literals

```
(define a 1)
(define b a)
(print b)
```

literals

```
(define a 1)
(define b a)
(print b)
1
```

literals

```
(define a 1)
(define b a)
(print b)
1
```

```
(define a 1)
(define b 'a)
(print b)
a
```

literals

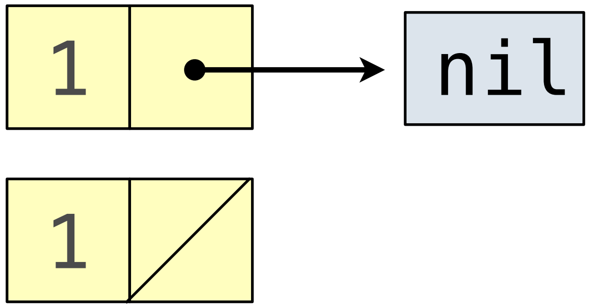
```
(define a 1)
(define b a)
(print b)
1
```

```
(define a 1)
(define b 'a)
(print b)
a
```

```
(define b (quote a))
(print b)
a
```

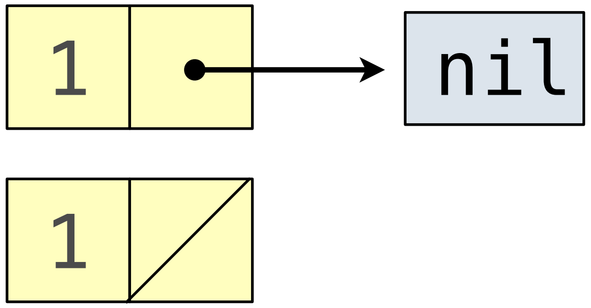
Scheme (Linked) Lists

```
(cons 1 '())
(1)
```

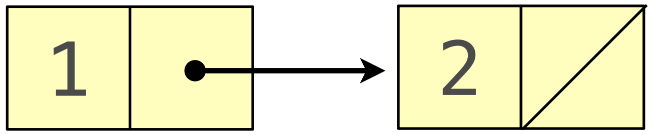


Scheme (Linked) Lists

```
(cons 1 '())
(1)
```

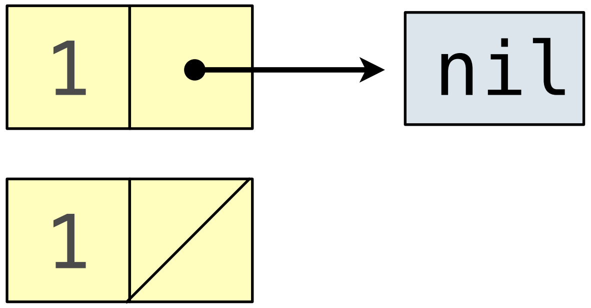


```
(cons 1 (cons 2 '()))
(1 2)
```

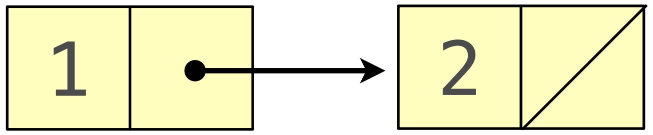


Scheme (Linked) Lists

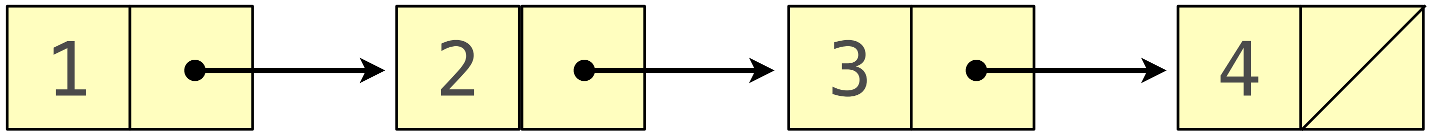
```
(cons 1 '())
(1)
```



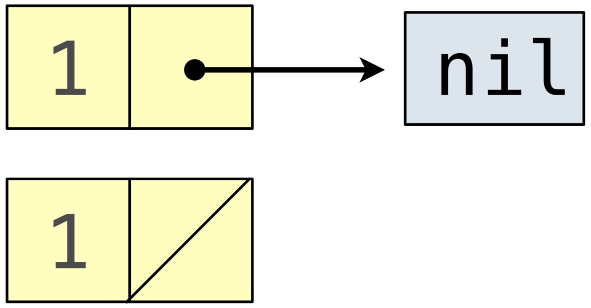
```
(cons 1 (cons 2 '()))
(1 2)
```



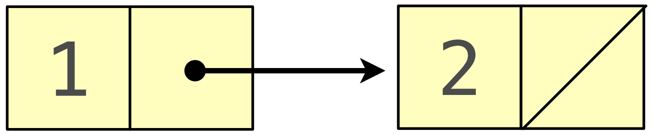
```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
(1 2 3 4)
```



