CS251 Programming Languages **Handout # 7**
Prof. Lyn Turbak **February 1, 2001**
Wellesley College

MIT Scheme

1 Introduction

MIT Scheme is the version of Scheme that we will be using in this course. This handout describes the most important things you need to know in order to use MIT Scheme. For more detailed information, you should consult the *MIT Scheme User's Manual*, available at:

`http://www.swiss.ai.mit.edu/projects/scheme/documentation/user.html`.

We will be using the most recent release (Release 7.5) of MIT Scheme on the Linux machines. If you have your own PC running Windows 95, 98, NT or Linux and would like to install MIT Scheme 7.5, this is possible; see:

`http://www.swiss.ai.mit.edu/projects/scheme/7.5/`.

There are dozens of other implementations of the Scheme programming language which run on a wide variety of computers. Like MIT Scheme, many of them are free. You can find a list of such implementations at:

`http://www.cs.cmu.edu/Web/Groups/AI/html/faqs/lang/scheme/part2/faq.html`.

Be aware that while most Scheme implementations adhere to the R5RS standard, some do not, and those that do often offer a dizzying array of non-standard features. Throughout this course, we will assume that MIT Scheme is the default Scheme implementation.

There are two ways to run MIT Scheme on the Linux workstations: within a Unix shell and within Emacs. These approaches are described below in Sections 2 and 3.

MIT Scheme has a powerful debugger, but it only has a textual interface, not a graphical one. Section 4 is an introduction to the MIT Scheme debugger. **Learning how to use the debugging tools of MIT Scheme effectively will save you *lots* of time in CS251, so I strongly encourage you to invest the time it takes to learn them.**

2 Running Scheme within a Unix Shell

2.1 Entering Scheme

The simplest way to run Scheme on a Linux machine is within a Unix shell window. (See Handout #3 for details on how to log onto the Linux machines and create an shell window.)

To launch Scheme in a shell, execute `scheme`. This will print a herald on the screen and eventually the `1 ]=>` prompt of the Scheme interpreter will appear. Type a simple Scheme expression after the prompt and hit the return key. The interpreter will evaluate the expression, print out its value, and give you another prompt.

2.2 Exiting Scheme

To terminate your session with the Scheme interpreter, evaluate the expression `(exit)`. Make sure that you are really done before exiting, since there is no way to get back to your session once you have exited. (You can always start a new session, but it will not have any of the definitions from your old session.)

2.3 Errors/Debugging

If the interpreter encounters an error when evaluating an expression, it will enter a special read-eval-print loop (REPL) for debugging. This special REPL will be indicated by a prompt that begins with something other than `1 ]=>`, e.g. `2 error>`. You have two basic options at this point:

1. Ignore the error and return to the top-level REPL. You can do this by typing `C-c C-c`.[1] `C-c C-c` is an example of an interrupt sequence. Other interrupt sequences are documented in Section 3.1.2 of the *MIT Scheme User's Manual*.

2. Use the debugger to track down the source of the error. See Section 4 of this handout for more information.

2.4 Loading Files

It is tedious to type all expressions directly at the Scheme interpreter. It is especially frustrating to type in a long definition only to notice that you made an error near the beginning and you have to type it in all over again. In order to reduce your frustration level, it is wise to use a text editor (e.g. Emacs) to type in all but the simplest Scheme expressions. This way, it is easy to correct bugs and to save your definitions between different sessions with the Scheme interpreter.

If *filename* is the name of a file containing Scheme expressions and definitions, evaluating `(load "` *filename*`")` will evaluate all of the expressions in the file, one by one, as if you had typed them in by hand. A loaded file can even contain expressions that load other files!

By default, `load` will only display the value of the last expression in the file. If you would like load to display the values of all of the expressions in the file, you can change the default behavior by evaluating `(set! load-noisily? #t)`. You can return to the default behavior by evaluating `(set! load-noisily? #f)`.

