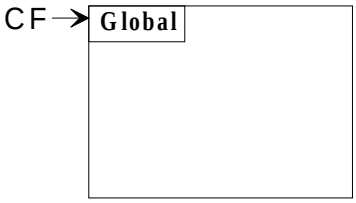
Evaluate the following:

```
> (define (factorial n)
    (if (= n 0) 1
        (* n (factorial (- n 1))) ))
> (factorial 3)
```

CF

```
(define (factorial n)
  (if (= n 0) 1
      (* n (factorial (- n 1))) ))
(factorial 3)
```

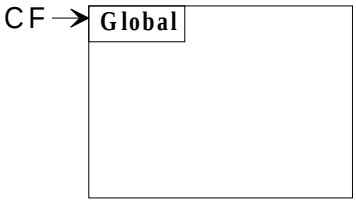


This has syntactic sugar, so translate it into real code.

Step 1

CF

```
(define factorial
  (lambda (n)
    (if (= n 0) 1
        (* n (factorial (- n 1))) ))
(factorial 3)
```

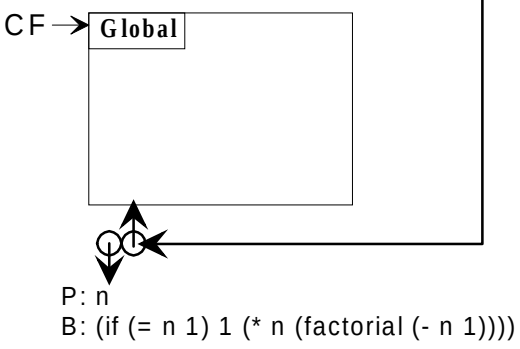


To evaluate the define expression, we must find out to what the symbol *factorial* is bound.

Step 2

CF

```
(define factorial □)
(factorial 3)
```

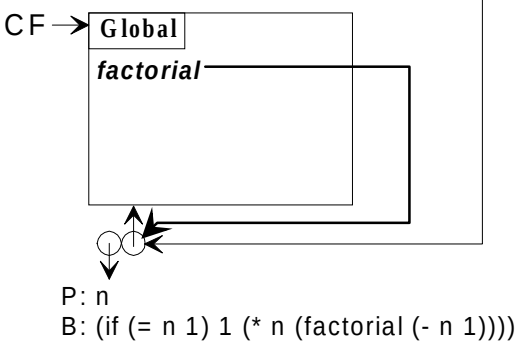


The *lambda* expression evaluates to a procedure. The procedure points to the frame it was evaluated in, and to its parameters and unevaluated body.

Step 3

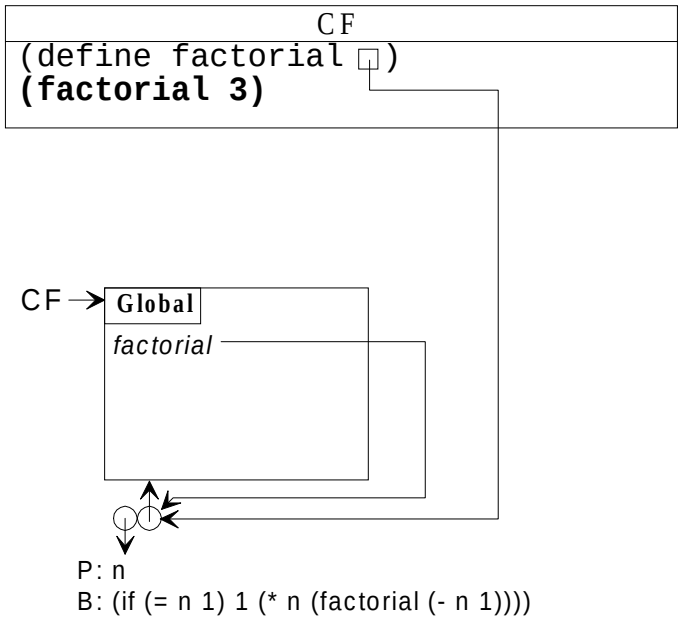
CF

```
(define factorial □)
(factorial 3)
```



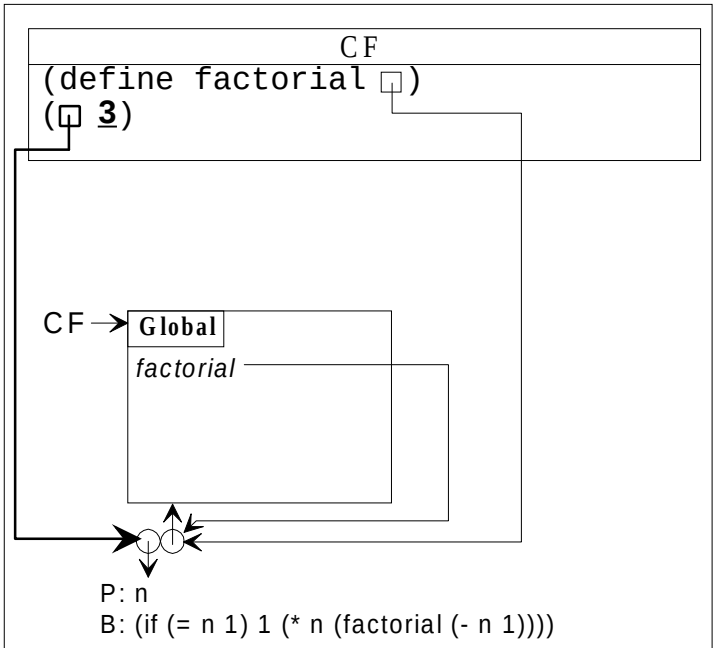
Now that we know to what *factorial* is bound, we create the symbol *factorial* and bind it to the value. In this case, the value is a procedure.

Step 4



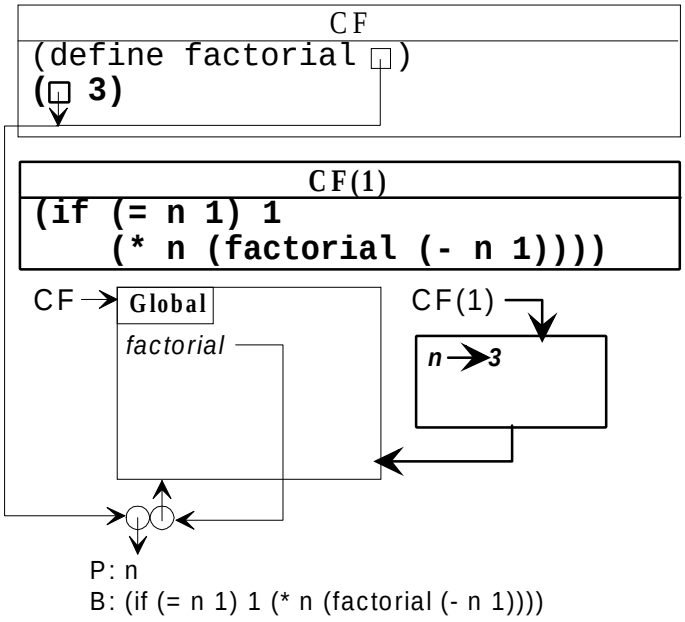
Finished with the first expression, we now move on to the second. This is a list, and not a special form, so we must evalute all of its sub-expressions.

Step 5



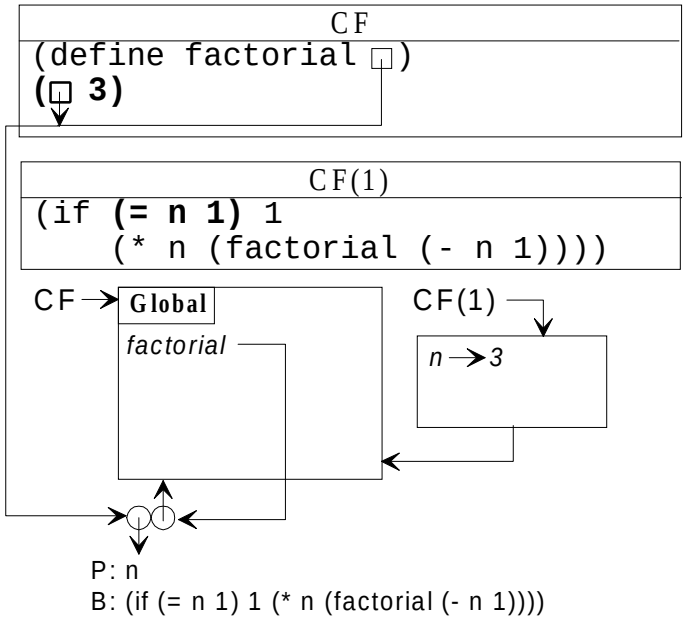
The symbol *factorial* evaluates to the procedure it's bound to, and the number *3* evaluates to itself.

