1 Introduction

There are many different ways to write functionally-equivalent programs. These differences can be either semantic or syntactic. On the semantic side, two fundamentally different procedures might compute the same function: e.g.,

\[ x^2 = x + \ldots + x \]

\[ x \times \text{times} \]

and

\[ x^2 = (x + 1)(x - 1) + 1 \]

In such cases, we rely on analysis of algorithms to help us decide which procedure to favor. One procedure may prove to be faster than another, or to use less space.

On the syntactic side, extra whitespace here and there does not change the functionality of a program. So the question arises, is one syntactic means of expressing a program preferable to another? For example, is \((sqr 17)\) preferable to \((sqr 17)\)? From the point of view of resource utilization, there is no difference between these two programs. But from the point of view of human readability, there might be.

Abelson and Sussman, two pioneers of Scheme, wrote in their book, *Structure and Interpretation of Computer Programs*, “Programs must be written for people to read, and only incidentally for machines to execute.” But in practice, it can be difficult to write human-readable code.

One important desideratum is that programs be concise, because it can be faster and easier to digest shorter programs. But at the same time, programs must not be too short (e.g., without any white
space at all), because they must be human readable. A style guide serves to encourage programs that occupy the space between too short—and hence unreadable—and unnecessarily verbose.

Some of the rules of English grammar are rigid: e.g., separate the elements in a series with commas. But some of the rules of English grammar are flexible. For example, the Oxford comma—the comma that precedes the last element in a series—is optional. Consider this example:

For dinner, they had broccoli, macaroni and cheese and toast.

Without the Oxford comma, we do not know if they had cheese and toast, and separately macaroni; or macaroni and cheese, and separately toast? A style guide for an English class would decide about Oxford commas one way or the other—most likely requiring them, as they can avoid ambiguities.

Similarly, a computer science style guide makes decisions for you in cases where a programming language grammar is flexible. Examples are the use of white space (indentation, line breaks, etc.), where to comment and to what extent, and naming conventions. Some CS style guides also guide you toward certain language constructs and away from others when their logic is equivalent: e.g., in Racket, we prefer cond to if.

Style is emphasized not only in CS 17; it is also emphasized in upper-level computer science courses in the department because style is an important part of the computer science industry. To our knowledge, all large companies (e.g., Apple, Google, Microsoft, Amazon) require that their employees abide by style guides. Even small companies enforce style guidelines if for no other reason than to prevent team members from rewriting one another’s code over and over again (one programmer likes to indent by two spaces, another by four).

As the year progresses, you may find that you do not like some of the style conventions we have adopted in CS 17 / 18. When this happens, keep in mind that some conventions are arbitrary, made because having everyone doing the same thing makes working together easier. For example, in North America, we drive on the right-hand side of the road. Surely it is preferable for you to abide by this arbitrary convention when it comes to driving, than your own individual preference, especially if your individual preference is to drive on the left! Likewise for writing software: it is better to abide by the standards of the course (or your institution) for everyone’s sanity, because code is often hard to read even when everyone follows the conventions. And don’t be deceived: writing unreadable code could be as detrimental to your well-being as driving on the left in a country where driving on the right is the norm.

Following industry’s lead, we abide by strict style guidelines in CS 17 / 18. These guidelines are intended to help you and others read—and hence, debug—your code more easily. Our style guidelines are set forth in this general style guide, and in two more language specific style guides, one for Racket, and another for OCaml.

The TAs expect you to follow the style guidelines set forth in these documents. Your assignments will be graded on style. See the section of the course missive on grading rubrics for more details.

2 Formatting

Here are some formatting basics that will apply across the board in CS 17 (meaning in both Racket and OCaml).
1. Limit the width of your programs to 80 characters or less (that way they will render well even in small text windows, and print well, too).

2. Never insert a blank line in the midst of an expression.

3. Skip lines between procedure definitions.

4. Use only blanks to insert space, not tabs. (It is okay to use the tab key only if your editor responds to the tab key by inserting the proper number of spaces.)

3 Naming

Here is a loose guideline to follow in choosing identifier names: * Prefer names that are meaningful to you and that you think will be meaningful to others as well.

To name identifiers, there are two standard approaches:

- Choose a name that indicates the type of data expected: e.g., `num`, for a number; `alon`, for a list of numbers; `proc`, for a procedure; etc.

- Choose a name that indicates the role the identifier plays: e.g., `num-iterations` or `items_to_sort`.

The first kind of name is appropriate for arguments to very short, simple procedures. For more involved procedures, the second kind can be more useful.

When using long descriptive phrases as names, there are two main styles of separating words.

1. East: In the AI (Artificial Intelligence) research lab at MIT, words in names were traditionally separated by underscores. For example, `my_procedure`.

