

# How to Debug Chez Scheme Programs

R. Kent Dybvig

August 2002

When a program fails to operate as it should, it is said to have a *bug*. A bug is the root cause of an observed behavior, such as failure to terminate, failure to perform some action, termination with an error message, or merely producing incorrect results.

The process of debugging a program is one of finding all of the bugs and “exterminating” them. This process first requires feeding the program a representative set of test cases, observing the results, and determining for which of the test cases the program fails. If the program fails for any of the test cases, the next step is to find the cause of the failure, i.e., the bug. Once a bug has been found, it must be fixed and the debugging process started again from scratch. It is important to retest all of the test cases, even those that originally succeeded, after every change to the program.

Designing good test cases involves the notion of “complete coverage.” Each part of the program must be tested with sets of values that are representative of those it might receive in actual use. For small programs, this is usually straightforward. For large or complex programs, obtaining complete coverage is difficult and often labor-intensive.

Finding a bug once failure has been noted, while often easy, is sometimes quite challenging. The most important thing to remember is that the basic law of cause and effect is always at work. That is, the incorrect behavior exhibited by a program is the result of a bug in that program. (In some cases, it may also be the result of a bug in the operating system.) An important key to successful debugging is to develop an understanding of the cause of the erroneous behavior. Only then can an appropriate fix be made.

Once a bug has been found, it is often a simple matter to fix the bug, although a bug that results from a serious logic error on the part of the programmer may precipitate a complete rewrite of the program.

It is sometimes tempting to hack at a piece of code repeatedly without truly understanding why it’s not doing what it should do until something approximating correct behavior materializes. More often, this simply wastes a lot of time. Even if hacking at the code appears to work, the result will likely just be more subtly incorrect, and ugly.

This document focuses on techniques for finding bugs. It is aimed primarily at beginning programmers who are using Chez Scheme or Petite Chez Scheme to write fairly simple programs, but most of the techniques described here are also useful for more complex programs and for other Scheme systems or even other programming languages. The document is organized into two sections:

- Basic debugging techniques: techniques that will isolate 95% or more of all bugs.
- Advanced debugging techniques: techniques to use when the basic ones do not work.

Regardless of the technique used, debugging programs and testing after each addition increases your odds of finding a bug quickly and fixing it easily. Don’t wait to test and debug your program until after it is all written, or you’ll probably spend more time overall on debugging and may find you have no easy way to fix the bugs you do find.

*Acknowledgements:* This document was influenced by Gary Leaven’s “Chez Scheme error messages and what they might mean,” which appeared on the web around 1993.

## 1. Basic debugging techniques

### 1.1. Kinds of failure

A bug may cause a program to fail in one or more of several ways. Among others, the program may

- terminate with an error message,
- produce the wrong value,
- print the wrong output, or
- fail to terminate (assuming it's supposed to).

The way in which a program fails usually helps determine the steps needed to find the underlying cause. If a program terminates with an error message, the first step is understanding what the error message is telling you. This is the subject of the following section.

## 1.2. Understanding error messages

When an incorrect program terminates with an error message, you can usually determine what's happening pretty quickly. The most important thing is to read the error message and try to understand what it's saying.

In Chez Scheme, an error message has one of the following forms:

**Error:** *message*.

**Error in** *name*: *message*.

When a name is given, it identifies the procedure or syntactic form that detected the error. This is your first clue. For example, if the error message is

**Error in car:** () is not a pair.

you know to look for calls to `car` and try to determine why, for the input you provided, the program applied `car` to the empty list.

Errors that pertain to variable reference or applications and some that don't pertain to any particular syntactic form or procedure aren't identified with a *name*. For example, you might see the following error message.

**Error:** attempt to apply non-procedure 3.

This tells you that there's an application somewhere in your program whose procedure expression evaluates to 3. Often this is caused by a missing quote.

```
> (car (3 4))
```

```
Error: attempt to apply non-procedure 3.
```

In this case, `(3 4)` was not intended to be an application at all; it was intended to be data.

If you're testing something you've just written, chances are that the cause of the error will be immediately obvious. Sometimes, however, the error message does not directly reflect the root cause of the problem. In this case, you must figure out the indirect cause of the error message.

