IN2040: Funksjonell Programmering

Muterbare data

Martin Steffen

Universitetet i Oslo

7. uke (2. 10. 2023)



Agenda

Siste gang

- Prosedyrebasert objektorientering
- Lokale tilstandsvariabler
- Innkapsling + `set!`
- Eksempel: bankkonto
- Omgivelsesmodellen

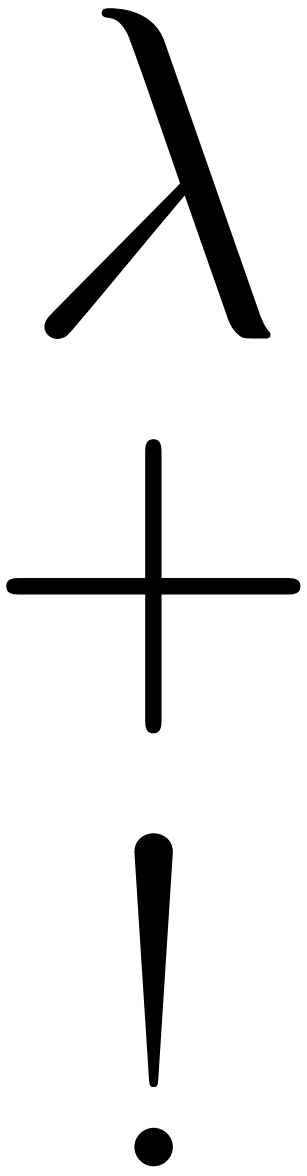
Agenda

Siste gang

- Prosedyrebasert objektorientering
- Lokale tilstandsvariabler
- Innkapsling + `set!`
- Eksempel: bankkonto
- Omgivelsesmodellen

I dag

- Kort om parallelitet
- Mer om likhet
- Lister som muterbar datastruktur
- Destruktive listeoperasjoner
- Køer som abstrakt datatype



Repetisjon: Prosedyrebasert objektorientering



```
(define (make-account balance)
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (withdraw amount)
    (set! balance (- balance amount))
    balance)
  (define (dispatch message)
    (cond ((eq? message 'deposit) deposit)
          ((eq? message 'withdraw) withdraw)
          ((eq? message 'balance) balance)))
  dispatch)
```

Brukseksempler

```
? (define peter
    (make-account 0))

? (define john
    (make-account 0))

? (eq? peter john) → #f

? ((peter 'deposit) 50)
→ 50
```

Repetisjon: Prosedyrebasert objektorientering

```
(define (make-account balance)
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (withdraw amount)
    (set! balance (- balance amount))
    balance)
  (define (dispatch message)
    (cond ((eq? message 'deposit) deposit)
          ((eq? message 'withdraw) withdraw)
          ((eq? message 'balance) balance)))
  dispatch)

;; Litt syntaktisk sukker:

(define (deposit amount account)
  ((account 'deposit) amount))

(define (withdraw amount account)
  ((account 'withdraw) amount))

(define (balance account)
  (account 'balance))
```

Brukseksempler

```
? (define peter
    (make-account 0))

? (define john
    (make-account 0))

? (eq? peter john) → #f

? ((peter 'deposit) 50)
→ 50
```

Repetisjon: Prosedyrebasert objektorientering

```
(define (make-account balance)
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (withdraw amount)
    (set! balance (- balance amount))
    balance)
  (define (dispatch message)
    (cond ((eq? message 'deposit) deposit)
          ((eq? message 'withdraw) withdraw)
          ((eq? message 'balance) balance)))
  dispatch)

;; Litt syntaktisk sukker:

(define (deposit amount account)
  ((account 'deposit) amount))

(define (withdraw amount account)
  ((account 'withdraw) amount))

(define (balance account)
  (account 'balance))
```

Brukseksempler

```
? (define peter
    (make-account 0))

? (define john
    (make-account 0))

? (eq? peter john) → #f

? ((peter 'deposit) 50)
→ 50

? (deposit 50 peter)
→ 100
```

Repetisjon: Prosedyrebasert objektorientering

```
(define (make-account balance)
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (withdraw amount)
    (set! balance (- balance amount))
    balance)
  (define (dispatch message)
    (cond ((eq? message 'deposit) deposit)
          ((eq? message 'withdraw) withdraw)
          ((eq? message 'balance) balance)))
  dispatch)

;; Litt syntaktisk sukker:

(define (deposit amount account)
  ((account 'deposit) amount))

(define (withdraw amount account)
  ((account 'withdraw) amount))

(define (balance account)
  (account 'balance))
```

Brukseksempler

```
? (define peter
    (make-account 0))

? (define john
    (make-account 0))

? (eq? peter john) → #f

? ((peter 'deposit) 50)
→ 50

? (deposit 50 peter)
→ 100

? (balance john) → 0
```

