

8 Recursion

This section introduces *recursive functions*. Defining recursive functions in Scheme requires no new computational constructs (i.e., no new special forms); instead, we simply combine existing constructs in a new way. In many cases, recursive functions can provide compact and elegant solutions to interesting computational problems.

We begin by recalling that the evaluation of a non-empty list according to the Default Rule typically involves the application of a function to zero or more inputs.¹¹ For convenience, we make the following definition:

⇒ Suppose *expr* is a Scheme expression that represents a non-empty list, *L*, whose evaluation is governed by the Default Rule. Then we say that *expr* is a *function-call expression*. Furthermore, suppose *f* is the function entity that results from evaluating the first element of the list *L*. Then we say that *expr* *calls* *f*.

Thus, for example, the expression, `(+ 2 3)`, is a function-call expression that calls the built-in *addition* function. Similarly, `(symbol? 'x)` is a function-call expression that calls the built-in *symbol?* function. In contrast, the expressions, `(define myVar 3)` and `(lambda (x) (* x x))`, represent special forms and, thus, are *not* function-call expressions.

Recursive Function.

⇒ A function, *f*, is said to be *recursive* if its *body* contains a function-call expression that calls *f*.

At first glance, this might seem like a crazy idea—after all, a function *calling itself* sounds like the kind of circularity that might lead to infinite loops. However, this dreaded form of circularity is generally quite easy to avoid, as follows.

⇒ A recursive function typically includes a conditional statement that tests some *stopping condition* (or *base case*). If the stopping condition evaluates to boolean *true*, then no recursive function call is made. Not only that, in cases where the recursive function call is made, it typically involves applying the function to *different inputs*.

Thus, as will be amply demonstrated, a typical sequence of recursive function calls is less like a circle that forever loops back on itself, and more like a spiral that converges on some stopping point.

Defining Recursive Functions in Scheme. In Scheme, the typical characteristics of the definition of a recursive function, *f*, are:

- a **define** special form that effectively gives a name to *f*;
- a conditional expression (in the body) that distinguishes the base case from the recursive case; and
- a function-call expression (in the body) that typically involves applying *f* to other input(s).

Thus, no new Scheme constructs are required to support recursion.

Example 8.1 The factorial function. The factorial function, $f(n) = n!$, is frequently defined as follows:

$$\bullet f(n) = n! = n \cdot (n - 1) \cdot (n - 2) \dots \cdot 3 \cdot 2 \cdot 1$$

This kind of definition is somewhat casual, as evidenced by the “dot-dot-dot”. What does the “dot-dot-dot” mean exactly?

We can give a more precise, recursive definition of the factorial function, as follows:

- Base Case ($n = 1$): $1! = 1$.
- Recursive Case ($n > 1$): $n! = n \cdot (n - 1)!$

According to this definition, the following equalities hold:

- $4! = 4 \cdot 3!$
- $3! = 3 \cdot 2!$

¹¹Of course, if the first element of the list evaluates to something other than a function, then no function application can happen.

- $2! = 2 \cdot 1!$
- $1! = 1$

Putting all of this information together yields:

$$4! = 4 \cdot 3! = 4 \cdot (3 \cdot 2!) = 4 \cdot (3 \cdot (2 \cdot 1!)) = 4 \cdot (3 \cdot (2 \cdot 1)) = 24.$$

The following Scheme expression defines a recursive function, `facty-v1`, whose definition is based on the above insights.¹² The main job of `facty-v1` is to use recursion to compute the factorial of its input, `n`.

```
(define facty-v1
  (lambda (n)
    (if (= n 1)
        1
        (* n (facty-v1 (- n 1))))))
```

Notice that the `define` special form effectively gives the name, `facty-v1`, to the function defined by the `lambda` special form. Notice, too, that the body of this function includes a conditional expression that distinguishes the base case (i.e., when `n = 1`) from the recursive case (i.e., when `n > 1`). Finally, notice that the body includes a function-call expression that calls `facty-v1`. (We'll have more to say about this!)

