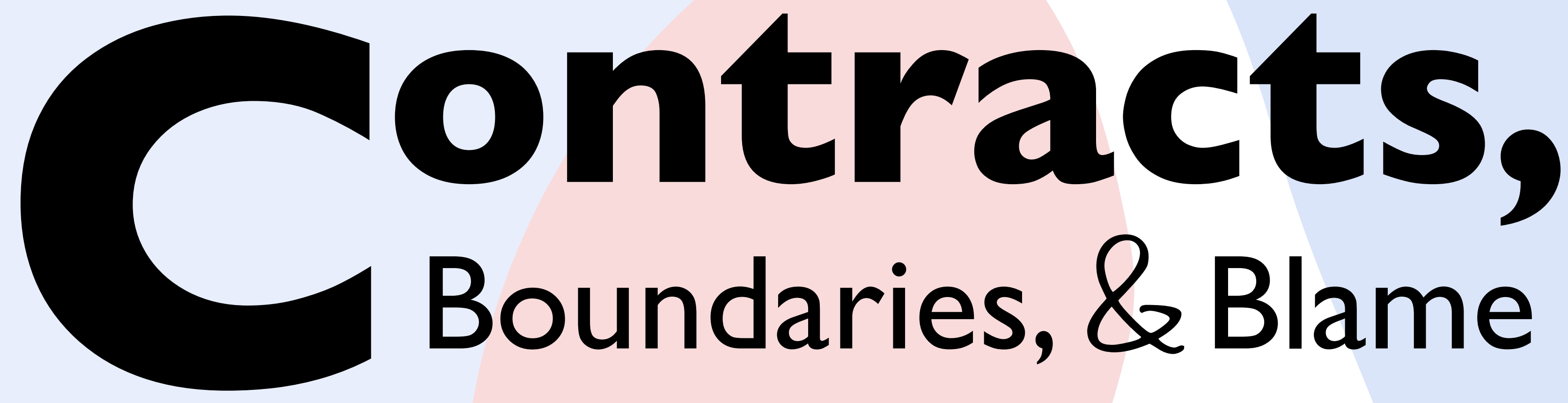


Robby Findler
Northwestern & PLT

point-in-module

```
(provide point-in?)

(define (point-in? p x y)
  (define p-dot
    (pin-under p x y (disk 1)))
  (equal?
   (pict->argb-pixels p-dot)
   (pict->argb-pixels p)))
```

point-in-module

```
(provide point-in?)

(define (point-in? p x y)
  (define p-dot
    (pin-under p x y (disk 1)))
  (equal?
   (pict->argb-pixels p-dot)
   (pict->argb-pixels p)))
```

```
(point-in? (cloud 100 100) 0 0)
```

point-in-module

```
(provide point-in?)

(define (point-in? p x y)
  (define p-dot
    (pin-under p x y (disk 1)))
  (equal?
   (pict->argb-pixels p-dot)
   (pict->argb-pixels p)))
```

```
(point-in?        0 0)
```

```
(define (point-in? p x y)
  (define p-dot
    (pin-under p x y (disk 1)))
  (equal?
   (pict->argb-pixels p-dot)
   (pict->argb-pixels p)))

(point-in?        0 0)
```

```
(define p-dot
  (pin-under       0 0 (disk 1)))
(equal?
 (pict->argb-pixels p-dot)
 (pict->argb-pixels      ))
```

```
(define p-dot
                                        )

(equal?
  (pict->argb-pixels p-dot)
  (pict->argb-pixels       ))
```

```
(equal?
 (pict->argb-pixels      )
 (pict->argb-pixels      ))
```

```
(equal?
 #"\0\377\377\377\0\3...")
 #"\377\377\0\0\377\3..."))
```

point-in-module

```
(provide point-in?)

(define (point-in? p x y)
  (define p-dot
    (pin-under p x y (disk 1)))
  (equal?
   (pict->argb-pixels p-dot)
   (pict->argb-pixels p)))
```

```
(point-in? (cloud 100 100) 50 50)
```

```
(define p-dot
  (pin-under       50 50 (disk 1)))
(equal?
 (pict->argb-pixels p-dot)
 (pict->argb-pixels      ))
```

```
(define p-dot
                                                  )
(equal?
 (pict->argb-pixels p-dot)
 (pict->argb-pixels     ))
```

```
(equal?
 (pict->argb-pixels      )
 (pict->argb-pixels      ))
```

```
(equal?
 #"\0\377\377\377\0\3...")
 #"\0\377\377\377\0\3..."))
```

```
> (point-in? (cloud 100 100) #f #f)
```

```
> (point-in? (cloud 100 100) #f #f)
pin-under: contract violation
  expected: (or/c real? pict-path?)
  given: #f
  in: the dx/fp argument of
      (->i
       ((base pict?)
        (dx/fp (or/c real? pict-path?))
        (dy/f
         (dx/fp)
         (if (real? dx/fp)
           real?
           (->
            pict?
            pict-path?
            (values real? real?))))
        (pict pict?))
       (result pict?))
  contract from: <pkgs>/pict-lib/pict/main.rkt
  blaming: point-in-module
   (assuming the contract is correct)
```

```
> (point-in? (cloud 100 100) #f #f)
pin-under: contract violation
  expected: (or/c real? pict-path?)
  given: #f
  in: the dx/fp argument of
      (->i
       ((base pict?)
        (dx/fp (or/c real? pict-path?))
        (dy/f
         (dx/fp)
         (if (real? dx/fp)
           real?
           (->
            pict?
            pict-path?
            (values real? real?))))
        (pict pict?))
```

blaming: point-in-module

```
(-> pict? real? real?
    boolean?)
```

point-in-module

```
(provide/contract
 [point-in? (-> pict? real? real?
                boolean?)])
(define (point-in? p x y)
  (define p-dot
    (pin-under p x y (disk 1)))
  (equal?
   (pict->argb-pixels p-dot)
   (pict->argb-pixels p)))
```

point-in-module

```
(provide/contract
 [point-in? (-> pict? real? real?
                boolean?)])
(define (point-in? p x y)
  (define p-dot
    (pin-under p x y (disk 1)))
  (equal?
   (pict->argb-pixels p-dot)
   (pict->argb-pixels p)))
```

```
(point-in? (cloud 100 100) #f #f)
```

point-in-module

```
(provide/contract
 [point-in? (-> pict? real? real?
                boolean?)])
(define (point-in? p x y)
  (define p-dot
    (pin-under p x y (disk 1)))
  (equal?
   (pict->argb-pixels p-dot)
   (pict->argb-pixels p)))
```

```
(point-in?       #f #f)
```

```
point-in?: contract violation
  expected: real?
  given: #f
  in: the 2nd argument of
      (-> pict? real? real? boolean?)
  contract from: point-in-module
  blaming: top-level
   (assuming the contract is correct)
```

Nope:

Contracts are not Types

point-in-module

```
(provide/contract
 [point-in? (-> pict? real? real?
                boolean?)])
(define (point-in? p x y)
  (define p-dot
    (pin-under p x y (disk 1)))
  (equal?
   (pict->argb-pixels p-dot)
   (pict->argb-pixels p)))
```

```
(point-in? (cloud 100 100) -50 -75)
```

```
(define p-dot
  (pin-under        -50 -75 (disk 1)))
(equal?
 (pict->argb-pixels p-dot)
 (pict->argb-pixels       ))
```

```
(define p-dot
                                          )
(equal?
 (pict->argb-pixels p-dot)
 (pict->argb-pixels    ))
```

```
(equal?
 (pict->argb-pixels       )
 (pict->argb-pixels      ))
```

```
(equal?
 #"\0\377\377\377\0\3...")
 #"\0\377\377\377\0\3..."))
```

```
(-> pict?
    real?
    real?
    boolean?)
```

```
(-> pict?
    (>=/c 0)
    (>=/c 0)
    boolean?)
```

```
(->        pict?
           (>=/c 0)
           (>=/c 0)
           boolean? )
```

```
(->i ([p pict?]
      [x (>=/c 0)]
      [y (>=/c 0)])
     [res boolean?])
```

```
(->i ([p pict?]
      [x     (>=/c 0)]
      [y     (>=/c 0)])
     [res boolean?])
```

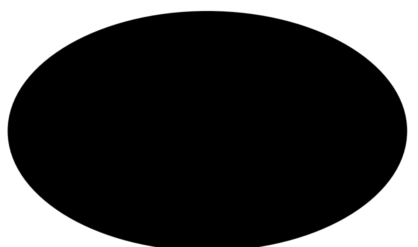
```
(->i ([p pict?]
      [x (p) (>=/c 0)]
      [y (p) (>=/c 0)])
    [res boolean?])
```

```
(->i ([p pict?]
      [x (p) (>=/c 0)                    ]
      [y (p) (>=/c 0)                    ])
     [res boolean?])
```

```
(->i ([p pict?]
      [x (p) (real-in 0 (p-w p))]
      [y (p) (real-in 0 (p-h p))])
     [res boolean?])
```

```
> (point-in? (cloud 100 100) -50 -75)
point-in?: contract violation
  expected: (real-in 0 100)
  given: -50
  in: the w argument of
      (->i
       ((p pict?)
        (w (p) (real-in 0 (pict-width p)))
        (h (p) (real-in 0 (pict-height p))))
       (res boolean?))
  contract from: point-in-module
  blaming: top-level
   (assuming the contract is correct)
```

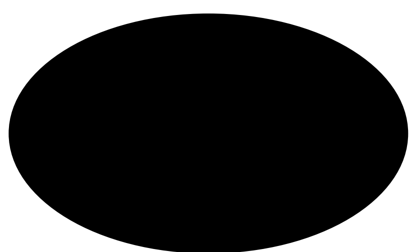
```
(dc (λ (dc dx dy)
      (send dc draw-ellipse
            dx dy 200 120))

    200
    120)
```



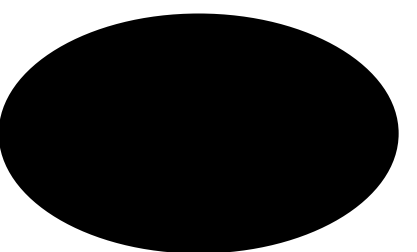
```
(colorize
 (dc (λ (dc dx dy)
       (send dc draw-ellipse
             dx dy 200 120))
     200
     120)
 "blue")
```

```
(dc (λ (dc dx dy)

       (send dc draw-ellipse
             dx dy 200 120)
                                )
    200
    120)
```

