

psd: A Scheme Debugger—An Example

Peter Møller Neergaard

When learning a new programming language, your first step is often to write a “hello world”-program. The next step you want to take is to figure out how the debugger works: both good and bad programmers make programming errors, but the former knows and prepares for it. This is where the debugger comes in: despite the name it can’t find your error, but it can help *you* locate the place where your code does not work as you expect.

In general a debugger is a simple but rather effective program. It allows you to insert *breakpoints* in your program. When the program reaches the breakpoint, it will be paused, and you will enter the *debugger*. Inside the debugger you can: *look up* or *change* a variable, run the program *step by step* or *line by line*, or simply resume execution until the program reaches another breakpoint (or terminates). You can thus not only pinpoint the place where the programs behaves wrongly, by setting variables you can recover from the error and continue the execution. Together the functions allow you perform a fine surgery on a faulty program while it is running.

For this course, we provide a symbolic debugger, psd. The debugger is written in Scheme and works as follows: it takes your Scheme program, say, `foo.scm`, and turns it into another Scheme program, say, `foo-annotated.scm`, with annotations that allow it to be debugged. When used in the Scheme interpreter `foo.scm` and `foo-annotated.scm` behave the same except that you can follow the evaluation of the latter step by step or stop it at an arbitrary location. When used inside Emacs (not XEmacs) this can be done behind the scene so it works (almost) as running your Scheme program.

The workings of psd is well-described in the [Quick Introduction](#) accompanying psd. This document therefore contains only notes on using psd on the Berry Patch and a walk-through example.

1 Using psd

Before you can use psd you need to make sure that it is available in Emacs. This is done through the general Emacs settings file for the class. You can do one of two things:

1. Explicitly load the file using the `load-file` command.

```
<ESC> x load-file <RET> ~cs21b/emacs/emacs.el
```

2. Include the following in your `.emacs`-file

```
(load-file (expand-file-name "~cs21b/emacs/emacs.el"))
```

2 An example

We will now walk through an example of using the debugger. Though contrived it will illustrate most of the functions of the debugger. We consider the following *faulty* implementation of Fibonacci (can you spot the error right away?):

```
(define (fib n)
  (if (= n 1)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

We now proceed as follows:

1. We start Emacs and make sure to load the class Scheme definitions as described above in Sec. 1. We open a new buffer and type in the code above (or simply [download it](#)).
2. We start a Scheme buffer using the Emacs command `run-scheme` by typing `M-x run-scheme`.
3. We enable debugging in the Scheme buffer using `psd-mode` by typing `M-x psd-mode`. This should result in something like the following output:

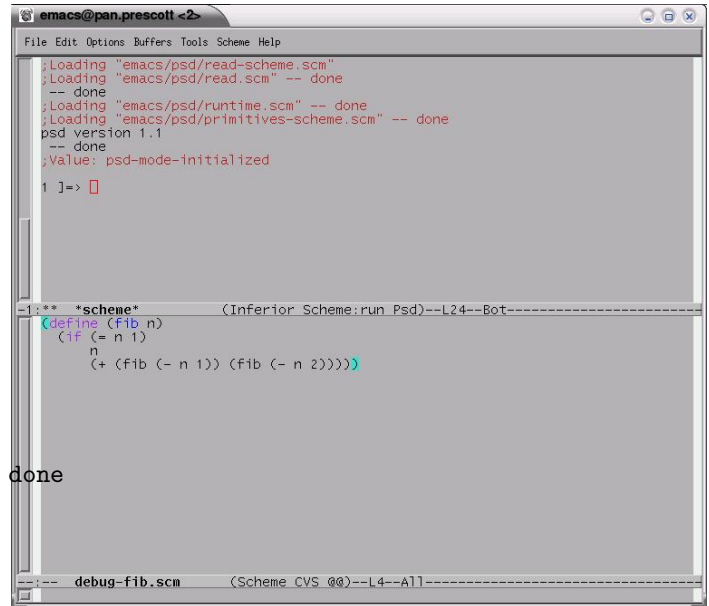
```
1 ]=>
;Loading "emacs/psd/psd-scheme.scm"
;Loading "emacs/psd/qp.scm" -- done
;Loading "emacs/psd/version.scm" -- done
;Loading "emacs/psd/instrum-scheme.scm" -- done
;Loading "emacs/psd/pexpr.scm" -- done
;Loading "emacs/psd/read-scheme.scm"
;Loading "emacs/psd/read.scm" -- done
-- done
;Loading "emacs/psd/runtime.scm" -- done
;Loading "emacs/psd/primitives-scheme.scm" -- done
psd version 1.1
-- done
;Value: psd-mode-initialized
```

- We switch back to the buffer with the Fibonacci program by typing C-x 4 b <RET>. The screen should now look something like the picture to the right.
- We place the cursor after the last parenthesis of the fib definition and send the definition to the Scheme buffer by typing C-u C-c C-e (instead of the usual C-c C-e).

This moves us to the Scheme buffer and we get a response like

```
1 ]=>
;Loading "/tmp/psd223936Sxz" -- done
;Value: ok
```

The loading reveals the annotation described in the introduction on Page 1: our definition has been annotated, stored in a temporary file, and is loaded from that file.



- We type (fib 5) <RET> to find Fibonacci of 5. After a while,¹ this results in an error:

```
1 ]=> (fib 5)

