場合はトレースを行わない。`(cl:or foo1 bar2)` を指定する場合、foo1, bar2 のいずれかが実行中である場合はトレースを行わない。

例えば、`(trace (deeper :not-inside deep))` は deeper を deep の実行中以外の場合にトレースする。`(trace (deeper :not-inside deep :inside (cl:or foo1 bar2)))` は、deeper を deep の呼び出し中ではなく、かつ foo1 または bar2 の呼び出し中にトレースする。

:print-before

引数: expr

expr はオブジェクトまたはオブジェクトのリストであり、その評価結果が関数の実行前に出力される。

:print-after

引数: expr

expr はオブジェクトまたはオブジェクトのリストである。その評価結果は関数が終了した直後に出力される。

:print-all

引数: expr

expr はオブジェクトまたはオブジェクトのリストである。その評価結果は関数の開始直前と終了直後に出力される。

上記のとおり、:inside では、指定した関数の呼び出し中にトレースを行う。:not-inside ではその逆で、指定した関数の呼び出し中はトレースを行わない。ただし、特殊な場合には期待した動作を

しない場合がある。

そのひとつの原因として、関数の末尾呼び出しがある。bar が baz を、baz が foo をそれぞれ末尾で呼んだ場合、foo は baz の中ではなく、bar の中で呼ばれたように扱われてしまう (tail merge)。このため、baz の実行中に foo をトレースするつもりでも、トレースされないことがある。

:inside や :not-inside にメソッドを指定することができる。また、generic function を :inside や :not-inside に指定することもできる。自分がどちらを指定したいのかを注意しないと、意図しないトレースを行うことがある。例えば、次のように :inside に generic function を指定する。

```
:trace (foo :inside device-read)
```

この場合、全ての device-read メソッドの実行中に foo がトレースされる。
次のようにメソッドを指定すると、指定したメソッドの実行中だけトレースされる。

```
: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))))
```

5.2 実行例

次の関数 foo, bar, baz, bzz を定義し、:inside, :not-inside の動作を確認してみる。

```
CL-USER(38): (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(39): (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(40): (defun baz (from)
               (progn (format t "baz calling bzz from ~S~%" from)
                      (bzz 'baz)))
BAZ
CL-USER(41): (defun bzz (from)
               (format t "bzz called from ~S~%" from))
BZZ
CL-USER(42): (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
```

少々わかりづらいが、

```
foo -> bar -> baz -> bzz
           -> bzz
    -> baz -> bzz
    -> bzz
```

という順で呼ばれている。ここで :inside **を利用して、**bzz **を** foo **と** baz **の両方が実行中の場合だけトレースしてみる。**

```
CL-USER(43): :trace (bzz :inside (foo baz))
(BZZ)
CL-USER(44): (foo)
foo calling bar          ;; foo -> bar
bar calling baz from FOO ;; foo -> bar -> baz
baz calling bzz from BAR ;; foo -> bar -> baz ->bzz
 0[2]: (BZZ BAZ)         ;; foo と baz が実行中
bzz called from BAZ
 0[2]: returned NIL
bar calling bzz from FOO ;; foo-> bar ->  bzz
bzz called from BAR      ;; foo の実行中であるが baz の実行中でない
foo calling baz          ;; foo -> baz
baz calling bzz from FOO ;; foo -> baz -> bzz
 0[2]: (BZZ BAZ)         ;; foo と baz が実行中
bzz called from BAZ
 0[2]: returned NIL
foo calling bzz          ;; foo の実行中であるが baz の実行中でない
bzz called from FOO
NIL
```

次に、:inside **と** :not-inside **を組み合わせて利用し、**foo **の実行中かつ** bar **の実行中でない場合に** bzz **をトレースする。**

```
CL-USER(45): :untrace    ;; 一度設定をクリア
NIL
CL-USER(46): :trace (bzz :inside foo :not-inside bar)
(BZZ)
CL-USER(47): (foo)
foo calling bar
bar calling baz from FOO
baz calling bzz from BAR
bzz called from BAZ      ;; foo と bar の実行中
bar calling bzz from FOO
bzz called from BAR      ;; 同上
foo calling baz
baz calling bzz from FOO
 0[2]: (BZZ BAZ)         ;; foo が実行中で bar が実行中でない
bzz called from BAZ
 0[2]: returned NIL
foo calling bzz
 0[2]: (BZZ FOO)         ;; 同上
bzz called from FOO
 0[2]: returned NIL
NIL
```

次に generic function **とメソッドをトレースしてみる。次のメソッド** my-function **を定義する。**

```
CL-USER(59): (defmethod my-function ((x integer))
               (cons x :integer))
#<STANDARD-METHOD MY-FUNCTION (INTEGER)>
CL-USER(60): (defmethod my-function ((x float))
               (cons x :float))
#<STANDARD-METHOD MY-FUNCTION (FLOAT)>
```