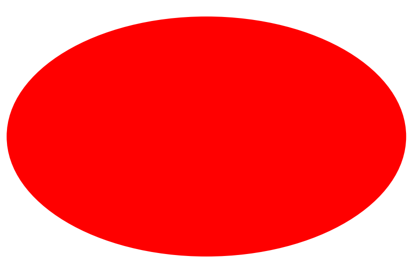
```
(dc (λ (dc dx dy)
      (define oldb (send dc get-brush))
      (send dc draw-ellipse
            dx dy 200 120)
      (send dc set-brush oldb))
    200
    120)
```



```
(dc (λ (dc dx dy)
      (define oldb (send dc get-brush))

      (send dc draw-ellipse
               dx dy 200 120)
      (send dc set-brush oldb))
    200
    120)
```

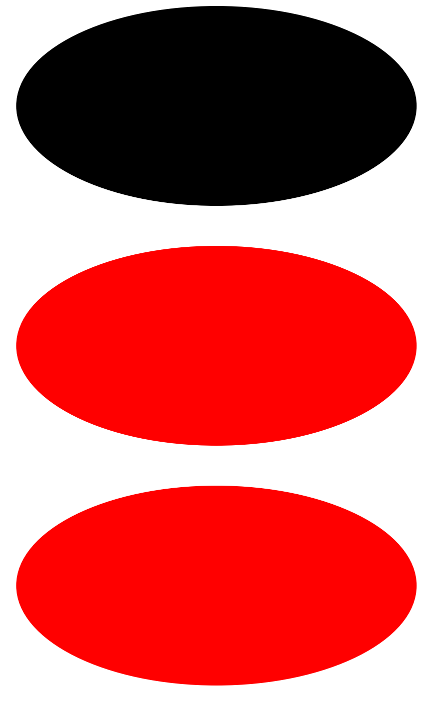


```
(dc (λ (dc dx dy)
      (define oldb (send dc get-brush))
      (send dc set-brush "red" 'solid)
      (send dc draw-ellipse
               dx dy 200 120)
      (send dc set-brush oldb))
    200
    120)
```

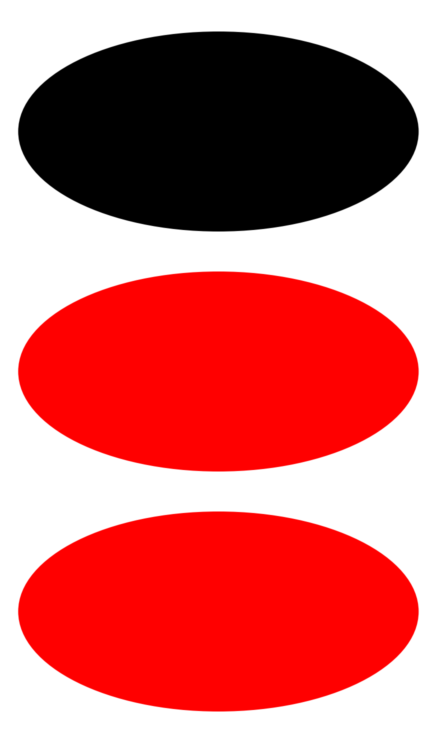


```
(colorize
 (dc (λ (dc dx dy)
       (define oldb (send dc get-brush))
       (send dc set-brush "red" 'solid)
       (send dc draw-ellipse
                dx dy 200 120)
       (send dc set-brush oldb))
     200
     120)
 "blue")
```

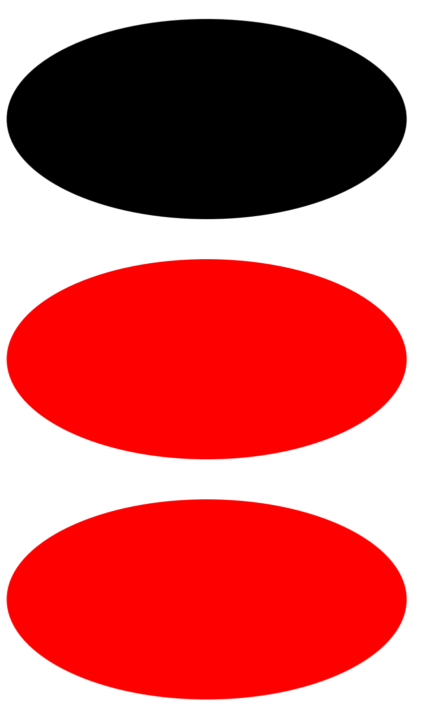
```
(define eps
  (dc
   (λ (dc dx dy)
     (send dc draw-ellipse
           dx dy 200 100))
   200 100))
(define red
  (unsafe:dc
   (λ (dc dx dy)
     (send dc set-brush
           "red" 'solid)
     (send dc draw-ellipse
           dx dy 200 100))
   200 100))
(colorize
 (vc-append 20 eps red eps)
 "black")
```



```
(define eps
  (dc
   (λ (dc dx dy)
     (send dc draw-ellipse
           dx dy 200 100))
   200 100))
(define red
  (unsafe:dc
   (λ (dc dx dy)
     (send dc set-brush
           "red" 'solid)
     (send dc draw-ellipse
           dx dy 200 100))
   200 100))
(colorize
 (vc-append 20 eps red eps)
 "black")
```



```
(define eps
  (dc
   (λ (dc dx dy)
     (send dc draw-ellipse
           dx dy 200 100))
   200 100))
(define red
  (unsafe:dc
   (λ (dc dx dy)
     (send dc set-brush
           "red" 'solid)
     (send dc draw-ellipse
           dx dy 200 100))
   200 100))
(colorize
 (vc-append 20 eps red eps)
 "black")
```



```
(->i ([f (-> dc<%> real? real?
             void?)]
       [w real?]
       [h real?])
      [result pict?])
```

Wrapping **f**: bad idea!

- checking happens too late
- too expensive

```
(->i ([f (-> dc<%> real? real?
             void?)]
      [w real?]
      [h real?])
     [result pict?])
```

```
(->i ([f (-> dc<%> real? real?
              void?)]
      [w real?]
      [h real?])


     [result pict?])
```

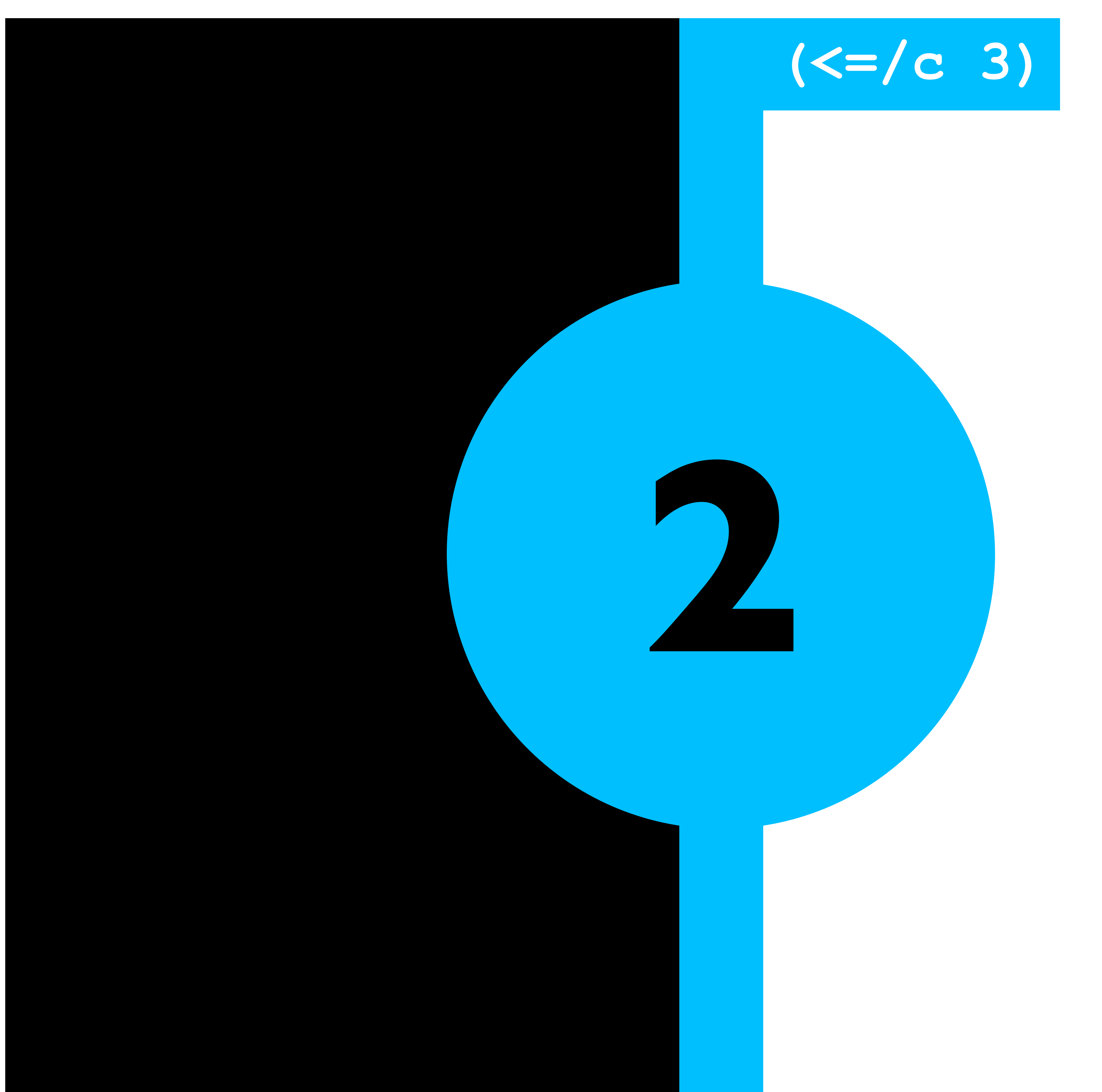
```
(->i ([f (-> dc<%> real? real?
             void?)]
      [w real?]
      [h real?])
     #:pre (f)
     (restores-state-after-call? f)
     [result pict?])
```

