Variable-Arity Procedures

Summary: We explore a variety of issues with variable-arity procedures, including effects of the different formats and detailed consideration of the sample procedures from the corresponding reading.

Contents:

- Exercises
  - Exercise 1: Basic Experiments
  - Exercise 2: Experiments with `display-line`
  - Exercise 3: Extending `display-line`
  - Exercise 4: Experiments with `display-separated-line`
  - Exercise 5: Acronyms
  - Exercise 6: A Clicker
  - Exercise 7: Multiple Separators
  - Exercise 8: Unused Letters

Exercises

Exercise 1: Basic Experiments

Consider the following procedures:

```
(define proc1
  (lambda (stuff)
    stuff))
(define proc2
  (lambda stuff
    stuff))
(define proc3
  (lambda (more stuff)
    stuff))
(define proc4
  (lambda (more . stuff)
    stuff))
(define proc5
  (lambda (even more . stuff)
    stuff))
```

a. What do you expect the result of `(proc n 1)` to be for each variant?

b. Check your answer experimentally.

c. What do you expect the result of `(proc n 1 2)` to be for each variant?
d. Check your answer experimentally.

e. What do you expect the result of \((\text{proc}\ n\ 1\ 2\ 3)\) to be for each variant?

f. Check your answer experimentally.

**Exercise 2: Experiments with display-line**

Here is the `display-line` procedure from the reading.

```scheme
;;; Procedure:
;;;   display-line
;;; Parameters:
;;;   val1 ... valn, 0 or more values
;;; Purpose:
;;;   Displays the strings terminated by a carriage return.
;;; Produces:
;;;   [Nothing]
;;; Preconditions:
;;;   0 or more values given as parameters.
;;; Postconditions:
;;;   All of the values have been displayed.
;;;   The output is now at the beginning of a new line.
(define display-line
 (lambda arguments
  (let kernel ((rest arguments))
   (if (null? rest)
      (newline)
      (begin
       (display (car rest))
       (kernel (cdr rest)))))))
```

a. Try out some other calls to `display-line` to check what it prints. For example, try the following:

- `(display-line "going" "going" "gone")`
- `(display-line "countdown:" 5 4 3 2 1 "done")`
- `(display-line)` ;; apply `display-line` to no arguments

b. Explain your results.

**Exercise 3: Extending display-line**

The current version of `display-line` prints all text together without spaces. Modify the code so that one space is printed between any two adjacent values supplied as arguments to `display-line`. For instance, after your modifications, the example from the reading will change. It will now be ...

```scheme
> (display-line "+-" "Here is a string!" "+-+)
--- Here is a string! ---
```

You may not use `display-separated-line` in your answer to this question, although you may refer to it for ideas.
Exercise 4: Experiments with display-separated-line

Here is the display-separated-line procedure from the corresponding reading.

;;; Procedure:
;;;   display-separated-line
;;; Parameters:
;;;   separator, a string
;;;   val1 ... valn, 0 or more additional values.
;;; Purpose:
;;;   Displays the values separated by the separator and followed
;;;   by a carriage return.
;;; Preconditions:
;;;   The separator is a string.
;;; Postconditions:
;;;   All the values have been displayed.
;;;   The output is now at the beginning of a new line.
(define display-separated-line
  (lambda (separator . parameters)
    (if (null? parameters)
      (newline)
      (let kernel ((rest parameters))
        (display (car rest))
        (if (null? (cdr rest))
          (newline)
          (begin
            (display separator)
            (kernel (cdr rest))))))))

a. What do you think should happen if you invoke display-separated-line without giving it any arguments? Verify your results experimentally.

b. What do you think should happen when you give it only one argument? Check your answer experimentally.

c. What do you think should happen when you give it two arguments? Check your answer experimentally.

d. What do you think should happen when you give it three arguments? Check your answer experimentally.

Exercise 5: Acronyms

Develop a procedure, (acronym str0 ... strn), that takes any number of non-empty strings as arguments and returns one string consisting of the initial characters of those strings, thus:

> (acronym "Mothers" "Against" "Drunk" "Driving")
"MADD"
Exercise 6: A Clicker

Define and test a procedure, clicker, that takes one or more arguments, of which the first must be an integer and each of the others must be either the symbol ‘up’ or the symbol ‘down’. Clicker should start from the given integer, add 1 for each ‘up’ argument, subtract 1 for each ‘down’ argument, and return the result:

> (clicker 17 ‘up ‘up)
19
> (clicker -12 ‘down ‘up ‘down ‘down ‘down)
-15
> (clicker 100)
100

Exercise 7: Multiple Separators

In writing, we often separate the last element of a list using a different separator than for the prior elements. For example, we might separate the all but the last element with commas and the last element with a comma and “and”.

Extend display-separated-line so that it requires two parameters (the default separator and the final separator) and supports as many the client provides.

Exercise 8: Unused Letters

Here’s something kind of fun to do with set-difference: finding the letters not used in a list of words. How do we do that?

- We can convert each word to a list of characters with string->list.
- We can build a list of all characters with (string->list "abcdefghijklmnopqrstuvwxyz").
- We use set-difference to subtract the characters in the words from the list of all characters.
- We use list->string to put everything back together.

a. Write a Scheme expression that makes a string that represents not used in the strings "computers" "are" "sentient" "and" "malicious".

b. We can also compute the opposite set (that is, the characters that are used in a list of strings) by first computing the characters not in those strings (as in step a) and then removing those characters from a list of all characters.

Using this technique, write an expression that computes an alphabetical list of all the characters that are used in the strings "always" "trust" "your" "cs" "professor".

Copyright © 2007 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/2.5/ or send a letter to Creative
Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.