

# a shallow Scheme embedding of $\perp$ -avoiding streams

William E. Byrd, Daniel P. Friedman,  
Ramana Kumar, Joseph P. Near

$(\text{car } (\text{cons } \perp (\text{cons } 1 '())))$

$=$

$\perp$

$(\text{car}_{\perp} (\text{cons}_{\perp} \perp (\text{cons}_{\perp} 1 '())))$

$=$

1

convergent elements form the prefix in  
some order

divergent elements form the suffix

once determined, the ordering is fixed

examples

```
(define or⊥
  ( $\lambda_t$  (s)
    (cond
      ((null? s) #f)
      ((car⊥ s) (car⊥ s))
      (else (or⊥ (cdr⊥ s))))))
```

```
(define or ⊥  
  (λt (s)  
    (cond  
      ((null? s) #f)  
      ((car ⊥ s) (car ⊥ s))  
      (else (or ⊥ (cdr ⊥ s))))))
```

```
(or ⊥ (list ⊥ ⊥ #t ⊥ ⊥ ⊥)) ⇒ #t
```

```

(define or⊥
  (λt (s)
    (cond
      ((null? s) #f)
      ((car⊥ s) (car⊥ s))
      (else (or⊥ (cdr⊥ s))))))

```

$(or_{\perp} (\mathbf{list}_{\perp} \perp \perp \#t \perp \perp \perp)) \Rightarrow \#t$

$(\mathbf{let} ((s (\mathbf{list}_{\perp} \alpha \beta \dots))))$   
 $(eq? (or_{\perp} s) (or_{\perp} s)) \Rightarrow \#t$

```
(define map⊥
  ( $\lambda_t$  (f ls)
    (cond
      ((null? ls) '())
      (else (cons⊥ (f (car⊥ ls)) (map⊥ f (cdr⊥ ls)))))))
```

```
(define map⊥
  (λt (f ls)
    (cond
      ((null? ls) '())
      (else (cons⊥ (f (car⊥ ls)) (map⊥ f (cdr⊥ ls)))))))
```

```
(take⊥ 2 (map⊥ add1 (list⊥ ⊥ 2 ⊥ 3 ⊥))) ⇒ (3 4)
```

```
(define append⊥
  ( $\lambda_t$  (s1 s2)
    (cond
      ((null? s1) s2)
      (else (cons⊥ (car⊥ s1) (append⊥ (cdr⊥ s1) s2))))))
```

```
(define append⊥
  (λt (s1 s2)
    (cond
      ((null? s1) s2)
      (else (cons⊥ (car⊥ s1) (append⊥ (cdr⊥ s1) s2))))))
```

$(take_{\perp} 2 (append_{\perp} (list_{\perp} 1) (list_{\perp} \perp 2))) \Rightarrow (1 2)$

```

(define append⊥
  (λt (s1 s2)
    (cond
      ((null? s1) s2)
      (else (cons⊥ (car⊥ s1) (append⊥ (cdr⊥ s1) s2))))))

```

$(take_{\perp} 2 (append_{\perp} (list_{\perp} 1) (list_{\perp} \perp 2))) \Rightarrow (1 2)$

$(take_{\perp} 2 (append_{\perp} (list_{\perp} \perp 1) (list_{\perp} 2))) \Rightarrow \perp$

# logic combinators

$\alpha \vee \beta$

$\perp \vee \beta$

(disj  $\perp$   $\beta$ )

$(\text{disj } \perp (\equiv \alpha \beta))$

$(\text{conj} (\text{disj } \perp (\equiv \alpha \beta)) \gamma)$

goal:  $\sigma \rightarrow [\sigma]_{\perp}$

$\sigma: \{ v \mapsto t \}$

```
(define-syntax disj⊥
  (syntax-rules ()
    ((- g1 g2) (λt (σ) (mplus⊥ (g1 σ) (g2 σ))))))
```

```
(define-syntax conj⊥
  (syntax-rules ()
    ((- g1 g2) (λt (σ) (bind⊥ (g1 σ) g2))))))
```

```
(define-syntax disj⊥
  (syntax-rules ()
    ((- g1 g2) (λt (σ) (mplus⊥ (g1 σ) (g2 σ))))))
```

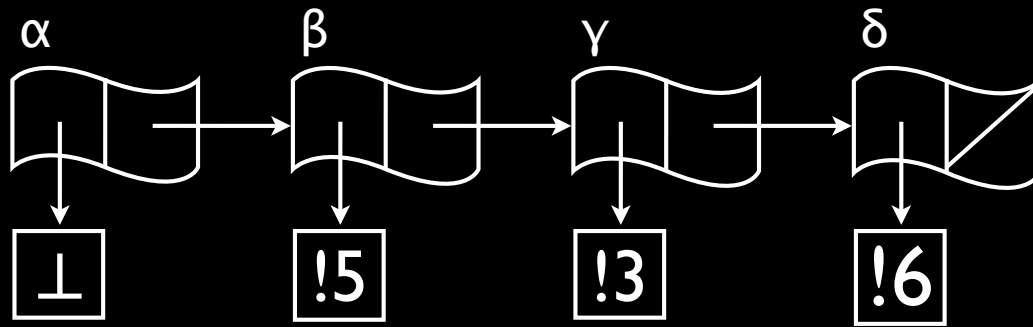
```
(define-syntax conj⊥
  (syntax-rules ()
    ((- g1 g2) (λt (σ) (bind⊥ (g1 σ) g2))))))
```

```
(run⊥ 1 (disj⊥ ⊥ (≡⊥ x 3))) ⇒ ({x/3})
```

promotion

```
(let (( $\delta$  (cons⊥ (! 6) '())))  
  (let (( $\gamma$  (cons⊥ (! 3)  $\delta$ )))  
    (let (( $\beta$  (cons⊥ (! 5)  $\gamma$ )))  
      (let (( $\alpha$  (cons⊥ ⊥  $\beta$ )))  
        (take⊥ 3  $\alpha$ ))))))
```

