CSE221
Structure and Interpretation of Computer Programs

Introduction to Data Abstraction

Status So Far

	data	procedures
primitive	X	X
combinations		X
abstraction		X

Now we treat compound data

Why Compound Data?

A rational number can be thought of as a numerator and a denominator (i.e. 2 numbers) or as "a rational number" (i.e. 1 number that happens to consist of 2 numbers)

Abstraction and Expressivity

We will introduce
- composition of data
- abstraction of data
- genericity of code
- programs as data (i.e. symbolic data)

Overal goal: structure our programs so that they can operate on abstract data

Illustration: Rational Numbers

Suppose we're writing a mathematical system.

```
(make-rat <n> <d>)
(numer <r>)
(denom <r>)
```

Suppose we had these

Then this is possible!

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
```

Structuring Data in Scheme

```
(car (cdr (cons x y)) = x
```

```
(car (cons x y)) = x
(cdr (cons x y)) = y
```

```
> (define x (cons 1 2))
> (car x)
1
> (cdr x)
2
```

```
> (define x (cons 1 2))
> (define y (cons 3 4))
> (define z (cons x y))
> (car (car z))
1
> (car (cdr z))
3
```

Any structure can be made

Back to our Rational Numbers

4 alternative implementations

```
(define make-rat cons)
(define numer car)
(define denom cdr)
```

```
(define (make-rat n d)
  (cons n d))
(define (numer r)
  (car r))
(define (denom r)
  (cdr r))
```

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
(define (numer r)
  (car r))
(define (denom r)
  (cdr r))
```

```
(define (make-rat n d)
  (cons n d))
(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g)))
(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g)))
```

Interludium: Textual Output

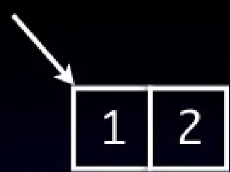
```
(display <expression>)
(newline)
```

```
(define (print-rat x)
  (display (numer x))
  (display "/")
  (display (denom x))
  (newline))
```

A body with multiple expressions

Box-and-pointer Diagrams

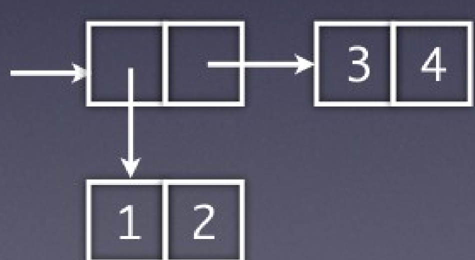
```
(cons 1 2)
```



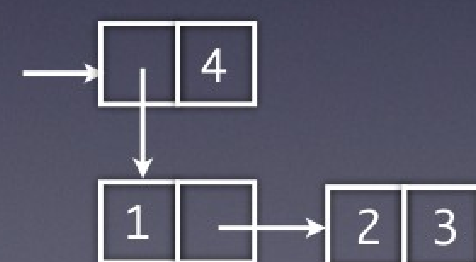
A pair's elements can be pairs again.

Closure property for pairs

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




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



What is meant by Data ?

Procedural representation of cons cells

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1 -- cons" m))))
  dispatch)

(define (car z)
  (z 0))
(define (cdr z)
  (z 1))
```

This curiosity will form the basis of object-oriented programming in Scheme (c.f. chapter 3)

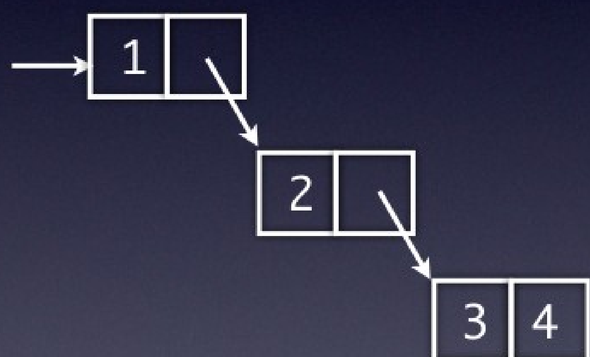
```scheme
(define (cons x y)
   (lambda (m) (m x y)))

(define (car z)
   (z (lambda (p q) p)))

(define (cdr z)
   (z (lambda (p q) q)))
```

CSE221
Structure and Interpretation of Computer Programs

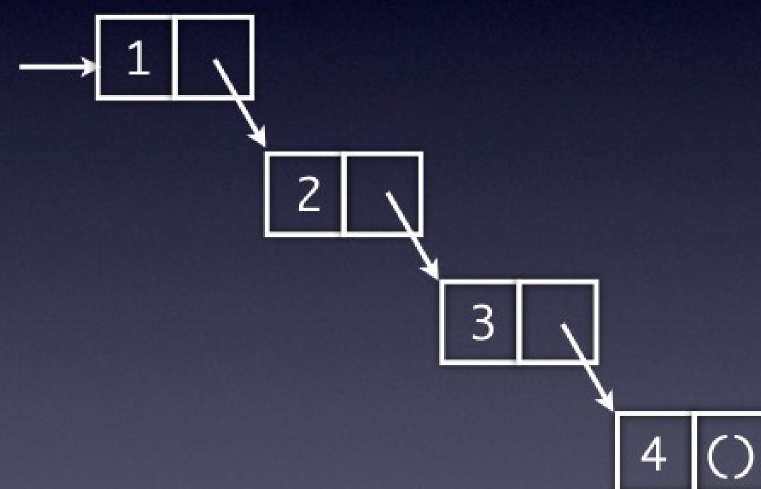
Hierarchical Data and Closure Property

lists

A list is
- either the empty list '()
- any pair whose cdr is a list



not a list!



a list

For your convenience

```
> (list 1 2 3 4)
(1 2 3 4)
> (cons 1 (cons 2 (cons 3 (cons 4 '()))))
(1 2 3 4)
> (caddr (list 1 2 3 4))
3
> (caar (cons (cons 1 2) (cons 3 4)))
1
> (null? '())
#t
> (null? (list))
#t
> (null? (list 1))
#f
> (null? (cons 1 2))
#f
```

A Very Common Pitfall

```
> (1 2 3 4 5)
⊕ procedure application: expected procedure, given: 1; arguments
were: 2 3 4 5
>
```

Remember the evaluation rule for combinations

List Operations: `list-ref`

```
(define squares (list 1 4 9 16 25))
squares
'(1 4 9 16 25)

(list-ref squares 3)
16
```

Say What is your Name?

Say “ What is your Name?”

List Operations: `list-ref`

```
(define squares (list 1 4 9 16 25))
squares
'(1 4 9 16 25)

(list-ref squares 3)
16
```

List Operations: `list-ref`

```
(define (list-ref items n)
  ??
 )
```

List Operations: `list-ref`

```
(list-ref '(1 4 9 16 25) n)

if n = 0  ??

Otherwise ??
```

List Operations: `list-ref`

`(list-ref` **`'(1 4 9 16 25)`** `n)`

`if n = 0 car of the list be returned`

`Otherwise ??`

List Operations: `list-ref`

```
(define (list-ref items n)
   (if (= n 0)
         (car items)
         ??
 )
```

List Operations: `list-ref`

`(list-ref` **`'(1 4 9 16 25)`** `n)`

`if n = 0 car of the list be returned`

`Otherwise ??`

List Operations: `list-ref`

`(list-ref '(1 4 9 16 25) n)`

`if n = 0 car of the list be returned`

`Otherwise take cdr of list and`
`decrement n by 1`

List Operations: `list-ref`

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
```

List Operations: `length`

```
(define odds (list 1 3 5 7))
odds
'(1 3 5 7)

(length odds)
4
```

List Operations: `length`

```
(define (length items)
  ??
 )
```

List Operations: `length`

```
(length '(1 3 5 7))
```

List Operations: `length`

```
(length '(1 3 5 7))
```

`length = 1 + length of cdr of the list`

List Operations: `length`

```
(define (length items)
  ??
  (+ 1 (length (cdr items)))
 )
```

List Operations: `length`

`(length '(1 3 5 7))`

length = 1 + length of cdr of the list

length of empty list is 0

List Operations: `length`

```
(length '(1 3 5 7))
```

`length = 1 + length of cdr of the list`

`length of empty list is 0`

`(how to check an empty list?)`

List Operations: `length`

`(length '(1 3 5 7))`

length = 1 + length of cdr of the list

length of empty list is 0

(how to check an empty list?)

Luckily scheme provides null?

List Operations: `length`

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

List Operations: `append`

```
squares
'(1 4 9 16 25)

odds
'(1 3 5 7)

(append squares odds)
```

List Operations: `append`

```
squares
'(1 4 9 16 25)

odds
'(1 3 5 7)

(append squares odds)
'(1 4 9 16 25 1 3 5 7)
```

List Operations: `append`

```
squares
'(1 4 9 16 25)

odds
'(1 3 5 7)

(append odds squares)
```

List Operations: `append`

```
squares
'(1 4 9 16 25)

odds
'(1 3 5 7)

(append odds squares)
'(1 3 5 7 1 4 9 16 25)
```

List Operations: `append`

```
(define (append list1 list2)
  ??
 )
```

Cons, Car, Cdr and List

```
(define one-through-four (list 1 2 3 4))

one-through-four
```

Cons, Car, Cdr and List

```
(define one-through-four (list 1 2 3 4))

one-through-four
```
`(1 2 3 4)`

Cons, Car, Cdr and List

```
(define one-through-four (list 1 2 3 4))
```

```
One-through-four
```

`(1 2 3 4)`

```
(car one-through-four)
```

Cons, Car, Cdr and List

```
(define one-through-four (list 1 2 3 4))
```

```
One-through-four
```

`(1 2 3 4)`

`(car one-through-four)` → **`1`**

Cons, Car, Cdr and List

```
(define one-through-four (list 1 2 3 4))
```

```
One-through-four
```

(1 2 3 4)

`(car one-through-four)` → **1**

```
(cdr one-through-four)
```

Cons, Car, Cdr and List

```
(define one-through-four (list 1 2 3 4))
```

One-through-four

(1 2 3 4)

`(car one-through-four)`→ **1**

`(cdr one-through-four)` → **(2 3 4)**

Cons, Car, Cdr and List

```
(define one-through-four (list 1 2 3 4))
```

```
One-through-four
```

(1 2 3 4)

`(car one-through-four)` → **1**

`(cdr one-through-four)` → **(2 3 4)**

```
(car (cdr one-through-four))
```

Cons, Car, Cdr and List

```
(define one-through-four (list 1 2 3 4))
```

One-through-four

(1 2 3 4)

(car one-through-four)→ **1**

(cdr one-through-four) → **(2 3 4)**

(car (cdr one-through-four)) → **2**

Cons, Car, Cdr and List

```
(define one-through-four (list 1 2 3 4))
```

```
One-through-four
```

`(1 2 3 4)`

`(car one-through-four)`→ **`1`**

`(cdr one-through-four)` → **`(2 3 4)`**

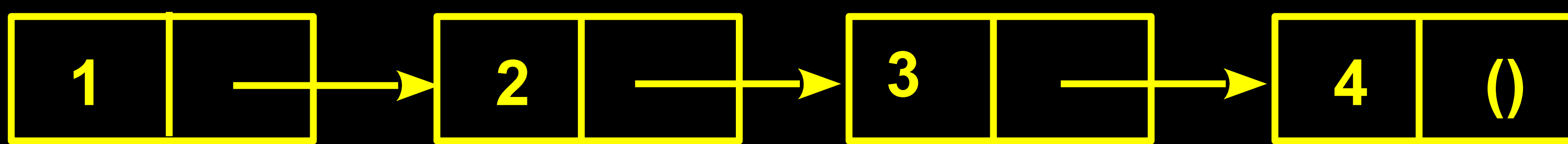
`(car (cdr one-through-four))` → **`2`**

```
(cons 10 one-through-four)
```

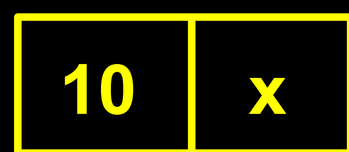
Cons, Car, Cdr and List

```
(define one-through-four (list 1 2 3 4))

One-through-four
(1 2 3 4)

(car one-through-four)→ 1

(cdr one-through-four) → (2 3 4)

(car (cdr one-through-four)) → 2

(cons 10 one-through-four) → (10 1 2 3 4)
```

