



```
( define talk  
  ( make-presentation  
    ( make-title  
      "Recursion"  
      "Lather. Rinse. Repeat." )  
    ( make-author  
      "Mike Harris" "@MikeMKH" ) ) )
```



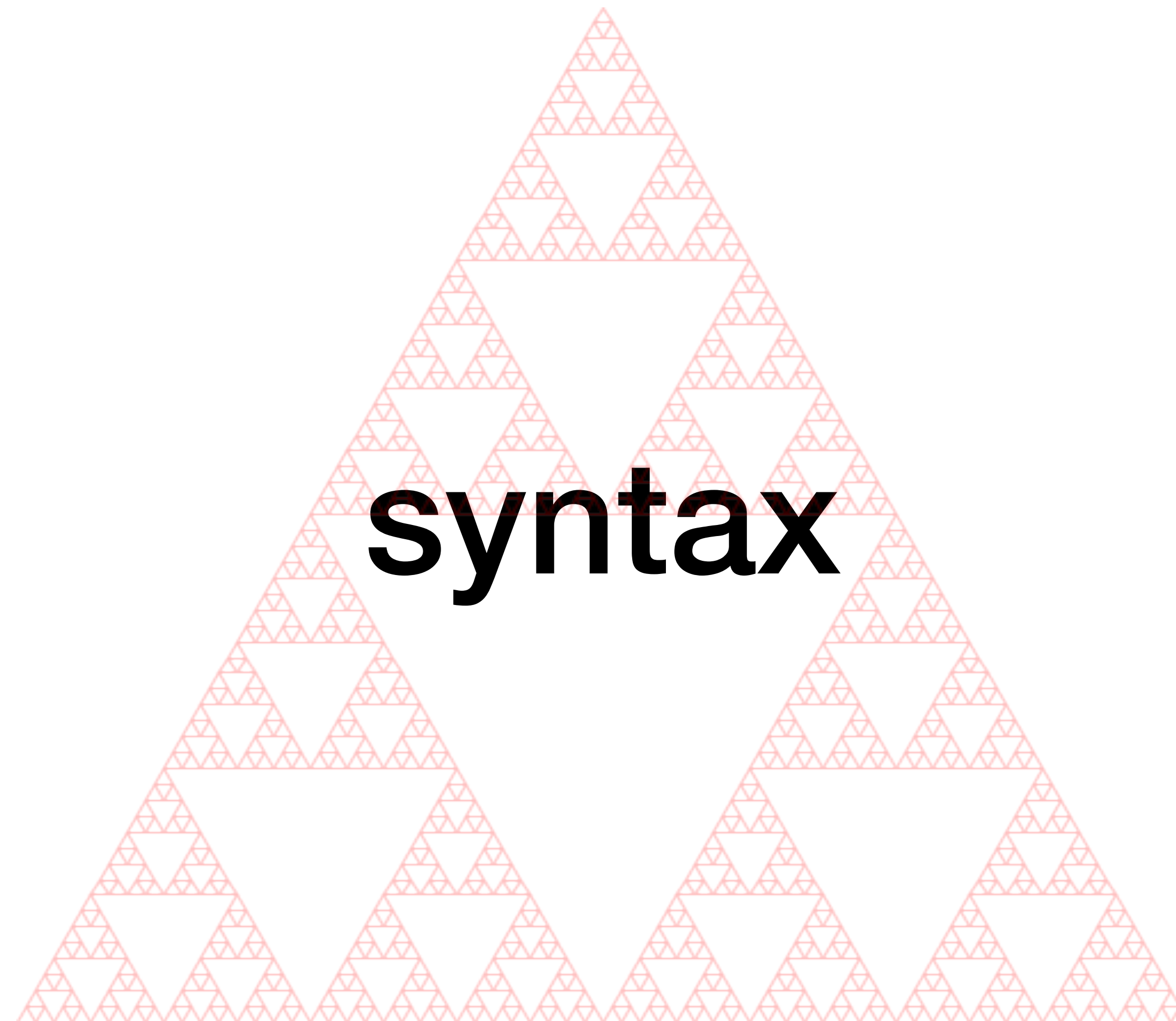


```
(define talk-goals  
  ('("intro to solving problems with recursion"  
     "help in understanding FP examples")))
```



```
(define topics  
  (make-agenda  
    ('("Recursive Data Type"  
       "Structural Recursion"  
       "General Recursion"))))
```





**syntax**

5

5

**atom**



**plus5 (n : number) : number**  
**n + 5**

```
(define (plus-5 n)  
  (+ n 5))
```

**definition\***





**plus5(42)**  
**// 47**

( plus-5 42 )  
; 47

**function application**



```
(define topics  
  (make-agenda  
    (' ("Recursive Data Type"  
        "Structural Recursion"  
        "General Recursion" )))
```





**grammar**

*expr* = (**name** *expr* *expr* ...)

| **'()**

| *number*


| *boolean*

| *magic*

| (**cond** [*expr* *expr*] ...

**[else** *expr* *expr*])

*magic* = (define *name expr*)  
| (define (*name expr*))  
| (check-expect *expr expr*)  
| (check-satisfied  
    *expr expr*)



**cons list**

*list* = (**cons** *value list*)  
| '()

*value* = *number*  
| *boolean*

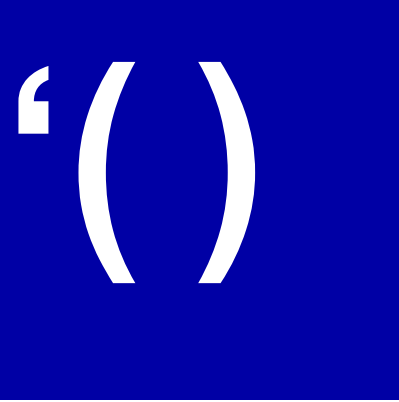


# constructors

*; list – constructors*  
*; – (cons value list)*  
*; – '()*



**empty**

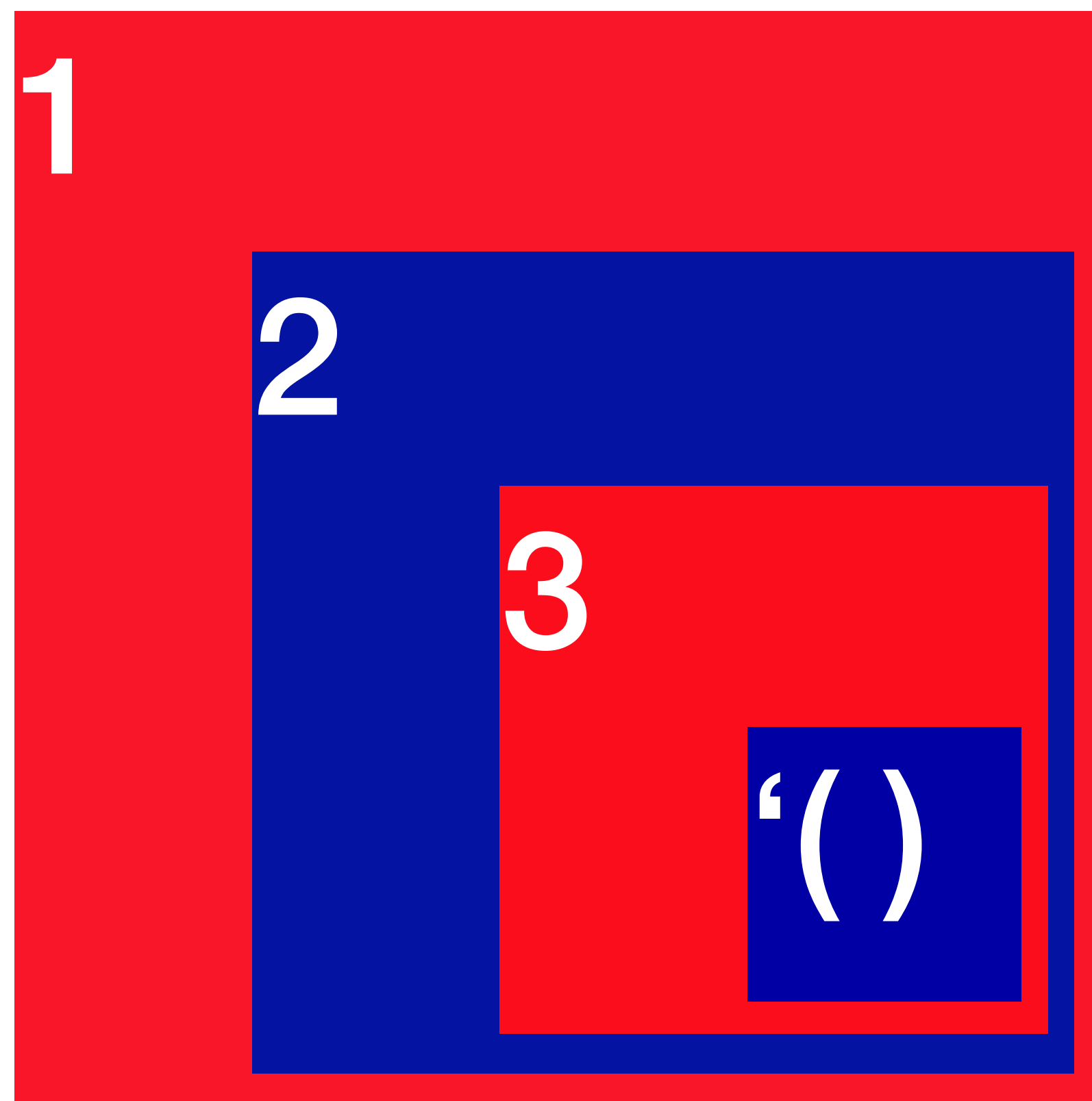


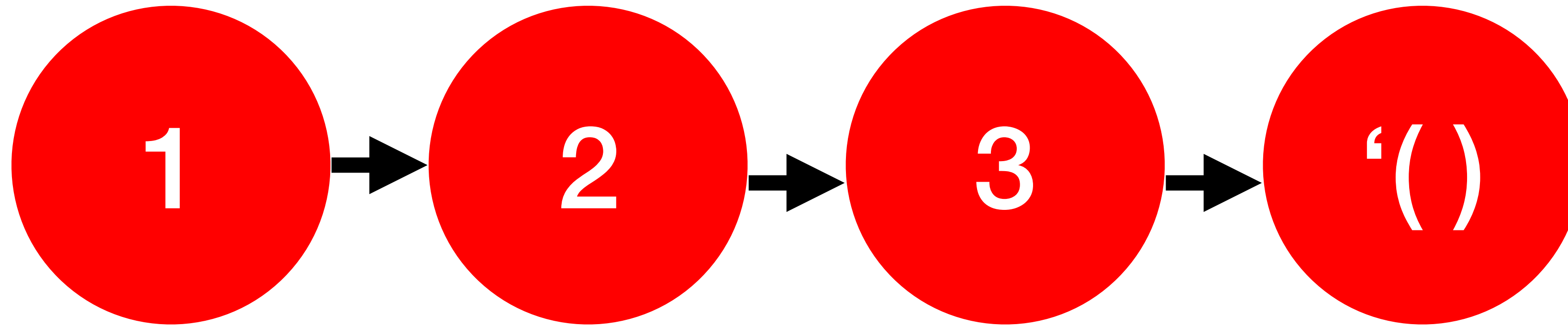


' ( )



**cons**





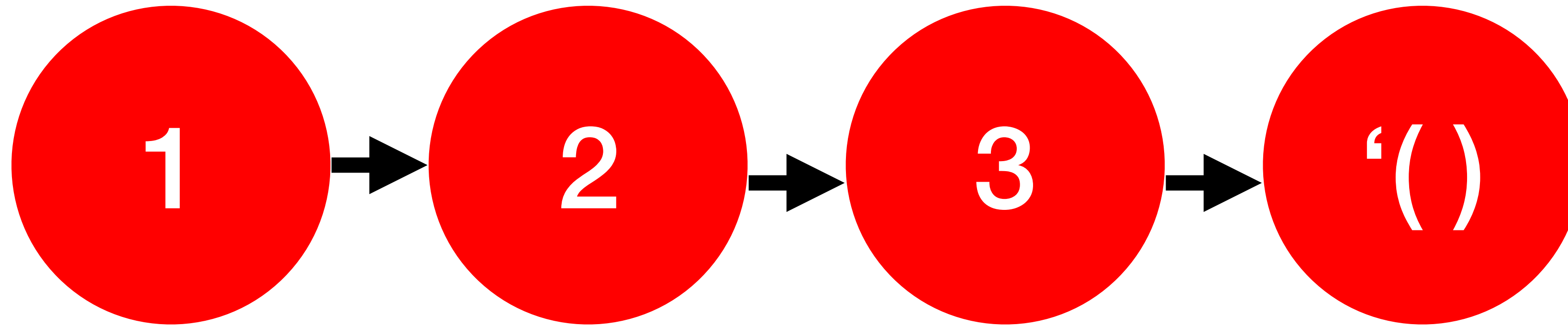
( cons 1

( cons 2

( cons 3 ' ( ) ) ) )

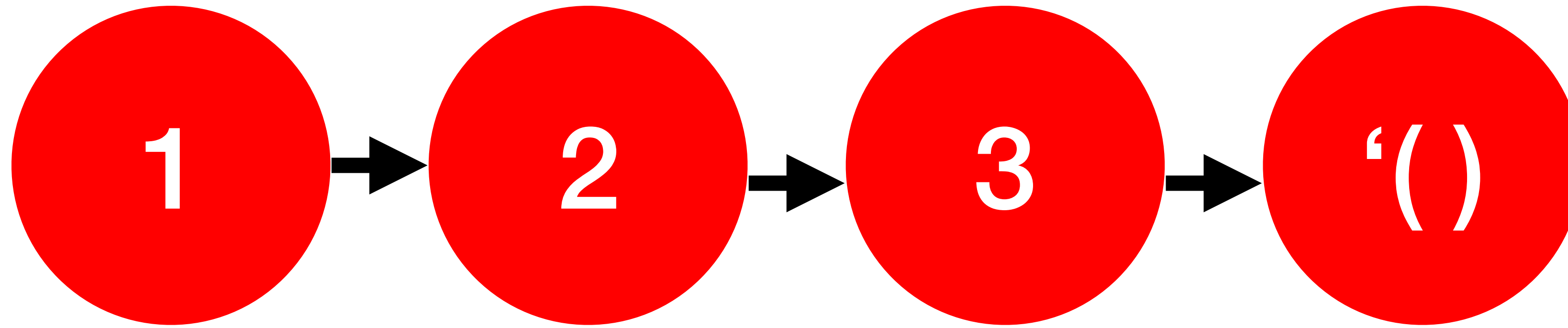






( cons 1 ( cons 2 ( cons 3 ' ( ) ) ) )





' ( 1 2 3 )

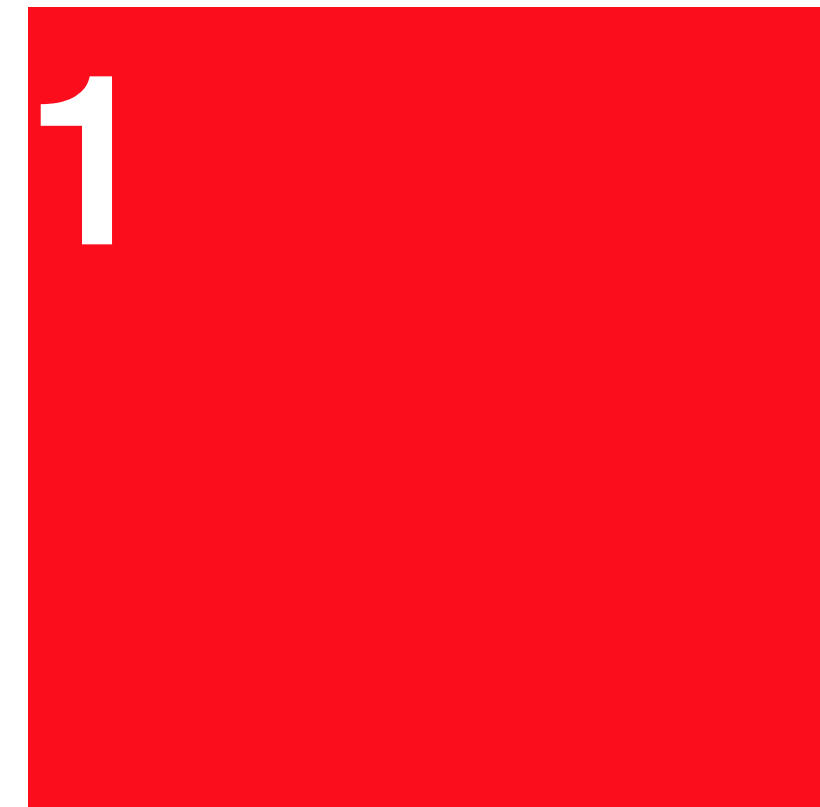
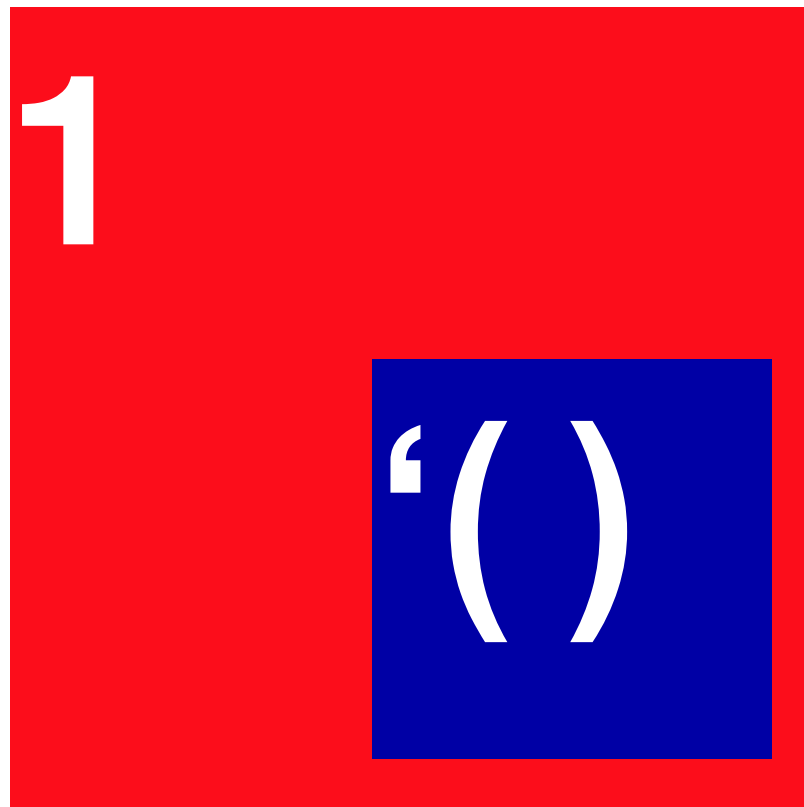