2.5 Scheme Initialization File

You can customize Scheme to automatically load certain features whenever you start a new Scheme session. You do this by creating a file named `.scheme.init` in your home directory and putting into it whatever Scheme expressions you want evaluated when Scheme starts up. For example, if you would like file loading to always be noisy, you can include `(set! load-noisily? #t)` in your `.scheme.init` file.

[1] Notational convention: `C-c`, pronounced "control C", is entered by typing the *control* key (labelled `Ctrl` on most keyboards) and the `c` key at the same time.

3 Running Scheme within Emacs

You could do all of your Scheme programming in CS251 using just the techniques outlined in Section 2 above. However, you will find yourself constantly swapping attention between the Emacs editor (where you write your code) and the Scheme interpreter (where you evaluate your code). In particular, whenever you make a change to your Emacs file, you will have to save the file and reload it in Scheme.

To reduce the overhead of swapping between Emacs and Scheme, you can run Scheme within Emacs. There are Emacs commands for starting Scheme, evaluating Scheme expressions, evaluating Scheme buffers, and so on; these are outlined below. For more information, see Chapter 6 of the *MIT Scheme User's Manual*.

3.1 Entering Scheme in Emacs

To start a Scheme interpreter within Emacs, execute the Emacs command `M-x run-scheme`.[2] This will create an Emacs buffer named `*scheme*` into which the result of all expression evaluations will be displayed.

3.2 Exiting Scheme in Emacs

You can exit Scheme in Emacs by killing the `*scheme*` buffer (using `C-x k`).

3.3 Evaluating an Expression

To evaluate a Scheme expression in Emacs, position the cursor after the last character of the expression and type `C-x C-e`. The value will appear in the Emacs mini-buffer and also in the `*scheme*` buffer. You can evaluate a Scheme expression within any Emacs buffer that is in Scheme mode (see below).

3.4 Evaluating a Definition

You can evaluate the definition in which the cursor currently resides by typing `M-z`. This is equivalent to finding the end of the definition and typing `C-x C-e`.

Whenever you modify a definition, it is wise to let the Scheme interpreter know that you have done so by evaluating it. It can be confusing when the Scheme definitions in the Emacs buffer do not reflect the current bindings within the Scheme interpreter. In fact, this is a common source of problems in CS251. If your program doesn't seem to be working, it's always worthwhile to load it from scratch before doing further debugging.

3.5 Evaluating a Buffer

You can evaluate all of the expressions in a buffer by typing `M-o`. This is equivalent to saving the buffer into a file and loading that file within Scheme. However, typing `M-o` is much faster!

[2]Notational convention: `M-x`, pronounced "meta x", is entered by typing the *meta* key (labelled `Alt` on PC keyboards) and the `x` key at the same time. If you do not have a *meta* key, instead type the *escape* key (typically labelled `Esc`) followed by the `x` key.

3.6 Scheme Mode

An Emacs buffer can be in various modes. Each mode tells the buffer how keystrokes in that buffer should be interpreted. The most useful mode for editing Scheme code is Scheme mode. You can tell a buffer to enter Scheme mode by typing `M-x scheme-mode`. You can tell that a buffer is in Scheme mode by the appearance of the word `Scheme` in the status line at the bottom of Emacs.

Whenever you edit a file that ends in `.scm`, Emacs will automatically put the buffer for that file into Scheme mode. For this reason, it is wise to use the `.scm` extension for all of your Scheme files.

Scheme mode helps you write Scheme code because it understands the formatting conventions for Scheme code. Like many Emacs modes, it helps you match parentheses by flashing the matching open parenthesis whenever you type a close parenthesis. Addtionally, Scheme mode helps you put your code in pretty-printed format. Typing the `Tab` key will indent the code on that line according to the Scheme indentation conventions. You should get into the habit of hitting `Tab` after every return so that you start typing the next line at the appropriate indentation level. You can format an entire expression by typing `C-M-q`[3] when the cursor is at the first character of the expression.