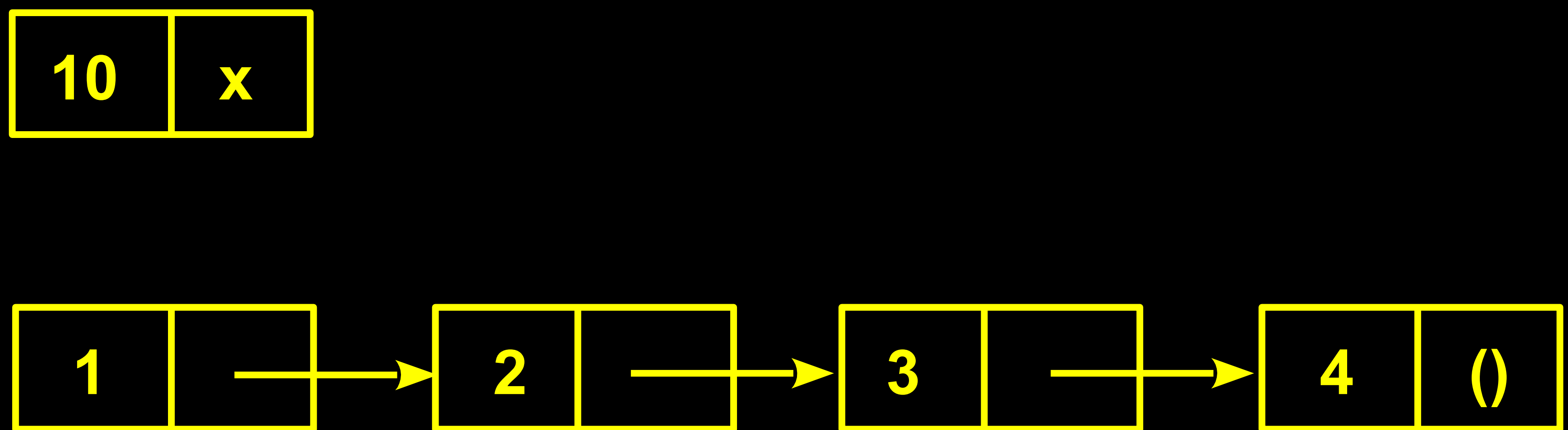
one-through-four



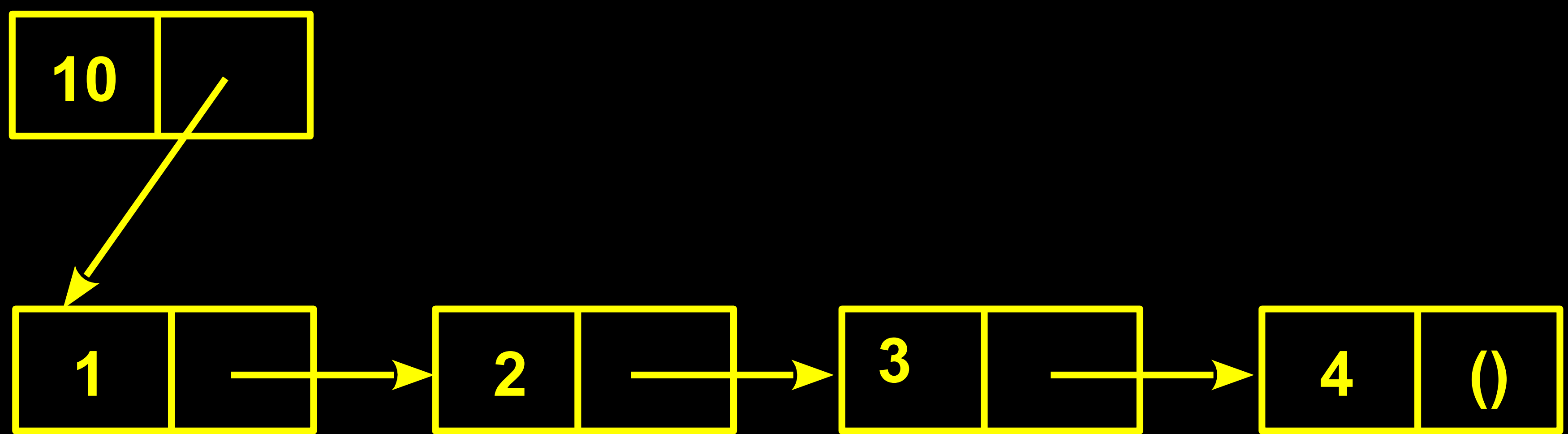
(cons 10 x)



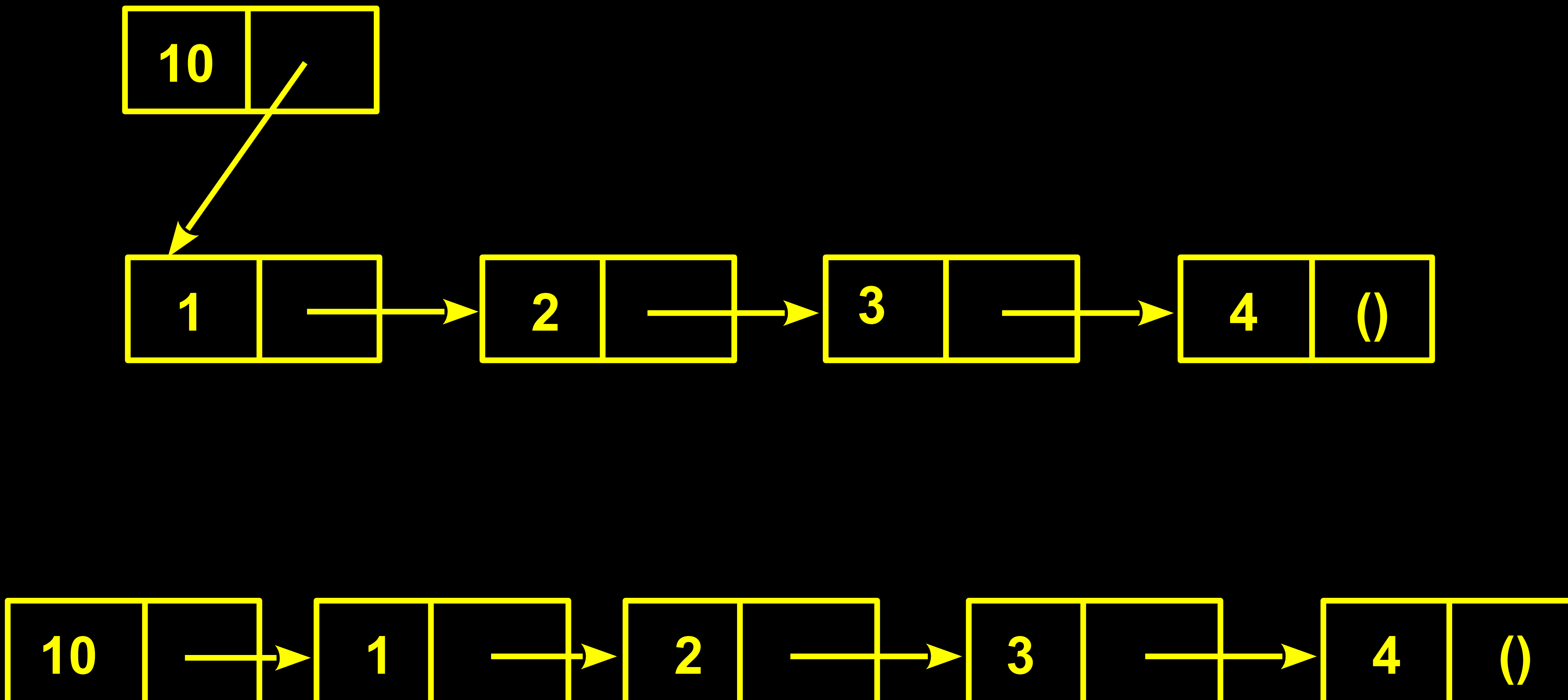
(cons 10 one-through-four)



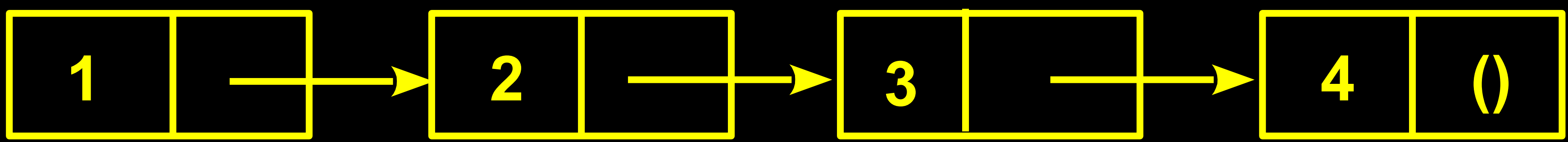
(cons 10 one-through-four)



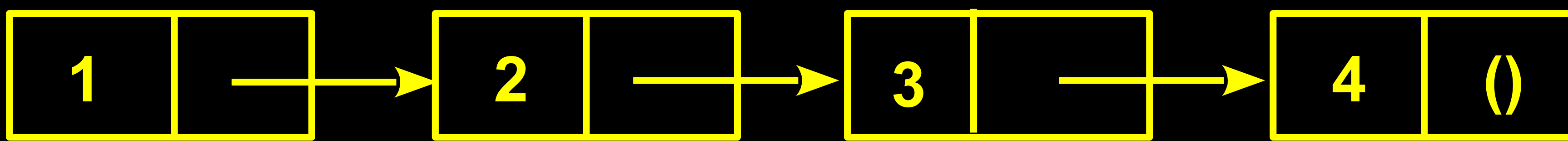
(cons 10 one-through-four)

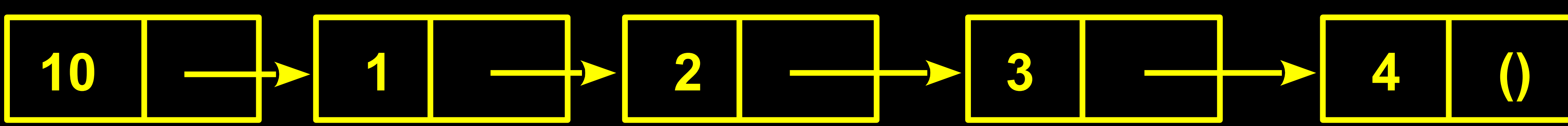


(1 2 3 4)

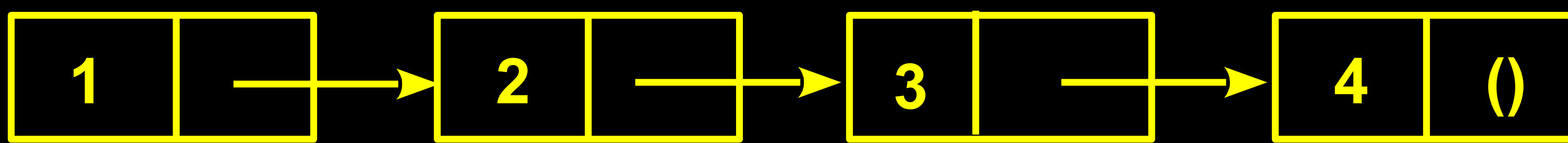


(1 2 3 4)

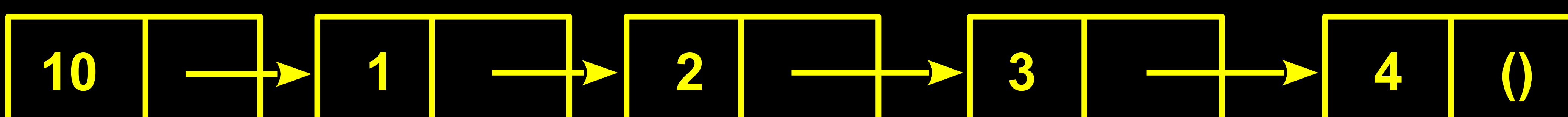




(1 2 3 4)



(10 1 2 3 4)



Cons, Car, Cdr and List

```
(define one-through-four (list 1 2 3 4))
```

One-through-four

(1 2 3 4)

(car one-through-four)→ **1**

(cdr one-through-four) → **(2 3 4)**

(car (cdr one-through-four)) → **2**

(cons 10 one-through-four) → **(10 1 2 3 4)**

List Operations: `append`

```
squares
'(1 4 9 16 25)

odds
'(1 3 5 7)

(append odds squares)
'(1 3 5 7 1 4 9 16 25)
```

List Operations: `append`

`squares`

`(1 4 9 16 25)`

`odds`

`(1 3 5 7)`

`(append odds squares)`

`(1 3 5 7 1 4 9 16 25)`

`(cons 7 squares)`

List Operations: `append`

`squares`

(1 4 9 16 25)

`odds`

(1 3 5 7)