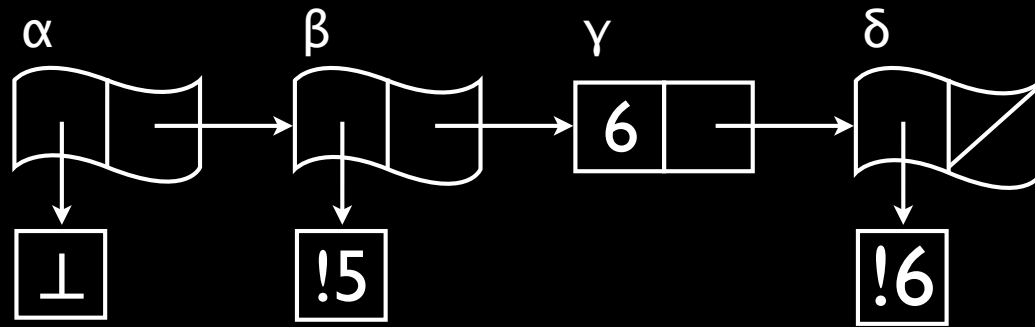
```
(let ((δ (cons⊥ (! 6) '())))  
  (let ((γ (cons⊥ (! 3) δ)))  
    (let ((β (cons⊥ (! 5) γ)))  
      (let ((α (cons⊥ ⊥ β)))  
        (take⊥ 3 α))))))
```



```

(let ((δ (cons⊥ (! 6) '())))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

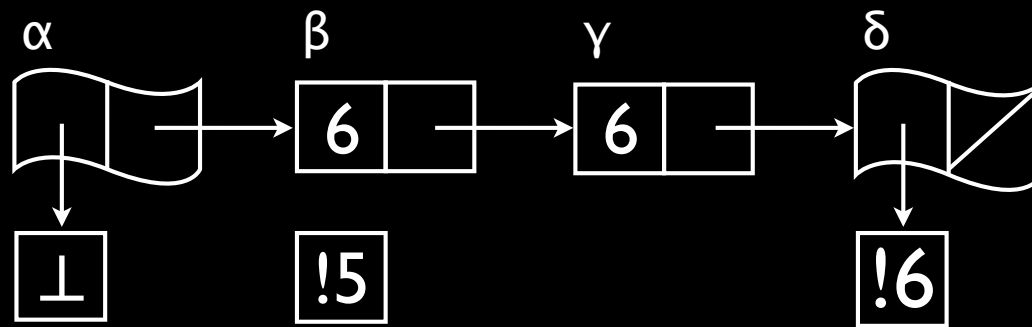
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

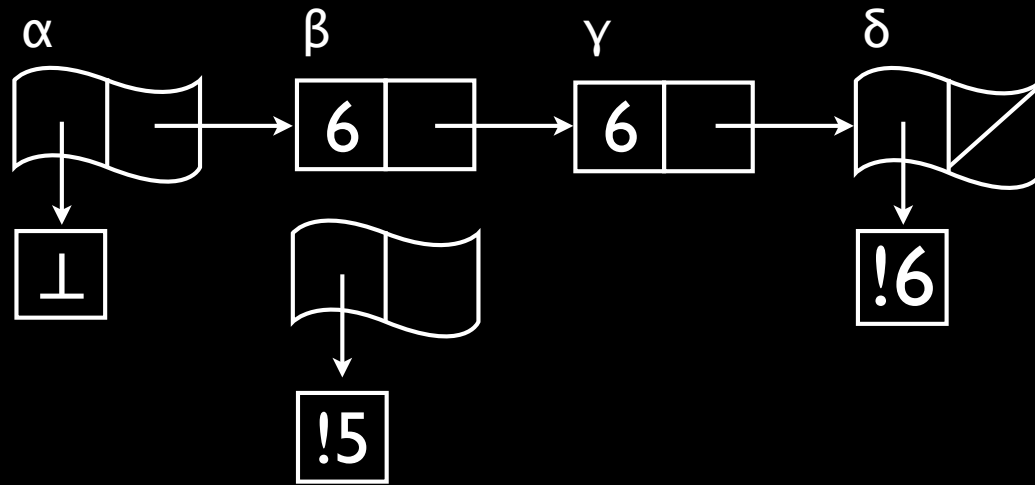
```



```

(let ((δ (cons⊥ (! 6) '())))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

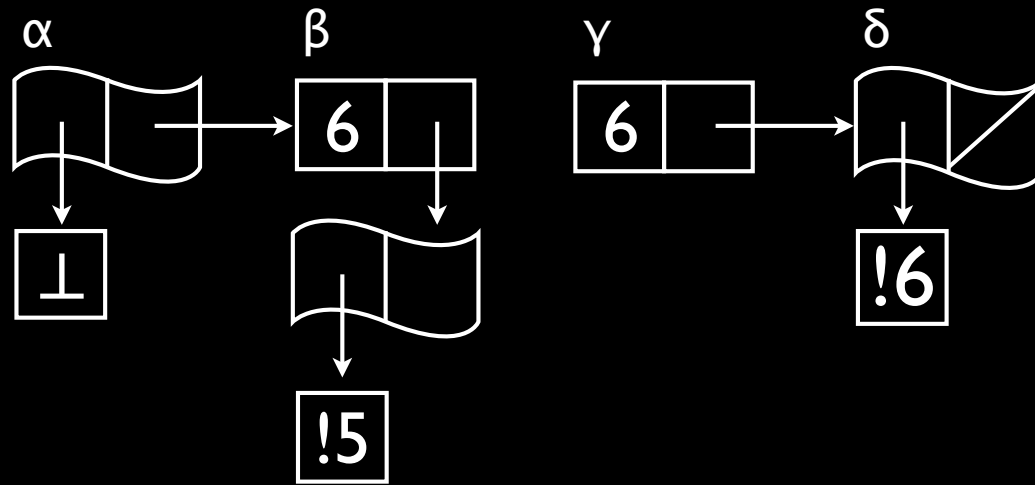
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

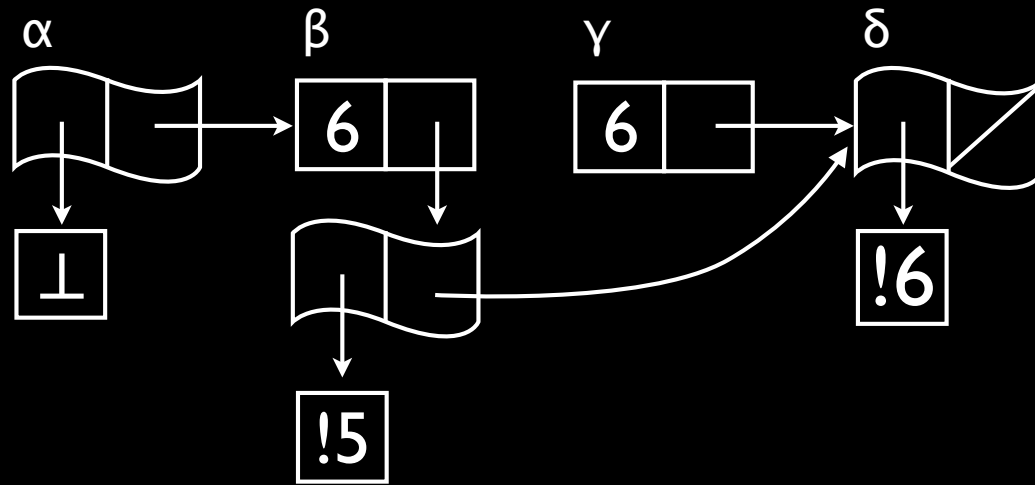
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

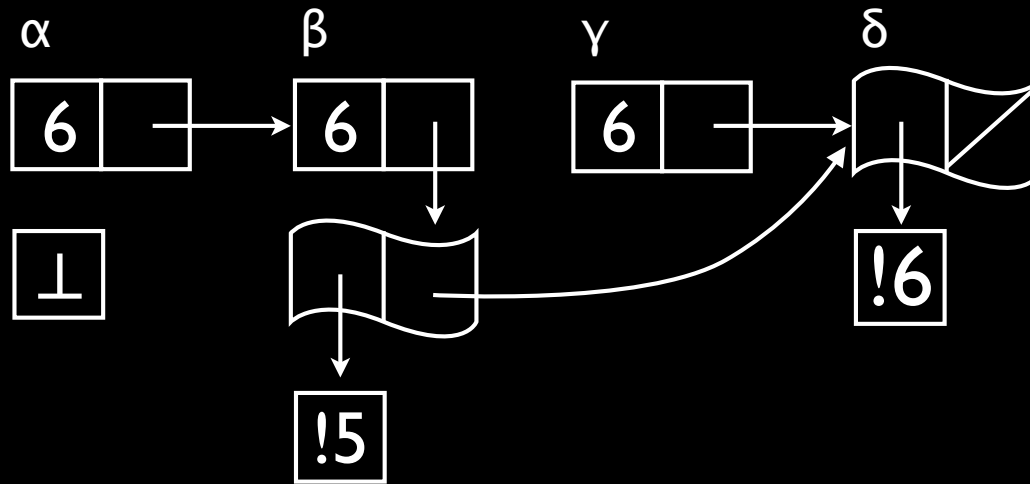
```



```