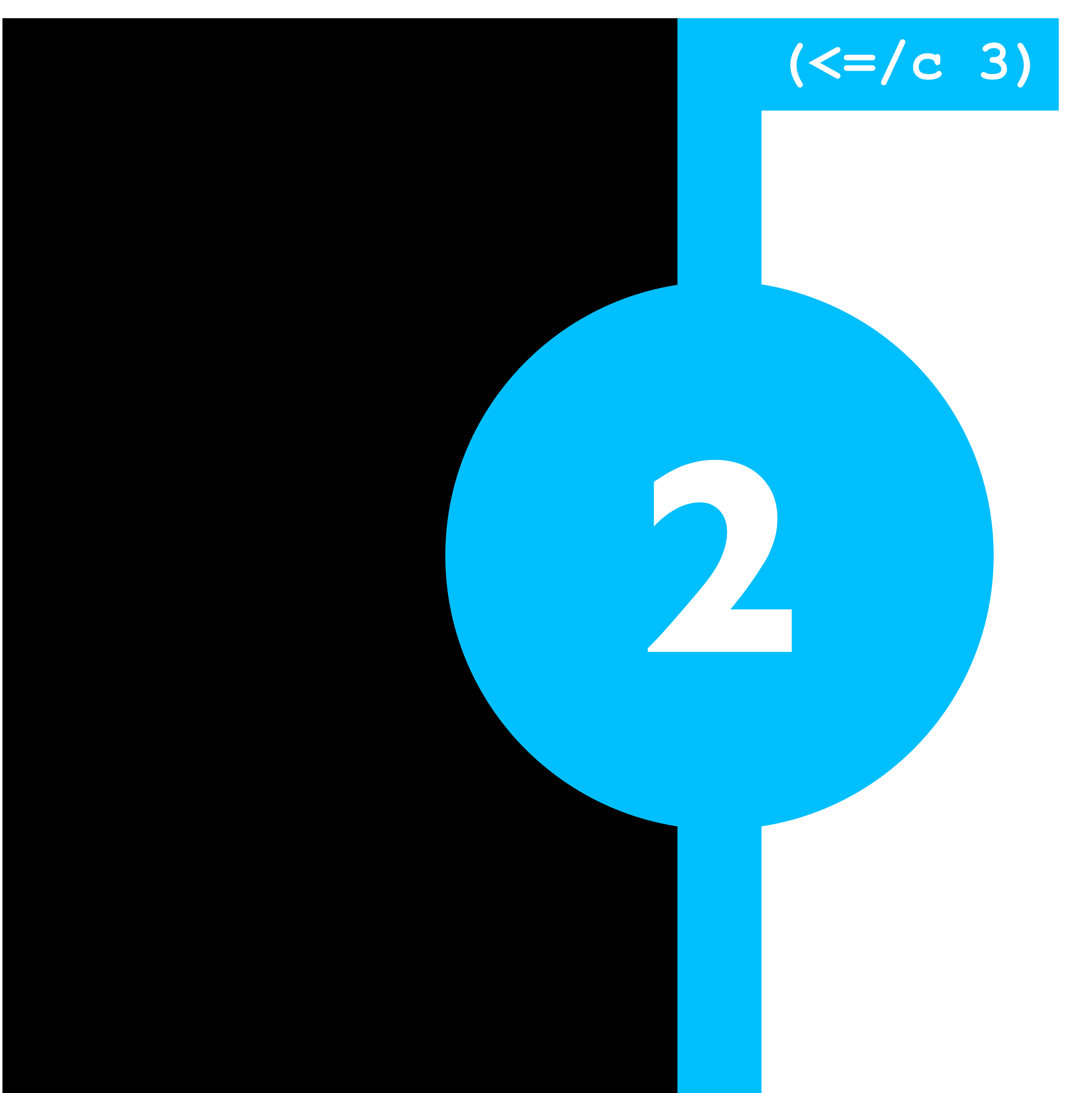
```
(define (restores-state-after-call? f)
  (define a-dc (make-bitmap-backed-dc))
  (randomize-state a-dc)
  (define before (get-dc-state a-dc))
  (f a-dc 0 0)
  (equal? before (get-dc-state a-dc)))
```

Semantics:
Boundaries

```
(<=/c 3)
```

2


(<=/c 3)
2

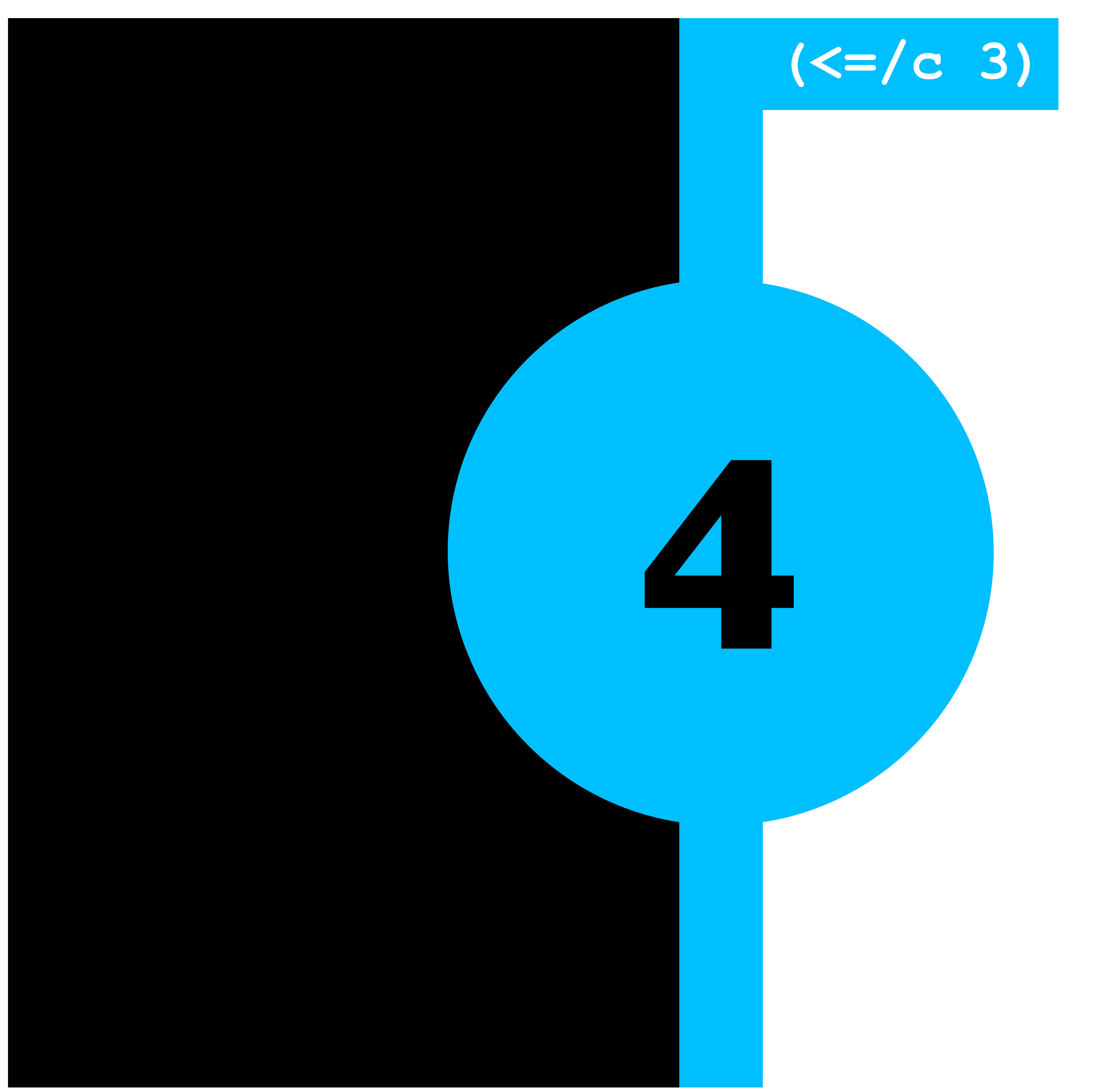

(<=/c 3)
2

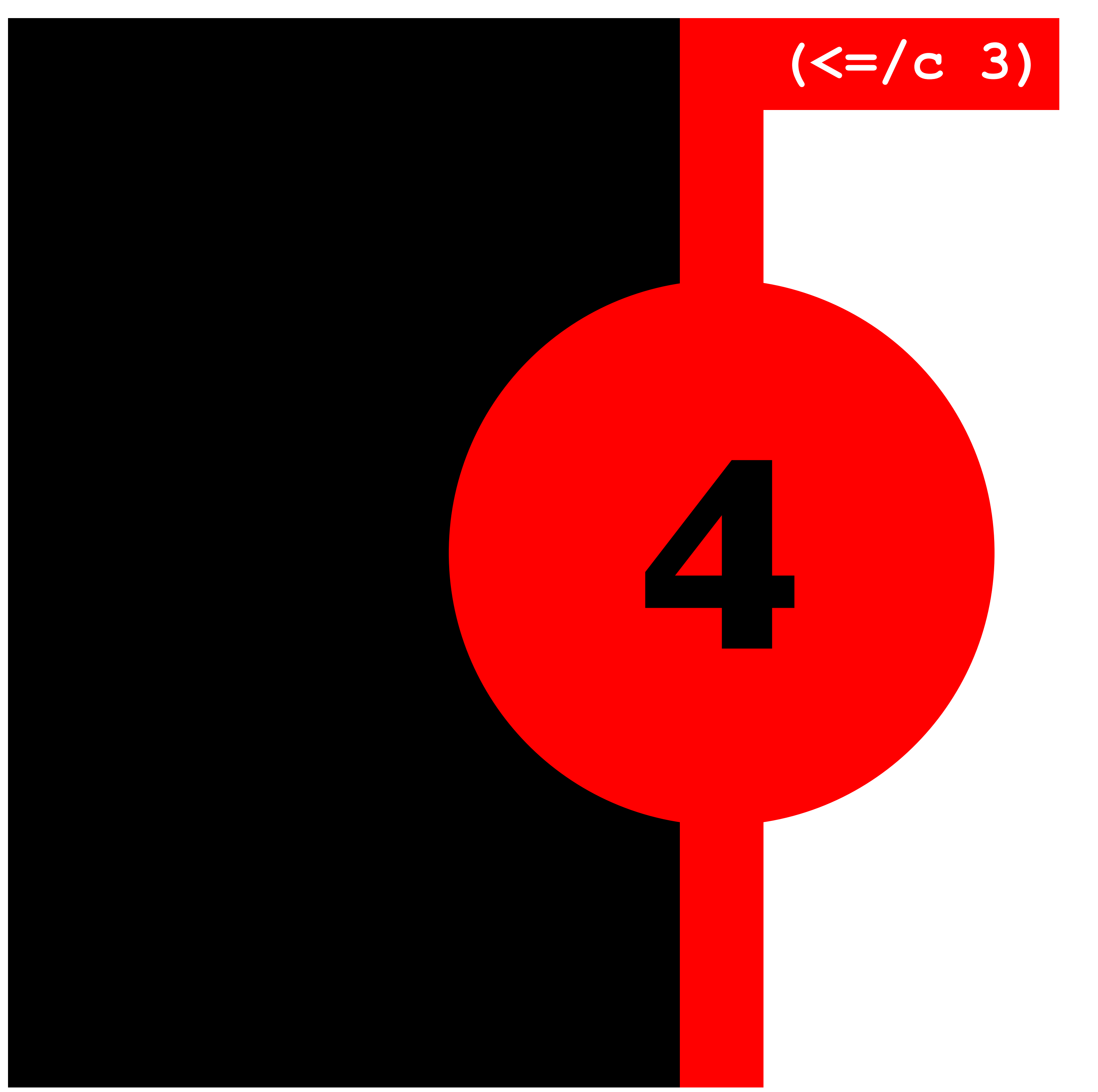
```
(<=/c 3)
```

2

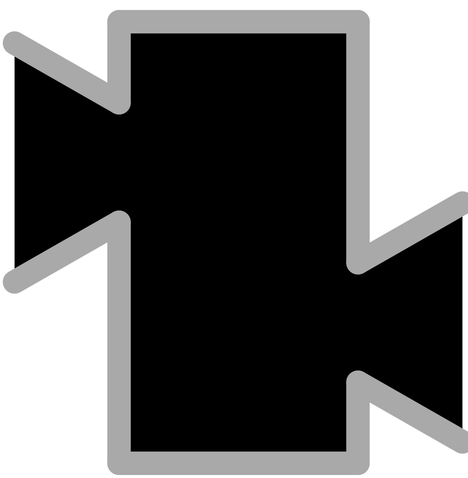
(<=/c 3)