Keeping your Scheme expressions indented properly is important for reading and debugging code. Indenting the code will often highlight that the parenthesis structure of the expression has a bug. See Section 4.1 for more discussion of this point.

4 Debugging Programs in MIT Scheme

This section gives a few hints about debugging Scheme programs in MIT Scheme. See Chapter 5 of the *MIT Scheme User's Manual* for more information. Please see me for a personal tutorial if you want to learn more about Scheme debugging. **Learning how to use the debugging tools of MIT Scheme effectively will save you *lots* of time in CS251, so I strongly encourage you to invest the time it takes to learn them.**

We will explore strategies for debugging Scheme programs in the context of the following buggy version of the `flatten` procedure:

```
(define flatten
  (lambda (tree)
    (if (null? tree)
        '()
        (if (atom? tree)
            tree
            (append (flatten (car (tree))))
                    (flatten (cdr tree))))))
```

4.1 Indentation

The parenthesis-matching feature of Emacs helps you to balance parenthesis. But it does not help to make sure that the parentheses are in the right places. Syntactic errors are often caused by

[3]Notational convention: `C-M-q`, pronounced "control-meta-q", is entered by typing the *control*, *meta*, and `q` keys all at the same time. If your keyboard does not have a *meta* key, first type the *escape* key, and then type the *control* and `q` keys at the same time.

misplaced parentheses. For example, if we evaluate the above definition of flatten, Scheme signals the following error:

```
;SYNTAX: if: too many forms ((flatten ...))
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.
```

A useful strategy for checking parenthesization is to use indentation. Typing the Tab key on any line indents that line according to the Scheme pretty-printing conventions. If you wish to indent a whole s-expression, move the cursor to the open paren of the s-expression and type C-M-q. In our example, indentation gives the following result:

```
(define flatten
  (lambda (tree)
    (if (null? tree)
        '()
         (if (atom? tree)
             tree
             (append (flatten (car (tree))))
             (flatten (cdr tree))))))
```

Note that what was intended to be the second argument to **append** is instead the fourth subexpression of the if expression. Since an if expression must have either two or three subexpressions, this is a syntactic error.

4.2 The stack debugger

Suppose we fix the original defintion of flatten to fix the bug found in Section 4.1:

```
(define flatten
  (lambda (tree)
    (if (null? tree)
        '()
        (if (atom? tree)
            tree
            (append (flatten (car (tree)))
                    (flatten (cdr tree)))))))
```

We now have no trouble evaluating the definition, but **flatten** doesn't work if we call it on an argument:

```
(flatten '((a b) (c) (d (e))))
;The object ((d (e))) is not applicable.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify a procedure to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.
```

If you try this out, you will see that the Scheme level number is now 2, indicating we are in an error read-eval-print loop. We could debug this example, but a good strategy is to try it on a simpler example. First, we type C-c C-c to go back to level 1, and then try again:

```
(flatten '(a b c))
;The object (c) is not applicable.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify a procedure to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.
```

It still doesn't work. OK, let's debug it. In level 2, we evaluate the `(debug)` form. This puts us in Scheme level 3 and gives us access to a stack debugger:

```
(debug)
There are 9 subproblems on the stack.

Subproblem level: 0 (this is the lowest subproblem level)
Expression (from stack):
    ('(c))
There is no current environment.
The execution history for this subproblem contains 1 reduction.
```

Unfortunately, the information is accessed through a textual interface rather than a graphical one. Every debugger command is invoked by a single character. Typing `?` gives a list of all the commands:

```
   ?   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
   J   return TO the current subproblem with a value
   K   continue the program using a standard restart option
   L   (List expression) pretty print the current expression
   M   (Frame elements) show the contents of the stack frame, in raw form
   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 FROM the current subproblem with a value
```