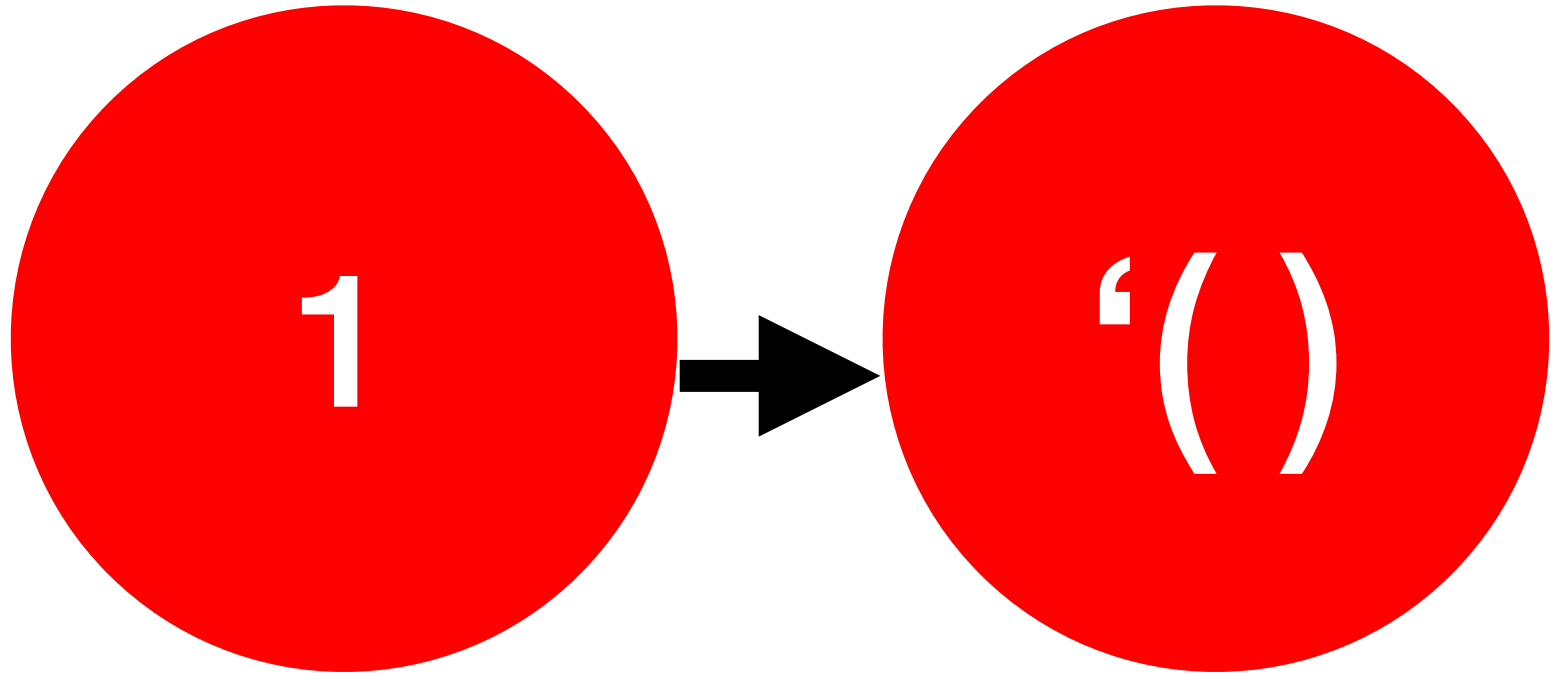
**selectors**

*; list - selectors*  
*; - first : list -> value*  
*; - rest : list -> list*



**first : list -> value**





```
( check-expect  
  ( first ( cons 1 ' ( ) ) )  
  1 )
```

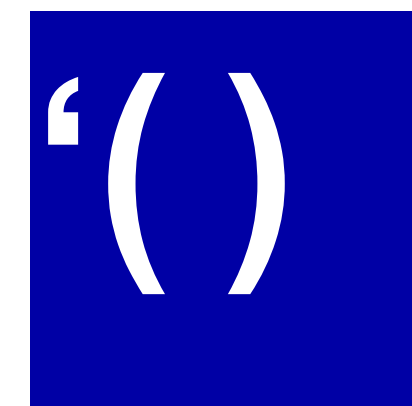
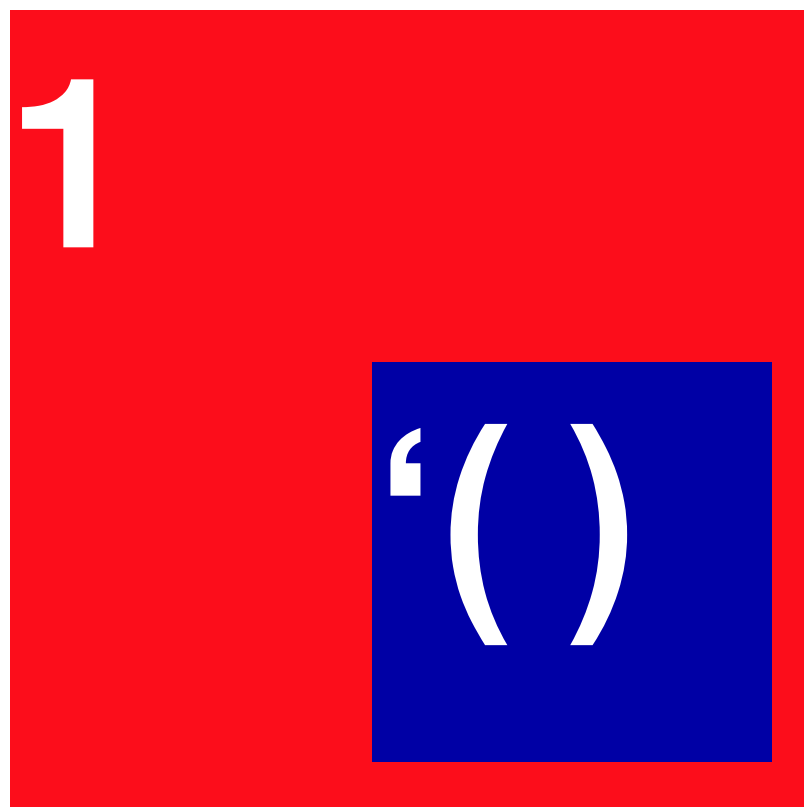


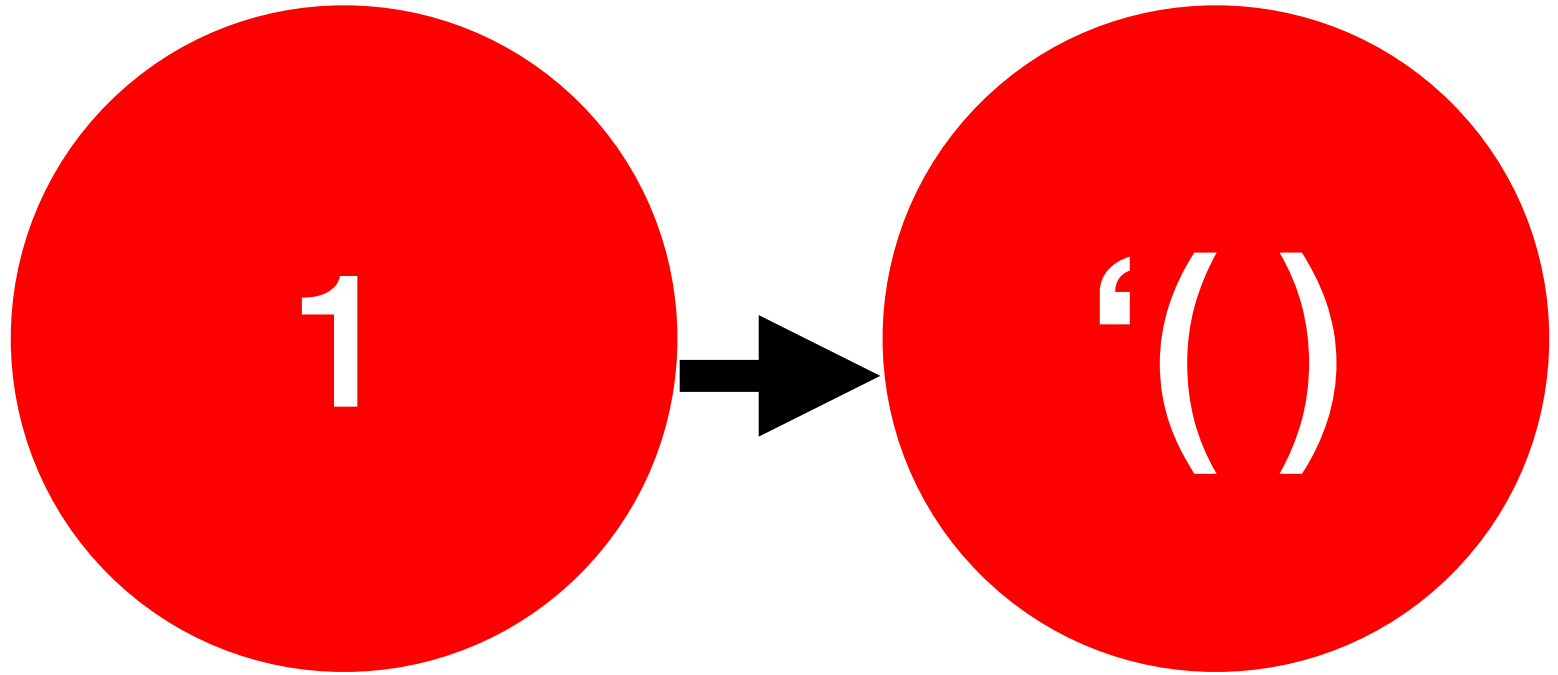


```
( check-expect  
  ( first '(1))  
  1)
```



**rest : list -> list**





```
( check-expect  
  ( rest ( cons 1 ' ( ) ) )  
  ' ( ) )
```



```
( check-expect  
  ( rest '( 1 ) )  
  ' ( ) )
```



**predicates**

*; list - predicates*  
*; - empty? : list -> boolean*

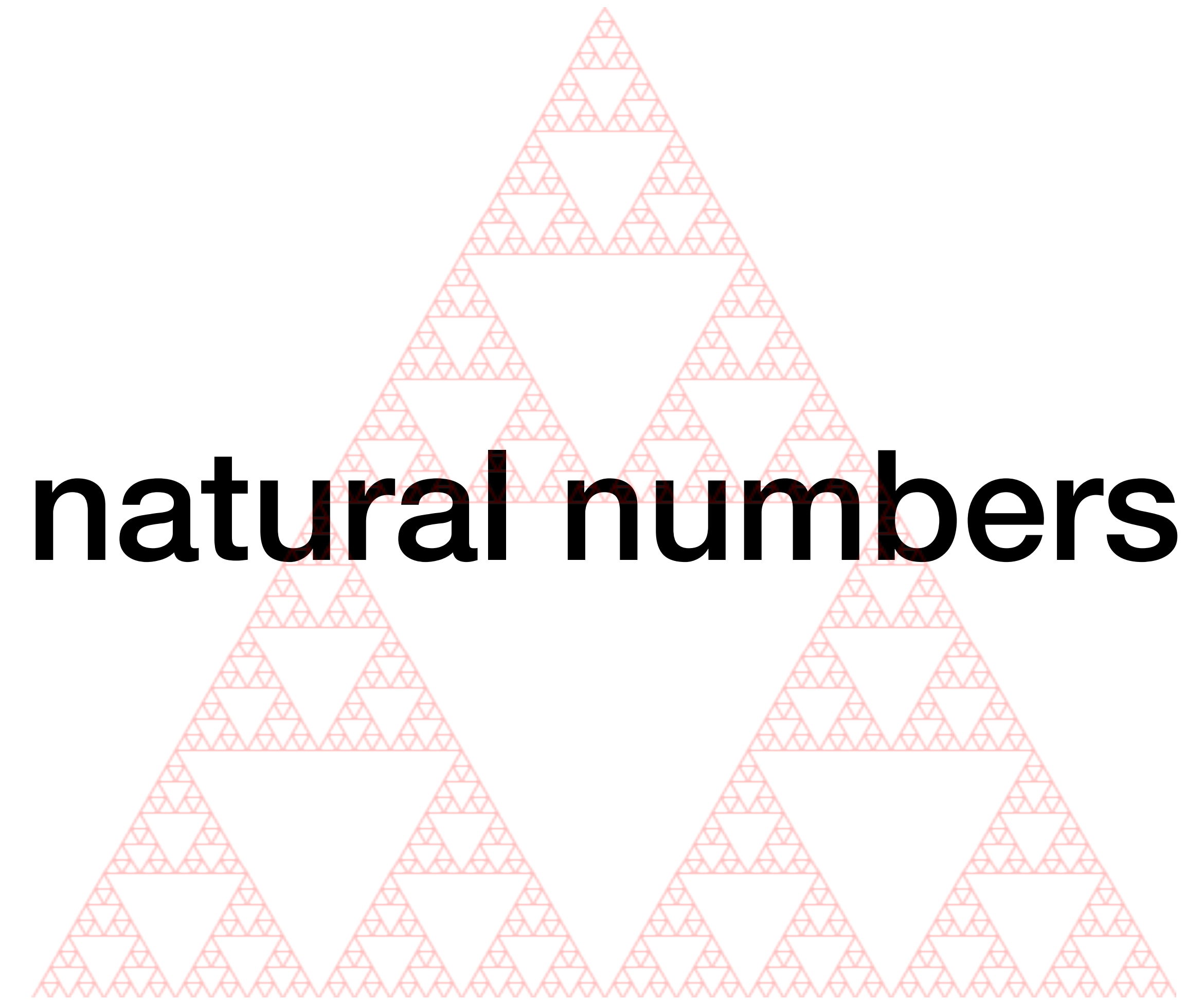




**empty? : list -> boolean**

( check-satisfied  
' (  
empty? )





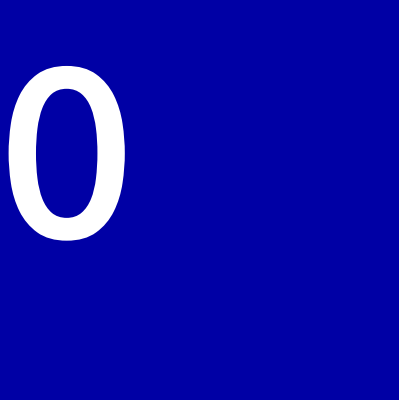
*nat* = 0  
| (add1 *nat*)

# constructors

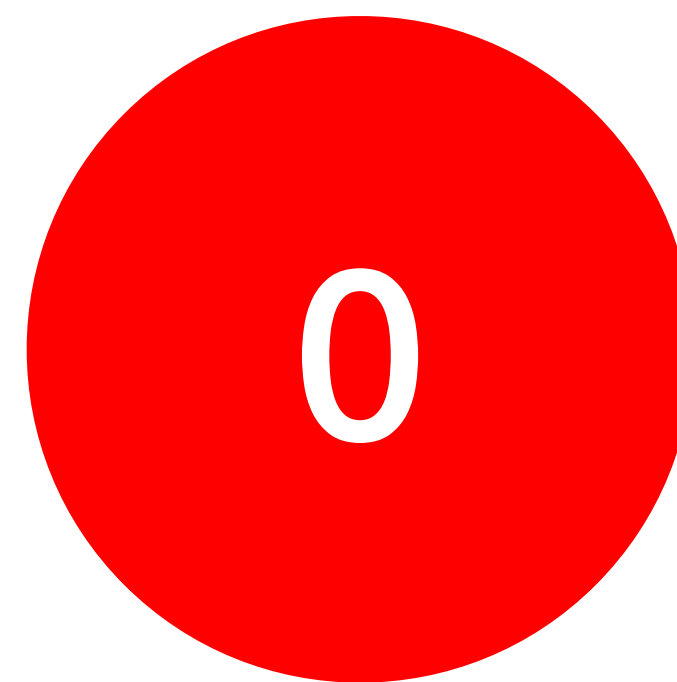
*; nat – constructors*  
*; – 0*  
*; – (add1 nat)*



**zero**



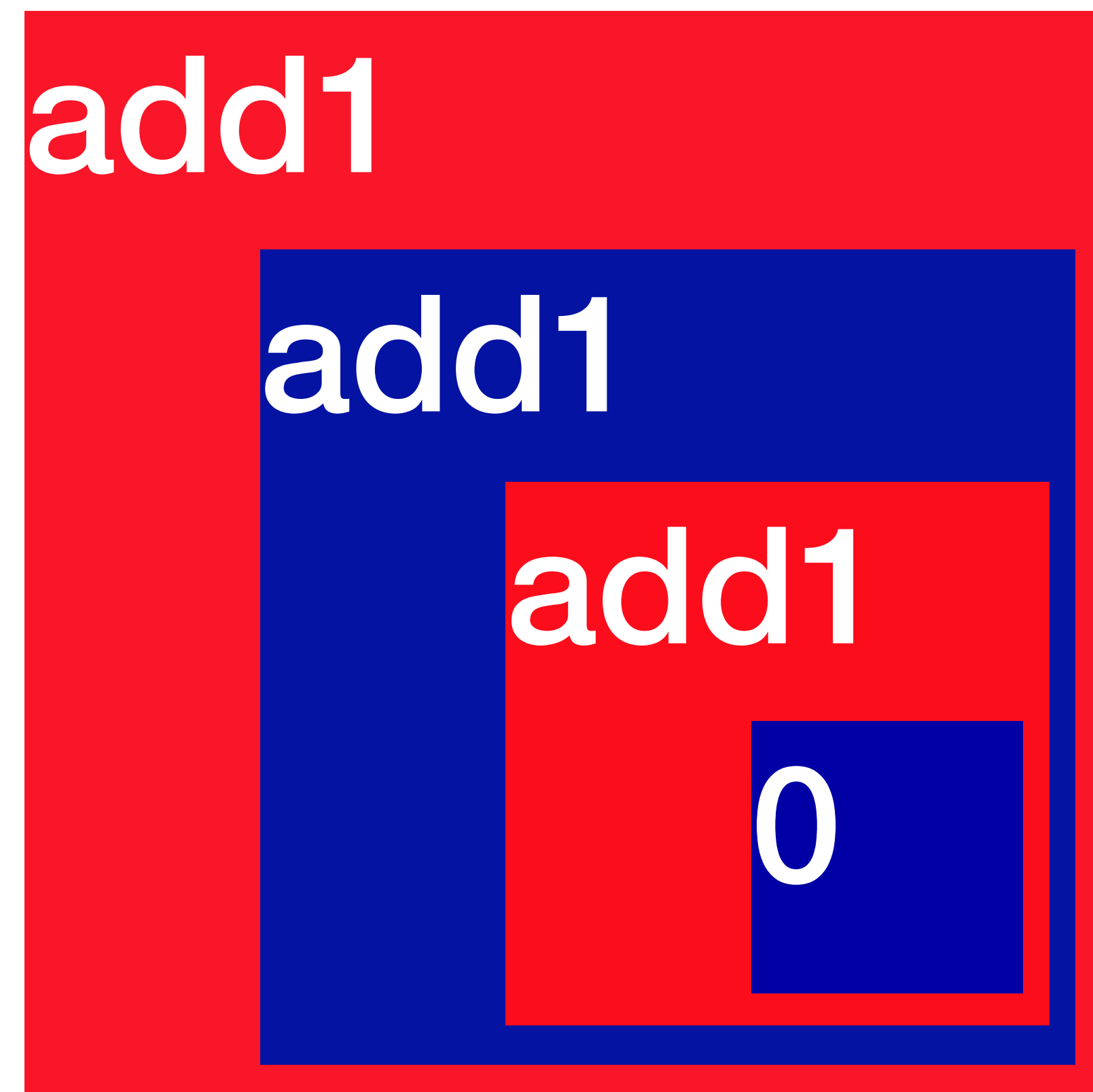


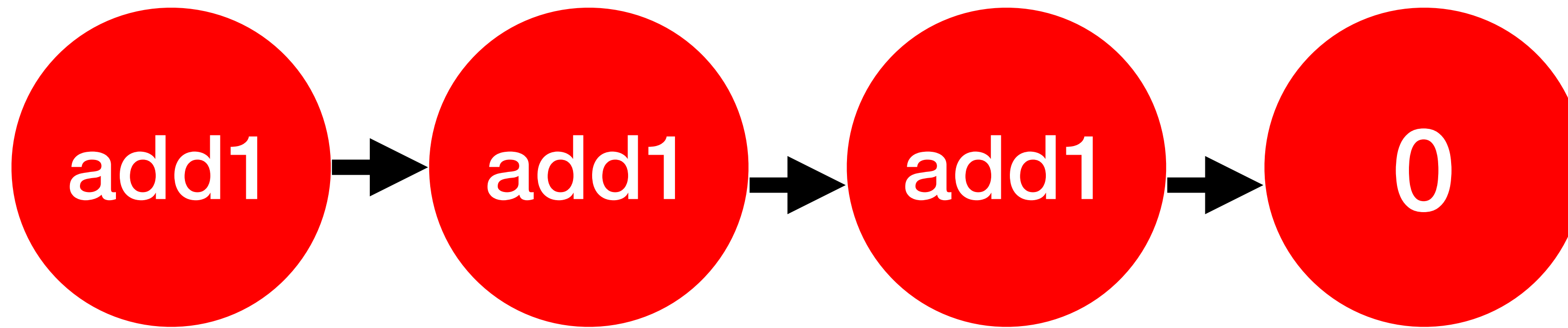


0



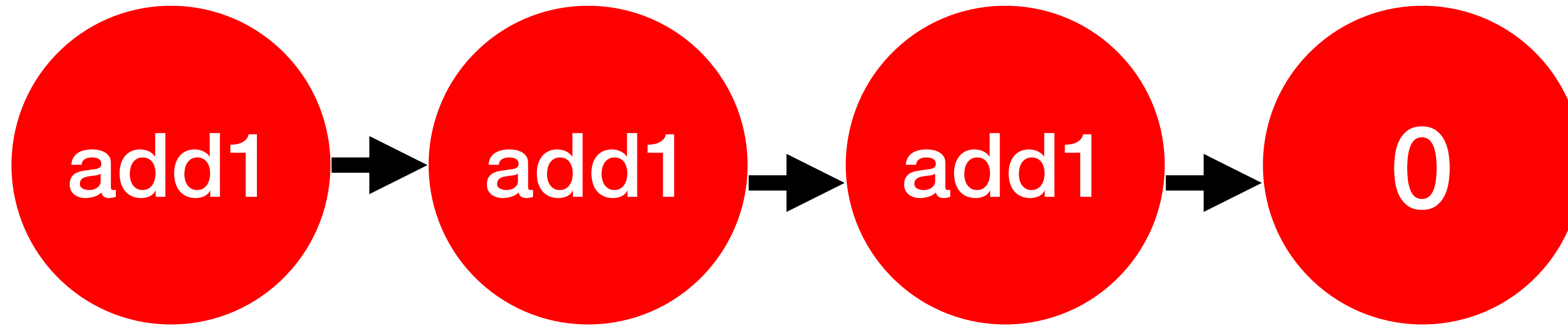
**successor**





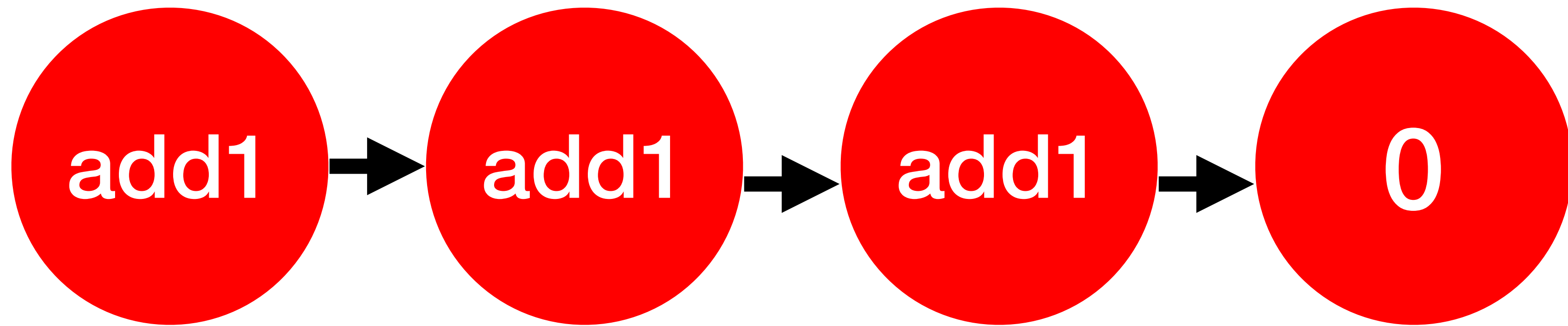
( add1  
  ( add1  
    ( add1 0 ) ) )