Step 6



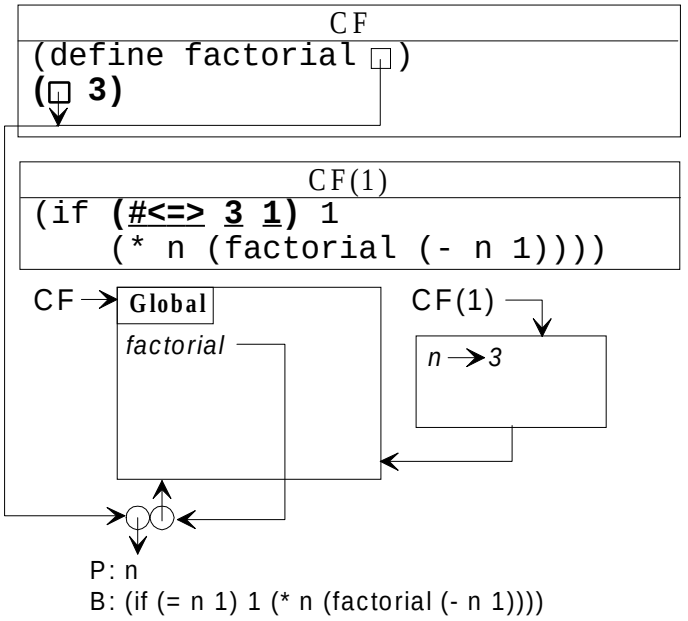
Now we call the procedure on the actual parameters in the list. This creates a frame that points to the same frame as the procedure. We also bind the formal parameter symbols to their respective actual parameter values in the new frame, and start evaluating the procedure's body in the new current frame.

Step 7



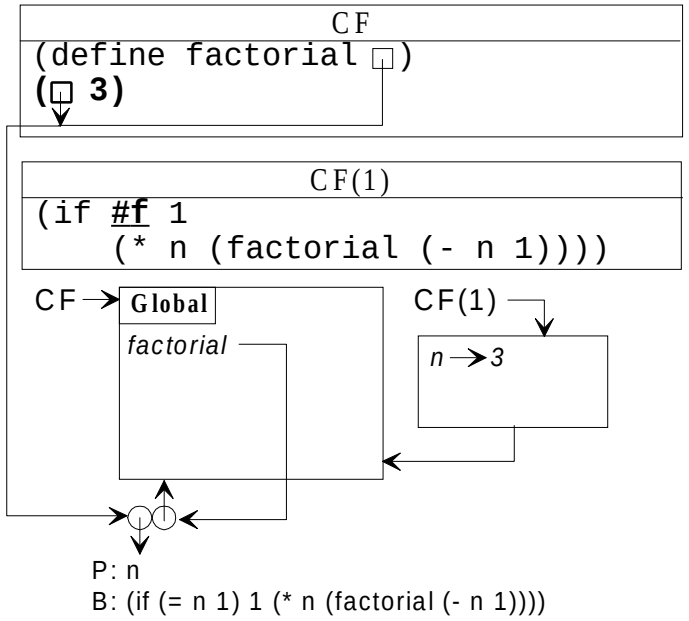
The first expression in the body of the procedure is an *if*, which is a special form. The predicate of the *if* expression is always evaluated first to determine whether to evaluate the consequent or alternate.

Step 8



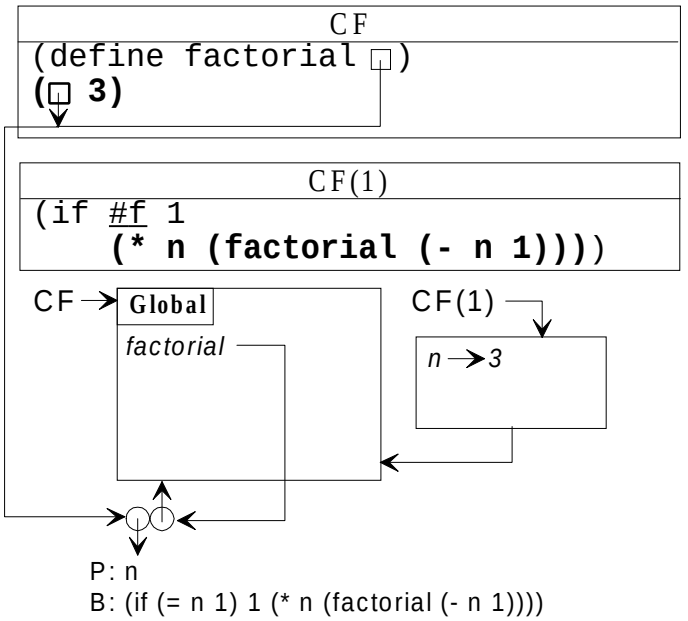
The symbol = evaluates to the primitive procedure defined in Scheme's global environment. The symbol *n* evalutes to the numerical value 3. The number *1* is self evaluating and evaluates to itself.

Step 9



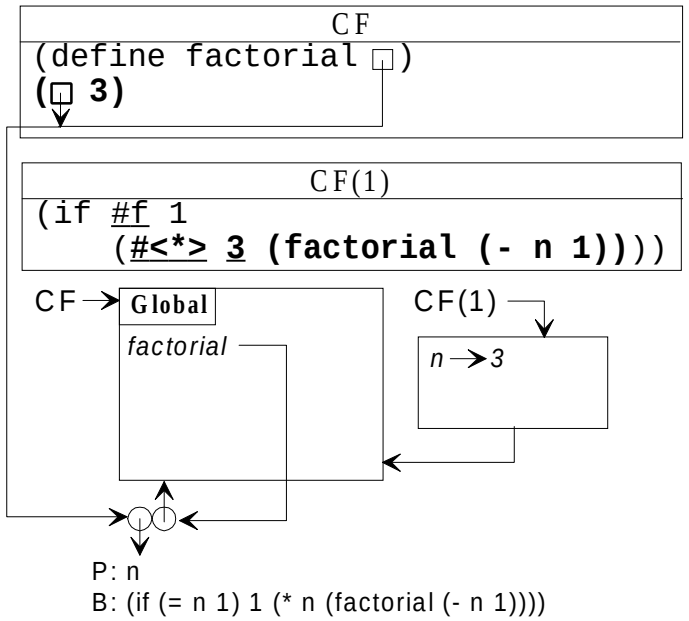
Calling the primitive procedure on the actual parameters 3 and 1 returns #f. This means we will be evaluating the alternate expression.

Step 10



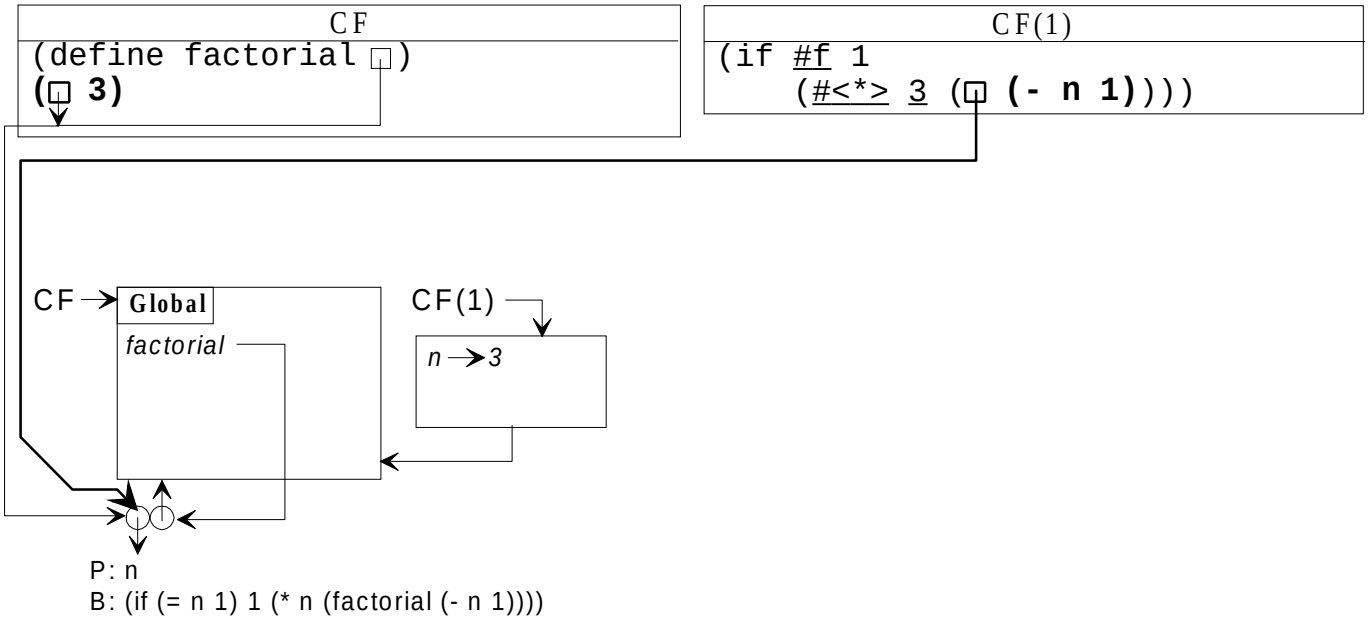
Looking at the alternate expression, we see that it is a non-special list form. We must evaluate all of its sub-expressions before we can finish evaluating it.

