CSG111 Style Guide

Fall 2007

Programs must be written for people to read,
and only incidentally for machines to execute.
–Abelson & Sussman, *SICP*

Always code as if the guy who ends up maintaining your code
will be a violent psychopath who knows where you live.
–John F. Woods

Introduction

Your CSG111 assignments are graded on two independent criteria: Correctness and style. Sure, you could conceivably turn in a program that works perfectly but gets few style points. In practice, this doesn't happen. Poor style makes it more hard to write correct code, harder to maintain it, and nearly impossible to grade it.

Since you are writing in languages that may be new to you, such as Scheme, it's important to be aware of idiomatic style in those languages, as it may be different from your usual language. If you are ever in doubt about a point of style, please ask.

Some fine print We might show you some examples the deviate from the following guidelines. We apologize; please tell us if we do this! We promise to tell you when you do.

This document is not comprehensive. There are always new and creative ways to write bad code. If something seems fishy, check.

General guidelines

Many elements of style are the same in all programming languages (or even in all forms of communication). In this section, we discuss style guidelines that should be relevant to any language, Scheme and DemeterJ included. We then discuss points of Scheme-specific style.

Correctness, readability, simplicity, efficiency

Most important is that your code is correct, but readability is a close second. A reader of your code should be able to grasp your code and form a reasonable expectation of its behavior without executing it.

- Your code should implement easy-to-understand, concise solutions to the problems given.
- Naming is important. Use function and variable names that make sense. As a rule of thumb, the larger the scope of a name, the more descriptive it need be. Global variables need long names, but arguments for one-shot `lambda`s probably don't.

  Example 1(a). Bad:

  ```
  (define convert
    (lambda (w)
      (... (lambda (each-whatsit-thingy) ...) ...))
  ```

  Example 1(b). Good:

  ```
  (define whatsit->widget
    (lambda (whatsit)
      (... (lambda (x) ...) ...))
  ```

- Magic numbers are opaque and difficult to maintain. Define a variable or a constant instead. (Zero and 1 are usually not magic numbers.)
- Choose appropriate algorithms and data structures. While maintaining reasonable efficiency and correctness, prefer the simple to the complex.

  Example 2(a). Using an $O(\log n)$ red-black tree may be faster than an $O(n)$ association list, but the list is simpler; only use the tree if you need the speed-up (or if we tell you to).

  Example 2(b). Building a list in order with repeated `append`s is $O(n^2)$ in the length of the list. Building a list in reverse and then reversing it when you're done is $O(n)$. The latter isn't much harder and is a big win.

Copy and paste makes waste

Omit needless code. Factor things out whenever possible (and reasonable).

Example 3(a). Wasteful:

```
if ( denominator == 0 ) {
     printf("Division␣yields␣the␣following␣result:␣");
     printf("Division␣by␣zero\n");
} else {
     printf("Division␣yields␣the␣following␣result:␣");
     printf("%d", numerator/denominator);
}
```

Example 3(b). Thrifty:

```
printf("Division␣yields␣the␣following␣result:␣");
```

```
if ( denominator == 0 ) {
    printf("Division␣by␣zero\n");
} else {
    printf("%d", numerator/denominator);
}
```

Consider extracting code that appear multiple times into a function.

- Getting a particular functionality in one place means you can't update it in one place and forget to in other places.
- Repeated code leads to repeated bugs.
- Moving your code into a separate function allows you to isolate and test it.
- Do not reinvent functionality provided by the language. You do not need to memorize the entire library, but you should be familiar with what is available and know how to search for it.

Comments

Comment your code. The utility curve for comments is ∩-shaped: Too few and it's not clear what your code is for, but too many can be distracting.

Comments should most often explain *what* a procedure will do, less often *why* a certain choice was made, and occasionally *how* it does it. You should assume your reader knows the language you are using.

Example 4(a). Don't paraphrase:

```
(car lst)                    ; take the car of lst
```

Example 4(b). Do explain the unexpected:

```
(lambda (a b) (frob b))      ; don't frob a! (for now)
```

Comments on the same line as code must be extremely short. Never write multiple-line comments to the right of your code. If you want to explain something in more than a short phrase, put it in a comment on the line above the code; leave a blank space before the comment line but not after it.

