CS 357 - Lab 002

Session 5
let-> lambda (revisited)
reduce/fold(r)

Kage Weiss

CS 357 - Lab 002

TA: Kage Weiss
Office Hours: email, or by appointment.
Contact: mmweiss@unm.edu
Website: **http://cs.unm.edu/~kageweiss/TA/cs357.html -- SLIDES POSTED**

- Today we are working with the reduce pattern
- NOTE: Homework 2 are up, exam grades in progress
- I know who cheated on the homework, and that is your loss, this does mean that we do have to watch your exam answers more closely though

let, let*, letrec

let

- Allows for scoped definitions
- Does not allow for individual definitions to reference each other
- Does not allow for individual definitions to reference themselves

```
(define fn
    (let
        ( ;; definitions
            (id1 def1)
            (id2 def2)
            …
        )
        ( ;; usage
            (lambda (x y) (id2 (id1 x …)))
        )
    )
)
```

let, let*, letrec

let*

- Allows for scoped definitions
- Allows for individual definitions to reference each other
- Does not allow for individual definitions to reference themselves

```
(define fn
    (let*
        ( ;; definitions
            (id1 def1)
            (id2 (… def2 … id1 …))
            …
        )
        ( ;; usage
            (lambda (x y) (id2 (id1 x …)))
        )
    )
)
```

let, let*, letrec

letrec

- Allows for scoped definitions
- Allows for individual definitions to reference each other
- Allows for individual definitions to reference themselves

```
(define fn
    (letrec
        ( ;; definitions
            (id1 (… id1 …))
            (id2 (… def2 … def1 …))
            …
        )
        ( ;; usage
            (lambda (x y) (id2 (id1 x …)))
        )
    )
)
```

let, let*, letrec

let

- Allows for scoped definitions
- Does not allow for individual definitions to reference each other
- Does not allow for individual definitions to reference themselves

```
(let
    (
        ;pairs of names to defintions
        (id1 defintion1)
        (id2 defintion2)
    )
    (   ; begin scope of id1, id2
        (map (id1 id2) '())
    )   ; end scope of id1, id2
)
```

let, let*, letrec

```
(let
    (
        ;pairs of names to defintions
        (id1 defintion1)
        (id2 defintion2)
    )
    ( ; begin scope of id1, id2
        (map (id1 id2) '())
    ) ; end scope of id1, id2
)
```

```
((lambda
    (id1 id2) ;names
    ( ; begin scope of id1, id2
        (map (id1 id2) '())
    ) ; end scope of id1, id2
) defintion1 defintion2) ; application of defintions to names
```

let, let*, letrec

```
(let*
    (
        ;pairs of names to defintions
        (id1 defintion1)
        (id2 defintion2)
        (id3 (id1 id2))
    )
    (   ; begin scope of id1, id2, id3
        (map id3 '())
    )   ; end scope of id1, id2, id3
)
```

```
;; let* as let(s)
(let
    (
        ;pairs of names to defintions
        (id1 defintion1)
        (id2 defintion2)
    )
    (   ; begin scope of id1, id2
        (let
            (
                ;pairs of names to defintions
                (id3 (id1 id2))
            )
            (   ; begin scope of id3
                (map id3 '())
            )   ; end scope of id3
        )
    )   ; end scope of id1, id2
)
```

let, let*, letrec

```
(let*
    (
        ;pairs of names to defintions
        (id1 defintion1)
        (id2 defintion2)
        (id3 (id1 id2))
    )
    (   ; begin scope of id1, id2, id3
        (map id3 '())
    )   ; end scope of id1, id2, id3
)
```

```
;; let* as lambda(s)
((lambda
    (id1 id2) ;names
    (   ; begin scope of id1, id2
        ((lambda
            (id3) ;names
            (   ; begin scope of id3
                (map id3 '())
            )   ; end scope of id3
        ) (id1 id2)) ; application of defintions to names
    )   ; end scope of id1, id2
) defintion1 defintion2) ; application of defintions to names
```

let, let*, letrec

```
;; let* as let(s)
(let
    (
        ;pairs of names to defintions
        (id1 defintion1)
        (id2 defintion2)
    )
    (   ; begin scope of id1, id2
        (let
            (
                ;pairs of names to defintions
                (id3 (id1 id2))
            )
            (   ; begin scope of id3
                (map id3 '())
            )   ; end scope of id3
        )
    )   ; end scope of id1, id2
)
```

```
;; let* as lambda(s)
((lambda
    (id1 id2) ;names
    (   ; begin scope of id1, id2
        ((lambda
            (id3) ;names
            (   ; begin scope of id3
                (map id3 '())
            )   ; end scope of id3
        ) (id1 id2)) ; application of defintions to names
    )   ; end scope of id1, id2
) defintion1 defintion2) ; application of defintions to names
```

fold(r)

fold

- Allows for abstract definitions over lists
- Parameters:
  - combiner function
  - seed value
  - list to be applied to

```
(define test '(1 2 3 4 5))
(define list-reduce
    (lambda args
        (foldr ;;???
    )
)
(equal? (apply list-reduce test) test)
```

CS 357 - Lab 002

Go forth,
write your software.

Remember, these slides are available:

cs.unm.edu/~kageweiss/TA/cs357.html

CS 357 - Lab 002

Basic info has been put in the provided file for you.
Running (C-x h, C-c C-r) should result in:
> #t
> > #t
> > #t
> > #t

Note: you do not have to define map to properly handle varargs lists for this, or to handle functions that take varargs

e.g (map + test test test) => '(3 6 9 12 15)
e.g (map-reduce + test test test) => ; map-reduce: arity mismatch; …
BUT: (map +-reduce test test test) => '(3 6 9 12 15)