Repetisjon: Prosedyrebasert objektorientering

```
(define (make-account balance)
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (withdraw amount)
    (set! balance (- balance amount))
    balance)
  (define (dispatch message)
    (cond ((eq? message 'deposit) deposit)
          ((eq? message 'withdraw) withdraw)
          ((eq? message 'balance) balance)))
  dispatch)

;; Litt syntaktisk sukker:

(define (deposit amount account)
  ((account 'deposit) amount))

(define (withdraw amount account)
  ((account 'withdraw) amount))

(define (balance account)
  (account 'balance))
```

Brukseksempler

```
? (define peter
    (make-account 0))

? (define john
    (make-account 0))

? (eq? peter john) → #f

? ((peter 'deposit) 50)
→ 50

? (deposit 50 peter)
→ 100

? (balance john) → 0

? (define paul peter)
```

Repetisjon: Prosedyrebasert objektorientering

```
(define (make-account balance)
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (withdraw amount)
    (set! balance (- balance amount))
    balance)
  (define (dispatch message)
    (cond ((eq? message 'deposit) deposit)
          ((eq? message 'withdraw) withdraw)
          ((eq? message 'balance) balance)))
  dispatch)

;; Litt syntaktisk sukker:

(define (deposit amount account)
  ((account 'deposit) amount))

(define (withdraw amount account)
  ((account 'withdraw) amount))

(define (balance account)
  (account 'balance))
```

Brukseksempler

```
? (define peter
    (make-account 0))

? (define john
    (make-account 0))

? (eq? peter john) → #f

? ((peter 'deposit) 50)
→ 50

? (deposit 50 peter)
→ 100

? (balance john) → 0

? (define paul peter)

? (eq? paul peter) → #t
```

Repetisjon: Prosedyrebasert objektorientering

```
(define (make-account balance)
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (withdraw amount)
    (set! balance (- balance amount))
    balance)
  (define (dispatch message)
    (cond ((eq? message 'deposit) deposit)
          ((eq? message 'withdraw) withdraw)
          ((eq? message 'balance) balance)))
  dispatch)

;; Litt syntaktisk sukker:

(define (deposit amount account)
  ((account 'deposit) amount))

(define (withdraw amount account)
  ((account 'withdraw) amount))

(define (balance account)
  (account 'balance))
```

Brukseksempler

```
? (define peter
    (make-account 0))

? (define john
    (make-account 0))

? (eq? peter john) → #f

? ((peter 'deposit) 50)
→ 50

? (deposit 50 peter)
→ 100

? (balance john) → 0

? (define paul peter)

? (eq? paul peter) → #t

? (withdraw 25 paul)
→ 75
```

Repetisjon: Prosedyrebasert objektorientering



```
(define (make-account balance)
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (withdraw amount)
    (set! balance (- balance amount))
    balance)
  (define (dispatch message)
    (cond ((eq? message 'deposit) deposit)
          ((eq? message 'withdraw) withdraw)
          ((eq? message 'balance) balance)))
  dispatch)

;; Litt syntaktisk sukker:

(define (deposit amount account)
  ((account 'deposit) amount))

(define (withdraw amount account)
  ((account 'withdraw) amount))

(define (balance account)
  (account 'balance))
```

