CS:314 Fall 2024

Section **04**

Recitation **9**

chris.tu@rutgers.edu

Office hours: 2-3pm @ Thursday CoRE 335

Topics Covered

- **Scheme**
  - Expressions
  - Lists
  - Types
  - Functions

Scheme Overview

Includes:

1. Set of **function definitions** and/or
2. Set of **function applications**

```
>   (define foo
       (lambda (x y)
           (cond                ((null? x) y)
                                ((null? y) x)
                )               (else (cons (car x)(foo (cdr x) y) ))
           )
       )
```

```
>   (foo '(a b c ) '(1 2))
```

Scheme Expressions

Written in prefix, parenthesized form:

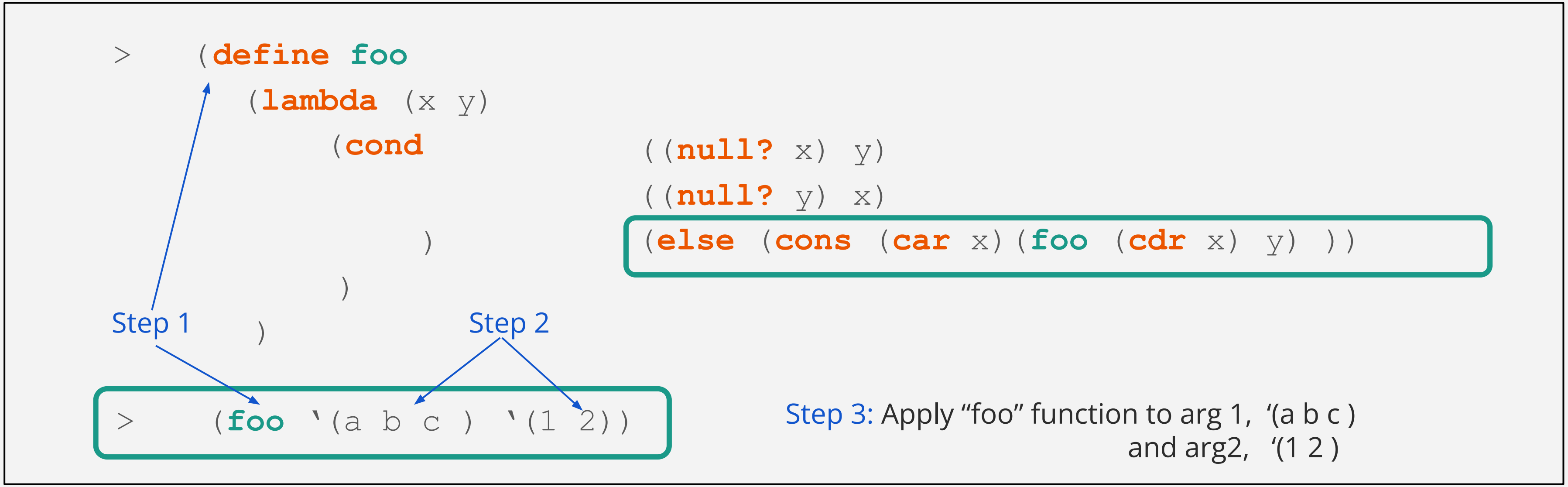
(f arg_1 arg_2 … arg□)

(+ 4 5)

(* (- 5 3) 2)

Function application:

1. Evaluate function f to a function value
2. Evaluate each arg_1 in order to obtain value
3. Apply function value to parameter values



```
>    (define foo
       (lambda (x y)
          (cond                        ((null? x) y)
                                       ((null? y) x)
                                       (else (cons (car x)(foo (cdr x) y) ))
              )
            )
         )
>    (foo '(a b c ) '(1 2))
```

Step 3: Apply "foo" function to arg 1, '(a b c)
and arg2, '(1 2)

Scheme Lists
& Fundamental Functions

Lists can be:

- Arbitrary length
  - Implemented w/ Linked List
- Heterogenous
  - Different Data Types