Step 11



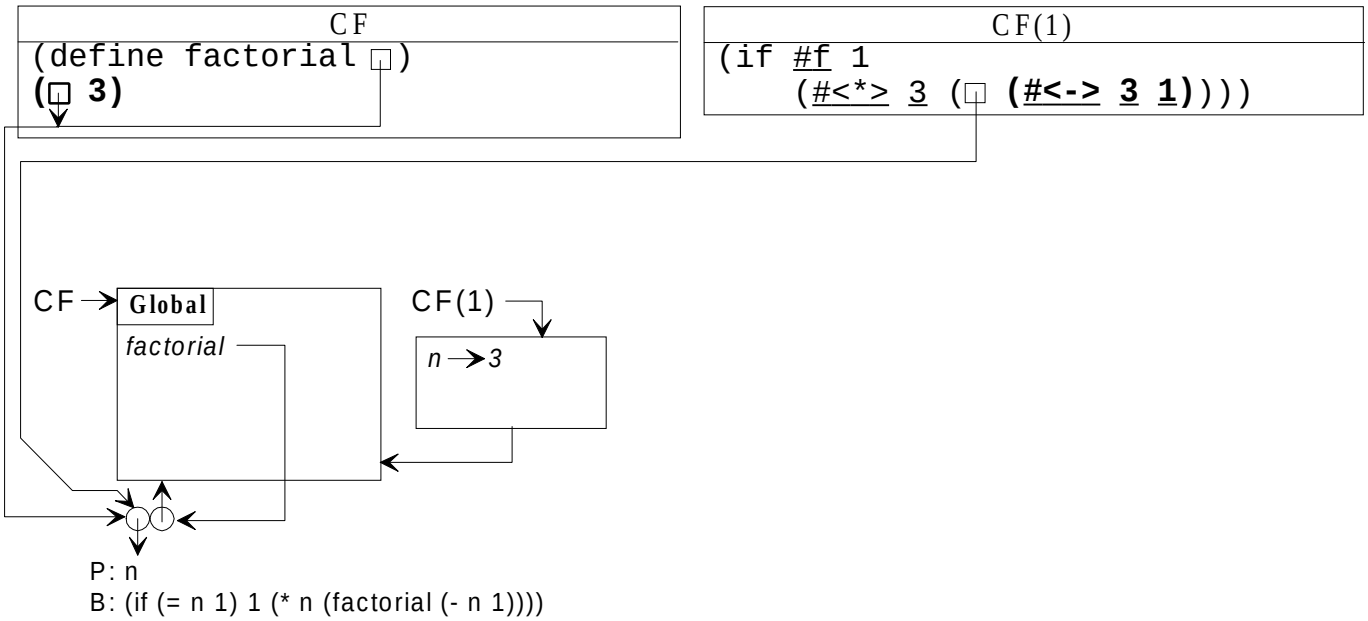
The symbol * evalutes to Scheme's respective primitive procedure, and the number *3* evalutes to itself. Notice that the third expression is a non-special list form. We must evalute all of its sub-expressions before we can finish evaluting it.

Step 12



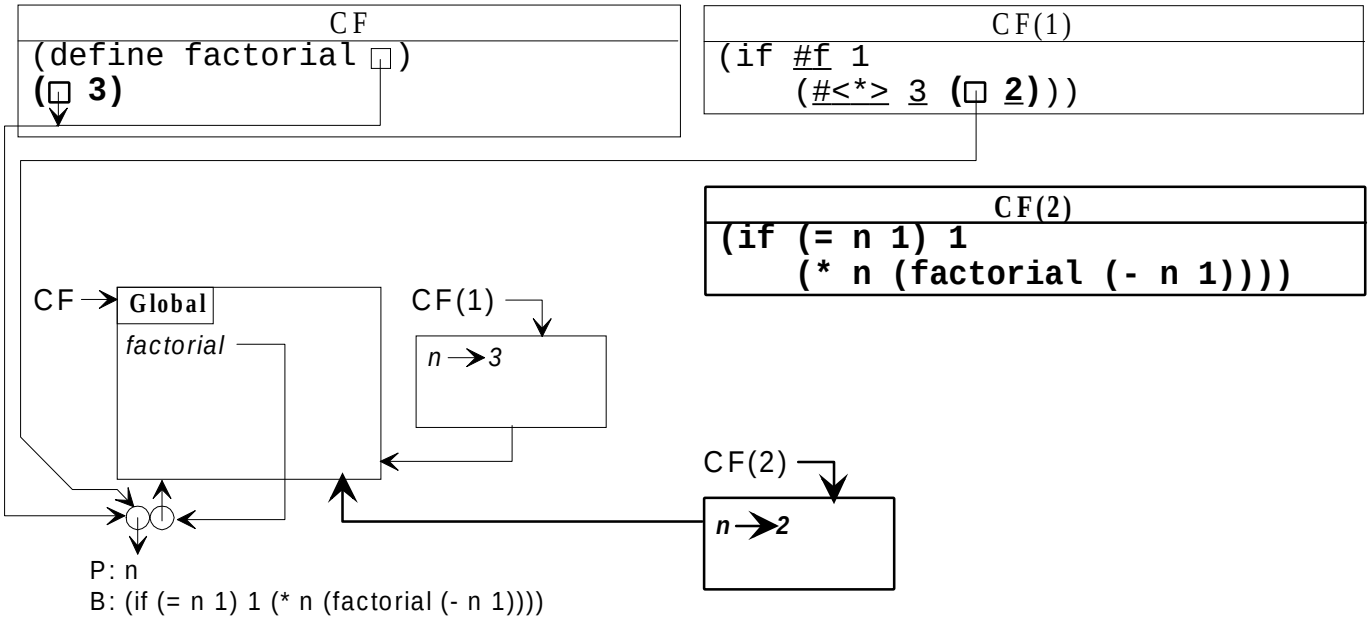
The symbol *factorial* evalutes to the respective procedure in the current frame's environment. The next expression to evalute is yet another non-special form list. We must evalute all of its sub expressions before we can finish evaluting it.

Step 13



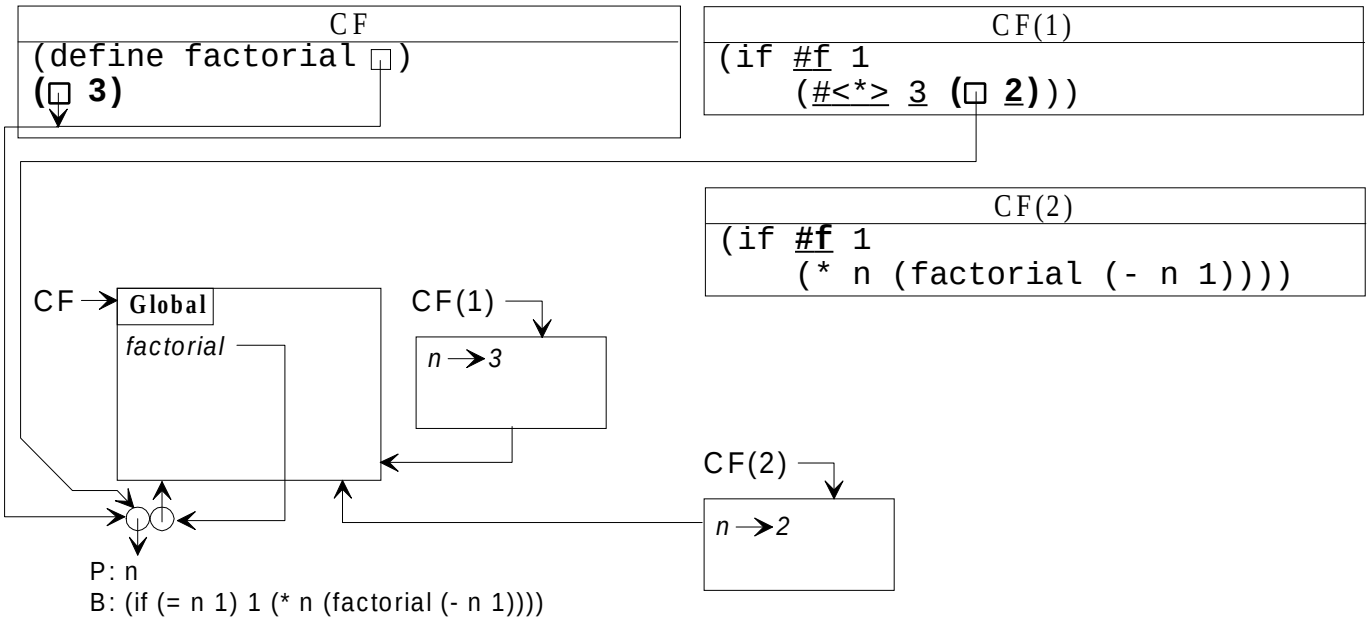
The symbol - evalutes to its respective primitive procedure. The numbers evalute to themselves. Now that all of the elments in this list have been evaluated, we can call the #<-> primitive procedure on the actual parameters.