Every function or method must be commented with a description of its interface. This should explain how to use the function, but not *necessarily* how it works.

Example 5(a). Good:

```
;; count-tens: [List-of Numbers] -> Number
;; To count the number of times 10 shows up in a list.
;; Examples:
;;   (count-tens '(1 2 3 4)) ==> 0
;;   (count-tens '())        ==> 0
;;   (count-tens '(5 10))     ==> 1
;;   (count-tens '(10 3 10)) ==> 2
(define count-tens
  (lambda (lst)
    ; Implementation doesn't matter!  We know what it does from the
    ; contract above.
```

Example 5(b). Still good:

```
// Count the number of 10s in the collection:
int countTens() {
    // ...
```

- Many functions are simple enough that describing the algorithm is unnecessary. However, you must always have at least an explanation of the function's purpose and a few examples.
- If a function's implementation requires explanation, write what it's for first. Then skip a line, and *then* write how it works.
- When you modify any code, ensure that the comments remain correct. Incorrect comments can be worse than no comments.

Formatting and consistency

Regardless of the language, your code should be well-aligned in an easy-to-read fashion. Do not turn in any code or text with lines over 80 characters.[1]

Formatting should be consistent throughout the program.

- Indent your code to show structure.
- Break up long lines in a readable way.
- Consistently space tokens.

Write direct code

Avoid complicated expressions where simple ones suffice.

Example 6(a). Don't check a boolean value in order to return a boolean value: `(if (pred? thing) #t #f)`.

Example 6(b). Do use the boolean value directly: `(pred? thing)`.

Example 7(a). Don't string out lots of conditionals: `(if a (if b c d) d)`.

Example 7(b). Do use `and` and `or`: `(if (and a b) c d)`.

[1]Otherwise, your assignments will contain linewraps when printed, and linewraps make them hard to read. If you need help getting your editor to enforce this, please ask.

Scheme

Scheme has some unique stylistic conventions that ought to be respected. Some make obvious sense given the language, but others seem a bit funny to the uninitiated.

Awkward things

Example 8(a). Don't quote numbers or booleans: `'#f '9`.

Example 8(b). Do quote symbols and lists of things:

```
(quux 'a 'b 'c)
'(3 5 7 (3 3) 11)
'(red green blue)
```

Example 9(a). Don't create a singleton list just to prepend it: `(append (list a) lst)`.

Example 9(b). Do use `cons`: `(cons a lst)`.

Don't use nested `if`s when you can use `cond`.

Example 10(a). Drab:

```
(if a b
  (if c d
    (if e f
      g)))
```

Example 10(b). Sparkling:

```
(cond
  (a    b)
  (c    d)
  (e    f)
  (else g))
```

Spacing and indentation

Multiple s-expressions on the same line should be separated by exactly one space character. (Lining up a `cond` or a `let` supersedes this rule.)

Example 10(c). Naughty: `(a(b) (c  (d)) e)`.

Example 11(a). Nice: `(a (b) (c (d)) e)`.

Parentheses should not be on lines by themselves; the first and last items of a list should not be separated from the enclosing parentheses.

Example 12(a). Gag!:

```
(cond
      [(null? x)
      '()
     ]
           [(pair? x) 'moo]
[else
'hello
]
)
```

Example 12(b). Lack of gagging:

```
(cond
  [(null? x)   '()]
  [(pair? x)   'moo]
  [else        'hello])
```

Choose reasonable line-breaks (and indentation) within the structure of your code to help set off the different pieces. DrScheme can help you with this; so can a good general purpose text editor such as Vim or Emacs. They don't generally know where to break lines, but if you figure that out then they all know how to indent pretty well.

Identifiers

Example 13(a). Don't use underscores: `read_bigrams`.

Example 13(b). Nor mixed capitalization: `readBigrams`.

Example 13(c). Do separate words with hyphens: `read-bigrams`.

Many languages have their own identifier conventions. This convention applies to Scheme and Lisp, but not to C++ or Java, for example.

Things Not to Use

Some features of Scheme that are common in the wild are still not suitable for CSG111.

Side effects The use of `set!`, `set-cdr!` and other mutation operators is not allowed on any CSG111 Scheme assignment.

Named `let` Named `let` (often written `(let loop (()` can be very confusing if not used carefully. Use recursion instead.