CSCI-C311 Programming Languages

Racket: Scope and Binding

Dr. Hang Dinh

1

Reading Assignment for This Lecture

- [Tutorial] Quick: An Introduction to Racket with Pictures
  - https://docs.racket-lang.org/quick/index.html
  - Part 5 - Local Binding
  - Part 7 – Lexical Scope
- The Racket Guide
  - https://docs.racket-lang.org/guide/index.html
  - 2.2.8 Local Binding with `define`, `let`, and `let*`
  - 4.6.3 Recursive Binding: letrec
  - 4.9 Assignment: `set!`

2

Local Binding with `define`

- The `define` form can be used in some places to create local bindings. For example, it can be used inside a function body:

```
#lang slideshow

(define (four p)
  (define two-p (hc-append p p))
  (vc-append two-p two-p));append pictures vertically
```

In a function body, definitions can appear before the body expression.

```
> (four (circle 15))
```

3

Local Binding

- Definitions at the start of a function body are local to the function body.

```
#lang racket
(define (converse s)
  (define (starts? s2) ; local to converse
    (define spaced-s2 (string-append s2 " ")) ; local to starts?
    (string-prefix? s spaced-s2))
  (cond
   [(starts? "hello") "hi!"]
   [(starts? "goodbye") "bye!"]
   [else "huh?"]))
```

```
> (converse "hello")
"huh?"
> (start? "hello")
start?: undefined;
 cannot reference an identifier before its definition
```

4

Local Binding with `let`

- Another way to create local bindings is the `let` form.
- Advantages of `let`
  - it can be used in any expression position.
  - `let` binds many identifiers at once, instead of requiring a separate `define` for each identifier.
- Syntax of `let`:

  `( let ( {[`‹*id*› ‹*expr*›`]}* ) ‹*expr*›⁺ )`
  - Each binding clause is of the form `[`‹*id*› ‹*expr*›`]`, where the ‹*id*› is bound to the result of the ‹*expr*› for use in the body of the `let`.
  - The expressions after the binding clauses are the body of the `let`

5

Local Binding with `let`

- Example 1:

```
(define (checker p1 p2)
  (let ([p12 (hc-append p1 p2)]
        [p21 (hc-append p2 p1)])
    (vc-append p12 p21)))

> (checker (colorize (square 10) "red")
           (colorize (square 10) "black"))
```

6

Local Binding with `let`

- Example 2:

```
> (let ([x (random 4)]
        [o (random 4)])
    (cond
     [(> x o) "X wins"]
     [(> o x) "O wins"]
     [else "cat's game"]))
"O wins"
```

Function call `(random k)` returns a random integer between 0 and `k-1`.

7

Local Binding with `let*`

- The bindings of a `let` form cannot refer to each other.
- The `let*` form allows later clauses to use earlier bindings:

```
> (let* ([x (random 4)]
         [o (random 4)]
         [diff (number->string (abs (- x o)))])
    (cond
     [(> x o) (string-append "X wins by " diff)]
     [(> o x) (string-append "O wins by " diff)]
     [else "cat's game"]))
"O wins by 3"
```

Function call `number->string` converts a number to string.

8

Local Binding with `letrec`

- `let` makes its bindings available only in the body,
- `let*` makes its bindings available to any later binding *expr*,
- `letrec` makes its bindings available to all other *expr*s—even earlier ones. In other words, `letrec` bindings are recursive.
- The *expr*s in a `letrec` form are most often `lambda` forms for recursive and mutually recursive functions

```
> (letrec ([is-even? (lambda (n) (or (= n 0) (is-odd? (- n 1))))]
           [is-odd? (lambda (n) (is-even? (- n 1)))])
          (is-odd? 11))
#t
```

9

Lexical Scope (Static Scope)

- Racket is a *lexically scoped* language, which means that
  - *whenever an identifier is used as an expression, something in the **textual environment** of the expression determines the identifier's binding*.
- A textual environment is one of two things:
  - The global environment
  - Forms (e.g., `let`, `let*`, `lambda`) where identifiers are bound
- This lexical scoping rule applies to identifiers in a lambda body as well as anywhere else.

11

Lexical Scope: Global Definitions

- Identifiers are bound in the global environment with define

```
> (define ten 10)
> ten
10
```

- The values of identifiers bound in the global environment are available everywhere.
  - So, they should be used sparingly
  - Global definitions should normally be reserved for functions and constants
  - However, there are other legitimate uses for global variables.

12

Lexical Scope

- Identifiers bound within a form will *normally* not be defined outside of the form environment. For example,

```
> ([lambda (x y) (+ (* 2 x) y)] 4 5)
13
> (+ x y)
x: undefined;
 cannot reference an identifier before its definition
```

Within this expression, x and y are bound to 4 and 5.

Once the lambda expression has returned a value, the identifiers x and y are no longer defined.

  - Equivalent `let` expression:

```
> (let ([x 4] [y 5])
       (+ (* 2 x) y))
13
```

13

Lexical Scope

- Identifiers bound within a form may be defined outside of the form environment.

```
#lang slideshow
(define (series mk)
  [hc-append 5 (mk 10) (mk 15) (mk 20)])

(define (rb-series mk)
  (vc-append
      (series [lambda (sz) (colorize (mk sz) "red")])
      (series [lambda (sz) (colorize (mk sz) "blue")])))

(define (square n) (filled-rectangle n n))
```

```
> (rb-series circle)
```

```
> (rb-series square)
```

The function returned by the lambda expression contains variable mk, which is defined outside of the function.

The mk in each lambda form refer to the argument of rb-series, since mk is available within the lexical environment where the lambda expressions are defined

14

Lexical Scope

- Another example:

```
#lang slideshow
(define (series mk)
  [hc-append 5 (mk 10) (mk 15) (mk 20)])

(define (rb-maker mk)
   [lambda (sz)
       (vc-append (colorize (mk sz) "red")
                  (colorize (mk sz) "blue"))])
```

```
> (rb-maker circle)
#<procedure:...Pictures-lambda.rkt:6:3>
> (series (rb-maker circle))
```

The function returned by the lambda expression contains variable mk, which is defined outside of the function.

Function `rb-maker` returns a function that remembers the binding of mk.

15

Assignment: `set!`

- Syntax to assign to a variable using `set!`

```
(set! id expr)
```

  - *id* which must be bound in the enclosing environment.
  - The result of the `set!` expression itself is #<void>.

```
(define (make-running-total)
  (let ([n 0])
    (lambda ()
      (set! n (+ n 1))
      n)))
(define win (make-running-total))
(define lose (make-running-total))
```

```
> (win)
1
> (win)
2
> (lose)
1
> (win)
3
```

16