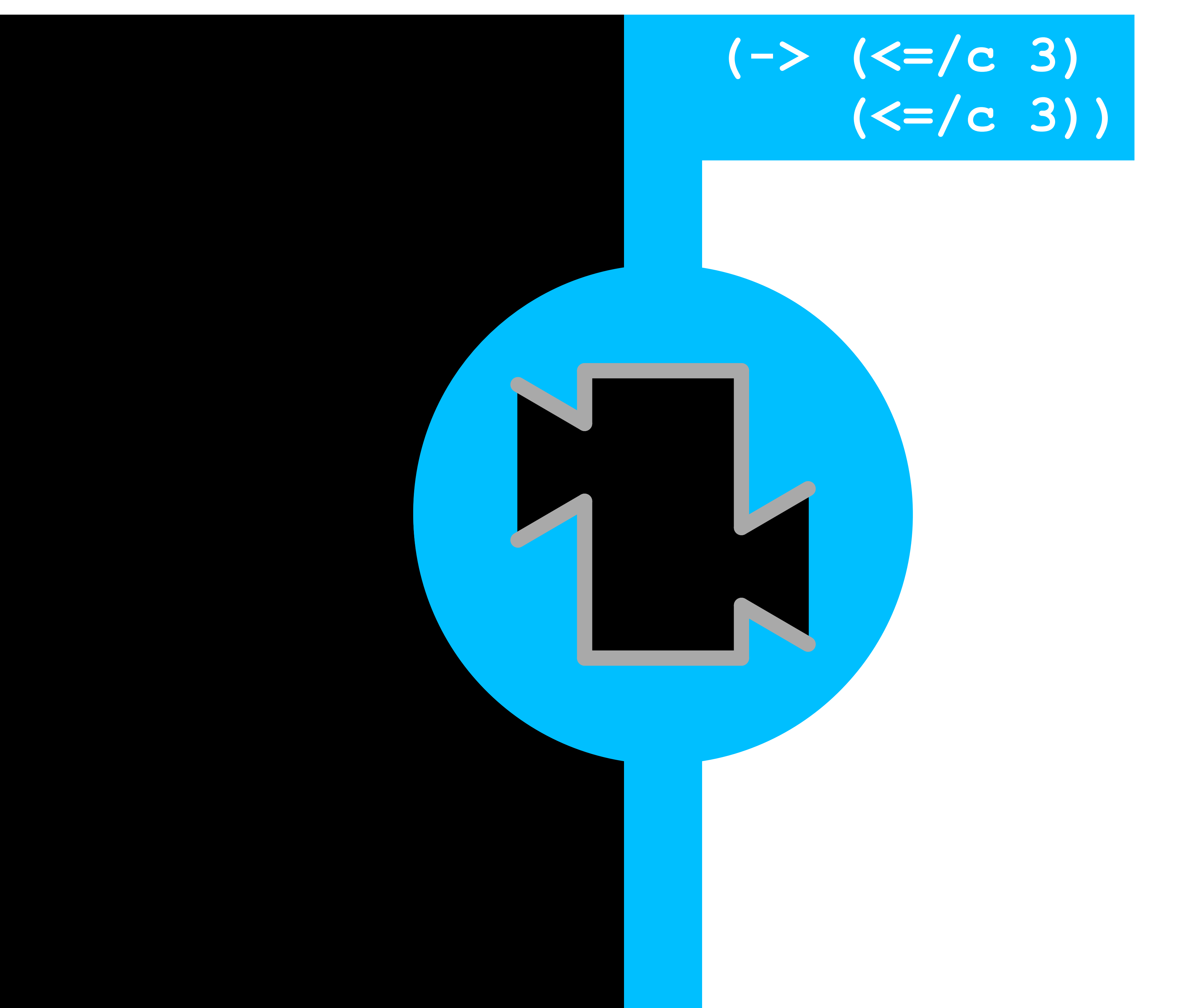
4

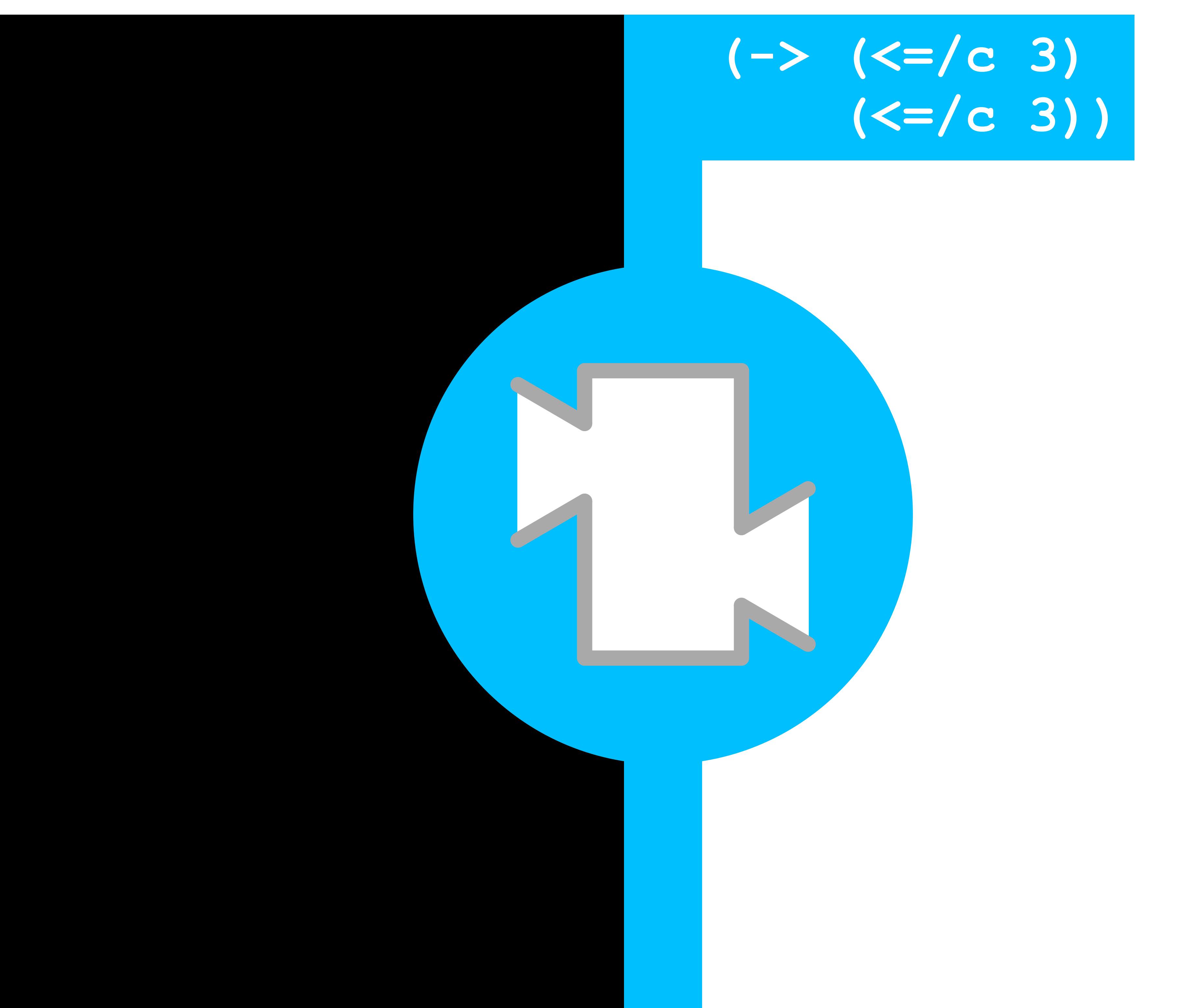

(<=/c 3)
4

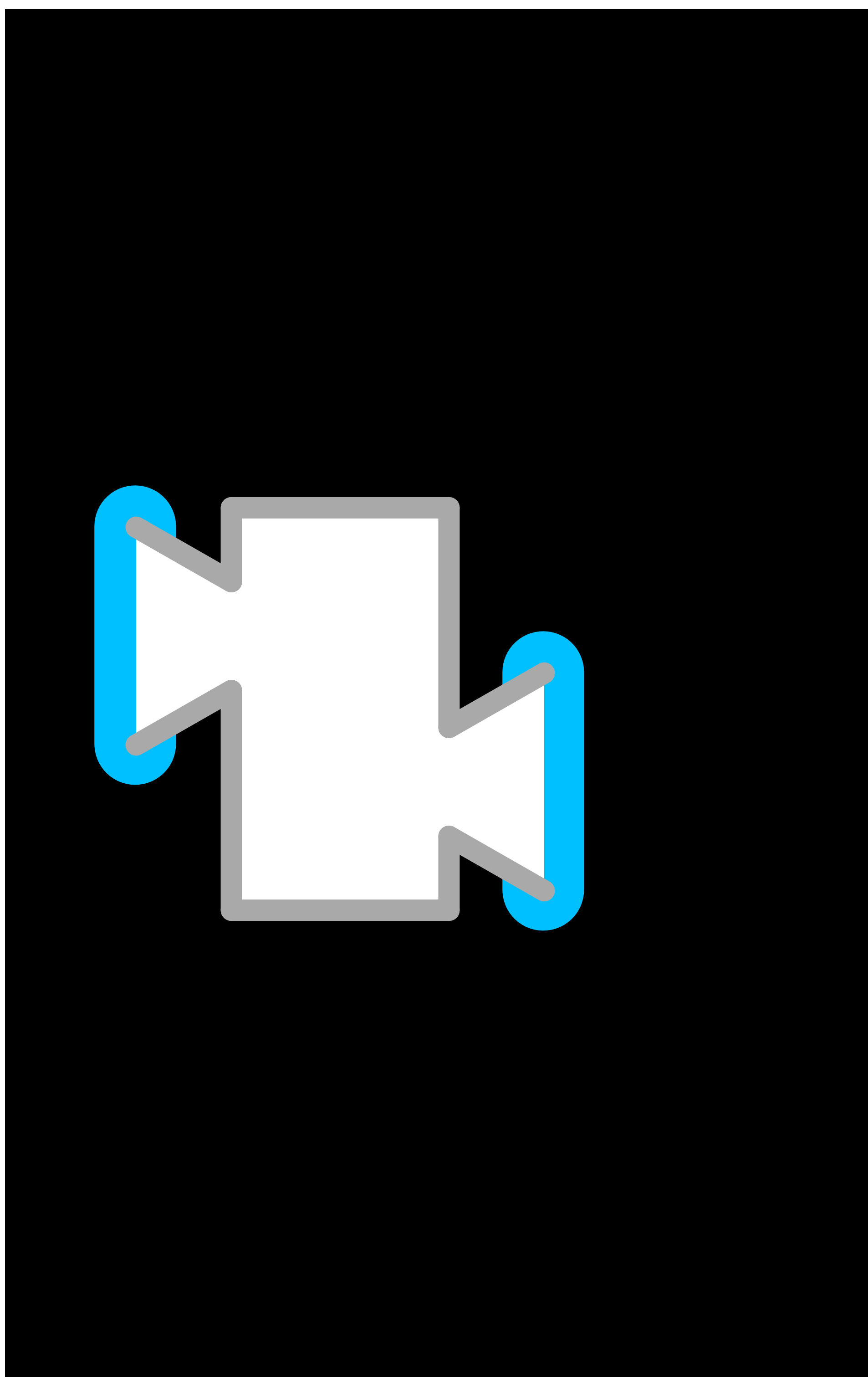

(<=/c 3)