(let ((δ (cons⊥ (! 6) '())))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

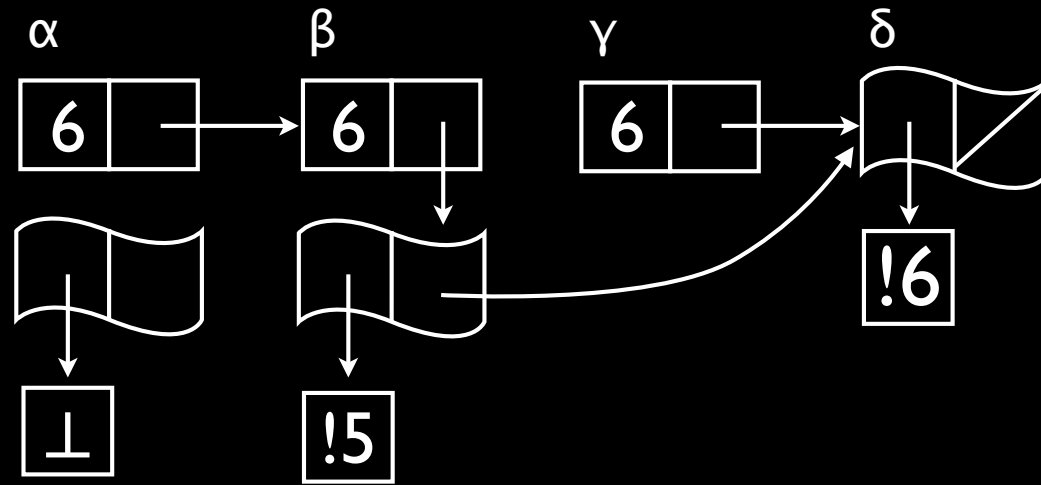
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

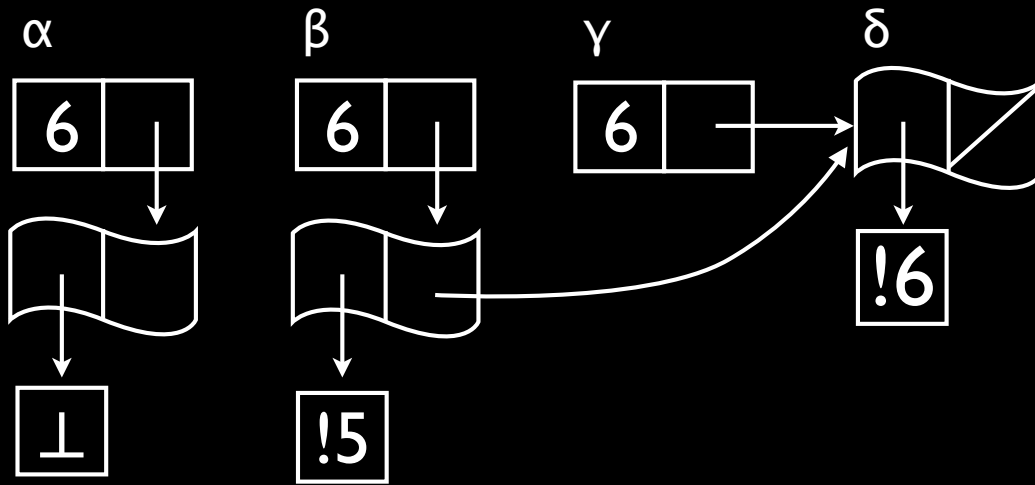
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

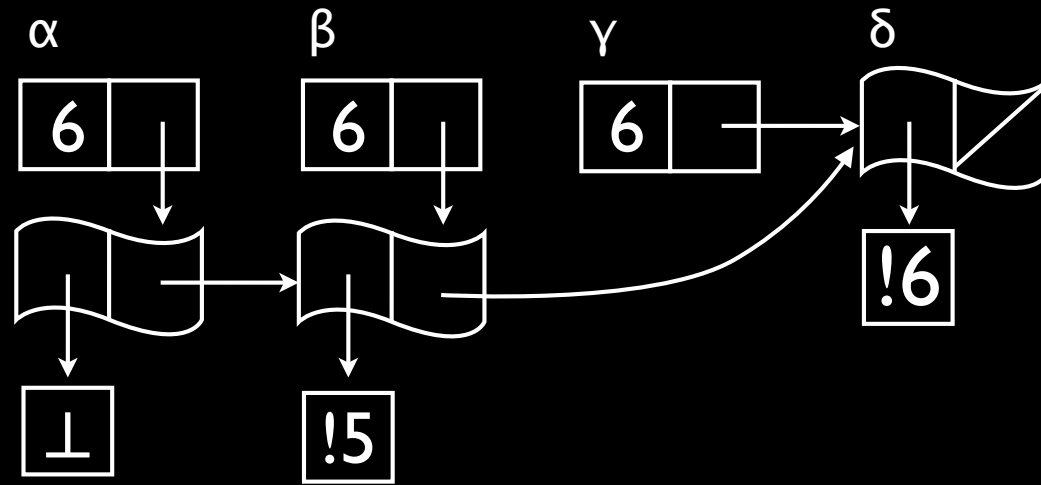
```



```

(let ((δ (cons⊥ (! 6) '())))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

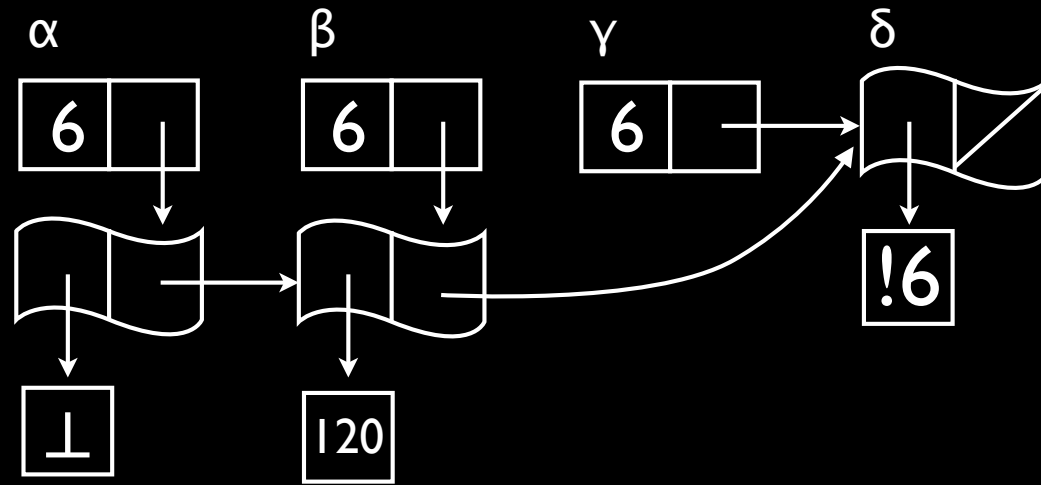
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

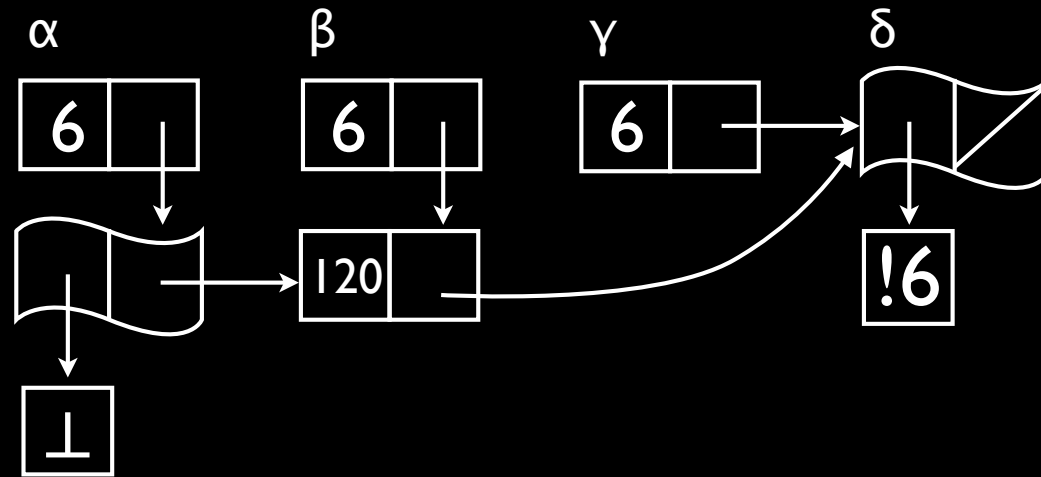
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

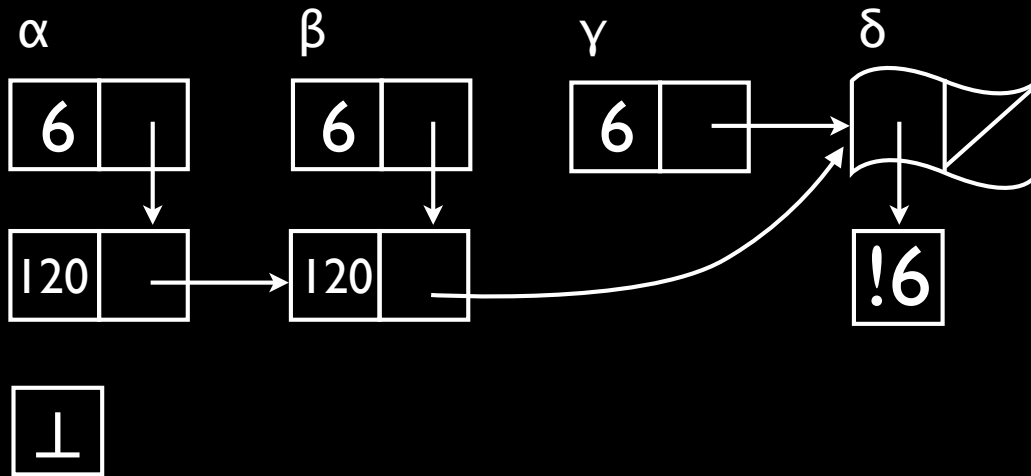
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

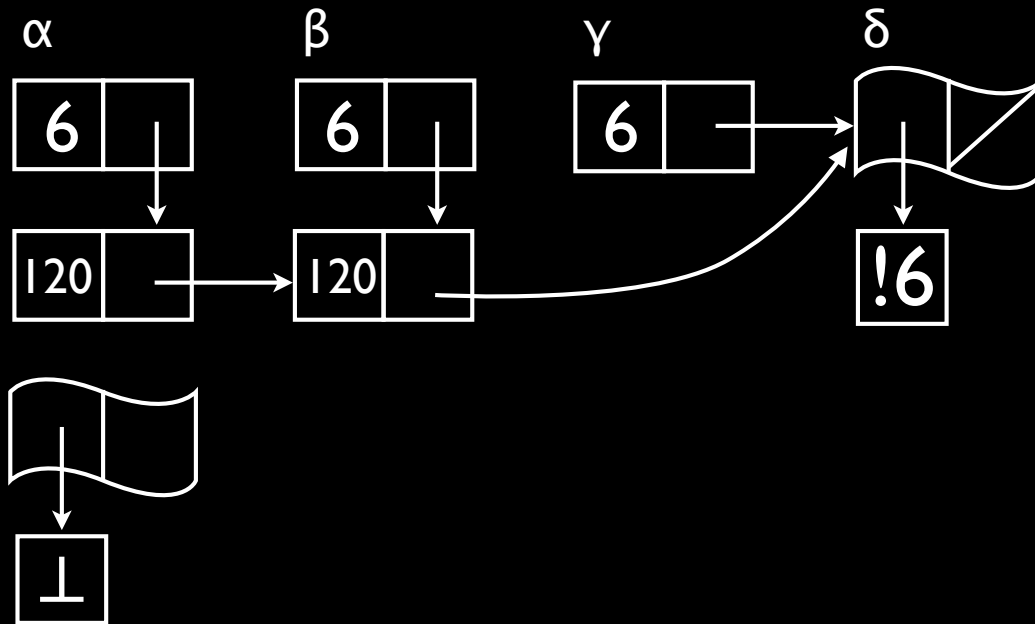
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

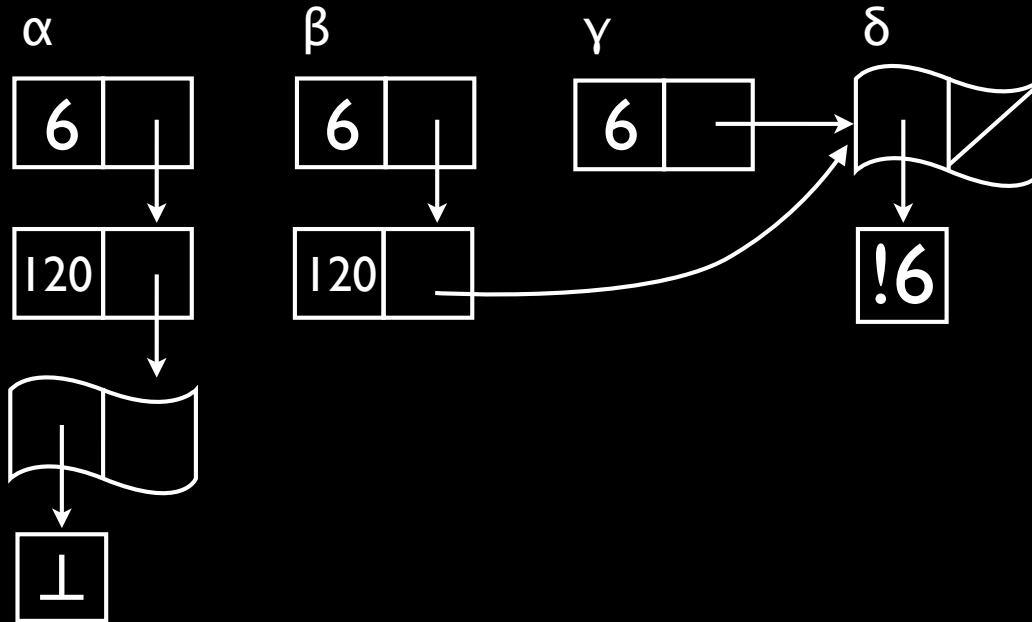
