CS 314 Principles of Programming Languages

Lecture 18: Functional Programming

Zheng (Eddy) Zhang

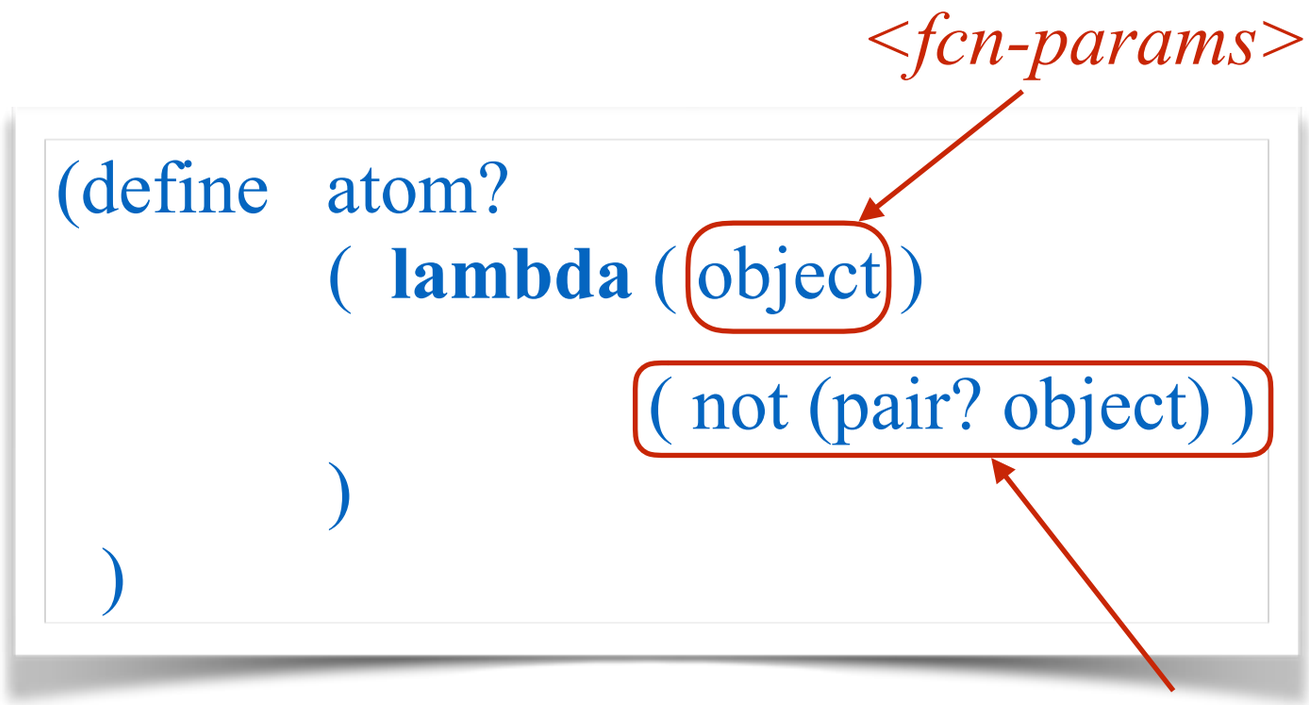
Rutgers University

April 9, 2018

Review: Defining Scheme Functions

(define <fcn-name> **(lambda (** <*fcn-params*> **)** <*expression*> **))**

Example: Given function **pair?** (true for non-empty lists, false o/w)
and function **not** (boolean negation):



Scheme: Functions as First Class Values (Higher-Order)

- **Higher-order functions**
  - Take a function as argument
  - Or return a function as a result

Scheme: Functions as First Class Values (Higher-Order)

Functions as arguments:

(define f (**lambda** (g x) (g x)))

- (f number? 0) ⇒ (number? 0) ⇒ #t
- (f length '(1 2)) ⇒ (length '(1 2)) ⇒ 2
- (f (lambda (x) (* 2 3)) 3) ⇒ ((lambda (x) (* 2 3)) 3) ⇒ (* 2 3) ⇒ 6

Review: Higher-Order Functions

Functions as returned value:

```
(define plusn
        ( lambda ( n )
                ( lambda (x) (+ n x) )
        )
)
```

Return a function

- **(plusn 5)** evaluates to a function that adds 5 to its argument:

- **((plusn 5) 6)**

Review: Higher-Order Functions

In general, any n-ary function

(**lambda** (x_1 x_2 … x_n) e)

can be rewritten as a nest of *n* unary functions:

(**lambda** (x_1)
(**lambda** (x_2)
(… (**lambda**(x_n) e) …)))

