Debugging Tools for Gambit Scheme under SchematiX

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

September 12, 1995

This handout describes the environment we have on the SGI computers; another handout entitled "Debugging Tools for MIT Scheme and Schematik" describes the corresponding information for the NeXT computers. Since the debugging tools available on the NeXTs are currently considerably more sophisticated, if you find the tools described here inadequate, you might want to temporarily switch to a NeXT for debugging. (A lab instructor would be a good person to consult on whether this would be likely to be fruitful.)

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 debugging commands themselves are described in section 3. Section 4 describes the *trace* facility, which is also useful for debugging. Finally, a full listing of the debugging 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 clicking on the Yes button. 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 debugging

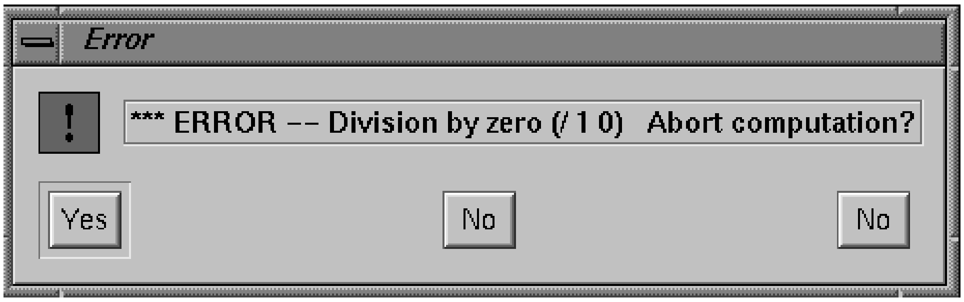

Figure 1: An example error panel, from doing `(/ 1 0)`

commands. 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 level 1 read-evaluate-print loop (REPL), rather than the original level 0 REPL, and this will be indicated by the Level field at the top of the interaction window changing from 0 to 1. See section 2 for more information on this. In the level 1 REPL you can evaluate any scheme expression, just like in the level 0, but they will be evaluated in the context in which the error occured. In practice, you will generally be most interested in using the debugging commands, which are 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 debugging command to proceed from the error or go up a level, 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. Section 3 has more information on how you can then use the debugging commands to explore the circumstances of the error.

2 Levels of interaction

As mentioned above, it is possible to put the original level 0 interaction with Scheme on hold in certain circumstances (such as an error) and enter into a new, level 1 interaction. This process can be repeated, leading to level 2, level 3, 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 way, and the output from Scheme still appears in the interaction window.

There are debugging commands that allow you to move back to the previous level of REPL or all the way back to the top level REPL (i.e., the level 0 REPL). You can do these commands by evaluating `,d` or `,t` respectively. (Note that these commands, like all debugging commands, start with a comma.) Instead of using these commands, you can also use one of the Abort commands in SchematiX's Actions menu. These abort any evaluation that is in progress, unlike the debugging commands; they then can additionally change levels. The Abort to Previous

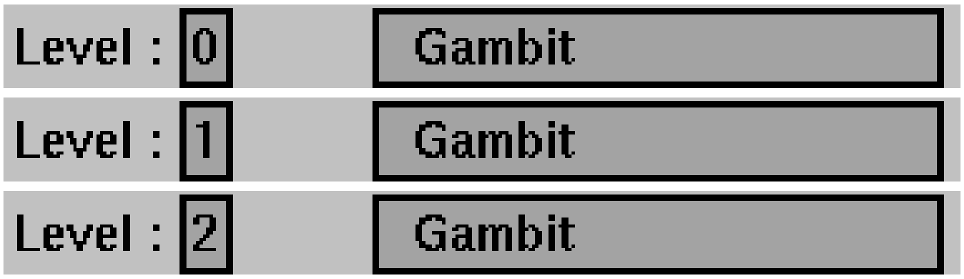

Figure 2: Some typical level indicators

command will abort any evaluation in progress and return you to the previous command loop, that is, the one with level number one smaller, as `,d` would. Abort to Top Level also aborts any ongoing computation, but returns you all the way back to the level 0 evaluator, like `,t`. Abort to Same doesn't change level numbers at all, but just aborts ongoing computation.