( add1 ( add1 ( add1 0 ) ) )





3



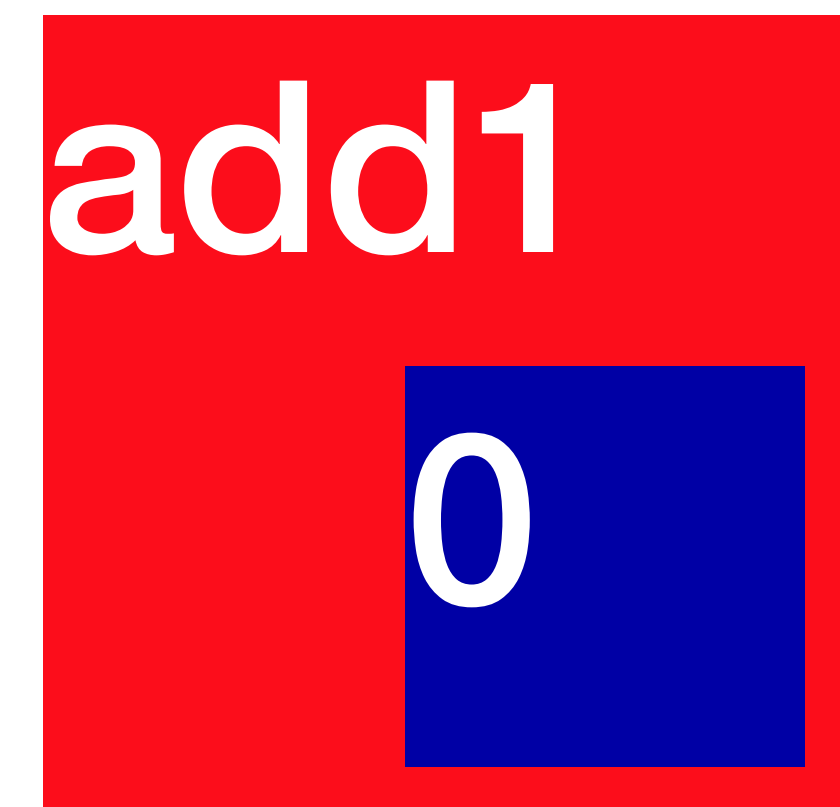
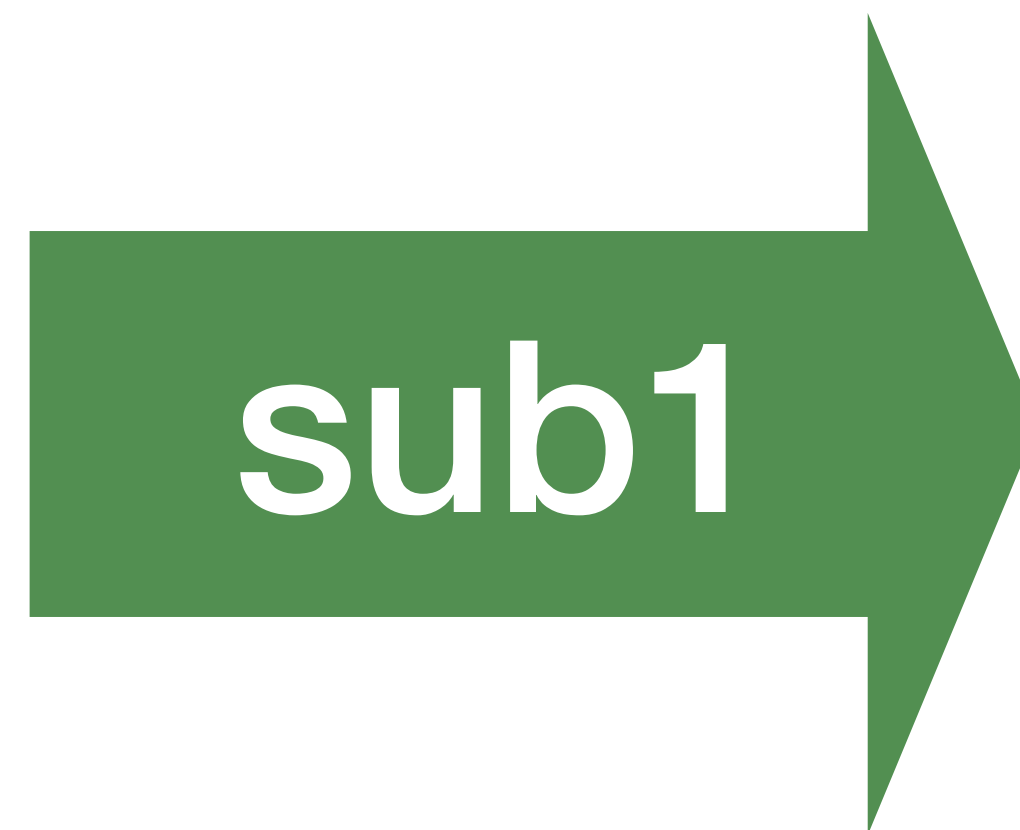
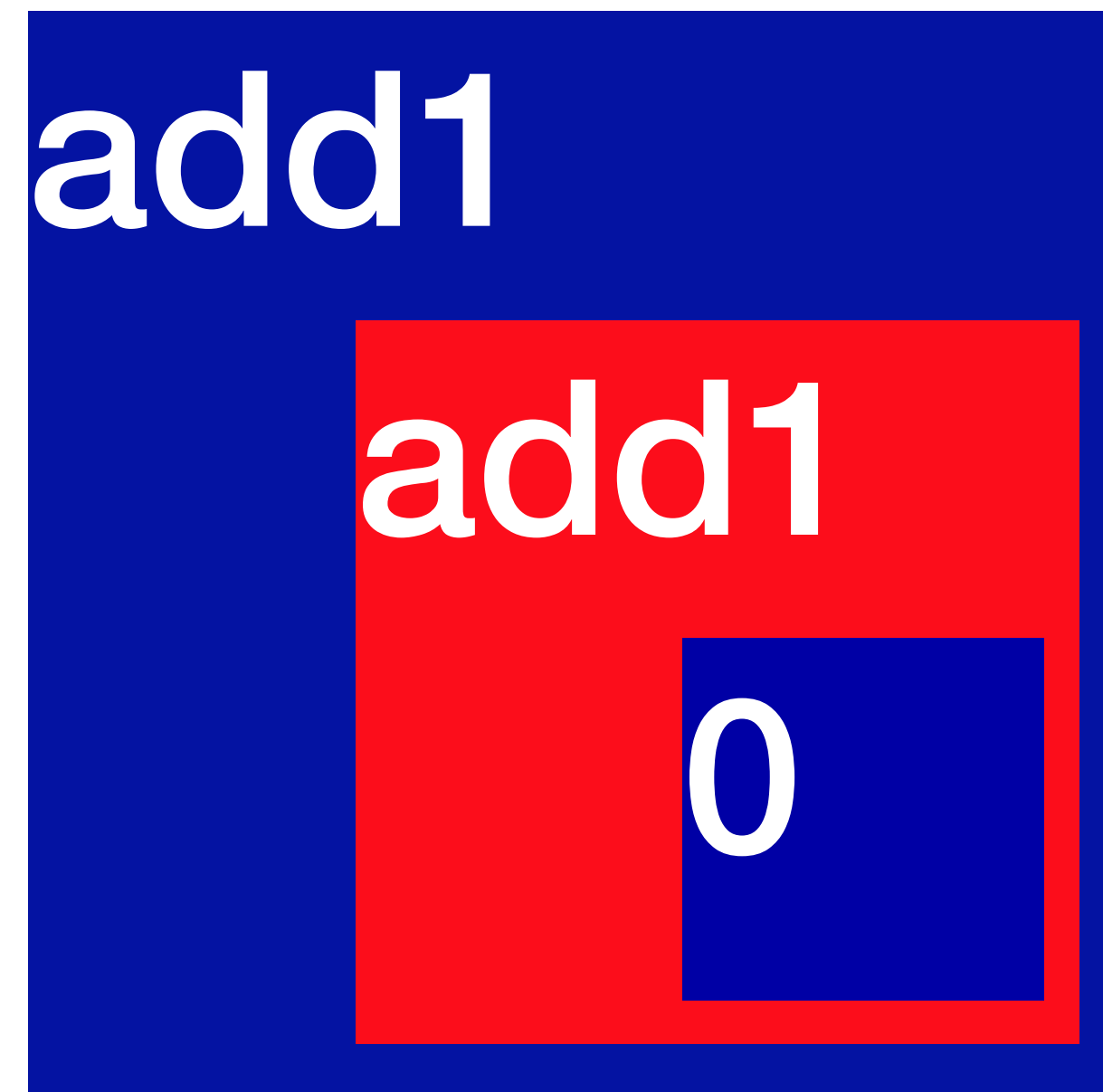
**selectors**

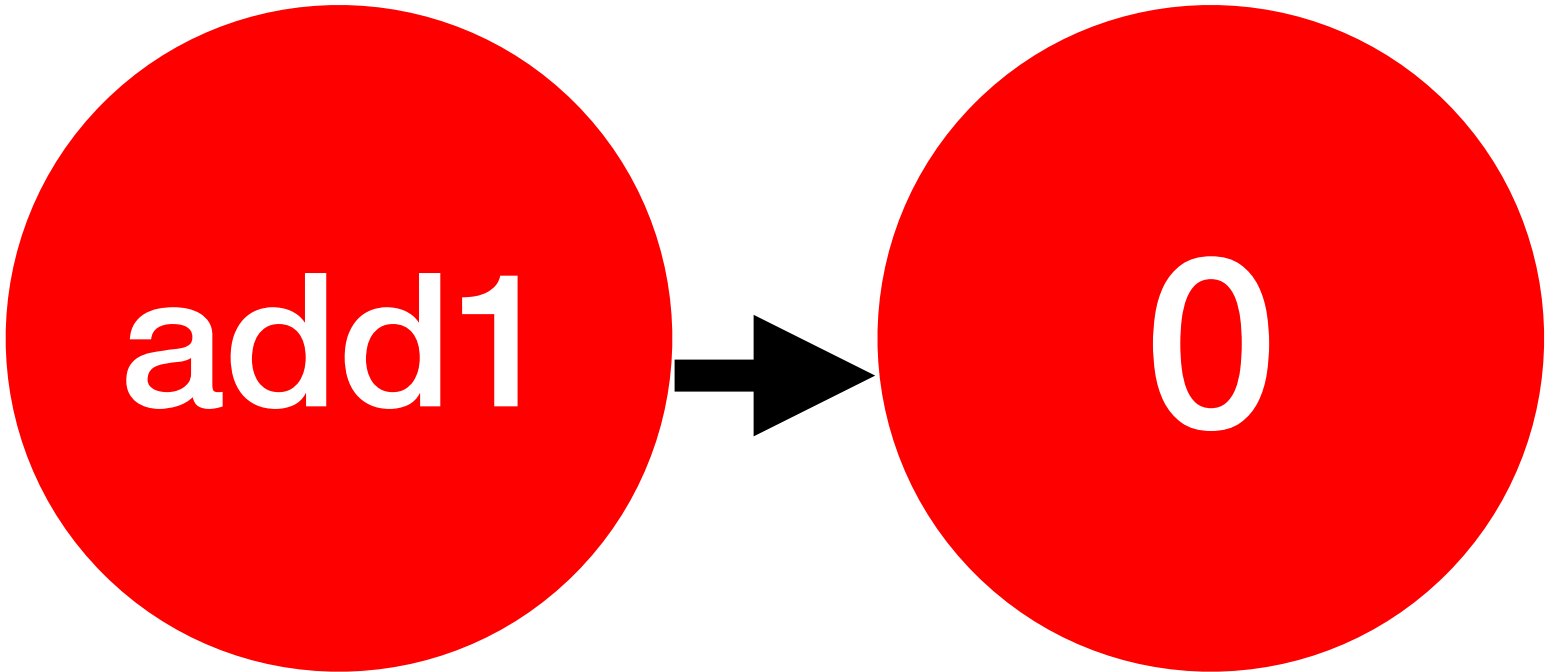
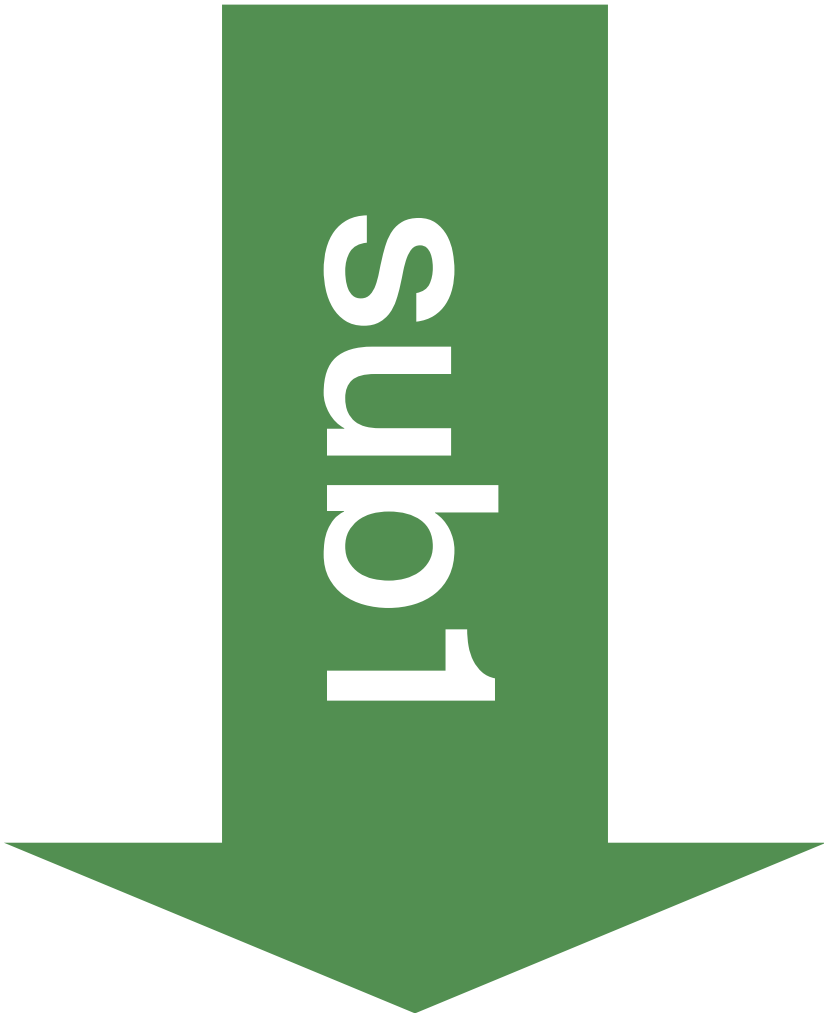
*; nat - selectors*  
*; - sub1 : nat -> nat*





**sub1 : nat -> nat**





```
( check-expect  
  ( sub1  
    ( add1  
      ( add1 0 ) ) ) )  
( add1 0 ) )
```



```
( check-expect  
  ( sub1 2 )  
  1 )
```



*; nat - predicates*  
*; - zero? : nat -> boolean*



**zero? : nat -> boolean**

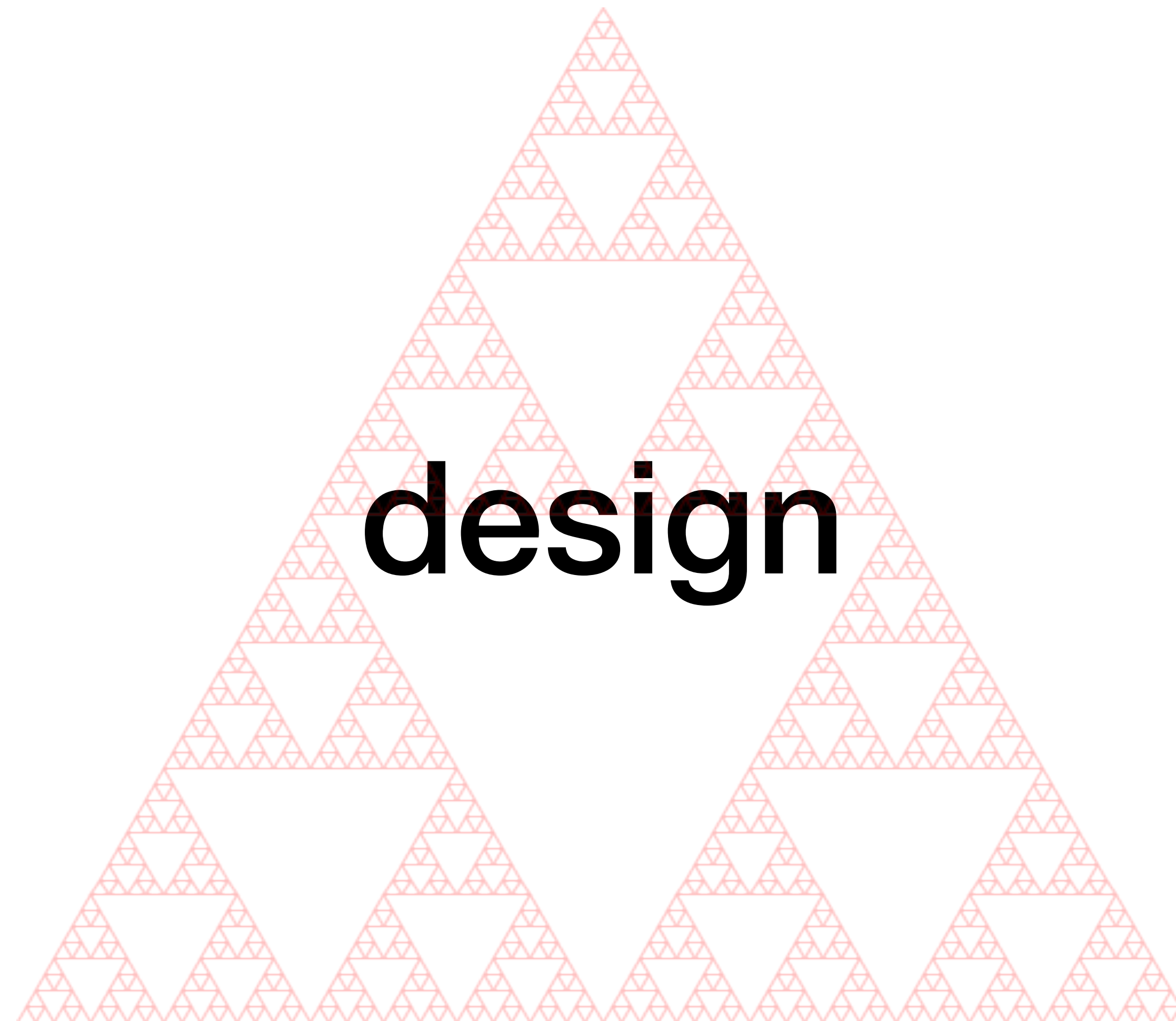
( check-satisfied  
0  
zero? )





```
(define topics  
  (make-agenda  
    ('("Recursive Data Type"  
       "Structural Recursion"  
       "General Recursion"))))
```





```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```





