Local Expressions

Lecture 17

11 November 2019
Computer Science

SUMMER OPPORTUNITIES PANEL

Come hear CS students talk about their summer research and internship experiences

WEDNESDAY 11/13
3:00PM IN SP105

Panelists:
Lindsey Coffee-Johnson, '20 - Research, Marquette Univ.
Eve Coulibaly, '20 - Intern, Google + YouTube
Colby Morrison, '20 - Intern, IBM + Facebook
Amy O'Connell '20 - Research, Univ. of South Carolina
Mohamad Safadieh, '21 - URSI
Frank Swiatowicz, '20 - Research, Clarkson Univ.
So random
The **random** operator is strange – it doesn’t return the same result every time for the same input:

```scheme
> (random 3)
0
> (random 3)
2
> (random 3)
1
> (random 3)
2
```
Suppose we need a `random-symbol` function:

```plaintext
> (random-symbol 'x 'y 'z)
'y
> (random-symbol 'x 'y 'z)
'x
> (random-symbol 'x 'y 'z)
'y
> (random-symbol 'x 'y 'z)
'z
```
Suppose we need a `random-symbol` function:

```lisp
> (random-symbol 'x 'y 'z)
 'y
> (random-symbol 'x 'y 'z)
 'x
> (random-symbol 'x 'y 'z)
 'y
> (random-symbol 'x 'y 'z)
 'z
```

Can we implement it using `random`?
 ;; Symbol Symbol Symbol → Symbol
 ;; Randomly return one of the three input symbols
 (define (random-symbol a b c)
  (cond [(= (random 3) 0) a]
        [(= (random 3) 1) b]
        [(= (random 3) 2) c])))
;; Symbol Symbol Symbol Symbol -> Symbol
;; Randomly return one of the three input symbols
(define (random-symbol a b c)
  (cond 
    [(= (random 3) 0)
     a]
    [(= (random 3) 1)
     b]
    [(= (random 3) 2)
     c]])

This won’t work because every time we call random, we may produce a different result.
On the other hand, we know we can save a value and use it multiple times:

\[
\text{(define N (random 3))}
\]
\[
\text{(list N N N)}
\]

Produces

\[
\text{(list 0 0 0)},
\]
\[
\text{(list 1 1 1), or}
\]
\[
\text{(list 2 2 2)}.
\]

The expression (random 3) is only evaluated once, when we define N.
So, does this work?

```
(define N (random 3))

;; Symbol Symbol Symbol -> Symbol
;; Randomly return one of the three input symbols
(define (random-symbol a b c)
  (cond [(= N 0) a]
        [(= N 1) b]
        [(= N 2) c])))
```

Not quite! It always picks the same symbol!
To get a different random symbol each time we call \texttt{random-symbol}, we need
\begin{verbatim}
(define N (random 3))
\end{verbatim}
to be local to our function’s body.
This works, in the Intermediate student (or Intermediate student with lambda) language:

```scheme
;; Symbol Symbol Symbol Symbol -> Symbol
;; Randomly return one of the three input symbols
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))
```
The `local` form has definitions and a body:

```
(local [⟨definition 1⟩
       ⟨definition 2⟩
       ⋮
       ⟨definition n⟩]
       ⟨body⟩)
```

Local definitions are only visible in the body.

Local definitions are evaluated only when the `local` is evaluated.

The result of `local` is the result of its body.
Evaluation with local
Evaluation with local

(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
      [(= n 1) b]
      [(= n 2) c])))

(random-symbol 'arthur 'ford 'marvin)

(random-symbol 'arthur 'ford 'marvin)
Evaluation with local

(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
      [(= n 1) b]
      [(= n 2) c])))

(local [(define n (random 3))]
  (cond [(= n 0) 'arthur]
    [(= n 1) 'ford]
    [(= n 2) 'marvin]))

(random-symbol 'arthur 'ford 'marvin)
Evaluation with local

(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
      [(= n 1) b]
      [(= n 2) c])))

(define n17 (random 3))
(cond [(= n17 0) 'arthur]
     [(= n17 1) 'ford]
     [(= n17 2) 'marvin]))

(random-symbol 'arthur 'ford 'marvin)
Evaluation with `local`

```
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))

(define n17 1)
(cond [(= n17 0) 'arthur]
      [(= n17 1) 'ford]
      [(= n17 2) 'marvin]))
```

`(random-symbol 'arthur 'ford 'marvin)

*The definition is evaluated*
Evaluation with `local`

```scheme
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))

(define n17 1)
(cond [(= 1 0) 'arthur]
     [(= n17 1) 'ford]
     [(= n17 2) 'marvin]))

(random-symbol 'arthur 'ford 'marvin)
```