;Aborting!: maximum recursion depth exceeded
```

because our program is faulty. So now it is time to put the debugger to use.

- We change back to the buffer with the source code (by typing C-x o).

We will insert a *breakpoint* at the second line of fib. We move the cursor to the line (if (= n 1) and set the breakpoint by typing C-x <SPC>. We see the response in the Scheme buffer:

```
1 ]=>
;Value 1: "Breakpoint at /home/turtle/teaching/cosi21b/fib.scm:2"
```

- We can now go back and run the program again: we type C-x o (fib 5) <RET>. We immediately get the response:

```
1 ]=> (fib 5)
(fib 5)
psd>
```

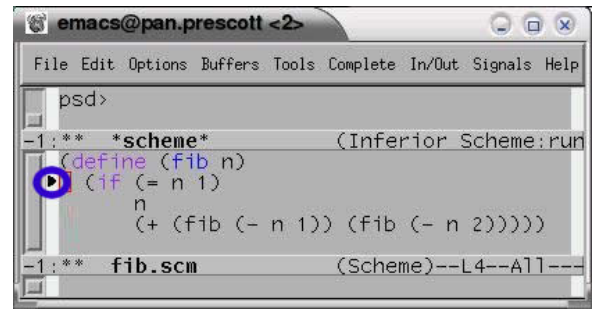
¹Because of the annotations the program runs slower than usual.

psd> is the prompt of the debugger indicating that it is expecting an input. The line (fib 5) describes the *context* which is the expression we are about to evaluate.

The position is also indicated in the program buffer by a ► in the left border next to the scroll bar. This is marked with a circle in the figure to the right.

9. We can check that it is indeed at the very first innovation of fib that the program has been paused by looking at the value of the variable n. We do this with the psd-command val:

```
psd> val n
5
```



10. To follow the execution we will first try to step through the program with **step** command:

```
psd> s
(= n 1)
psd> s
(= n 1) ==> ()
psd> s
(+ (fib (- n 1)) (fib (- n 2)))
psd> s
(fib (- n 2))
psd> s
(- n 2)
psd> s
(- n 2) ==> 3
psd> s
(fib 3)
psd> s
(= n 1)
psd> s
(= n 1) ==> ()
```

The debugger reveals all the small steps the program goes through: to evaluate (+ (fib (- n 1)) (fib (- n 2))) the subexpression (fib (- n 2)) is evaluated which again requires the evaluation of the subexpression (- n 2) which; the latter subexpression turns out to evaluate to the value 3.

11. The step-by-step evaluation quickly become tedious so the **next** command which goes line-by-line provides larger steps. We continue our evaluation:

```
psd> n
```

```

(+ (fib (- n 1)) (fib (- n 2)))
psd> n
(fib 1)
psd> n
n
psd> n
(fib 1) ==> 1
psd> n
(fib 2)

```

At first, this can look confusing, but the system is that the program paused every time the current line changes. Thus when evaluating the expression `(+ (fib (- n 1)) (fib (- n 2)))`, the program needs to evaluate the sub-expressions `(fib (- n 1))` and `(fib (- n 2))`. The latter is evaluated first which results in the call of `(fib 1)`; this call leads us to the conditional which occurs on a different line. And from the conditional we go to the evaluation of the *consequent* `n`. This brings us back to the original expression where `(fib (- n 1))` should be evaluated; this results in a call to `(fib 2)`.

12. Overloaded by the amount of information and still without a clue of the error, we resort to letting the program run until it encounters the next breakpoint. This is done with the `go`-command:

```

psd> g
(fib 0)
psd> g
(fib -2)

```

This finally reveals our mistake: we have forgot the `<` in the less-than-or-equal test.

13. A quick remedy is that we know that Fibonacci of 0 is 0.² We can use the debugger to tell that this is the return value. That way we can finish up the execution of the program and see if there are other errors. This results in the following run:

```

psd> r 0
(fib -1)
psd> r 0
(fib 1)
psd> g
(fib 4)
psd> g
(fib 2)

```

²The usual definition of Fibonacci is $\text{fib}(n) = 1$ when $n = 0, 1$ and $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$, but using $\text{fib}(n) = n$ when $n = 0, 1$ has some nicer mathematical properties including $\text{fib}(5) = 5$. What it has to do with procreation of rabbits I do however not know.

```

psd> g
(fib 0)
psd> r 0
(fib 1)
psd> g
(fib 3)
psd> g
(fib 1)
psd> g
(fib 2)
psd> g
(fib 0)
psd> r 0
(fib 1)
psd> g

;Value: 5

```

Notice that we have to specify the correct return value every time the program starts evaluating (fib 0).

14. We can now go back to the program buffer (C-x o) and correct the error (or [download the fixed program](#)):

```

(define (fib n)
  (if (<= n 1)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))

```

15. We repeat the step of sending and executing the program: C-u C-x C-e C-x o (fib 5) <RET>. We immediately get

```

1 ]=> (fib 5)

; Value: 5

```

revealing that our breakpoint was cleared when we provided the new definition.

3 Exercises

The following is a couple of small exercises that you can do before you throw yourself at debugging your own programs.

Exercise: tracing factorial

Take the two implementations of factorial in Section 1.2.1. Step through the execution to accomplish yourself with the debugger and how Scheme evaluates expressions.

Exercise: faulty fast exponentiation

Consider the following faulty implementation of the **fast exponentiation**.

```
(define (fast-expt b n)
  (cond ((= n 0) 0)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

Use the debugger to locate and correct the error.