```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



**given two natural numbers combine  
them to yield a natural number  
which is their total size**

```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



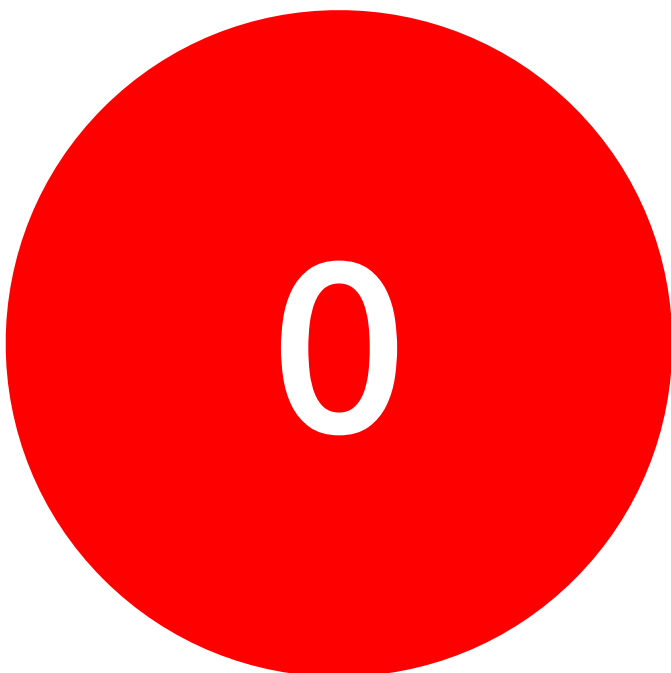
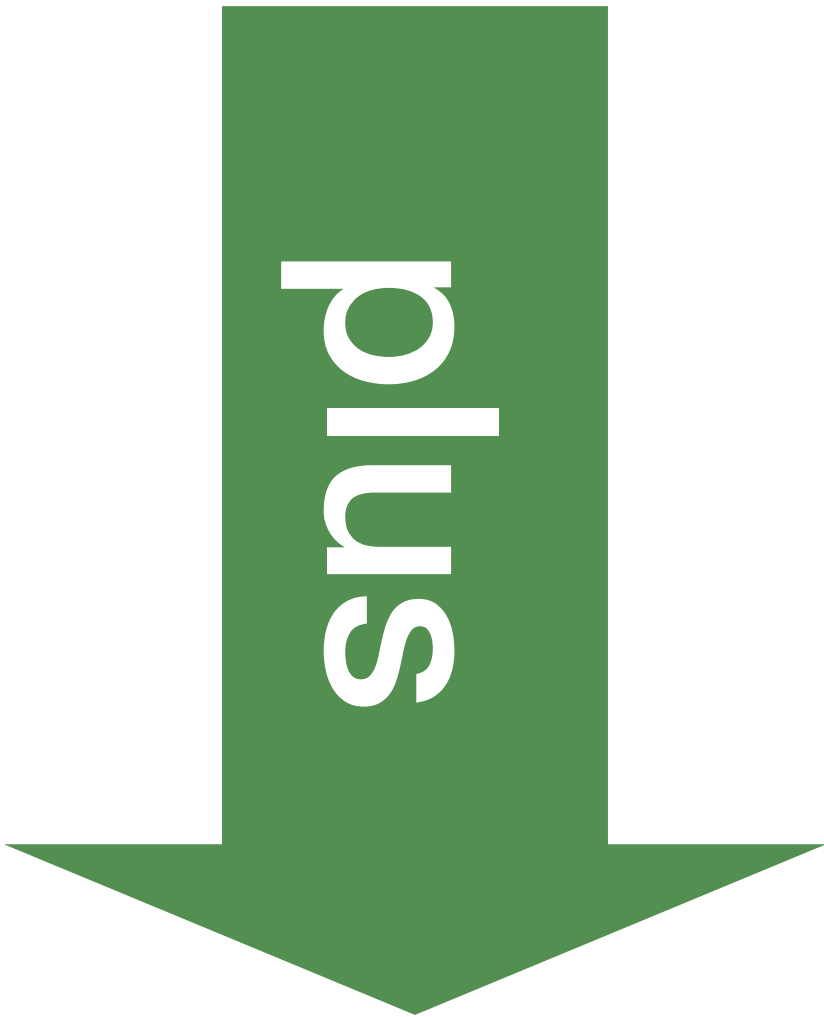
*; plus : nat -> nat -> nat*

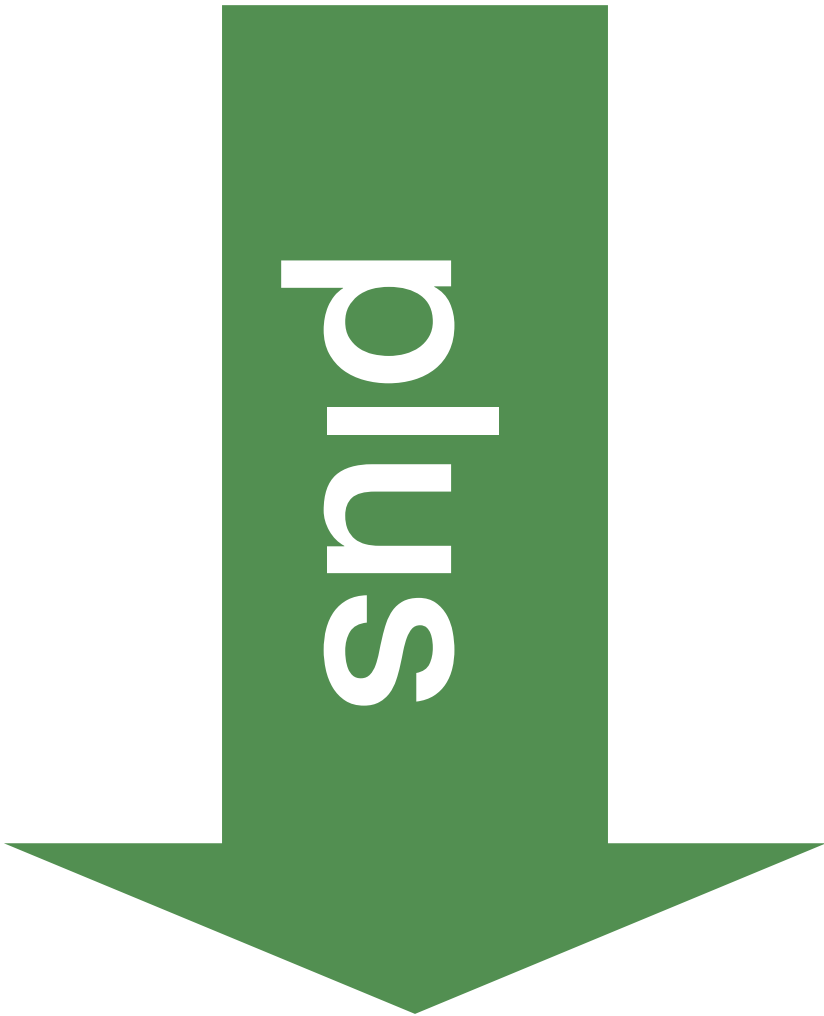
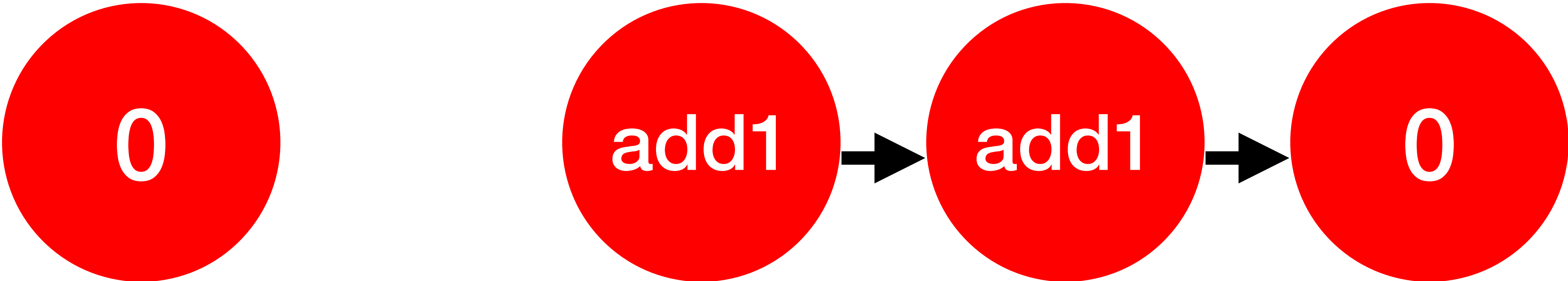


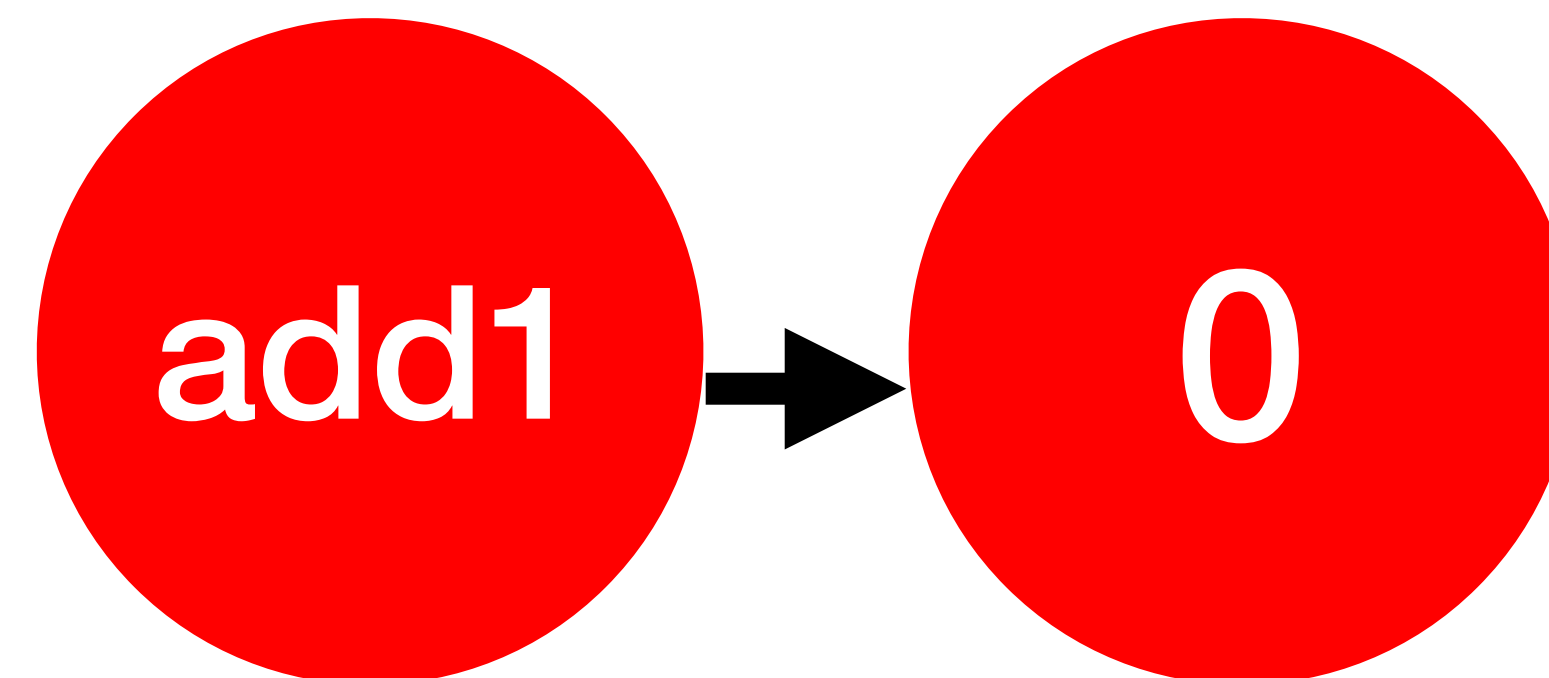
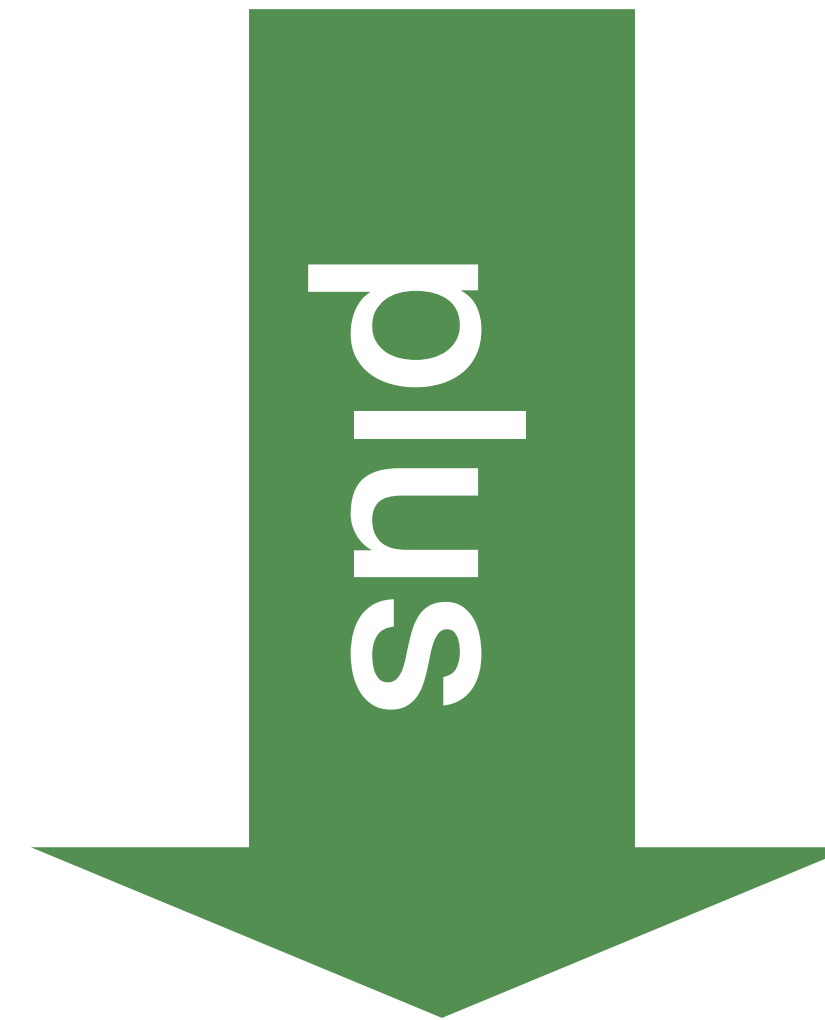
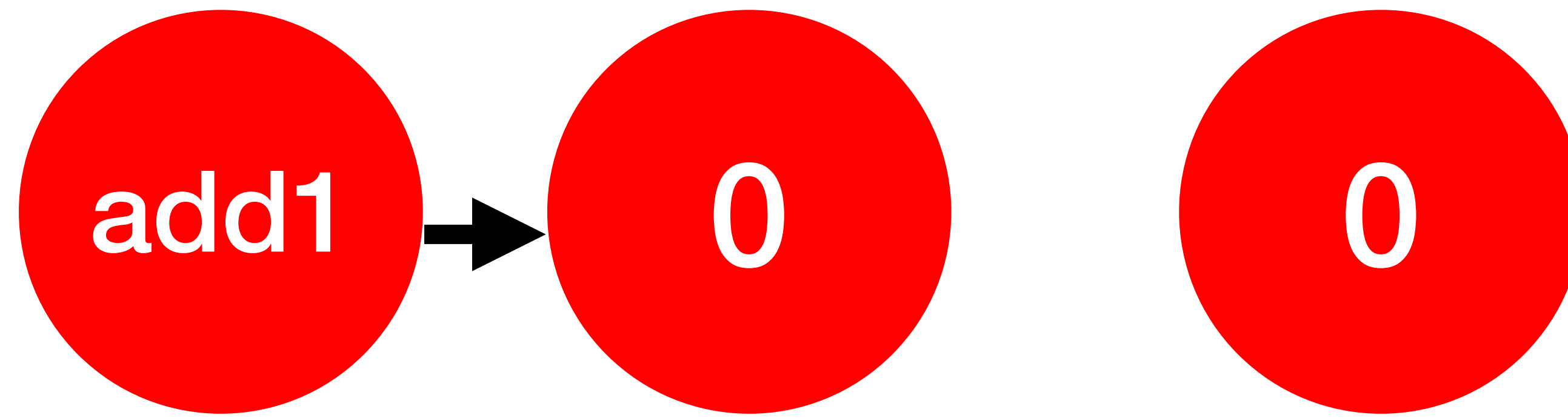


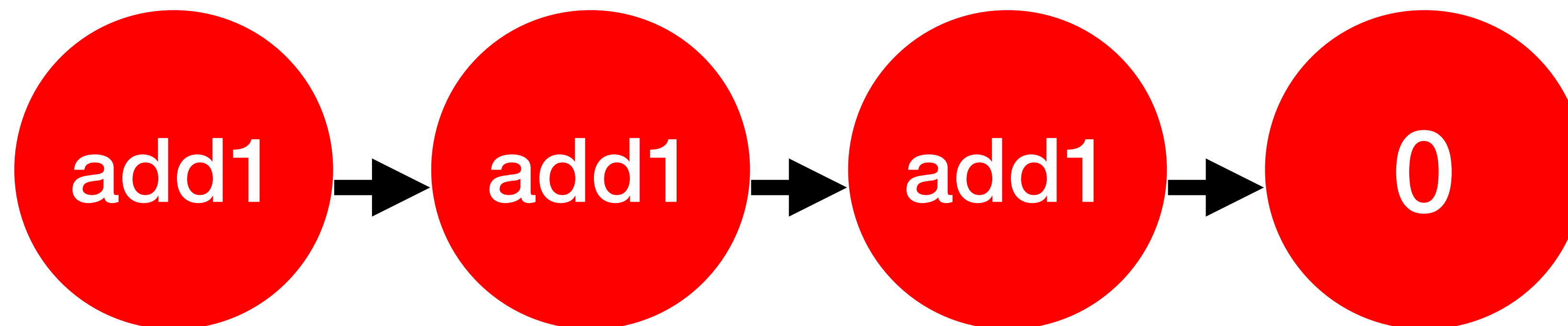
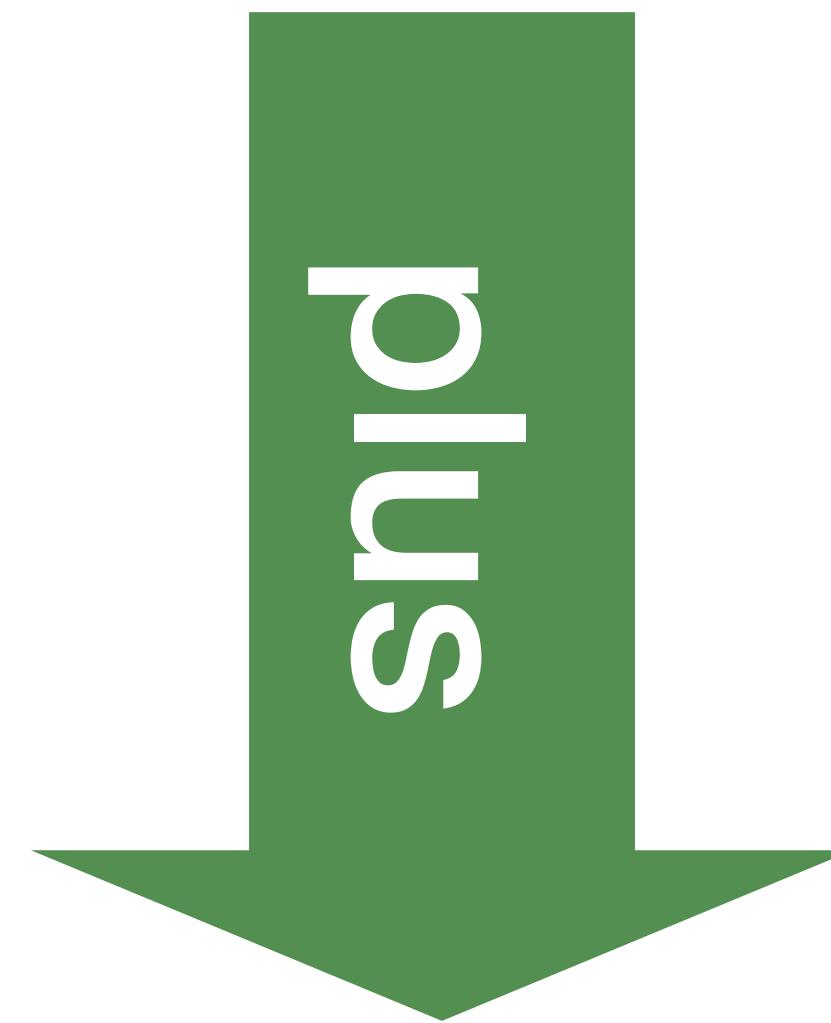
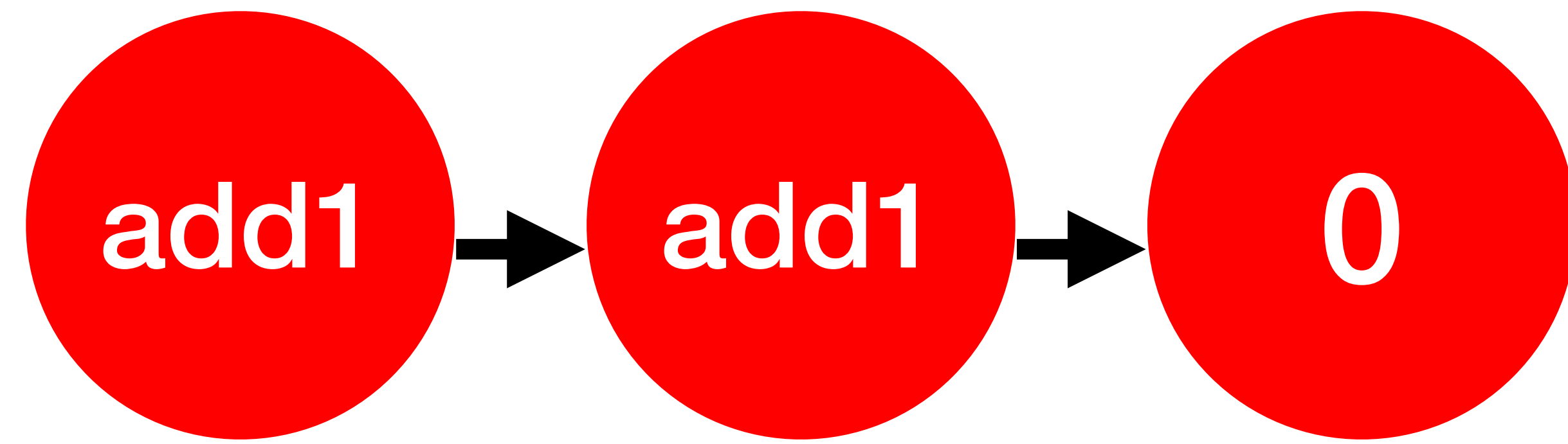
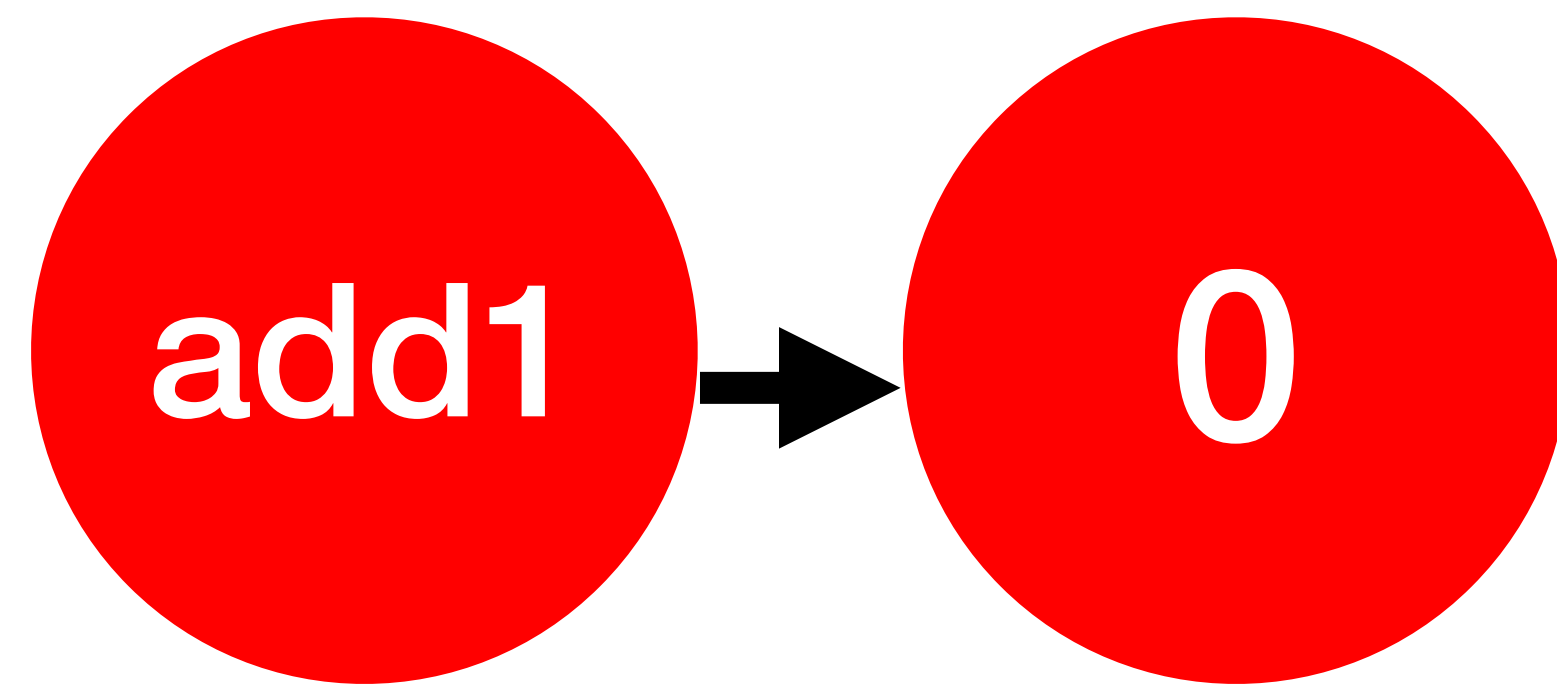
```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```











