Debugging Tools for MIT Scheme and Schematik

Max Hailperin, Mike Hvidsten, and Karl Knight
Gustavus Adolphus College

February 19, 1993

The Scheme system has a number of tools to help you *debug* your programs, that is, to figure out what mistakes you have made which cause incorrect behavior. These same tools can sometimes also be useful to help you understand the behavior of correctly executing programs. Remember that not all debugging is done with these debugging tools. For example, sometimes reading your program over carefully will help you find your error, or evaluating carefully chosen test expressions may narrow down the problem. Finally, your instructors, tutors, and fellow students can be invaluable debugging tools.

In this handout we will address those aids to debugging provided by the Scheme system. Sections 1 and 2 describe error panels and levels of interaction with Scheme. The debugger itself is described in section 3. Section 4 describes the *trace* and *break* facilities, which are also useful for debugging. Finally, a full listing of the debugger commands is given in the appendix.

1 Error panels

Often the most obvious sign of an error in your program will be an error panel that pops up, informing you that Scheme has noticed some sign of trouble, such as being asked to do something impossible. For example, if you try evaluating the expression `(/ 1 0)`, which asks Scheme to divide one by zero, you will get the error panel shown in figure 1.

At this point, you have two options available to you. The most common thing to do is to abort out of the error, by either clicking on the Yes button or pressing the Return key. (Any time a panel has a button marked with the hooked arrow icon, it means that the Return key will have the same effect.) This will insert a copy of the error message into the interaction window for your reference (unless you have disabled this feature with the preferences panel) and go back to waiting for a new evaluation.

The other option is to click the No button, which allows you to examine more carefully the conditions under which the error occurred using the debugger. This option also allows you in some cases to continue past the error so that you can see how the rest of your program behaves without first fixing the error.

If you click the No button, you will be in a second-level read-evaluate-print loop (REPL), which will be indicated by the Level field at the top of the interaction window changing from 1 to 2. See section 2 for more information on this. In the second-level REPL you can evaluate any scheme expression, just like in the

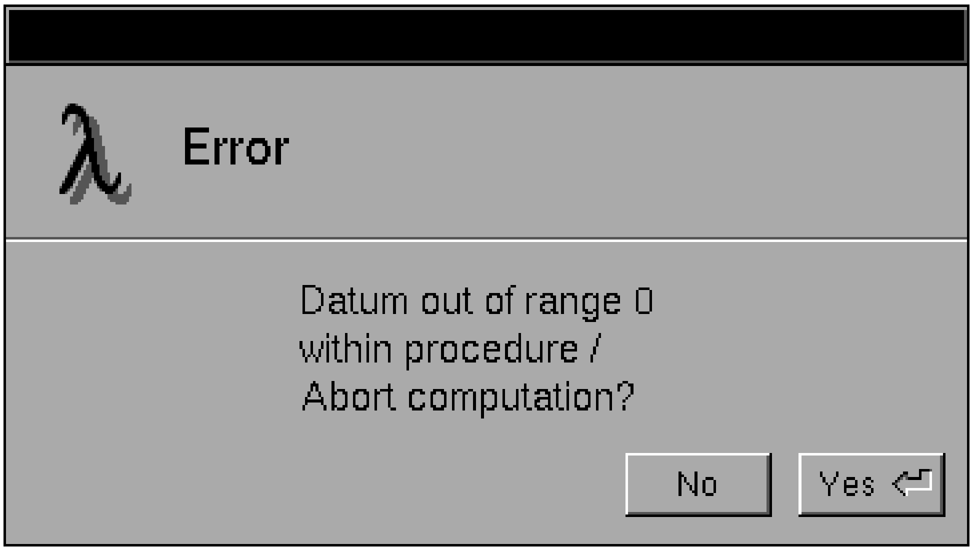

Figure 1: An example error panel, from doing `(/ 1 0)`

first-level. In practice, however, you will generally be most interested in using the debugger, which is described below. If you are done dealing with the error and want to go back to normal work, you should first go back to the previous level, either by using the appropriate debugger command to proceed from the error, or, more typically, by using the Abort to Previous command in the Actions menu.

The most common reason for clicking the No button on an error panel is to find out more information about where and why the error occurred. For example, it may not be obvious to you where in your program a division by zero is occurring. In this case, after clicking the No button, you should evaluate the expression `(debug)` to enter the debugger. Section 3 has more information on how you can then use the debugger to explore the circumstances of the error.