For example, misplaced parentheses often result in syntax errors or "incorrect argument count" errors.

```
> (car (cons 3) 4)
```

```
Error: incorrect number of arguments to #<procedure cons>.
```

```
> (let ([x 0]) (if (= x 3)) 4 5)
```

```
Error: invalid syntax (if (= x 3)).
```

Unbound-variable errors sometimes result when one forgets to define a help procedure or other top-level variable. A less obvious cause is a missing quote on a symbol or list containing symbols.

```
> (cons a '(b c d))
Error: variable a is not bound.
> (cons 'a (b c d))
Error: variable b is not bound.
```

Another common cause is a typo in the name of a variable. These are sometimes difficult to spot.

```
> (let ([x '(a b c)])
      (if (pair x) (car x) x))
Error: variable pair is not bound.
> (define mem
      (lambda (p? ls)
        (if (null? ls)
            '()
            (if (p (car ls))
                ls
                (mem p? (cdr ls)))))))
> (mem even? '(1 2 3 4 5))
Error: variable p is not bound.
```

End-of-file errors sometimes occur while loading files. These are usually caused by a missing double quote, close parenthesis, or close bracket. If the error message reads

```
Error in read: unexpected end-of-file reading string
```

You should look for missing double quote. If the error message reads simply

```
Error in read: unexpected end-of-file
```

the likely cause is a missing close parenthesis or bracket.

The parentheses and brackets used to enclose lists and structured forms may be used interchangeably in Chez Scheme, as long as they are properly matched. If they are not, an unexpected close parenthesis or unexpected close bracket error will occur.

```
> (let ([x 3]) (* x x))
Error in read: unexpected close parenthesis.
Type (debug) to enter the debugger.
> (let ([x (* 3 4)]) (+ x x))
Error in read: unexpected close bracket.
Type (debug) to enter the debugger.
```

### 1.3. Staring at the code

Regardless of the kind of failure, the first and most effective step is to look at the code. Try to understand exactly what it does for the input that exhibits the failure. Think mechanically about the program—like the computer does, trying not to let assumptions based on intended behavior prevent you from seeing what it actually does. Trace through the program in your mind or on paper for the input that causes the failure.

## 1.4. Simplifying the code and input

It's easier to debug a small piece of code on a small input than a large piece of code on a large input. If staring at the code doesn't help, try to simplify the input until it's as simple as you can make it while still causing the incorrect behavior. Try to isolate the portion of the code causing the error by stripping away portions of the code that aren't being executed. Quite often, this process leads to an understanding of the problem.

## 1.5. Printing messages

If staring at the code and simplifying the code and data don't work, try printing messages at selected places in the code. If you're not sure whether evaluation is reaching a particular part of the program, a message that just says "I am here" is useful. Often, it is even more useful to see the values of various variables or expressions at given points in the program. Don't be shy about printing messages from multiple places in the program; this can often provide insight about how the program is working (or not working).

Remember to remove or disable the debugging messages once your program is working properly.

## 1.6. Tracing

Each of the techniques described above are universal in that they work for virtually all languages. Most language implementations provide other debugging aids. Scheme implementations often provide a trace package that allows tracing of procedure calls. Tracing is really just an automated mechanism for printing messages.

In Chez Scheme, tracing is initiated in one of three ways.

1. For procedures bound to top-level variables, including built-in procedures, the `trace` syntactic form may be used. (`trace name ...`) traces the procedures bound to the top-level variables `name ...`.

```
> (define buggy-remove
  (lambda (x ls)
    (if (null? x)
        '()
        (if (equal? (car ls) x)
            (buggy-remove x (cdr ls))
            (cons (car ls) (buggy-remove x (cdr ls)))))))
> (trace buggy-remove null?)
(buggy-remove null?)
> (buggy-remove 'a '(a b a c a))
|(buggy-remove a (a b a c a))
| (null? a)
| #f
|(buggy-remove a (b a c a))
| (null? a)
| #f
|(buggy-remove a (a c a))
| |(null? a)
||#f
|(buggy-remove a (c a))
| |(null? a)
||#f
| |(buggy-remove a (a))
| |(null? a)
| |#f
```

```
| |(buggy-remove a ())
| | (null? a)
| | #f
Error in car: () is not a pair.
```

untrace is used to disable the tracing initiated by trace.

```
> (untrace buggy-remove null?)
(null? buggy-remove)
```