*; (plus 0 0) ; 0*  
*; (plus 0 b) ; b*  
*; (plus a 0) ; a*  
*; (plus a b) ; a + b*



```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



```
(define (plus a b)
  (cond
    [(zero? a) b]
    [else
     (add1
      (plus (sub1 a)
            b))]))
```





```
(define recursive-process  
  ('("identify principal"  
     "test basis"  
     "reduced recursion"  
     "combine results")))
```



**plus : (a : nat) -> nat -> nat**

principal	a : nat
basis	zero?
reducer	sub1
combine	add1

```
(define (plus a b)
  (cond
    [(zero? a) b]
    [else
     (add1
      (plus (sub1 a)
            b))]))
```



```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



*; (plus 0 0) ; 0*

(check-satisfied  
  (plus 0 0)  
  zero?)



*; (plus 0 b) ; b*

(check-expect  
  (plus 0 2)  
  2)



*; (plus a 0) ; a*

(check-expect  
  (plus 1 0)  
  1)



*; (plus a b) ; a + b*

(check-expect  
  (plus 1 2)  
  3)





*; (plus a b) ; a + b*

```
( check-expect
  ( plus
    ( add1 0 )
    ( add1 ( add1 0 ) ) )
  ( plus
    0
    ( add1 ( add1 ( add1 0 ) ) ) ) )
```





```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



**given a natural number  $n$  and a list  
yield a list with the first  $n$  elements**

```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```

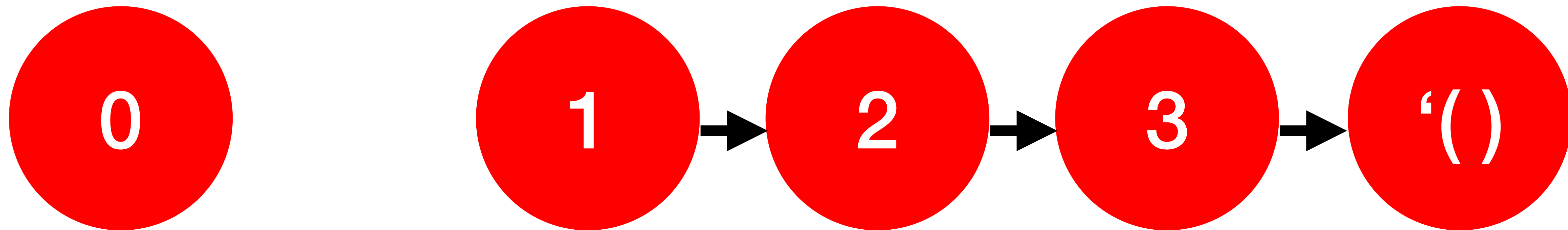


*; take : nat -> list -> list*



```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```

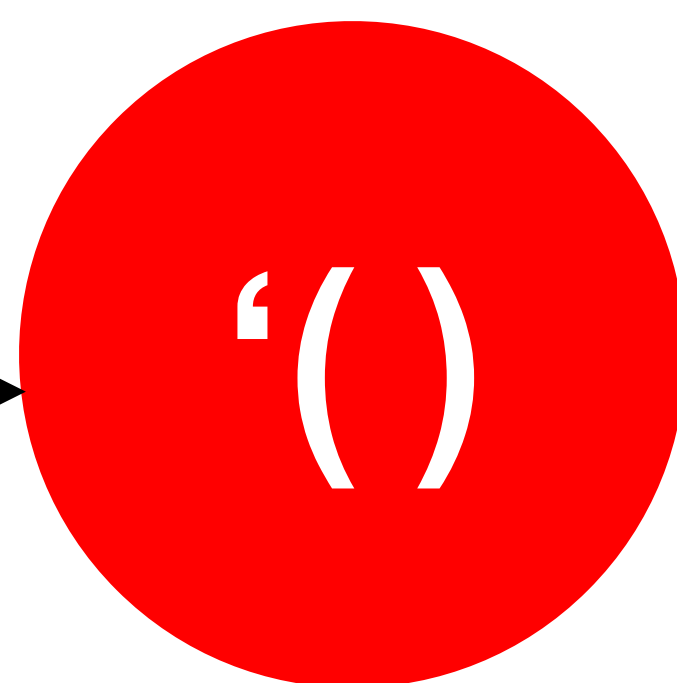
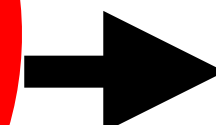
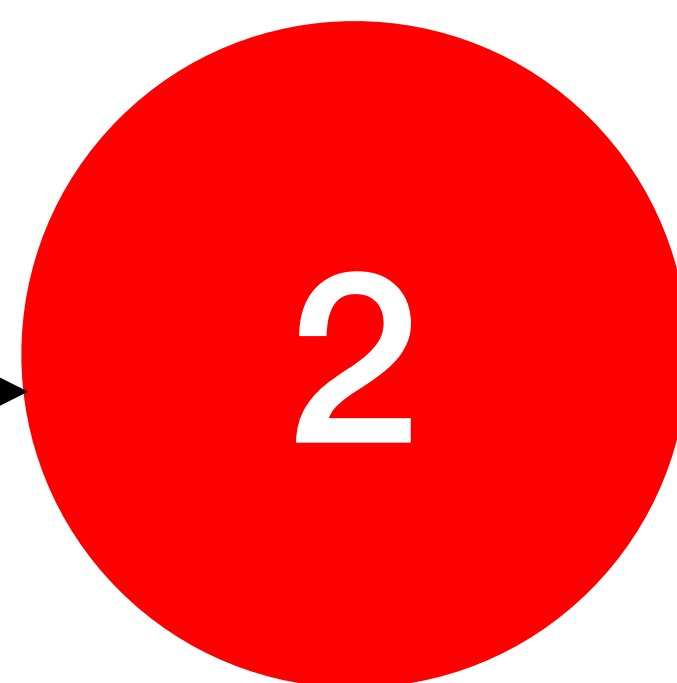
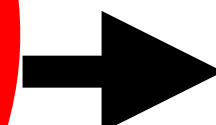
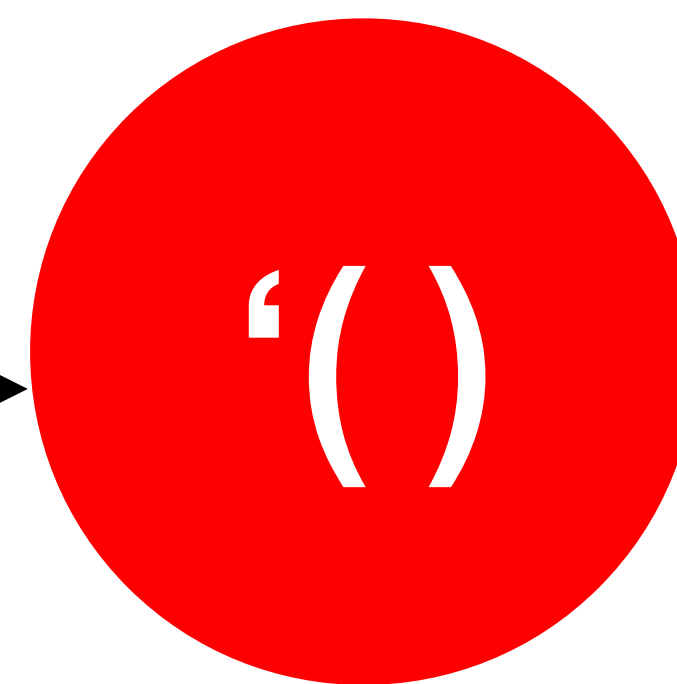
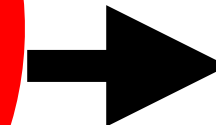
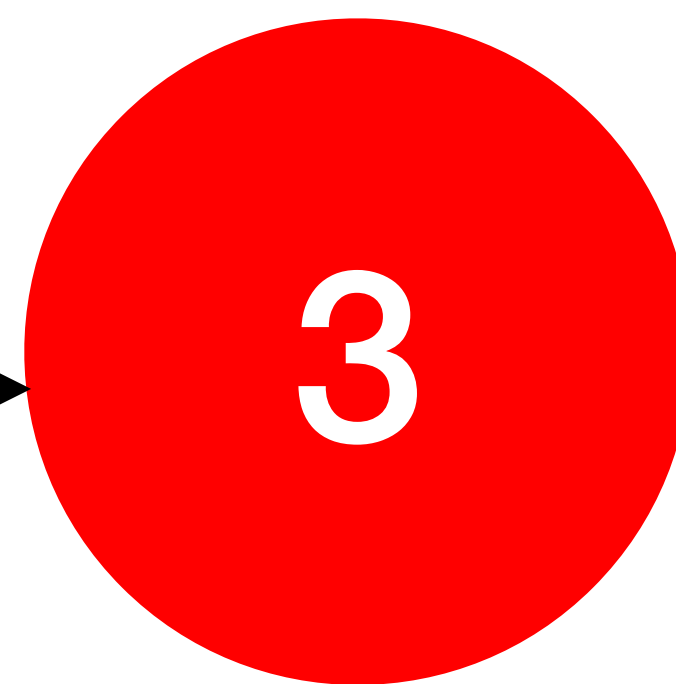
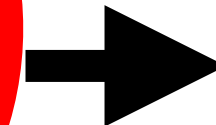
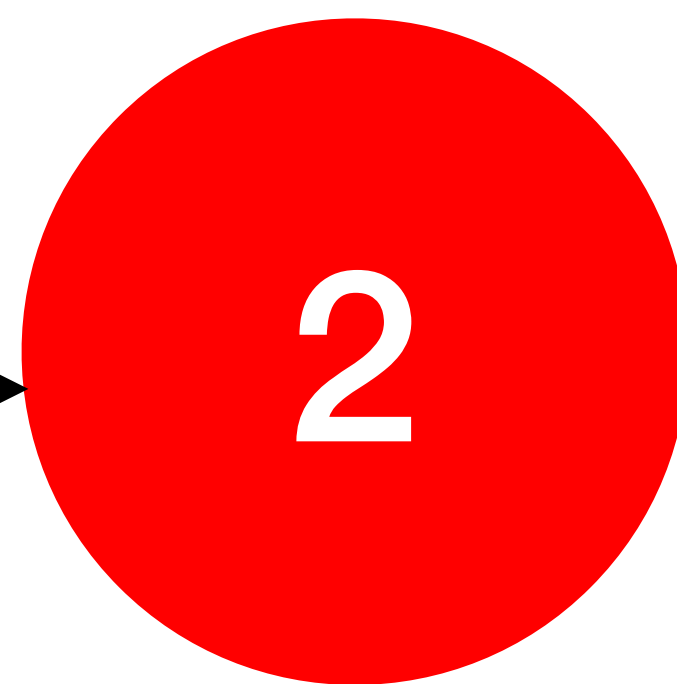
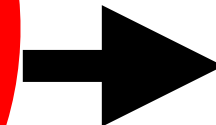
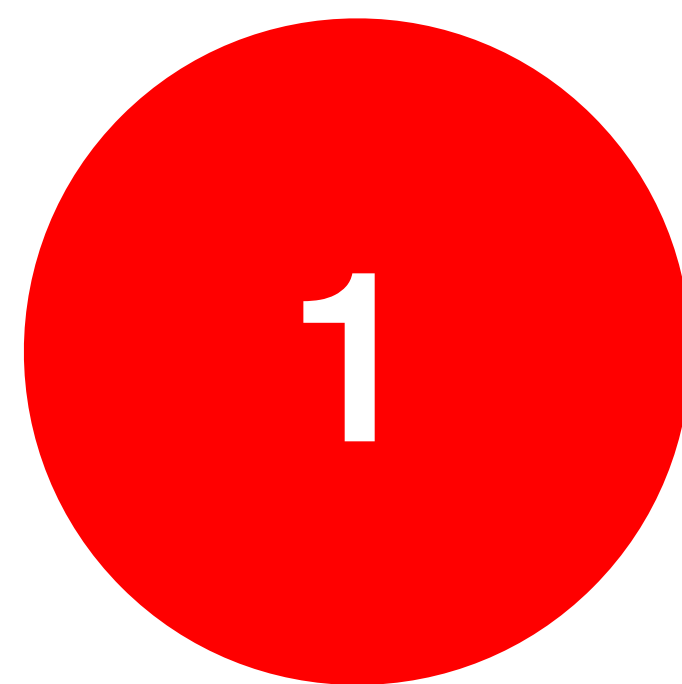


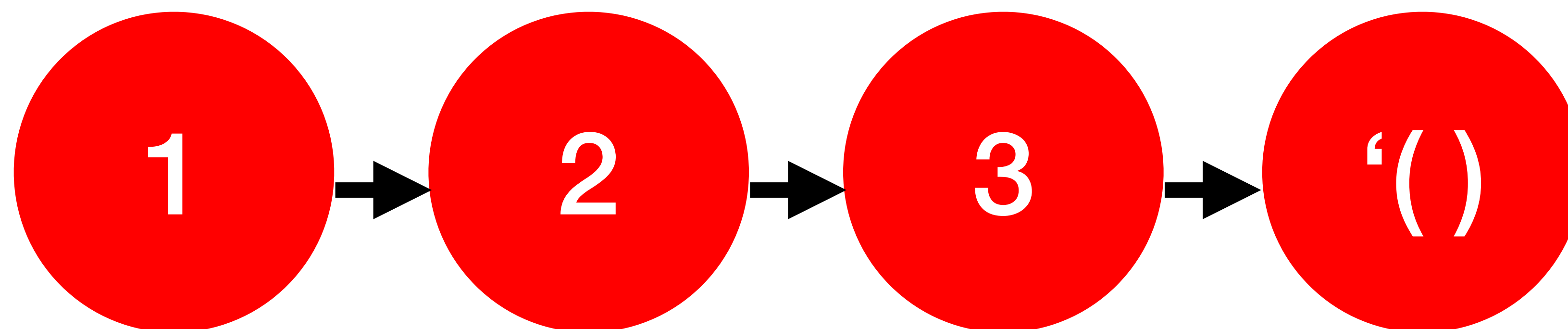
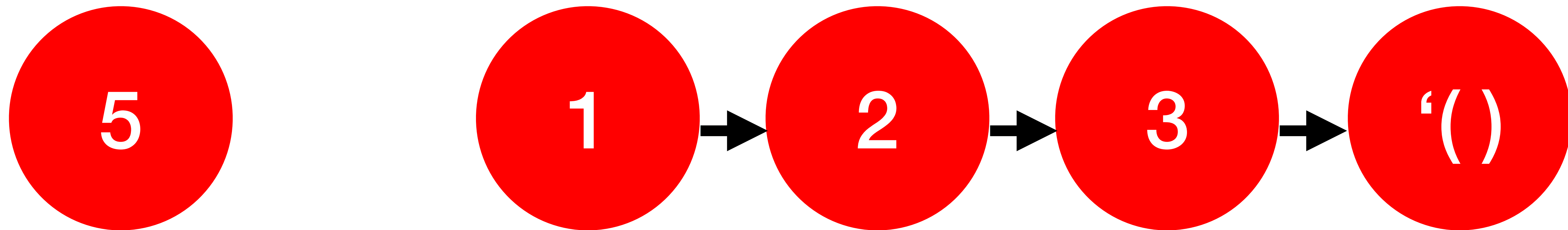












*; (take 0 l) ; '()*  
*; (take 0 '()) ; '()*  
*; (take n '()) ; '()*  
*; (take n l of size > n) ; l of size n*  
*; (take n l of size < n) ; l*



```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



```
(define (take n l)
  (cond
    [(zero? n) '()]
    [(empty? l) '()]
    [else
     (cons (first l)
           (take (sub1 n)
                 (rest l)))])
```



```
(define recursive-process  
  ('("identify principal"  
     "test basis"  
     "reduced recursion"  
     "combine results")))
```