Step 14



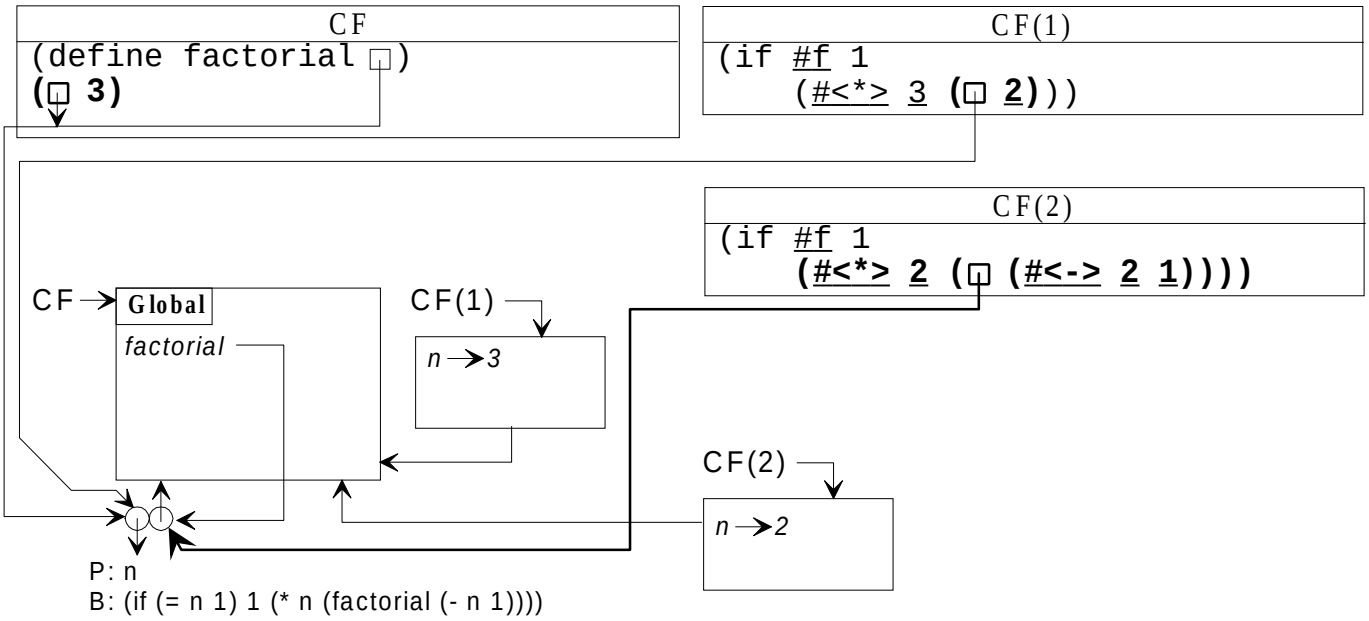
The result of calling that procedure is the numerical value 2. Now that all of the elements in this list have been evaluted, we can call the procedure (the first element in the list) with its actual parameters (the rest of the elements in the list). This is done as before, and we create a new frame that points to where the procedure points, bind the formal parameter symbols to their respective actual parameter values in the new frame, and start executing the procedure body in the new current frame.

Step 15



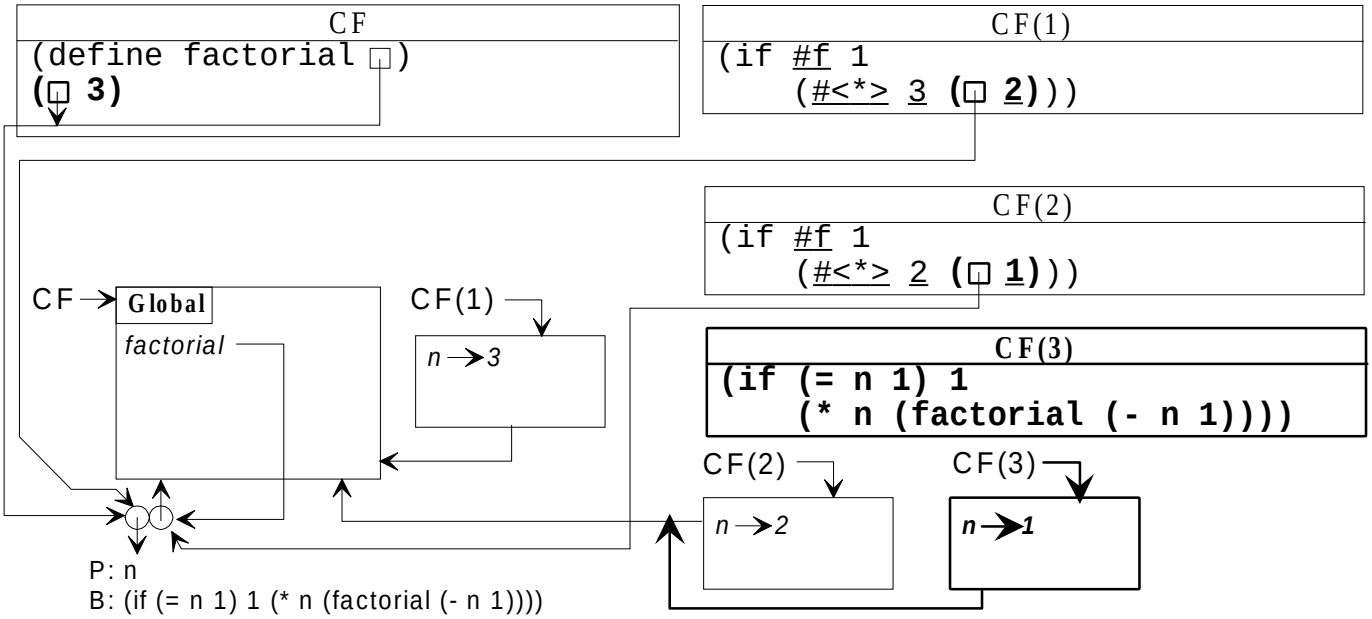
Paralleling steps 9 and 10, we first need to evalute the predicate of the *if* expression. This requires that we evaluate a non-special list form and consequently evalute all of the list's sub-expressions before calling the primitive procedure #<=> on the resultant actual parameters. The result of this is #f. This tells us to evalute the alternate expression.

Step 16



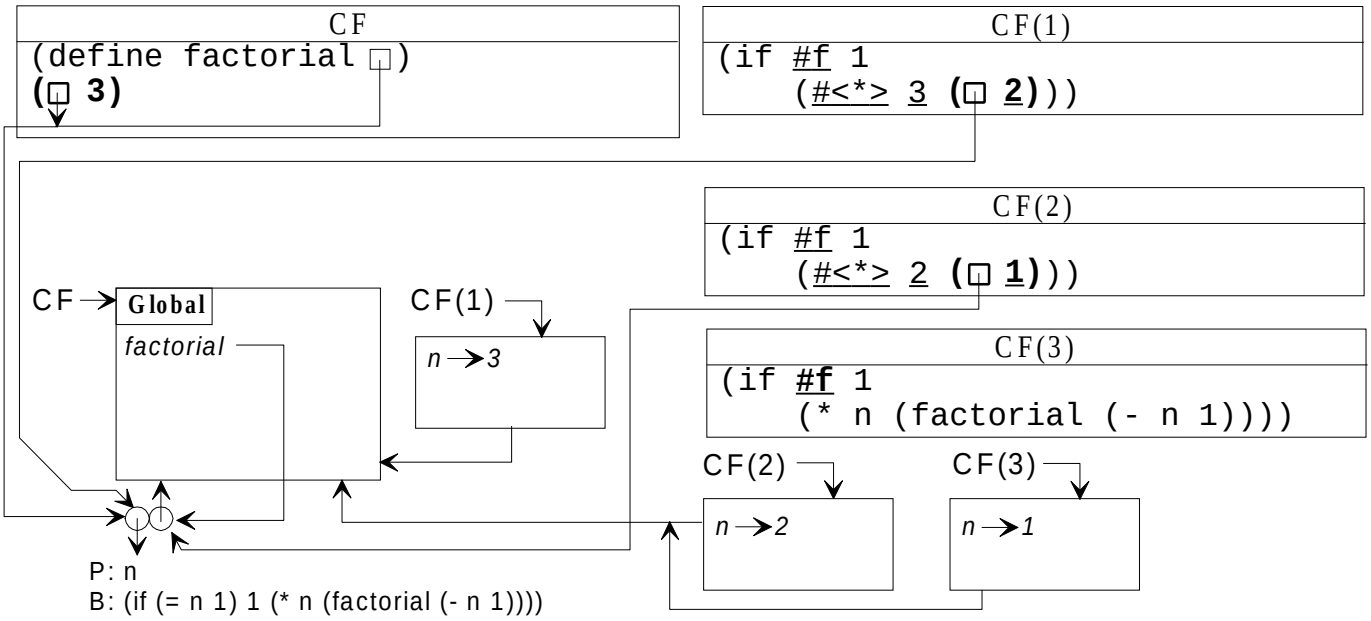
Paralleling steps 11 through 14, we first need to evalute all of the sub-expressions of the non-special list form alternate expression. It contains various other non-special list forms which also need their sub-expressions evaluted and we do this until we find a place where we can finally simplify the expression by calling a procedure.

