# Sample Execution

## Contents



---

Here's an example to see what's happening with the Scheme interpreter. I'll go through the steps of sorting a list with the version of Quicksort we discussed in class.

To make environments easier to read, I'll present them with tables. An association list is written as a table

```
scope-name:
    name1    value1
    name2    value2
```

and an environment is written as a list of such tables. If there is a cycle in the data structure, I just refer to the name of the environment.

## Define **quicksort**, **hi**, **lo**, **filter**

Let's start and define `quicksort`:

```
(define (quicksort ls)
  (if (null? ls) '()
    (let ((p (car ls)))
      (append (quicksort (lo p ls))
              (list p)
              (quicksort (hi p ls))))))
```

This results in the environment

```
global-env:
    global-scope:
        quicksort    Closure((lambda (ls) (if ...)), global-env)
```

*I.e.*, we make a lambda expression out of the `define` and construct a closure that refers back to the global environment.

Next, we define `lo`:

```
(define (lo p ls)
  (filter (lambda (x) (> p x)) ls))
```

Now we change the first element of `global-env`, by adding an element to the association list. By using `set-car!` to change the `global-scope` without touching the top cons-cell of `global-env` itself, the reference in the closure still points to the whole thing:

```
global-env:
    global-scope:
        lo           Closure((lambda (p ls) (filter ..)), global-env)
        quicksort    Closure((lambda (ls) (if ...)), global-env)
```

Similarly for defining `hi` and `filter`:

```
(define (hi p ls)
  (filter (lambda (x) (< p x)) ls))
```

```
(define (filter pred ls)
  (cond
    ((null? ls)
        '())
    ((pred (car ls))
        (cons (car (ls) (filter pred (cdr ls)))))
    (else
        (filter pred (cdr ls)))))
```

We get the environment:

```
global-env:
    global-scope:
        filter      Closure((lambda (pred ls) ...), global-env)
        hi          Closure((lambda (p ls) (filter ..)), global-env)
        lo          Closure((lambda (p ls) (filter ..)), global-env)
        quicksort   Closure((lambda (ls) (if ...)), global-env)
```

Everything that happened so far was done by `eval`.

## Calling `(quicksort '(2 1 3))`

Now let's call

```
(quicksort '(2 1 3))
```

`Eval` gets called with

```
(eval '(quicksort (quote (2 1 3))) global-env)
```

`Eval` evaluates the elements of the list. For evaluating `quicksort`, we look it up in the environment and find the closure. For evaluating the quote, we just extract the list contained in it. `Eval` then calls `apply`:

```
(apply 'Closure((lambda (ls) (if ...)), global-env) '(2 1 3))
```

Note: I'm using a C++-style constructor expression for representing the closure.

Now `apply` creates the scope for `quicksort` in which the parameter is bound to the argument:

```
    quicksort-scope:
        ls          (2 1 3)
```

Then we take the environment in which `quicksort` was defined out of the closure, put the new scope (association list) in front of it, and construct this way the environment for the body of `quicksort`:

```
quicksort-body-env:
    quicksort-scope:
        ls          (2 1 3)
    global-scope:
        filter      Closure((lambda (pred ls) ...), global-env)
        hi          Closure((lambda (p ls) (filter ..)), global-env)
        lo          Closure((lambda (p ls) (filter ..)), global-env)
        quicksort   Closure((lambda (ls) (if ...)), global-env)
```

The `global-env` is still around. *I.e.*, the closures in `global-scope` still point to `global-env`.

Now we evaluate the body of `quicksort`, find out that `ls` is not the empty list, and evaluate the let-expression. For the `let`, we need to create another scope again and put it in front of `quicksort-body-env`:

```
quicksort-let-env:
    quicksort-let-scope
        p           2
    quicksort-scope:
        ls          (2 1 3)
    global-scope:
        filter      Closure((lambda (pred ls) ...), global-env)
        hi          Closure((lambda (p ls) (filter ..)), global-env)
        lo          Closure((lambda (p ls) (filter ..)), global-env)
        quicksort   Closure((lambda (ls) (if ...)), global-env)
```

In the body of the `let`, we call now `(lo p ls)`. Again, `eval` looks up the values of the variables `lo`, `p`, and `ls`, and calls `apply` by passing the closure for `lo` and the list of parameters.

```
(apply 'Closure((lambda (p ls) ...), global-env) '(2 (2 1 3)))
```

Again, `apply` creates a new scope for `lo`:

```
    lo-scope:
        p            2
        ls           (2 1 3)
```

Again, `apply` takes the environment in which `lo` was defined out of the closure and puts the new scope in front:

```
lo-env:
    lo-scope:
        p            2
        ls           (2 1 3)
    global-scope:
        filter       Closure((lambda (pred ls) ...), global-env)
        hi           Closure((lambda (p ls) (filter ..)), global-env)
        lo           Closure((lambda (p ls) (filter ..)), global-env)
        quicksort    Closure((lambda (ls) (if ...)), global-env)
```

Note, that the environments `quicksort-let-env`, `quicksort-body-env`, and `global-env` are still around. All of them share the same `global-scope`.