2 Levels of interaction

As mentioned above, it is possible to put the original interaction with Scheme on hold in certain circumstances (such as an error) and enter into a new, second level interaction. This process can be repeated, leading to third level, fourth level, etc. interactions. At any time you can tell your current nesting level by the indicator at the top of the interaction window. Figure 2 shows some typical level indicators. Regardless of the level of interaction, you still send your input to Scheme in the same ways, and the output from Scheme still appears in the interaction window.

Each level of interaction (also called a *command loop*) can be any of a variety of types. The top of the interaction window also indicates the type of the current interaction, as illustrated in figure 2. The level one interaction is a normal read-evaluate-print loop (REPL)—that is, scheme is reading an expression from you, evaluating it, and then printing out the resulting value. This is indicated by `[Normal REPL]`. The lower level interaction you get into when you click No in an error panel is also a REPL, and hence operates in the same fashion. However, to remind you how you got into the lower level, the indicator is changed to

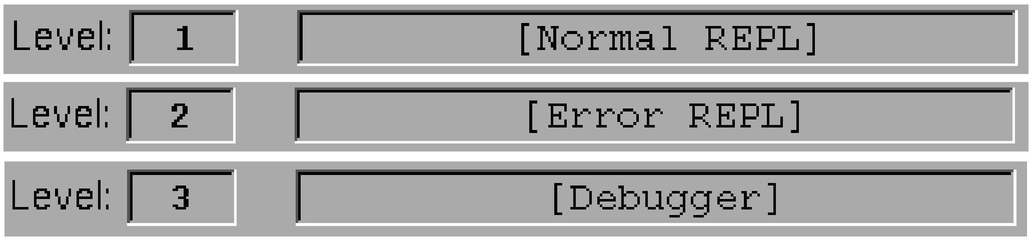

Figure 2: Some typical level indicators

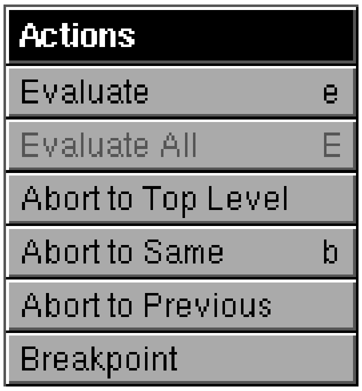

Figure 3: The Schematik Actions menu

`[Error REPL]`.

The debugger, described in section 3, is a different kind of interaction, in that you send Scheme not expressions to evaluate but rather single-character commands to execute. (For example, you might send a `?` to get a list of commands, or a `q` to quit out of the debugger.) This is indicated by the indicator at the top of the interaction window switching from `[Error REPL]` to `[Debugger]`. The most important thing to remember is that you still use the same mechanisms to send your input to Scheme. So, you might use the Evaluate command in the Actions menu to send the `?` command, even though it isn't really an expression to evaluate. (Of course, you could also use command-Return, Enter, or command-e.)

Generally there is some specific way to exit a nested command loop and return to a containing suspended one with lower level number. For example, you can exit an error or breakpoint REPL with the `proceed` procedure, and can quit a debugger with the `q` command. However, there are also general purpose mechanisms you can use to change levels, available from the Actions menu in Schematik, shown in figure 3. The Abort to Previous command will abort any evaluation in progress and return you to the previous command loop, that is, the one with level number one smaller. Abort to Top Level also aborts any ongoing computation, but returns you all the way back to the level one evaluator. (Abort to Same doesn't change level numbers at all, but just aborts ongoing computation.) Finally, there is the command Breakpoint, which suspends any ongoing computation and places you in a "Breakpoint REPL" interaction with level one level greater. If you leave that evaluator by evaluating `(proceed)`, the computation will resume where it left off.

3 The Debugger

As described above, you typically enter the debugger after an error panel informs you that Scheme has detected an error in one of your evaluations. After clicking the No button on the error panel, you can then enter the debugger by evaluating the expression `(debug)`. At this point, you can give the debugger a number of single character commands, which are evaluated using the usual Schematik mechanisms (i.e., command-Return, Enter, or command-e). The full list of commands is given in the appendix to this document; alternatively, you can see the list by sending the debugger the `?` command.

The key point to remember when using the debugger is that Scheme execution proceeds by a combination of two mechanisms: the *reduction* of problems to equivalent but simpler problems, and the solution of component *subproblems* of a problem.

For example, suppose we try the following:

```
(define f
  (lambda (a b)
    (* (+ a b) (- a v))))  ;the v is a typo, should be b
;Value: f

(f 5 4)
```

The result is an error "Unbound variable v" (*unbound* means that there is no value associated with that name).

