

Comments in Scheme

Programs are intended to be read both by people and by computers. Because people understand much richer and more flexible notations than computers -- real languages, as opposed to the extremely limited pseudo-languages used to direct the execution of algorithms -- it is essential to be able to include comments in program files. Comments are intended exclusively for human readers; the computer ignores them as if they were so much blank space.

In a Scheme program, any line that begins with one or more semicolons is a comment:

```
; This is a comment. DrScheme will store it along with the rest of the
; text of a program, but does not even attempt to execute any part of it.

;;; The following definition, however, is read and processed:

(define centimeters-in-an-inch 254/100)

;;; And the following expression is evaluated when this program is
;;; executed:

(* 36 centimeters-in-an-inch)
```

It is also possible to place a comment to the right of a definition or command. The semicolon and everything to its right (on the same line) are ignored during execution.

```
(define centimeters-in-a-foot
  (* 12 centimeters-in-an-inch)) ; One foot equals twelve inches.
```

We'll use comments for several purposes in Scheme programs:

Definitions are seldom completely self-explanatory. Usually, you'll want to preface each definition with a comment explaining what you want to define and how you want to define it.

```
;;; The number of seconds in a day is the product of the number of hours in
;;; a day (twenty-four), the number of minutes in each hour (sixty), and
;;; the number of seconds in each minute (also 60).
(define seconds-in-a-day (* 24 60 60))
```

I recommend that you write the comment before writing the actual definition. Writing the comment helps the programmer articulate and specify the idea that the definition will express. Especially when you're just starting to program, it's useful to separate this stage of clarifying your ideas from the stage in which you have to think about the syntactic structure of the programming-language notation and the idiosyncrasies of Scheme's expression-evaluator.

If the calculation that your program performs is tricky or subtle, or if it depends on some non-obvious condition that is satisfied in one particular case but usually can't be relied on, you should insert a comment to explain what's going on:

```
(quotient total sample-size) ; The sample size cannot be zero, since
                               ; at least one sample is constructed by
                               ; the make-base-sample procedure.
```

Even if you are the only one who will ever read your program, your future self, trying to understand the program, may need a reminder or two. It is astonishing, incidentally, how quickly this sort of detail fades from one's memory. Write it down, even if you don't now expect to forget it.

Since it's so easy to exchange programs over the Internet, programs that you write and send to others will eventually reach people who have never heard of you. It's a good idea to include a comment at the beginning of each program identifying yourself as the author and explaining how to reach you in case the reader has questions about your program:

```
;;; John David Stone
;;; Department of Mathematics and Computer Science
;;; Grinnell College
;;; stone@cs.grinnell.edu

;;; created October 14, 1997
;;; revised October 17, 1997
;;;   Added XXX
;;; revised January 31, 2001
;;;   Added XXX
```

A long program usually begins with a long opening comment that explains the purpose and structure of the program and presents an overview of what is to come. Here's a real-world example:

```
;;; The Apache server for hypertext documents records information about its
;;; actions in two log files, access_log and error_log. This program
;;; parses those logs and generates a summary of their contents for the
;;; benefit of the webmaster.
```

With comments, you can and should think of writing a program as rather like writing an essay in which you describe the problem you're trying to solve and your method of solution. The code sections fit into such an essay as exhibits showing the exact, formal algorithms that express your solution.

I also recommend that you make it a point to carefully document your procedures, often in advance of writing the procedures. I use a mnemonic of "the six P's" to remember what to document about procedures:

- *Procedure*: the name of the procedure;
- *Parameters*: the input to the procedure;
- *Purpose*: what the procedure does;
- *Produces*: what kind of result the procedure has;
- *Preconditions*: what must hold in order for the procedure to succeed (for example, some procedures may only work for positive integers);
- *Postconditions*: what you know still holds after the procedure finishes (you may know something about the form of the result).

However, there are also many other things you can document when describing your procedures. You might note:

- *Problems*: things that are likely to go wrong (your preconditions should prevent problems, but sometimes it's nice to add a few more notes);
- *Examples*: show your procedure in action (and not its inaction);
- *Notes*: comments that don't fit into any of the other categories.

Here's a modified example from one of my standard sets of utilities:

```
;;; Procedure:
;;; list->html
;;; Parameters:
;;; lst, a list of strings
;;; Purpose:
;;; Convert a list of strings to an HTML unordered list.
;;; Produces:
;;; A string representing the HTML for the unordered list.
;;; Preconditions:
;;; lst must be nonempty.
;;; each element of lst must be a string.
;;; each element of lst must be valid HTML.
;;; Postconditions:
;;; The string returned is valid HTML.
;;; A typical browser will render that list as something like:
;;; * first element of list
;;; * second element of list. we can assume that this
;;; goes on for more than one line
;;; * another element of list
```

If you document your procedures carefully, you'll quickly find that you're able to build up a library of procedures you can use without worrying about remembering exactly how they work. Note that you'll often find that the documentation for the procedure is longer than the implementation of the procedure.