Scheme Style Guide for First-Year CS (2008)

The code you submit for assignments, as with all code you write, can be made more readable and useful by paying attention to style. This includes the placement of comments, whitespace, indentation, and choice of variable and function names. None of these things affect the execution of a program, but they affect its readability and extensibility. There isn't a single strict set of rules that you must follow; just as in writing English prose, the goal is communication, and you need to think of the needs of the reader. This is especially important when the reader is assigning you a mark.

1 General Guidelines

The examples in course presentation slides and handouts are often condensed to fit them into a few lines; you should not imitate their style for assignments, because you don't have the same space restrictions. The examples in the "How To Design Programs" textbook are more appropriate, particularly those illustrating variations on the design recipe, such as Figure 3 in Section 2.5, Figure 11 in Section 6.5, and Figure 17 in Section 17.2. At the very least, a function should come with contract, purpose, examples, definition, and tests. After you have learned about them, and where it is appropriate, you should add data definitions. See the section below titled "The Design Recipe" for further tips.

You should prepare one file for each question in an assignment, containing all code and documentation. The file for question 3 of Assignment 8 should be called `a8q3.ss` or `a8q3.scm`, and all the files for Assignment 8 should be in the folder/directory `a8`. If the question asks you to write more than one function, the file should contain them in the order specified by the assignment. Helper functions should be put with the assignment function they are helping, but whether they are placed before or after is a judgement you will have to make depending on the situation. In working with several functions in one file, you will find it useful to use the Comment Out With Semicolons and Uncomment items in DrScheme's Scheme menu to temporarily block out successful tests for helper functions. Note: never include Comment Boxes, other special boxes, or images in your submissions, and do not cut-and-paste from the Interactions window into the Definitions window, as these will render your submissions unmarkable.

The file for a given question should start with a header, like the one below. The purpose of the header is to assist the reader.

```
;;
;; ***********************************************
;;
;; Assignment 13, Question 3
;; (solving the problem of world hunger)
;;
;; ***********************************************
;;
```

2 Block Comments and In-Line Comments

Anything after a semicolon on a line is treated as a comment and ignored by DrScheme. Functions are usually preceded by a block comment, which for your assignments will contain the contract, purpose, and examples. Block comments should be indicated by double semicolons at the start of a line, followed by a space.

```
;; distance: posn posn → num
;; Computes the Euclidean distance between posn1 and posn2
;; Example: (distance (make-posn 1 1) (make-posn 4 5)) ⇒ 5
```

You may or may not choose to put a blank line between the block comment and the header of the function (probably for longer headers it is appropriate), but there should be a blank line between the end of a function and the start of the next block comment. In your early submissions, you shouldn't need to put blank lines in the middle of functions; later, when we start using local definitions, they may be appropriate.

Use "in-line" comments sparingly in the middle of functions; if you are using standard design recipes and templates, and following the rest of the guidelines here, you shouldn't need many additional comments. Any such comment can either be put on its own line, or tacked onto the end of a line of code, providing it fits.

3 Indentation and Layout

Indentation plays a big part in readability. It is used to indicate level of nesting, to align related subexpressions (e.g. arguments of a function), and to make keywords more visible. DrScheme's built-in editor will help with these. If you start an expression (*my-fun* and then hit enter or return, the next line will automatically be indented a few spaces. However, DrScheme will never break up a line for you, and you can override its indentation simply by putting in more spaces or erasing them. DrScheme also provides a menu item for reindenting a selected block of code and another for reindenting an entire file.

When to hit enter or return is a matter of judgement. At the very least, don't let your lines get longer than about 70 characters. You don't want your code to look too horizontal, or too vertical.

```
;; don't do this
(define (squaresum x y) (+ (* x x) (* y y)))
```

```
;; don't do this, either
(define
  (squaresum x y)
    (+
      (*
        x
        x)
      (*
        y
        y)))
```

```
;; this is all right
(define (squaresum x y)
  (+ (* x x)
     (* y y)))
```

If indentation is used properly to indicate level of nesting, then closing parentheses can just be added on the same line as appropriate, and you will see this throughout the textbook and presentations. Some styles for other programming languages expect you to put closing parentheses or braces by themselves on a separate line lined up vertically with their corresponding open parenthesis or brace, but in Scheme this tends to affect readability.