Without a doubt, the most useful debugging command is `H`, which shows the state of the runtime stack when the error occurred. In fact, as a habit, I always type `H` after entering the debugger. In our example, this gives the following stack trace:

```
SL#  Procedure-name          Expression

0                            ('(c))
1    flatten                 (car (tree))
2    flatten                 (flatten (car (tree)))
3    flatten                 (append (flatten (car (tree))) (flatten (cdr t ...
4    flatten                 (append (flatten (car (tree))) (flatten (cdr t ...
5    flatten                 (append (flatten (car (tree))) (flatten (cdr t ...
6                            ;compiled code
7                            ;compiled code
8                            ;compiled code
```

The stack trace tells us that the error occured in the expression `('(c))` when evaluating the expression `(car (tree))` within the call `(flatten (car (tree)))`, which itself was nested within several calls to `append`. Each line represents a subproblem level. By default, the debugger is at subproblem level 0, but we can go to another level by typing G followed by a number and then a return. Suppose we go to subproblem level 1:

```
Subproblem level: 1
Expression (from stack):
    (car ###)
 subproblem being executed (marked by ###):
    (tree)
Environment created by the procedure: FLATTEN
 applied to: ((c))
The execution history for this subproblem contains 1 reduction.
```

We are told that the error occurred within the expression `(tree)`, which appeared within the context `(car (tree))`. If we type `C`, we see a list of the current variable bindings:

```
Environment created by the procedure: FLATTEN
Depth (relative to initial environment): 0
 has bindings:

tree = (c)
```

So tree is the list `(c)`, and we are trying to apply it as a procedure of zero arguments. This is the source of our error.

Before we exit the debugger, let's do a little bit of exploring to get a better sense for the stack. If we go to subproblem level 5 and type `C` we get:

```
Subproblem level: 5
Expression (from stack):
    (append (flatten (car (tree))) ###)
 subproblem being executed (marked by ###):
    (flatten (cdr tree))
Environment created by the procedure: FLATTEN
 applied to: ((a b c))
The execution history for this subproblem contains 4 reductions.
```

```
Environment created by the procedure: FLATTEN
Depth (relative to initial environment): 0
 has bindings:

tree = (a b c)
```

This stack frame is associated with the evaluation of the body to the top-level call `(flatten '(a b c))`. If we now go to subproblem level 4 and type `C` we find:

```
Subproblem level: 4
Expression (from stack):
    (append (flatten (car (tree))) ###)
 subproblem being executed (marked by ###):
    (flatten (cdr tree))
Environment created by the procedure: FLATTEN
 applied to: ((b c))
The execution history for this subproblem contains 4 reductions.


Environment created by the procedure: FLATTEN
Depth (relative to initial environment): 0
 has bindings:

tree = (b c)
```

This frame is associated with the evaluation of the body of the recursive call `(flatten '(b c))`. If we now type the `E` command, we enter a Scheme evaluator at level 4 that can evaluate expressions with the bindings of the stack frame in effect. For example:

```
;You are now in the environment for this frame.
;Type C-c C-u to return to the debugger.

(car tree)
;Value: b

(cdr tree)
;Value 1: (c)
```

To leave the level 4 Scheme evaluator, we can type `C-c C-u` to go up one level to level 3, or type `C-c C-c` to get back to level 1. Let's do the latter and fix our bug:

```
(define flatten
  (lambda (tree)
    (if (null? tree)
        '()
        (if (atom? tree)
            tree
            (append (flatten (car tree))
                    (flatten (cdr tree)))))))
```

```
;Value: flatten

(flatten '(a b c))
;The object c, passed as the first argument to cdr, is not the correct type.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify an argument to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.
```

Oops! We still have an error. Let's evaluate `(debug)` to enter the debugger, followed by `H` to give us a stack trace.

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

Subproblem level: 0 (this is the lowest subproblem level)
Expression (from stack):
    (cdr 'c)
There is no current environment.
The execution history for this subproblem contains 1 reduction.