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

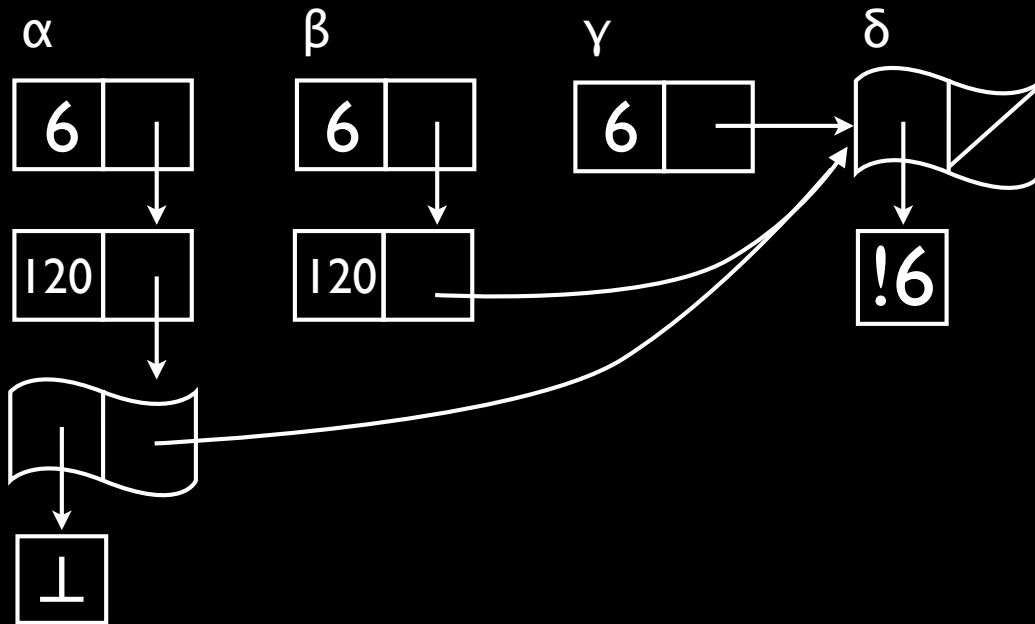
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

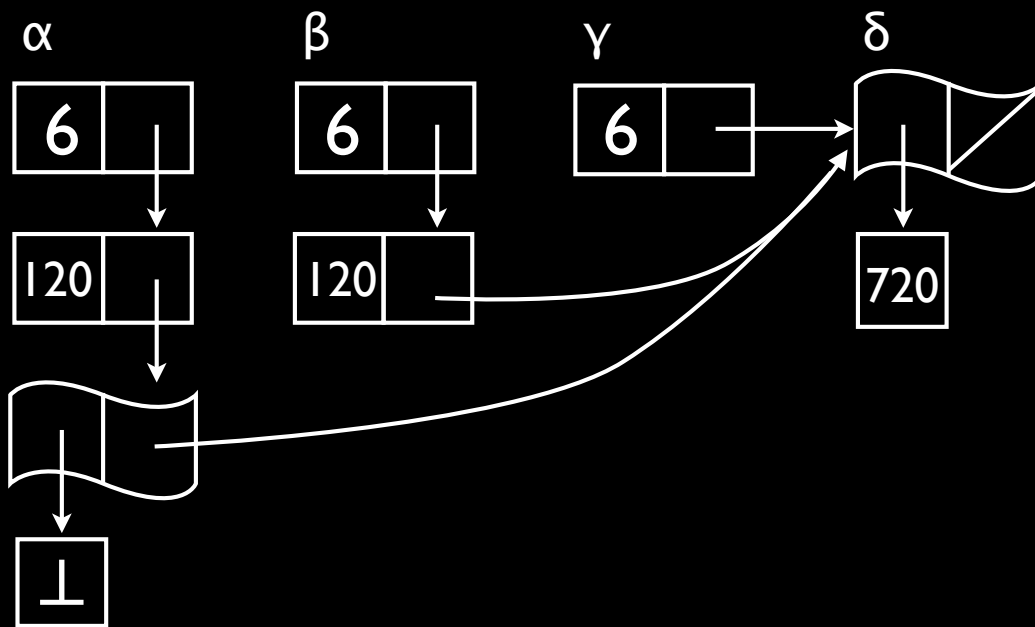
```



```

(let ((δ (cons⊥ (! 6) '())))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

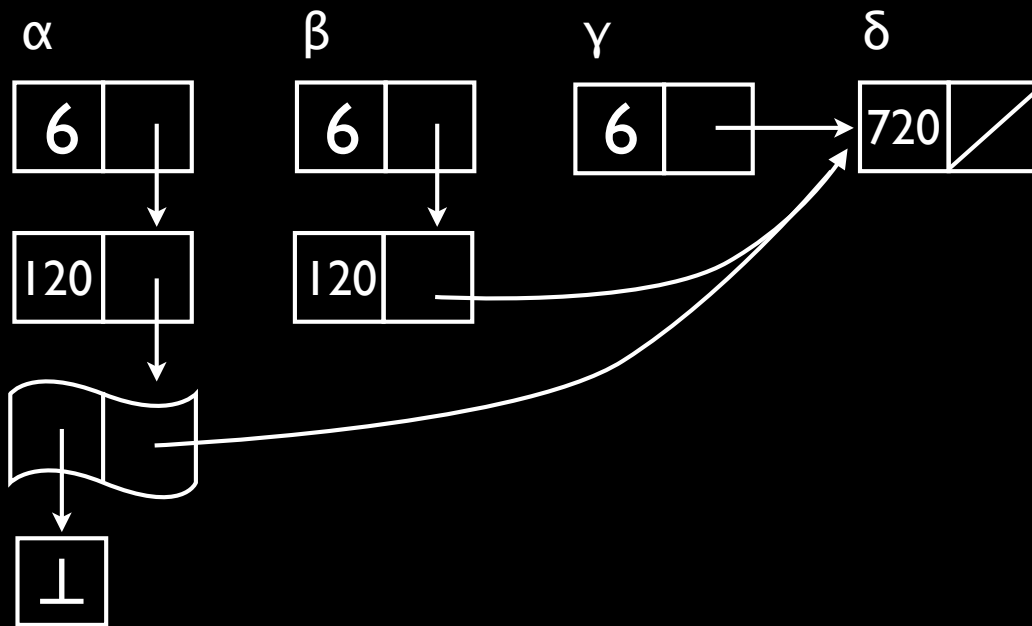
```



```

(let ((δ (cons⊥ (! 6) '()))
      (γ (cons⊥ (! 3) δ))
      (β (cons⊥ (! 5) γ))
      (α (cons⊥ ⊥ β)))
  (take⊥ 3 α))

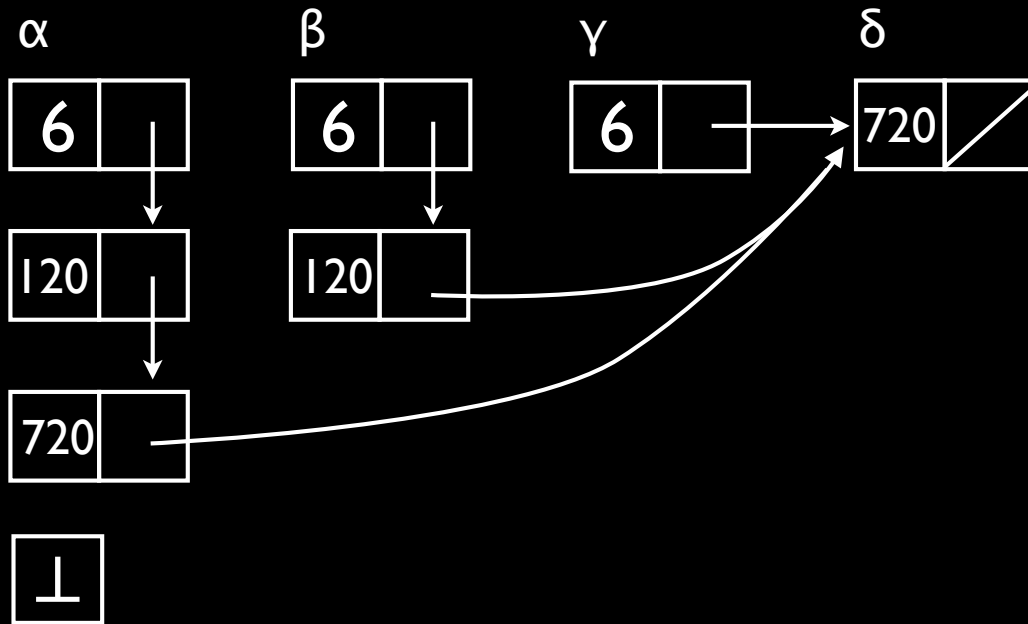
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

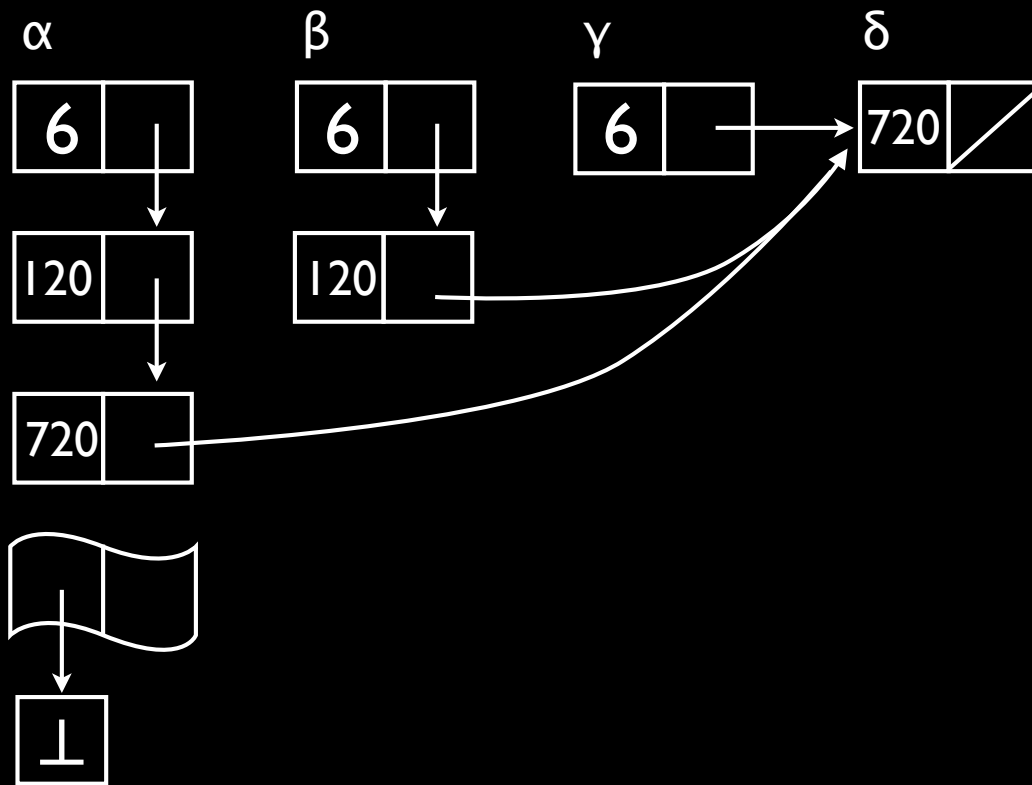
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

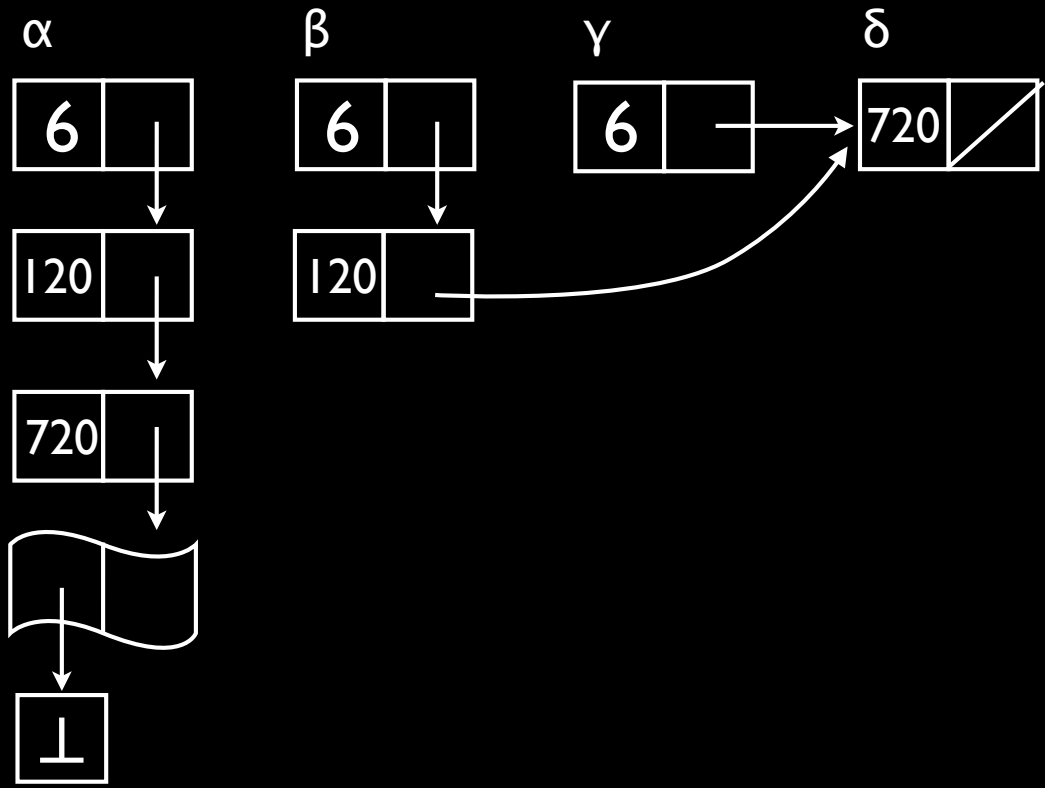
```



```

(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

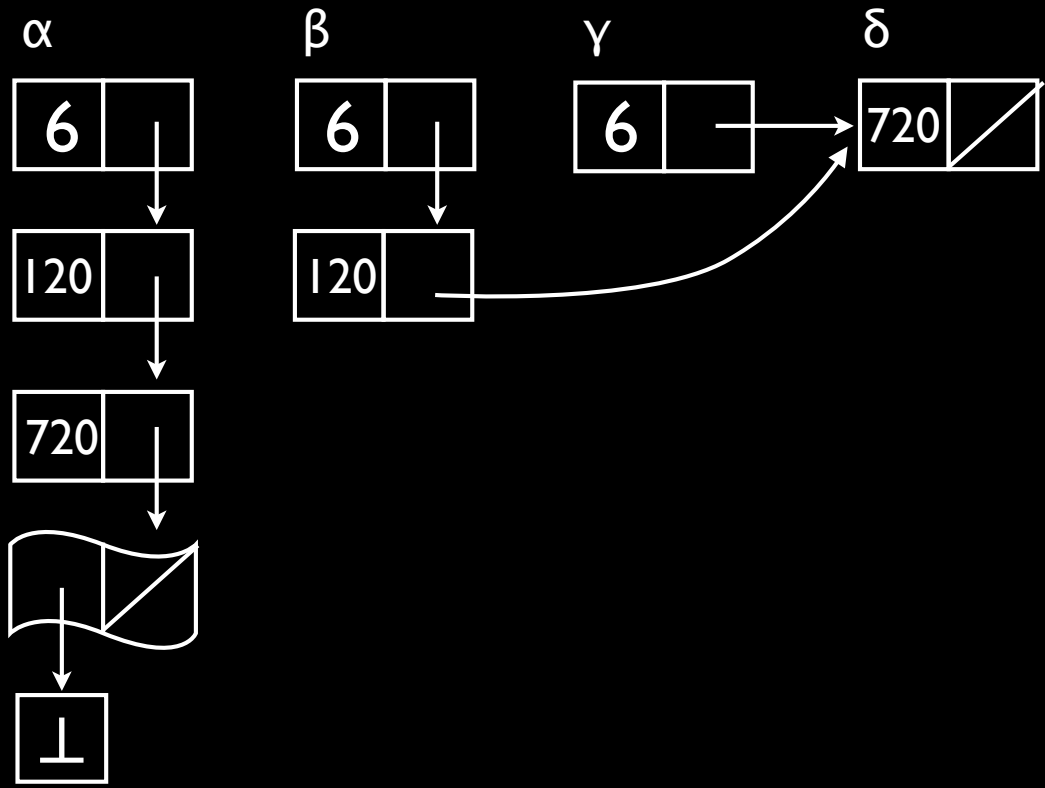
```



