CS 2500, Lab 6—Recursion with Style

Programs must be written for people to read, and only incidentally for machines to execute. - SICP

Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live. - c2.com

Part I: Style

Style is important! Programs have remarkably long lifespans. Sometimes they even outlive their creators (!). A program will almost always be modified, extended and repaired countless times by many different people, so it is important that we write code that will be easily readable by other programmers.

If you don't think that's a good excuse, consider that if a tutor can't figure out what your program does, then you won't get full credit. You (the brilliant student you are) want to make your grader's life as easy as possible; don't tempt fate!

Let's look at some guidelines on how to write clear, readable code in DrRacket.

Style Guidelines

Guideline 1: Break Lines

Consider the data definition for `Time`:

```
;; A Time is (make-time Number Number)
(define-struct time (hours minutes))
```

Consider this program:

```
;; Time -> Image
;; Produce an image of the given time, as it appears on a digital clock.
(define(time->text a-time)
   (text(string-append(number->string(time-hours a-time))":"(cond[(< (time-minutes
   a-time)10)"0"]
   [else ""])(number->string(time-minutes a-time)))30'red))
```

How many arguments are there to `text`?
How many `cond` clauses are there? What are the arguments to `string-append`?

This code is a disaster. Because the body is squeezed onto three lines, we cannot answer these questions within a few minutes. If we insert line breaks, the code becomes much more readable.

```
;; Time -> Image
;; Produce an image of the given time, as it appears on a digital clock.
(define (time->text a-time)
    (text (string-append (number->string (time-hours a-time))
                         ":"
                          (cond [(< (time-minutes a-time) 10) "0"]
                                [else ""])
                          (number->string (time-minutes a-time)))
          30
          'red)
```

With this code, it is easy to see the three arguments to text , the four strings being appended, and the two cond clauses. In general, try to break long lines by

- Giving each `cond` clause its own line. [Always a good idea.]
- If a `cond` question or answer is long, then start the answer on the next line. [Often a good idea.]
- Giving each function argument its own line. [Less common, but `string-append` is a typical offender.]

If these formatting hints do not help, consider designing and using a helper function to make the code more compact/readable. For example, in `time-> text` above, it's probably a good idea to factor out the `(string-append ..)` expression as a separate helper function.

Every line of code should be no more than 80 characters long. DrRacket can help you with this Can you find where it indicates the current column of the cursor?

Exercise 1: Rewrite `time -> text` by developing and using a helper function that computes the `(string-append ...)` portion of the body. Be sure to invent a good name for this helper function.

Guideline 2: Indentation

Consider the following program:

```
;; LOS -> Number
;; Determine how many symbols are in a-los
(define (count a-los)
(cond
[(empty?  a-los) 0]
[(cons?  a-los)
(+ 1 (count (rest a-los)))]))
```

This is not indented properly. Copy and paste this code into DrRacket. Then select the code and hit the Tab key. DrRacket will indent all the selected code automatically! (Note that this will not

help you with long lines and/or line breaks.

Make good use of this feature as you develop your programs. Also note that the Tab key can be used to automatically indent the line the cursor is currently on. Indentation is a very important factor of readability because it denotes the structure of the program at a glance. And we can't grade what we can't read!!

Note: When you use the automatic indentation you may notice it seems wrong... this usually means that your program is mis-parenthesized. Move the cursor through your code and use the grey highlighting to be sure that your parentheses are matched as you intend.

Guideline 3: Parentheses Layout

Let's reconsider count from above. The indentation is technically correct, but the parentheses are arranged a poorly:

```
;; LOS -> Number
;; Determine how many symbols are in a-los
(define (count a-los)
   (cond
      [(empty?  a-los) 0
      ]
      [(cons?  a-los
        )
        (+ 1 (count (rest a-los)
             )
        )
      ]
   )
)
```

A programmer who arranges their parentheses like this is probably trying to use the vertical alignment of the open and closing parentheses to visually determine the code structure. It is much easier to compress the closing parentheses together, and then eyeball the program structure using its indentation. When you need to match parentheses visually, use DrRacket's grey highlighting instead.

```
;; LOS -> Number
;; Determine how many symbols are in a-los
(define (count a-los)
   (cond
      [(empty?  a-los) 0]
      [(cons?  a-los)
        (+ 1 (count (rest a-los)))]))
```

Proper indentation and parentheses placement render the parentheses and brackets nearly invisible to the trained eye.

Part II: Recursion

We want you all to be able to write recursive functions as easily as `(list 1 2 3)`, so we're going to practice.

Please write each of the requested functions from scratch.

Exercise 2: Design a function, `string-of`, that takes a positive number (n) and a string, and returns a string that contains the given string repeated n times, separated by a space.

Examples:

```
(string-of 4 "Test") -> "Test Test Test Test"
(string-of 2 "What") -> "What What"
```

Exercise 3: Using your function above as a helper, create another function, `reducing` that takes a number and a string, and returns a list of strings. Each element of the list is the string returned from `string-of` with a reduced n.

Examples:

```
(reducing 4 "Test") -> (list "Test Test Test Test" "Test Test Test" "Test Test"
"Test")
(reducing 2 "What") -> (list "What What" "What")
```

Exercise 4: Now design the function `lookup` that takes a list of Symbols *los*, and a number n, and returns the nth symbol of the list.