The evaluation of the expression `(f 5 4)` involves first the reduction of that problem to `(* (+ a b) (- a v))` (in a naming environment where `a` is a name for 5 and `b` is a name for 4). Then, each of the subexpressions is evaluated as a subproblem. So, the error occurs in evaluating `(- a v)` as a subproblem of `(* (+ a b) (- a v))`, which is itself a reduction of `(f 5 4)`. (Actually, the error occurs one subproblem level further down, in the evaluation of `v` as a subproblem of evaluating `(- a v)`.) The important difference between a reduction and a subproblem is that the value of `(* (+ a b) (- a v))` is identical to the value of `(f 5 4)`, while the value of `v`, for example, is merely one component of what is necessary to compute the value of `(- a v)`.

We can take this as the basis for an example debugger session. Initially we try evaluation `(f 5 4)` and click No on the error panel:

```
(f 5 4)
;Unbound variable: v
```

Next, we enter the debugger. Scheme tells us again that the expression causing the error was `v`, lets us know that it occurred within the environment of `f` applied to 5 and 4, and reminds us how to use the debugger commands:

```
(debug)
There are 8 subproblems on the stack.

Subproblem level: 0 (this is the lowest subproblem level)
Expression (from stack):
    v
Environment created by the procedure: f
 applied to: (5 4)
The execution history for this subproblem contains 1 reduction.
;You are now in the debugger.  Type q to quit, ? for commands.
```

Suppose we don't realize exactly where we used `v`. We can see where the evaluation of `v` occurred by going backwards in the evaluation history with the B command, that is to a earlier reduction of the same subproblem or an earlier (larger) subproblem. We can do this repeatedly, as below:

```
B
;Now using information from the execution history.

Subproblem level: 0  Reduction number: 0
Expression (from execution history):
    v
Environment created by the procedure: f
 applied to: (5 4)

B
;No more reductions; going to the next (less recent) subproblem.

Subproblem level: 1  Reduction number: 0
Expression (from execution history):
    (- a v)
Environment created by the procedure: f
 applied to: (5 4)

B
;No more reductions; going to the next (less recent) subproblem.

Subproblem level: 2  Reduction number: 0
Expression (from execution history):
    (* (+ a b) (- a v))
Environment created by the procedure: f
 applied to: (5 4)

B
Subproblem level: 2  Reduction number: 1
Expression (from execution history):
    (f 5 4)
Environment named: (user)
```

At this point, we have traced the history back all the way to our initial interactive evaluation. Along the way, we saw that the `v` in question is the one that occurs in `(- a v)`, which itself is in `(* (+ a b) (- a v))`. Normally we would now abort back to the top level, correct and re-evaluate the procedure definition, and try again.

4 Tracing and breakpointing procedures

The error panels and debugger can be quite useful when Scheme notices a problem, such as a division by zero or an unbound variable. Sometimes, however, you will make a mistake that doesn't result in such overt symptoms. Rather, your program will simply compute the wrong answer or will run forever without producing any answer. For these situations the *trace* and *break* facilities described in this section are particularly useful.

4.1 Trace

Tracing a procedure means setting it up so each time the procedure is entered and/or exited a message will be printed with the procedure's name, argument values, and (on exit) result. You can trace a procedure with `trace-entry`, `trace-exit`, or `trace-both`, depending which information you are interested in. You can stop receiving trace information about a procedure by using `untrace` (or by re-evaluating the procedure definition). The example below illustrates these points.

```
(define factorial
  (lambda (n)
    (if (= n 1)
        1
        (* (factorial (- n 1))
           n))))
;Value: factorial

(trace-entry factorial)
;No value

(trace-exit factorial)
;No value

; or simply: (trace-both factorial)

(factorial 3)
[Entering #[compound-procedure 5 factorial]
    Args: 3]
[Entering #[compound-procedure 5 factorial]
    Args: 2]
[Entering #[compound-procedure 5 factorial]
    Args: 1]
```

```
[1
      <== #[compound-procedure 5 factorial]
    Args: 1]
[2
      <== #[compound-procedure 5 factorial]
    Args: 2]
[6
      <== #[compound-procedure 5 factorial]
    Args: 3]
;Value: 6

(untrace factorial)
;No value

(factorial 3)
;Value: 6
```