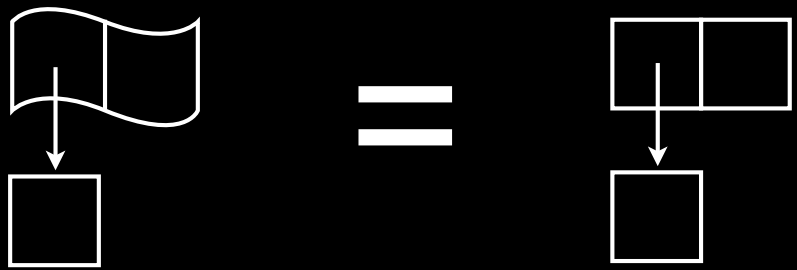
```

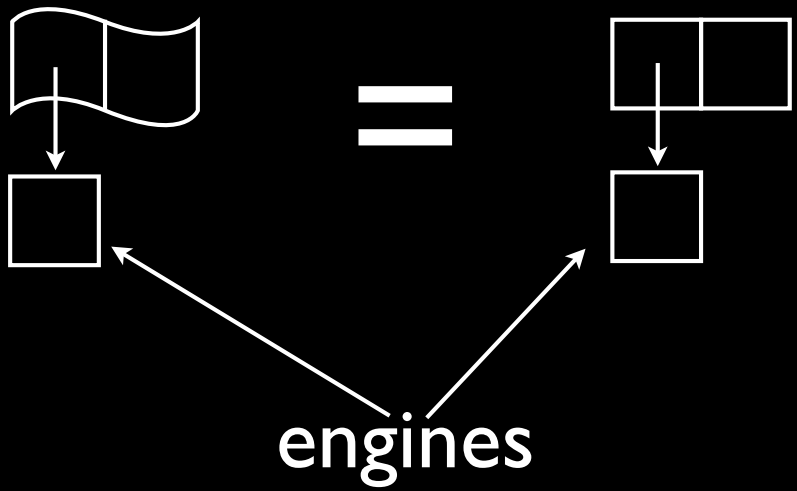
(let ((δ (cons⊥ (! 6) '()))))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (take⊥ 3 α))))))

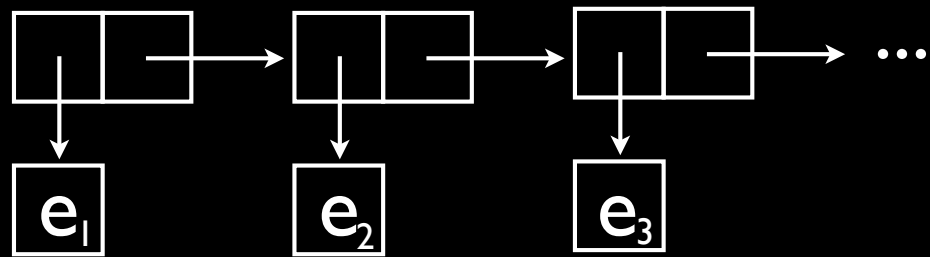
```