If you find that your indentation is causing your lines to go over 70 characters, consider breaking out some subexpression into a helper function, but do this logically rather than cutting out an arbitrary chunk.

For conditional expressions, you should place the keyword **cond** on a line by itself, and align not only the questions but the answers as well (provided that they are short; if not, put them on a separate, indented line).

```
(cond
    [(null? lon)  empty]
    [else         (rest lon)])
```

```
(cond
  [(zero? n) 0]
  [(= n 1)   1]
  [else
    (* n (fact (- n 1)))])
```

4 Variable and Function Names

Try to choose names for variables and functions that are descriptive, not so short as to be cryptic, but not so long as to be awkward. It is a Scheme convention to use lower-case letters and hyphens, as in the identifier *top-bracket-amount*. (DrScheme distinguishes upper-case and lower-case letters by default, but not all Scheme implementations do.) In other languages, one might write this as `TopBracketAmount` or `top_bracket_amount`, but try to avoid these styles in Scheme.

You will notice some conventions in naming functions: predicates that return a Boolean value usually end in a question mark (e.g. *zero?*), and functions that do conversion use a hyphen and greater-than sign to make a right arrow (e.g. *string->number*). This second convention is also used in contracts to separate what a function consumes from what it produces. A “double right arrow” (which we use to indicate the result of partially or fully evaluating an expression) can be made with an equal sign and a greater-than sign. In the typesetting in this document and in the presentations, we have fonts with single and double arrow symbols, which you can see in our contracts and comments, but DrScheme isn’t equipped with them.

5 Summary

- Use block comments to separate functions
- Use the design recipe
- Comment sparingly inside body of functions
- Indent to indicate level of nesting and align related subexpressions
- Avoid overly horizontal or vertical code layout
- Use reasonable line lengths
- Align questions and answers in conditional expressions where possible
- Choose meaningful identifier names and follow the naming conventions used in the textbook and in lecture
- Do not include Comment Boxes or images.

6 The Design Recipe

We hope you will use the design recipe as part of the process of working out your solutions to assignment questions. If you hand in only code, even if it works perfectly, you will earn only a fraction of the marks available. We depend on the visible evidence of the design recipe components to understand your code.

The design recipe comes in several variants, depending on the form of the code being developed. As we learn about new language features and ways of using them, we also discuss how the design recipe is adapted. Consequently, not everything in this section will make sense on first reading. We suggest that you review it before each assignment.

The design recipe for a function starts with contract, purpose, and examples. You should develop these before you write the code for the function.

```
;; distance: posn posn → num
;; Computes the Euclidean distance between posn1 and posn2
;; Examples:
;; (distance (make-posn 1 2) (make-posn 1 2)) ⇒ 0
;; (distance (make-posn 1 2) (make-posn 1 4)) ⇒ 2
;; (distance (make-posn 1 1) (make-posn 4 5)) ⇒ 5

(define (distance posn1 posn2)
  ...)
```

The contract contains the name of the function, the types of the arguments it consumes, and the type of the value it produces. We are being more strict about types than the textbook. There are built-in types (*num*, *image*, *boolean*, *char*, *string*, *symbol*), structure types (*posn* is the only built-in structure type, but *cons* and *empty* can be considered as such, and the name of any structure defined by **define-struct** is a type), and compound types (formed with (*listof* ...) and (*union* ...)). Where the textbook uses a type like *list-of-numbers*, we are being more careful and using the type (*listof num*). Since a contract is a comment, errors in contracts will not be caught by DrScheme. As this is not the case with statically typed languages such as Java, it is good practice to write them correctly.

The purpose is a brief one- or two-line description of what the function should compute. Note that it does not have to be a description of how to compute it; the code shows how. It is a good idea to try to use the names of the parameters in the purpose, so as to make it clear what they mean (choosing meaningful parameter names helps also).