In `lo`, it gets interesting now, since we have a lambda expression there. `Eval` evaluates the body of `lo`, *i.e.*, the call to `filter`, in the environment `lo-env`. Evaluating `filter` and `ls` is easy, we just look up the values in the environment. For evaluating the lambda expression, we form a new closure. Since the lambda expression was defined inside `lo-env`, the environment pointer in the closure points to `lo-env`. So we end up with the following call to `apply`:

```
(apply 'Closure((lambda (pred ls) ...), global-env)    ;; filter
       '(Closure((lambda (x) (> p x)), lo-env)         ;; new closure
       (2 1 3)))                                       ;; ls
```

`Apply`, in turn creates the scope for `filter` and puts it in front of `global-env`:

```
filter-env:
    filter-scope:
        pred         Closure((lambda (x) (> p x)), lo-env)
        ls           (2 1 3)
    global-scope:
        filter       Closure((lambda (pred ls) ...), global-env)
        hi           Closure((lambda (p ls) (filter ..)), global-env)
        lo           Closure((lambda (p ls) (filter ..)), global-env)
        quicksort    Closure((lambda (ls) (if ...)), global-env)
```

When evaluating the body of `filter`, we find out that `ls` is not the empty list, and proceed to call `(pred (car ls))`. `Eval` looks up `pred` in the environment, gets the closure out of it, calls `apply` to get the result of `(car ls)` and then calls `apply` as follows:

```
(apply 'Closure((lambda (x) (> p x)), lo-env) '(2))
```

Now `apply` again extracts the environment from the closure. This time, the environment is `lo-env`, since that's where the lambda- expression was defined. Then we create a scope for the lambda expression and put it in front of `lo-env`:

```
lambda-env:
    lambda-scope:
        x           2
    lo-scope:
        p           2
        ls          (2 1 3)
    global-scope:
        filter      Closure((lambda (pred ls) ...), global-env)
        hi          Closure((lambda (p ls) (filter ..)), global-env)
        lo          Closure((lambda (p ls) (filter ..)), global-env)
        quicksort   Closure((lambda (ls) (if ...)), global-env)
```

Now, when we evaluate the body of the lambda expression, we find `p` in the scope of `lo` just as it was at the time the lambda was defined.

Now let's fast-forward. This call to the lambda expression returns `#f`, the `filter` call eventually returns the list `'(1)`, since `1` was the only element in the list `'(2 1 3)` that was less than `2`. `Lo` then returns `'(1)` as well, and we end up back in `quicksort`.

The environment we had, when we called `lo` is still there:

```
quicksort-let-env:
    quicksort-let-scope:
        p           2
    quicksort-scope:
        ls          (2 1 3)
    global-scope:
        filter      Closure((lambda (pred ls) ...), global-env)
        hi          Closure((lambda (p ls) (filter ..)), global-env)
        lo          Closure((lambda (p ls) (filter ..)), global-env)
        quicksort   Closure((lambda (ls) (if ...)), global-env)
```

Now we have a recursive call to `quicksort` with the argument `'(1)`. *I.e.*, we call `apply` as follows:

```
(apply 'Closure((lambda (ls) (if ...)), global-env) '(1))
```

which results in the following environment for the recursive call:

```
quicksort-2-env:
    quicksort-2-scope:
        ls          (1)
    global-scope:
        filter      Closure((lambda (pred ls) ...), global-env)
        hi          Closure((lambda (p ls) (filter ..)), global-env)
        lo          Closure((lambda (p ls) (filter ..)), global-env)
        quicksort   Closure((lambda (ls) (if ...)), global-env)
```

When the recursive call returns, it'll return with the sorted list `'(1)`. Then we call `(hi 2 '(2 1 3))`, which returns `'(3)`. Again, we call `quicksort`, this time with the environment

```
quicksort-3-env:
    quicksort-3-scope:
        ls          (3)
    global-scope:
        filter      Closure((lambda (pred ls) ...), global-env)
        hi          Closure((lambda (p ls) (filter ..)), global-env)
        lo          Closure((lambda (p ls) (filter ..)), global-env)
        quicksort   Closure((lambda (ls) (if ...)), global-env)
        quicksort   Closure((lambda (ls) (if ...)), global-env)
```

When this recursive call returns, it'll return with the sorted list `'(3)`. Now we are back to the environment `quicksort-let-env`, and call

```
(apply 'append '((1) (2) (3)))
```

which returns the sorted list `'(1 2 3)`. Now the let-expression is done and the outermost call to `quicksort` returns with the sorted list `'(1 2 3)`.

### When a function returns another function

What we didn't see in this example is what happens when a function returns another function. Suppose `lo` would return the lambda-expression instead of passing it to `filter`, and that we would call the lambda-expression after `lo` has terminated.

Nothing changes. The environment pointer in the closure resulting from the lambda-expression still would point to `lo-env` even though `lo` has terminated already. Since all these environments (lists) have been allocated on the heap instead of on the run-time stack, they stay around. So when the body of that lambda would then be evaluated, we still would have the original value of `p` the way it was in `lo`.