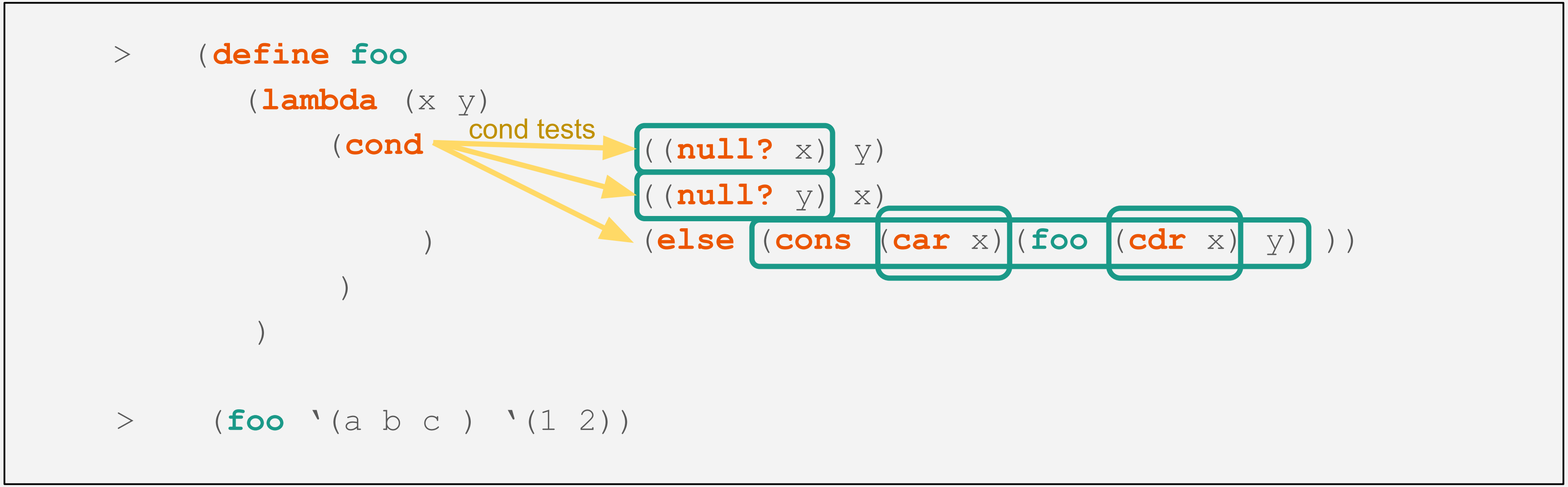
`cond`: control structure used for **cond**itional branching, kind of like if and else if statements, list of tests, and a default expr.

`null?` : is list empty?

`car` : return first element of list

`cdr` : return rest of list after first element

`(cons element list)` : constructs list by adding `element` to front of `list`



```
>    (define foo
        (lambda (x y)
            (cond              ((null? x)  y)
                               ((null? y)  x)
                  )            (else  (cons (car x)(foo (cdr x)  y)))
            )
        )

>    (foo ‘(a b c ) ‘(1 2))
```

Quotes

really think about this

‘ produces constants. Treat contents after ‘ as a list of literals

‘( ) is an empty list

```
(cons ‘a (cons ‘b ‘(c d)) )→ ‘(a b c d)

(cons ‘a ‘(cons ‘b ‘(c d)) )→ ‘(a cons ‘b ‘(c d))
```

within the parenthesis, becomes a quoted list, delimited by spaces

```
>    (define foo
       (lambda (x y)
          (cond                ((null? x) y)
                               ((null? y) x)
              )                (else (cons (car x)(foo (cdr x) y) ))
          )
       )

>    (foo ‘(a b c ) ‘(1 2))
```

List Manipulation Practice!

```
(cons `((a)) `(c d))                      =
(car `(a `(b c) d))                       =
(cdr `((a) b (cons c)))                   =
(car `((a) b (cons c)))                   =
(cons `e (cdr `(a b (c)))                 =
(cdr (cons (car `((a) b) `(c d))          =
```

List Manipulation Practice!

literal nested list

```
(cons ‘((a)) ‘(c d))                    = ‘( ((a)) c d )

(car ‘(a ‘(b c) d))                     = ‘a

(cdr ‘((a) b (cons c)))                 = ‘(b (cons c))

(car ‘((a) b (cons c)))                 = ‘(a)

(cons ‘e (cdr ‘(a b (c))))              = ‘(e b (c))

(cdr (cons (car ‘((a) b) ‘(c d))        = ‘(c d)
```

List Manip - Step By Step

```
(cdr (cons (car `((a) b) `(c d))))
```

First, evaluate the inner expression car `((a) b)
'((a) b) is a literal list with elements ‘(a) and ‘b. car selects ‘(a).

```
(cdr (cons (`(a) `(c d))))
```