Review: Higher-Order Functions

In general, any n-ary function

(**lambda** (x_1 x_2 … x_n) e)

can be rewritten as a nest of *n* unary functions:

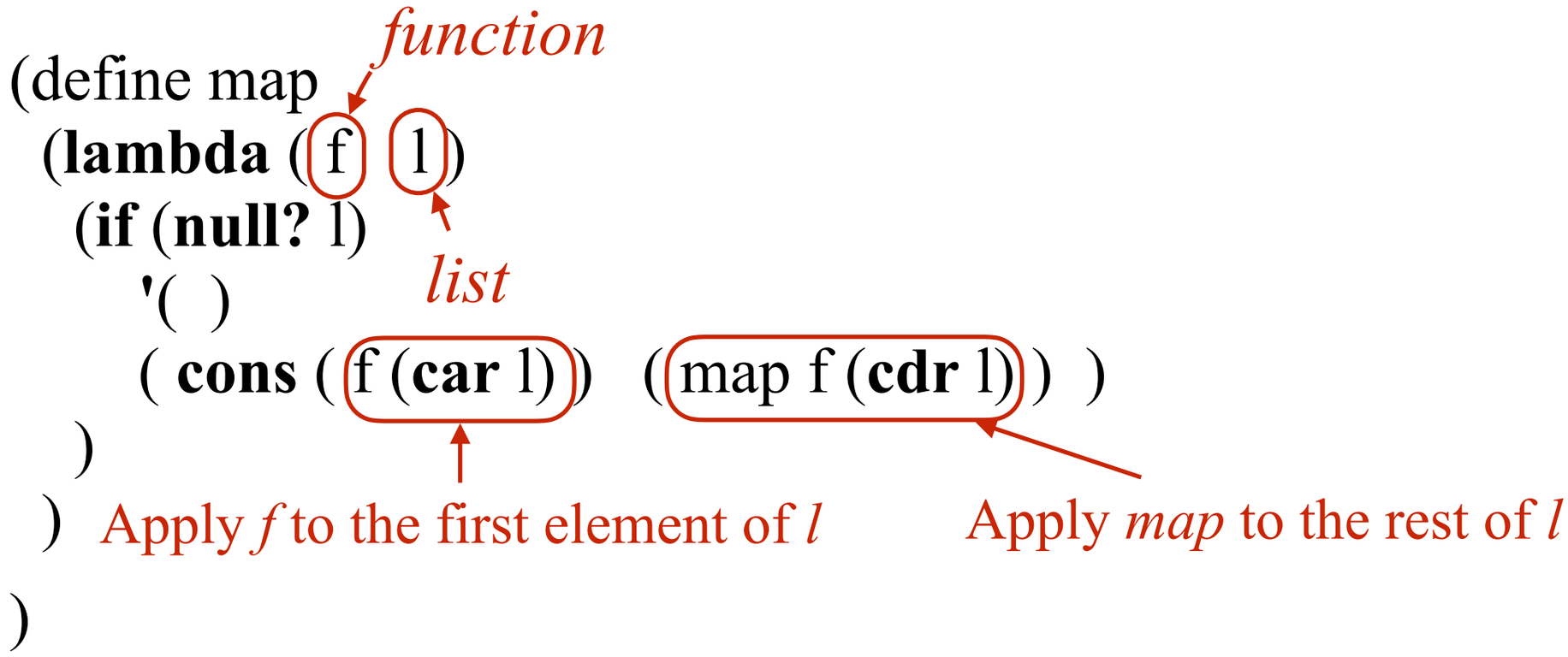
(**lambda** (x_1)
(**lambda** (x_2)
(… (**lambda**(x_n) e) …)))

This translation process is called currying. It means that having functions with multiple parameters do not add anything to the expressiveness of the language:

((**lambda** (x_1 x_2 ... x_n) e) v_1 v_2 ... v_n)

(… (((**lambda** (x_1)
(**lambda** (x_2)
...
(**lambda** (x_n) e)...)) v_1) v_2) ... v_n)

Higher-order Functions: map



- **map** takes two arguments: a function **f** and a list **l**
- **map** builds a new list by applying the function to every element of the (old) list

More on Higher-Order Functions

reduce Higher order function that takes a binary, associative operation and uses it to "roll up" a list

```
(define reduce
  (lambda ( op l id)
            ( if (null? l)
               id
              (op (car l) (reduce op (cdr l) id)) ) ) )
```