Step 17



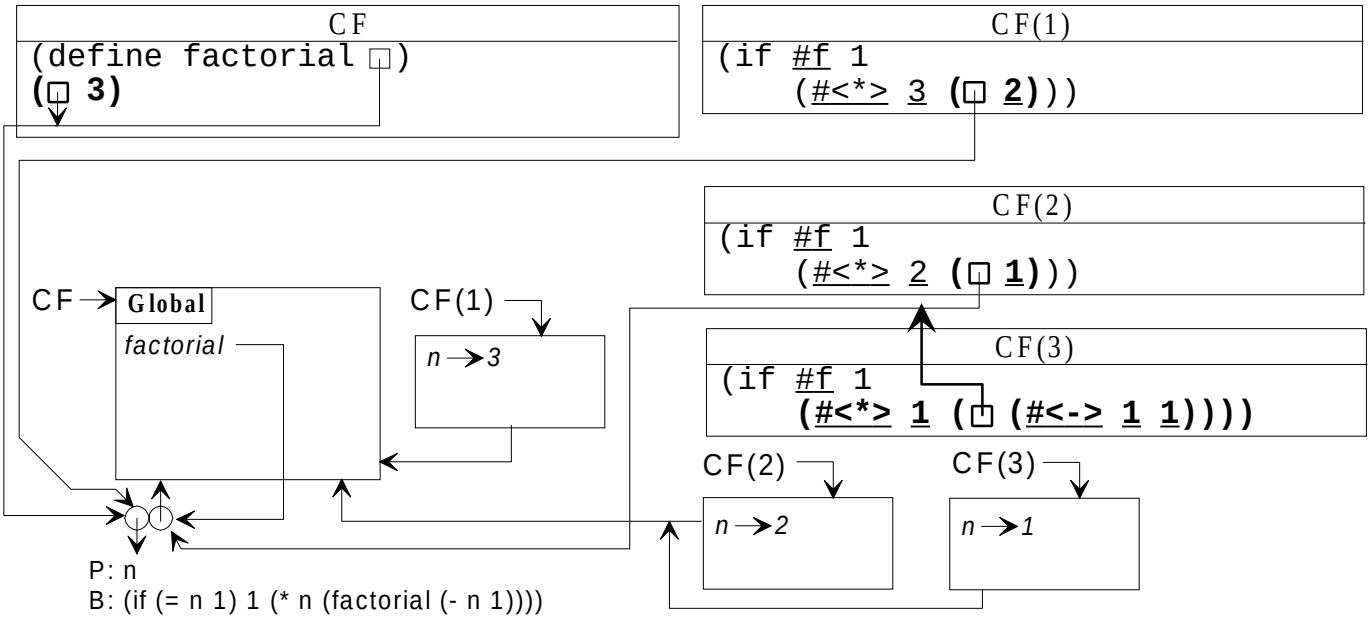
Paralleling step 15, calling the primitive procedure #<-> results in the numerical value 1, and then we can call the procedure on the actual parameter 1. The result is the creation of a new frame which point to where the procedure points (shown by pointing the arrow at another line that points at the correct frame in order to save space), binding the formal parameter symbols to their respective actual parameters, and evaluating the body of the procedure at the new current frame.

Step 18



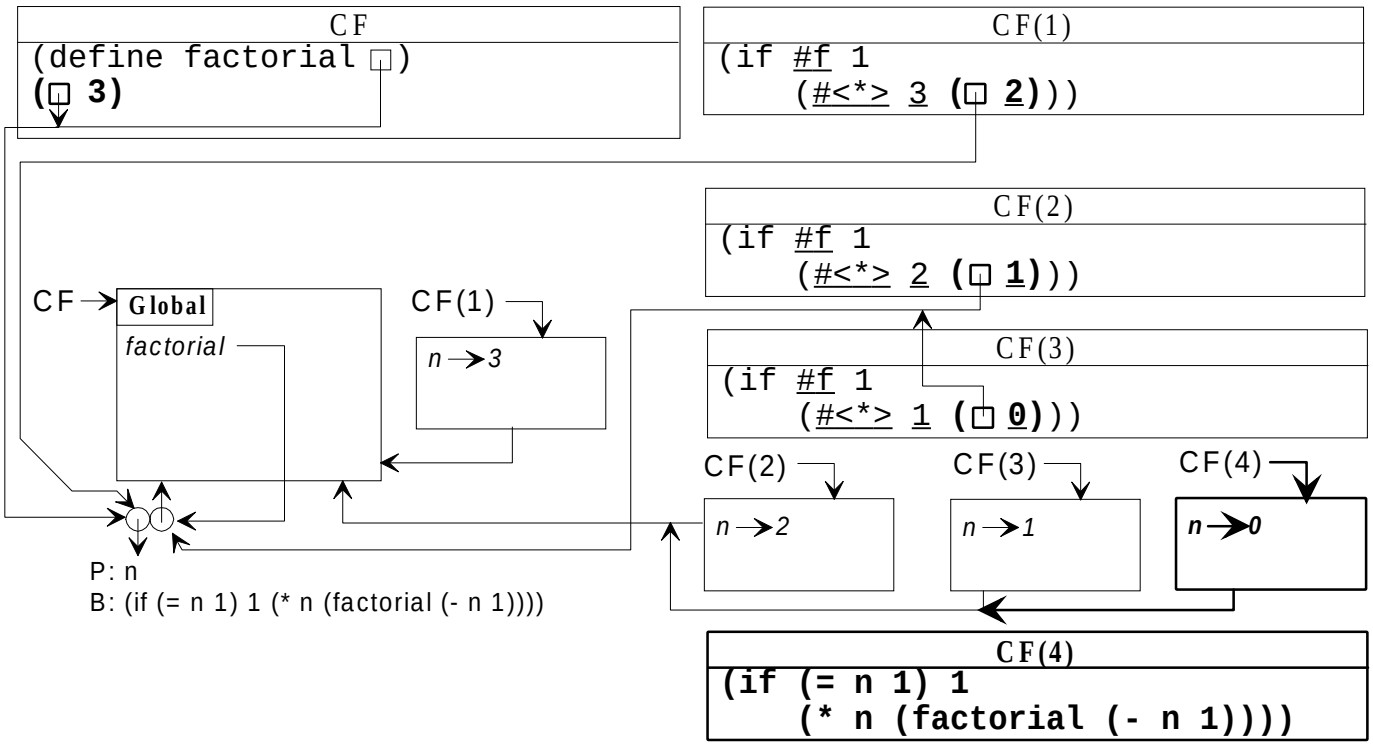
Paralleling steps 9 and 10, we first need to evalute the predicate of the if expression. This requires that we evaluate a non-special list form and consequently evalute all of the list's sub-expressions before calling the primitive procedure #<=> on the resultant actual parameters. The result of this is #f. This tells us to evalute the alternate expression.