Brukseksempler

```
? (define peter
    (make-account 0))

? (define john
    (make-account 0))

? (eq? peter john) → #f

? ((peter 'deposit) 50)
→ 50

? (deposit 50 peter)
→ 100

? (balance john) → 0

? (define paul peter)

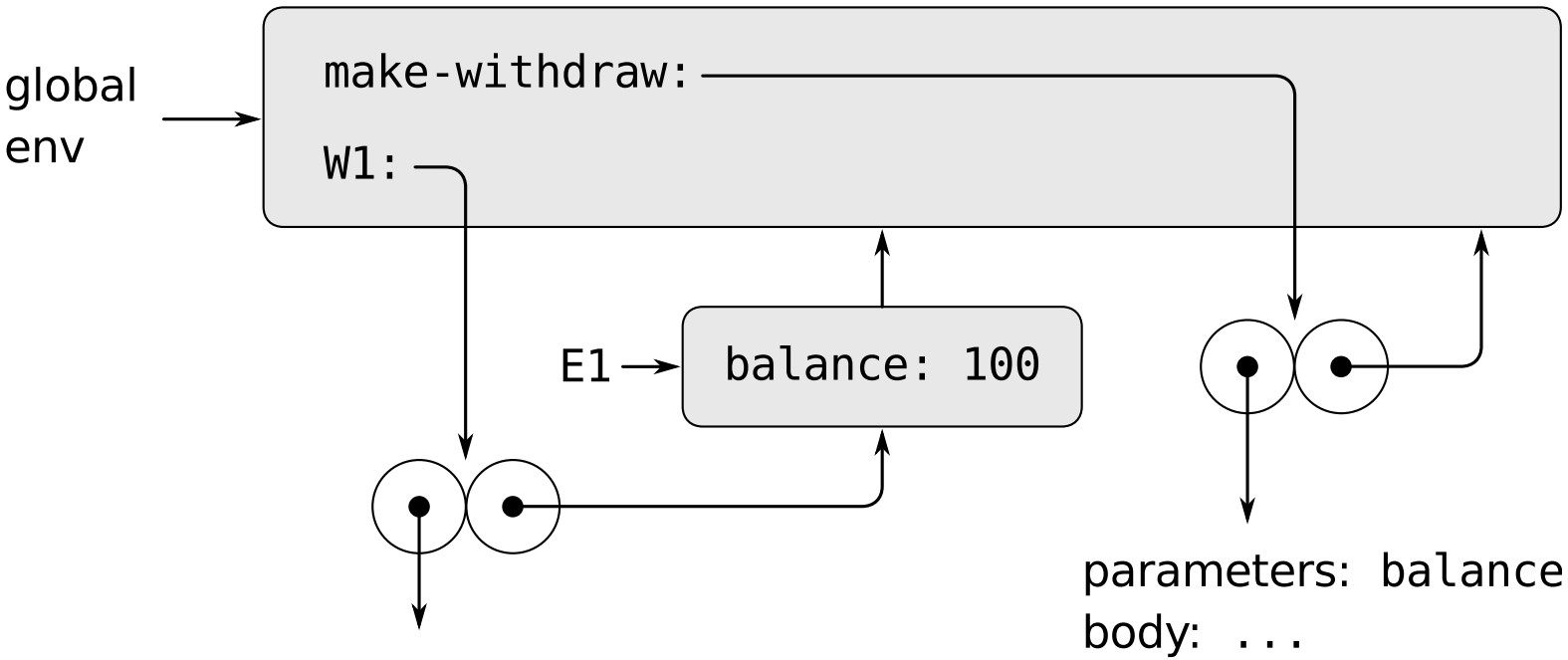
? (eq? paul peter) → #t

? (withdraw 25 paul)
→ 75

? (withdraw 10 peter)
→ 65
```

Repetisjon: Omgivelsesmodellen

```
? (define w1 (make-withdraw 100))
```



Diverse kommentarer

- En intern `define` etablerer bindingen sin i rammen som opprettes når den omsluttende prosedyren kalles (ikke når den defineres).

Diverse kommentarer

- En intern `define` etablerer bindingen sin i rammen som opprettes når den omsluttende prosedyren kalles (ikke når den defineres).
- Vi så et tidlig eksempel på bruk av *message passing* og innkapsling da vi implementerte par som prosedyrer.
- Det ekstra laget med "syntaktisk sukker" vi legger til for kontoobjekter over danner en effektiv abstraksjonsbarriere:
- Skjuler detaljene om hvordan kontoobjekter er representert; ikke lenger noe i grensesnittet som avslører at de er prosedyrer!

Veldig kort om parallelitet

- Med `set!` introduserte vi elementer fra imperativ programmering:
- Verditilordning og tilstandsendring som modifiserer objekter.
- Bankkonto er et muterbart objekt, med tidsavhengig tilstand.

Veldig kort om parallelitet

- Med `set!` introduserte vi elementer fra imperativ programmering:
- Verditilordning og tilstandsendring som modifiserer objekter.
- Bankkonto er et muterbart objekt, med tidsavhengig tilstand.

- I ren funskjonell programmering derimot var ting statisk over tid:
- Semantikken til et uttrykk var uavhengig av hvor og når det brukes.
- Derfor kan (rent) funksjonelle programmer uten videre paralleliseres.