Okay, so what happens when the above expression is evaluated? Well, the expression is a `define` special form. So, the symbol, `facty-v1`, is *not* evaluated. Only the third element of the `define` special form—namely the `lambda` expression—is evaluated. Like any `lambda` expression, the one above evaluates to a function entity. However:

⇒ It is important to remember that evaluating the above `lambda` expression only creates a function entity. It does *not* call the function! Thus, the expressions in the body of the `lambda` expression are *not* evaluated—yet!

The reason this is important is that the Global Environment does not yet contain an entry for the symbol, `facty-v1`. Thus, any attempt to evaluate that symbol, at this time, would cause an error.

However, after the `lambda` expression has been evaluated (to a function), the evaluation of the `define` special form can continue: in particular, by adding a new entry to the Global Environment that associates the symbol, `facty-v1`, with the newly created function.

Next, let's observe that `facty-v1` appears to correctly compute the factorial of its input:

```
> (facty-v1 1)
1
> (facty-v1 2)
2
> (facty-v1 3)
6
> (facty-v1 4)
24
```

Before delving deeper into why `facty-v1` works, observe that we can define an equivalent function, `facty-v2`, using a `cond` expression, as follows:

```
(define facty-v2
  (lambda (n)
    (cond
      ;; Base Case: n = 1
      ((= n 1)
       1)
      ;; Recursive Case: n > 1
      (#t
       (* n (facty-v2 (- n 1)))))))
```

¹²The function is called, `facty-v1`, because it is the first version of the factorial function we will look at.

Notice how the comments clearly distinguish the base case from the recursive case. Once again, this function appears to correctly compute the factorial of its input:

```
> (facty-v2 1)
1
> (facty-v2 2)
2
> (facty-v2 3)
6
> (facty-v2 4)
24
```

Finally, we can define another equivalent version of the factorial function, this time called `facty`. This function differs only in that it contains some `printf` expressions that will help us to trace what happens when an expression such as `(facty 3)` is evaluated:

```
(define facty
  (lambda (n)
    (cond
      ;; Base Case: n = 1
      ((= n 1)
       (printf "Base Case (n = 1)~%" )
       1)
      ;; Recursive Case: n > 1
      (#t
       (printf "Recursive Case (n = ~A)~%" n)
       (* n (facty (- n 1)))))))
```

Notice that the `printf` expressions do not affect the output of the function; they only cause some useful side-effect printing to occur.

Evaluating `(facty 3)`. Consider DrScheme's evaluation of the expression, `(facty 3)`. This is a function-call expression whose evaluation is governed by the Default Rule. Thus, the symbol `facty` and the number `3` must both be evaluated. The symbol `facty` evaluates to the function we just defined; and the number `3` evaluates to itself. Next, the `facty` function is applied to the input `3`.

