Scope and Local Expressions

20 October 2020
Assignment 1  Graded
Assignment 1  Graded

Assignment 2  Due at 5 p.m.
Assignment 1  Graded
Assignment 2  Due at 5 p.m.
Assignment 3  Out today
Assignment 1  Graded
Assignment 2  Due at 5 p.m.
Assignment 3  Out today
Exam 1  Graded someday
Assignment 1  Graded
Assignment 2  Due at 5 p.m.
Assignment 3  Out today
Exam 1  Graded someday
Lab 6  Toast?
So random
The `random` operator is strange – it doesn’t return the same result every time for the same input:

```lisp
> (random 3)
0
> (random 3)
2
> (random 3)
1
> (random 3)
2
```
Suppose we need a **random-string** function:

> (random-string "x" "y" "z")
> "y"
> (random-string "x" "y" "z")
> "x"
> (random-string "x" "y" "z")
> "y"
> (random-string "x" "y" "z")
> "z"
Suppose we need a \texttt{random-string} function:

\begin{verbatim}
> (random-string "x" "y" "z")
"y"
> (random-string "x" "y" "z")
"x"
> (random-string "x" "y" "z")
"y"
> (random-string "x" "y" "z")
"z"
\end{verbatim}

Can we implement it using \texttt{random}?
;; String, String, String -> String
;; Randomly return one of the three input strings
(define (random-string a b c)
  (cond [(= (random 3) 0) a]
        [(= (random 3) 1) b]
        [(= (random 3) 2) c])))
;; String, String, String -> String
;; Randomly return one of the three input strings
(define (random-string 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:

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

Produces

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

The expression (\text{random 3}) is only evaluated once, when we define \text{N}.  

So, does this work?

```
(define N (random 3))

;; String, String, String -> String
;; Randomly return one of the three input strings
(define (random-string 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 letter each time we call \texttt{random-string}, we need

\begin{verbatim}
(define N (random 3))
\end{verbatim}

to be local to our function’s body.
This works, in the Intermediate student language:

;; String, String, String -> String
;; Randomly return one of the three input strings
(define (random-string 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-string a b c)
  (local [(define n (random 3))]
    (cond [(= n 0) a]
          [(= n 1) b]
          [(= n 2) c])))

(random-string "arthur" "ford" "marvin")

(random-string "arthur" "ford" "marvin")
Evaluation with local

(define (random-string 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-string "arthur" "ford" "marvin")
Evaluation with `local`

```
(define (random-string 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-string "arthur" "ford" "marvin")
```
Evaluation with `local`

```
(define (random-string 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-string "arthur" "ford" "marvin")
```

*The definition is evaluated*
Evaluation with **local**

```
(define (random-string 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-string "arthur" "ford" "marvin")
```

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

(define (random-string 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-string "arthur" "ford" "marvin")
Evaluation with local

(define (random-string 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-string "arthur" "ford" "marvin")

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

(define (random-string 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-string "arthur" "ford" "marvin")
Evaluation with local

(define (random-string 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-string "arthur" "ford" "marvin")
Evaluation with \texttt{local}

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

(define \texttt{n17 1})
"ford"

(random-string "arthur" "ford" "marvin")
\end{verbatim}

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

\begin{verbatim}
(define (random-string 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`

```
(define (random-string 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-string 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-string 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-string 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 local

(define (random-string 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"
Lexical scope
Will this produce "incendio 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

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

Local scope

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

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

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

Global/top-level scope

Local scope

a, b

a, b

b
(define n 42)

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

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

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

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

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

These as could be changed to z without affecting the as no matter how the code runs.
Using **local**: *Clarity*
Using `local`, we can give more meaningful names to expressions:

```scheme
(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 (list 1 2) (list 3 4))
2.8284271247461903
```
Using `local`: Encapsulation
When we looked at higher-order functions last time, we saw that they let us avoid writing full recursive functions that followed some common patterns.
;; add-3-to-all : [List-of Number] → [List-of Number]
;; Adds 3 to every number
(check-expect (add-3-to-all (list 1 2 3))
 (list 4 5 6))
(define (add-3-to-all lon)
 (map add-three lon))

;; add-three : Number → Number
;; Adds three to the given number
(check-expect (add-three 0) 3)
(check-expect (add-three 107.5) 110.5)
(check-expect (add-three -15) -12)
(define (add-three x)
 (+ x 3))
add-three is a helper function; we only want to use it inside of add-3-to-all. Therefore we can encapsulate it inside that function, keeping them together and hiding the helper.
;; add-3-to-all : [List-of Number] -> [List-of Number]
;; Adds 3 to every number
(check-expect (add-3-to-all (list 1 2 3))
  (list 4 5 6))
(define (add-3-to-all lon)
  (local [;; add-three : Number -> Number
           (define (add-three x)
             (+ x 3))]
    (map add-three lon)))
Acknowledgments

This lecture incorporates material from:

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