**take : (n : nat) -> (l : list) -> list**

principal	n : nat	l : list
basis	zero?	empty?
reducer	sub1	rest
combine		cons

```
(define (take n l)
  (cond
    [(zero? n) '()]
    [(empty? l) '()]
    [else
     (cons (first l)
           (take (sub1 n)
                 (rest l)))])
  )
```



```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



*; (take 0 1) ; '()*

(check-satisfied  
  (take 0 '(1 2 3))  
  empty?)



*; (take 0 '()) ; '()*

(check-satisfied  
  (take 0 '())  
  empty?)



*; (take n '()) ; '()*

(check-satisfied  
  (take 2 '())  
  empty?)



*; (take n l of size > n) ; l of size n*

(check-expect  
 (take 2 '(1 2 3))  
 '(1 2))



*; (take n l of size > n) ; l of size n*

```
(check-expect
  (take
    (add1 (add1 0))
    (cons 1 (cons 2 (cons 3 ' ( ) ) ) ) )
  (cons 1 (cons 2 ' ( ) ) ) )
```





*; (take n l of size > n) ; l of size n*

(check-expect

(take

(add1 (add1 0))

'(1 2 3))

(cons (first '(1 2 3))

(cons (first '(2 3))

'( )))



*; (take n l of size < n) ; l*

(check-expect  
 (take 5 '(1 2 3))  
 '(1 2 3))





```
(define (take n l)
  (cond
    [(zero? n) '()]
    [(empty? l) '()]
    [else
     (cons (first l)
           (take (sub1 n)
                 (rest l)))])
```





drop

```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



**given a natural number  $n$  and a list  
yield a list without the first  $n$   
elements**

```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



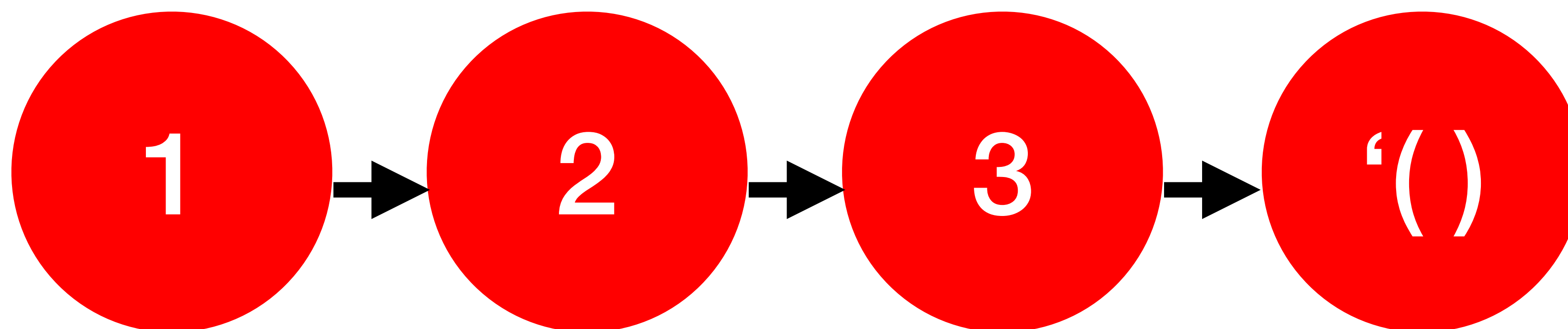
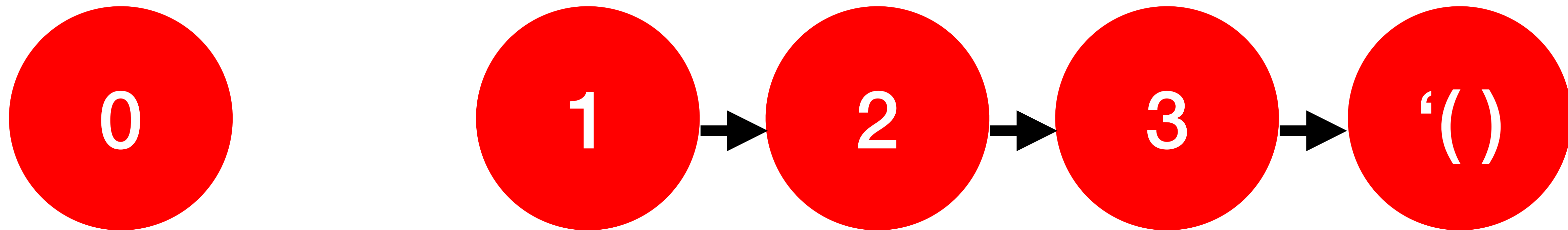


*; drop : nat -> list -> list*



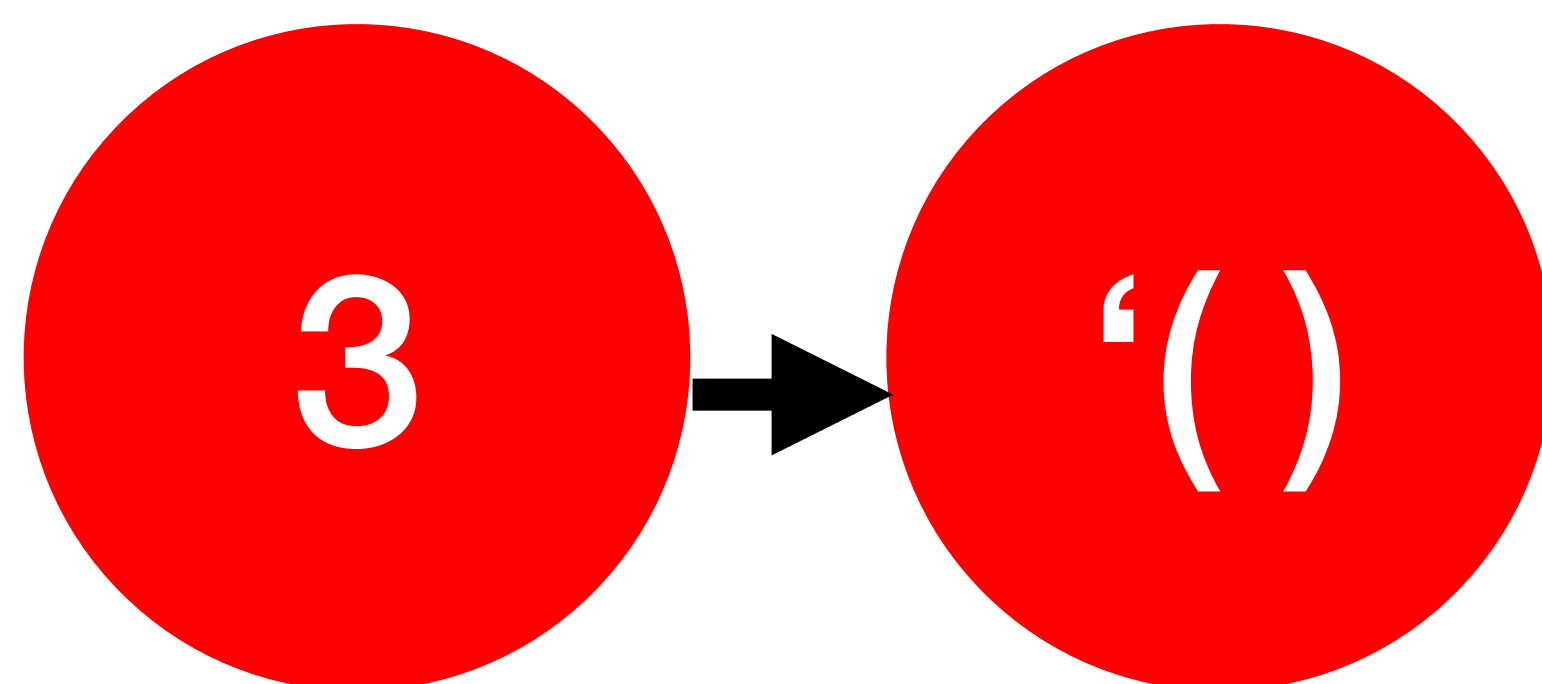
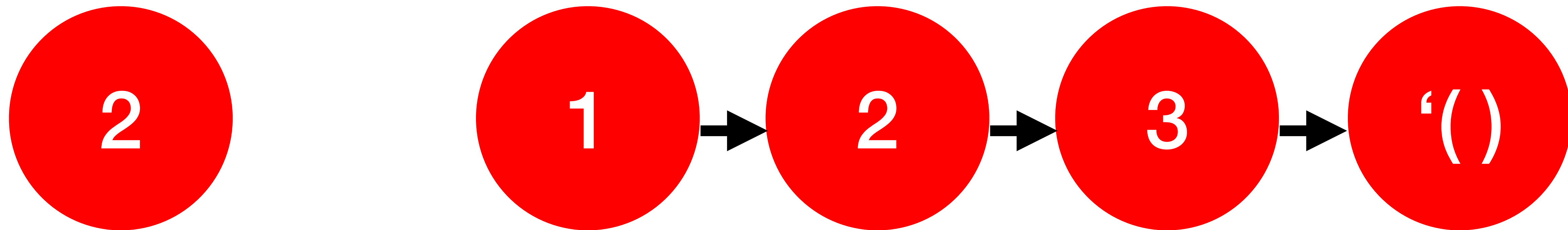
```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```

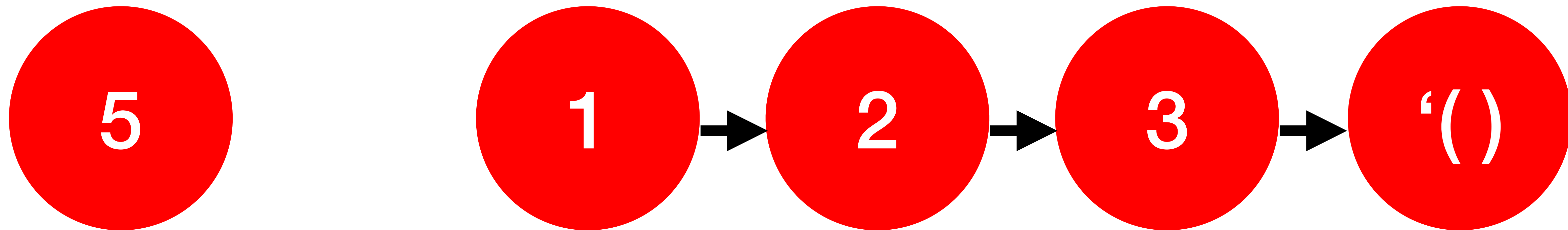












*; (drop 0 l) ; l*  
*; (drop 0 ' ( )) ; ' ( )*  
*; (drop n ' ( )) ; ' ( )*  
*; (drop n l of size n + m) ; l of size m*  
*; (drop n l of size < n) ; ' ( )*





```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



```
(define (drop n l)
  (cond
    [(zero? n) l]
    [(empty? l) '()]
    [else
     (drop (sub1 n)
           (rest l))]))
```



```
(define recursive-process  
  ('("identify principal"  
     "test basis"  
     "reduced recursion"  
     "combine results")))
```



**drop : (n : nat) -> (l : list) -> list**

principal	n : nat	l : list
basis	zero?	empty?
reducer	sub1	rest
combine		

```
(define (drop n l)
  (cond
    [(zero? n) l]
    [(empty? l) '()]
    [else
     (drop (sub1 n)
           (rest l))]))
```



```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



*; (drop 0 1) ; 1*

(check-expect  
 (drop 0 '(1 2 3))  
 '(1 2 3))



*; (drop 0 ' ( ) ) ; ' ( )*

( check-satisfied  
 ( drop 0 ' ( ) )  
 empty? )





*; (drop n '()) ; '()*

(check-satisfied  
 (drop 2 '())  
 empty?)



*; (drop n l of size n + m) ; l of size m*

(check-expect  
 (drop 2 '(1 2 3))  
 '(3))



*; (drop n l of size < n) ; ' ( )*

( check-satisfied  
 ( drop 5 ' ( 1 2 3 ) )  
 empty? )





```
(define (drop n l)
  (cond
    [(zero? n) l]
    [(empty? l) '()]
    [else
     (drop (sub1 n)
           (rest l))]))
```



**take vs. drop**

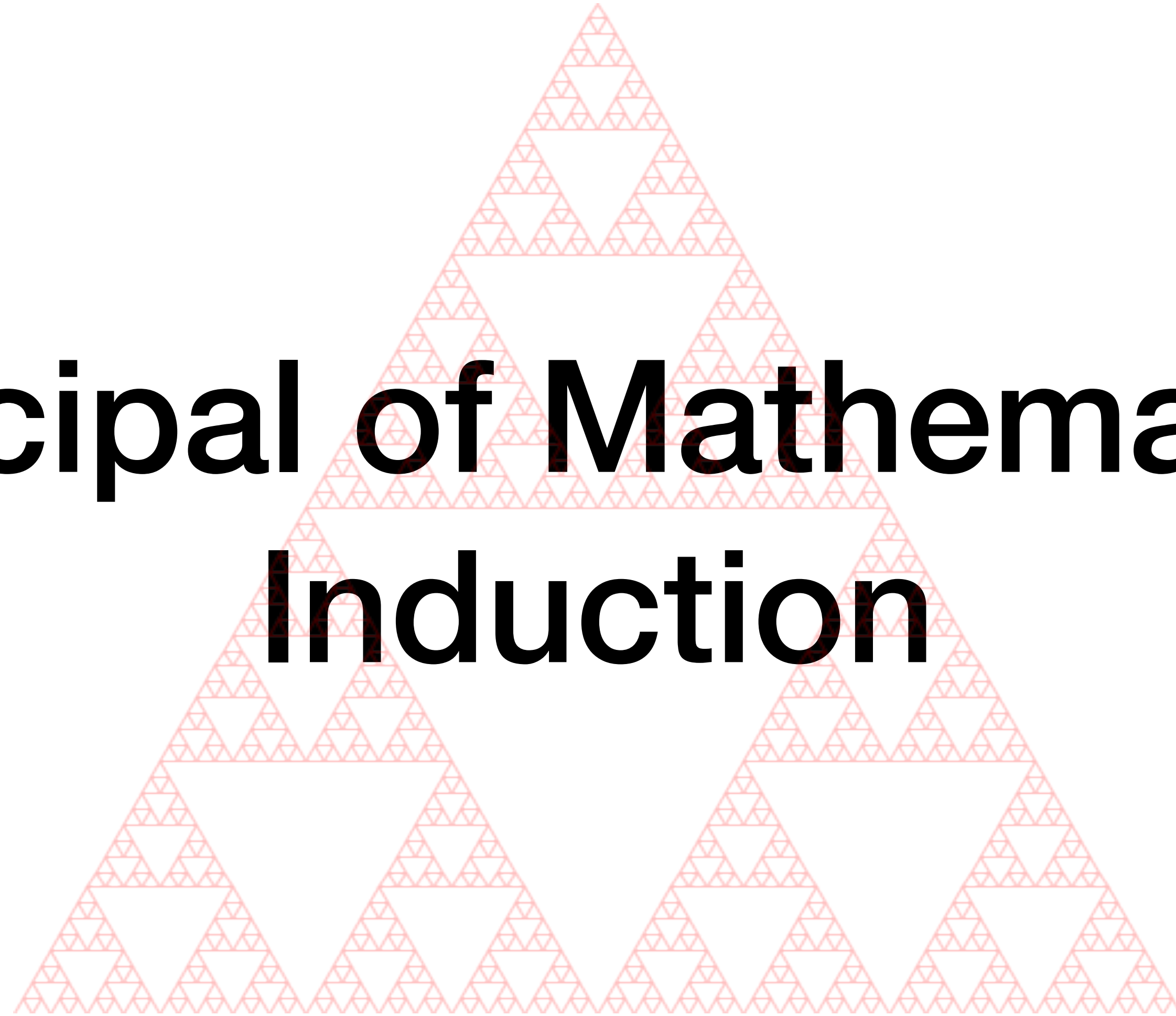


```
(define (take n l)
  (cond
    [(zero? n) '()]
    [(empty? l) '()]
    [else
     (cons (first l)
           (take (sub1 n)
                 (rest l)))])])
```

```
(define (drop n l)
  (cond
    [(zero? n) l]
    [(empty? l) '()]
    [else
     ; combine-solutions
     (drop (sub1 n)
           (rest l))]))
```