まず、generic function **をトレースする。引数の型に関係なく、**integer **の場合も** float **の場合もトレースされることがわかる。**

```
CL-USER(61): (trace my-function)
(MY-FUNCTION)
CL-USER(62): (my-function 1)
 0[2]: (MY-FUNCTION 1)
 0* #<STANDARD-METHOD MY-FUNCTION (INTEGER)>
 0[2]: returned (1 . :INTEGER)
(1 . :INTEGER)
CL-USER(63): (my-function 1.0)
 0[2]: (MY-FUNCTION 1.0)
```

```
 0* #<STANDARD-METHOD MY-FUNCTION (FLOAT)>
 0[2]: returned (1.0 . :FLOAT)
(1.0 . :FLOAT)
```

次に、integer **の場合だけトレースしてみる。指定方法は、**`((method` **メソッド名** `...))` **となる。**float **の場合はトレースされないことがわかる。**

```
CL-USER(64): (untrace) ;; 初期化
NIL
CL-USER(65): (trace ((method my-function (integer))))
((METHOD MY-FUNCTION (INTEGER)))
CL-USER(66): (my-function 1)
 0[2]: ((METHOD MY-FUNCTION (INTEGER)) 1)
 0[2]: returned (1 . :INTEGER)
(1 . :INTEGER)
CL-USER(67): (my-function 1.0)
(1.0 . :FLOAT)
CL-USER(68):
```

5.3 setf,:before,:after メソッド, 内部関数のトレース

この節では setf, :before, :after **メソッドと内部関数 (**flet **や** labels **で定義されたもの) のトレース方法を説明する。**
setf, :before, :after **メソッドはそれぞれ次のようにトレースする。**

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

`((method ...))` **というように、括弧は二重にする。**untrace **する場合は、この２つ目の括弧は必要ない。**

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

簡単な例として、メソッド foo **とその** :before **メソッドを定義し、トレースしてみる。**

```
CL-USER(16): (defmethod foo ((i integer))
               (print "foo")
               i)
#<STANDARD-METHOD FOO (INTEGER)>
CL-USER(17): (defmethod foo :before ((i integer))
```

```
               (print "foo :before"))
#<STANDARD-METHOD FOO :BEFORE (INTEGER)>
CL-USER(18):
CL-USER(18): (foo 1)

"foo :before"
"foo"
1
CL-USER(19): (trace ((method foo :before (integer))))
((METHOD FOO :BEFORE (INTEGER)))
CL-USER(20):
CL-USER(20): (foo 1)
 0[4]: ((METHOD FOO :BEFORE (INTEGER)) 1)

"foo :before"
 0[4]: returned "foo :before"

"foo"
1
CL-USER(21): (untrace (method foo :before (integer)))
((METHOD FOO :BEFORE (INTEGER)))
CL-USER(24):
```

次に、内部関数のトレースの例を見てみる。内部関数をトレースするには、関数をコンパイルしておく必要がある。

```
CL-USER(68): (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(69): (compile 'myfunc)
MYFUNC
NIL
NIL
CL-USER(70): (trace ((labels myfunc cubit)))
((LABELS MYFUNC CUBIT))
CL-USER(71): (myfunc 1 2 3)
 0[2]: ((LABELS MYFUNC CUBIT) 2)
 0[2]: returned 8
 0[2]: ((LABELS MYFUNC CUBIT) 3)
 0[2]: returned 27
```

```
35
CL-USER(72):
```

次に、メソッド内で定義された内部関数をトレースする。この場合は、以下を適当なファイルに記述してコンパイルし、fasl **を読み込まなければ** trace **を実行したときにエラーになる。(または** Emacs-Lisp **インタフェースなら** C-c C-x **でコンパイルしてもよい。)**

ファイルの中身

```
(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))))))
```

同様に、メソッド doit **中の** cubit **をトレースする。**

```
CL-USER(3): (trace ((labels (method doit (thing)) cubit)))
((LABELS (METHOD DOIT (THING)) CUBIT))
CL-USER(4): (setf thing1 (make-instance 'thing))
#<THING @ #x4ce469a>
CL-USER(5): (doit thing1)
 0[2]: ((LABELS (METHOD DOIT (THING)) CUBIT) 2)
 0[2]: returned 8
 0[2]: ((LABELS (METHOD DOIT (THING)) CUBIT) 2)
 0[2]: returned 8
16
CL-USER(6):
```

6　まとめ

本レポートでは Allegro CL のデバッグの手段として、スタックコマンド、tracer の使い方を説明した。スタックコマンドを使うと、

1. エラー発生箇所やどの関数が実行されてきたのかを確認 (:zoom) し、(ただし、設定によっては末尾で他の関数を呼び出している関数の frame が表示されないこともある。)
2. ポインタを目的の frame まで移動 (:dn, :up など) して、frame のローカル変数の値を確認 (:local) し、
3. 場合によっては
   - frame の返り値を指定 (:return) したり、
   - frame を指定したものと入れ替え (:restart) たり

   して計算を再開させる

ことができ、バグの発生箇所の確認をしたり、計算を再開させて動作を確認することができる。

tracer は関数だけでなく、generic function、メソッド (:before, after を含む)、内部関数などを対象として、:inside, :not-inside, その他の指定をすることでより複雑な条件下での trace を実行することができる。

デバッグプリントを複数個所に入れるなどのデバッグの手間を省くためにも、これらの機能をうまく利用したい。