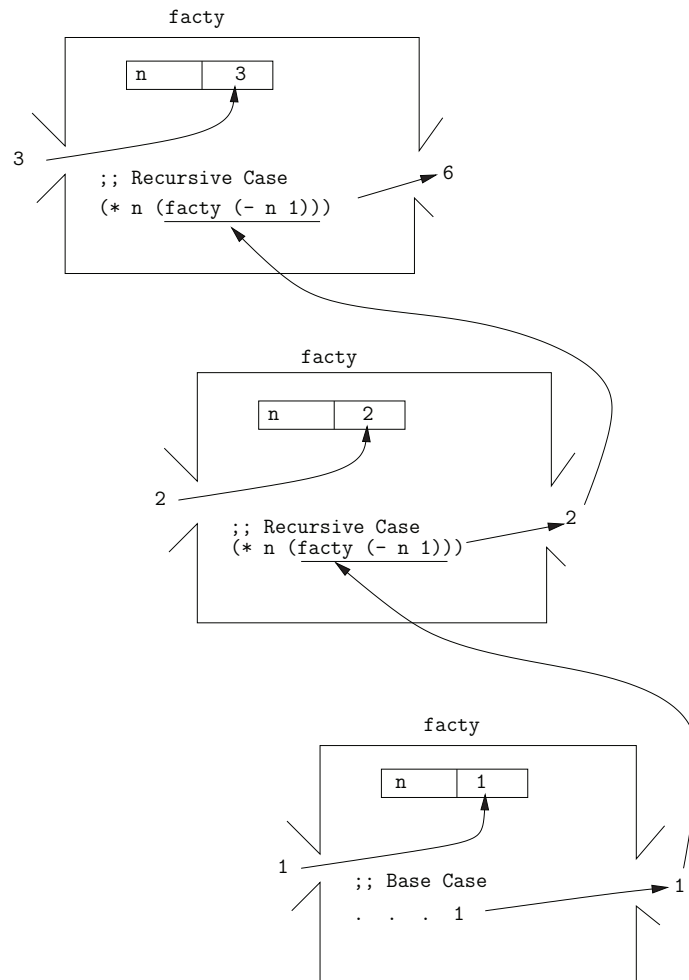
The application of the `facty` function to the input `3` is depicted at the top of Fig. 3. First, a local environment is created with an entry associating the input parameter `n` with the value `3`. Next, the expression in the body of the `facty` function, shown below, is evaluated with respect to that local environment.¹³

```
(cond
  ;; Base Case: n = 1
  ((= n 1)
   (printf "Base Case (n = 1)~%" )
   1)
  ;; Recursive Case: n > 1
  (#t
   (printf "Recursive Case (n = ~A)~%" n)
   (* n (facty (- n 1))))))
```

Since the value of `n` is `3` in the local environment, the condition, `(= n 1)`, evaluates to `#f`. Thus, we skip to the second condition, `#t`, which of course evaluates to `#t`. Thus, the expressions associated with the recursive case are evaluated in turn. The first expression causes the line, `Recursive Case (n = 3)`, to be displayed in the Interactions Window. Then, the second expression, `(* n (facty (- n 1)))`, must be evaluated—according to the Default Rule. The `*` symbol evaluates to the *multiplication* function, `n` evaluates to `3`, and `(facty (- n 1))` evaluates to ... Gosh, we need a new paragraph!

The expression, `(facty (- n 1))`, is evaluated according to the Default Rule. First, the `facty` symbol evaluates to the `facty` function; and `(- n 1)` evaluates to `2` (since `n` has the value `3`). Next, the `facty` function must be applied to the input value `2`, as depicted in the second box in Fig. 3.

¹³To decrease clutter, only a portion of the body is shown in each function-call box in the figure.

Figure 3: DrScheme's evaluation of `(facty 3)`

⇒ Notice that the evaluation of the expression, `(* (facty (- n 1)))`, in the top function-call box cannot continue until the subsidiary expression, `(facty (- n 1))`, is evaluated. However, this value cannot be known until the output value for the *second* function-call box has been generated! In other words, the evaluation of the expression in the top box must be *suspended*, pending the outcome of the second box.

The application of the `facty` function to the value 2, depicted in the second function-call box in the figure, is similar to the application of the `facty` function to 3 in the top box, except that the local environment in the second box associates the input parameter, `n`, with the value 2.

⇒ Crucially, the local environments in separate function-call boxes do not cause a conflict! They can't see one another. Neither knows that the other even exists! Thus, although the two input parameters are both called `n`, they are quite distinct!

Thus, the evaluation of the body of the function in the second box proceeds in the environment where `n` has the value 2. Thus, the base case is skipped and the expressions associated with the recursive case are evaluated. The evaluation of the `printf` expression causes the line, `Recursive Case (n = 2)`, to be displayed in the Interactions Window; and the evaluation of the expression, `(* n (facty (- n 1)))`, leads to yet another recursive function call—this time the application of the `facty` function to the input value 1, as illustrated in the third box in Fig. 3.

