

Documentation

Summary: We consider the reasons and techniques for documenting procedures.

Contents:

- Introduction
- The Audience for Your Documentation
- Documenting Procedures with the Six P's
- Documenting Preconditions and Postconditions
- Some Documentation Examples
- Preconditions and Postconditions as Contracts, Revisited
- Other P's

Introduction

When programmers write code, they also *document* that code; that is, they write English (or English-like text) and a bit of mathematics to clarify what their code does. The computer certainly doesn't need any such documentation (and even ignores it), so why should one take the time to write documentation? There are a host of reasons.

- The design of an algorithm may not be obvious. Documentation can explain how the algorithm works.
- Particular details of the implementation of the algorithm may include subtleties. Documentation can explain those subtleties.
- Programmers who use a procedure should be able to focus more on *what* the procedure does, rather than *how* the procedure does its job. (You can certainly use `sqrt`, `list-ref`, and a host of other procedures without understanding how they are defined.)

As all three examples suggest, when we write code, we write not just for the computer, but also for a human reader. Even the best of code needs to be checked again on occasion, and lots of code gets modified for new purposes. Good documentation helps those who must support or modify the code understand it.

The Audience for Your Documentation

As you should have learned in Tutorial, every writer needs to keep in mind not only the topic they are writing about, but also the *audience* for whom they are writing. This understanding of audience is equally important when writing documentation.

One way to think about your audience is in terms of how the reader will be using your code. Some readers will be reading your code to understand techniques that they plan to use in other situations. Other readers will be responsible for maintaining and updating your code. Most readers will be using the procedures you

write. You are often your own client. For example, you are likely to reuse procedures you wrote early in the semester. The documentation you write for your *client programmers* is the most important documentation you can write.

When thinking about those clients, you should first remember that they care most about *what* your procedures do: What values do they compute? What inputs do they take? Although you will be tempted to describe *how* you reach your results, most of your clients will not need to know your process, but only the result.

But you need to think about more than how your audience will use your code. You also need to think about what they know and don't know. Because you are novices, you should generally plan to write for people like you: Assume that your client programmers know very little about Scheme, the kinds of things your program might do, even the terminology you use.

Documenting Procedures with the Six P's

Different organizations have different styles of documentation. After too many years documenting procedures and teaching students to document procedures, I've developed a style that I'm happy with and that I find helps students think carefully about their work. (I've also received a few notes from folks in industry who see this documentation and praise me for teaching it to students.)

To keep it easy to remember what belongs in the documentation for a procedure, I say that students should focus on "the six p's": Procedure, Parameters, Purpose, Produces, Preconditions, and Postconditions.

The *procedure* section simply names the procedure. Although the name of the procedure should be obvious from the code, but including the name in the documentation, we make it possible for the client programmer to learn about the procedure *only* through the documentation.

The *parameters* section names the parameters to the procedure and gives them types. For example, if a procedure operates only on numbers, the parameters section should indicate so.

The *purpose* section presents a few sentences that describe what the procedure is supposed to do. The sentences need not be as precise as what you'd give a computer, but they should be clear to the "average" programmer. (As you've learned in your writing, write to your audience.)

The *produces* section names and types the result of the procedure. Often, the result is not named in the code of the procedure. So why do we both to include such a section? Because naming the result lets us discuss it (either in the purpose above or in the preconditions and postconditions below).

Documenting Preconditions and Postconditions

These first four P's give an *informal* definition of what the procedure does. The last two P's give a much more *formal* definition of the purpose of the procedure. You've seen at the beginning of this reading that the preconditions are the conditions that must be met in order for the procedure to work and that preconditions and postconditions are a form of contract. Since they are a contract, we do our best to specify them formally.

The *preconditions* section provides additional details on the valid inputs the procedures accepts and the state of the programming system that is necessary for the procedure to work correctly. For example, if a procedure depends on a value being named elsewhere, the dependency should be named in the preconditions section. The preconditions section can be used to include restrictions that would be too cumbersome to put in the parameters section. For example, in many programming languages, there is a cap on the size of integers. In such languages, it would therefore be necessary for a `square` procedure to put a cap on the size of the input value to have an absolute value less than or equal to the square root of the largest integer.

When documenting your procedures, you may wish to note whether a precondition is *verified* (in which case you should make sure to print an error message) or *unverified* (in which case you may still crash and burn, but the error will come from one of the procedures you call).

The *postconditions* section provides a more formal description of the results of the procedure. For example, we might say informally that a procedure reverses a list. In the postconditions section, we provide formulae that indicate what it means to have reversed the list.

Typically, some portion of the preconditions and postconditions are expressed with formulae or code.

How do you decide what preconditions and postconditions to write? It takes some practice to get good at it. I usually start by thinking about what inputs I'm sure it works on and what inputs I'm sure that it doesn't work on. I then try to refine my understanding so that for any value someone presents to me, I can easily decide which category is appropriate.