2. West: At Xerox PARC (back in the day), the word delimiter *en vogue* was capitalization; for instance, `myProcedure`.

In CS 17, we use the East Coast style. Use underscores to separate the words in long phrases in your programs.

**Note:** In class, we sometimes use very short names for a procedure’s arguments, like `n` and `l`, but that’s just because they’re easier to write on the board. You should not use names like these in procedures that you write. In addition to hindering readability, doing so makes it hard to search for all instances of a name. Searching for `n` or `l` is much harder than searching for `num` or `lst`.

4 Abstraction

As you begin to write longer and longer programs, you will often see opportunities to extract a block of code from deep within a procedure into its own procedure, which you can then call from inside the original procedure. This is called writing a *helper procedure*, and there are two situations in which you should do this.
The first situation is when you find that you’re repeating identical or very similar (non-trivial) code. By abstracting out the similarities into a helper procedure, you are making your code easier to read, understand, and debug. Always do this.

The second time to write a helper is when doing so makes your code much easier to read and understand. This is a hard area to define, and takes some experience reading and writing code. But one thing you can look out for is a block of code with well-compartmentalized functionality, that is somehow peripheral to the procedure that encompasses it. Another thing to look out for is a block of code that would be easier to reason about if it were standing alone.

The following sections provide examples of these two situations.

### 4.1 Reducing Repetition

Here is a procedure that calculates the hypotenuse of a right triangle, given its base and its height:

```scheme
define (hypotenuse base height)  
  (sqrt (+ (* base base) (* height height))))
```

This code works, but there are two expressions of equivalent form. We can use a helper procedure to abstract out this repetitive code:

```scheme
define (square x) (* x x)
(define (hypotenuse base height)  
  (sqrt (+ (square base) (square height))))
```

### 4.2 Clarifying Code

The following procedure finds the $y$-intercept of a line through two given points with distinct $x$-coordinates.

```scheme
define (y-intercept posn1 posn2)  
  (- (posn-y posn1)  
      (* (posn-x posn1) (/ (- (posn-y posn2) (posn-y posn1))  
                             (- (posn-x posn2) (posn-x posn1))))))
```

Consider that procedure rewritten with a helper procedure:

```scheme
define (slope posn1 posn2)  
  (/ (- (posn-y posn2) (posn-y posn1))  
      (- (posn-x posn2) (posn-x posn1))))
```

```scheme
define (y-intercept posn1 posn2)  
  (- (posn-y posn1)  
      (* (posn-x posn1) (slope posn1 posn2))))
```

Now we can see more easily that a big part of that code was simply finding the slope of the line between the two points.

**Note:** Helpers are procedures—as such, they should be developed using the design recipe in its entirety. (Yes, that means test cases!)
Avoid writing long (or expensive) expressions more than once  To further iterate on the first point, writing complex expressions more than once can be an error prone endeavor and can make your program ugly. Further, if you write code that computes the same value more than once, your program is inefficient.

Rather than defining a helper procedure, you can also use a `let` expression to bind a complex expression’s value to an identifier. Doing so has the added benefit of documenting the intent of the complicated expression with a well-chosen identifier name—which means less commenting.

5 Comments

You should strive to write self-explanatory code. That is, ideally, your code should not require comments beyond what is required by the design recipe. But we do not live in an ideal world. So we give you the following rule of thumb: before commenting your code, think things through again. Is there a way to rewrite your code so that it is closer to being self-explanatory? If, after repeating this exercise until convergence, your code is still not entirely self-explanatory, then add a helpful comment, in English, so that you and other users of your code can make sense of any hidden complexities that remain.

Here are some guidelines on how, where, and when to comment:

1. Comments that state the obvious are a travesty to programmers. Comments should state only the non-obvious.
   Here is an example of a useless comment:

   ```
   1 + x (* Add 1 to x *)
   ```

2. Comments generally go above the code they reference.
   Comment like this:

   ```
   (* sums a list of integers *)
   let sum = List.fold_left ( + ) 0
   ```

   Not like this:

   ```
   let sum = List.fold_left ( + ) 0
   (* sums a list of integers *)
   ```

   Sometimes, for improved readability, it may be preferable to insert a short comment at the end of a line of code, as in the (* Add 1 to x *) example.

3. Judicious choices of identifier names can frequently obviate the need for comments.

6 Conclusion

The conventions described in the CS 17 style guides are intended to help you and others read, understand, and ultimately, debug your code more easily. If you are ever unclear about whether there is a preferred style, seek advice from a TA.
People almost never write code only for themselves. (Even when you think you might be doing so, a reason to share can crop up many years later!) Since code is something you will likely share with your friends and colleagues, it behooves you to follow the language conventions of whatever language you find yourself programming in.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form:

http://cs.brown.edu/courses/cs017/feedback