Example:

```
(reduce + '(10 20 30) 0) ⇒
(+  10 (reduce + '(20 30) 0)) ⇒
(+  10 (+ 20 (reduce + '(30) 0))) ⇒
(+  10 (+ 20 (+ 30 (reduce + '() 0)))) ⇒
(+  10 (+ 20 (+ 30 0))) ⇒
60
```

Higher-Order Functions

Compose higher order functions to form compact powerful functions

```
(define sum
        (lambda (f  l)
                (reduce + ( map f  l ) 0) ) )
```

(sum (lambda (x) (* 2 x)) '(1 2 3)) ⇒

(reduce (lambda (x y) (+ 1 y)) '(a b c) 0) ⇒

Lexical Scoping and let, let*, **and** letrec

All are variable binding operations:

LET = let, let*, letrec

```
(LET (    (v1 e1)
          (v2 e2)
          …
          (vn en)  )
          e
)
```

Lexical Scoping and let, let*, **and** letrec

- **let**:
  - binds variables to values (no specific order), and evaluate body **e** using bindings
  - new bindings are not effective during evaluation of any e_i
- **let***:
  - binds variables to values in textual order of write-up
  - new binding e_{i-1} is effective for next e_i.
- **letrec**:
  - bindings of variables to values in no specific order
  - independent **evaluations of all** e_i to **values** have to be possible
  - new bindings effective for all e_i
  - mainly used for recursive function definitions

let **and** let* **example**

```
( let ( (a 5) (b 6) )
      ( + a b ) )
```

let **and** let* **example**

```
(  let (  (a 5)  (b 6)  )
        (  + a b  )    )          ;; ⇒ 11
```

let **and** let* **example**

```
( let ( (a 5) (b 6) )
      ( + a b ) )          ;; ⇒ 11
```

```
(let ( (a 5) (b (+ a 6)) )
     ( + a b )       )
```

let **and** let* **example**

```
(  let (  (a 5)  (b 6)  )
        (  + a b  )    )          ;; ⇒ 11
```

```
(let ( (a 5) (b (+ a 6)) )
     (  + a b  )        )
```

let **and** let* **example**

```
( let ( (a 5) (b 6) )
      ( + a b ) )          ;; ⇒ 11
```

```
(let ( (a 5) (b (+ a 6)) )
     ( + a b ) )           ;; ⇒ ERROR: a: undefined
```

let **and** let* **example**

```
(  let (  (a 5)  (b 6)  )
        (  + a b  )    )          ;; ⇒ 11
```

```
(let ( (a 5) (b (+ a 6)) )
     (  + a b  )        )          ;; ⇒ ERROR: a: undefined
```

```
(let* (  (a 5) (b (+ a 6))  )
        (  + a b  )       )
```

let **and** let* **example**

```
(  let (  (a 5)  (b 6)  )
        (  + a b  )    )          ;; ⇒ 11
```

```
(let ( (a 5) (b (+ a 6)) )
     (  + a b  )        )          ;; ⇒ ERROR: a: undefined
```

```
(let* (  (a 5) (b (+ a 6))  )
        (  + a b  )      )          ;; ⇒ 16
```

letrec **examples**

Typically used for local definitions for **recursive** functions

```
(letrec (  (a 5)
           (b ( lambda() (+ a 6) ) )  )
           (  + a (b)  )      )          ;; ⇒ 16
```

letrec **examples**

Typically used for local definitions for **recursive** functions

```
(letrec (  (a 5)
           (b ( lambda () (+ a 6) ) )  )
           (  + a (b)  )      )          ;; ⇒ 16

(letrec (  (b (lambda () (+ a 6) ) )
           (a 5)                        )
           (  + a (b) )       )          ;; ⇒ 16
```

letrec **examples**

Typically used for local definitions for **recursive** functions

```
(letrec ( (even? (lambda (x)
                   ( or (= x 0) (odd? (- x 1))) ) ) )
          (odd?  (lambda (x)
                   ( and (not (= x 0)) (even? (- x 1))) )))  )

(  list (even? 3) (even? 20) (odd? 21) ) )      ;;  ⇒ (#f #t #t)
```

letrec **examples**

Typically used for local definitions for **recursive** functions

```
(letrec (  (even? (lambda (x)
                    ( or (= x 0) (odd? (- x 1))) ) ) )
           (odd?  (lambda (x)
                    ( and (not (= x 0)) (even? (- x 1))) )) )

   (  list (even? 3) (even? 20) (odd? 21) ) )      ;;  ⇒ (#f #t #t)
```

Next Lecture

Things to do:

- Read Scott, Chapter 11.1 - 11.3 (4th Edition) or Chapter 10.1 - 10.3 (3rd Edition)