Local Expressions, continued

Lecture 18
13 November 2019
Recall: local
The `local` form has definitions and a body:

```
(local [⟨definition 1⟩
       ⟨definition 2⟩
       ...
       ⟨definition n⟩]
  ⟨body⟩)
```
Global/top-level scope

```
(define a 1)
(define b 2)

(+ a
  (local [(define b 3)]
    (+ a b)))
```

Local scope
Nesting environments

Each new `local` environment lies inside the previous one.

To find the value of a variable, Racket looks at the innermost environment first, and proceeds outward until finding a value for the variable.
Uses of `local`

We saw that we can use `local` to:

- Make a function clearer, by giving names to subexpressions.
- Encapsulate functions inside another function so only the function we want the user or other parts of the program to use is visible, and the helpers are hidden.
Using local: Efficiency
“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.”

Donald Knuth, “Computer Programming as an Art”, Turing Award Lecture, 1974
Nonetheless, there are some big problems of efficiency that we want to be sure we take care of. We can use `local` to deal with redundant computation that can lead to this kind of poor performance.
Recall our *files and directories* domain.
home

kate

escape.rkt

map.pdf

hugo

snacks

lotto.zip

4 8 15 16 23 42

;; A Pittsburgh road is ...
Directory depth: 0
Directory depth: 1
Directory depth: 2
Directory depth: 3
;;; make-skinny : Natural -> Directory
;;; Produce a skinny directory tree n+1 levels deep, where the leaf is a file named "Y" of size 1.
(define (make-skinny n)
  (cond [(zero? n)
          (make-file "Y" 1)]
        [else
         (make-dir "X" (list (make-skinny (sub1 n))))]))

(check-expect (make-skinny 0) (make-file "Y" 1))
(check-expect (make-skinny 2) (make-dir "X" (list (make-dir "X" (list (make-file "Y" 0))))))
(make-skinny 10) will produce 10 directories named "X" and then a file "Y".

(make-skinny 100) will produce 100 directories named "X" and then a file "Y".
If we want to search a directory tree to find the size of a file with a given name, we could use our template with encapsulated helper functions to quickly write it.
find-size : Directory String -> [Maybe Integer]
;; Search the given directory tree for a file with the given
;; name. Produce its size if found; #false otherwise.
(define (find-size d name)
  (local ([define (fun-for-file f)
            (if (string=? (file-name f) name)
              (file-size f)
              #false))
            [define (fun-for-file-or-directory fod)
              (cond [[(file? fod) (fun-for-file fod)]
                     [([dir? fod) (find-size fod name)]))]
            [define (fun-for-lofd lofd)
              (cond [[[empty? lofd) #false]
                     [([cons? lofd)
                       (if (not (false?
                                (fun-for-file-or-directory
                                 (first lofd))))
                         (fun-for-file-or-directory
                          (first lofd)))]
                       (fun-for-file-or-directory
                        (first lofd))
                       (fun-for-lofd (rest lofd)))]])]
            (fun-for-lofd (dir-content d))))
Let’s look at how long it takes to run our \texttt{find-size} function on directory trees of different depths.

To do this, we can use a new Racket primitive named \texttt{time}, which return the time it takes to evaluate its operands:

\begin{verbatim}
> (time (+ 1 2))
cpu time: 1 real time: 0 gc time: 0 3
\end{verbatim}
> (time (find-size (make-skinny 10) "Y"))
cpu time: 1 real time: 2 gc time: 0
1
> (time (find-size (make-skinny 11) "Y"))
cpu time: 3 real time: 3 gc time: 0
1
> (time (find-size (make-skinny 12) "Y"))
cpu time: 5 real time: 6 gc time: 0
1
> (time (find-size (make-skinny 13) "Y"))
cpu time: 10 real time: 10 gc time: 0
1
> (time (find-size (make-skinny 14) "Y"))
cpu time: 20 real time: 21 gc time: 0
1
> (time (find-size (make-skinny 15) "Y"))
cpu time: 47 real time: 47 gc time: 0
1
> (time (find-size (make-skinny 16) "Y"))
cpu time: 95 real time: 94 gc time: 0
1
> (time (find-size (make-skinny 17) "Y"))
cpu time: 185 real time: 184 gc time: 0
1
;; find-size : Directory String -> [Maybe Integer]
;; Search the given directory tree for a file with the given
;; name. Produce its size if found; #false otherwise.
(define (find-size d name)
  (local [(define (fun-for-file f)
             (if (string=? (file-name f) name)
                (file-size f)
                #false))
           (define (fun-for-file-or-directory fod)
             (cond [[(file? fod) (fun-for-file fod)]
                    [[(dir? fod) (find-size fod name)]])
           (define (fun-for-lofd lofd)
             (cond [[(empty? lofd) #false]
                    [(cons? lofd)
                       (if (not (false?
                                  (fun-for-file-or-directory (first lofd))))
                           (fun-for-file-or-directory (first lofd))
                           (fun-for-lofd (rest lofd)))]
               (fun-for-lofd (dir-content d))))

Here’s the performance problem:
search tree with 10 Xs
search tree with 10 Xs

search tree with 9 Xs

search tree with 9 Xs
search tree with 10 Xs

<table>
<thead>
<tr>
<th>search tree with 9 Xs</th>
<th>search tree with 9 Xs</th>
</tr>
</thead>
<tbody>
<tr>
<td>search tree with 8 Xs</td>
<td>search tree with 8 Xs</td>
</tr>
</tbody>
</table>

2

4
search tree with 10 Xs

search tree with 9 Xs

search tree with 8 Xs

search tree with 7 Xs

search tree with 6 Xs

search tree with 5 Xs

search tree with 4 Xs

search tree with 3 Xs

search tree with 2 Xs

search tree with 1 Xs

and so on
(define (find-size d name)
  (local [(define (fun-for-file f)
            (if (string=? (file-name f) name)
                (file-size f)
                #false))]
    (define (fun-for-file-or-directory fod)
      (cond [(file? fod) (fun-for-file fod)]
            [(dir? fod) (find-size fod name)]))
    (define (fun-for-lofd lofd)
      (cond [(empty? lofd) #false]
            [(cons? lofd)
              (if (not (false?
                           (fun-for-file-or-directory (first lofd))))
                  (fun-for-file-or-directory (first lofd))
                  (fun-for-lofd (rest lofd))))]
      (fun-for-lofd (dir-content d))))
(define (find-size d name)
  (local [(define (fun-for-file f)
            (if (string=? (file-name f) name)
                (file-size f)
                #false))]
    (define (fun-for-file-or-directory fod)
      (cond [(file? fod) (fun-for-file fod)]
            [(dir? fod) (find-size fod name)]))
    (define (fun-for-lofd lofd)
      (cond [(empty? lofd) #false]
            [(cons? lofd)
             (local [(define try
                         (fun-for-file-or-directory (first lofd)))]
               (if (not (false? try))
                   try
                   (fun-for-lofd (rest lofd))))]))
    (fun-for-lofd (dir-content d))))
In this case, we got a big speed up because the repeated computation was in a recursive function.

If we had a repeated computation in a non-recursive function, the improvement could be much smaller, and using `local` might just make the program harder to read.
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Gregor Kiczales
Shriram Krishnamurthi
Marc Smith