Scheme (Linked) Lists

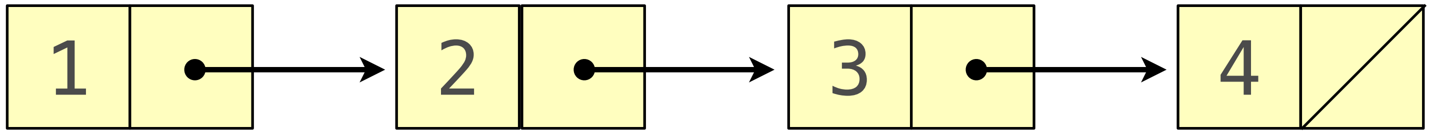


```
(cons 1 '())
(1)
```



```
(cons 1 (cons 2 '()))
(1 2)
```

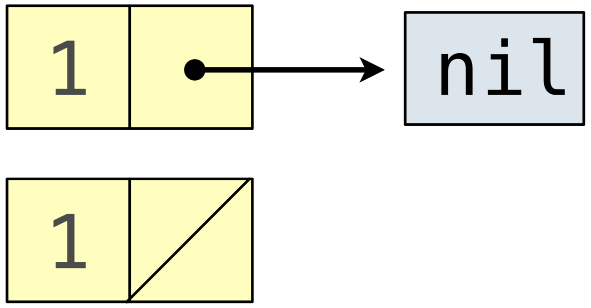
```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
(1 2 3 4)
```



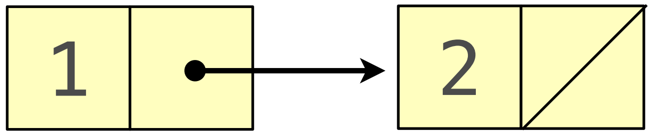
```
(list 1 2 3 4)
(1 2 3 4)
```

Scheme (Linked) Lists

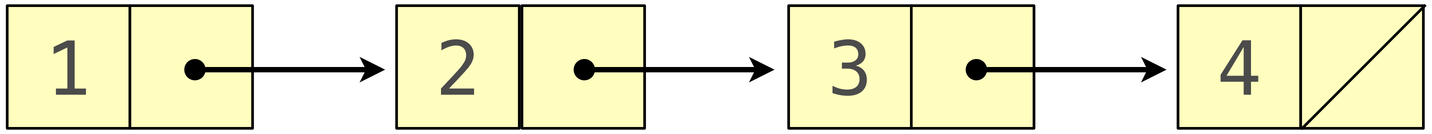
```
(cons 1 '())
(1)
```



```
(cons 1 (cons 2 '()))
(1 2)
```



```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
(1 2 3 4)
```

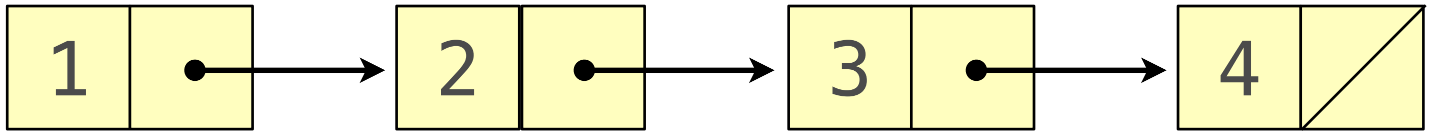


```
(list 1 2 3 4)
(1 2 3 4)
```

```
(cons 1 2)
(1 . 2) ; just FYI, we won't deal with pairs
```

Scheme (Linked) Lists

```
(define x (list 1 2 3 4))
```

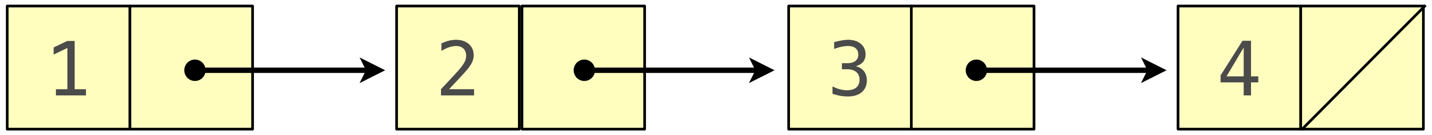


```
(car x)
1
```

```
(cdr x)
(2 3 4)
```