`(append odds squares)`

(1 3 5 7 1 4 9 16 25)

`(cons 7 squares)` → **(7 1 4 9 16 25)**

List Operations: `append`

`squares`

`(1 4 9 16 25)`

`odds`

`(1 3 5 7)`

`(append odds squares)`

`(1 3 5 7 1 4 9 16 25)`

`(cons 7 squares)` → **`(7 1 4 9 16 25)`**

`(cons 5 (cons 7 squares)` → **`(5 7 1 4 9 16 25)`**

List Operations: `append`

`squares` `odds`

(1 4 9 16 25) **(1 3 5 7)**

`(append odds squares)`

(1 3 5 7 1 4 9 16 25)

`(cons 7 squares)` → **(7 1 4 9 16 25)**

`(cons 5 (cons 7 squares)` → **(5 7 1 4 9 16 25)**

`(cons 3 (cons 5 (cons 7 squares)` → **(3 5 7 1 4 9 16 25)**

List Operations: `append`

`squares` `odds`

`(1 4 9 16 25)` **`(1 3 5 7)`**

`(append odds squares)`

`(1 3 5 7 1 4 9 16 25)`

`(cons 7 squares)` → **`(7 1 4 9 16 25)`**

`(cons 5 (cons 7 squares)`→ **`(5 7 1 4 9 16 25)`**

`(cons 3 (cons 5 (cons 7 squares)` →

`(3 5 7 1 4 9 16 25)`

`(cons 1 (cons 3 (cons 5 (cons 7 squares)` →

`(1 3 5 7 1 4 9 16 25)`

List Operations: `append`

```
(define (append list1 list2)
  ??
)
```

List Operations: `append`

`squares`

(1 4 9 16 25)

`odds`

(1 3 5 7)

`(append odds squares)`

(1 3 5 7 1 4 9 16 25)

`(cons 7 squares)` → **(7 1 4 9 16 25)**

`(cons 5 (cons 7 squares)` → **(5 7 1 4 9 16 25)**

`(cons 3 (cons 5 (cons 7 squares)` →
(3 5 7 1 4 9 16 25)

`(cons 1 (cons 3 (cons 5 (cons 7 squares)` →
(1 3 5 7 1 4 9 16 25)

List Operations: `append`

```
(define (append list1 list2)
  ??
  (cons (car list1)
        (append (cdr list1) list2))
)
```

List Operations: `append`

`squares`

(1 4 9 16 25)

`odds`

(1 3 5 7)

`(append odds squares)`

(1 3 5 7 1 4 9 16 25)

`(cons 7 squares)` → **(7 1 4 9 16 25)**

`(cons 5 (cons 7 squares)` → **(5 7 1 4 9 16 25)**

`(cons 3 (cons 5 (cons 7 squares)` →
(3 5 7 1 4 9 16 25)

`(cons 1 (cons 3 (cons 5 (cons 7 squares)` →
(1 3 5 7 1 4 9 16 25)

List Operations: `append`

```
(define (append list1 list2)
   (if (null? List1)
      list2
      (cons (car list1)
            (append (cdr list1) list2))
   )
)
```

List Operations: `list-ref`

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
```

List Operations: `length`

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

List Operations: `scale-list`

```
(scale-list (list 1 2 3 4) 5)
```

`(5 10 15 20)`

List Operations: `scale-list`

```
(scale-list (list 1 2 3 4) 5)
```

`(5 10 15 20)`

```
(1 2 3 4) ≡
  (cons 1 (cons 2 (cons 3 (cons 4 nil))))
```

List Operations: `scale-list`

```
(scale-list (list 1 2 3 4) 5)
```

`(5 10 15 20)`

```
(1 2 3 4) ≡
   (cons 1 (cons 2 (cons 3 (cons 4 nil))))
```

```
(define (scale-list items factor)

)
```

List Operations: `scale-list`

```
(scale-list (list 1 2 3 4) 5)
```

`(5 10 15 20)`

```
(1 2 3 4) ≡
   (cons 1 (cons 2 (cons 3 (cons 4 nil))))
```

```
(define (scale-list items factor)
  (cons (car items)
        (scale-list (cdr items) factor ))
)
```

List Operations: `scale-list`

```
(scale-list (list 1 2 3 4) 5)
```

`(5 10 15 20)`

```
(1 2 3 4) ≡
   (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(define (scale-list items factor)
 (if (null? items)
    nil
    (cons (car items)
          (scale-list (cdr items) factor ))
 ))
```

List Operations: `scale-list`

```
(scale-list (list 1 2 3 4) 5)
(5 10 15 20)

(1 2 3 4) ≡
  (cons 1 (cons 2 (cons 3 (cons 4 nil))))

(define (scale-list items factor)
 (if (null? items)
   nil
   (cons (* (car items) factor)
         (scale-list (cdr items) factor))
 ))
```

List Operations: `scale-list`

```
(define (scale-list items factor)
 (if (null? items)
   nil
   (cons (* (car items) factor)
         (scale-list (cdr items) factor))
 ))
```

List Operations: `scale-list`

```
(define (scale-list items factor)
  (if (null? items)
     nil
     (cons (* (car items) factor)
           (scale-list (cdr items) factor))
  ))
```

List Operations: `scale-list`

```
(define (~~scale-list~~ items proc)
 (if (null? items)
   nil
   (cons (proc (car items) )
         (~~scale-list~~ (cdr items) proc))
 ))
```

List Operations: `map`

```
(define (map items proc)
 (if (null? items)
   nil
   (cons (proc (car items) )
         (map (cdr items) proc))
 ))
```

List Operations: `map`

```
(map (list -10 2 -3 4) abs)
```

List Operations: `map`

```
(map (list -10 2 -3 4) abs)
```

`(10 2 3 4)`

List Operations: `map`

```
(map (list -10 2 -3 4) abs)
```

`(10 2 3 4)`

```
(map (lambda (x) (* x x))
     (list 1 2 3 4))
```

List Operations: `map`

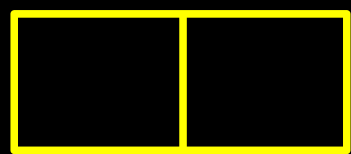
```
(map (list -10 2 -3 4) abs)
```
`(10 2 3 4)`

```
(map (lambda (x) (* x x))
     (list 1 2 3 4))
```
`(1 4 9 16)`

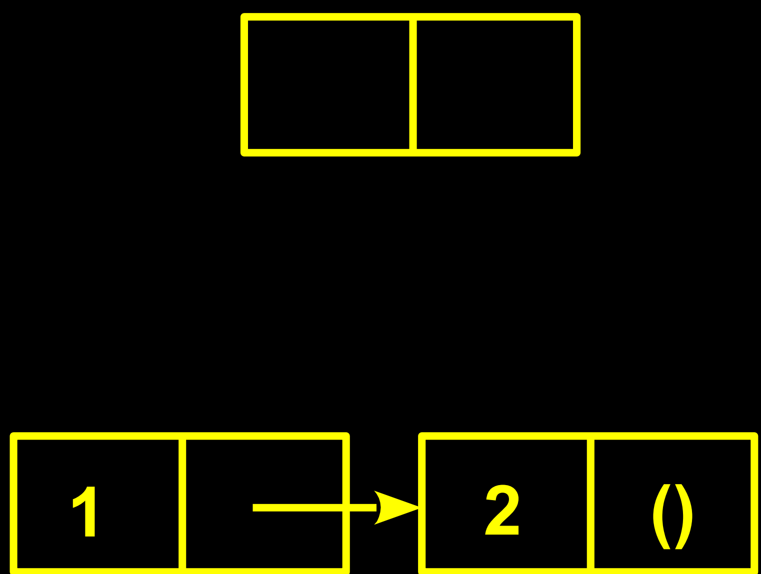
```
(cons (list 1 2) (list 3 4) )
```

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

((1 2) 3 4)
```