*Evaluation of a constant name finds the value*
Evaluation with `local`

```
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))

(define n17 1)
(cond [#false 'arthur]
     [(= n17 1) 'ford]
     [(= n17 2) 'marvin])

(random-symbol 'arthur 'ford 'marvin)
```
Evaluation with `local`

```scheme
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))

(define n17 1)
( cond [(= n17 1) 'ford]
       [(= n17 2) 'marvin] )

(random-symbol 'arthur 'ford 'marvin)
```

When a condition evaluates to `#false`, the case is removed from the `cond`. 
Evaluation with `local`

```lisp
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c]))

(define n17 1)
(cond [(= 1 1) 'ford]
     [(= n17 2) 'marvin])

(random-symbol 'arthur 'ford 'marvin)
```
Evaluation with local

(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))

(define n17 1)
(cond [#true 'ford]
     [(= n17 2) 'marvin]))

(random-symbol 'arthur 'ford 'marvin)
Evaluation with `local`

```
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))

(define n17 1)
'ford

(random-symbol 'arthur 'ford 'marvin)
```

When a condition evaluates to `#true`, the whole `cond` is replaced by its consequent ("then")
Evaluation with \texttt{local}

\begin{verbatim}
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))

(define n17 1)
'ford

(local [(define n (random 3))]
  (cond [(= n 0) 'arthur]
        [(= n 1) 'ford]
        [(= n 2) 'marvin]))
\end{verbatim}
Evaluation with `local`

```scheme
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
      [(= n 1) b]
      [(= n 2) c])))

(define n17 1)
'ford

(define n45 (random 3))
(cond [(= n45 0) 'arthur]
     [(= n45 1) 'ford]
     [(= n45 2) 'marvin]))
```

Evaluation of `local` picks a new name each time.
Evaluation with local

(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))

(define n17 1)
'ford

(define n45 0)
(cond [(= n45 0) 'arthur]
     [(= n45 1) 'ford]
     [(= n45 2) 'marvin]))
Evaluation with local

(define (random-symbol a b c)
  (local [((define n (random 3)))]
    (cond [ (= n 0) a]
      [ (= n 1) b]
      [ (= n 2) c])))

(define n17 1)
'ford

(define n45 0)
(cond [ (= 0 0) 'arthur]
      [ (= n45 1) 'ford]
      [ (= n45 2) 'marvin]])
Evaluation with local

(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))

(define n17 1)
  'ford

(define n45 0)
  (cond [#true 'arthur]
        [(= n45 1) 'ford]
        [(= n45 2) 'marvin]))
Evaluation with \texttt{local}

\begin{verbatim}
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))

(define n17 1)
  'ford

(define n45 0)
  'arthur
\end{verbatim}
Lexical scope
(define p "incendio ")

(local [[[define p "accio ")
            (define (fetch n)
                (string-append p n))]]
         (fetch "portkey"))

Will this produce "incedio portkey" or "accio portkey"?
Global/top-level scope

```
(define p "incendio ")

(local [(define p "accio ")
          (define (fetch n)
              (string-append p n))]
          (fetch "portkey"))
```
Global/top-level scope

```scheme
(define p "incendio ")
```

Local scope

```scheme
(local [(define p "accio ")
    (define (fetch n)
      (string-append p n))]
  (fetch "portkey"))
```
Global/top-level scope

(define p "incendio ")

Local scope

(local [(define p "accio ")
   (define (fetch n)
     (string-append p n))]
   (fetch "portkey"))
(define a 1)
(define b 2)

(+ a
  (local [(define b 3)]
    (+ a b))
b)
Global/top-level scope

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

Local scope

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

Global/top-level scope

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

Local scope

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

Global/top-level scope

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

Local scope

```
(+ a
 (local [(define b 3)]
   (+ a b))
 b)
```
(define n 42)

(define (random-symbol a b c)
  (local [(define (real-random-symbol a b c)
            (local [(define n (random 3))]
                (cond [(= n 0) a]
                      [(= n 1) b]
                      [(= n 2) c]))])))
  (if (and (symbol? a) (symbol? b) (symbol? c))
      (real-random-symbol a b c)
      (error 'random-symbol "not a symbol")))
(define \text{n} 42)

(define \text{random-symbol} \text{a} \text{b} \text{c})

(local [[(define \text{real-random-symbol} \text{a} \text{b} \text{c})

  (local [[(define \text{n} (random 3))]

    (cond [ (= \text{n} 0) \text{a}] 
            [ (= \text{n} 1) \text{b}] 
            [ (= \text{n} 2) \text{c}]))])

  (if (and (symbol? \text{a}) (symbol? \text{b}) (symbol? \text{c}))
      (real-random-symbol \text{a} \text{b} \text{c})
      (error 'random-symbol "not a symbol"))))
(define n 42)

