Recitation - 05
Scheme

Jahnavi - jp5867@nyu.edu

Prefix Vs Infix Notation:

Prefix: The function/operator comes before the arguments

Ex: f(x,y)

Infix: The function/operator comes in between the arguments

Ex:(1+(2+3))

In general, most math operators are infix while functions are prefix

In Scheme, every operator is used a prefix

(+1 (+2 3))

Also note that you do not put parentheses between function and argument

- f(x,y) → f x y

Data types:

- Numbers
  - Integer •42
  - Floating point • 3.1415
- Strings
  - “HelloWorld”
- Symbols
  - Identifiers for variables→Non self evaluating
  - ‘xyz,‘E
- Characters
  - #\a,#\b,...
- Boolean
  - #t,#f
- Pairs and lists
  - ‘(1,2,3,4),(list 1,2,3,4),(1.(2.(3.(4.()))))

Comments:

; followed by anything is comment

```
#|

Comment multiple lines

|#
```

Lists

- Single most important built in data type in Scheme is the list.
- Lists are immutable, unbounded, dynamically typed collections of data.
- Important functions that operate on lists: length -- length of a list

equal? -- test if two lists are equal

car -- first element of a list

cdr -- rest of a list

cons -- make a new list cell

list -- make a list

append – returns the concatenation of two lists. null? – returns #t if the argument is a null list

Lists

car: Returns the head of the list without changing the list

(car'(26 1 4 5)) - returns 2

cdr: Returns everything except the head of the list

(cdr'(2 6 1 4 5)) - returns (6 1 4 5)

cons: Creates a new list, with the first argument at the head.

(cons 1 '(2 6 1 4 5)) - returns (1 2 3 1 4 5)

Predicates:

Null? -- is the list empty?

equal?

Conditionals:

Control constructs

- (if condition expr1 expr2)
- (cond

```
        (pred1 expr1)
        (pred2 expr2)
        . . .
        (else exprn))
```

Variables:

- Scheme has both local and global variables.
- No type declarations for variables.

Define :

- Can be used to define functions or variables.
  - (define <name> <expression>)
  - (define (<fn-name> <param1> ... <paramN>) ...)
- Body is not evaluated. Just a binding is created. • Define is primarily used to bind global variables.

Local Variables:

In a let expression, the initial values are computed before any of the variables become bound;

In a let* expression, the bindings and evaluations are performed sequentially;

```
(let ((x 2) (y 3))
   (let ((x 7)
         (z (+ x y)))
     (* z x)))
```

35

```
(let* ((x 2) (y 3))
   (let ((x 7)
         (z (+ x y)))
     (* z x)))
```

70

While in a letrec expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

Letrec

letrec works for multiple mutually recursive local procedures, too. You can define several local procedures that can call each other, like this:

```
(define (my-proc)
  (letrec ((local-proc-1 (lambda ()
                           ...
                           (local-proc-2)
                           ...))
           (local-proc-2 (lambda ()
                           ...
                           (local-proc-1)
                           ...)))
     (local-proc-1))) ;
```