Examples:

```
(lookup (list 'a 'b 'c 'd) 0) -> 'a
(lookup (list 'a 'b 'c 'd) 2) -> 'c
```

Exercise 5: Next design the function `replace` that takes a list of Symbols *los*, a symbol s, and a a number n. The function returns *los* with the nth symbol replaced with s.

Examples:

```
(replace (list 'a 'b 'c 'd) 'new 2) -> (list 'a 'b 'new 'd)
(replace (list 'a 'b 'c 'd) 'yay 0) -> (list 'yay 'b 'c 'd)
(replace (list 'a 'b 'c 'd) 'end 3) -> (list 'a 'b 'c 'end)
```

Exercise 5: Consider the following problem:

Given two lists of strings, return a list of strings that contains all combinations of elements from the first list with elements from the second list.

Let's call the function `all-comb`. Here's an example:

```
;; This example
(all-comb (list "Student:  " "Faculty:  ") (list "Mr." "Ms." "Mrs."))

;; Results in...
(list "Student:  Mr." "Student:  Ms." "Student:  Mrs."
      "Faculty:  Mr." "Faculty:  Ms." "Faculty:  Mrs.")
```

How can we design such a function? Well, lets start with a smaller problem. How can we take a string, *s*, and a list-of-strings, *los*, and produce a list that contains the strings from *los* with *s* on the front.

Go for it!! Call this function `all-comb-help`.

Here's an example:

```
(all-comb-help "A" (list "B" "C" "D")) -> (list "AB" "AC" "AD")
```

Now... how can you put the helper function to work to solve the entire problem? Ask a TA/Tutor if you need help. *Hint:*— you can use append (or define your own for practice), which appends two lists.

Part III:Common Mistakes)

If debugging is the process of removing bugs, then programming must be the process of putting them in. - Edsger W. Dijkstra

The design recipe is a powerful tool, but it only works when used properly. The staff has identified a number of common errors that have been showing up on homeworks and the midterm. Let's go over a few of them in detail.

Violating the Contract

Writing a contract is only useful if your function satisfies the contract. Consider the following example:

```
;; only-evens :  list-of-numbers -> list-of-numbers
;; to create a list containing only the even numbers in a-list-of-nums
(define (only-evens a-list-of-nums)
   (cond [(empty?  a-list-of-nums)
          0]
         [(even?  (first a-list-of-nums))
          (cons (first a-list-of-nums)
              (only-evens (rest a-list-of-nums)))]
         [else
           (only-evens (rest a-list-of-nums))]))
```

There is a bug in the definition above. The first branch of the `cond` clause is violating the contract. 0 is a number, not a list of numbers. In its place we should be using empty. By carefully making sure each branch of our `cond` statements satisfy our contract we can avoid such errors.

No Data Definition

A contract is only as useful as the information it provides. If we fail to fully specify the kinds of data our functions consume and produce then we defeat the purpose of the contract. Consider the following example.

```
;; name->greeting :  name -> greeting
;; to create a greeting from the provided name
```

It would make sense to assume that name and greeting are strings. We could write the following function for the contract:

```
(define (name->greeting name)
   (string-append "Hello, "
                  name
                  "!"))
```

But what if some other part of our program thought that name was a structure, `(define-struct name (first last))`, an equally reasonable assumption? We can only avoid such errors by providing data definitions for each kind of data our functions consume and produce.

The following data definition clears up the ambiguity:

```
;; name->greeting :  Name -> string
;; to create a greeting from the provided name
;; a Name is a structure:  (make-name first last) where first and last
```

```
;; are strings.
```

Practice

Exercise 8: Identify whether there is a contract violation or a lack of a data definition. Correct the problem, fixing the code if necessary.

```
;; a Dog is (define-struct dog (name age breed)) where name is a
;; string, age is a number and breed is a symbol
(define-struct dog (name age breed))

;; dogs-older-than :  list-of-Dogs number -> list-of-Dogs
;; to list all the dogs older than age
(define (dogs-older-than dogs age)
   (cond [(empty?  dogs) empty]
         [(> (dog-age (first dogs)) age)
             (cons (dog-name (first dogs))
             (dogs-older-than (rest dogs) age))]
         [else (dogs-older-than (rest dogs) age)]))
```

Exercise 9: Identify whether there is a contract violation or a lack of a data definition. Correct the problem, fixing the code if necessary.

```
;; numbers-between :  list-of-numbers -> number-range
;; to list all the numbers between the low number and the high number
(define (numbers-between low high)
   (cond [(> low high) empty]
         [else (cons low
               (numbers-between (add1 low)
                                high))]))
```

Exercise 10: Challenge: Identify whether there is a contract violation or a lack of a data definition. Correct the problem, fixing the code if necessary.

```
;; pairify :  list-of-any -> list-of-list
;; to group the elements of the input list into a list of two-element lists
(define (pairify a-list)
   (cond [(or (empty?  a-list)
          (empty?  (rest a-list)))
           empty]
         [else (cons (cons (first a-list)
                           (first (rest a-list)))
                     (pairify (rest (rest a-list))))]))
```