Second, evaluate inner expression cons (`(a) `(c d))
‘(a) is the element parameter. ‘(c d) is the literal list parameter.
cons creates a new list prepending ‘(a) to the front of ‘(c d)

```
(cdr (`((a)c d)))
```

cdr returns the rest of the list after the first element.
The first element is ‘(a). The rest of the list has the elements c and d

Result: ‘(c d)

Types

SCHEME is

Dynamically Typed : type checks at runtime

Strongly Typed : typing is strictly enforced

Operations only performed on compatible types

- Example:
    - (+ 5 "10") ; Error: incompatible types (number and string)

Explicit type casting

```
>    (define foo
       (lambda (x y)
           (cond                         ((null? x) y)
                                         ((null? y) x)
               )                         (else (cons (car x)(foo (cdr x) y) ))
          )
       )

>    (foo ‘(a b c ) ‘(1 2))
```

x and y
must be lists

Functions

Syntax:

```
(define <func-name>
            (lambda (<func-params>) <expression>) )
```

```
>   (define foo
       (lambda (x y)
           (cond           ((null? x) y)
                           ((null? y) x)
                )          (else (cons (car x)(foo (cdr x) y) ))
            )
       )

>    (foo ‘(a b c ) ‘(1 2))
```

More Fundamental Functions and Conditions Syntax

`pair?` : true for non-empty lists, else false

`not` : boolean negation

```
(if <condition> <result1> <result2>)

(cond

    (<conditional1> <result1>)

    (<conditional2> <result2>)

    ...

    (<conditionN> <resultN>)

    (else <else_result>) )
```

Functions

What does **foo** do?

```
     >   (define foo
            (lambda (x y)
                (cond          ((null? x) y)
                               ((null? y) x)
                     )         (else (cons (car x)(foo (cdr x) y) ))
                )
            )

     >    (foo `(a b c ) `(1 2))
```

Functions

What does **foo** do?

Recursively merges two lists by appending x in the front of y

```
>    (define foo
        (lambda (x y)
            (cond           ((null? x) y)
                            ((null? y) x)
                )           (else (cons (car x)(foo (cdr x) y) ))
            )
        )

>    (foo `(a b c ) `(1 2))
```

Higher Order Functions

`(map <function> <list>):`

Builds a new list by applying <function> to each element of `<list>`

```
(map (lambda (x) (+ 2 x)) '(2 -1 5))

     →  ?
```

`(reduce <operation> <list> <i>):`

"Folds up" a list starting with `<i>` using `<operation>`

```
(reduce * '(2 3 4) 1)

     →  ?
```

Higher Order Functions

```
(map <function> <list>):
```

Builds a new list by applying <function> to each element of `<list>`

```
(map (lambda (x) (+ 2 x)) '(2 -1 5))

    →   '(4 1 7)
```

```
(reduce <operation> <list> <i>):
```

“Folds up” a list starting with `<i>` using `<operation>`

```
(reduce * '(2 3 4) 1)

    →  ?
```

Higher Order Functions

```
(map <function> <list>):
```

Builds a new list by applying <function> to each element of `<list>`

```
(map (lambda (x) (+ 2 x)) ‘(2 -1 5))

    →   ‘(4 1 7)
```

```
(reduce <operation> <list> <i>):
```

“Folds up” a list starting with initial value `<i>` using `<operation>`

```
(reduce * ‘(2 3 4) 1)

    →   24
```

Higher Order Functions

1 * 2 * 3 *4 = 24

```
(reduce <operation> <list> <i>):
```

“Rolls up” a list starting with initial value `<i>` using `<operation>`

```
(reduce * ‘(2 3 4) 1)

    →   24
```

more examples:

(reduce + '(1 2 3 4) 0)

=> **10**

(reduce + '(1 2 3 4) 1)

=> **11**

Scheme Function Practice

Define a function **`final`** that returns the last element of a list, and ‘( ) if the list is empty

HINT: RECURSION!!!

Scheme Function Practice

Define a function `final` that returns the last element of a list, and ‘( ) if the list is empty

```
(define final
    (lambda (lst)
        (cond ((null? lst) ‘())
              ((null? (cdr lst)) (car lst))
              (else (final (cdr lst)))
         )
    )
)
```