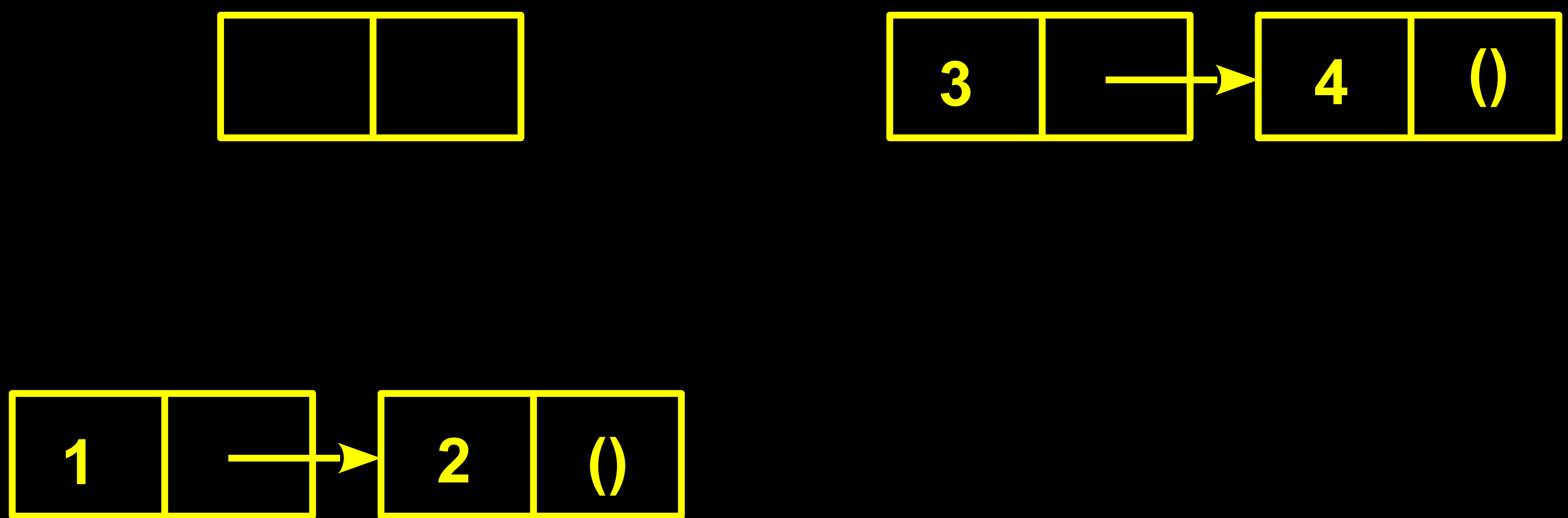
```
(cons (list 1 2) (list 3 4) )
```



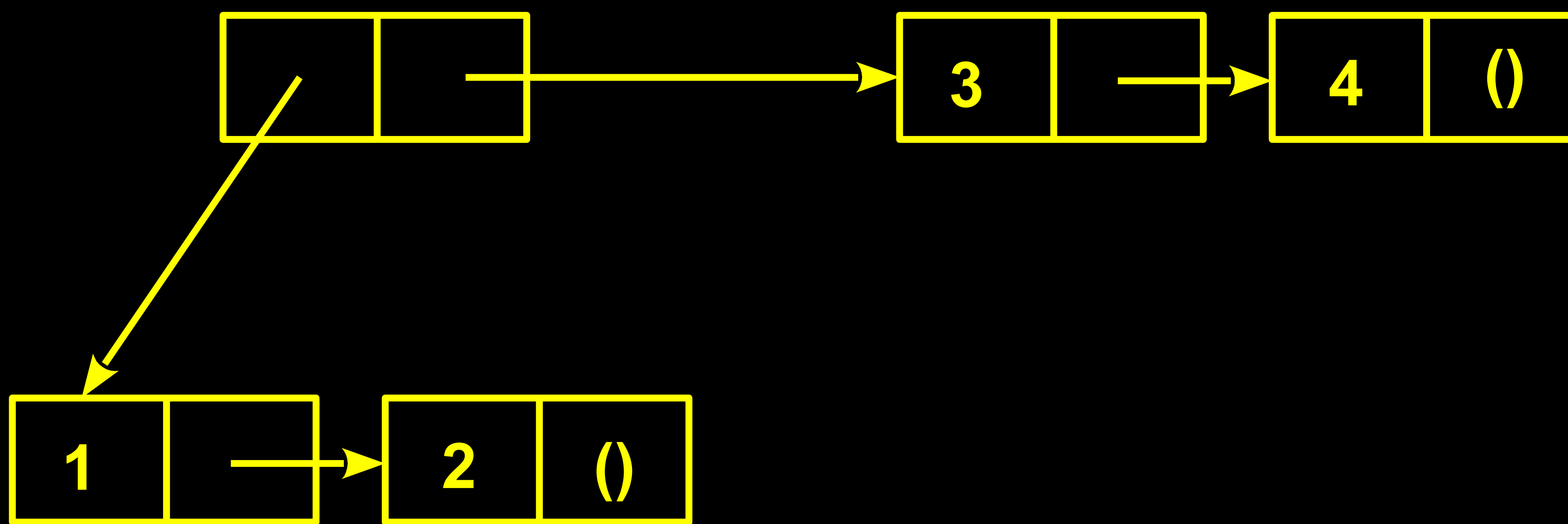
```
(cons (list 1 2) (list 3 4) )
```



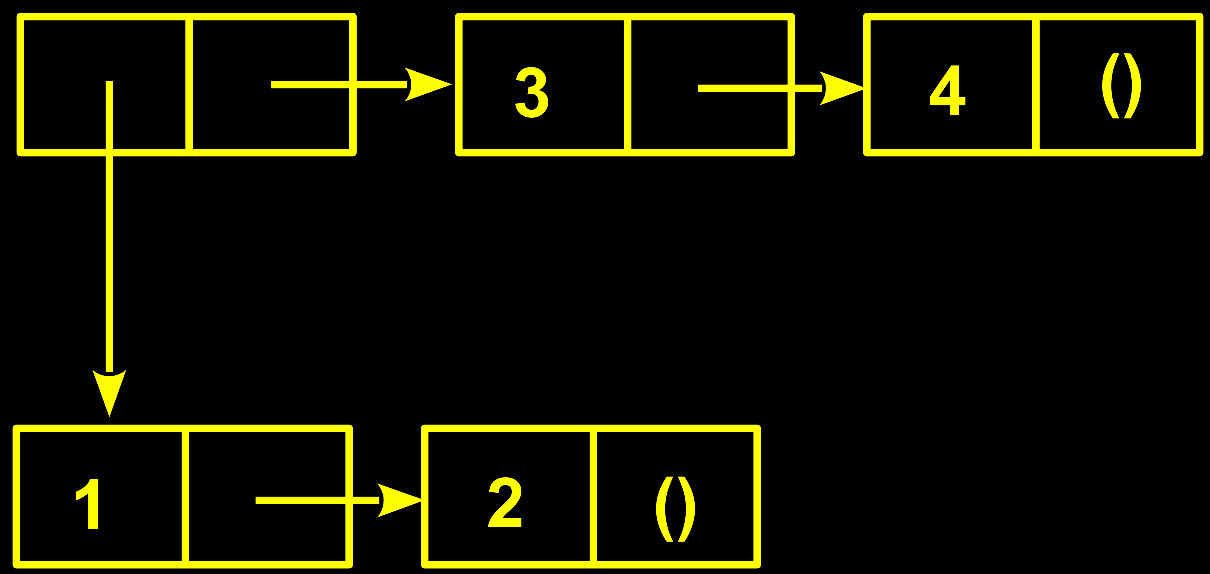
(cons (list 1 2) (list 3 4))



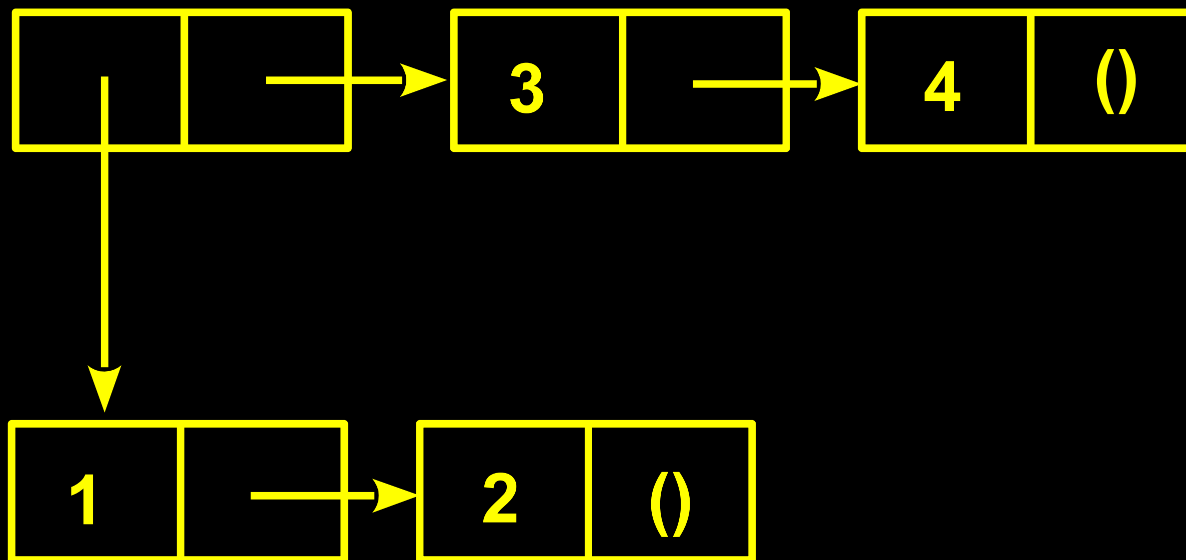
```
(cons (list 1 2) (list 3 4) )
```



(cons (list 1 2) (list 3 4))

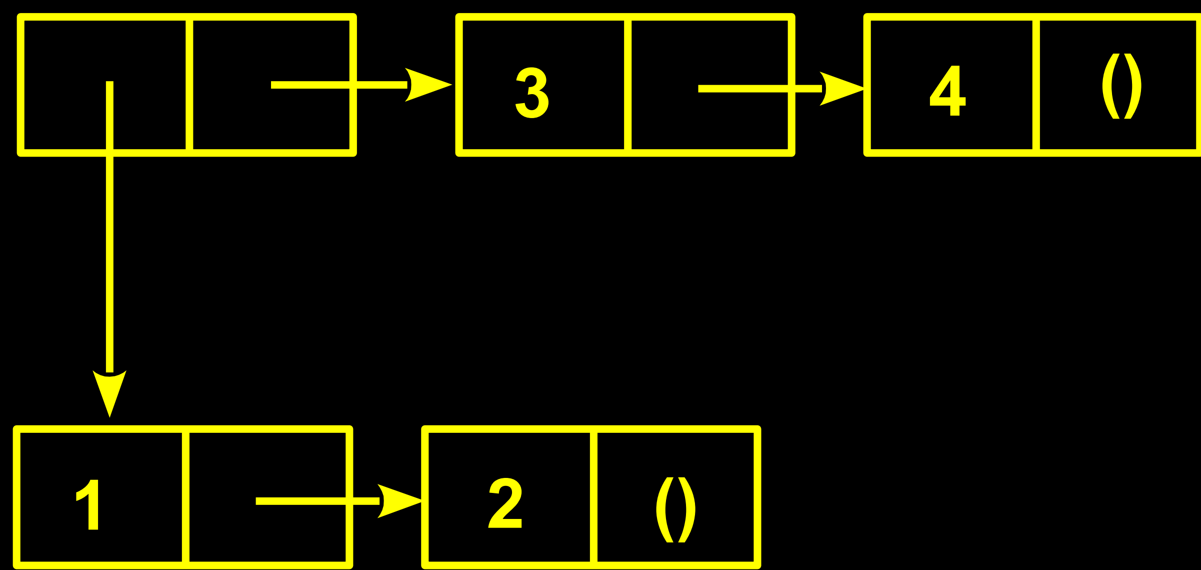


(cons (list 1 2) (list 3 4))



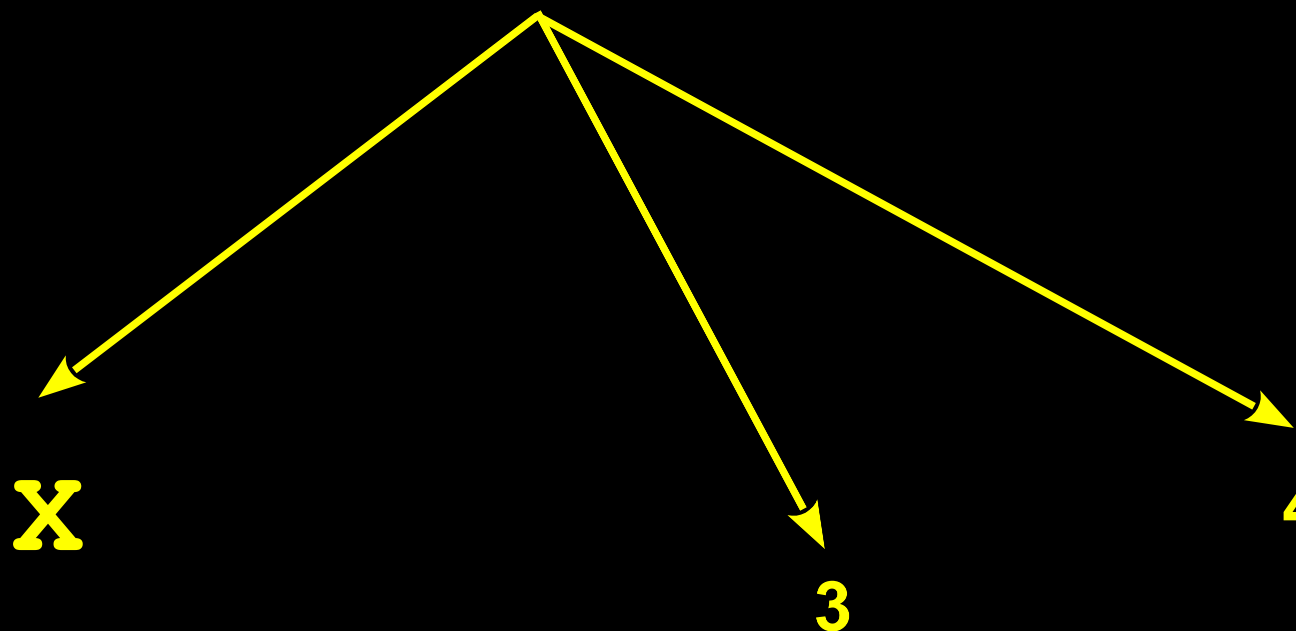
(x 3 4)

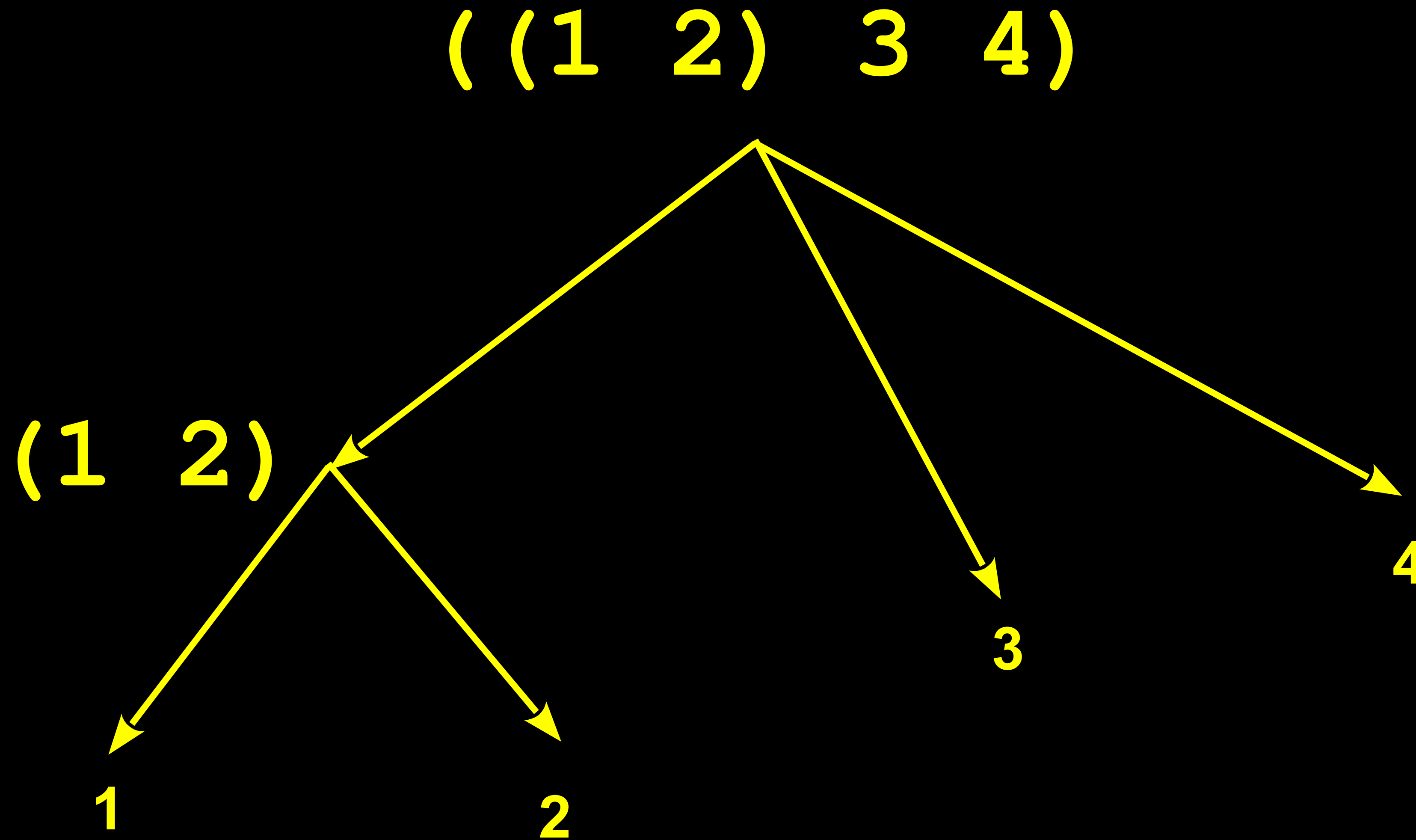
```
(cons (list 1 2) (list 3 4) )
```




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

((1 2) 3 4)