Letrec

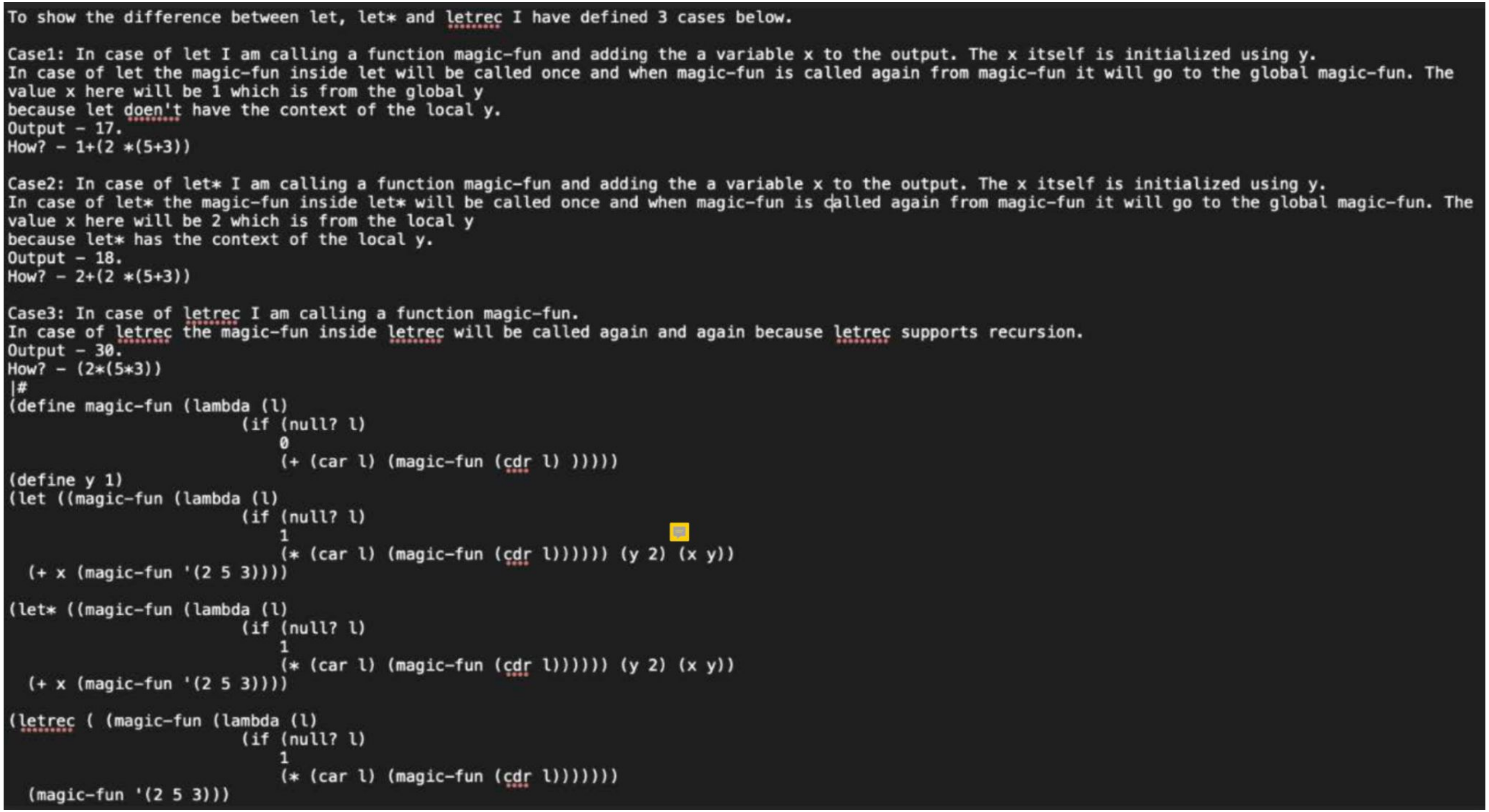
You can also define plain variables while you're at it, in the same letrec, but letrec is mostly interesting for defining local procedures

When the initial value of a letrec variable is not a procedure, you must be careful that the expression does not depend on the values of any of the other letrec variables. Like let, the order of initialization of the variables is undefined.

For example, the following is illegal:

```
(letrec ((x 2)
         (y (+ x x)))
  ...)
```

In this case, the attempt to compute (+ x x) may fail, because the value of x may not have been computed yet. For this example, let* would do the job--the second initialization expression needs to see the result of the first