# Principal of Mathematical Induction

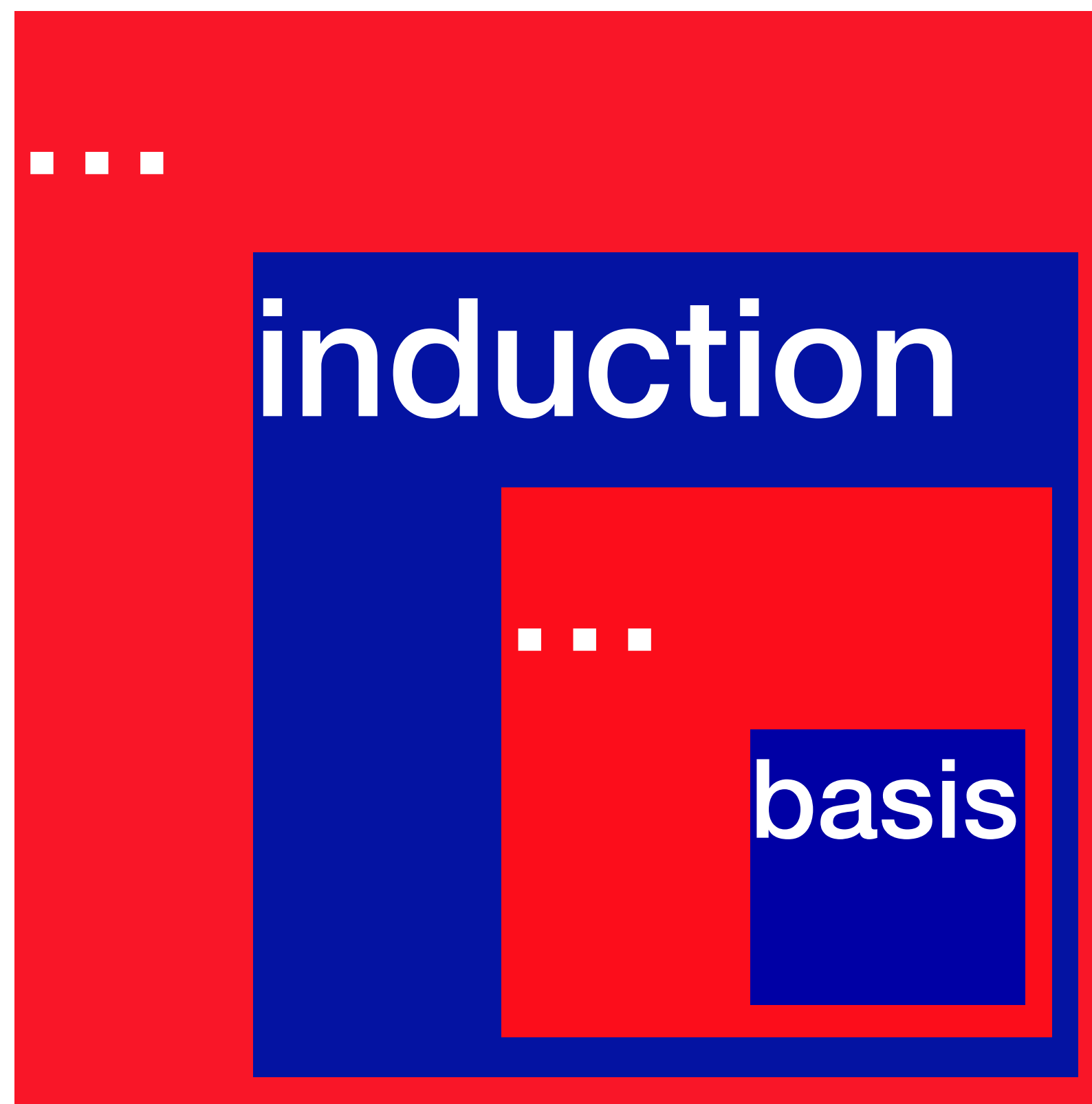


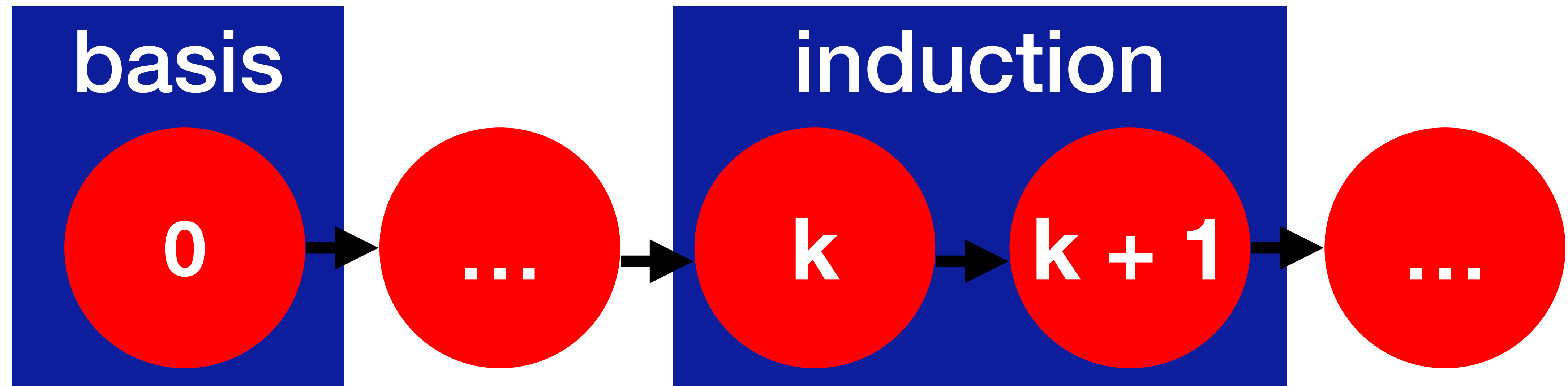


**P** holds for all values 0, 1, 2, ... if

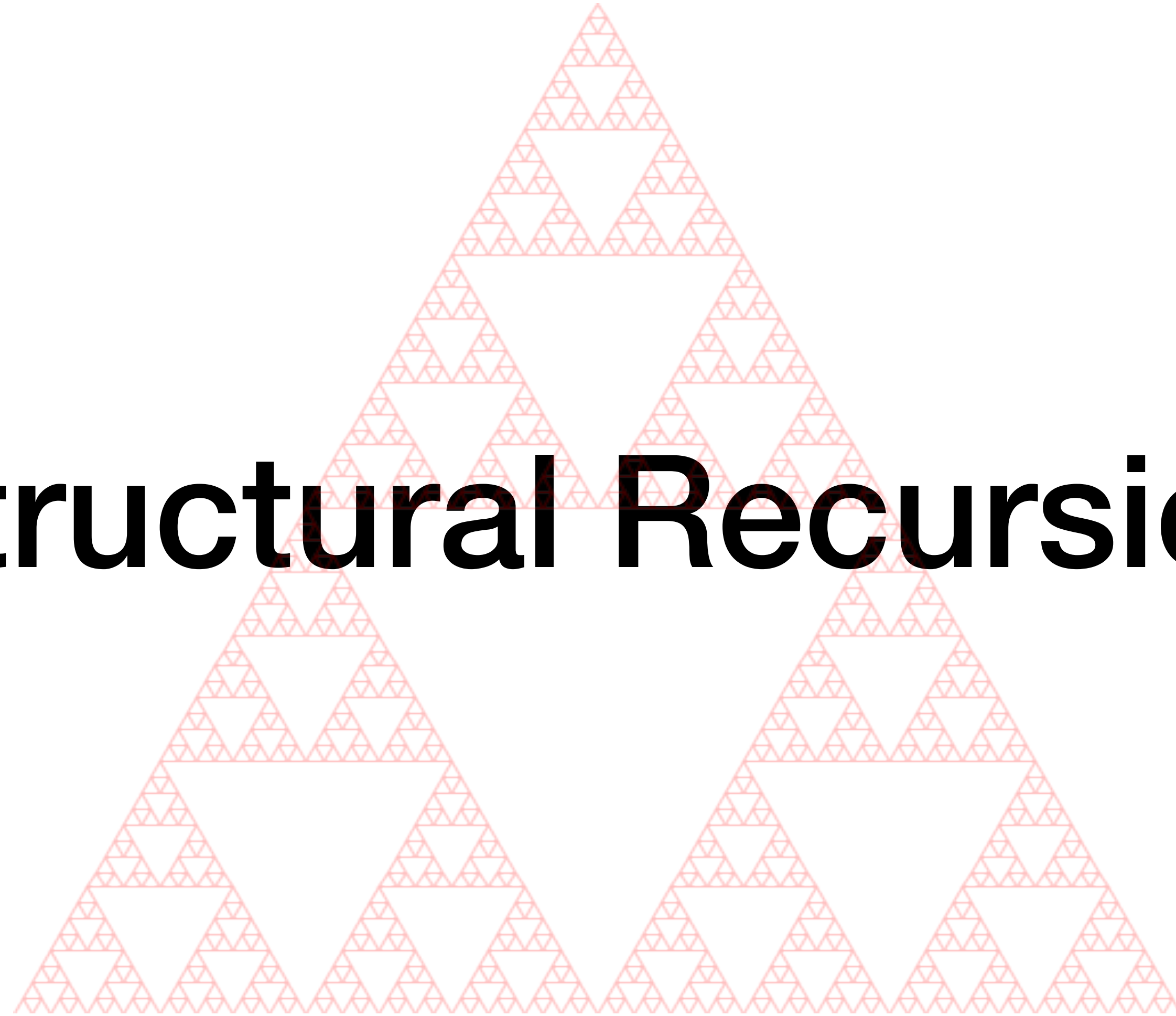
**basis step:** **P**(0) is true

**induction step:** **P**(k) is true  $\Rightarrow$  **P**(k + 1) is also true





# Structural Recursion



```
(define (structural P)
  (cond
    [(basis? P) (solve P)]
    [else
     (combine-solutions
      P
      (structural
       (induction P)))])
```



**take vs. drop**



```
(define (take n l)
  (cond
    [(zero? n) '()]
    [(empty? l) '()]
    [else
     (cons (first l)
           (take (sub1 n)
                 (rest l)))])])
```

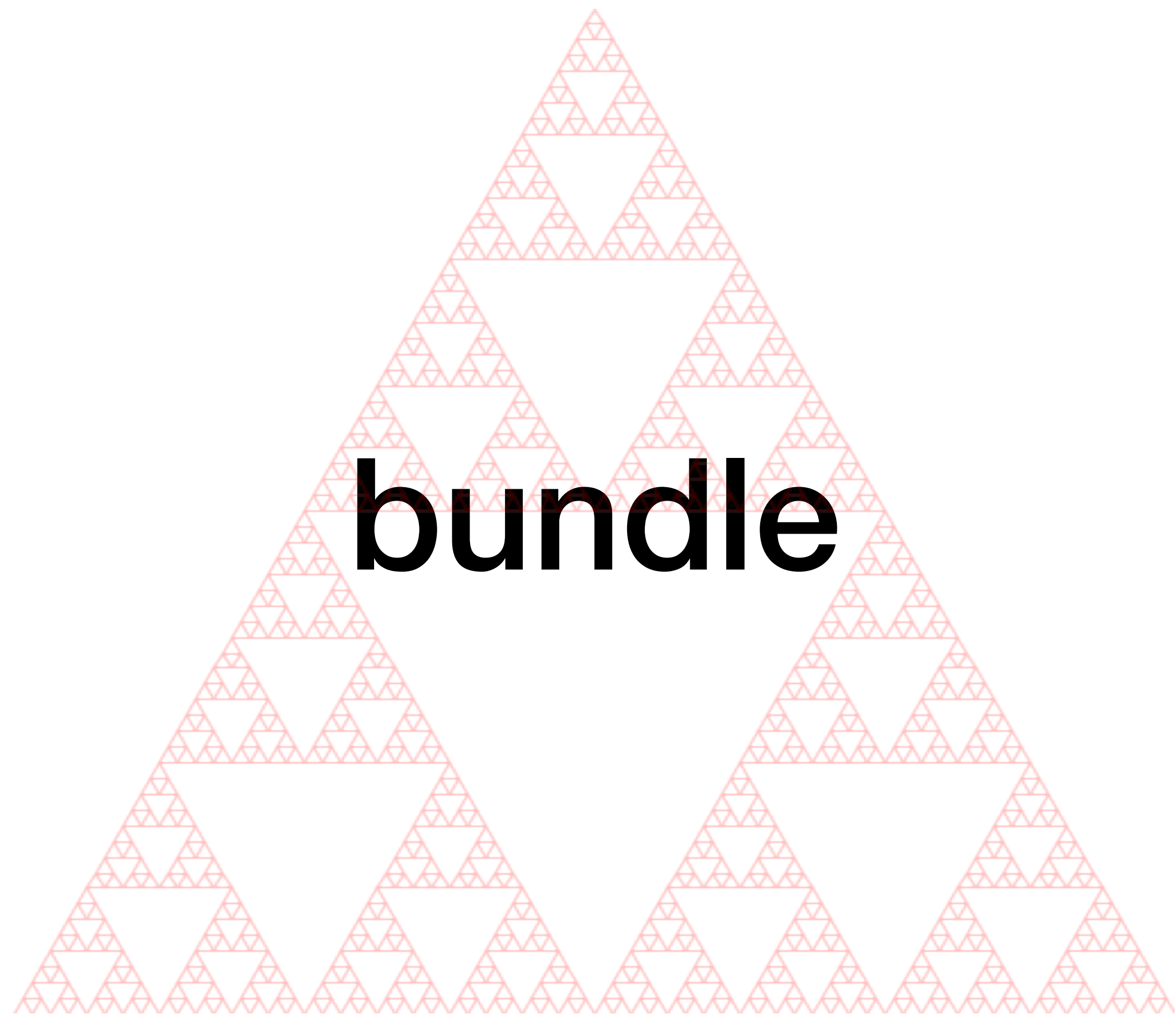
```
(define (drop n l)
  (cond
    [(zero? n) l]
    [(empty? l) '()]
    [else
     ; combine-solutions
     (drop (sub1 n)
           (rest l))]))
```



```
(define topics  
  (make-agenda  
    ('("Recursive Data Type"  
       "Structural Recursion"  
       "General Recursion"))))
```







**bundle**

```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



**given a natural number  $n$  and a list  
yield a list of lists each with each  
list containing at least  $n$  elements**

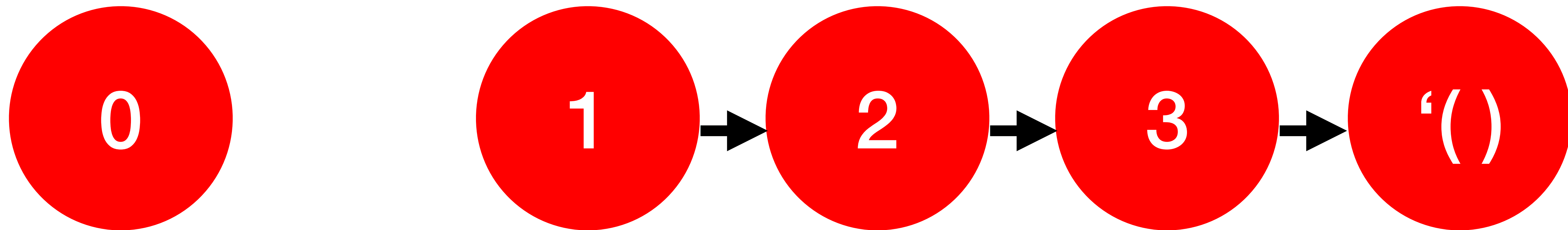
```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



*; bundle : nat -> list -> list*

```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```

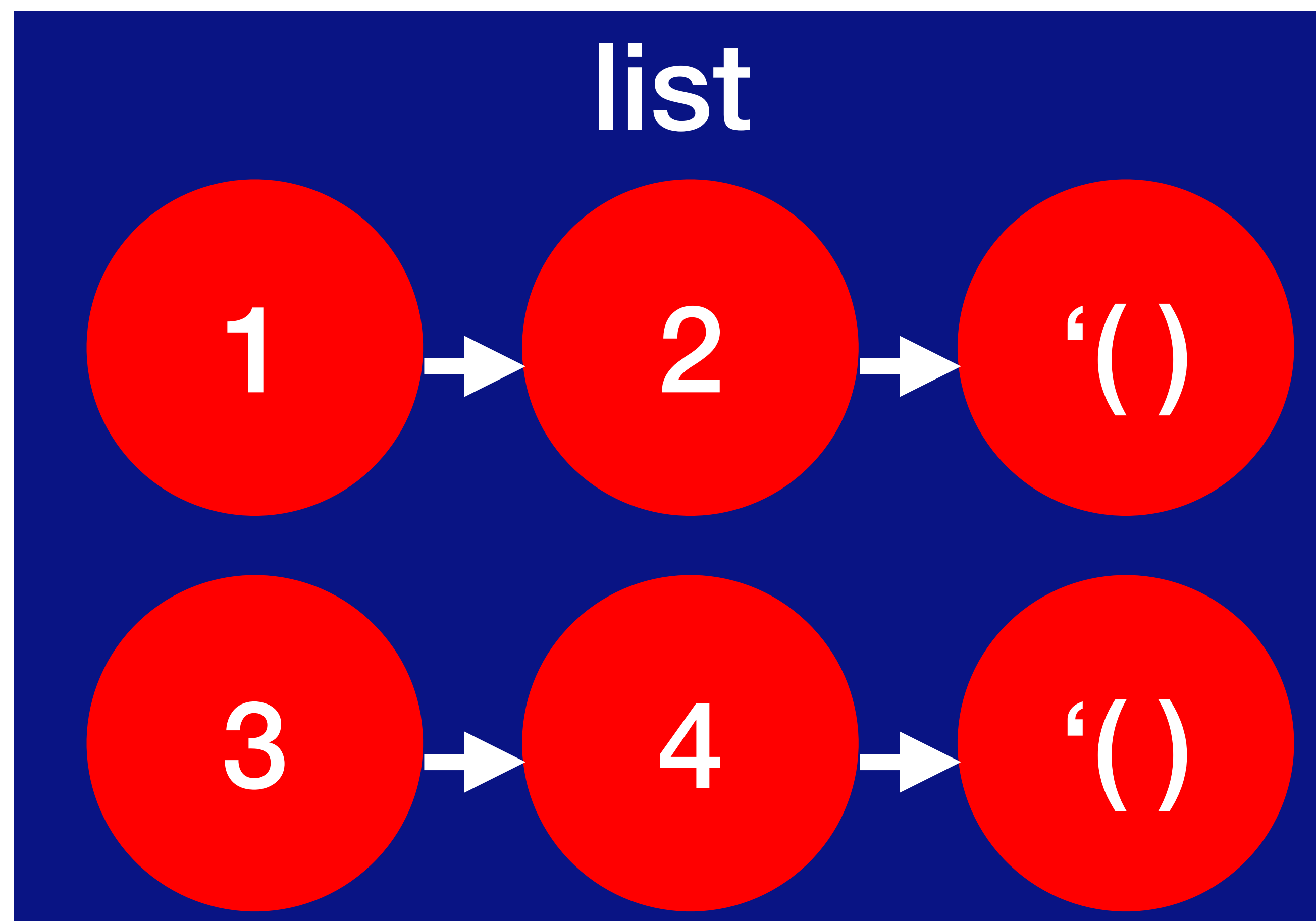
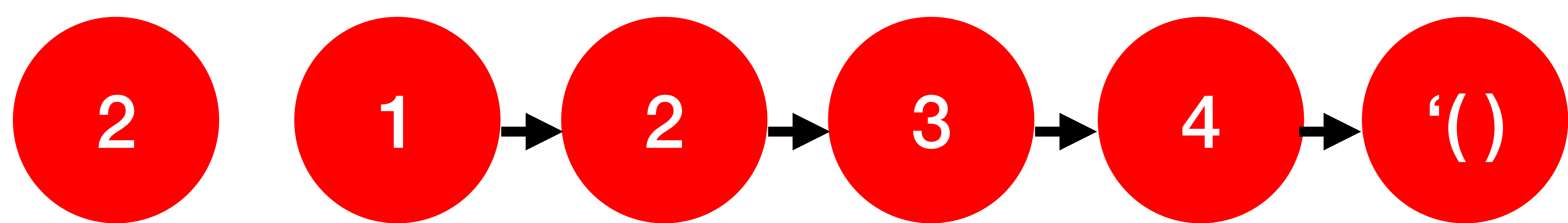




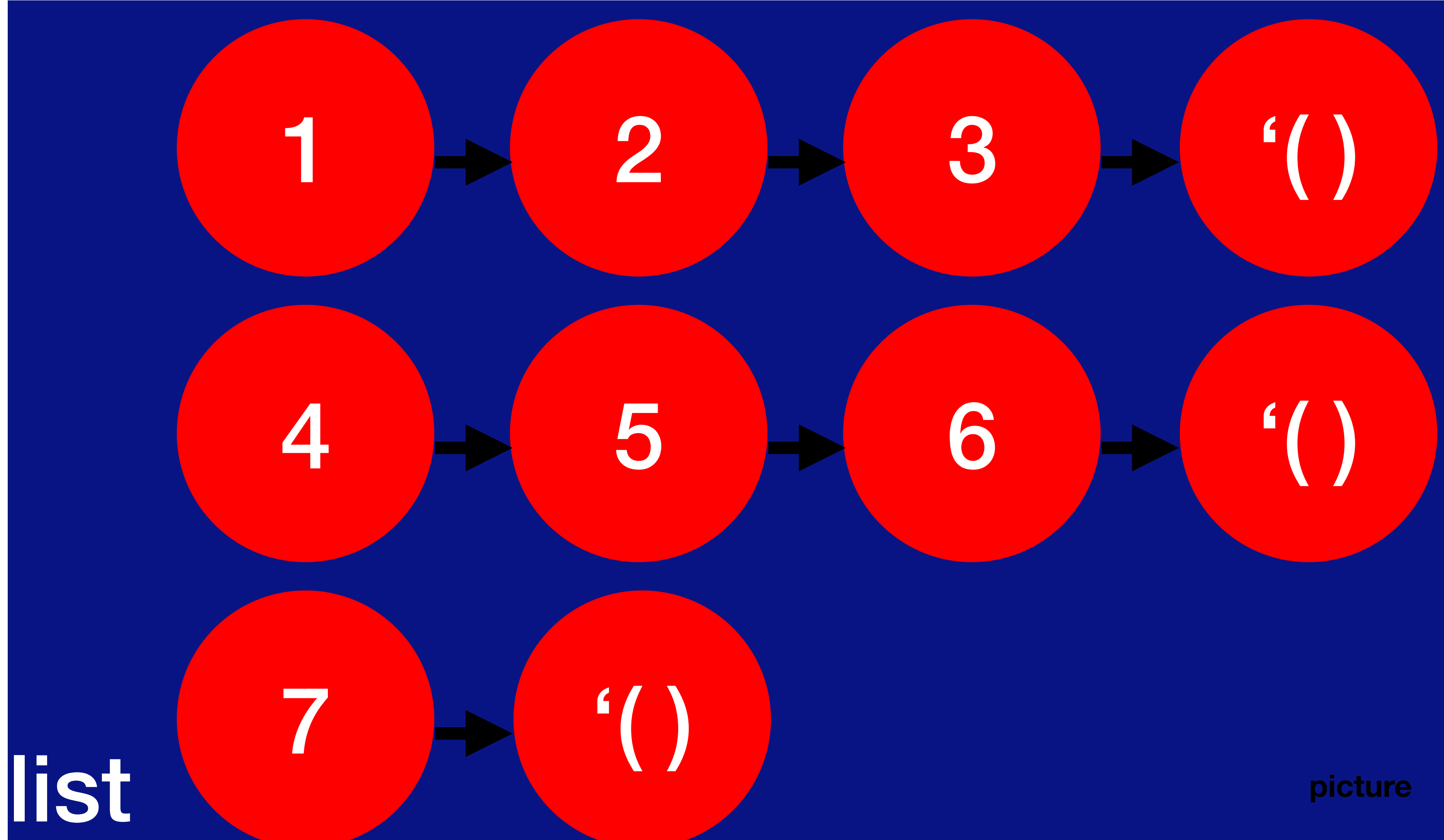
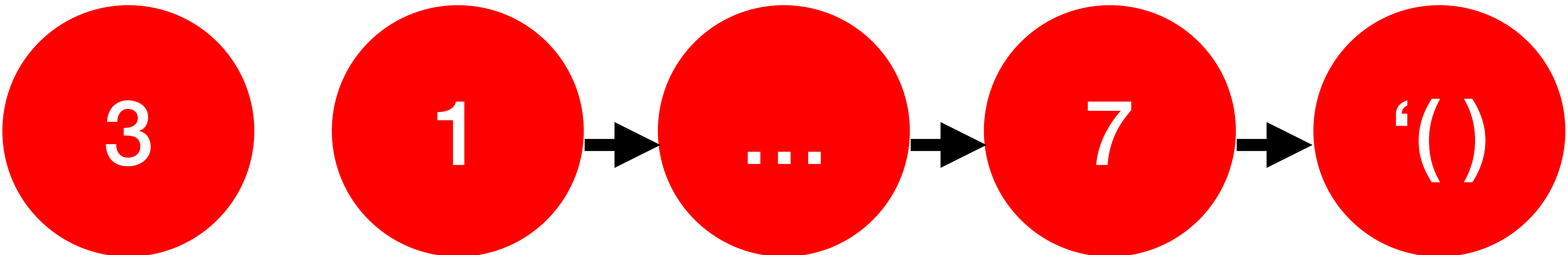








picture



```
; (bundle 0 l) ; '()  
; (bundle 0 '()) ; '()  
; (bundle n '()) ; '()  
; (bundle n l of size  $n * m$ ) ; '(l of size  $m$ , ...)  
; (bundle n l of size  $n * m + p$ )  
    ; '(l of size  $m$ , ..., l of size  $p$ )  
; (bundle n l of size  $< n$ ) ; '(l)
```



```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



```
(define (bundle n l)
  (cond
    [(zero? n) '()]
    [(empty? l) '()]
    [else
     (cons (take n l)
           (bundle n (drop n l)))])
```



```
(define recursive-process  
  ('("identify principal"  
     "test basis"  
     "reduced recursion"  
     "combine results")))
```



**bundle : (n : nat) -> (l : list) -> list list**

principal		l : list
basis	zero?	empty?
reducer		take n
combine		cons



```
(define (bundle n l)
  (cond
    [(zero? n) '()]
    [(empty? l) '()]
    [else
     (cons (take n l)
           (bundle n (drop n l)))])
```



```
( define design-steps  
  ( "problem analysis"  
    "function signature"  
    "examples"  
    "function definition"  
    "tests" ) )
```



*; (bundle 0 1) ; '()*

(check-satisfied  
 (bundle 0 '(1 2 3))  
 empty?)