4

```
(-> (<=/c 3)
    (<=/c 3))
```

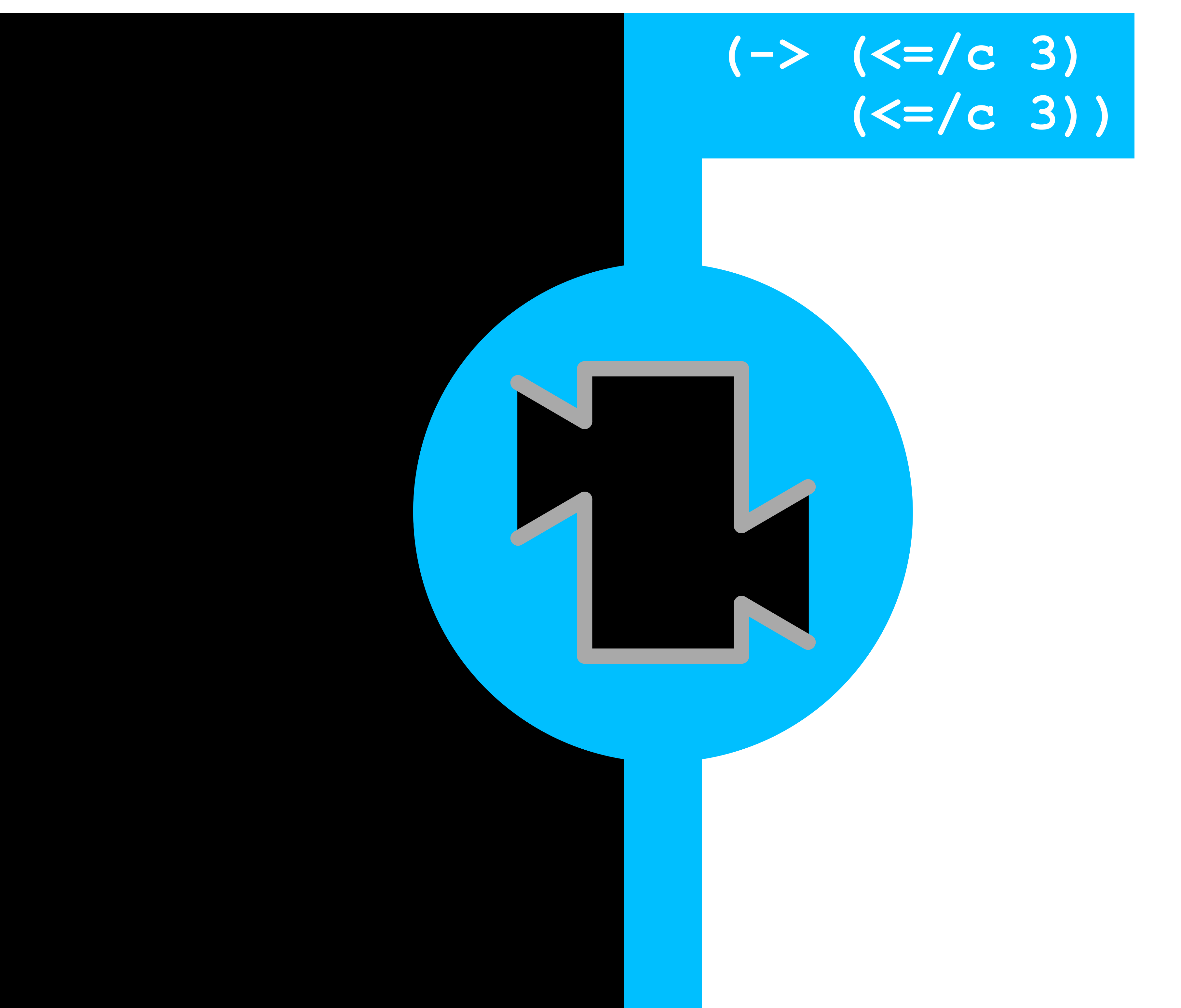


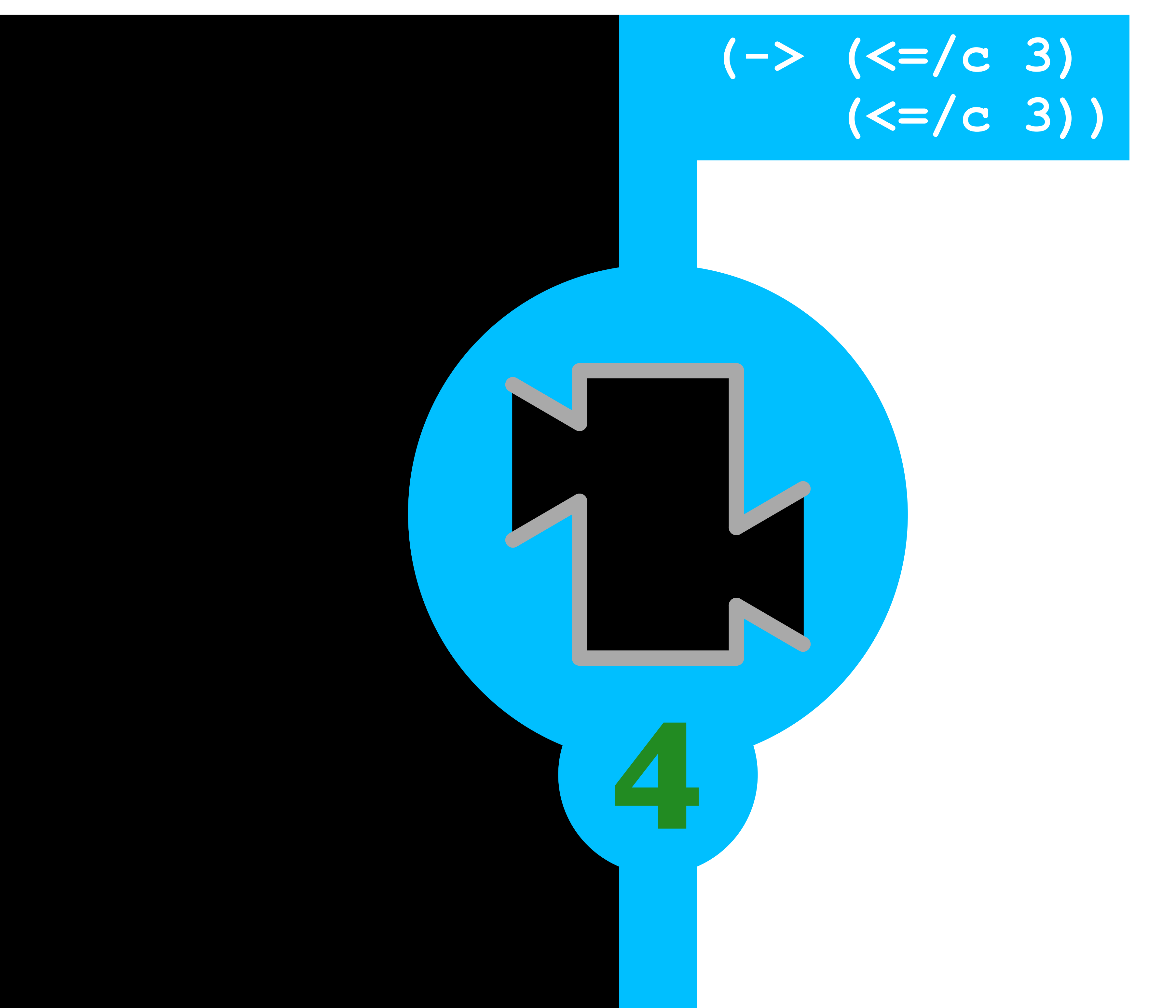

(-> (<=/c 3)
(<=/c 3))

