Style Rules

In addition to basic structure of the design recipes, there are a number of small details that matter when designing programs.

These rules, or style conventions, are common through the world of software development. Different communities have their own conventions, so what's considered proper for one language is not proper for another. Sometimes, even within a programming language, one group of users will use different conventions than another.

And so it is with our course. We have our own set of style conventions which we ask you to follow.

Notation and Naming Conventions

1. **Comments that are a permanent part of the program should have TWO semi-colons followed by a space.**
   - These always go at the beginning of the line.
   - These include signatures, purpose statements, etc.
   - Eg.

```
;; Number -> Number
;; multiples number n by 2
```

2. **Comments that are temporary should have ONE semi-colon.**
   - These are comments that would be deleted in a real program.
   - These include commenting out stubs.
   - Eg.

```
;(define (double n)  0)  ; this is the stub
```

- Comments that are at the end of a line should also have 1 semi-colon, also as shown above.

3. **Function signatures should be formatted this way:**

- `WorldState -> Image`

4. **Use names that make sense with respect to the problem.**

- Eg. Good names for a variable representing a count of some sort include `count`, `cnt` or possibly `c`.

5. **Do not use _ in names, use - instead**

- Eg. `render-body` rather than `render_body`

6. **Data definition names (type names) should use upper camel case**

- Eg. `WorldState` rather than `world-state`, `world_state` or `worldState`.

7. **Function names and parameter names should be hyphenated and not capitalized**

- Eg. `posn-x` or `tock-ball`.

8. **Functions that produce Boolean should have names that end in ?**

- Eg. `pass?` or `aisle?`

9. **Constant names should be in ALL-CAPS.**

- Eg. `HEIGHT`, `STOP-WHEN`.

Design and Layout Choices

1. **No line should span more than 80 characters.**

- The bottom right of Dr. Racket shows how long a line is.
- Break lines at those points suggested by examples you see on the websites.
- Cond clauses, and the three parts of an if statement, should be on all separate lines.

2. **Your code should be indented using the conventions seen in our examples**

- Ctrl+I on PC, or command+I on Mac, are shortcuts that will automatically indent.

3. **Spaces where they belong**

- Use spaces in a manner consistent with the examples you see in lectures, videos etc.
- Eg.

```
;; This is good
(define (foo n)
  (cond [(odd? n) 1]
        [(even? n) 2]))
;; This is not
(define(foo n)
  (cond[ (odd? n)1]
       [ (even? n)2]))
```

4. **No dangling parenthesis**

- No parentheses left dangling on a line without any other text.
- Eg.

```
(define (double n)
        (* 2 n)
) ; This is a dangling parenthesis
```

5. **No function should span more than five to ten lines.**

- If it does, reconsider your interpretation of the "one task, one function" guideline.
- This rule is relaxed when we learn `local`

6. **In the functions section of a program, the most important functions should be first, and the least important ones last**

- For world programs the `main` function should come first, followed by the `big-bang` handler functions.
- NOTE: You don't have to design the functions in this order, you just have to arrange the program this way.

7. **Use `cond` rather than `if` when the template of a function is handling multiple cases of a one of data definition.**
8. **Don't leave extra examples of making the program do something lying around- use `check-expects` instead.**

A Small Comprehensive Example

Good

```
;; String -> String
;; Produce a greeting by adding
"Hello" before a string.
(check-expect (greet "World!")
"Hello World!")
(check-expect (greet "goodbye")
"Hello goodbye")
(check-expect (greet "loneliness")
"Hello loneliness")

;(define (greet o)  ; stub
;  "a")

(define (greet o)
  (string-append "Hello" " " o))
```

Bad

```
; string -> string

; (greet "foo") produces "Hello
foo"


(define (greet str)
              ;; put "Hello "
before string
              (string-append "Hello
" str)
)


(check-expect (greet "World!")
"Hello World!")
(check-expect (greet "goodbye")
"Hello goodbye")
(check-expect (greet "lonliness")
"Hello lonliness")

;(define (accio o)  ; stub
;  "a")

(greet "World!")
(greet "goodbye")
(greet "lonliness")
```