*; (bundle 0 ' ( ) ) ; ' ( )*

( check-satisfied  
  (bundle 0 ' ( ) )  
  empty? )



*; (bundle n '()) ; '()*

(check-satisfied  
 (bundle 2 '())  
 empty?)



*; (bundle n l of size  $n * m$ ) ; ' (l of size m, ...)*

( check-expect  
 ( bundle 2 ' ( 1 2 3 4 ) )  
 ' ( ( 1 2 ) ( 3 4 ) ) )



```
; (bundle n l of size n * m + p)
; ' (l of size m, ..., l of size p)
```

```
(check-expect
 (bundle 3 '(1 2 3 4 5 6 7))
 '((1 2 3) (4 5 6) (7)))
```

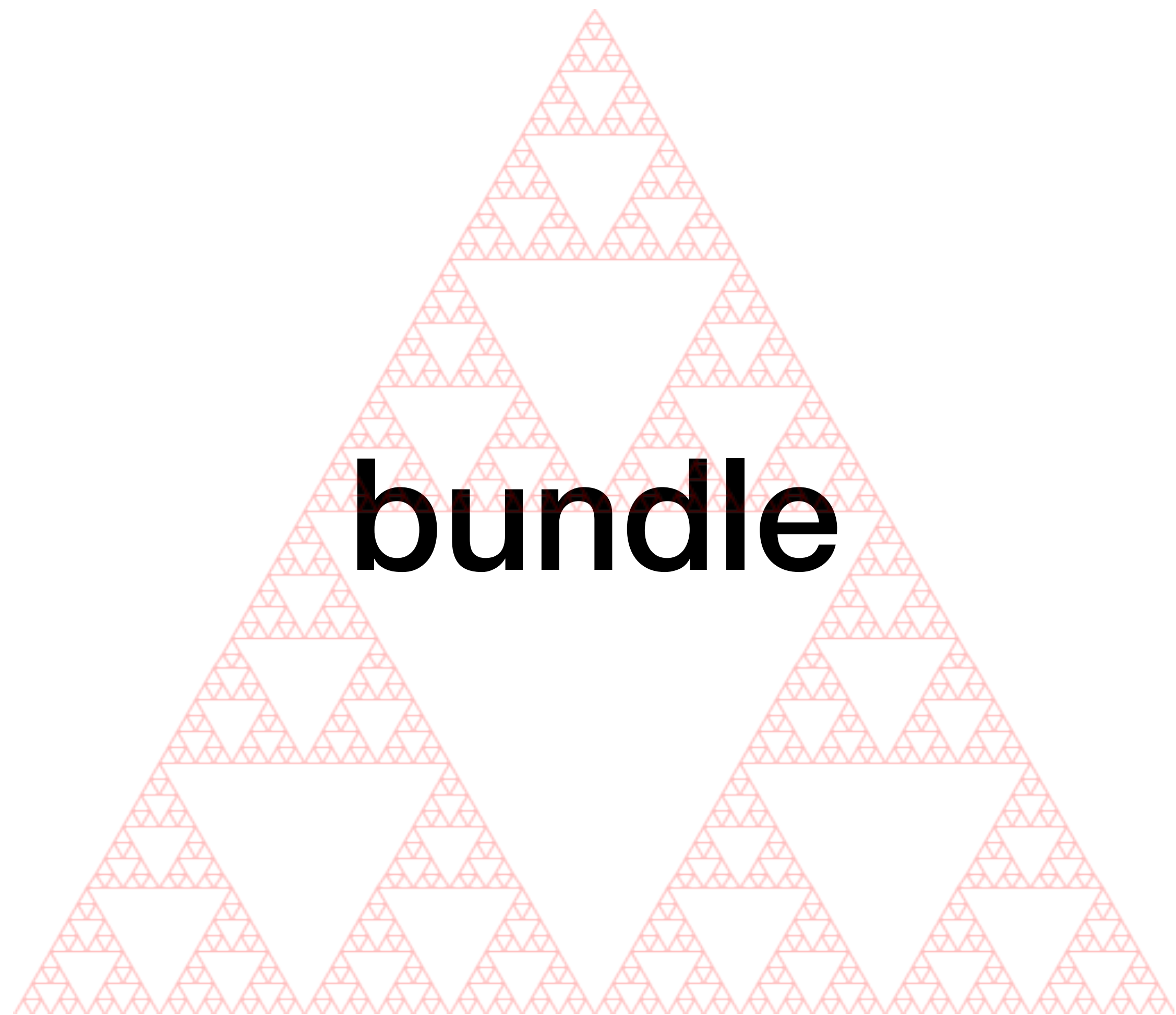


*; (bundle n l of size < n) ; ' (l)*

(check-expect  
 (bundle 3 ' (1 2))  
 ' ((1 2)))





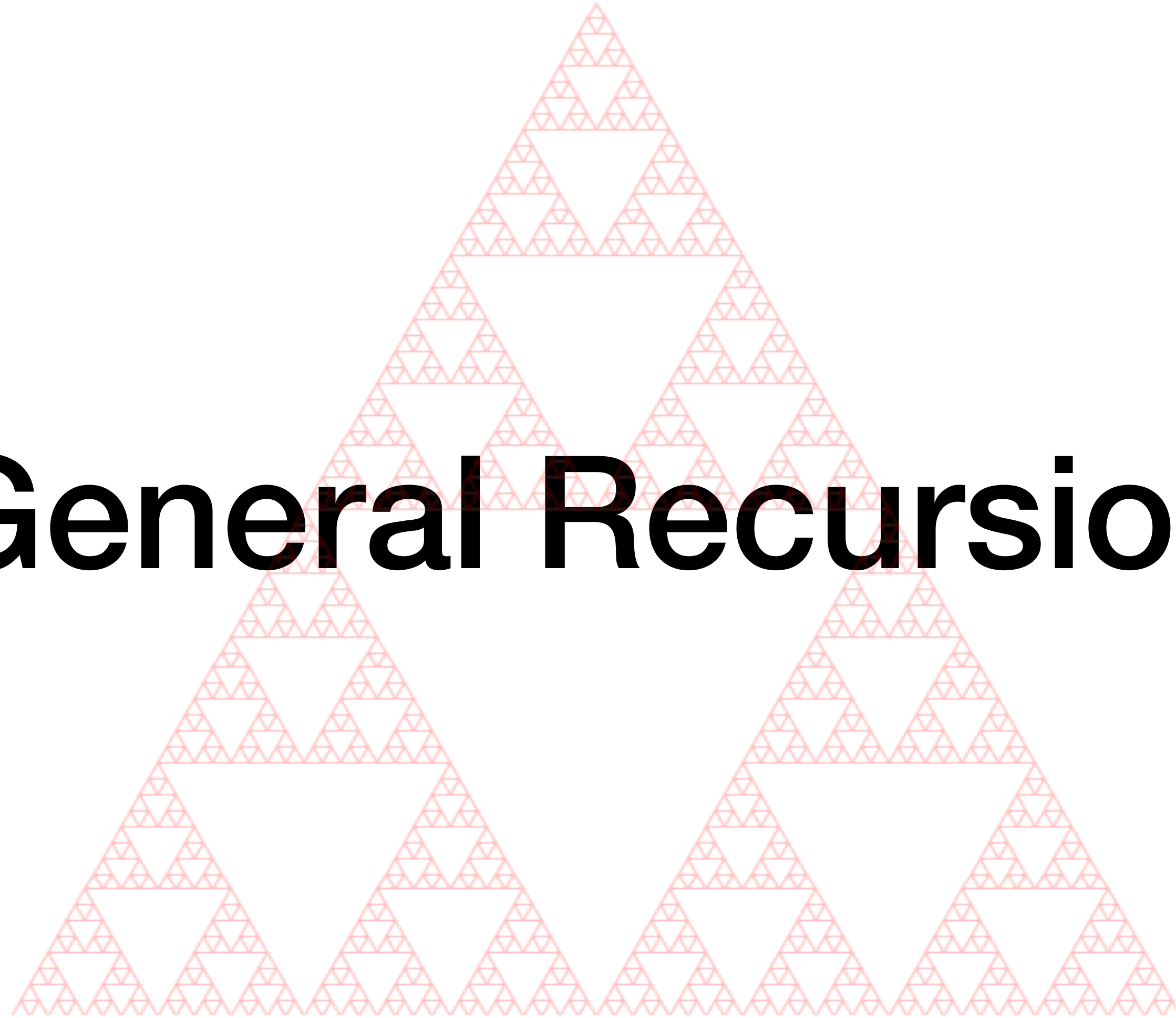


**bundle**

```
(define (bundle n l)
  (cond
    [(zero? n) '()]
    [(empty? l) '()]
    [else
     (cons (take n l)
           (bundle n (drop n l)))])
```



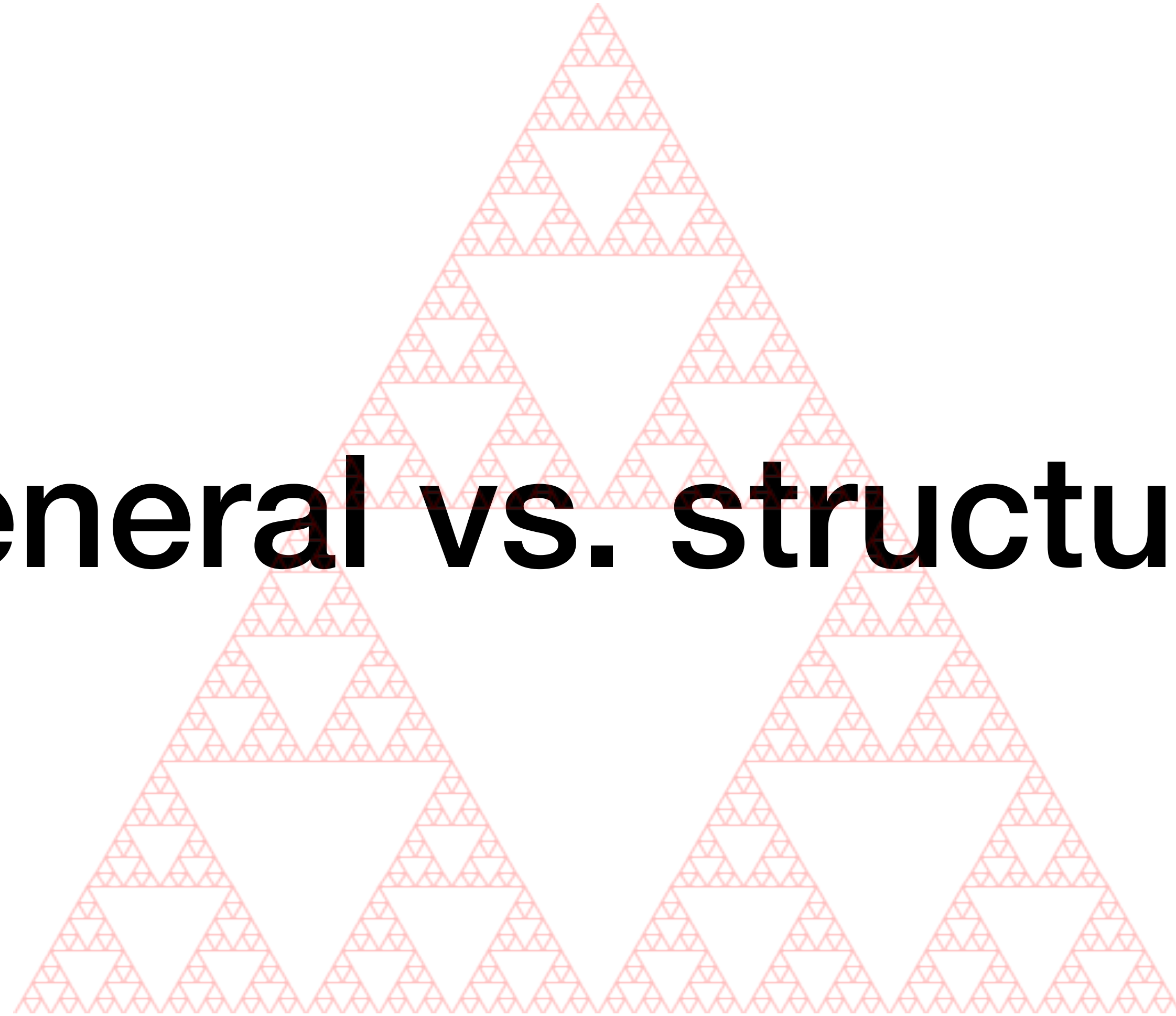
# General Recursion



```
(define (general P)
  (cond
    [(trivial? P) (solve P)]
    [else
     (combine-solutions
      P
      (general
       (generate P)))])
```



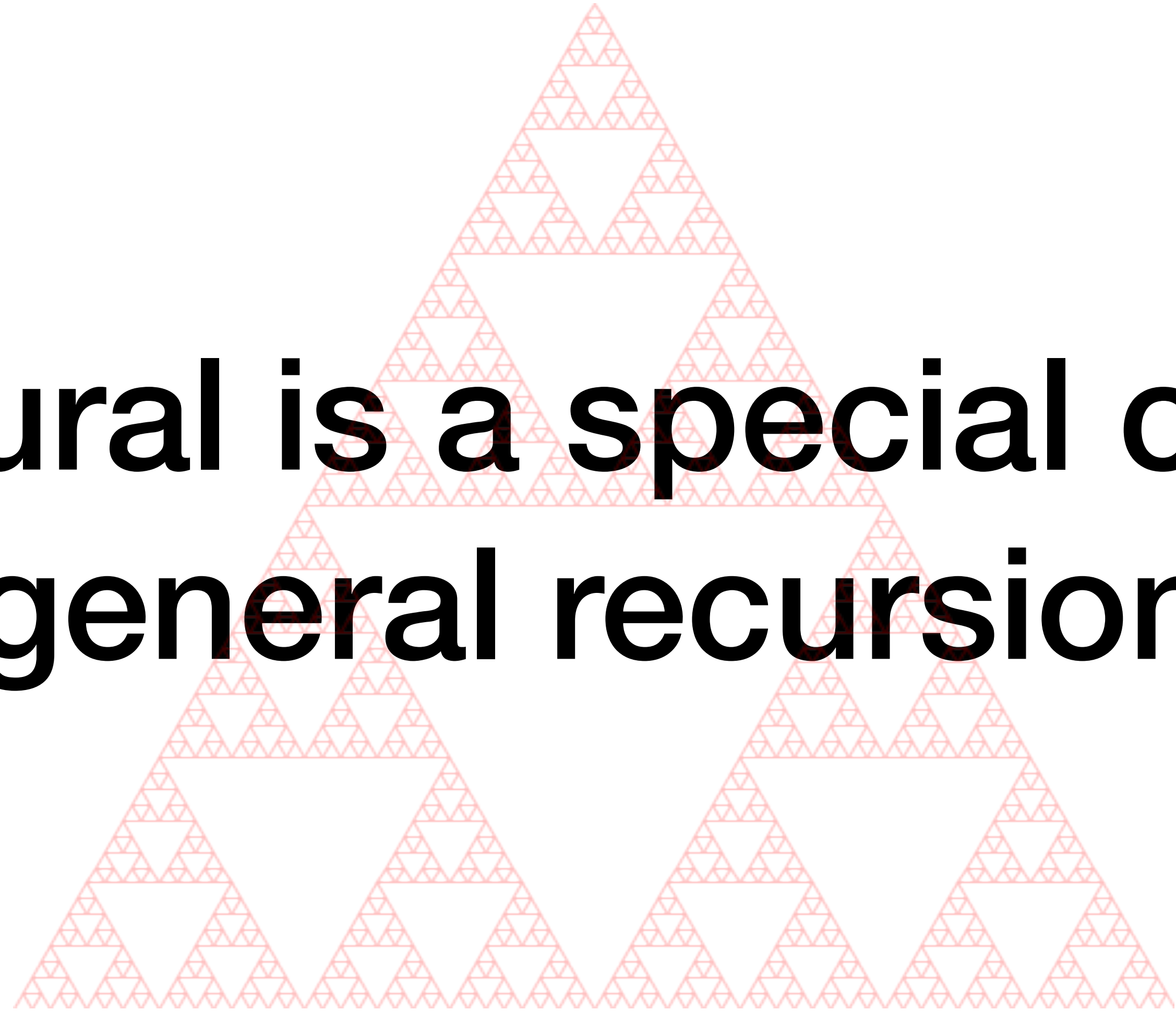
**general vs. structural**



<pre> (define (general P)   (cond     [(trivial? P) (solve P)]     [else      (combine-solutions       P       (general        (generate P)))])) </pre>	<pre> (define (structural P)   (cond     [(basis? P) (solve P)]     [else      (combine-solutions       P       (structural        (induction P)))])) </pre>
---	--



**structural is a special case of  
general recursion**







```
(define topics  
  (make-agenda  
    ('("Recursive Data Type"  
       "Structural Recursion"  
       "General Recursion"))))
```



# Things to Consider

- How would we write a tail call version of plus, take, and bundle?
- What would these examples look like in a strongly typed language?
- How did we get around not having to use the Y combinator or a fix point?

# Further Readings

- **How to Design Programs** 2nd edition by Matthias Felleisen et. al.  
<https://htdp.org/2018-01-06/Book/>
- **An Introduction to Functional Programming Through Lambda Calculus** by Greg Michaelson  
<http://store.doverpublications.com/0486478831.html>
- **Type-Driven Development with Idris** by Edwin Brady  
<https://www.manning.com/books/type-driven-development-with-idris>
- **Coq'Art: The Calculus of Inductive Constructions** by Yves Bertot and Pierre Castéran  
<https://www.springer.com/us/book/9783540208549>



```
(define programmer-recursive-joke  
  (make-joke  
    "Why did the programmer run out of shampoo?"  
    "The instructions said: lather, rinse, repeat."))
```







# Thank you

Mike Harris

Recursion -  
Lather. Rinse. Repeat.  
<https://bit.ly/2qjVHU0>

@MikeMKH



# Source

- examples and slides: <https://github.com/MikeMKH/talks/tree/master/recursion-lather-rinse-repeat>

# Images

- Racket logo: by Matthias.f at en.wikipedia, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14868979>
- Self: taken by Kelsey Harris at StrangeLoop

**appendix**

**(plus 1 2) expanded using DrRacket**



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
```

```
(plus 1 2)
```



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
```

```
(cond
  ((zero? 1) 2)
  (else
   (add1 (plus (sub1 1) 2))))
```

```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))

(cond
  ((zero? 1) 2)
  (else
   (add1 (plus (sub1 1) 2))))
```



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))

(cond
  (#false 2)
  (else
   (add1 (plus (sub1 1) 2))))
```

```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
```

```
(cond
  (#false 2)
  (else
   (add1 (plus (sub1 1) 2))))
```



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
```

```
(cond
  (else
   (add1 (plus (sub1 1) 2))))
```

```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
```

```
(cond
  (else
   (add1 (plus (sub1 1) 2))))
```



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
(add1 (plus (sub1 1) 2))
```



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
(add1 (plus (sub1 1) 2))
```



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
(add1 (plus 0 2))
```

```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
(add1 (plus 0 2))
```



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
(add1
  (cond
    ((zero? 0) 2)
    (else
     (add1 (plus (sub1 0) 2)))))
```

```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))

(add1
 (cond
  ((zero? 0) 2)
  (else
   (add1 (plus (sub1 0) 2)))))
```



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))

(add1
 (cond
  (#true 2)
  (else
   (add1 (plus (sub1 0) 2)))))
```

```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
```

```
(add1
 (cond
  (#true 2)
  (else
   (add1 (plus (sub1 0) 2)))))
```



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
(add1 2)
```



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
```

```
(add1 2)
```



```
(define (plus a b)
  (cond
    ((zero? a) b)
    (else
     (add1 (plus (sub1 a) b)))))
```

3