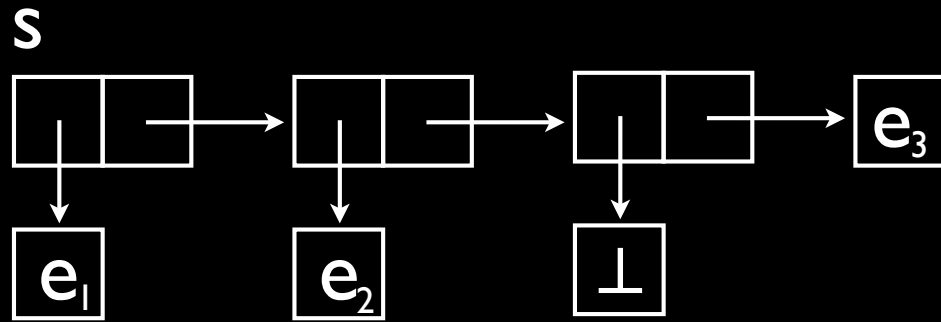
implementation



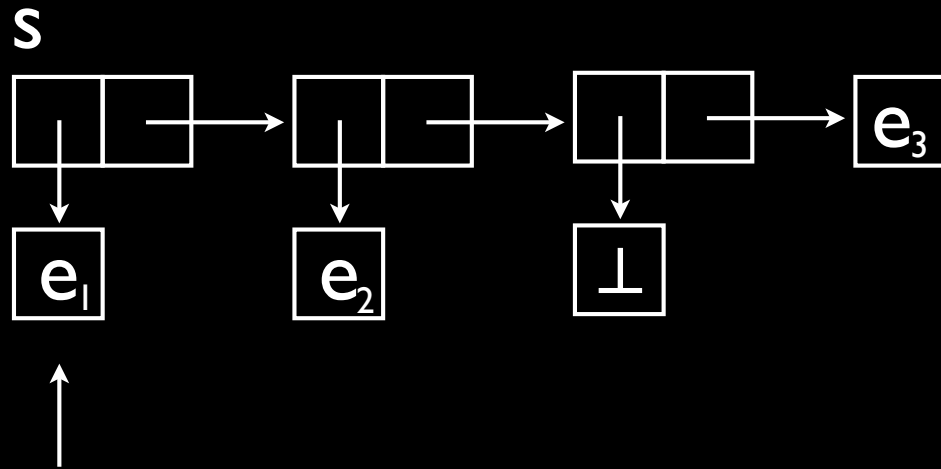




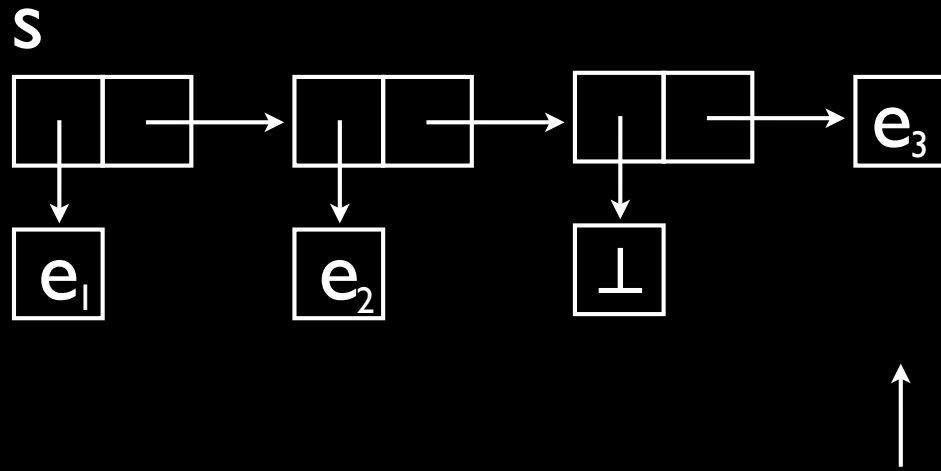
$\text{car}_{\perp} s$



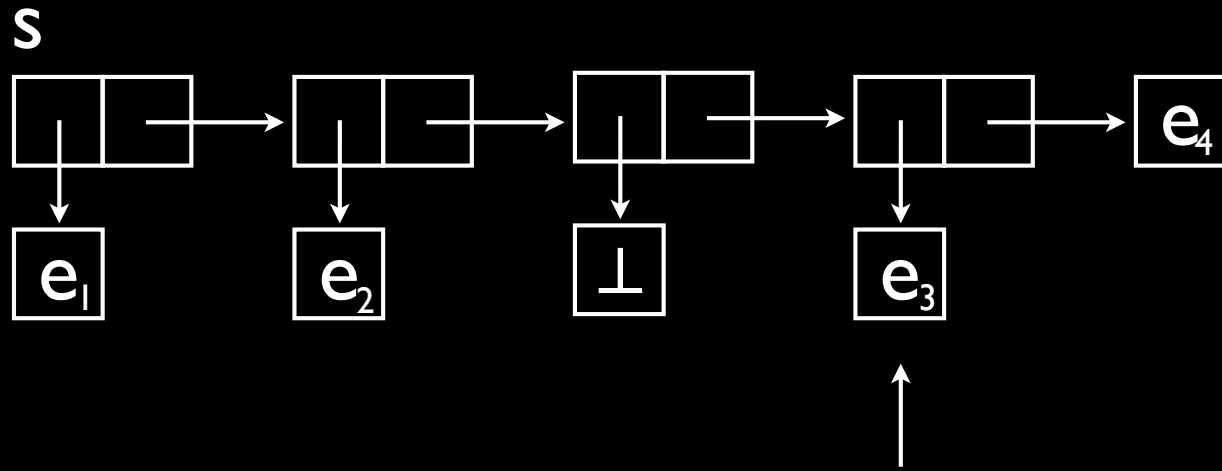
car<sub>⊥</sub> s



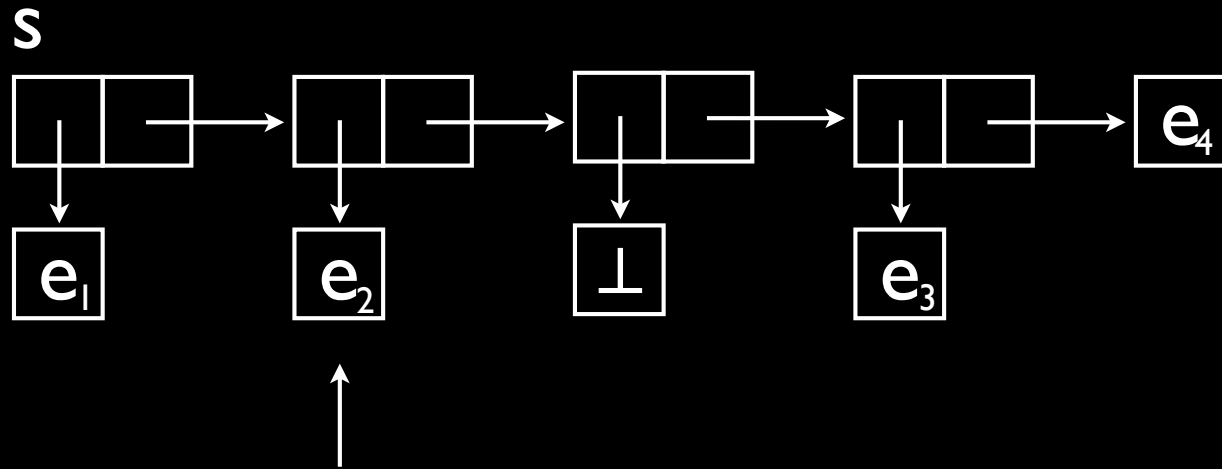
car<sub>⊥</sub> s



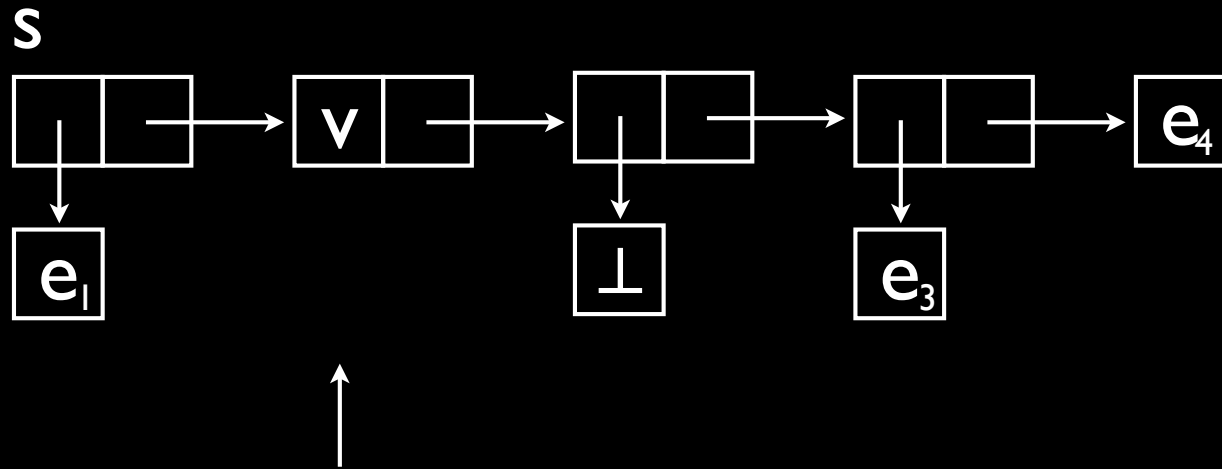
car<sub>⊥</sub> s



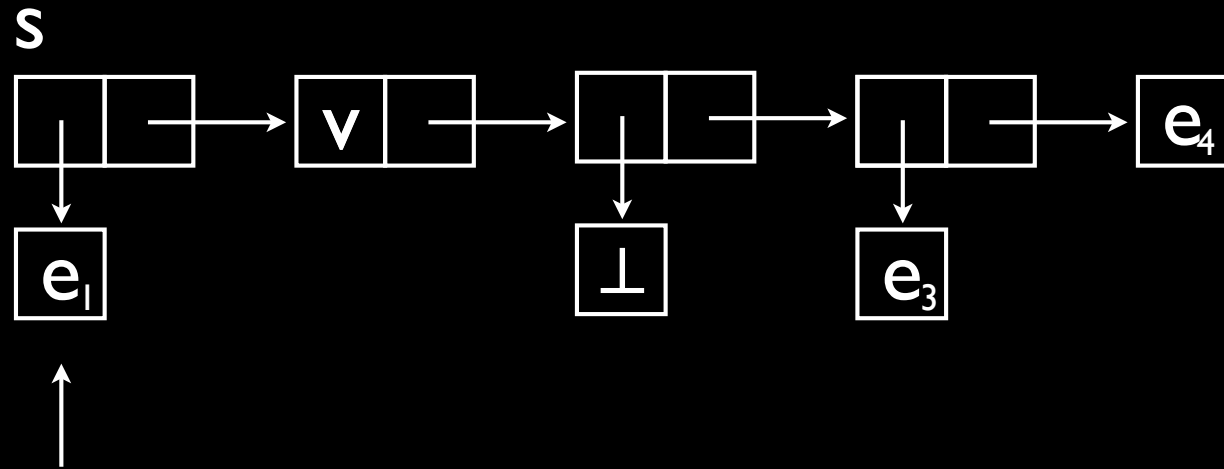
car<sub>⊥</sub> s



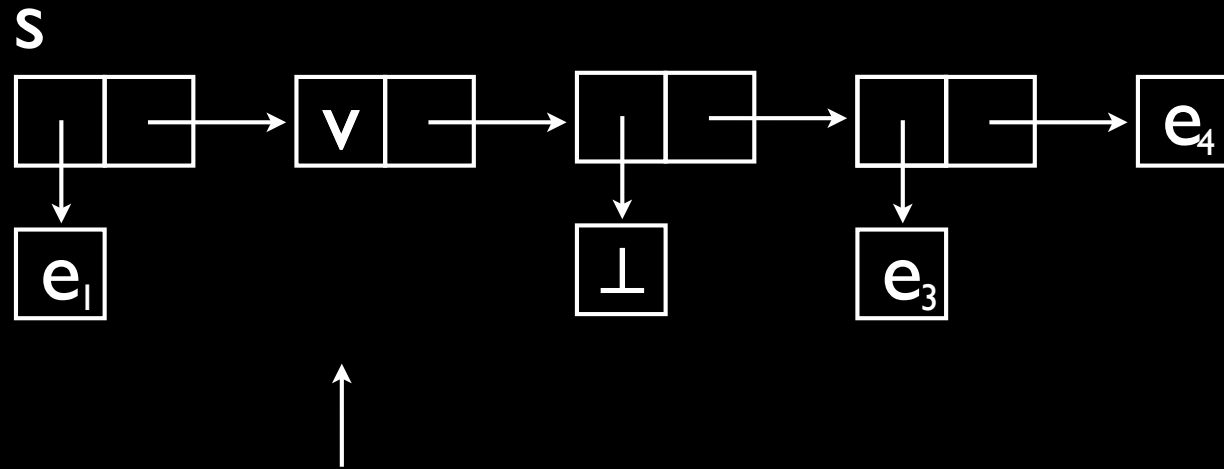
$\text{car}_{\perp} s$



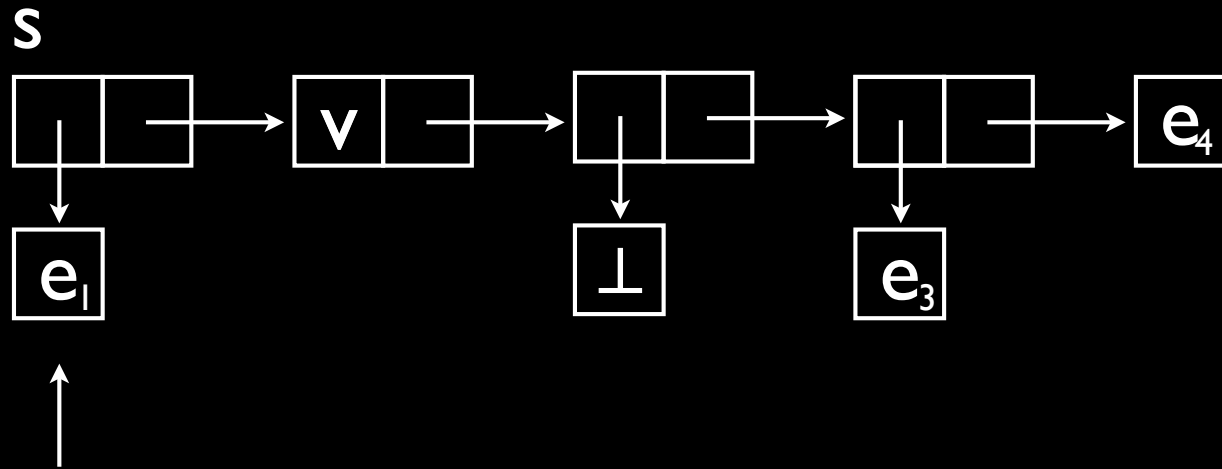
promote  $s$



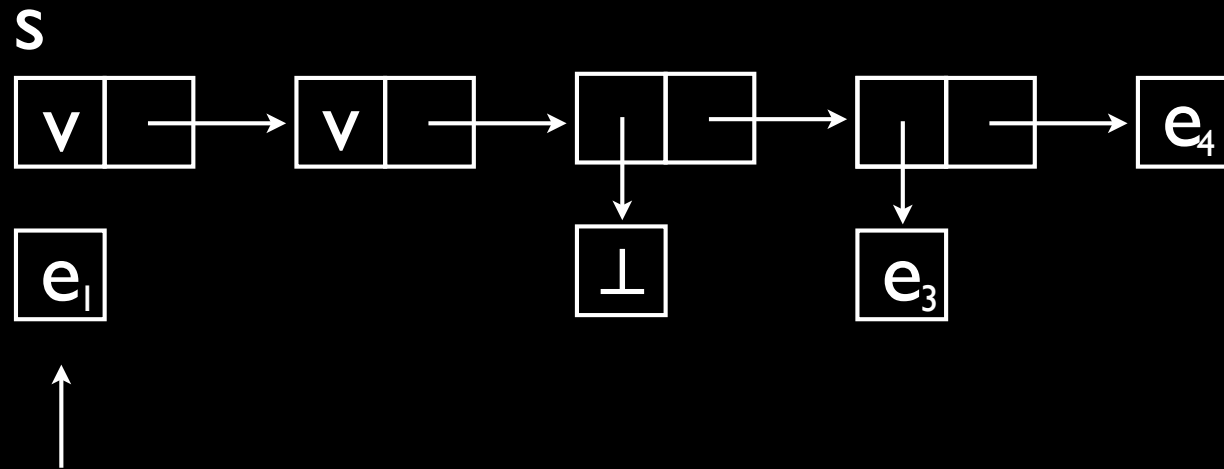
promote  $s$



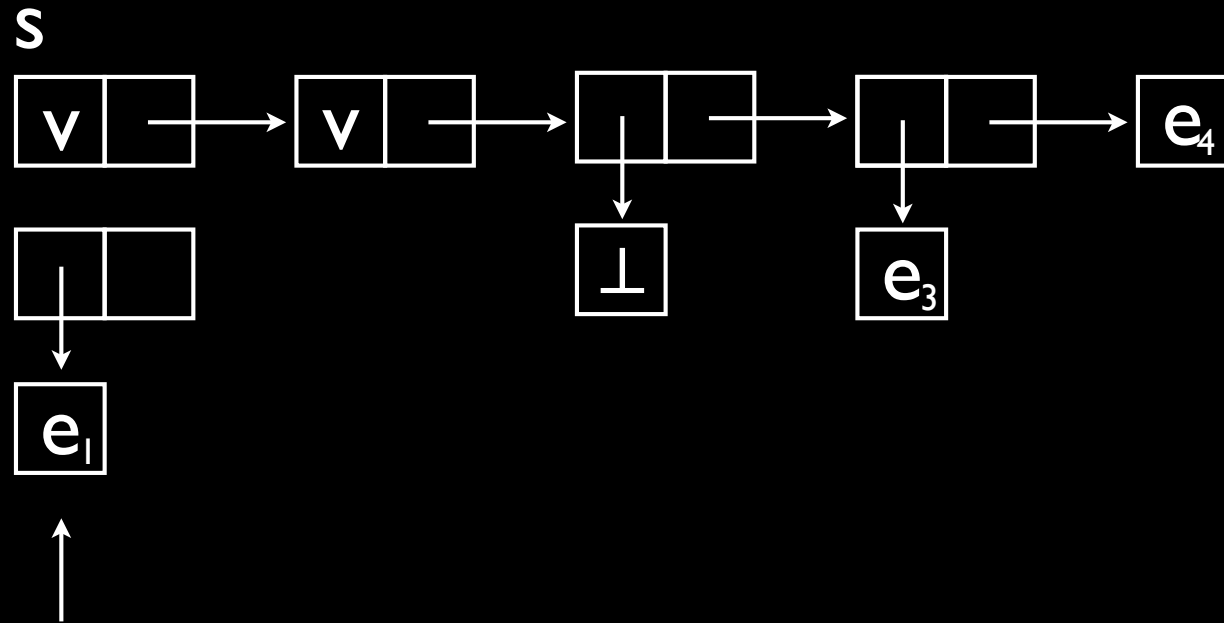
promote  $s$



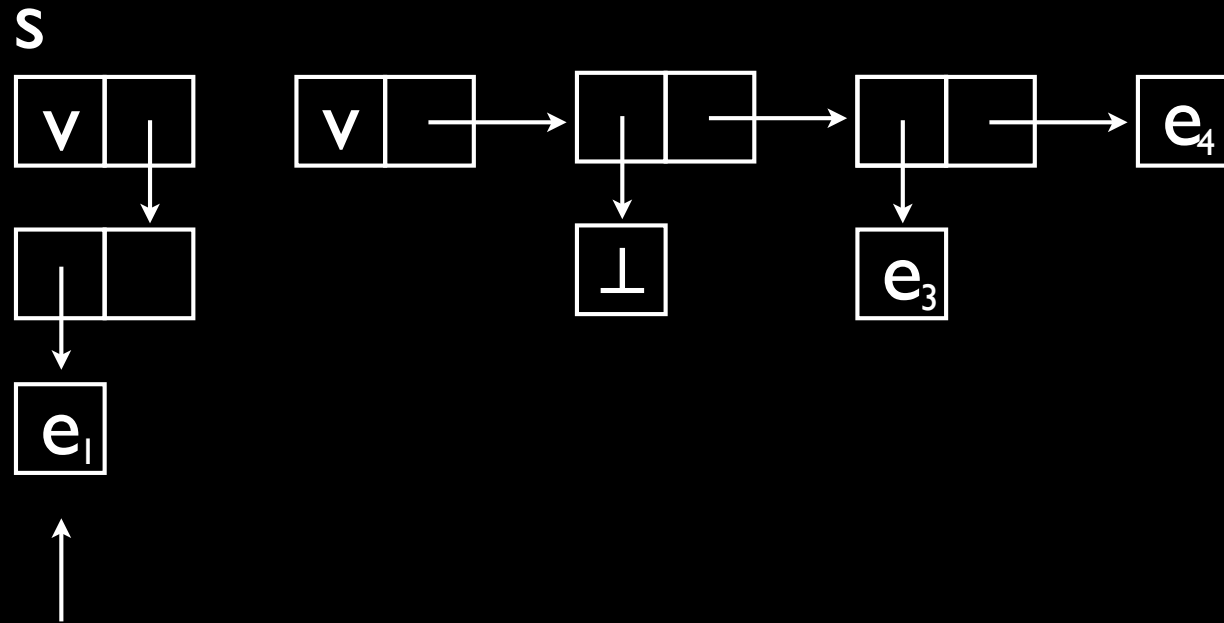
promote  $s$



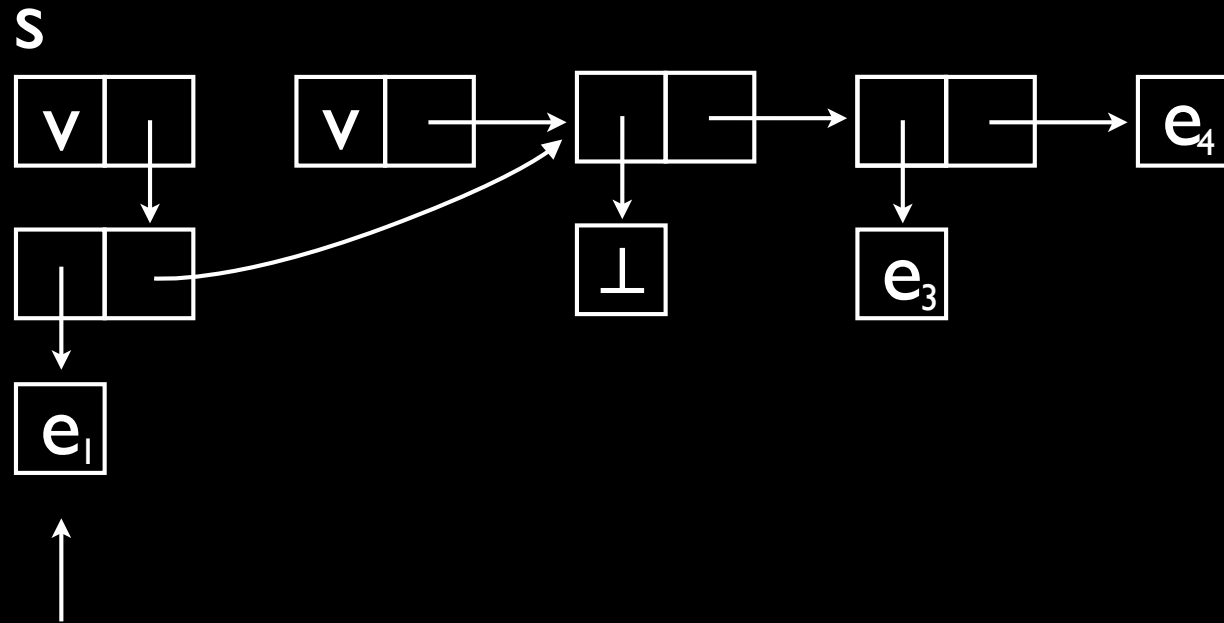
promote  $s$