SL#  Procedure-name          Expression

0                            (cdr 'c)
1    append                  (append (cdr lst1) lst2)
2    append                  (cons (car lst1) (append (cdr lst1) lst2))
3    flatten                 (append (flatten (car tree)) (flatten (cdr tree)))
4    flatten                 (append (flatten (car tree)) (flatten (cdr tree)))
5                            ;compiled code
6                            ;compiled code
7                            ;compiled code
```

The problem is that we are trying to take the `cdr` of the symbol `c`. Let's go to subproblem level 1 to find out why:

```
Subproblem level: 1
Expression (from stack):
    (append ### lst2)
 subproblem being executed (marked by ###):
    (cdr lst1)
Environment created by the procedure: APPEND
 applied to: (c ())
The execution history for this subproblem contains 1 reduction.


Environment created by the procedure: APPEND
Depth (relative to initial environment): 0
 has bindings:

lst1 = c
lst2 = ()
```

Aha! We are trying to append a symbol to a list, but append requires two lists as arguments.

4.3 Tracing

In Section 4.2, we found the where an error occurred, but did not find why it occurred. Why is append getting a symbol as its first argument? The stack debugger doesn't show us the history of the computation, just the current state. To get a better sense for how the computation reached this point, we would like to see a history of calls to various functions.

In many languages, the way you get this sort of history is by sprinkling `print` statements throughout your code in order to track down the source of a bug. Such `print` statements often indicate when a particular function is being called, along with its parameters, or when a particular function returns, along with its return value. While such `print` statements can greatly aid debugging, adding them to and removing them from a program is a tedious and error prone process.

MIT Scheme has a wonderful procedure tracing facility that greatly simplifies debugging by making it unnecessary to insert such `print` statements in common situations. Instead, all one needs to do to get the effect of adding such `print` statements is to "trace" the procedure. This is done by evaluating the expression `(trace` *prof*`)`, where *proc* is an expression that denotes the procedure for which you want information printed on every entry and exit to that function.

4.3.1 Simple Tracing Examples

Here is a simple example of function tracing. Consider the following summation function:

```
(define sum
  (lambda (lst)
    (if (null? lst)
        0
        (+ (car lst)
           (sum (cdr lst))))))
```

We can trace the execution of this procedure by evaluating the expression `(trace sum)`. If we now evaluate `(sum '(1 2 3 4))`, Scheme prints out the following information. Note how the name of traced procedure and its arguments are printed whenever the procedure is invoked, and its name, arguments, and return value are printed whenever it returns (`<==`):

```
[Entering #[compound-procedure 1 sum]
    Args: (1 2 3 4)]
[Entering #[compound-procedure 1 sum]
    Args: (2 3 4)]
[Entering #[compound-procedure 1 sum]
    Args: (3 4)]
[Entering #[compound-procedure 1 sum]
    Args: (4)]
[Entering #[compound-procedure 1 sum]
    Args: ()]
[0
      <== #[compound-procedure 1 sum]
    Args: ()]
[4
      <== #[compound-procedure 1 sum]
    Args: (4)]
[7
      <== #[compound-procedure 1 sum]
    Args: (3 4)]
[9
      <== #[compound-procedure 1 sum]
    Args: (2 3 4)]
[10
      <== #[compound-procedure 1 sum]
    Args: (1 2 3 4)]
;Value: 10
```

You can trace as many procedures as you want to at one time. For instance, consider the following two procedures that implement insertion sort:

```
(define insertion-sort
  (lambda (lst)
    (if (null? lst)
        '()
        (insert (car lst)
                (insertion-sort (cdr lst))))))
(define insert
  (lambda (elt sorted-lst)
    (if (null? sorted-lst)
        (list elt)
        (if (< elt (car sorted-lst))
            (cons elt sorted-lst)
            (cons (car sorted-lst)
                  (insert elt (cdr sorted-lst)))))))
```

We can trace both such procedures by evaluating the following:

```
(trace insertion-sort)
(trace insert)
```

Below is an example of a sample trace. Study it carefully to make sure you understand how insertion sort is working.