((1 2) 3 4)
(1 2)
1
2
3
4

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

((1 2) 3 4)

```
(length x)
```

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

((1 2) 3 4)

(length x) → **3**

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

((1 2) 3 4)

`(length x)` → **3**

`(list x x)`

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

((1 2) 3 4)

`(length x)` → **3**

`(list x x)` → **((1 2) 3 4)**

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

((1 2) 3 4)

(length x) → **3**

(list x x) → **((1 2) 3 4)((1 2) 3 4)**

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

((1 2) 3 4)

`(length x)` → **3**

`(list x x)` → **(((1 2) 3 4)((1 2) 3 4))**

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

(length x) → 3
(list x x) → (((1 2) 3 4)((1 2) 3 4))
(length (list x x))
```

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

((1 2) 3 4)

(length x) → **3**

(list x x) → **(((1 2) 3 4)((1 2) 3 4))**

(length (list x x))→ **2**

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

((1 2) 3 4)

(length x) → **3**

(list x x) → **(((1 2) 3 4)((1 2) 3 4))**

(length (list x x))→ **2**

(count-leaves x)→ **4**

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

(length x) → 3
(list x x) → (((1 2) 3 4)((1 2) 3 4))
(length (list x x))→ 2

(count-leaves x)→ 4
(count-leaves (list x x))
```

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

((1 2) 3 4)

(length x) → **3**

(list x x) → **(((1 2) 3 4)((1 2) 3 4))**

(length (list x x))→ **2**

(count-leaves x)→ **4**

(count-leaves (list x x))→ **8**

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

(((1 2) 3 4)((1 2) 3 4))

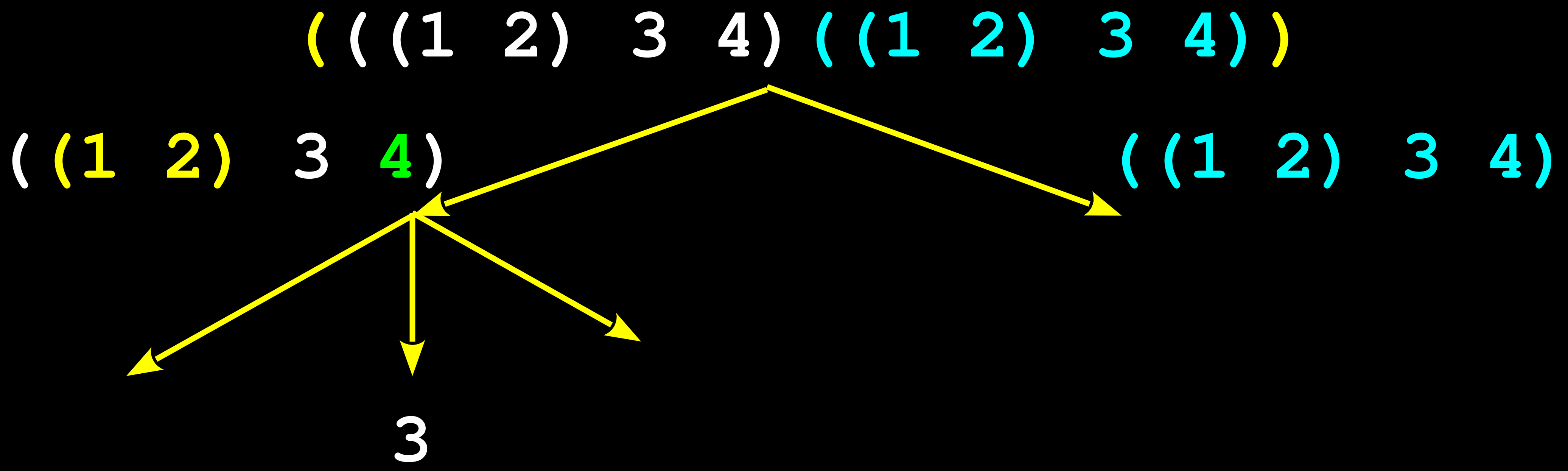
(((1 2) 3 4)((1 2) 3 4))

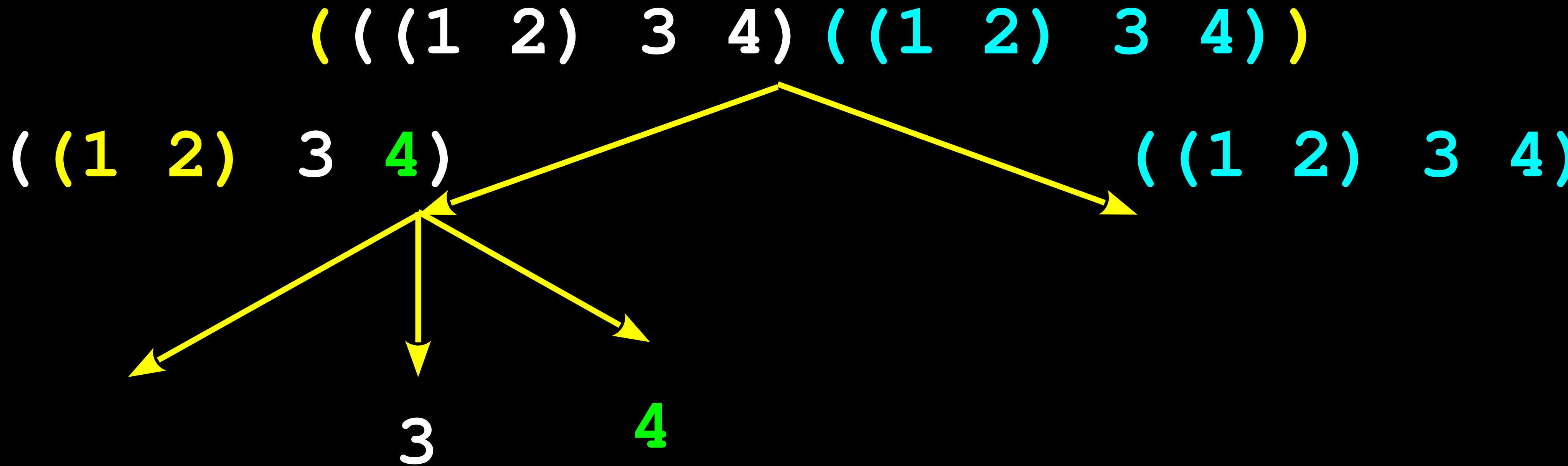
((1 2) 3 4)

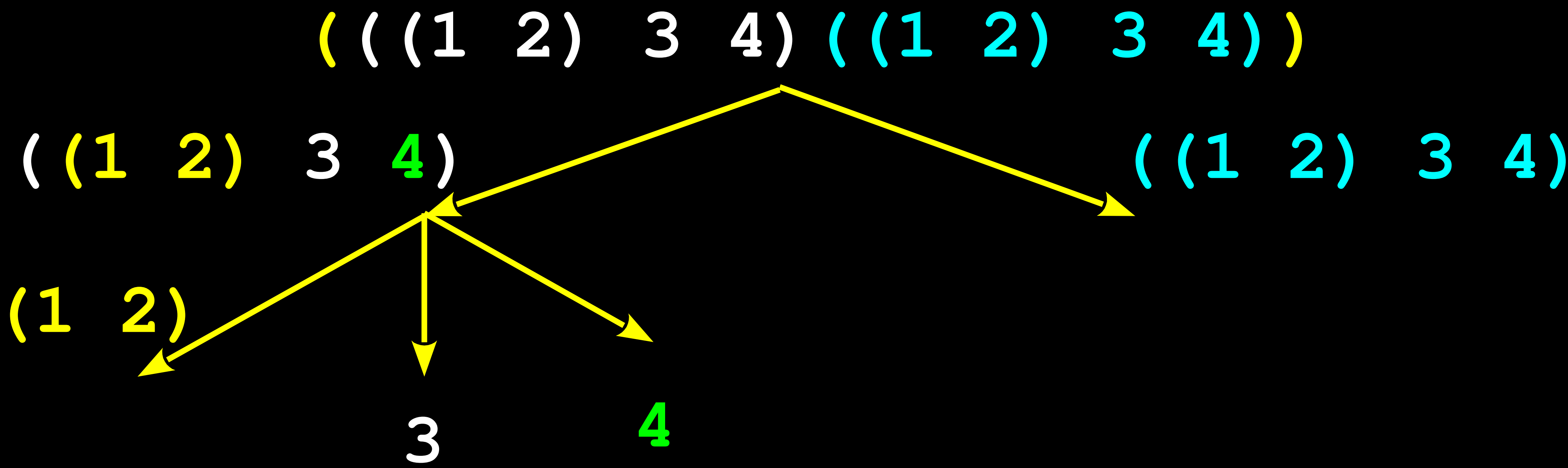
(((1 2) 3 4)((1 2) 3 4))

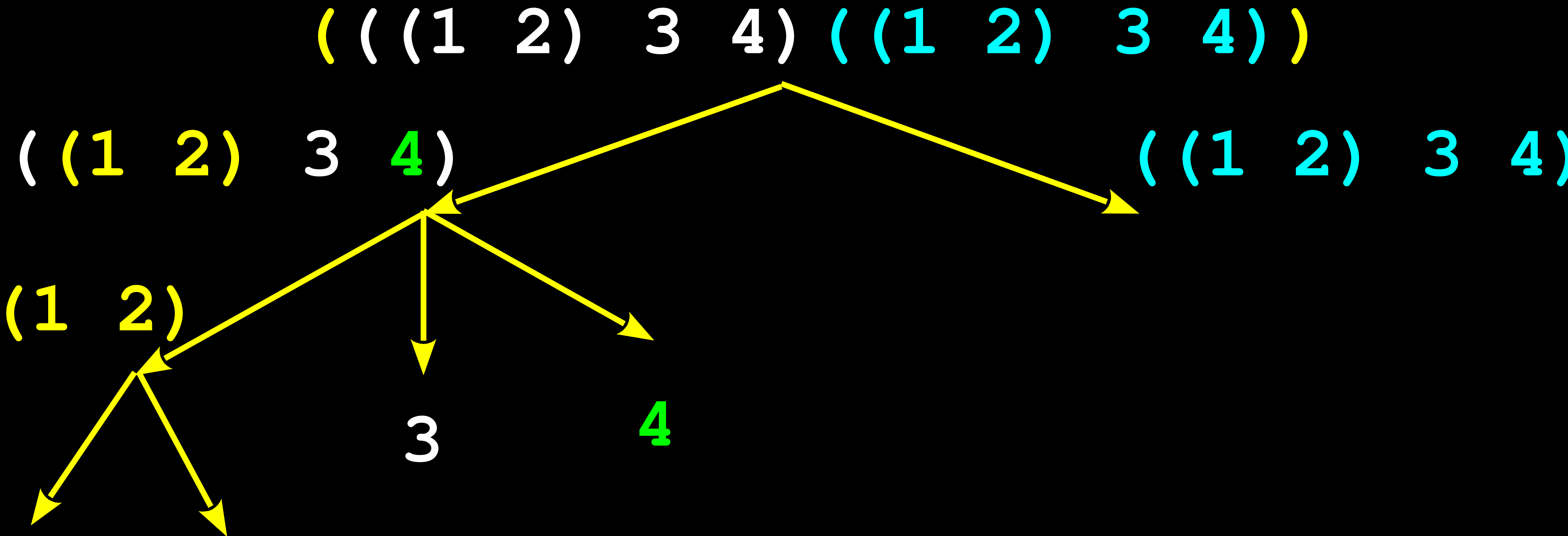
((1 2) 3 4) ((1 2) 3 4)

