Variable-Arity Procedures

A procedure’s *arity* is the number of arguments it takes. For instance, the arity of the cons procedure is 2 and the arity of the predicate char-uppercase? is 1. You’ll probably have noticed that while some of Scheme’s built-in procedures always take the same number of arguments others have variable arity -- that is, they can take any number of arguments. All one can say about the arity of some procedures -- such as list, +, or string-append -- is that it is some non-negative integer.

Still other Scheme procedures, such as map and display, require at least a certain number of arguments, but will accept one or more additional arguments if they are provided. For example, the arity of map is “2 or more”, and the arity of display is “1 or 2”. These procedures, too, are said to have variable arity, because their arity varies from one call to another.

It is possible for the programmer to define new variable-arity procedures in Scheme, by using alternate forms of the lambda-expression. The simplest takes the following form

\[(\text{lambda} \ arg)\]

In all of the programmer-defined procedures that we have seen so far, the keyword lambda has been followed by a list of parameters -- names for the values that will be supplied by the caller when the procedure is invoked. If, instead, what follows lambda is a single identifier -- not a list, but a simple identifier, not enclosed in parentheses -- then the procedure denoted by the lambda-expression will accept any number of arguments, and the identifier following lambda will name a list of all the arguments supplied in the call.

Here’s a simple example: We’ll define a display-line procedure that takes any number of arguments and prints out each one (by applying the display procedure to it), then terminates the output line (by invoking newline). Note that in the lambda-expression, the identifier arguments denotes a list of all the items to be printed:

```
;;; Procedure:
;;;   display-line
;;; Parameters:
;;;   0 or more values
;;; Purpose:
;;;   Displays the strings terminated by a carriage return.
;;; Produces:
;;;   Nothing
;;; Preconditions:
;;;   None
;;; Postconditions:
;;;   The standards
(define display-line
  (lambda arguments
     (let kernel ((rest arguments))
       (if (null? rest)
```
When display-line is invoked, however, the caller does not assemble the items to be printed into a list, but just supplies them as arguments:

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

> (display-line "ratio = " 35/15)
ratio = 7/3

If the programmer wishes to require some fixed minimum number of arguments while permitting (but not requiring) additional ones, she can use yet another form of the lambda-expression, in which a dot is placed between the last two identifiers in the parameter list. This form looks like the following:

(lambda (arg1 arg2 ... argn . remaining-args)
  body)

All the identifiers to the left of this dot correspond to single required arguments. The identifier to the right of the dot designates the list of all of the remaining arguments, the ones that are optional.

For instance, we can define a procedure called display-separated-line that always takes at least one argument, separator, but may take any number of additional arguments. Display-separated-line will print out each of the additional arguments (by invoking display) and terminate the line, just as display-line does, but with the difference that a copy of separator will be displayed between any two of the remaining values. Here is some sample output:

> (display-separated-line "...
  going...going...gone"

> (display-separated-line ":-" 5 4 3 2 1 'done)
5:-4:-3:-2:-1:-done

> (display-separated-line #\space "+---" "Here is a string!" "---+")
+-- Here is a string!--+

> (display-separated-line (integer->char 9) 1997 'foo 'wombat 'quux)
1997    foo     wombat  quux
; (INTEGER->CHAR 9) is the tab character.

And here is the definition of the procedure:

;;; Procedure:
;;;   display-separated-line
;;; Parameters:
;;;   separator, a string
;;;   zero 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:
;;;   The standard ones
As another example, let’s look at the set-difference procedure, which takes one or more lists \(l_1, l_2, \ldots, l_n\) as arguments and returns a list containing all of the elements of \(l_1\) that are not also elements of any of \(l_2, \ldots, l_n\). For example, the value of

\[
> \text{set-difference (list 'a 'b 'c 'd 'e 'f 'g)}
\]

\[
(\text{list 'a 'e}) (\text{list 'b}) (\text{list 'a 'f 'h})
\]

\[
(\text{c d g})
\]

We get the result because the elements 'c, 'd, and 'g of the first argument are not elements of any of the subsequent list. The set-difference procedure allows the caller to supply any number of lists of values to be “pruned out” of the initial list.

Here’s the definition:

;;; Procedure:
;;; set-difference
;;; Parameters:
;;; initial, a set
;;; other-1 ... other-n, Zero or more additional sets
;;; Purpose:
;;; Removes elements of the additional sets from the original set.
;;; Produces:
;;; The new set whose elements consist of the elements of the first
;;; set that do not appear in the other sets.
;;; Preconditions:
;;; All the sets are represented as lists.
;;; Postconditions:
;;; Does not affect any of the parameters. (Yes, this is a standard
;;; postcondition, but I considered it important to restate since this
;;; procedure is described as "removing" elements from its parameters;
;;; it doesn’t.)
;;; The standards.
(define set-difference
(lambda (initial . others)
  (let kernel ((set initial)
              (remaining others))
    (if (null? kernel) set
        (kernel ((remove (right-section member (car remaining)))
                    set)
            (cdr remaining))))))
In English: Call the initial list initial and collect all of the other arguments into a list called others. Using list recursion, fold over others: In the base case, where remaining is null, just return set. In any other case, separate remaining into its car and its cdr, remove all elements in the car from the set, and then repeat with the cdr.

The dot notation can be used to specify any number of initial values. Thus, a parameter list of the form

\[(first-value second-value . remaining-values)\]

indicates that the first two arguments are required, while additional arguments will be collected into a list named remaining-values.