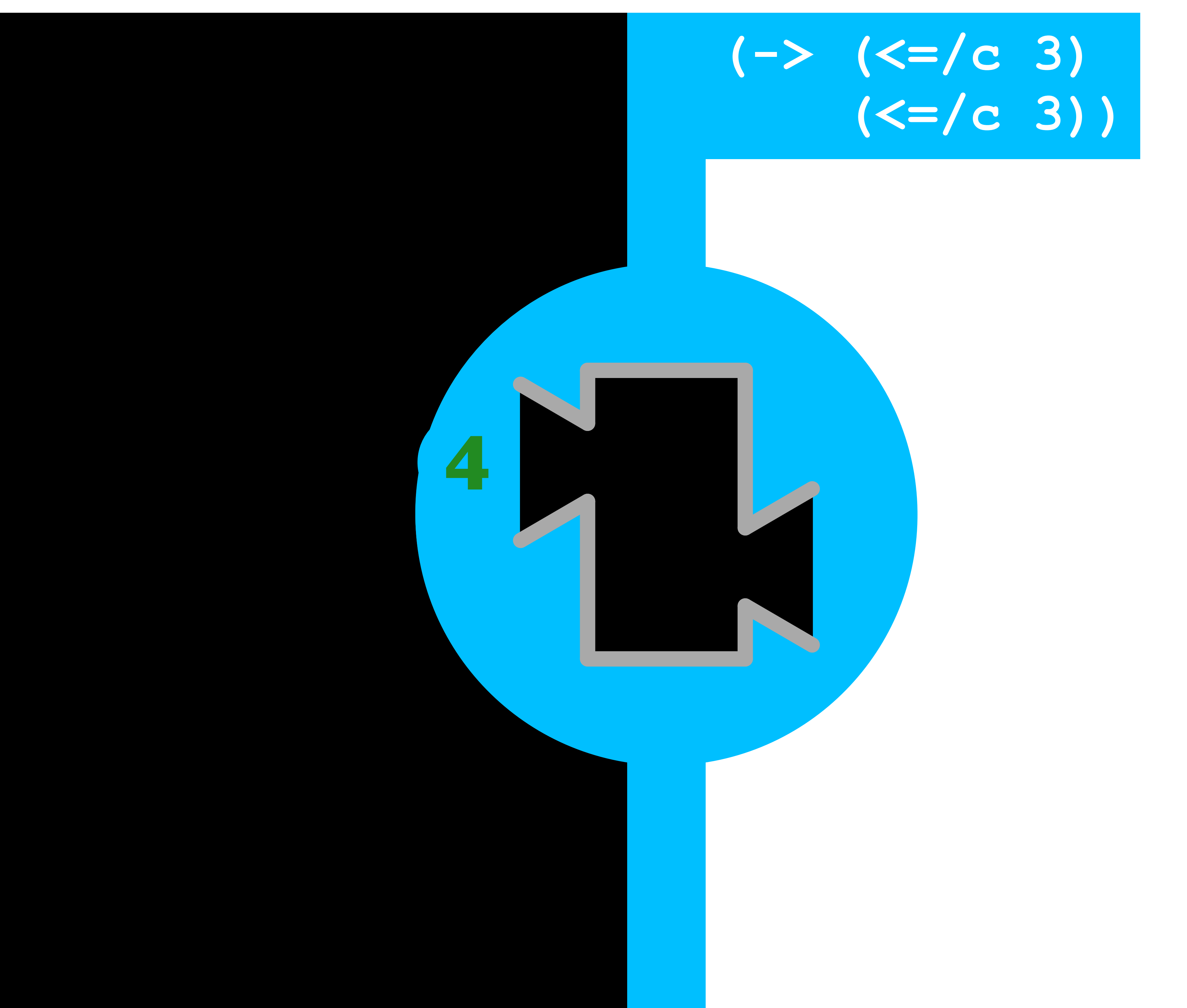

(-> (<=/c 3)
(<=/c 3))



```
(-> (<=/c 3)
    (<=/c 3))
```


(-> (<=/c 3)
(<=/c 3))


(-> (<=/c 3)
(<=/c 3))
4


(-> (<=/c 3)
(<=/c 3))
4

Application: **Random Testing**

My
Code

Your
Code

My Code

My
Code

Randomness

```racket
#lang racket/base
(require pict pict/tree-layout
         racket/match racket/contract)

(struct binary-heap (size vec) #:mutable #:transparent)
(define (valid-heap? heap)
  (match heap
    [(binary-heap size vec)
     (let loop ([i 0]
                [parent -inf.0])
       (cond
         [(< i size)
          (define this (vector-ref vec i))
          (and (<= parent this)
               (loop (left-child i) this)
               (loop (right-child i) this))]
         [else #t]))]))
(define heap/c (and/c binary-heap? valid-heap?))

(provide
 (contract-out

  [new-heap
   (-> heap/c)]

  [insert!
   (->i ([h heap/c]
         [i integer?])
        [result void?]
        #:post (h) (valid-heap? h))]

  [delete!
   (->i ([h heap/c])
        [res (or/c integer? #f)]
        #:post (h) (valid-heap? h))]))

(define (new-heap) (binary-heap 0 (make-vector 1 #f)))

(define (insert! heap nv)
  (match heap
    [(binary-heap size vec)
     (unless (< size (vector-length vec))
       (define new-vec
         (make-vector (* (vector-length vec) 2) #f))
       (vector-copy! new-vec 0 vec)
       (set-binary-heap-vec! heap new-vec)
       (set! vec new-vec))
     (vector-set! vec size nv)
     (set-binary-heap-size! heap (+ size 1))
     (let loop ([i size] [iv nv])
       (unless (= i 0)
         (define p (parent i))
         (define pv (vector-ref vec p))
         (when (< iv pv)
           (vector-set! vec p iv)
           (vector-set! vec i pv)
           (loop p iv))))]))

(define (delete! heap)
  (match heap
    [(binary-heap size vec)
     (cond
       [(= size 0) #f]
       [else
        (define ans (vector-ref vec 0))
        (vector-set! vec 0 (vector-ref vec (- size 1)))
        (set-binary-heap-size! heap (- size 1))
        (let ([size (- size 1)])
          (let loop ([i 0])
            (when (< i size)
              (define v (vector-ref vec i))
              (define li (left-child i))
              (define ri (right-child i))
              (when (< li size)
                (define-values (smaller-child-index smaller-child-val)
                  (cond
                    [(< ri size)
                     (define l (vector-ref vec li))
                     (define r (vector-ref vec ri))
                     (if (<= l r)
                         (values li l)
                         (values ri r))]
                    [else (values li (vector-ref vec li))]))
                (when (< smaller-child-val v)
                  (vector-set! vec smaller-child-index v)
                  (vector-set! vec i smaller-child-val)
                  (loop smaller-child-index))))))
        ans])]))

(define (left-child i) (+ (* i 2) 1))
(define (right-child i) (+ (* i 2) 2))
(define (parent i) (quotient (- i 1) 2))
```

```
(provide
 (contract-out

   [new-heap
    (-> heap/c)]

   [insert!
    (->i ([h heap/c]
          [i integer?])
         [result void?]
         #:post (h) (valid-heap? h))]

   [delete!
    (->i ([h heap/c])
         [res (or/c integer? #f)]
         #:post (h) (valid-heap? h))]))
```

```
#lang racket/base
(require pict pict/tree-layout
         racket/match racket/contract)

(struct binary-heap (size vec) #:mutable #:transparent)
(define (valid-heap? heap)
  (match heap
    [(binary-heap size vec)
     (let loop ([i 0]
                [parent -inf.0])
       (cond
         [(< i size)
          (define this (vector-ref vec i))
          (and (<= parent this)
               (loop (left-child i) this)
               (loop (right-child i) this))]
         [else #t]))]))
(define heap/c (and/c binary-heap? valid-heap?))

(provide
 (contract-out

  [new-heap
   (-> heap/c)]

  [insert!
   (->i ([h heap/c]
         [i integer?])
        [result void?]
        #:post (h) (valid-heap? h))]

  [delete!
   (->i ([h heap/c])
        [res (or/c integer? #f)]
        #:post (h) (valid-heap? h))]))

(define (new-heap) (binary-heap 0 (make-vector 1 #f)))

(define (insert! heap nv)
  (match heap
    [(binary-heap size vec)
     (unless (< size (vector-length vec))
       (define new-vec
         (make-vector (* (vector-length vec) 2) #f))
       (vector-copy! new-vec 0 vec)
       (set-binary-heap-vec! heap new-vec)
       (set! vec new-vec))
     (vector-set! vec size nv)
     (set-binary-heap-size! heap (+ size 1))
     (let loop ([i size] [iv nv])
       (unless (= i 0)
         (define p (parent i))
         (define pv (vector-ref vec p))
         (when (< iv pv)
           (vector-set! vec p iv)
           (vector-set! vec i pv)
           (loop p iv))))]))

(define (delete! heap)
  (match heap
    [(binary-heap size vec)
     (cond
       [(= size 0) #f]
       [else
        (define ans (vector-ref vec 0))
        (vector-set! vec 0 (vector-ref vec (- size 1)))
        (set-binary-heap-size! heap (- size 1))
        (let ([size (- size 1)])
          (let loop ([i 0])
            (when (< i size)
              (define v (vector-ref vec i))
              (define li (left-child i))
              (define ri (right-child i))
              (when (< li size)
                (define-values (smaller-child-index smaller-child-val)
                  (cond
                    [(< ri size)
                     (define l (vector-ref vec li))
                     (define r (vector-ref vec ri))
                     (if (<= l r)
                         (values li l)
                         (values ri r))]
                    [else (values li (vector-ref vec li))]))
                (when (< smaller-child-val v)
                  (vector-set! vec smaller-child-index v)
                  (vector-set! vec i smaller-child-val)
                  (loop smaller-child-index))))))
        ans])]))

(define (left-child i) (+ (* i 2) 1))
(define (right-child i) (+ (* i 2) 2))
(define (parent i) (quotient (- i 1) 2))
```

```
#lang racket/base
(require pict pict/tree-layout
         racket/match racket/contract)

(struct binary-heap (size vec) #:mutable #:transparent)
(define (valid-heap? heap)
  (match heap
    [(binary-heap size vec)
     (let loop ([i 0]
                [parent -inf.0])
       (cond
         [(< i size)
          (define this (vector-ref vec i))
          (and (<= parent this)
               (loop (left-child i) this)
               (loop (right-child i) this))]
         [else #t]))]))
(define heap/c (and/c binary-heap? valid-heap?))

(provide
 (contract-out

  [new-heap
   (-> heap/c)]

  [insert!
   (->i ([h heap/c]
         [i integer?])
        [result void?]
        #:post (h) (valid-heap? h))]

  [delete!
   (->i ([h heap/c])
        [res (or/c integer? #f)]
        #:post (h) (valid-heap? h))]))

(define (new-heap) (binary-heap 0 (make-vector 1 #f)))

(define (insert! heap nv)
  (match heap
    [(binary-heap size vec)
     (unless (< size (vector-length vec))
       (define new-vec
         (make-vector (* (vector-length vec) 2) #f))
       (vector-copy! new-vec 0 vec)
       (set-binary-heap-vec! heap new-vec)
       (set! vec new-vec))
     (vector-set! vec size nv)
     (set-binary-heap-size! heap (+ size 1))
     (let loop ([i size] [iv nv])
       (unless (= i 0)
         (define p (parent i))
         (define pv (vector-ref vec p))
         (when (< iv pv)
           (vector-set! vec p iv)
           (vector-set! vec i pv)
           (loop p iv))))]))

(define (delete! heap)
  (match heap
    [(binary-heap size vec)
     (cond
       [(= size 0) #f]
       [else
        (define ans (vector-ref vec 0))
        (vector-set! vec 0 (vector-ref vec (- size 1)))
        (set-binary-heap-size! heap (- size 1))
        (let ([size (- size 1)])
          (let loop ([i 0])
            (when (< i size)
              (define v (vector-ref vec i))
              (define li (left-child i))
              (define ri (right-child i))
              (when (< li size)
                (define-values (smaller-child-index smaller-child-val)
                  (cond
                    [(< ri size)
                     (define l (vector-ref vec li))
                     (define r (vector-ref vec ri))
                     (if (<= l r)
                         (values li l)
                         (values ri r))]
                    [else (values li (vector-ref vec li))]))
                (when (< smaller-child-val v)
                  (vector-set! vec smaller-child-index v)
                  (vector-set! vec i smaller-child-val)
                  (loop smaller-child-index))))))
        ans])]))

(define (left-child i) (+ (* i 2) 1))
(define (right-child i) (+ (* i 2) 2))
(define (parent i) (quotient (- i 1) 2))
```

```
> (contract-exercise new-heap insert! delete!)
```

```
> (contract-exercise new-heap insert! delete!)
```