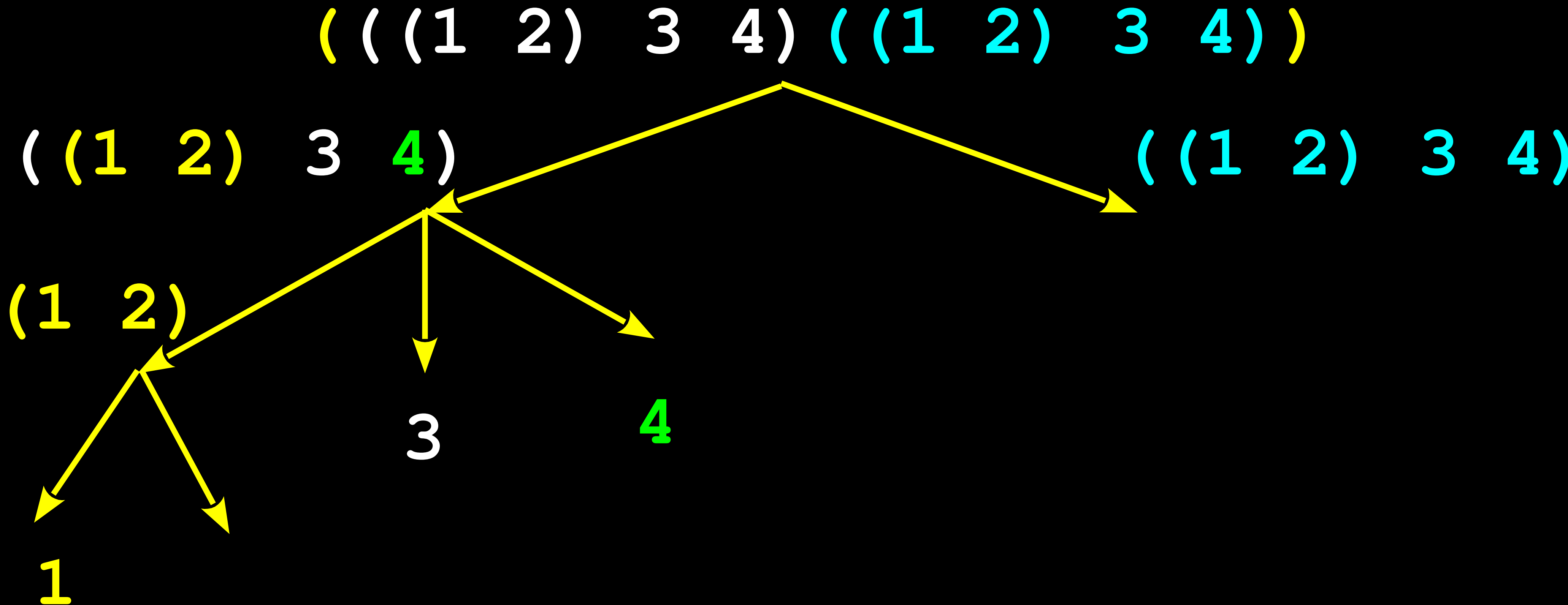
```
(((1 2) 3 4)((1 2) 3 4))

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


(((1 2) 3 4)((1 2) 3 4))
((1 2) 3 4)
((1 2) 3 4)
3

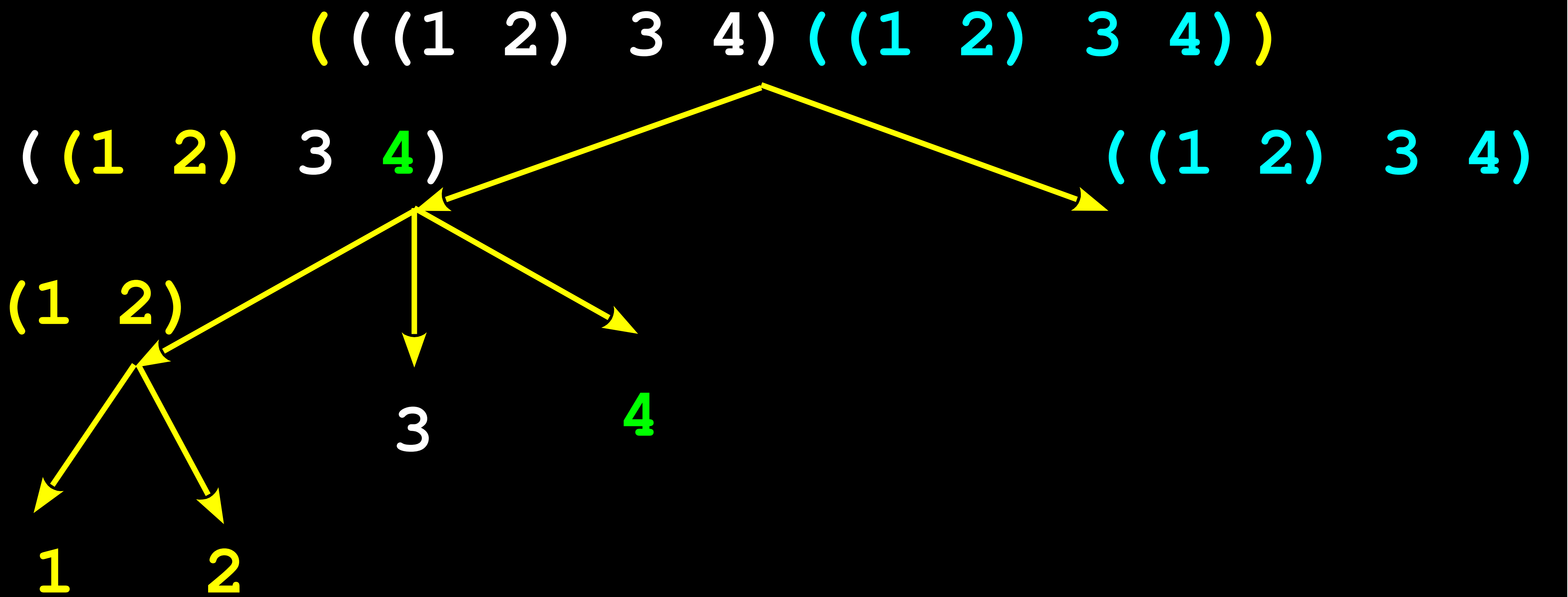

(((1 2) 3 4)((1 2) 3 4))
((1 2) 3 4)
((1 2) 3 4)
3
4


(((1 2) 3 4)((1 2) 3 4))
((1 2) 3 4)
((1 2) 3 4)
(1 2)
3
4

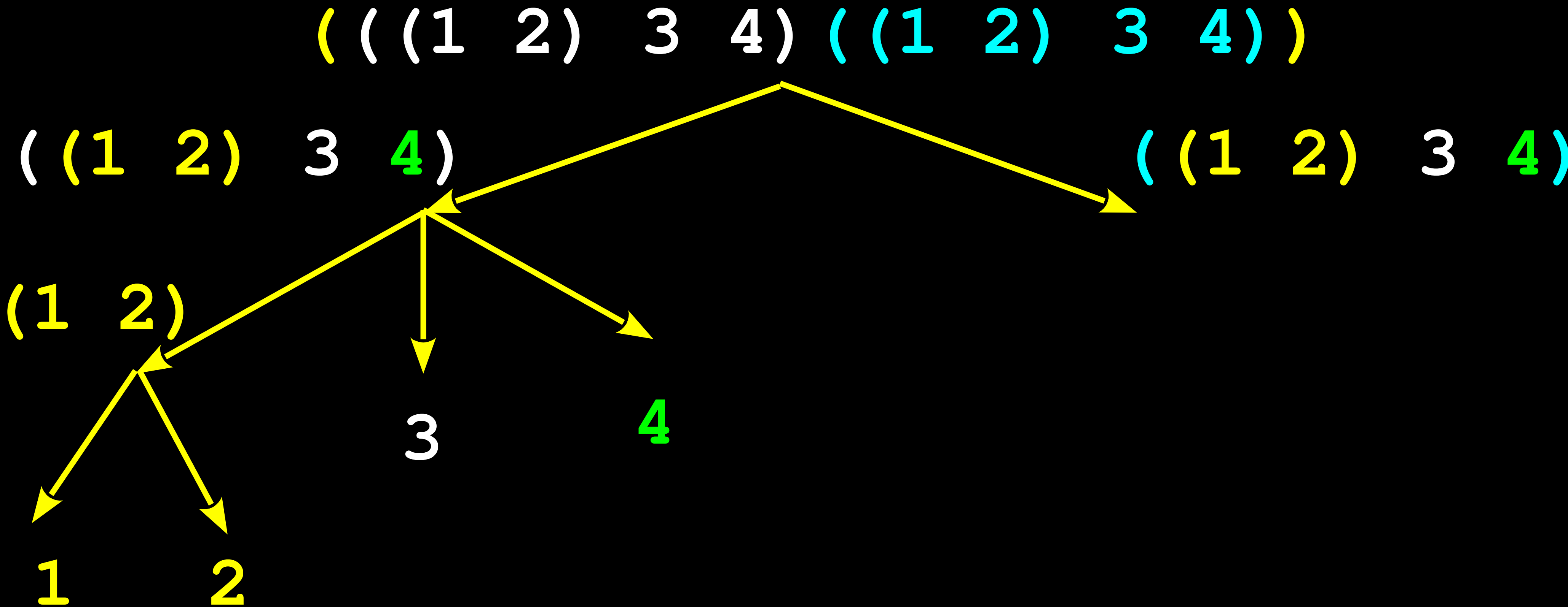

(((1 2) 3 4)((1 2) 3 4))
((1 2) 3 4)
((1 2) 3 4)
(1 2)
3
4

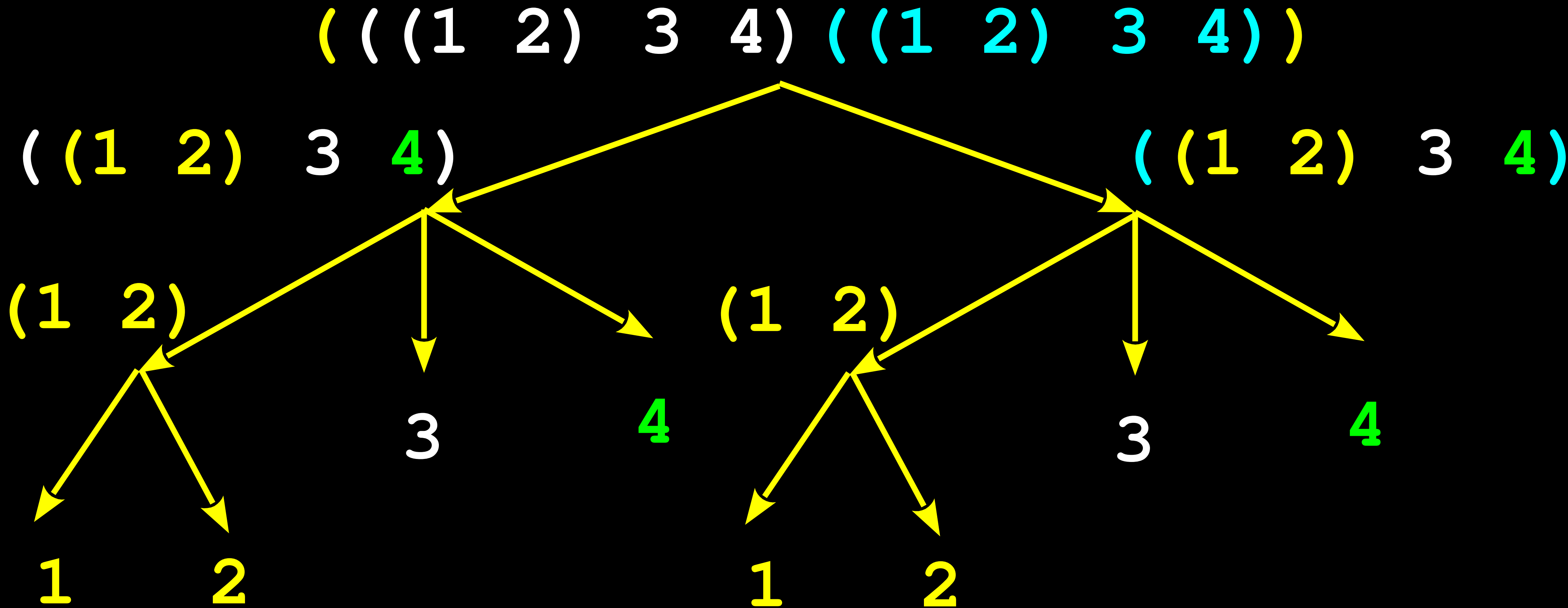


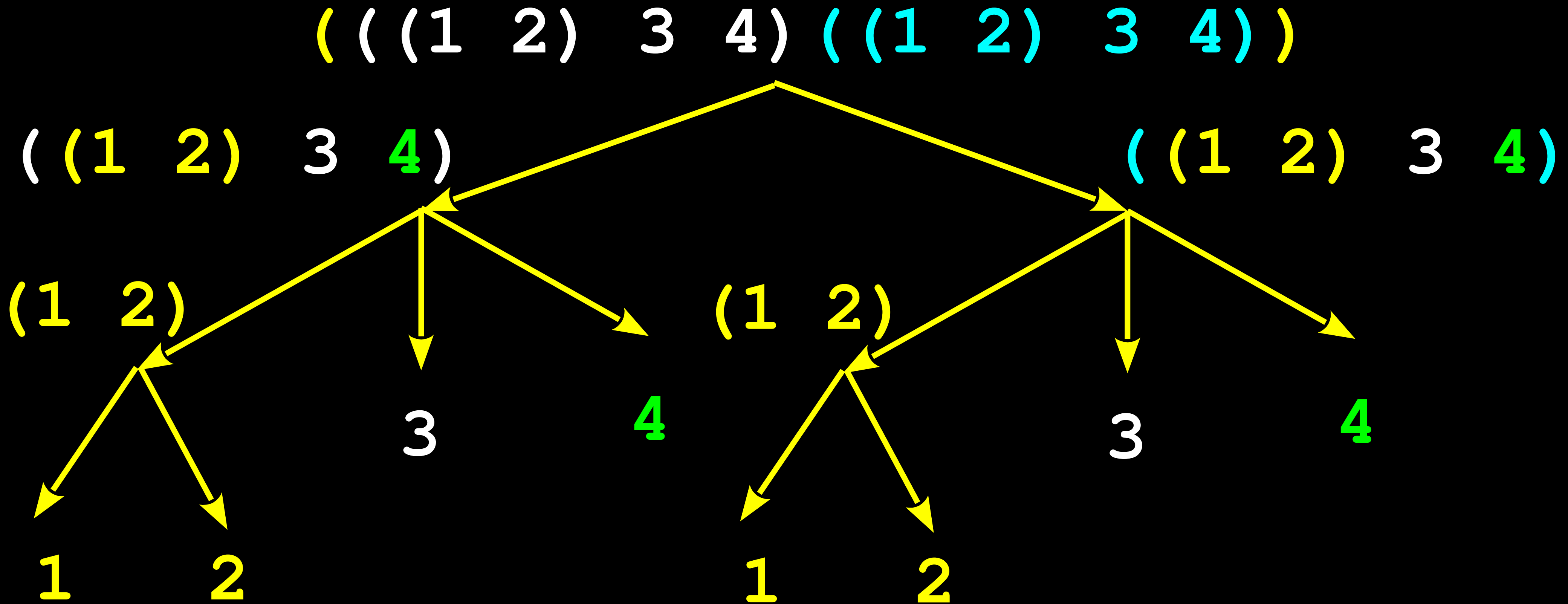
(((1 2) 3 4)((1 2) 3 4))
(((1 2) 3 4)
((1 2) 3 4)
(1 2)
3
4
1



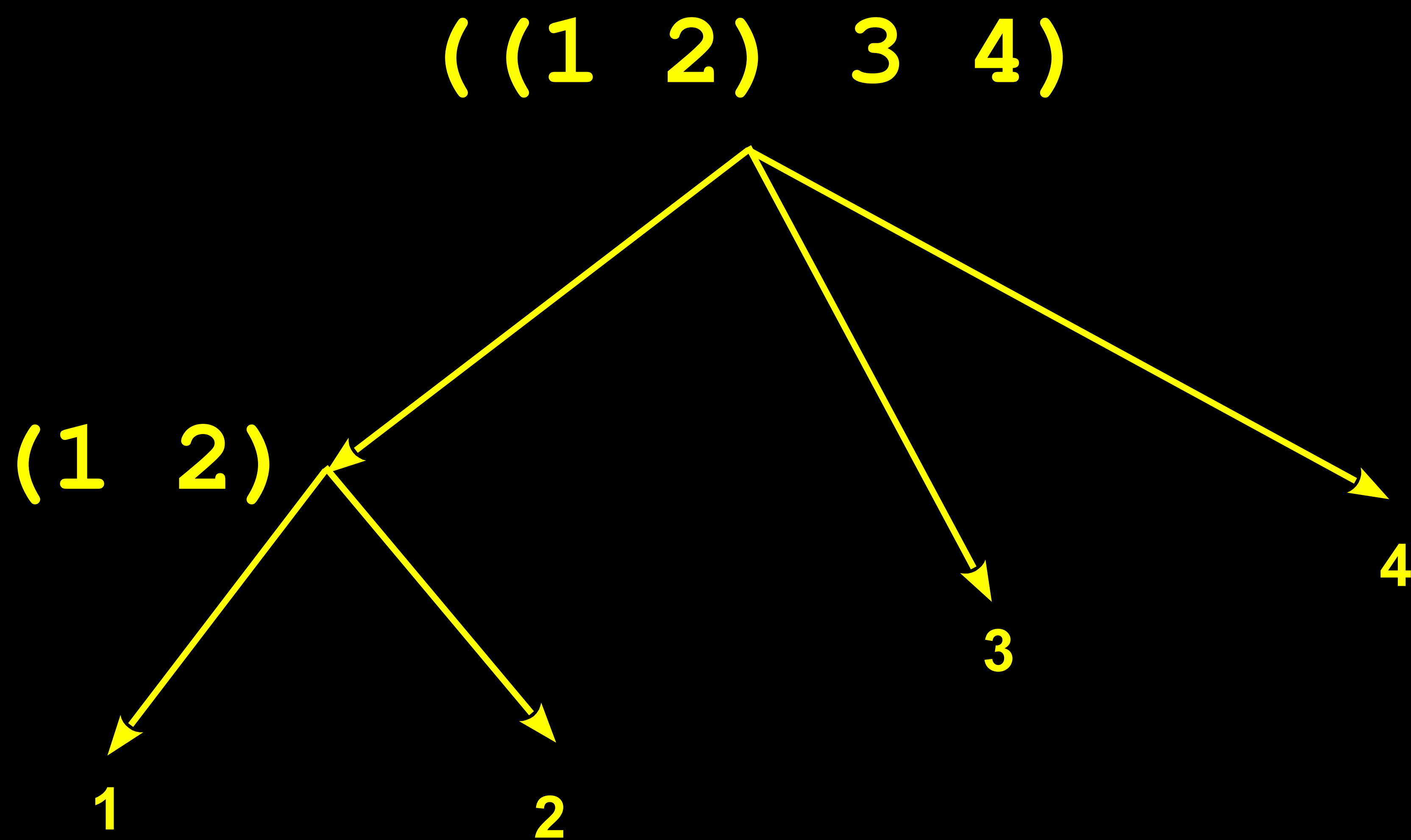
(((1 2) 3 4)((1 2) 3 4))
((1 2) 3 4)
((1 2) 3 4)
(1 2)
3
4
1
2


(((1 2) 3 4)((1 2) 3 4))
((1 2) 3 4)
((1 2) 3 4)
(1 2)
3
4
1
2


(((1 2) 3 4)((1 2) 3 4))
((1 2) 3 4)
((1 2) 3 4)
(1 2)
3
4
(1 2)
3
4
1
2
1
2