Step 19



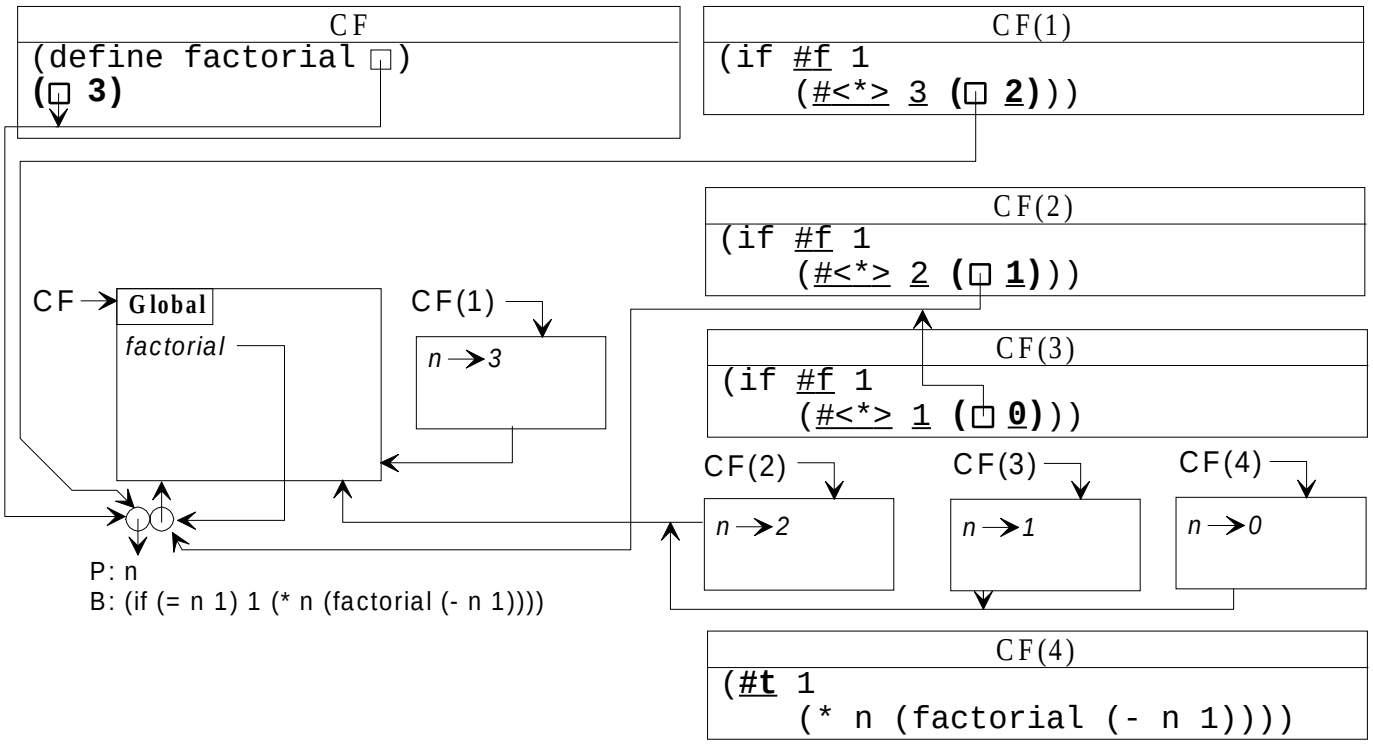
Paralleling steps 11 through 14, we first need to evalute all of the sub-expressions of the non-special list form alternate expression. It contains various other non-special list forms which also need their sub-expressions evaluted and we do this until we find a place where we can finally simplify the expression by calling a procedure. Also note that I pointed the arrow to another line that points to the procedure to save space.

Step 20



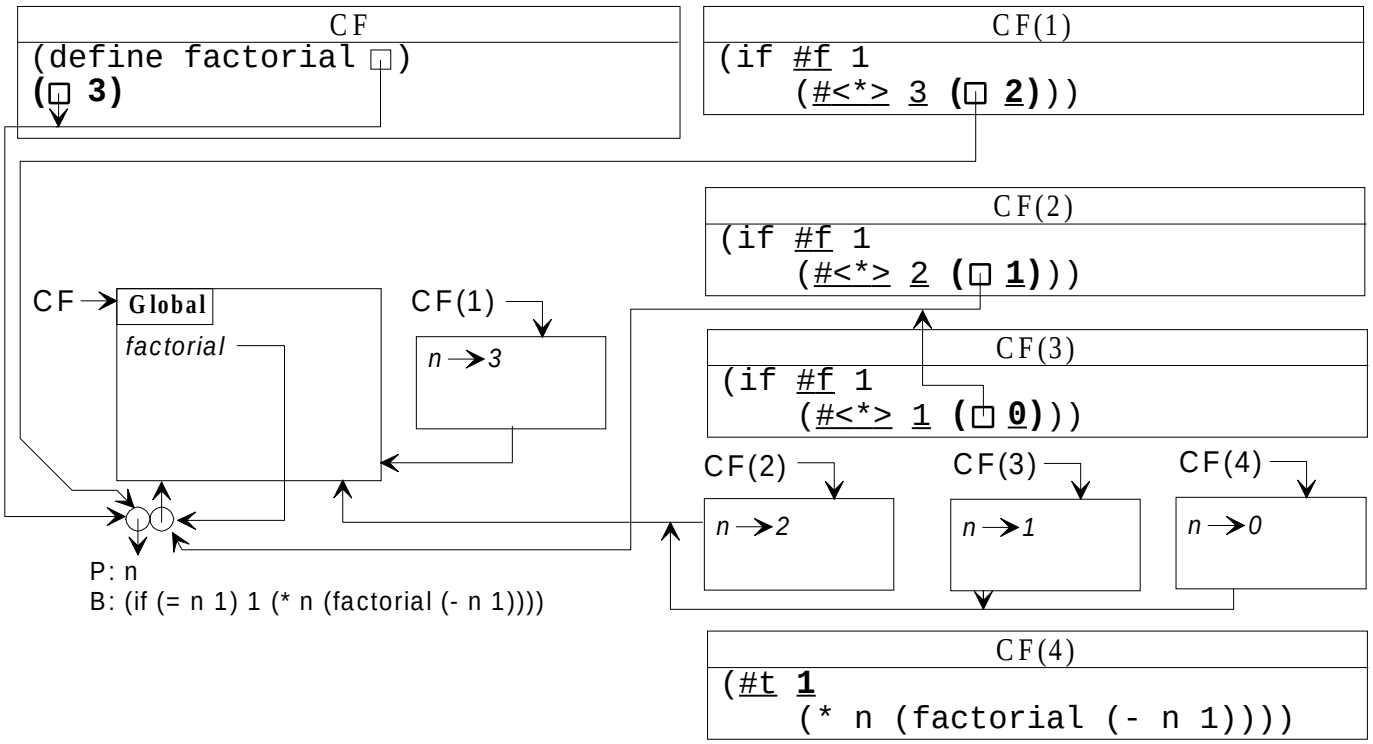
Paralleling step 15, calling the primitive procedure #<-> results in the numerical value 1, and then we can call the procedure on the actual parameter 1. The result is the creation of a new frame which point to where the procedure points (shown by pointing the arrow at another line that points at the correct frame in order to save space), binding the formal parameter symbols to their respective actual parameters, and evaluating the body of the procedure at the new current frame.

Step 21



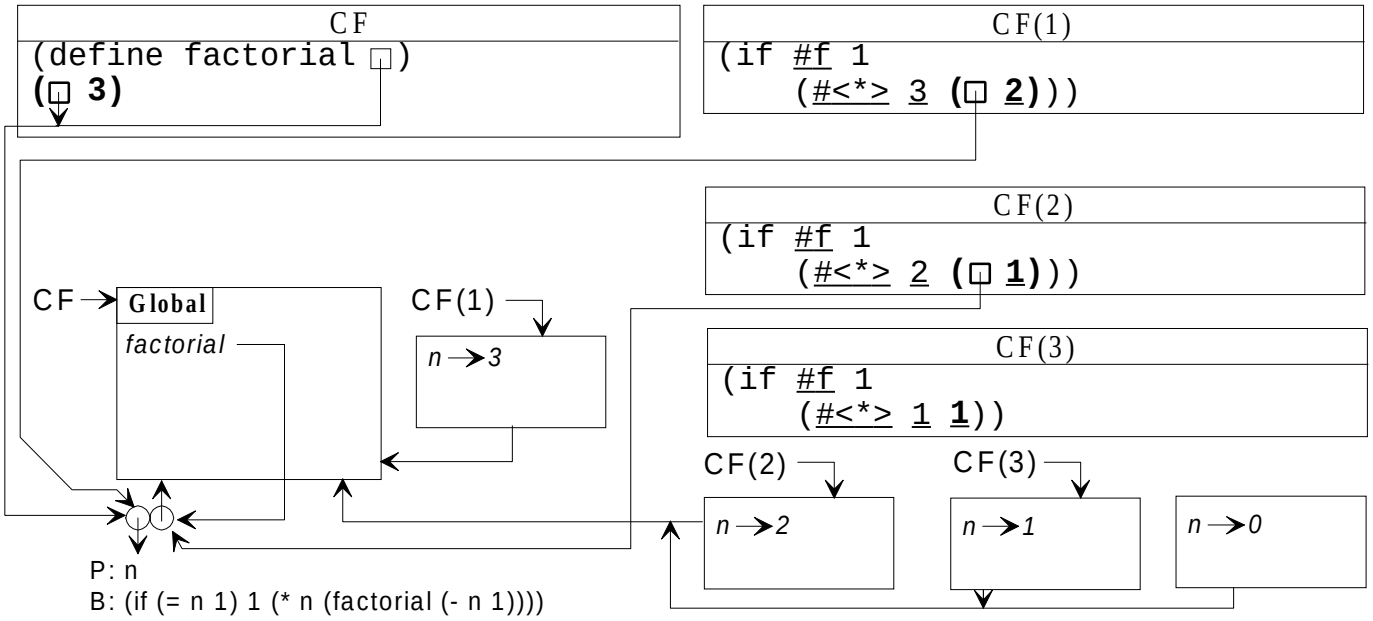
Paralleling steps 9 and 10, we first need to evalute the predicate of the if expression. The result of this is #t. This tells us to evalute the consequent expression.