3 The debugging commands

As described above, you typically use debugging commands 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 give any of a number of debugging commands, each of which starts with a comma. These commands are evaluated using the usual SchematiX mechanisms (i.e., ctrl-Enter, keypad-Enter, or alt-e). The full list of commands is given in the appendix to this document; alternatively, you can see the list by using the the `,?` command. (The output produced by a debugging command will start on the same line as the command itself, which looks ugly, unless you first start a new line yourself before giving the evaluate command.)

For example, suppose we try the following:

```
(define f
  (lambda (a b)
    (+ (/ a b) 3)))
;Value: f

(define g
  (lambda (x y)
    (if (= y 0)
        (/ 5 x)
        (f x (- x y)))))
;Value: g

(g 5 5)
```

The result is an error "Division by zero." Which of the two divisions caused the error: the one in `g` or the one in `f`? In this case the answer should be clear just by looking at the program, but in a more complex program it might be helpful to use the debugging commands to find out. In this case, we could do this by first clicking the No button on the error panel, and then using debugging commands, as shown below. Let's start by asking to see the expression where the error occured, using `,y`:

```
,y
(/ a b)
```

OK, so the division by zero happened while dividing `a` by `b`. What were `a` and `b`? You could find out by using the `,l` command, which shows the local naming context:

```
,l
a = 5
b = 0
```

Alternatively, you could simply evaluate `a` and/or `b`, since any evaluations you do in this level 1 REPL are done in the local context of the error:

```
a
;Value: 5

b
;Value: 0
```

This information about the erroneous expression may be enough to tell you where in your program the error happened. But, if you have several `(/ a b)` expressions, you might want to know which procedure this one is in. We can do that with the `,i` command:

```
,i
#[procedure f] =
(lambda (a b) (+ (/ a b) 3))
```

So now we know the error occured in procedure `f`. Suppose you wanted to know why proecure `f` was being used. It could be helpful to see what use is going to be made of the value produced by the `f` procedure. We can find this information out using the `,b` (backtrace) command:

```
,b
0     f                          (/ a b)
-1    (top level)                (g 5 5)
-2    ##dynamic-env-bind
-3    ##read-eval-print
-4    ##dynamic-env-bind
-5    ##dynamic-env-bind
-6    ##load
-7    (top level)                (load "/usr/local/lib/schematix-gsi/topl
-8    ##dynamic-env-bind
-9    c#batch-run
-10   c#batch-run
-11   ##initial-continuation
```

Of all the information in this backtrace, only the first two lines are of any interest; everything below that refers to the "inner workings" of the Gambit Scheme system. In general, you can stop looking at the backtrace when you hit the line with `(top level)` in it. What the backtrace shows in this particular instance is that the expression at hand is `(/ a b)` in the procedure `f`, and that the higher level expression for which `f`'s value is needed is `(g 5 5)`, which was evaluated at top level (i.e., by the user).

Normally we would now abort back to the top level, correct and re-evaluate the procedure definition, and try again. (There are some other occasionally useful debugging commands shown in the appendix, and it is also possible to gain information by evaluating arbitrary expressions in the error context, much as we evaluated `a` and `b` above.)

4 Tracing procedures

The error panels and debugging commands 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* facility described in this section is particularly useful.

Tracing a procedure means setting it up so each time the procedure is entered a message will be printed with the procedure's name and argument values, and when the procedure exits, another message will show the result. You can trace a procedure with `trace`, and 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 factorial)
;Value: factorial

(factorial 3)
Entry (factorial 3)
|Entry (factorial 2)
| Entry (factorial 1)
| ==> 1
|==> 2
==> 6
;Value: 6
```

```
(untrace factorial)
;Value: factorial

(factorial 3)
;Value: 6
```

Appendix: Debugging commands

These are the commands you can use for navigating between REPLs and obtaining debugging information. They are evaluated via the standard SchematiX mechanisms (i.e., ctrl-Enter, keypad-Enter, or alt-e).

,? Summary of commands

,+ and ,- Move to next or previous frame of continuation

,n Move to particular frame ($n \leq 0$)

,b Display frames of continuation (i.e. backtrace)

,i Display procedure attached to current frame

,y Display subproblem of current frame

,l Display list of local variables accessible in current frame

,t Transfer to top-level REP loop

,d Transfer to previous REP loop

,r Return from REP loop

,q Quit