```
(insertion-sort '(3 5 1 4 2))
[Entering #[compound-procedure 2 insertion-sort]
    Args: (3 5 1 4 2)]
[Entering #[compound-procedure 2 insertion-sort]
    Args: (5 1 4 2)]
[Entering #[compound-procedure 2 insertion-sort]
    Args: (1 4 2)]
[Entering #[compound-procedure 2 insertion-sort]
    Args: (4 2)]
[Entering #[compound-procedure 2 insertion-sort]
```

```
    Args: (2)]
[Entering #[compound-procedure 2 insertion-sort]
    Args: ()]
[()
      <== #[compound-procedure 2 insertion-sort]
    Args: ()]
[Entering #[compound-procedure 4 insert]
    Args: 2
          ()]
[(2)
      <== #[compound-procedure 4 insert]
    Args: 2
          ()]
[(2)
      <== #[compound-procedure 2 insertion-sort]
    Args: (2)]
[Entering #[compound-procedure 4 insert]
    Args: 4
          (2)]
[Entering #[compound-procedure 4 insert]
    Args: 4
          ()]
[(4)
      <== #[compound-procedure 4 insert]
    Args: 4
          ()]
[(2 4)
      <== #[compound-procedure 4 insert]
    Args: 4
          (2)]
[(2 4)
      <== #[compound-procedure 2 insertion-sort]
    Args: (4 2)]
[Entering #[compound-procedure 4 insert]
    Args: 1
          (2 4)]
[(1 2 4)
      <== #[compound-procedure 4 insert]
    Args: 1
          (2 4)]
[(1 2 4)
      <== #[compound-procedure 2 insertion-sort]
    Args: (1 4 2)]
[Entering #[compound-procedure 4 insert]
    Args: 5
          (1 2 4)]
[Entering #[compound-procedure 4 insert]
    Args: 5
          (2 4)]
    Args: 5
          (4)]
```

```
[Entering #[compound-procedure 4 insert]
    Args: 5
          ()]
[(5)
      <== #[compound-procedure 4 insert]
    Args: 5
          ()]
[(4 5)
      <== #[compound-procedure 4 insert]
    Args: 5
          (4)]
[(2 4 5)
      <== #[compound-procedure 4 insert]
    Args: 5
          (2 4)]
[(1 2 4 5)
      <== #[compound-procedure 4 insert]
    Args: 5
          (1 2 4)]
[(1 2 4 5)
      <== #[compound-procedure 2 insertion-sort]
    Args: (5 1 4 2)]
[Entering #[compound-procedure 4 insert]
    Args: 3
          (1 2 4 5)]
[Entering #[compound-procedure 4 insert]
    Args: 3
          (2 4 5)]
[Entering #[compound-procedure 4 insert]
    Args: 3
          (4 5)]
[(3 4 5)
      <== #[compound-procedure 4 insert]
    Args: 3
          (4 5)]
[(2 3 4 5)
      <== #[compound-procedure 4 insert]
    Args: 3
          (2 4 5)]
[(1 2 3 4 5)
      <== #[compound-procedure 4 insert]
    Args: 3
          (1 2 4 5)]
[(1 2 3 4 5)
      <== #[compound-procedure 2 insertion-sort]
    Args: (3 5 1 4 2)]
;Value 5: (1 2 3 4 5)
```

4.3.2 More Tracing Details

Here are a few more helpful notes on tracing:

- To stop tracing a procedure, evaluate `(untrace  `*proc*`)`, where *proc* is an expression denoting the procedure you no longer want to trace. Evaluating `(untrace)` will untrace all currently traced procedures.
- If you define a new version of a function, any tracing for the previous version of the function will no longer be active. You will need to trace the new version of the function.
- Tracing is an excellent way to localize bugs in your program. But it is also a good way to gain a better understanding for correct code. For instance, if you don't understand how a particular function works, trace it and try it out on some example inputs.
- Procedure tracing is a simple form of a sophisticated Scheme debugging feature that allows arbitrary computations to be performed at procedure entry and exit. See Section 5.4 of the *MIT Scheme User's Manual* for more details.

4.3.3 The `flatten` Example

Now let's return to the `flatten` example that started this whole mess. Assume that we have quit the stack debugger, and will now use Scheme's `trace` procedure to print out information every time the Scheme interpreter enters and exits the `flatten` and `append` procedures:

```
(trace flatten)
;No value
(trace append)
;No value

(flatten '(a b c))
[Entering #[compound-procedure 2 flatten]
    Args: (a b c)]
[Entering #[compound-procedure 2 flatten]
    Args: (b c)]
[Entering #[compound-procedure 2 flatten]
    Args: (c)]
[Entering #[compound-procedure 2 flatten]
    Args: ()]
[()
      <== #[compound-procedure 2 flatten]
    Args: ()]
[Entering #[compound-procedure 2 flatten]
    Args: c]
[c
      <== #[compound-procedure 2 flatten]
    Args: c]
[Entering #[compound-procedure 3 append]
    Args: c
          ()]
;The object c, passed as the first argument to cdr, is not the correct type.
;To continue, call RESTART with an option number:
; ... <other error notices omitted>
```

The trace shows us that the problem is that `(flatten 'c)` returns `c` rather than a singleton list whose sole element is `c`. We can fix this bug, and try again:

```
(define flatten
  (lambda (tree)
    (if (null? tree)
        '()
        (if (atom? tree)
            (list tree)
            (append (flatten (car tree))
                    (flatten (cdr tree)))))))
;Value: flatten

(flatten '(a b c))
[Entering #[compound-procedure 3 append]
    Args: (c)
          ()]
....
```

Oops, we forgot to turn off tracing via `untrace`! Let's turn it off and try again:

```
(untrace)
;No value

(flatten '(a b c))
;Value 5: (a b c)

(flatten '((a b) (c) (d (e))))
;Value 6: (a b c d e)
```

Hey, it works!

4.4 Environment Inspector

In addition to the stack debugger, there is an environment inspector. Suppose we define `test` and `test1` as follows:

```
(define test
  (lambda (a)
    (lambda (b)
      (lambda (c)
        (+ a (* b c))))))
;Value: test

(define test1 ((test 1) 2))
;Value: test1
```

If we print the definition of `test1`, we don't know what the values of the free variables `a` and `b` are:

```
(pp test1)
(lambda (c)
  (+ a (* b c)))
```

We can use the environment inspector to find this information. The environment inspected is invoked via `where`:

```
(where test1)
Environment created by a LAMBDA special form
Depth (relative to initial environment): 0
 has bindings:

b = 2
```

Here are the commands understood by the environment debugger:

```
   ?   help, list command letters
   A   show All bindings in current environment and its ancestors
   C   show bindings of identifiers in the Current environment
   E   Enter a read-eval-print loop in the current environment
   O   pretty print the procedure that created the current environment
   P   move to environment that is Parent of current environment
   Q   Quit (exit environment inspector)
   S   move to child of current environment (in current chain)
   V   eValuate expression in current environment
   W   enter environment inspector (Where) on the current environment
```

Typing `C` shows us the values of the (non-top-level) free variables in `test1`:

```
----------------------------------------
Environment created by a LAMBDA special form
Depth (relative to initial environment): 0
 has bindings:

b = 2

----------------------------------------
Environment created by the procedure: TEST
Depth (relative to initial environment): 1
 has bindings:

a = 1

----------------------------------------
Environment named: (user)
Depth (relative to initial environment): 2

----------------------------------------
Environment named: ()
Depth (relative to initial environment): 3

12
```