The examples should be chosen to illustrate the uses of the function and to illuminate some of the difficulties to be faced in writing it. Many of the examples can be reused as tests (though it may not be necessary to include them all). The examples don't have to cover all the cases that the code examines; that is the job of the tests, which are designed after the code is written.

Next comes the header and body of the function itself. You'd be surprised how many students have lost marks in the past because we asked for a function *my-fun* and they wrote a function *my-fn*. They failed all of our tests, of course, because they didn't provide the function we asked for.

To avoid this situation, use the provided "interface" files, such as the file `assninterface3.ss` for Assignment 3. These contain the headers of the functions asked for, and perhaps definitions of some structures. If you use these as a starting point, you are less likely to misspell key identifiers.

Finally, we have tests. You should make sure that the tests exercise every line of code, and furthermore, that the tests are directed: each one should aim at a particular case, or section of code. Some people write tests that use a large amount of data; this is not necessarily the best idea, because if they fail, it is difficult to figure out why. Others write lots of tests, but have several tests in a row that do essentially the same thing. It's not a question of quantity, but of quality. You should design a small, comprehensive test suite.

It is tempting, when creating tests, to apply your function to some value, cut the result from the Interactions window, and paste it into a test expression in the Definitions window. Please don't. First of all, this renders the test useless, as it is guaranteed to pass even if your function is incorrect. But also, as we said before, what the Interactions window produces is not necessarily plain text, and this action may interfere with automated marking of your work.

Later on in the course, there are other components to the design recipe, such as data definitions and templates. We usually discuss these briefly in class, but the book goes into more detail. If you keep up with lecture attendance and reading, you will know how to adapt the design recipe for each new assignment.

7 A Sample Submission

```
;;
;; **************************************************
;;
;; CS 135 Assignment 17, Question 1
;; John A. MacDonald, 20123456
;; (Canadian tax computation for 2005)
;;
;; **************************************************

;;
;; Defined constants
;;
```

bpi is the ith breakpoint, at which the rate changes from ratei to rate(i+1) we've left out bp0 which is zero

```
(define bp1 37178)
(define bp2 74357)
(define bp3 120887)
```

ratei is the rate applicable in the interval [bp(i-1),bpi)

```
(define rate1 15/100)
(define rate2 22/100)
(define rate3 26/100)
(define rate4 29/100)

;; tax-calc: num num num num → num
;; computes base + (high-low)*rate
;; examples: (tax-calc 0 0 100 0.1) ⇒ 10
;; (tax-calc 550 1000 1200 0.2) ⇒ 590

(define (tax-calc base low high rate)
  (+ base (* (− high low) rate)))

;; tax-calc tests (using DrScheme v4.1 feature)
(check-expect (tax-calc 0 0 100 0.1) 10)
(check-expect (tax-calc 550 1000 1200 0.2) 590)
```

```
;; basei is the base amount for interval [bpi,bp(i+1))
;; that is, tax payable at income bpi

(define base1 (tax-calc 0 0 bp1 rate1))
(define base2 (tax-calc base1 bp1 bp2 rate2))
(define base3 (tax-calc base2 bp2 bp3 rate3))

;; tax-payable: num → num
;; computes income tax given salary
;; examples: (tax-payable 1000) ⇒ 150
;; (tax-payable 50000) ⇒ 8508.35

(define (tax-payable salary)
  (cond
    [(< salary 0)    0]
    [(< salary bp1) (tax-calc 0 0 salary rate1)]
    [(< salary bp2) (tax-calc base1 bp1 salary rate2)]
    [(< salary bp3) (tax-calc base2 bp2 salary rate3)]
    [else           (tax-calc base3 bp3 salary rate4)]))

;; tax-payable tests
;; (incomplete, should test all intervals and boundary values)

(check-expect (tax-payable 0) 0)
(check-expect (tax-payable 1000) 150)
(check-expect (tax-payable 50000) 8397.54)
(check-expect (tax-payable 382999.92) 101866.6268)
```