... nothing happens

```
#lang racket/base
(require pict pict/tree-layout
         racket/match racket/contract)

(struct binary-heap (size vec) #:mutable #:transparent)
(define (valid-heap? heap)
  (match heap
    [(binary-heap size vec)
     (let loop ([i 0]
                [parent -inf.0])
       (cond
         [(< i size)
          (define this (vector-ref vec i))
          (and (<= parent this)
               (loop (left-child i) this)
               (loop (right-child i) this))]
         [else #t]))]))
(define heap/c (and/c binary-heap? valid-heap?))

(provide
 (contract-out

  [new-heap
   (-> heap/c)]

  [insert!
   (->i ([h heap/c]
         [i integer?])
        [result void?]
        #:post (h) (valid-heap? h))]

  [delete!
   (->i ([h heap/c])
        [res (or/c integer? #f)]
        #:post (h) (valid-heap? h))]))

(define (new-heap) (binary-heap 0 (make-vector 1 #f)))

(define (insert! heap nv)
  (match heap
    [(binary-heap size vec)
     (unless (< size (vector-length vec))
       (define new-vec
         (make-vector (* (vector-length vec) 2) #f))
       (vector-copy! new-vec 0 vec)
       (set-binary-heap-vec! heap new-vec)
       (set! vec new-vec))
     (vector-set! vec size nv)
     (set-binary-heap-size! heap (+ size 1))
     (let loop ([i size] [iv nv])
       (unless (= i 0)
         (define p (parent i))
         (define pv (vector-ref vec p))
         (when (< iv pv)
           (vector-set! vec p iv)
           (vector-set! vec i pv)
           (loop p iv))))]))

(define (delete! heap)
  (match heap
    [(binary-heap size vec)
     (cond
       [(= size 0) #f]
       [else
        (define ans (vector-ref vec 0))
        (vector-set! vec 0 (vector-ref vec (- size 1)))
        (set-binary-heap-size! heap (- size 1))
        (let ([size (- size 1)])
          (let loop ([i 0])
            (when (< i size)
              (define v (vector-ref vec i))
              (define li (left-child i))
              (define ri (right-child i))
              (when (< li size)
                (define-values (smaller-child-index smaller-child-val)
                  (cond
                    [(< ri size)
                     (define l (vector-ref vec li))
                     (define r (vector-ref vec ri))
                     (if (<= l r)
                         (values li l)
                         (values ri r))]
                    [else (values li (vector-ref vec li))]))
                (when (< smaller-child-val v)
                  (vector-set! vec smaller-child-index v)
                  (vector-set! vec i smaller-child-val)
                  (loop smaller-child-index))))))
        ans])]))

(define (left-child i) (+ (* i 2) 1))
(define (right-child i) (+ (* i 2) 2))
(define (parent i) (quotient (- i 1) 2))
```

```
#lang racket/base
(require pict pict/tree-layout
         racket/match racket/contract)

(struct binary-heap (size vec) #:mutable #:transparent)
(define (valid-heap? heap)
  (match heap
    [(binary-heap size vec)
     (let loop ([i 0]
                [parent -inf.0])
       (cond
         [(< i size)
          (define this (vector-ref vec i))
          (and (<= parent this)
               (loop (left-child i) this)
               (loop (right-child i) this))]
         [else #t]))]))
(define heap/c (and/c binary-heap? valid-heap?))

(provide
 (contract-out

  [new-heap
   (-> heap/c)]

  [insert!
   (->i ([h heap/c]
         [i integer?])
        [result void?]
        #:post (h) (valid-heap? h))]

  [delete!
   (->i ([h heap/c])
        [res (or/c integer? #f)]
        #:post (h) (valid-heap? h))]))

(define (new-heap) (binary-heap 0 (make-vector 1 #f)))

(define (insert! heap nv)
  (match heap
    [(binary-heap size vec)
     (unless (< size (vector-length vec))
       (define new-vec
         (make-vector (* (vector-length vec) 2) #f))
       (vector-copy! new-vec 0 vec)
       (set-binary-heap-vec! heap new-vec)
       (set! vec new-vec))
     (vector-set! vec size nv)
     (set-binary-heap-size! heap (+ size 1))
     (let loop ([i size] [iv nv])
       (unless (= i 0)
         (define p (parent i))
         (define pv (vector-ref vec p))
         (when (< iv pv)
           (vector-set! vec p iv)
           (vector-set! vec i pv)
           (loop p iv))))]))

(define (delete! heap)
  (match heap
    [(binary-heap size vec)
     (cond
       [(= size 0) #f]
       [else
        (define ans (vector-ref vec 0))
        (vector-set! vec 0 (vector-ref vec (- size 1)))
        (set-binary-heap-size! heap (- size 1))
        (let ([size (- size 1)])
          (let loop ([i 0])
            (when (< i size)
              (define v (vector-ref vec i))
              (define li (left-child i))
              (define ri (right-child i))
              (when (< li size)
                (define-values (smaller-child-index smaller-child-val)
                  (cond
                    [(< ri size)
                     (define l (vector-ref vec li))
                     (define r (vector-ref vec ri))
                     (if (<= l r)
                         (values li l)
                         (values ri r))]
                    [else (values li (vector-ref vec li))]))
                (when (< smaller-child-val v)
                  (vector-set! vec smaller-child-index v)
                  (vector-set! vec i smaller-child-val)
                  (loop smaller-child-index))))))
        ans])]))

(define (left-child i) (+ (* i 2) 1))
(define (right-child i) (+ (* i 2) 2))
(define (parent i) (quotient (- i 1) 2))
```

```
> (contract-exercise new-heap insert! delete!)
insert!: broke its own contract
  #:post condition violation; variables are:
      h: (binary-heap 2 '#(2147483647 -7))
  in: (->i
       ((h (and/c binary-heap? valid-heap?))
        (i integer?))
       (result void?)
       #:post
       (h)
       (valid-heap? h))
  contract from: heap
  blaming: heap
   (assuming the contract is correct)
```

```
#lang racket/base
(require pict pict/tree-layout
         racket/match racket/contract)

(struct binary-heap (size vec) #:mutable #:transparent)
(define (valid-heap? heap)
  (match heap
    [(binary-heap size vec)
     (let loop ([i 0]
                [parent -inf.0])
       (cond
         [(< i size)
          (define this (vector-ref vec i))
          (and (<= parent this)
               (loop (left-child i) this)
               (loop (right-child i) this))]
         [else #t]))]))
(define heap/c (and/c binary-heap? valid-heap?))

(provide
 (contract-out

  [new-heap
   (-> heap/c)]

  [insert!
   (->i ([h heap/c]
         [i integer?])
        [result void?]
        #:post (h) (valid-heap? h))]

  [delete!
   (->i ([h heap/c])
        [res (or/c integer? #f)]
        #:post (h) (valid-heap? h))]))

(define (new-heap) (binary-heap 0 (make-vector 1 #f)))

(define (insert! heap nv)
  (match heap
    [(binary-heap size vec)
     (unless (< size (vector-length vec))
       (define new-vec
         (make-vector (* (vector-length vec) 2) #f))
       (vector-copy! new-vec 0 vec)
       (set-binary-heap-vec! heap new-vec)
       (set! vec new-vec))
     (vector-set! vec size nv)
     (set-binary-heap-size! heap (+ size 1))
     (let loop ([i size] [iv nv])
       (unless (= i 0)
         (define p (parent i))
         (define pv (vector-ref vec p))
         (when (< iv pv)
           (vector-set! vec p iv)
           (vector-set! vec i pv)
           (loop p iv))))]))

(define (delete! heap)
  (match heap
    [(binary-heap size vec)
     (cond
       [(= size 0) #f]
       [else
        (define ans (vector-ref vec 0))
        (vector-set! vec 0 (vector-ref vec (- size 1)))
        (set-binary-heap-size! heap (- size 1))
        (let ([size (- size 1)])
          (let loop ([i 0])
            (when (< i size)
              (define v (vector-ref vec i))
              (define li (left-child i))
              (define ri (right-child i))
              (when (< li size)
                (define-values (smaller-child-index smaller-child-val)
                  (cond
                    [(< ri size)
                     (define l (vector-ref vec li))
                     (define r (vector-ref vec ri))
                     (if (<= l r)
                         (values li l)
                         (values ri r))]
                    [else (values li (vector-ref vec li))]))
                (when (< smaller-child-val v)
                  (vector-set! vec smaller-child-index v)
                  (vector-set! vec i smaller-child-val)
                  (loop smaller-child-index))))))
        ans])]))

(define (left-child i) (+ (* i 2) 1))
(define (right-child i) (+ (* i 2) 2))
(define (parent i) (quotient (- i 1) 2))
```

```
#lang racket/base
(require pict pict/tree-layout
         racket/match racket/contract)

(struct binary-heap (size vec) #:mutable #:transparent)
(define (valid-heap? heap)
  (match heap
    [(binary-heap size vec)
     (let loop ([i 0]
                [parent -inf.0])
       (cond
         [(< i size)
          (define this (vector-ref vec i))
          (and (<= parent this)
               (loop (left-child i) this)
               (loop (right-child i) this))]
         [else #t]))]))
(define heap/c (and/c binary-heap? valid-heap?))

(provide
 (contract-out

  [new-heap
   (-> heap/c)]

  [insert!
   (->i ([h heap/c]
         [i integer?])
        [result void?]
        #:post (h) (valid-heap? h))]

  [delete!
   (->i ([h heap/c])
        [res (or/c integer? #f)]
        #:post (h) (valid-heap? h))]))

(define (new-heap) (binary-heap 0 (make-vector 1 #f)))

(define (insert! heap nv)
  (match heap
    [(binary-heap size vec)
     (unless (< size (vector-length vec))
       (define new-vec
         (make-vector (* (vector-length vec) 2) #f))
       (vector-copy! new-vec 0 vec)
       (set-binary-heap-vec! heap new-vec)
       (set! vec new-vec))
     (vector-set! vec size nv)
     (set-binary-heap-size! heap (+ size 1))
     (let loop ([i size] [iv nv])
       (unless (= i 0)
         (define p (parent i))
         (define pv (vector-ref vec p))
         (when (< iv pv)
           (vector-set! vec p iv)
           (vector-set! vec i pv)
           (loop p iv))))]))

(define (delete! heap)
  (match heap
    [(binary-heap size vec)
     (cond
       [(= size 0) #f]
       [else
        (define ans (vector-ref vec 0))
        (vector-set! vec 0 (vector-ref vec (- size 1)))
        (set-binary-heap-size! heap (- size 1))
        (let ([size (- size 1)])
          (let loop ([i 0])
            (when (< i size)
              (define v (vector-ref vec i))
              (define li (left-child i))
              (define ri (right-child i))
              (when (< li size)
                (define-values (smaller-child-index smaller-child-val)
                  (cond
                    [(< ri size)
                     (define l (vector-ref vec li))
                     (define r (vector-ref vec ri))
                     (if (<= l r)
                         (values li l)
                         (values ri r))]
                    [else (values li (vector-ref vec li))]))
                (when (< smaller-child-val v)
                  (vector-set! vec smaller-child-index v)
                  (vector-set! vec i smaller-child-val)
                  (loop smaller-child-index))))))
        ans])]))

(define (left-child i) (+ (* i 2) 1))
(define (right-child i) (+ (* i 2) 2))
(define (parent i) (quotient (- i 1) 2))
```

```
> (contract-exercise new-heap insert! delete!)
delete!: broke its own contract
  #:post condition violation; variables are:
      h: (binary-heap 3 '#(349065506 28.0 ...
  in: (->i
        ((h (and/c binary-heap? valid-heap?)))
        (res (or/c integer? #f))
        #:post
        (h)
        (valid-heap? h))
  contract from: heap
  blaming: heap
   (assuming the contract is correct)
```

```
> (contract-exercise new-heap insert! delete!)
delete!: broke its own contract
  #:post condition violation; variables are:
      h: (binary-heap 3 '#(1018653970 1242...
  in: (->i
        ((h (and/c binary-heap? valid-heap?)))
        (res (or/c integer? #f))
        #:post
        (h)
        (valid-heap? h))
  contract from: heap
  blaming: heap
   (assuming the contract is correct)
```

```
> (contract-exercise new-heap insert! delete!)
delete!: broke its own contract
  #:post condition violation; variables are:
      h: (binary-heap 2 '#(-267600327.0 -1...
  in: (->i
        ((h (and/c binary-heap? valid-heap?)))
        (res (or/c integer? #f))
        #:post
        (h)
        (valid-heap? h))
  contract from: heap
  blaming: heap
   (assuming the contract is correct)
```

```
> (contract-exercise new-heap insert! delete!)
delete!: broke its own contract
  #:post condition violation; variables are:
      h: (binary-heap 2 '#(728834549.0 631...
  in: (->i
        ((h (and/c binary-heap? valid-heap?)))
        (res (or/c integer? #f))
        #:post
        (h)
        (valid-heap? h))
  contract from: heap
  blaming: heap
   (assuming the contract is correct)
```