```
(count-leaves (list x x))→ 8
```



(count-leaves x) → **4**

length

`(length '(1 3 5 7))`

length = 1 + length of cdr of the list

length of empty list is 0

count-leaves

(count-leaves **'((1 2) 3 4)**)

count-leaves of empty list is 0

count-leaves

(count-leaves **'((1 2) 3 4)**)

count-leaves of empty list is 0

count-leaves of tree =

    count-leaves of car +

    count-leaves of cdr

count-leaves

```
(count-leaves '((1 2) 3 4) )

count-leaves of empty list is 0

count-leaves of tree =
      count-leaves of car +
      count-leaves of cdr

count-leaves of a leaf is 1
```

count-leaves

`(count-leaves` **`'((1 2) 3 4)`** `)`

`=`

count-leaves

```
(count-leaves '((1 2) 3 4) )

= (count-leaves (1 2)) +
  (count-leaves (3 4))
```

count-leaves

```
(count-leaves '((1 2) 3 4) )

= (count-leaves (1 2)) +
  (count-leaves (3 4))

= ((count-leaves 1) +
  (count-leaves (2)))
            +
  ((count-leaves 3) +
   (count-leaves (4))
```

count-leaves

```
= ((count-leaves 1) +
  (count-leaves (2)))
            +
  ((count-leaves 3) +
   (count-leaves (4))

=
```

count-leaves

```
= ((count-leaves 1) +
  (count-leaves (2)))
            +
  ((count-leaves 3) +
   (count-leaves (4))

= 1 + (count-leaves 2)
    + (count-leaves ())
             +
  1 + (count-leaves 4)
    + (count-leaves ())
```

count-leaves

```
= 1 + (count-leaves 2)
    + (count-leaves ())
              +
  1 + (count-leaves 4)
    + (count-leaves ())
```

count-leaves

```
= 1 + (count-leaves 2)
    + (count-leaves ())
          +
  1 + (count-leaves 4)
    + (count-leaves ())

= 1 + 1 + 0
      +
  1 + 1 + 0

=
```

count-leaves

```
= 1 + (count-leaves 2)
    + (count-leaves ())
          +
  1 + (count-leaves 4)
    + (count-leaves ())

= 1 + 1 + 0
      +
  1 + 1 + 0

= 4  (count-leaves '((1 2) 3 4) )
```

count-leaves

```
(count-leaves '((1 2) 3 4) )

count-leaves of empty list is 0

count-leaves of tree =
      count-leaves of car +
      count-leaves of cdr

count-leaves of a leaf is 1
```

count-leaves

```
(count-leaves '((1 2) 3 4) )

count-leaves of empty list is 0

count-leaves of tree =
      count-leaves of car +
      count-leaves of cdr

count-leaves of a leaf is 1
(how to check a leaf?)
```

count-leaves

```
(count-leaves '((1 2) 3 4) )

count-leaves of empty list is 0

count-leaves of tree =
      count-leaves of car +
      count-leaves of cdr

count-leaves of a leaf is 1
(how to check a leaf?)
(a leaf is a non-pair!!)
```

count-leaves

(count-leaves **'((1 2) 3 4)**)

count-leaves of empty list is 0

count-leaves of tree =
        count-leaves of car +
        count-leaves of cdr

count-leaves of a leaf is 1
(how to check a leaf?)
(a leaf is a non-pair!! - **pair?**)

count-leaves

```
(define (count-leaves x)
??
)
```

count-leaves

(count-leaves **'((1 2) 3 4)**)

count-leaves of empty list is 0

count-leaves of tree =
count-leaves of car +
count-leaves of cdr

count-leaves of a leaf is 1
(how to check a leaf?)
(a leaf is a non-pair!! – **pair?**)

count-leaves

```
(define (count-leaves x)
  (cond ( (null? x) 0)
        (               )
        ( else          )
)
```

count-leaves

(count-leaves **'((1 2) 3 4)**)

count-leaves of empty list is 0

count-leaves of tree =
    count-leaves of car +
    count-leaves of cdr

count-leaves of a leaf is 1
(how to check a leaf?)
(a leaf is a non-pair!! - **pair?**)

count-leaves

```
(define (count-leaves x)
   (cond ( (null? x) 0)
         ( (not (pair? x) 1)
         ( else          )
)
```

count-leaves

(count-leaves **'((1 2) 3 4)**)

count-leaves of empty list is 0

count-leaves of tree =
    count-leaves of car +
    count-leaves of cdr

count-leaves of a leaf is 1
(how to check a leaf?)
(a leaf is a non-pair!! - **pair?**)

count-leaves

```
(define (count-leaves x)
   (cond ( (null? x) 0)
         ( (not (pair? x) 1)
         ( else (+ (count-leaves (car x))
                   (count-leaves (cdr x))))
  ))
```

List Operations: `scale-list`