Step 22



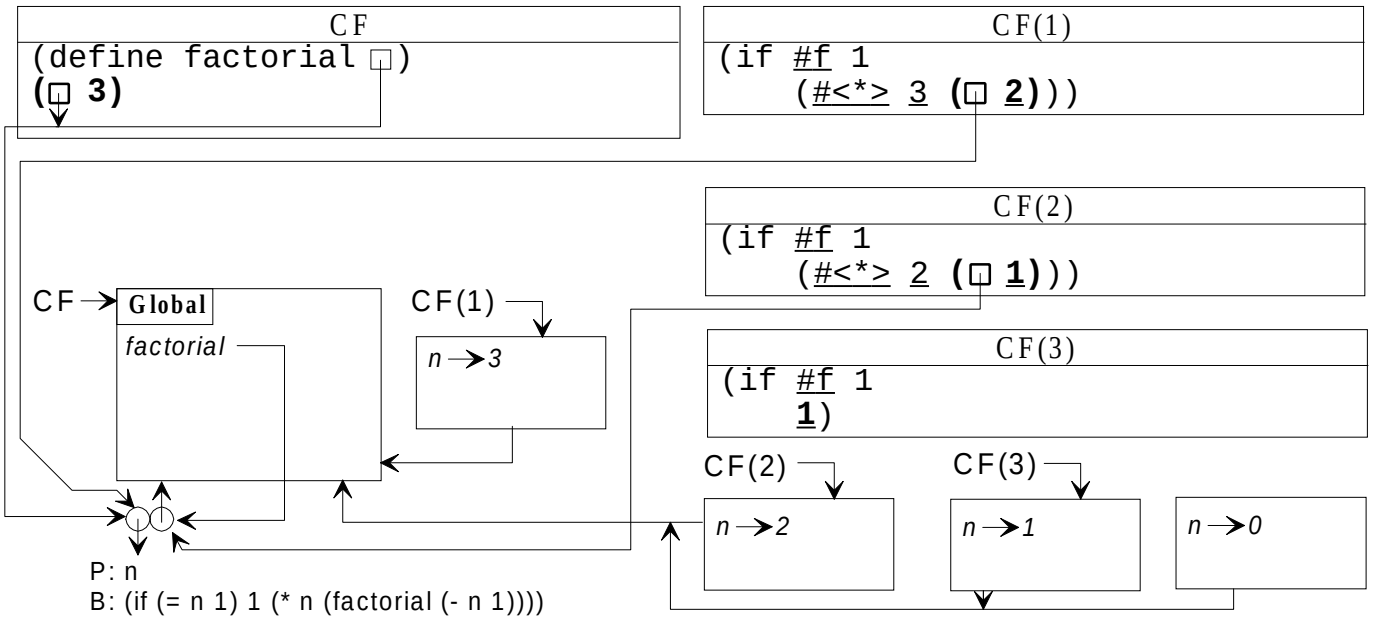
Evaluating the consequent expression, the number *1*, we get itself. Because this is the last expression to evaluate in this procedure, it becomes the resultant value of the procedure call.

Step 23



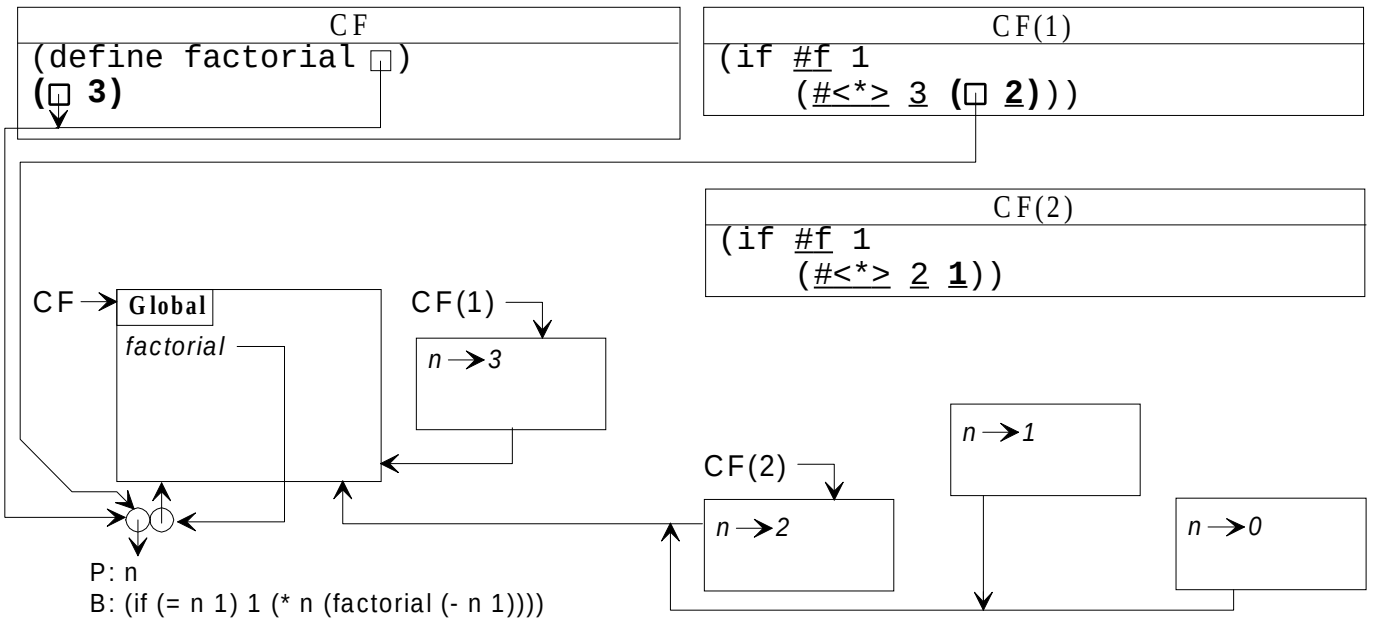
Replacing the expression that called the procedure with the resultant value, 1, we see that we can now finish evaluating the non-special list form. Notice that how the current frame is related to the expressions being evaluated. We called the procedure from CF(3) and the return value replaces the expression in CF(3) and we finish evaluating the rest of the expressions in CF(3).

Step 24



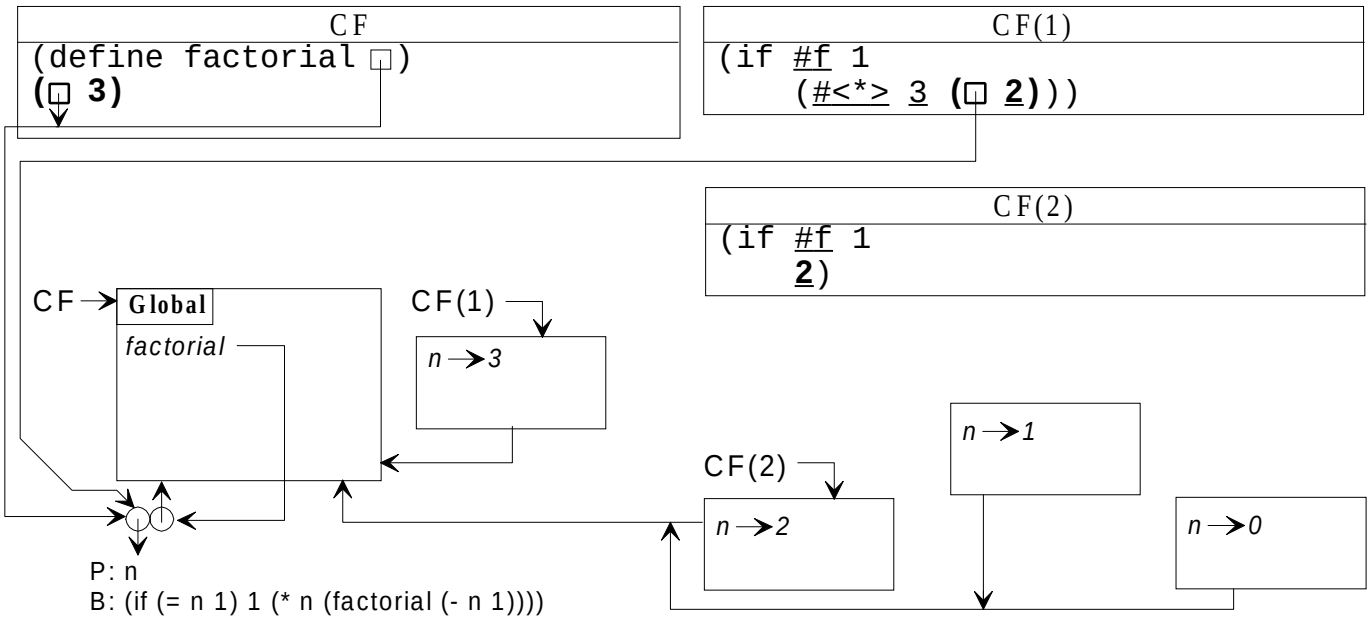
Evaluating the primitive procedure #<*> on the actual parameters 1 and 1, results in the numerical value 1. This is the last expression to evalute in this procedure body, so its result becomes the result of calling this procedure.