2. Any procedure bound to a variable using `define` can be traced by changing the `define` to a `trace-define`. This works for both top-level and internal definitions.

```
> (define reverse
  (lambda (ls)
    (trace-define buggy-helper
      (lambda (ls1 ls2)
        (if (null? ls1)
            '()
            (buggy-helper (cdr ls1) (cons (car ls1) ls2))))))
    (buggy-helper ls '())))
> (reverse '(a b c))
|(buggy-helper (a b c) ())
|(buggy-helper (b c) (a))
|(buggy-helper (c) (b a))
|(buggy-helper () (c b a))
|()
|()
()
```

3. Anonymous lambda expressions may be traced by changing `lambda` to `trace-lambda` and adding an identifying name.

```
> (define square-all
  (lambda (ls)
    (map (trace-lambda buggy-square (x) (+ x x)) ls)))
> (square-all '(1 2 3 4 5))
|(buggy-square 5)
|10
|(buggy-square 4)
|8
|(buggy-square 3)
|6
|(buggy-square 2)
|4
|(buggy-square 1)
|2
(2 4 6 8 10)
```

Remember to disable tracing once the code is working properly.

## 1.7. Debugging load errors

Sometimes, you can't even get a file to load properly, and this prevents you from running your program to see what it can do.

When the problem is a syntactic error, such as a mismatched bracket or a malformed syntactic form, line and character number information is often provided as part of the error message.

```
Error: message at line line, char char of filename.  
Error in name: message at line line, char char of filename.
```

When a line number is given, the first step is to inspect the code in and around that line to see if you can spot the problem.

If no line number is given, or the line number doesn't help, here are a couple of tricks you can use.

For low-level syntactic errors such as unexpected end-of-file or mismatched parentheses or brackets, try loading the file with `pretty-print` as the evaluation procedure. Normally, you pass one argument to `load`, the name of the file you wish to load. `load` takes an optional second argument, which is the procedure used to evaluate each of the expressions in the file. (This defaults to `eval`.)

If you pass `pretty-print` instead, you'll see all of the forms that could be read successfully; the first one that did not print is the one causing the error. Let's say that the file `t1.ss` contains the following Scheme code.

```
(define-record student (name grades))  
  
(define userid->name  
  (lambda (x)  
    (student-name (lookup x db))))  
  
(define userid->grades  
  (lambda (x)  
    (student-grades (lookup x db))))  
  
(define lookup  
  (lambda (x db)  
    (let ([a (assq x db)])  
      (if a  
          (cdr a)  
          (error 'lookup "no ~s in ~s" x db))))  
  
(define joebob-name (userid->name 'joebob))  
  
(define db  
  (list (cons 'joebob (make-student "Joe" '(a a- b+ a a)))  
        (cons 'jimbob (make-student "Jim" '(b+ a- a- b+ a-)))))
```

Simply loading `t1.ss` results in an end-of-file error, something like this.

```
> (load "t1.ss")
```

```
Error in read: unexpected end-of-file
```

With the default system settings, the error message should include the line number and character position of the unmatched open parentheses. If not, loading `t1.ss` with `pretty-print` as the evaluation procedure yields the following more informative output.

```
> (load "t1.ss" pretty-print)  
(define-record student (name grades))  
(define userid->name  
  (lambda (x) (student-name (lookup x db))))  
(define userid->grades  
  (lambda (x) (student-grades (lookup x db))))
```

Error in read: unexpected end-of-file at char 503 of #<input port t1.ss>.  
Type (debug) to enter the debugger.

From this we can see that the error occurred while reading the fourth expression in the file, i.e., the definition of lookup.

If we correct lookup by adding a parenthesis to the end of the definition and put the result in t2.ss, we now get the following error.

```
> (load "t2.ss")
```

Error: variable db is not bound.  
Type (debug) to enter the debugger.

Loading with pretty-print is uninformative in this case, since all of the expressions read and print without difficulty.

```
> (load "t2.ss" pretty-print)
(define-record student (name grades))
(define userid->name
  (lambda (x) (student-name (lookup x db))))
(define userid->grades
  (lambda (x) (student-grades (lookup x db))))
(define lookup
  (lambda (x db)
    (let ([a (assq x db)])
      (if a (cdr a) (error 'lookup "no ~s in ~s" x db)))))
(define joebob-name (userid->name 'joebob))
(define db
  (list (cons 'joebob (make-student "Joe" '(a a- b+ a a)))
        (cons 'jimbob (make-student "Jim" '(b+ a- a- b+ a-)))))
```

We can, however, use an evaluation procedure that prints as well as evaluates the input; this is more informative.

```
> (load "t2.ss" (lambda (x) (pretty-print x) (eval x)))
(define-record student (name grades))
(define userid->name
  (lambda (x) (student-name (lookup x db))))
(define userid->grades
  (lambda (x) (student-grades (lookup x db))))
(define lookup
  (lambda (x db)
    (let ([a (assq x db)])
      (if a (cdr a) (error 'lookup "no ~s in ~s" x db)))))
(define joebob-name (userid->name 'joebob))
```

Error: variable db is not bound.  
Type (debug) to enter the debugger.