```
(define (scale-list items factor)
 (if (null? items)
    nil
    (cons (* (car items) factor)
          (scale-list (cdr items) factor))
 ))
```

```
(scale-tree
  (list 1 (list 2 (list 3 4) 5) (list 6 7))
  10)
```

```
(scale-tree
  (list 1 (list 2 (list 3 4) 5) (list 6 7))
  10)
```

(10 (20 (30 40) 50) (60 70))

```
count-leaves of empty list is 0

count-leaves of tree =
       count-leaves of car +
       count-leaves of cdr

count-leaves of a leaf is 1
```

```
scale-tree of empty list is 0

scale-tree of tree =
       scale-tree of car +
       scale-tree of cdr

scale-tree of a leaf is
                     (* leaf factor)
```

scale-tree

```
(define (scale-tree tree factor)
 ??
)
```

```
scale-tree of empty list is 0

scale-tree of tree =
       scale-tree of car +
       scale-tree of cdr

 scale-tree of a leaf is
                     (* leaf factor)
```

scale-tree

```
(define (scale-tree tree factor)
   (cond (null? tree) nil)
         (                      )
         (else                  )
)
```

```
scale-tree of empty list is 0

scale-tree of tree =
       scale-tree of car +
       scale-tree of cdr

 scale-tree of a leaf is
                     (* leaf factor)
```

scale-tree

```
(define (scale-tree tree factor)
  (cond (null? tree) nil)
        ((not (pair? tree)) (* tree factor))
        (else                )
)
```

```
scale-tree of empty list is 0

scale-tree of tree =
       scale-tree of car +
       scale-tree of cdr

 scale-tree of a leaf is
                     (* leaf factor)
```

scale-tree

```
(define (scale-tree tree factor)
  (cond (null? tree) nil)
        ((not (pair? tree)) (* tree factor))
        (else (cons
                (scale-tree (car tree) factor)
                (scale-tree (cdr tree) factor)
         ))))
```

List Operations: `scale-list`

```
(define (scale-list items factor)
 (if (null? items)
    nil
    (cons (* (car items) factor)
          (scale-list (cdr items) factor))
 ))
```

List Operations: `map`

```
(define (map proc items)
 (if (null? items)
   nil
   (cons (proc (car items) )
         (map proc (cdr items)))
 ))
```

`scale-list` in terms of `map`

```
(define (scale-list items factor)
  (map (lambda (x) (* x factor)) items)
```

`scale-tree` in terms of `map`

```
(define (scale-tree tree factor)
 (map (lambda (sub-tree)
         (if (pair? sub-tree)
           (scale-tree sub-tree factor)
           (* sub-tree factor)))
      tree))
```

CSE221
Structure and Interpretation of Computer Programs

Conventional Interfaces

sum-odd-squares (ref. scale-tree)

```
(define (scale-tree tree factor)
  (cond (null? tree) nil)
        ((not (pair? tree)) (* tree factor))
        (else (cons
                (scale-tree (car tree) factor)
                (scale-tree (cdr tree) factor)
         ))))
```

sum-odd-squares
(ref. scale-tree)

```
(define (scale-tree tree factor)
   (cond (null? tree) nil)
         ((not (pair? tree)) (* tree factor))
         (else (cons
                 (scale-tree (car tree) factor)
                 (scale-tree (cdr tree) factor)
          ))))
```

sum-odd-squares

```
(define (sum-odd-squares tree)
  (cond (null? tree) )
        ((not (pair? tree)) )
        (else (
             (sum-odd-squares (car tree) )
             (sum-odd-squares (cdr tree) )
         ))))
```

sum-odd-squares

```
(define (sum-odd-squares tree)
  (cond (null? tree) 0)
        ((not (pair? tree)) )
        (else (
             (sum-odd-squares (car tree) )
             (sum-odd-squares (cdr tree) )
         ))))
```

sum-odd-squares

```
(define (sum-odd-squares tree)
  (cond (null? tree) 0)
        ((not (pair? tree))
          (if (odd? tree) (square tree) 0))
        (else (
             (sum-odd-squares (car tree) )
             (sum-odd-squares (cdr tree) )
         ))))
```

sum-odd-squares

```
(define (sum-odd-squares tree)
  (cond (null? tree) 0)
        ((not (pair? tree))
          (if (odd? tree) (square tree) 0))
        (else (+
               (sum-odd-squares (car tree) )
               (sum-odd-squares (cdr tree) )
         ))))
```

???

```
(define (??? n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
(next 0))
```

even-fibs

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
(next 0))
```

Abstract Description of `even-fibs`

- Enumerate the integers from *0* to *n*
- Compute Fibonacci number of each integer
- Filter them selecting even ones
- Accumulate result using `cons`, starting with empty list

- Enumerate the integers from *0* to *n*
  (0 1 2 3 4 5 6 7 8 9 10)

- *Compute Fibonacci number of each* integer
  (0 1 1 2 3 5 8 13 21 34 55)

- Filter them selecting even ones
  (0 2 8 34)

- Accumulate them (0 2 8 34)

sum-odd-squares

```
(define (sum-odd-squares tree)
  (cond (null? tree) 0)
        ((not (pair? tree))
          (if (odd? tree) (square tree) 0))
        (else (
             (sum-odd-squares (car tree) )
             (sum-odd-squares (cdr tree) )
         ))))
```

Abstract Description of `sum-odd-squares`

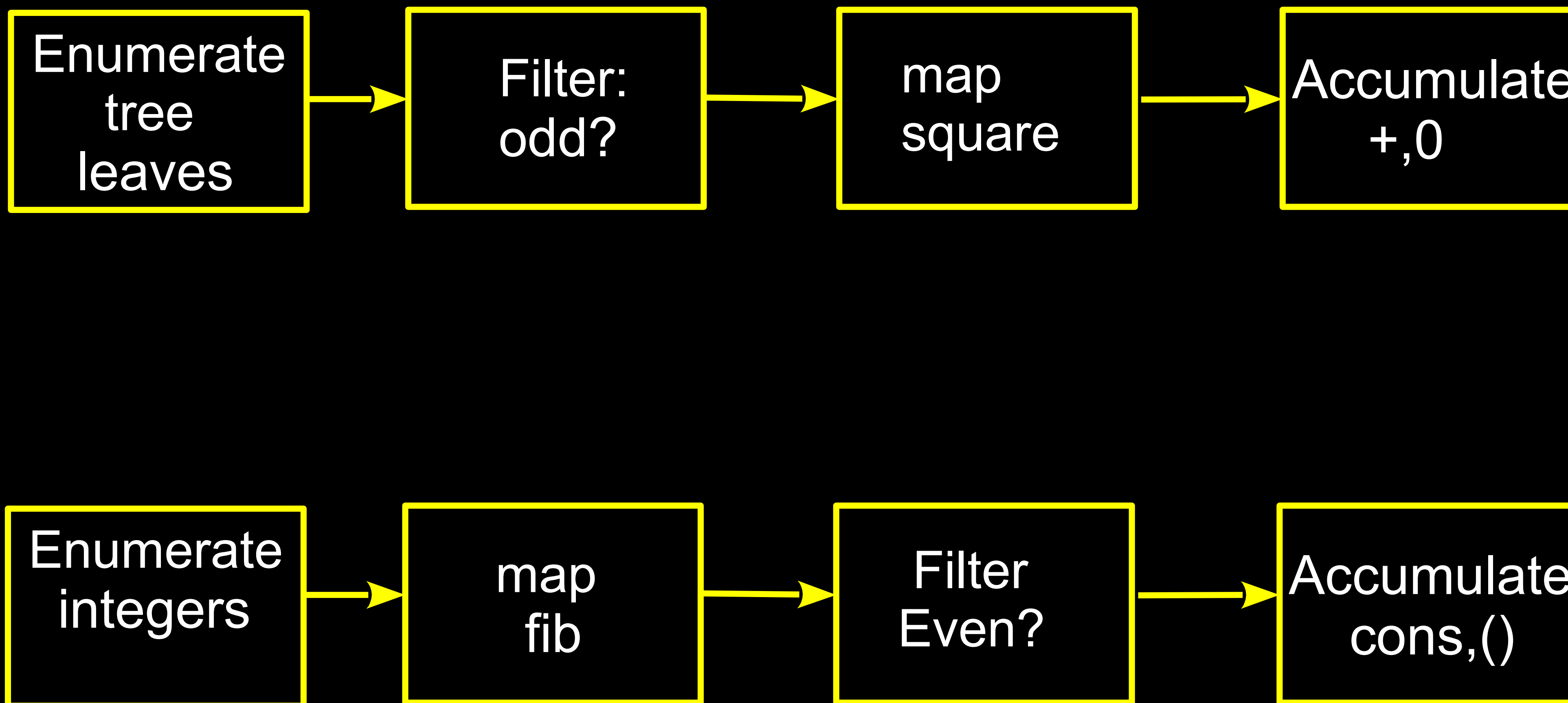
- Enumerate the leaves of a tree
- Filter them selecting odd ones
- Square each of selected odd ones
- Accumulate result using `+`, starting with 0

- Enumerate the leaves of tree
  (list 1 (list 2 (list 3 4)) 5)
  (1 2 3 4 5)

- Filter them selecting odd ones
  (1 3 5)

- Square each of selected ones
  (1 9 25)

- Accumulate them 35


Enumerate tree leaves
Filter: odd?
map square
Accumulate +,0
Enumerate integers
map fib
Filter Even?
Accumulate cons,()

`enumerate-interval` and `enumerate-tree`

```
(enumerate-interval 1 7)
```

`(1 2 3 4 5 6 7)`

enumerate-interval

```
(define (enumerate-interval low high)
 (if (> low high)
    nil
    (cons low
          (enumerate-interval (+ low 1) high)
     )))
```

`enumerate-interval` and `enumerate-tree`

```
(enumerate-interval 1 7)
```
`(1 2 3 4 5 6 7)`

```
(enumerate-tree (list 1 (list 2 3) 4))
```
`(1 2 3 4)`

enumerate-tree

```
(define (enumerate-tree tree)
 (cond ((null? tree) nil)
       ((not (pair? tree)) (list tree))
       (else (append
              (enumerate-tree (car tree))
              (enumerate-tree (cdr tree))))
 ))
```

filter

```
(filter odd? (list 1 2 3 4 5))
```

(1 3 5)

filter

```
(define (filter predicate sequence)
 (cond ((null? sequence) nil)
       ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
       (else (filter predicate (cdr sequence)))))
```

accumulate

```
(accumulate + 0 (list 1 2 3 4 5))
```
15

```
(accumulate * 1 (list 1 2 3 4 5))
```
120

```
(accumulate cons nil (list 1 2 3 4 5))
(1 2 3 4 5)
```

accumulate

```
(define (accumulate op initial sequence)
 (if (null? sequence)
     initial
     (op (car sequence)
         (accumulate op initial (cdr sequence)))
 ))
```

sum-odd-squares

```
(define (sum-odd-squares tree)
  (cond (null? tree) 0)
        ((not (pair? tree))
          (if (odd? tree) (square tree) 0))
        (else (+
             (sum-odd-squares (car tree) )
             (sum-odd-squares (cdr tree) )
         ))))
```

sum-odd-squares

```
(define (sum-odd-squares tree)

 ??

)
```

sum-odd-squares

```
(define (sum-odd-squares tree)




                       (enumerate-tree tree)
                    )
```

sum-odd-squares

```
(define (sum-odd-squares tree)



                  (filter odd?
                      (enumerate-tree tree)
                   ))
```

sum-odd-squares

```
(define (sum-odd-squares tree)

             (map square
                  (filter odd?
                          (enumerate-tree tree)
                  )))
```

sum-odd-squares

```
(define (sum-odd-squares tree)
 (accumulate +
             0
             (map square
                  (filter odd?
                          (enumerate-tree tree)
                   ))))
```

even-fibs

```
(define (even-fibs n)
 (accumulate cons
             nil
             (filter even?
                     (map fib
                          (enumerate-interval
                            0 n)
                     ))))
```

list-fib-squares

```
(define (list-fib-squares n)
 (accumulate cons
             nil
             (map square
                  (map fib
                       (enumerate-interval
                         0 n)
                   ))))
```

prod-sqr-odd-elements

```
(define (prod-sqr-odd-elements sequence)
 (accumulate *
             1
             (map square
                  (filter odd? sequence)
              )))
```

sal-of-hghst-paid-pgmr

```
(define (sal-of-hghst-paid-pgmr records)
 (accumulate max
             0
             (map salary
                  (filter programmer?
                          records)
              )))
```