4.2 Break

Setting a breakpoint on a procedure's entry and/or exit is similar to tracing it, in that you receive the same information when the procedure is entered or exited. However, additionally the execution is suspended at this point and you are placed in a new evaluator level. You can resume execution with the `proceed` procedure. If you supply an argument to `proceed`, it is returned in place of the break-pointed procedure's result. (If you do this in an entry breakpoint, the procedure is skipped entirely.) There are `break-entry`, `break-exit`, `break-both`, and `unbreak` procedures, analogous to the tracing procedures. Within a breakpoint, you can access the arguments, result, or procedure by using the procedures of no arguments `*args*`, `*result*`, and `*proc*`. (Often you won't need to, as this information has already been printed out.) Here is a simple example, using the same `factorial` procedure as above:

```
(break-exit factorial)
;No value

(factorial 2)
[1
      <== #[compound-procedure 5 factorial]
    Args: 1]
;Breakpoint on exit
```

At this point the level indicator switches from `1 [Normal REPL]` to `2 [Breakpoint REPL]` and the interaction window switches back from `Evaluating...` to `Interaction Window`. The message `;Breakpoint on exit` which is inserted into the interaction window identifies the breakpoint. We will make this inner `factorial` return -1 instead of 1 and observe the effect.

```
(proceed -1)
[-2
      <== #[compound-procedure 5 factorial]
    Args: 2]
;Breakpoint on exit
```

When the `proceed` command was sent, the title switched briefly back to `Evaluating...`, but then the breakpoint message and `Interaction Window` title return, as the outer call to `factorial` prepares to exit. This time, we will just let it go ahead and return -2.

```
(proceed)
;Value: -2
```

4.3 Tracing and breakpointing internal procedures

It is also possible to trace or breakpoint a procedure that is defined within another. For example, `(trace-both foo '(bar baz))` means to trace the procedure `baz` which is defined within the procedure `bar` defined within the procedure `foo`. This can be used to reach any number of levels into the internal structure of a procedure. Also, the same technique works with the breakpointing procedures. Here is a simple example:

```
(define factorial
  (lambda (n)
    (define factorial-product
      (lambda (a b) ; compute a * b!
        (if (= b 0)
            a
            (factorial-product (* a b) (- b 1)))))
    (factorial-product 1 n)))
;Value: factorial

(trace-both factorial)
;No value

(factorial 5)
[Entering #[compound-procedure 5 factorial]
    Args: 5]
[120
      <== #[compound-procedure 5 factorial]
    Args: 5]
;Value: 120

(trace-both factorial '(factorial-product))
;No value

(factorial 5)
[Entering #[compound-procedure 5 factorial]
```

```
    Args: 5]
[Entering #[compound-procedure 6 factorial-product]
    Args: 1
          5]
[Entering #[compound-procedure 6 factorial-product]
    Args: 5
          4]
[Entering #[compound-procedure 6 factorial-product]
    Args: 20
          3]
[Entering #[compound-procedure 6 factorial-product]
    Args: 60
          2]
[Entering #[compound-procedure 6 factorial-product]
    Args: 120
          1]
[Entering #[compound-procedure 6 factorial-product]
    Args: 120
          0]
[120
      <== #[compound-procedure 6 factorial-product]
    Args: 120
          0]
[120
      <== #[compound-procedure 6 factorial-product]
    Args: 120
          1]
[120
      <== #[compound-procedure 6 factorial-product]
    Args: 60
          2]
[120
      <== #[compound-procedure 6 factorial-product]
    Args: 20
          3]
[120
      <== #[compound-procedure 6 factorial-product]
    Args: 5
          4]
[120
      <== #[compound-procedure 6 factorial-product]
    Args: 1
          5]
[120
      <== #[compound-procedure 5 factorial]
    Args: 5]
;Value: 120
```

Appendix: Debugger commands

These are the commands you can use when in the debugger. They are evaluated via the standard Schematik mechanisms (i.e., command-Return, Enter, or command-e). Note that since Scheme is case-insensitive, you may issue these commands in upper or lower case.

? help, list command letters

A show All bindings in current environment and its ancestors

B move (Back) to next reduction (earlier in time)

C show bindings of identifiers in the Current environment

D move (Down) to the previous subproblem (later in time)

E Enter a read-eval-print loop in the current environment

F move (Forward) to previous reduction (later in time)

G Go to a particular subproblem

H prints a summary (History) of all subproblems

I redisplay the error message Info

L (List expression) pretty print the current expression

O pretty print the procedure that created the current environment

P move to environment that is Parent of current environment

Q Quit (exit debugger)

R print the execution history (Reductions) of the current subproblem level

S move to child of current environment (in current chain)

T print the current subproblem or reduction

U move (Up) to the next subproblem (earlier in time)

V eValuate expression in current environment

W enter environment inspector (Where) on the current environment

X create a read eval print loop in the debugger environment

Y display the current stack frame

Z return (continue with) an expression after evaluating it