⇒ Once again, the evaluation of the expression, `(* n (facty (- n 1)))`, in the second box cannot continue until the output value for the third box has been generated. In other words, the evaluation of the expression in the second box must be suspended, pending the outcome of the third box.

The application of the `facty` function to the value 1 begins by creating a local environment entry that associates the input parameter `n` with the value 1. (Again, this is a new input parameter, distinct from the other `n`'s!) Next, the `cond` expression in the body of the function is evaluated. This time, however, the condition `(= n 1)` evaluates to `#t`; thus, the base case expressions are evaluated. Evaluating the `printf` expression causes the line, `Base Case (n = 1)`, to be displayed in the Interactions Window. Next, the expression, `1`, evaluates to itself, yielding the output value for the application of the `facty` function to the value 1 (i.e., the output value for the third box).

This output value, 1, is the value of the expression, `(facty (- n 1))`, that was being evaluated in the middle function-call box (where `n` has the value 2). Now that that the value of `(facty (- n 1))` is in hand, the evaluation of the expression, `(* n (facty (- n 1)))`, in the middle box can continue. To wit, the *multiplication* function is applied to 2 and 1, yielding the output value 2 for the middle function-call box.

This output value, 2, is the value of the expression, `(facty (- n 1))`, that was being evaluated in the top function-call box (where `n` has the value 3). Now that the value of `(facty (- n 1))` is in hand, the evaluation of the expression, `(* n (facty (- n 1)))`, in the top box can continue. To wit, the *multiplication* function is applied to 3 and 2, yielding the output value 6 for the top function-call box.

Phew!

Here is what it looks like when `(facty 3)` is evaluated in the Interactions Window:

```
> (facty 3)
Recursive Case (n = 3)
Recursive Case (n = 2)
Base Case (n = 1)
6
```

■

Example 8.1 illustrates many of the features that are frequently found in recursive functions.

- The body of the function contains a conditional expression that enables a stopping condition—commonly called a *base case*—to be recognized. If that stopping condition evaluates to `#t`, then no more recursive function calls are made.
- The body of the function contains an expression that involves a recursive call to that same function—but with different input(s). It is crucial that the inputs to the recursive function call be different in some way; otherwise, that recursive function call would lead to another identical recursive function call, and so on,

ad infinitum.¹⁴ Because the inputs to the recursive function call are different in some way, the recursive function call is not circular; instead, the sequence of recursive function calls is more like a spiral that eventually stops when the base case is arrived at.

- Although the expression in the body of the function is identical in each recursive function call, it is evaluated with respect to a different local environment. In other words, the evaluation of the body is affected by the value of the input parameter(s). This helps to avoid circularity and infinite loops.

Example 8.2 Summing Squares. Consider the function, $g(n) = 1^2 + 2^2 + 3^2 + \dots + n^2$. Notice that $g(n)$ sums the squares of the integers between 1 and n , inclusive. We can define g recursively, as follows:

- Base Case ($n = 1$): $g(1) = 1$
- Recursive Case ($n > 1$): $g(n) = n^2 + g(n - 1)$

Notice that $g(1) = 1$, $g(2) = 1^2 + 2^2 = 5$, $g(3) = 1^2 + 2^2 + 3^2 = 14$, and so on.

In Scheme, we can define a function, called `sum-squares`, that computes the sum of the squares from 1 to its input value n , as follows:

```
(define sum-squares
  (lambda (n)
    (cond
      ;; Base Case: n = 1
      ((= n 1)
       1)
      ;; Recursive Case: n > 1
      (#t
       (+ (* n n) (sum-squares (- n 1)))))))
```

We can test the function in the Interactions Window, as follows:

```
> (sum-squares 1)
1
> (sum-squares 2)
5
> (sum-squares 3)
14
> (sum-squares 4)
30
```

■

¹⁴There are exceptions to this observation that involve destructive functions. However, this course focuses primarily on non-destructive functions.