Step 25



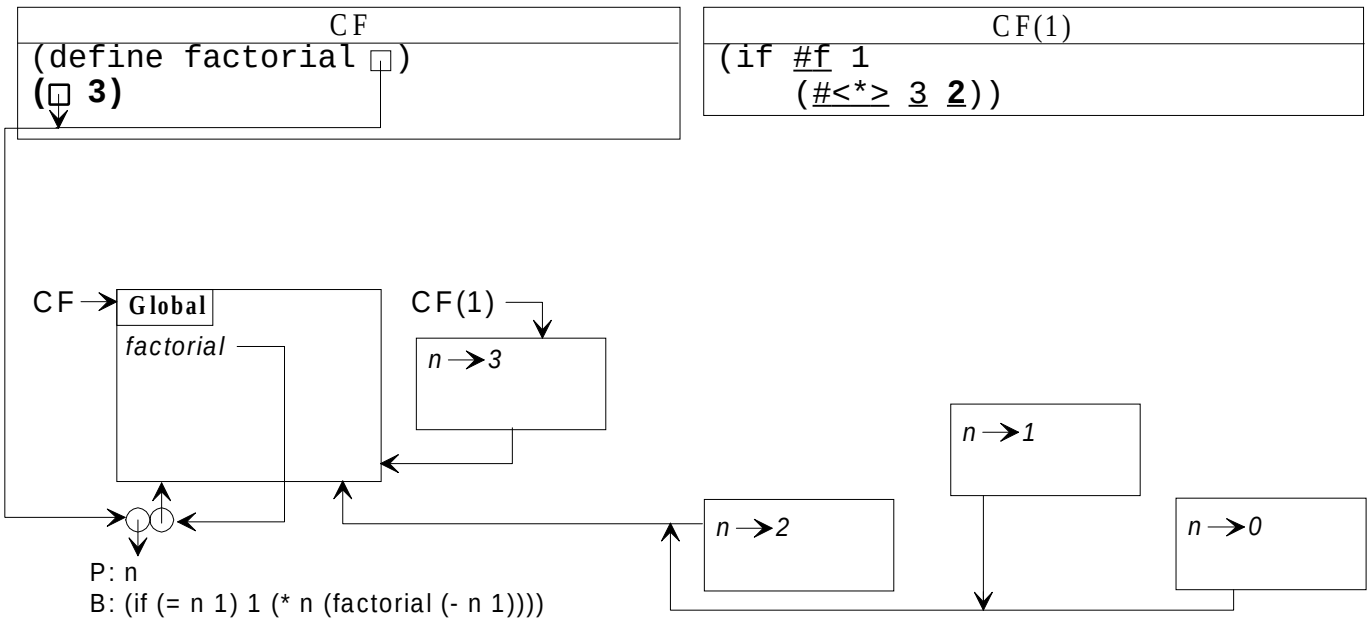
Replacing the expression that called the procedure with the resultant value, 1, we see that we can now finish evaluating the non-special list form. Notice how none of the old frames are being discarded. This is because Scheme conceptually discards nothing!

Step 26



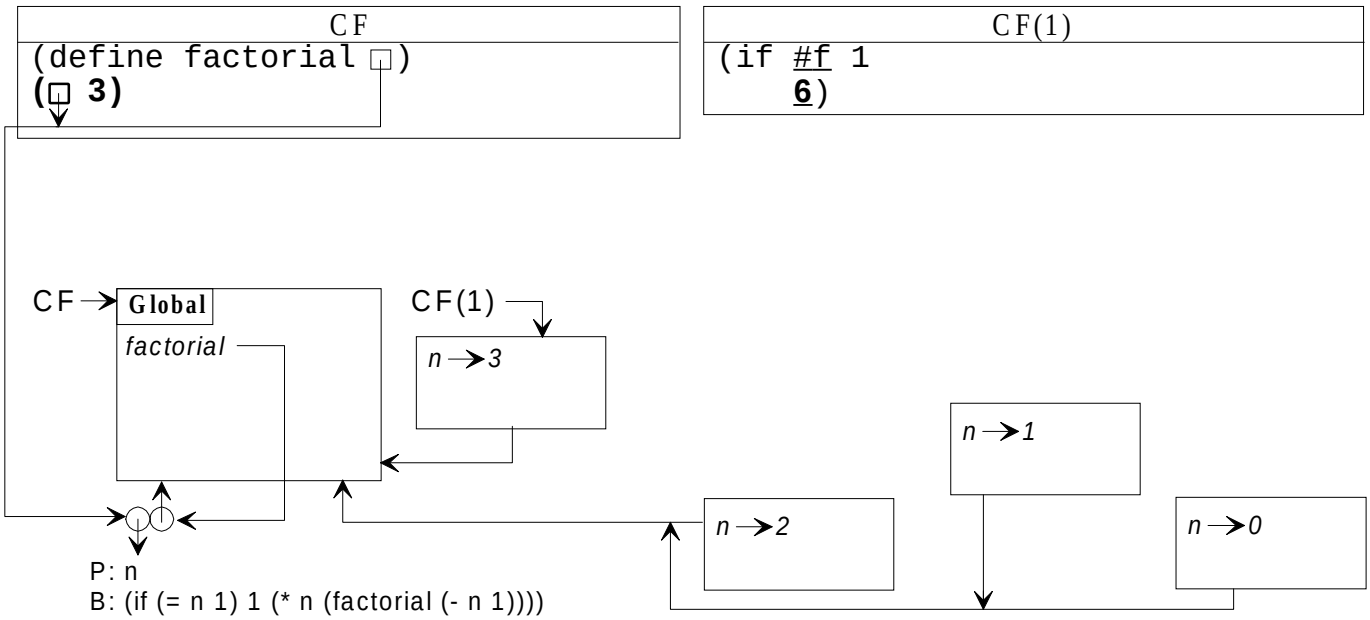
Evaluating the primitive procedure #<*> on the actual parameters 2 and 1, results in the numerical value 2. This is the last expression to evalute in this procedure body, so its result becomes the result of calling this procedure.

Step 27



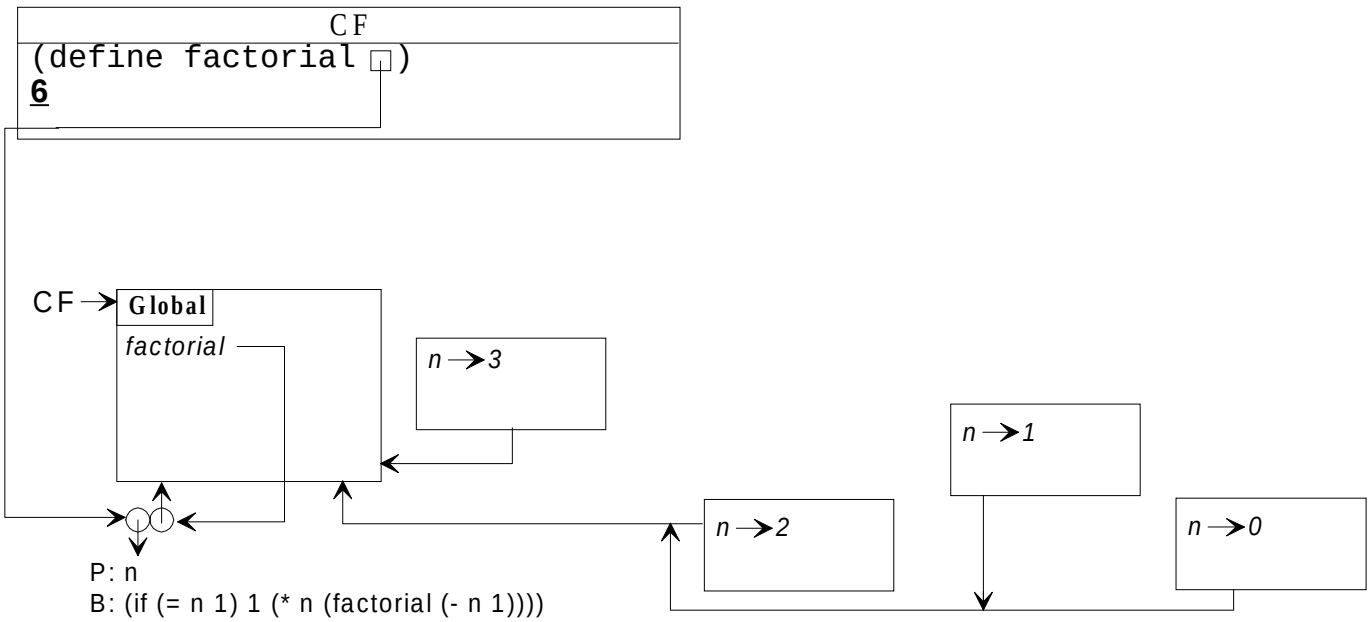
Replacing the expression that called the procedure with the resultant value, 2, we see that we can now finish evaluating the non-special list form. We're almost done!

Step 28



Evaluating the primitive procedure #<*> on the actual parameters 3 and 2, results in the numerical value 6. This is the last expression to evalute in this procedure body, so its result becomes the result of calling this procedure.

Step 29



Replacing the expression that called the procedure with the resultant value, 6, we see that we've finished the evaluation of the initial expression! Well done!

Step 30