For example, if I design a procedure to work on numbers, my general sense is that it will work on numbers, but not on the things that are not numbers. Then I start to think about what kinds of number it won't work on. For example, will it work correctly with real numbers, with imaginary numbers, with negative numbers, with really large numbers? As I reflect on each case, I refine my understanding of the procedure, and get closer to writing a good precondition.

The postcondition is a bit more tricky. I try to phrase what I expect of the procedure as concisely and clearly as I can, using math when I can, code when it's clearer than the math, and English when I can't quite figure out the math or code. But I always remember, in the back of my mind, that English is ambiguous, so I try to use formal notations whenever possible.

Some Documentation Examples

Let us first consider a simple procedure that squares its input value and that restricts that value to an integer. Here is one possible set of documentation for an environment in which there is a limit to the size of integers.

```
;;; Procedure:
;;;   square
;;; Parameters:
;;;   val, an integer
;;; Purpose:
;;;   Computes the square of val.
;;; Produces:
;;;   val-squared, an integer
```

```

;;; Preconditions:
;;;   MAXINT is defined and is the size of the largest possible
;;;   integer.
;;;   val <= (sqrt MAXINT)
;;; Postconditions:
;;;   val-squared = val*val

```

You will note that the preconditions specified are those described in the narrative section: We must ensure that `val` is not too large and we must ensure that the value we use to cap `val` is defined. Here, I started with the idea of numbers (or integers) and, as I started to think about special cases, I realized that I couldn't have too large numbers, so I added the restriction on its size.

Let us then turn to the famous `reverse` procedure, a procedure that reverses the order of elements in the list.

```

;;; Procedure:
;;;   reverse
;;; Parameters:
;;;   lst, a list
;;; Purpose:
;;;   Reverses lst.
;;; Produces:
;;;   reversed, a new list.
;;; Preconditions:
;;;   (none)
;;; Postconditions:
;;;   Let n be the length of lst
;;;   For all i such that 0 <= i < n
;;;     (list-ref reversed i) equals (list-ref lst (- n i 1))

```

There are many things to note about this definition. First, we don't list any preconditions, even though we require that `lst` is a list. We can avoid other preconditions because the types specified in the parameters section are included implicitly. Next, the postconditions are stated quite formally, with a combination of mathematical and Scheme notation. Using a formal notation helps avoid ambiguities. Finally, we don't specify every postcondition. For example, `reverse` has no effect on its parameters. Typically, unless a postcondition specifies something happens, we assume that the thing does not happen. In particular, we assume that procedures do not modify their parameters unless the postcondition specifies so. (If the procedure's name ends with an exclamation point, we are more likely to expect that something will be modified, but the documentation should specify what.)

Preconditions and Postconditions as Contracts, Revisited

As noted above, the preconditions and postconditions form a contract with the client programmer: If the client programmer meets the type specified in the parameters section and the preconditions specified in the preconditions section, then the procedure must meet the postconditions specified in the postconditions section.

As with all contracts, there is therefore a bit of adversarial balance between the preconditions and postconditions. The implementer's goal is to do as little as possible to meet the postconditions, which means that the client's goal is to make the postconditions specify the goal in such a way that the implementer cannot make compromises. Similarly, one of the client's goals may be to break the procedure

(so that he may sue or reduce payment to the implementer), so the implementer needs to write the preconditions and parameter descriptions in such a way that she can ensure that any parameters that meet those descriptions can be used to compute a result.

Other P's

Although we typically suggest using the basic six P's (procedure, parameters, purpose, produces, preconditions, and postconditions) to document your procedures, there are a few other optional sections that you may wish to include. For the sake of alliteration, we also begin those sections with the letter P.

In a *Problems* section, you might note special cases or issues that are not sufficiently covered. Typically, all the problems are handled by eliminating invalid inputs in the preconditions section, but until you have a carefully written preconditions section, the problems section provides additional help (e.g., “the behavior of this procedure is undefined if the parameter is 0”).

In a *Practicum* section, you can give some sample interactions with your procedure. We find that many programmers best understand the purpose of programs through examples, and the Practicum section gives you the opportunity to give clarifying examples.

In a *Philosophy* section, you can discuss the broader role of this procedure. Often, procedures are written as part of a larger collection. This section gives you an opportunity to specify these relationships.

In a *Process* section, you can discuss a bit about the algorithm you use to go from parameters to product. In general, the client should not know your algorithm, but there are a few times it is helpful to reveal a bit about the algorithm.

You may find other useful P's as your program. Feel free to introduce them into the documentation. (Feel free to use terms that do not begin with P, too.)

Copyright © 2007 Janet Davis, Matthew Kluber, and Samuel A. Rebelsky. (Selected materials copyright by John David Stone and Henry Walker and used by permission.) This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.