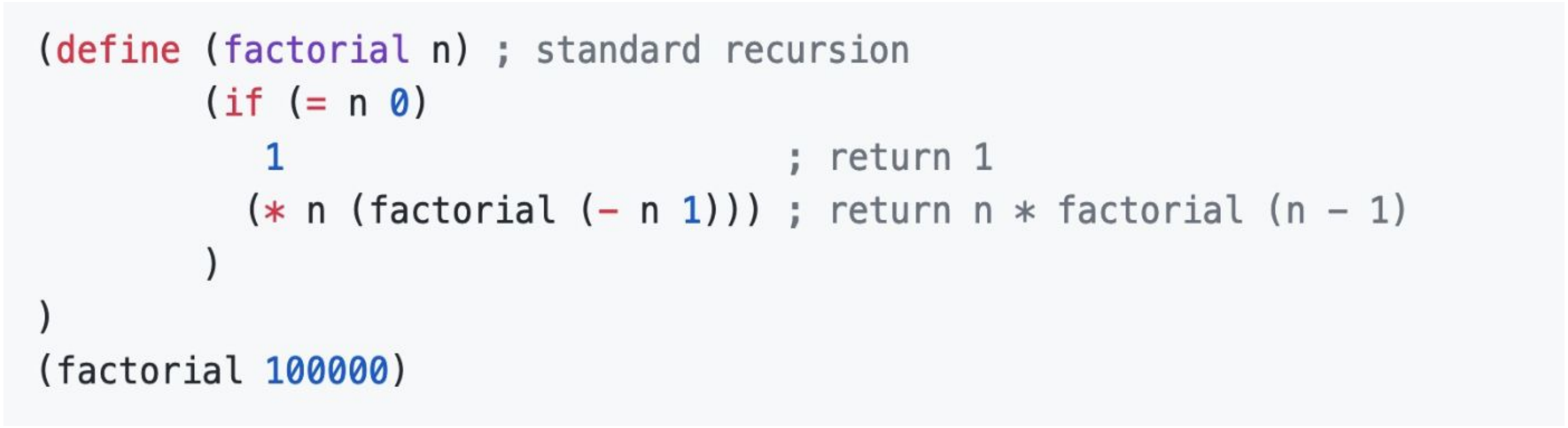


```
To show the difference between let, let* and letrec I have defined 3 cases below.

Case1: In case of let I am calling a function magic-fun and adding the a variable x to the output. The x itself is initialized using y.
In case of let the magic-fun inside let will be called once and when magic-fun is called again from magic-fun it will go to the global magic-fun. The
value x here will be 1 which is from the global y
because let doen't have the context of the local y.
Output - 17.
How? - 1+(2 *(5+3))

Case2: In case of let* I am calling a function magic-fun and adding the a variable x to the output. The x itself is initialized using y.
In case of let* the magic-fun inside let* will be called once and when magic-fun is called again from magic-fun it will go to the global magic-fun. The
value x here will be 2 which is from the local y
because let* has the context of the local y.
Output - 18.
How? - 2+(2 *(5+3))

Case3: In case of letrec I am calling a function magic-fun.
In case of letrec the magic-fun inside letrec will be called again and again because letrec supports recursion.
Output - 30.
How? - (2*(5*3))
|#
(define magic-fun (lambda (l)
                        (if (null? l)
                            0
                            (+ (car l) (magic-fun (cdr l) )))))
(define y 1)
(let ((magic-fun (lambda (l)
                        (if (null? l)
                            1
                            (* (car l) (magic-fun (cdr l)))))) (y 2) (x y))
  (+ x (magic-fun '(2 5 3))))

(let* ((magic-fun (lambda (l)
                        (if (null? l)
                            1
                            (* (car l) (magic-fun (cdr l)))))) (y 2) (x y))
  (+ x (magic-fun '(2 5 3))))

(letrec ( (magic-fun (lambda (l)
                        (if (null? l)
                            1
                            (* (car l) (magic-fun (cdr l)))))))
  (magic-fun '(2 5 3)))
```

Tail recursion:

Tail Recursion

- Def. it is a recursion in which no additional computation ever follows a recursive call.
- The recursive function returns what the next recursive call returns. No more computations after the recursive call.



```
(define (factorial n) ; standard recursion
        (if (= n 0)
           1                        ; return 1
          (* n (factorial (- n 1))) ; return n * factorial (n - 1)
        )
)
(factorial 100000)
```

Tail recursion:

```scheme
(define (fac x) ; tail recursion
  (letrec
    ( ; rec param
     (fac-tr (lambda (x acc)
       (if (zero? x) acc                ; return acc
         (fac-tr (- x 1) (* x acc)))))  ; return fac-tr(x - 1, x * acc)
    )
    (fac-tr x 1) ; rec body
  )
)
(fac 100000) ; This one is faster than standard recursion
```

Tail recursion:

- The most benefit for tail recursion is that the compiler can reuse the current activation record at the time of the recursive call, eliminating the need to allocate a new one, i.e. constant stack space.
- The Scheme compiler will detect the tail recursion fac and performs tail-call elimination to ensure that fac will run in constant stack space.
- Effectively, tail call elimination yields an implementation that is equivalent to one that uses a loop:

Higher order functions:

```
(map function list)            ;; general form

(map null? '(3 () () 5))       => (() T T ())

(map round '(3 3.3 4.6 5))       => (3 3 5 5)

(map cdr '((1 2) (3 4) (5 6)))  => ((2) (4) (6))

(filter (lambda (n) (> n 10)) '(5 10 15 20))  => (15 20)
```

Examples:

Length of list:

```
(define length
  (lambda (ls)
    (if (null? ls)
        0
        (+ (length (cdr ls)) 1))))
```

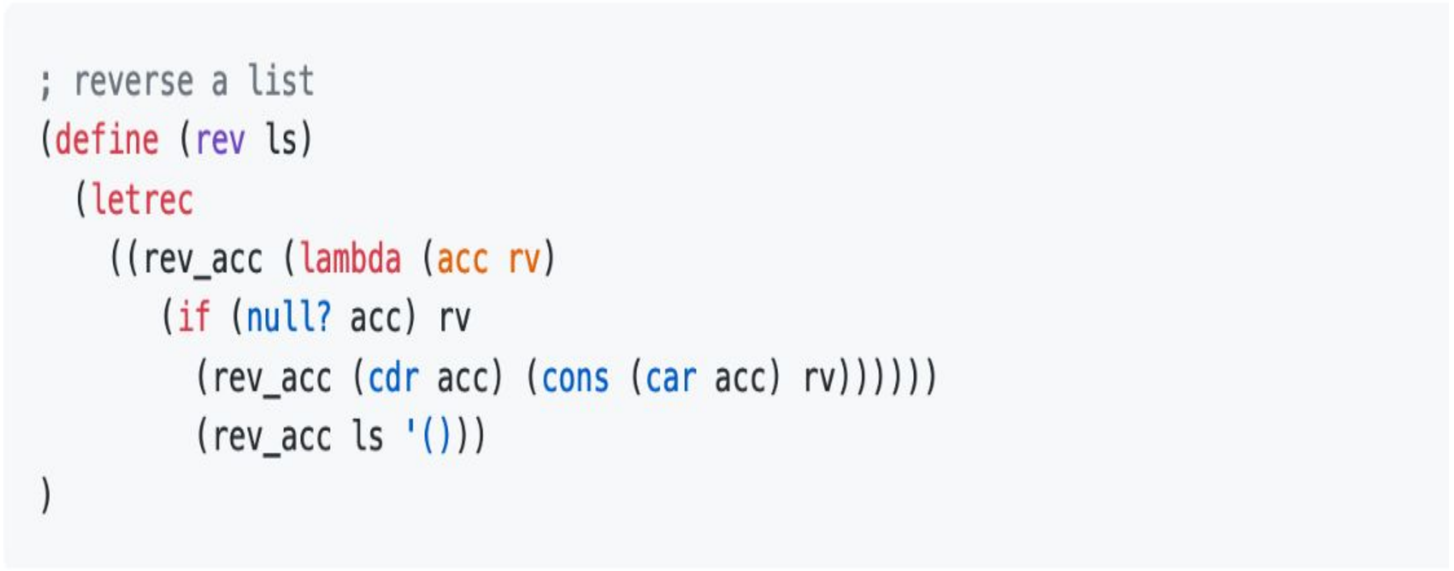
Examples:

Remove element from list:

```
(define remv
  (lambda (x ls)
    (cond
      ((null? ls) '())
      ((eqv? (car ls) x) (remv x (cdr ls)))
      (else (cons (car ls) (remv x (cdr ls)))))))
```

Examples:

Reverse a list:



```scheme
; reverse a list
(define (rev ls)
  (letrec
    ((rev_acc (lambda (acc rv)
       (if (null? acc) rv
         (rev_acc (cdr acc) (cons (car acc) rv))))))
         (rev_acc ls '()))
)
```

Thank you