Scheme (Linked) Lists

```
(define x (list 1 2 3 4))
```



```
(car x)
1


(cdr x)
(2 3 4)


(car (cdr x) )
2


(cdr (cdr x))
(3 4)
```

Scheme (Linked) Lists

```
(define x (list 1 2 3))
(list? x)
#t
```

```
(null? x)
#f
```

Scheme equal? and eq?

```
(define x (list 1 2 3))
(define y x)

(equal? x '(1 2 3))
#t

(equal? x y)
#t



(eq? x '(1 2 3))
#f

(eq? x y)
#t
```

Scheme (Linked) Lists

```
(define x '(a b c))
(append x (list 'd))
(a b c d)
```

Scheme (Linked) Lists

```
(define x '(a b c))
(append x (list 'd))
(a b c d)
```

```
(define s (list 1 4 9 16 25))
(append s s)
(1 4 9 16 25 1 4 9 16 25)
```

Scheme (Linked) Lists

```
(define x '(a b c))
(append x (list 'd))
(a b c d)
```

```
(define s (list 1 4 9 16 25))
(append s s)
(1 4 9 16 25 1 4 9 16 25)

(cons s s)
((1 4 9 16 25) 1 4 9 16 25)
```

Scheme (Linked) Lists

```
(define x '(a b c))
(append x (list 'd))
(a b c d)


(define s (list 1 4 9 16 25))
(append s s)
(1 4 9 16 25 1 4 9 16 25)

(cons s s)
((1 4 9 16 25) 1 4 9 16 25)

(append (list 1 4 9) (list 1 4 9))
```

Scheme (Linked) Lists

```
(define x '(a b c))
(append x (list 'd))
(a b c d)


(define s (list 1 4 9 16 25))
(append s s)
(1 4 9 16 25 1 4 9 16 25)

(cons s s)
((1 4 9 16 25) 1 4 9 16 25)

(append (list 1 4 9) (list 1 4 9))
(1 4 9 1 4 9)

(append (list (list 1 4 9)) (list 1 4 9))
```

Scheme (Linked) Lists

```
(define x '(a b c))
(append x (list 'd))
(a b c d)
```

```
(define s (list 1 4 9 16 25))
(append s s)
(1 4 9 16 25 1 4 9 16 25)

(cons s s)
((1 4 9 16 25) 1 4 9 16 25)

(append (list 1 4 9) (list 1 4 9))
(1 4 9 1 4 9)

(append (list (list 1 4 9)) (list 1 4 9))
((1 4 9) 1 4 9)
```

Practice

```
(define a (list 1 2 (list 3 4 5) 6 7))
(1 2 (3 4 5) 6 7)

(car a)
```

Practice

```
(define a (list 1 2 (list 3 4 5) 6 7))
(1 2 (3 4 5) 6 7)

(car a)
1
```

Practice

```
(define a (list 1 2 (list 3 4 5) 6 7))
(1 2 (3 4 5) 6 7)

(car a)
1



(car (cdr (cdr a)))
```

Practice

```
(define a (list 1 2 (list 3 4 5) 6 7))
(1 2 (3 4 5) 6 7)

(car a)
1


(car (cdr (cdr a)))
(3 4 5)
```

Practice

```
(define a (list 1 2 (list 3 4 5) 6 7))
(1 2 (3 4 5) 6 7)

(car a)
1


(car (cdr (cdr a)))
(3 4 5)


(define b '((1) 2 (3)))
??? ; 2
```

Practice

```
(define a (list 1 2 (list 3 4 5) 6 7))
(1 2 (3 4 5) 6 7)

(car a)
1



(car (cdr (cdr a)))
(3 4 5)



(define b '((1) 2 (3)))
(car (cdr b)) ; 2
```

Practice

```
(define a (list 1 2 (list 3 4 5) 6 7))
(1 2 (3 4 5) 6 7)

(car a)
1


(car (cdr (cdr a)))
(3 4 5)


(define b '((1) 2 (3)))
(car (cdr b)) ; 2
???; 3
```

Practice

```
(define a (list 1 2 (list 3 4 5) 6 7))
(1 2 (3 4 5) 6 7)

(car a)
1



(car (cdr (cdr a)))
(3 4 5)



(define b '((1) 2 (3)))
(car (cdr b)) ; 2
(car (car (cdr (cdr b)))) ; 3
```

Scheme (Linked) Lists

```
(define (isEven num)
   (= (modulo num 2) 0))

(define x '(1 2 3 4 5 6))

(map isEven x)
(#f #t #f #t #f #t)
```