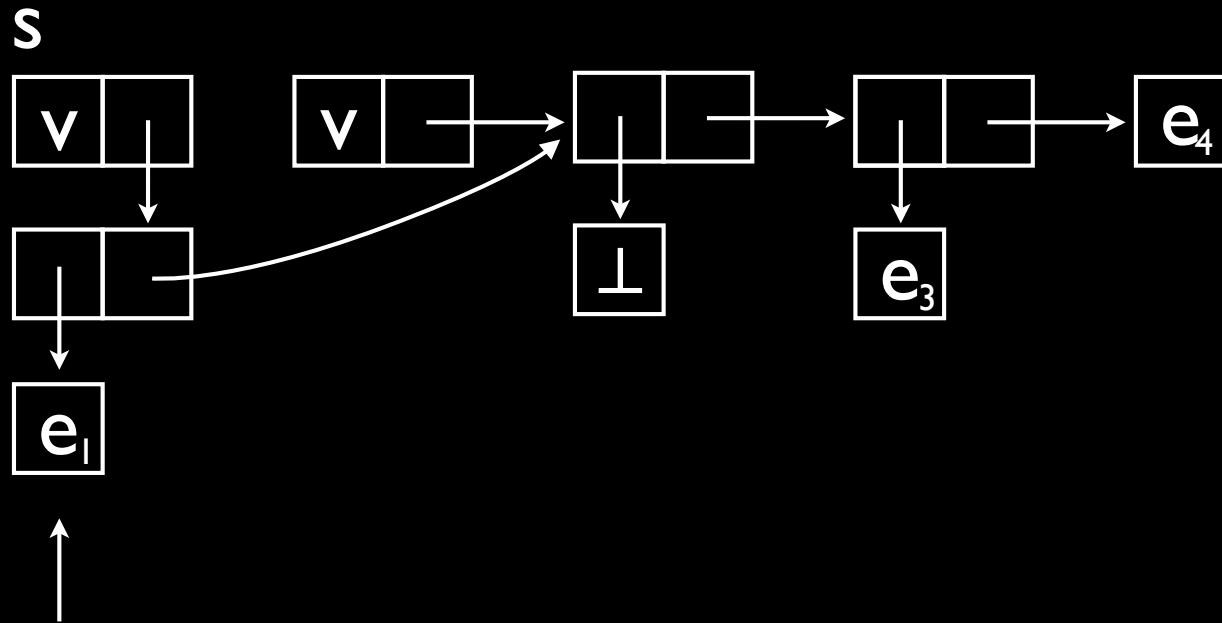
promote  $s$



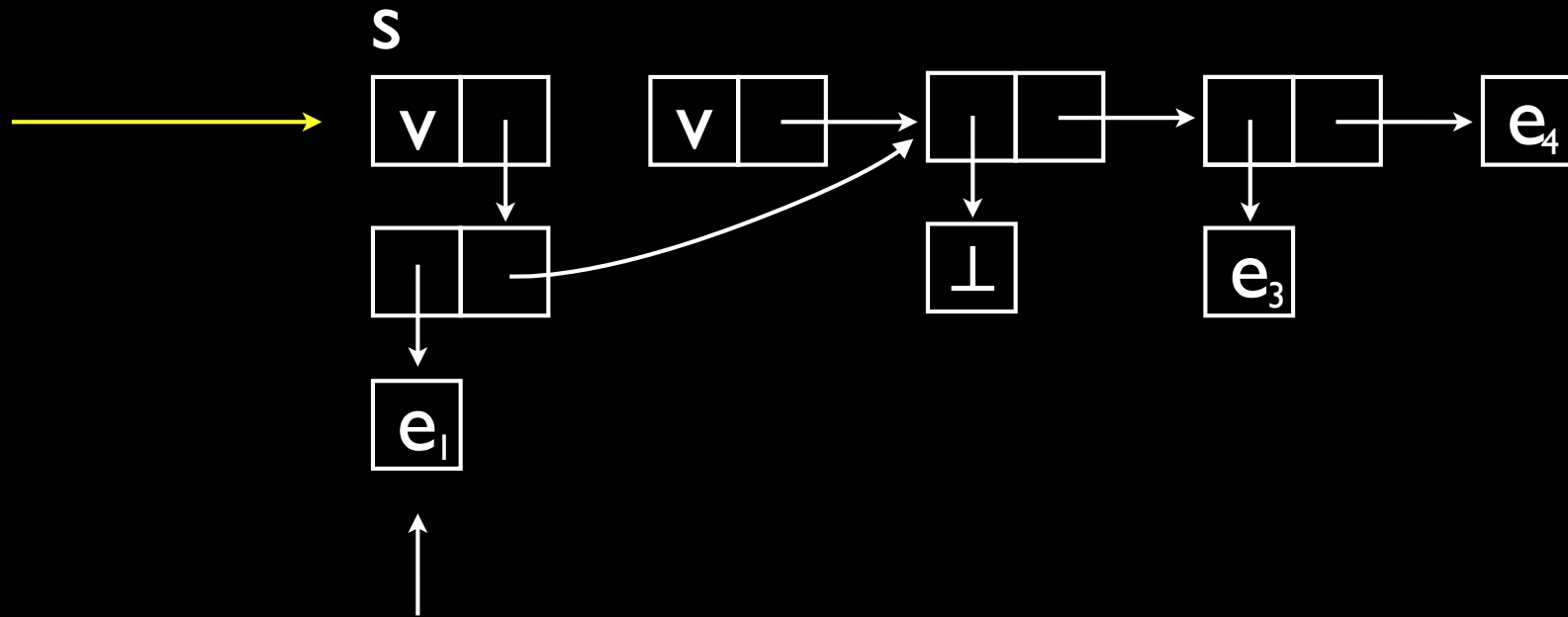
promote  $s$



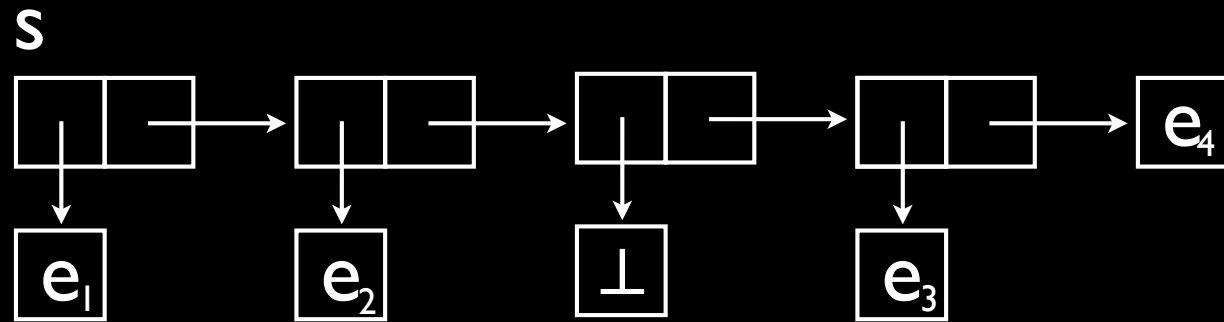
$\text{car}_\perp s$



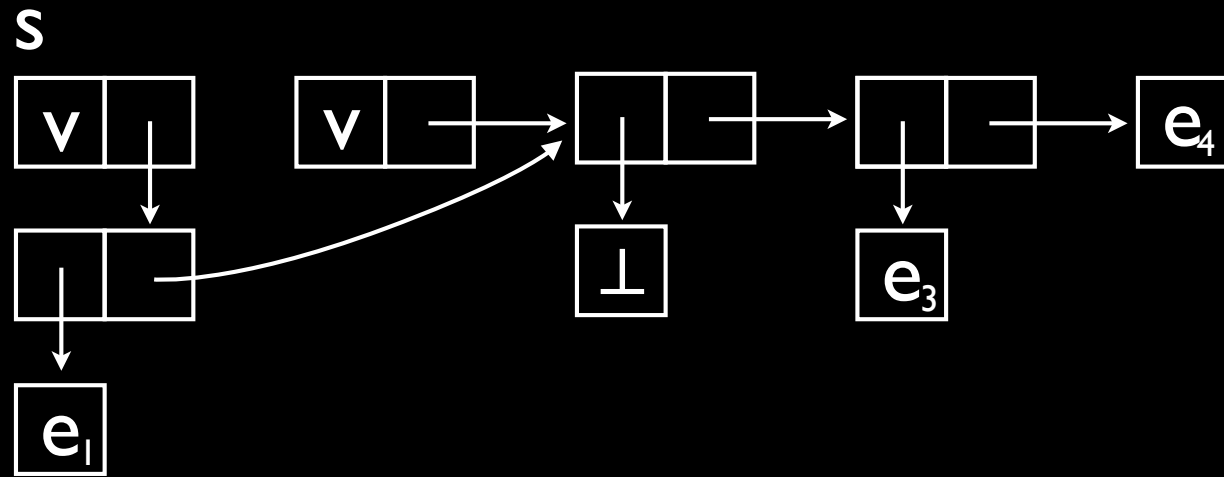
$\text{car}_\perp s$



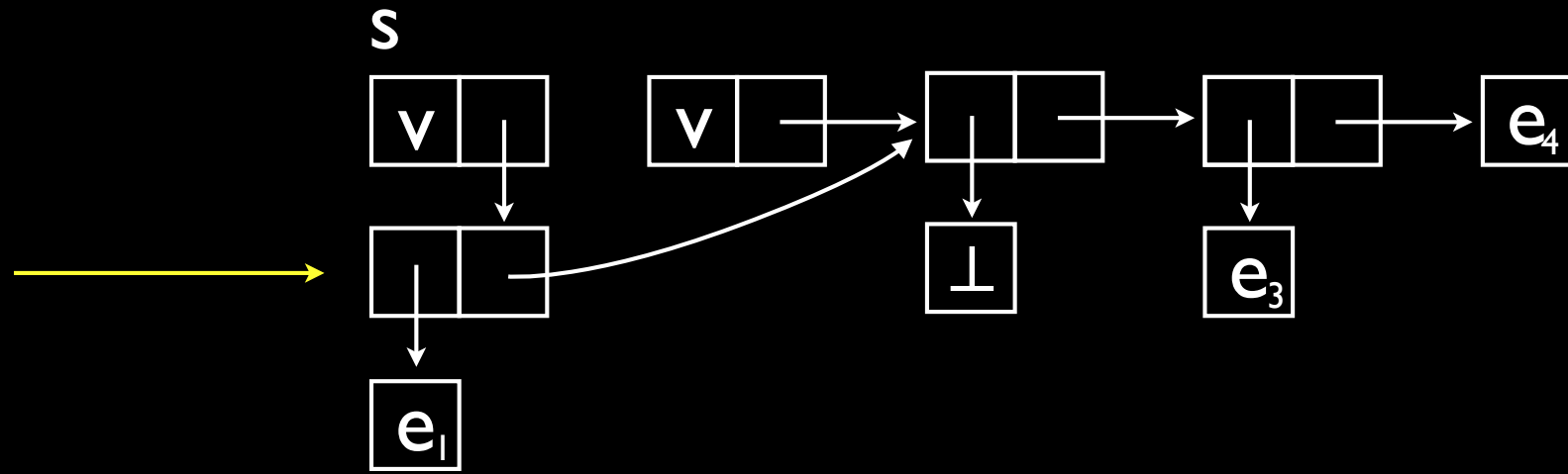
$cdr_1 s$



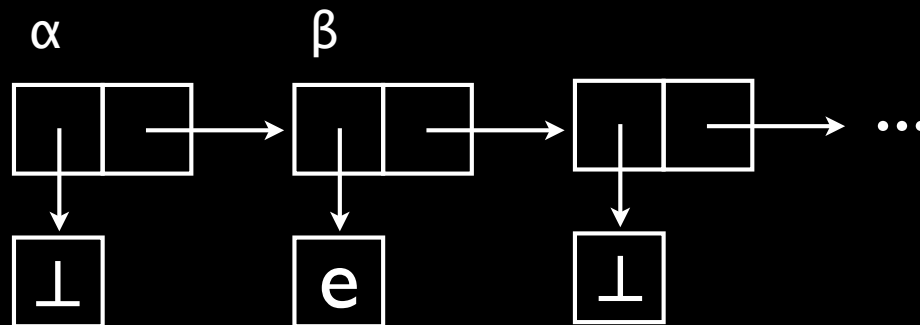
$cdr_1 s$



$cdr_1 s$



one more thing...



```
(define-syntax cons⊥
  (syntax-rules ()
    ((- a d) (cons (cons 'U (engine a)) (cons 'U (engine d))))))
```

```
(define car⊥
  (λt (p)
    (letrec ((racea
              (λt (q)
                (cond
                  ((La? q) (wait nsteps) (raced q))
                  ((Ua? q) (coaxa q) (raced q))
                  ((not (pair? q)) (racea p))
                  (else (promote p) (car p))))))
      (raced
        (λt (q)
          (cond
            ((Ld? q) (racea p))
            ((Ud? q) (coaxd q) (racea p))
            (else (racea (cdr q)))))))
      (racea p))))
```

```
(define promote
  (λt (p)
    (cond
      ((La? p) (wait nsteps) (promote p))
      ((Ua? p)
       (set-car! (car p) 'L)
       (lett ((te (car p)))
         (lett ((r (promote (cdr p))))
           (replace! p (car r) (cons te (cdr r)))
           (set-car! te 'U)
           p)))
      (else p))))
```

```
(define cdr⊥ (λt (p) (car⊥ p) (cdrs p)))
```

```
(define cdrs
  (λt (p)
    (cond
      ((Ld? p) (wait nsteps) (cdrs p))
      ((Ud? p) (coaxd p) (cdrs p))
      (else (cdr p))))
```

conclusions

download the library

<http://www.cs.indiana.edu/~webyrd/ferns.html>