

## Procedures that Return Multiple Values

When we invoke any of the procedures that we have discussed so far in the course, we get back a single result. However, there are many computations that we can describe most naturally as having two or more results. A typical example is the division of integers, as taught in elementary schools: When you divide 647 by 7, you get a quotient of 92 and a remainder of 3. It is not obvious that one of these results is more important or more significant than the other, and in fact Scheme provides a primitive for each one: `quotient` for the quotient, `remainder` for the remainder:

```
> (quotient 647 7)
92
> (remainder 647 7)
3
```

But since the same underlying computation is carried out in either case, it would make more sense to have a single procedure `divide` that would, when invoked, produce both results:

```
> (divide 647 7)
92
3
```

The procedure call `(divide 647 7)` is an example of a multiple-valued expression in Scheme. Evaluating a multiple-valued expression gives you several results, as the interaction displayed above shows. Note that this result is not the same as displaying multiple values nor is it the same as returning a list of values. We are, in fact, returning “many values” from one procedure.

Other kinds of expressions can also have multiple values:

- An `if`-expression is multiple-valued if the selected branch, the consequent or the alternate, is multiple-valued.
- A `cond`-expression is multiple-valued if the last subexpression in the selected `cond`-clause is multiple-valued.
- A `let`-, `let*`-, `letrec`-, or named `let`-expression is multiple-valued if the last expression in its body is multiple-valued.

However, identifiers, constants, `lambda`-expressions, `and`-expressions, and `or`-expressions are never multiple-valued.

Of course, we have to be careful about where we write multiple-valued expressions. We can't put one in the test position of an `if`-expression, for instance. If the test expression could be multiple-valued, it might have both `#t` and `#f` among its values! No, the test has to produce exactly one value, so that we know unambiguously whether to evaluate the consequent or the alternate.

Similarly, we can't write a multiple-valued expression as a subexpression of a procedure call, since the procedure to be called has to be some one particular procedure, and each of the arguments that we supply to it has to be some one particular value.

However, we *can* write a multiple-valued expression as the body of a lambda-expression, thus constructing our own multiple-result procedures, and this is the context in which you can expect to see them most frequently.

## The `values` procedure

All multiple-valued expressions enter Scheme, directly or indirectly, through calls to the primitive procedure `values`. `Values` is a variable-arity procedure that returns its arguments without change -- *all* of them.

```
> (values 650 682 513 861)
650
682
513
861
```

When the `values` procedure is given only one argument, it returns that argument without change:

```
> (values #\a)
#\a
```

It is also possible to call `values` with no arguments, in which case, of course, it returns no values:

```
> (values)
```

This is not an error or a non-terminating computation, but simply a computation that produces nothing when it finishes, like a committee that decides not to issue a report (which is sometimes the most useful thing a committee can do). You've seen procedures that return nothing before; `newline` and `display` both return nothing.

As a quick example, let's define a procedure `mixed-number-parts` that takes a rational number as its argument and returns its integer part and its proper fractional part:

```
;;; Procedure:
;;;   mixed-number-parts
;;; Parameters:
;;;   rat, A rational number
;;; Purpose:
;;;   Separate rat into whole and fractional parts.
;;; Produces:
;;;   whole, The whole part
;;;   fractional, The fractional part
;;; Preconditions:
;;;   The parameters must be a rational number.
;;; Postconditions:
;;;   rat = whole + fractional
(define mixed-number-parts
  (lambda (num)
    (let ((integer-part (truncate num)))
      (values integer-part (- num integer-part)))))
```

## The call-with-values procedure

Sometimes we want to use the values of a multiple-valued expression in some further computation instead of returning them directly. The fact that a multiple-valued expression cannot be used as an argument in a procedure call makes it difficult to do this.

Scheme's solution is another primitive procedure, `call-with-values`, that manages the flow of data from a multiple-valued expression into a larger computation. `Call-with-values` is a higher-order procedure that takes two arguments, a *producer* procedure that generates multiple values and a *consumer* procedure that accepts them.

The producer procedure takes no arguments and has a multiple-valued expression as its body. `Call-with-values` invokes the producer and collects all of the values that it returns. `Call-with-values` then invokes the consumer, providing the collected values as arguments. The arity of the consumer must therefore be compatible with the number of values delivered by the producer. `Call-with-values` returns whatever the consumer returns.

Suppose, for instance, that we want to recover the integer part and the proper fractional part of  $1173/83$  and then subtract the fractional part from the integer part. Here's how we could invoke `call-with-values` to perform the computation:

```
> (call-with-values (lambda () (mixed-number-parts 1173/83)) -)
1151/83
```

The producer in this case is `(lambda () (mixed-number-parts 1173/83))`, which returns two values when invoked. The consumer is `-`, which accepts two values and returns their difference.

As a more practical example, let's define a procedure that takes two arguments, a predicate `pred` and a list `ls`, and returns two lists, one consisting of all of the elements of `ls` that satisfy `pred`, the other consisting of all of the elements of `ls` that do *not* satisfy `pred`.

Our basic strategy is list recursion. In the base case, where `ls` is the empty list, we want to return two lists, both empty. That's easy -- we'll just write `(values null null)`. In any other case, we divide `ls` into its `car` and its `cdr` and invoke the procedure recursively to deal with the `cdr`. The recursive call will return two lists: the list of elements of the `cdr` that satisfy `pred`, and the list of elements of the `cdr` that do not. We `cons` the `car` of the list onto one or the other of these recursive results, depending on whether it does or does not satisfy `pred`, and return both the result of the `cons` and the other recursive result.

Translating into Scheme:

```
;;; Procedure:
;;; partition
;;; Parameters:
;;; pred?, a predicate
;;; ls, a list
;;; Purpose:
;;; Separate the list into two parts, those that meet the
;;; predicate and those that fail to meet the predicate.
;;; Produces:
;;; acceptable, a list
```

```

;;; unacceptable, a list
;;; Preconditions:
;;; pred? can be applied to every element of ls.
;;; Postconditions:
;;; Every element of acceptable is in ls.
;;; Every element of unacceptable is in ls.
;;; Every element of ls appears in exactly one of acceptable
;;; and unacceptable.
;;; pred? holds for every element of acceptable.
;;; pred? fails to hold for every element of unacceptable.
(define partition
  (lambda (pred? ls)
    (letrec ((recurrer
              (lambda (ls)
                (if (null? ls)
                    (values null null)
                    (call-with-values
                     (lambda () (recurrer (cdr ls)))
                     (lambda (ins outs)
                       (if (pred? (car ls))
                           (values (cons (car ls) ins) outs)
                           (values ins (cons (car ls) outs))))))))
              (recurrer ls))))))

```

Notice how one uses `call-with-values` to manage the transfer of two values from the multiple-valued expression `(recurrer (cdr ls))` into the part of the computation that consumes those values.

Think of the body of the producer, the expression `“(recurrer (cdr ls))”`, as ready to supply its results when asked. Think of the body of the consumer, the inner `if`-expression, as ready to receive those results and operate on them to construct the final values that `recurrer` will return. The role of `call-with-values` is to activate and mediate these two packaged components of the overall computation.