Scheme (Linked) Lists

```
(define (isEven num)
   (= (modulo num 2) 0))

(define x '(1 2 3 4 5 6))

(map isEven x)
(#f #t #f #t #f #t)


(filter isEven x)
(2 4 6)
```

Scheme (Linked) Lists

```
(define (isEven num)
   (= (modulo num 2) 0))

(define x '(1 2 3 4 5 6))

(map isEven x)
(#f #t #f #t #f #t)


(filter isEven x)
(2 4 6)


(apply + x)
21
```

Scheme Lists

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

Scheme Lists

```
(define (myFilter f L)
   (if (null? L)
       L
       (if (f (car L))
           (cons (car L) (myFilter f (cdr L)))
           (myFilter f (cdr L))
       )))

(myFilter isEven '(1 2 3 4 5 6))
(2 4 6)
```

Scheme Lists

```
(define (reverse L)
   (if (null? L)
       L
       (append (reverse (cdr L)) (list (car L)))
    ))

(reverse '(1 2 3 4 5 6))
(6 5 4 3 2 1)
```

Scheme Lists

```
; write a function that returns all nonempty subsets of ’s’
(define (subsets s)
    (if (null? s)
        nil
        (let ((rest (subsets (cdr s))))
           (append rest
                   (insert (car s) rest)
                   (list (list (car s)))
           )
         )
     )
)

(subsets '(2 3))
((3) (2 3) (2))

(subsets '(1 2 3))
((3) (2 3) (2)) “+” ((1 3) (1 2 3) (1 2)) “+” ((1))
```

Scheme Lists

```
; write a function that returns all nonempty subsets of ’s’
(define (subsets s)
    (if (null? s)
        nil
        (let ((rest (subsets (cdr s))))
           (append rest
                   (insert (car s) rest)
                   (list (list (car s)))
           )
         )
     )
)

(subsets '(2 3))
((3) (2 3) (2))

(subsets '(1 2 3))
((3) (2 3) (2)) “+” ((1 3) (1 2 3) (1 2)) “+” ((1))

(define (insert a rest) (map (lambda (t) (cons a t)) rest))
(insert 1 '((3) (2 3) (2)))
((1 3) (1 2 3) (1 2))
```

Scheme Lists

```
; write a function that returns all nonempty subsets of ’s’
(define (subsets s)
    (if (null? s)
        nil
        (let ((rest (subsets (cdr s))))
           (append rest
                   (insert (car s) rest)
                   (list (list (car s)))
           )
         )
     )
)
(define (insert a rest) (map (lambda (t) (cons a t)) rest))

(subsets '(3))
   (append (subsets '()) (insert 3 '()) (list (list 3)))
   (append '() '() '((3)))
   ((3))
```

Scheme Lists

```scheme
; write a function that returns all nonempty subsets of ’s’
(define (subsets s)
    (if (null? s)
        nil
        (let ((rest (subsets (cdr s))))
           (append rest
                   (insert (car s) rest)
                   (list (list (car s)))
           )
         )
     )
)
(define (insert a rest) (map (lambda (t) (cons a t)) rest))

(subsets '(2 3))
   (append (subsets '(3)) (insert 2 '((3))) (list (list 2)))
   (append '((3)) '((2 3)) '((3)))
   ((3) (2 3) (2))
```

Scheme Lists

```
; write a function that returns all nonempty subsets of ’s’
(define (subsets s)
    (if (null? s)
        nil
        (let ((rest (subsets (cdr s))))
           (append rest
                   (insert (car s) rest)
                   (list (list (car s)))
           )
         )
     )
)
(define (insert a rest) (map (lambda (t) (cons a t)) rest))
```

```
(subsets ‘(1 2 3))
   (append (subsets '((3) (2 3) (2))) (insert 1 '((3) (2 3) (2))) (list (list 1)))
   (append '((3) (2 3) (2)) '((1 3) (1 2 3) (1 2)) '((1)))
   ((3) (2 3) (2) (1 3) (1 2 3) (1 2) (1))
```

Scheme Lists

```
; write a function that returns all nonempty subsets of ’s’
(define (subsets s)
    (if (null? s)
        nil
        (let ((rest (subsets (cdr s))))
           (append rest
                   (insert (car s) rest)
                   (list (list (car s)))
           )
         )
     )
)
(define (insert a rest) (map (lambda (t) (cons a t)) rest))

(define (subsets s)
    (if (null? s)
        nil
        (let ((rest (subsets (cdr s))))
           (append rest
                   (map (lambda (t) (cons (car s) t)) rest)
                   (list (list (car s)))
           )
         )
     )
)
```