

Variable-Arity Procedures

Summary: We consider how and why to write procedures that may take different numbers of parameters.

Contents:

- Introduction
- Creating Your Own Variable-Arity Procedures
 - A Simple Example: Counting Parameters
 - Another Example: Displaying a Line
- Enforcing a Minimum Arity
 - An Example: Displaying a Line with Separators
 - An Example: Set Difference
 - Requiring More than One Parameter

Introduction

A procedure's *arity* is the number of parameters it takes. For instance, the arity of the `cons` procedure is two and the arity of the predicate `char-uppercase?` is one. You'll probably have noticed that while some of Scheme's built-in procedures always take the same number of parameters others have *variable arity* -- that is, they can take any number of parameters. 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 `+` and `display`, require at least a certain number of parameters, but will accept one or more additional parameters if they are provided. For example, the arity of `+` is "1 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.

Creating Your Own Variable-Arity Procedures

All of the procedures you've written so far have had a fixed arity. Is it possible to define your own new variable-arity procedures in Scheme? Yes! You can create variable-arity procedures by using alternate forms of the `lambda`-expression.

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 that is not enclosed in parentheses) then the procedure denoted by the `lambda`-expression will accept any number of parameters, and the identifier following `lambda` will name a list of all the parameters supplied in the call.

This variant takes the following form.

```
(lambda ARGS
  BODY)
```

A Simple Example: Counting Parameters

Here's one of the simplest variable-arity procedures, one that simply reports on how many parameters it is called with:

```
(define num-parameters
  (lambda params
    (length params)))
```

As the following examples show, we can call this procedure with different numbers of parameters and still get a result.

```
> (num-parameters)
0
> > (num-parameters 1 2 3 4 5)
5
> (num-parameters (list 1 2 3))
1
```

Another Example: Displaying a Line

Here's a slightly more complicated example: We'll define a `display-line` procedure that takes any number of parameters 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 `parameters` denotes a list of all the items to be printed:

```
;;; 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 parameters
    (let kernel ((rest parameters))
      (if (null? rest)
          (newline)
          (begin
             (display (car rest))
             (kernel (cdr 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 parameters:

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

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

Enforcing a Minimum Arity

If you wish to require some fixed minimum number of parameters while permitting (but not requiring) additional ones, you 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 required parameters. The identifier to the right of the dot designates the list of all of the remaining parameters, the ones that are optional.

An Example: Displaying a Line with Separators

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 parameters. `Display-separated-line` will print out each of the additional parameters (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")
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
;;; 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))))))))))

```

An Example: Set Difference

We often use variable-arity procedures to provide more general versions of fixed-arity procedures. For example, you may recall writing a procedure, `(intersect lst1 lst2)` that computes the intersection of two lists. Let's look at a variant of that procedure, `(set-difference initial others)`, that, given two lists, removes all elements `others` from `initial`. We document it first.

```

;;; Procedure:
;;;   set-difference
;;; Parameters:
;;;   initial, a list
;;;   others, a list
;;; Purpose:
;;;   Removes all elements of others from initial.
;;; Produces:
;;;   newset, a set
;;; Preconditions:
;;;   (none)
;;; Postconditions:
;;;   All values in newset appear in initial.
;;;   No values in newset appear in others.
;;;   All values in initial appear in either newset or others.

```

How do we implement `intersect`? Hmmm ... it sounds like we're going to be removing values from a list. Since we've written procedures that remove elements from a list in the past, we should start by writing a more general, higher-order, `remove` procedure that removes all values for which a predicate holds. (Remember, when you find yourself writing the same code again and again, you should write the higher-order equivalent.)

```

;;; Procedure:
;;;   remove
;;; Parameters:
;;;   pred?, a unary predicate
;;;   lst, a list
;;; Purpose:
;;;   Remove all values from lst for which predicate holds.
;;; Produces:
;;;   newlst, a list.
;;; Preconditions:
;;;   pred? can be applied to every element of lst.

```

```

;;; Postconditions:
;;; (pred? (list-ref newlst i)) is #f for all reasonable i.
;;; Every element of newlst appears in lst.
;;; For all values, v, in lst for which pred? is #f, v appears
;;; in newlst.
(define remove
  (lambda (pred? lst)
    (cond
      ((null? lst) null)
      ((pred? (car lst)) (remove pred? (cdr lst)))
      (else (cons (car lst) (remove pred? (cdr lst)))))))

```

The simple version of `set-difference` is then fairly straightforward: We want to remove every element of `initial` that is a member of `others`. We've defined the `remove` part above. Next, "is a member of `others`" can be written `(right-second member others)`. Putting it together, we get

```

(define set-difference
  (lambda (initial others)
    (remove (right-second member others) initial)))

```

Now, this version of `set-difference` accepts only two lists. What if we want a `set-difference` procedure that takes one or more lists $l_1, l_2, \dots, l_n$ as parameters and returns a list containing all of the elements of l_1 that are *not* also elements of any of $l_2, \dots, l_n$. For example,

```

> (set-difference (list 'a 'b 'c 'd 'e 'f 'g)
                  (list 'a 'e) (list 'b) (list 'a 'f 'h))
(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 lists. This improved `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:
;;; newset, a set
;;; Preconditions:
;;; All the sets are represented as lists.
;;; Postconditions:
;;; All values in newset appear in initial.
;;; No values in newset appear in other-1 ... other-n.
;;; All values in initial appear in either newset or other-1...other-n.
;;; 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.)
(define set-difference

```

```
(lambda (initial . others)
  (let kernel ((set initial)
               (remaining others))
    (if (null? remaining) set
        (kernel (remove (right-section member? (car remaining)) set)
                  (cdr remaining))))))
```

In English: Call the initial list `initial` and collect all of the other parameters 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 (just as we did in the original `set-difference`), and then repeat with the cdr.

Requiring More than One Parameter

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 parameters are required, while additional parameters will be collected into a list named `remaining-values`.

You will have the opportunity to explore such procedures in the laboratory.

Copyright © 2006 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.