(define (random-symbol a b c)
  (local [(define (real-random-symbol a b c)
               (local [(define n (random 3))]
                 (cond [ (= n 0) a]
                       [ (= n 1) b]
                       [ (= n 2) c)]))]
    (if (and (symbol? a) (symbol? b) (symbol? c))
      (real-random-symbol a b c)
      (error 'random-symbol "not a symbol"))))

Run Check Syntax and hover over variables!
(define n 42)

(define (random-symbol a b c)
  (local [(define (real-random-symbol a b c)
           (local [(define n (random 3))]
             (cond [(= n 0) a]
                   [(= n 1) b]
                   [(= n 2) c]))]
        (if (and (symbol? a) (symbol? b) (symbol? c))
            (real-random-symbol a b c)
            (error 'random-symbol "not a symbol"))))

Bold-italic a could be changed to z without affecting non-bold-italic a, no matter how the code runs.
Using **local**: *Clarity*
Using `local`, we can give more meaningful names to expressions:

```
(define (distance pt1 pt2)
  (local [(define x1 (first pt1))
            (define y1 (second pt1))
            (define x2 (first pt2))
            (define y2 (second pt2))]
    (sqrt (+ (sqr (- x2 x1))
             (sqr (- y2 y1))))))

> (distance '(1 2) '(3 4))
2.8284271247461903
```
Using local: Encapsulation
When we’ve had mutually recursive functions, we’ve defined them all at the top-level.

Given these data definitions:

```scheme
;; A Directory is
;; (make-dir String LOFD)
(define-struct dir [name content])

;; A File is a String

;; A FileOrDirectory is either:
;; - File
;; - Directory

;; A LOFD is either:
;; - '()
;; - (cons FileOrDirectory LOFD)
```
We wrote:

```plaintext
;; find :
;;   Directory String -> Boolean

;; find-for-file :
;;   File String -> Boolean

;; find-for-file-or-directory :
;;   FileOrDirectory String -> Boolean

;; find-for-lofd :
;;   LOFD String -> Boolean
```

But the only one another programmer or a user wants to see is `find`; the other functions are just for making `find` work.
We can encapsulate those functions inside `find`:

```scheme
;; Directory String -> Boolean
;; Return true if a File or a Directory with name `n`
;; exists somewhere under Directory `d`
(define (find d name)
  (local [(define (find-for-file ... ) ...)
            (define (find-for-file-or-directory ... ) ...)
            (define (find-for-lofd ... ) ...)]
    (or (string=? (dir-name d) name)
        (find-for-lofd (dir-content d) name))))
```

The functions can still call each other, but they’re not exposed at the top-level.

Because of this, `find` is the only one of the functions we need to publish a signature for, and the only one of them we need to write tests for.
A good candidate for encapsulation is a function that has one or more helpers closely linked to it, where the outside program really only wants to call the main function, not the helpers.
Try doing the same for du or for one of the rumor mill functions.
Since we end up encapsulating our functions, maybe we should have encapsulation in our template!

If we do this, we can do less renaming. It’s fine for every one of our functions that operates on a directory to have `local` functions that have the names `fun-for-file, fun-for-dir`, etc.
fun-for-dir : Directory -> ...

Template for a function that consumes a Directory

(define fun-for-dir d)
  (local [(define fun-for-file f)
           (... (file-name f) ...
                (file-size f))]
          (define fun-for-file-or-directory fod)
          (cond [(file? fod)
                   (... (fun-for-file fod))]
                [(dir? fod)
                 (... (fun-for-dir fod))])]
          (define fun-for-lofd lofd)
          (cond [(empty? lofd) ...]
                [(cons? lofd)
                 (... (fun-for-file-or-directory (first lofd)) ...
                      (fun-for-lofd (rest lofd))))])]
  (... (dir-name d) ...
       (fun-for-lofd (dir-content d) ...))
;; Directory String -> Boolean
;; Return true if a File or a Directory with name `n`
;; exists somewhere under Directory `d`
(define (find d name)
  (local [(define (fun-for-file f)
            (... (file-name f) ...
                 (file-size f)))
       (define (fun-for-file-or-directory fod)
         (cond [[(file? fod)
                     (... (fun-for-file fod))]
                [(dir? fod)
                 (... (find fod))])]
       (define (fun-for-lofd lofd)
         (cond [[(empty? lofd) ...]
                [(cons? lofd)
                 (... (fun-for-file-or-directory
                         (first lofd)) ...
                        (fun-for-lofd (rest lofd)))]))]
         (... (dir-name d) ...
              (fun-for-lofd (dir-content d)) ...)))
Acknowledgments

This lecture incorporates material from:

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