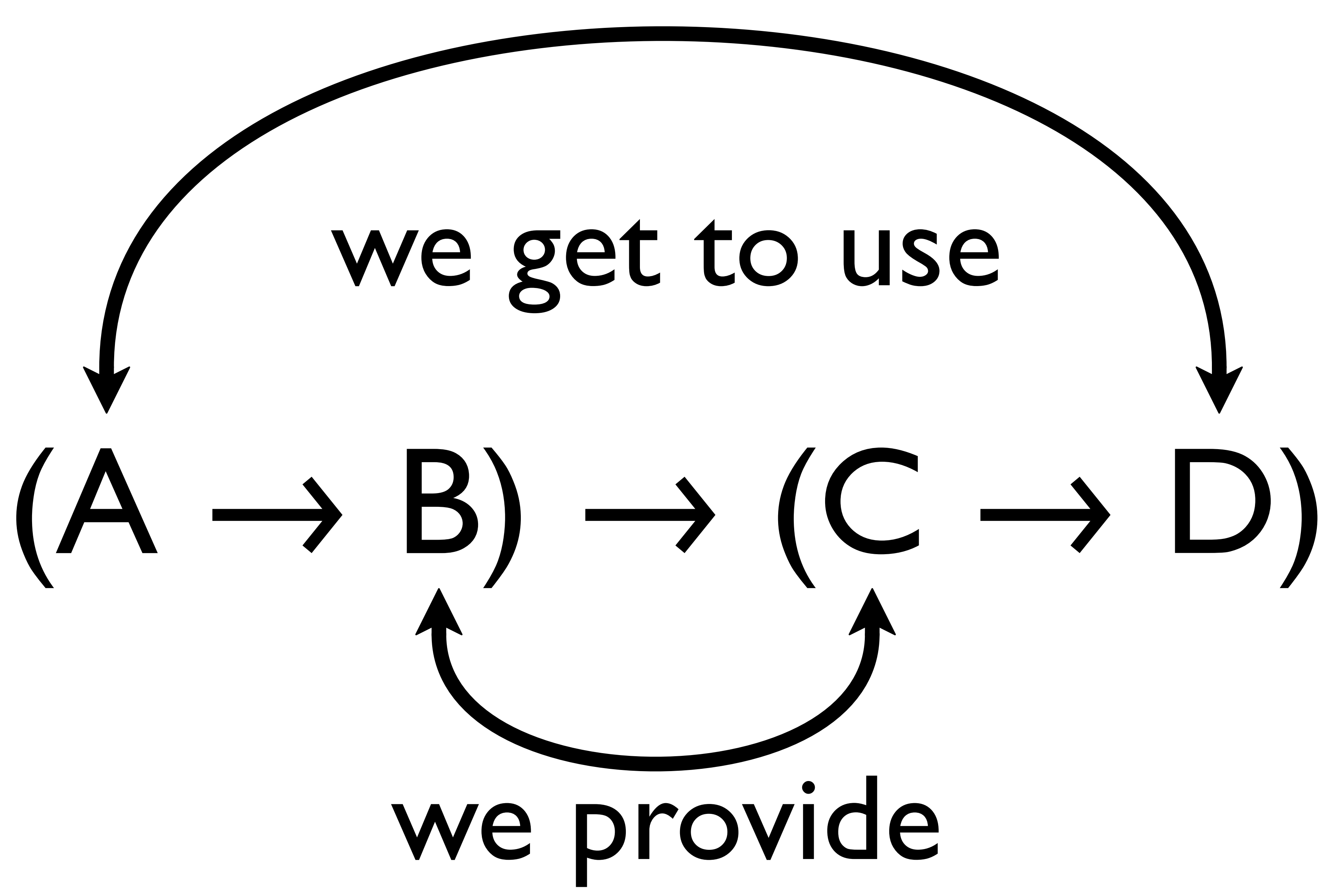
```
> (contract-exercise new-heap insert! delete!)
delete!: broke its own contract
  #:post condition violation; variables are:
      h: (binary-heap 3 '#(-180.0 -1343510...
  in: (->i
        ((h (and/c binary-heap? valid-heap?)))
        (res (or/c integer? #f))
        #:post
        (h)
        (valid-heap? h))
  contract from: heap
  blaming: heap
   (assuming the contract is correct)
```

```
#lang racket/base
(require pict pict/tree-layout
         racket/match racket/contract)

(struct binary-heap (size vec) #:mutable #:transparent)
(define (valid-heap? heap)
  (match heap
    [(binary-heap size vec)
     (let loop ([i 0]
                [parent -inf.0])
       (cond
         [(< i size)
          (define this (vector-ref vec i))
          (and (<= parent this)
               (loop (left-child i) this)
               (loop (right-child i) this))]
         [else #t]))]))
(define heap/c (and/c binary-heap? valid-heap?))

(provide
 (contract-out

  [new-heap
   (-> heap/c)]

  [insert!
   (->i ([h heap/c]
         [i integer?])
        [result void?]
        #:post (h) (valid-heap? h))]

  [delete!
   (->i ([h heap/c])
        [res (or/c integer? #f)]
        #:post (h) (valid-heap? h))]))

(define (new-heap) (binary-heap 0 (make-vector 1 #f)))

(define (insert! heap nv)
  (match heap
    [(binary-heap size vec)
     (unless (< size (vector-length vec))
       (define new-vec
         (make-vector (* (vector-length vec) 2) #f))
       (vector-copy! new-vec 0 vec)
       (set-binary-heap-vec! heap new-vec)
       (set! vec new-vec))
     (vector-set! vec size nv)
     (set-binary-heap-size! heap (+ size 1))
     (let loop ([i size] [iv nv])
       (unless (= i 0)
         (define p (parent i))
         (define pv (vector-ref vec p))
         (when (< iv pv)
           (vector-set! vec p iv)
           (vector-set! vec i pv)
           (loop p iv))))]))

(define (delete! heap)
  (match heap
    [(binary-heap size vec)
     (cond
       [(= size 0) #f]
       [else
        (define ans (vector-ref vec 0))
        (vector-set! vec 0 (vector-ref vec (- size 1)))
        (set-binary-heap-size! heap (- size 1))
        (let ([size (- size 1)])
          (let loop ([i 0])
            (when (< i size)
              (define v (vector-ref vec i))
              (define li (left-child i))
              (define ri (right-child i))
              (when (< li size)
                (define-values (smaller-child-index smaller-child-val)
                  (cond
                    [(< ri size)
                     (define l (vector-ref vec li))
                     (define r (vector-ref vec ri))
                     (if (<= l r)
                         (values li l)
                         (values ri r))]
                    [else (values li (vector-ref vec li))]))
                (when (< smaller-child-val v)
                  (vector-set! vec smaller-child-index v)
                  (vector-set! vec i smaller-child-val)
                  (loop smaller-child-index))))))
        ans])]))

(define (left-child i) (+ (* i 2) 1))
(define (right-child i) (+ (* i 2) 2))
(define (parent i) (quotient (- i 1) 2))
```

$$(A \to B) \to (C \to D)$$

$$(A \to B) \to (C \to D)$$

we provide


we get to use
(A → B) → (C → D)
we provide

```
(provide
 (contract-out

  [new-heap
   (-> heap/c)]

  [insert!
   (->i ([h heap/c]
         [i integer?])
        [result void?]
        #:post (h) (valid-heap? h))]

  [delete!
   (->i ([h heap/c])
        [res (or/c integer? #f)]
        #:post (h) (valid-heap? h))]))
```

Last Thoughts

Contracts: more & less than types

Boundaries+Blame: speed debugging

Specifications: lots of use

Last Thoughts

Contracts: complement types

Boundaries+Blame: speed debugging

Specifications: lots of use