From this output, we know that the error occurs while evaluating the definition of joebob-name, at which point we realize that we're calling userid->name before db is defined.

For more subtle load errors, there is another approach worth trying, which is to comment out portions of the code. This can be done using standard Scheme line comments, i.e., a semi-colon at the front of each line of the code to be commented out. Chez Scheme provides two other ways that are often more convenient: block comments and expression comments. All three forms of comments are illustrated below.

```

; the net result of the code shown below is to define x to be 3
; and z to be 5

; block comments begin with #| and end with|#
(define x #|block comments can span part of a line|# 3)
#|
block comments may span multiple
lines, and #| may be nested|#
|#

; expression comments begin with #; and include the following
; input expression, whatever it may be. they may be nested.
#;(define y 4)
(define z #;(* x #;y 17) 5)

```

## 2. Advanced debugging techniques

Probably 95% or more of all bugs can be found quickly using the techniques described above. For the remaining bugs, the inspection facilities built into Chez Scheme can be used.

### 2.1. Entering the inspector

Most error messages include the following suggestion.

Type `(debug)` to enter the debugger.

If you type `(debug)`, you will be placed in the debug handler.

```
> (let f ([n 10]) (if (= n 0) 'a (* n (f (- n 1)))))
```

```
Error in *: a is not a number.
```

Type `(debug)` to enter the debugger.

```
> (debug)
```

```
debug>
```

Type `“?”` to get a list of the options. Among them should be an option to inspect the error continuation. This is the continuation, or stack, at the point where the error occurred.

The inspector permits you to see the contents of the stack and the contents of any object found on the stack, including free variables of procedures. This information is more useful in Chez Scheme than in Petite Chez Scheme, since the compiler upon which Chez Scheme is based produces information that the inspector uses to identify variables by name. In general, the inspector allows you to navigate through, or inspect, any object that you come across.

### 2.2. Using the inspector

The inspector is menu driven, and you can always see the available options by typing `“?”`. For the most part, the options are self-explanatory. Consult the *Chez Scheme Version 7 User's Guide* for a detailed description of the various options. The command `“s”` (show) is probably the second command (after `“?”`) that you'll want to type; it shows the contents of the object being inspected.

## 2.3. Break handler

For computations that appear to loop indefinitely, the first step is to interrupt the computation. This is done by typing Control-C on most systems, and the result is to put you into the “break handler.” At this point, you have several options. Type “?” to see what these options are. Among them you should see options to reset to the top level, abort from scheme, and inspect the current continuation. Chose the latter if you wish to use the inspector to see where in the computation the apparent indefinite loop has occurred.

## 2.4. Explicit break points

It is also possible to enter the break handler explicitly by inserting a call to `break` into your source code before running it. `break` can be invoked without arguments

```
> (break)
break>
```

or it can be invoked with arguments similar to those used for a call to `error`.

```
> (let f ([x 9])
      (when (< x 0) (break 'f "x is ~s" x))
      (if (= x 0)
          1
          (* x (f (- x 2))))))
```

```
Break in f: x is -1.
break>
```

It can also be invoked with the name only and no message.

```
> (break 'foo)

Break in foo.
break>
```

## 2.5. Invoking the inspector directly

You can bypass the break handler and directly invoke the inspector on any object.

```
> (inspect '(a . b))
(a . b)                                     : s
  car:                                     a
  cdr:                                     b
```

You can use this to “break” into the inspector from an arbitrary point in a computation; using `(call/cc inspect)` you can even inspect the current continuation in this manner.

```
> (let f ([n 5])
      (if (= n 0)
          (begin (call/cc inspect) 1)
          (* n (f (- n 1)))))
#<continuation in f>                         : sf
0: #<continuation in f>
1: #<continuation in f>
2: #<continuation in f>
3: #<continuation in f>
```

```
4: #<continuation in f>  
5: #<continuation in f>  
6: #<system continuation in new-cafe>
```

“sf” (show frames) shows the frames on the stack, starting with the top-most frame.