Veldig kort om parallelitet



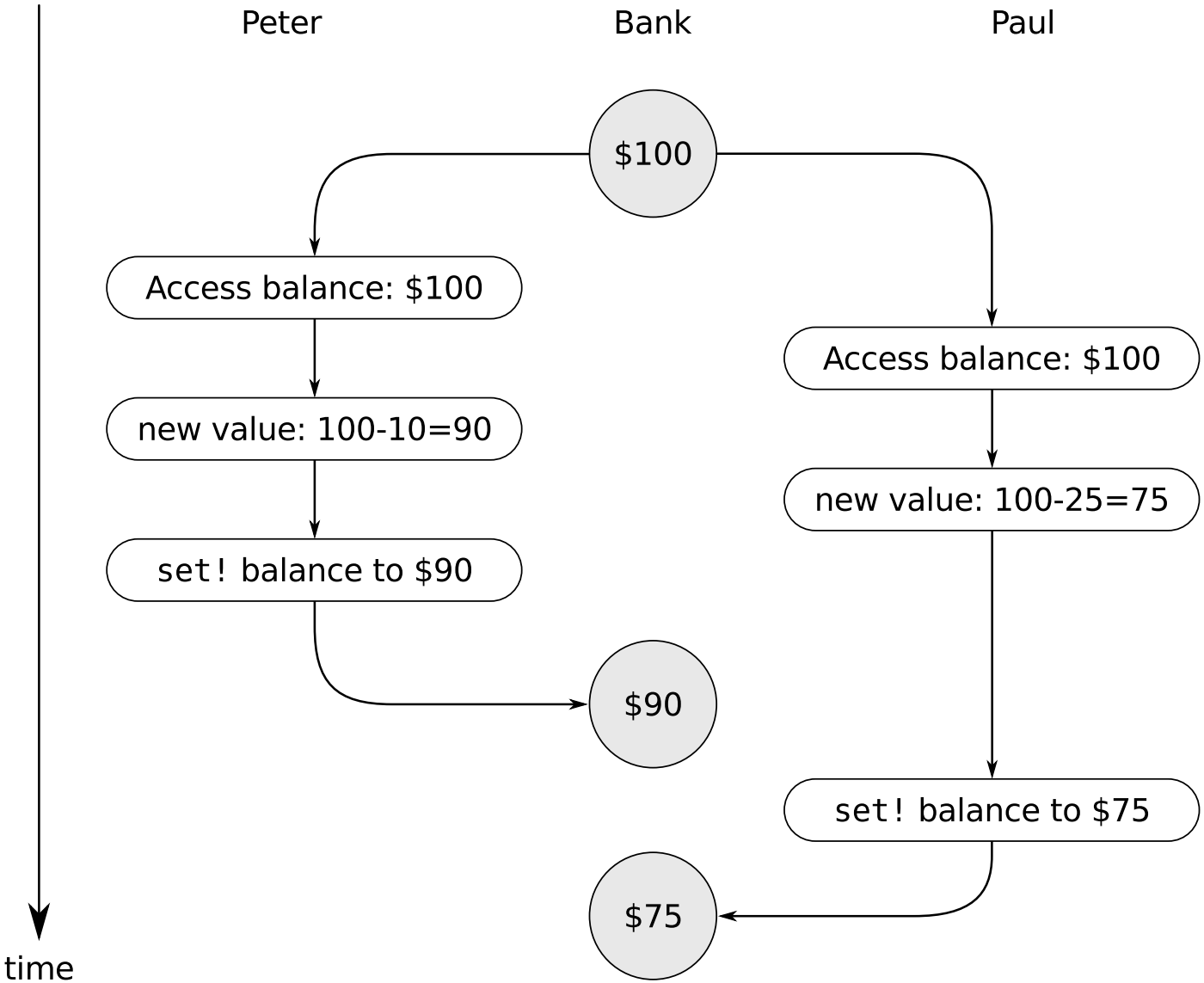
- Med `set!` introduserte vi elementer fra imperativ programmering:
- Verditilordning og tilstandsendring som modifiserer objekter.
- Bankkonto er et muterbart objekt, med tidsavhengig tilstand.

- I ren funskjonell programmering derimot var ting statisk over tid:
- Semantikken til et uttrykk var uavhengig av hvor og når det brukes.
- Derfor kan (rent) funksjonelle programmer uten videre paralleliseres.

- Med tidsavhengig tilstand trengs det mekanismer for synkronisering.
- Rekkefølge på operasjoner i en prosess vs. flere prosesser.
- Problem: destruktive prosedyrer som 'deler' på felles ressurser.

Veldig kort om parallelitet (forts.)

`(set! balance (- balance amount))` $\times 2$, samtidig på samme konto:



Veldig kort om parallelitet (forts.)

Fra SICP ex. 3.38:

- Suppose that Peter, Paul, and Mary share a joint bank account that initially contains $100.
- Concurrently, Peter deposits $10, Paul withdraws $20, and Mary withdraws half the money in the account.

Peter	`(set! balance (+ balance 10))`
Paul	`(set! balance (- balance 20))`
Mary	`(set! balance (- balance (/ balance 2)))`

Veldig kort om parallelitet (forts.)

Fra SICP ex. 3.38:

- Suppose that Peter, Paul, and Mary share a joint bank account that initially contains $100.
- Concurrently, Peter deposits $10, Paul withdraws $20, and Mary withdraws half the money in the account.

```
Peter   (set! balance (+ balance 10))
Paul    (set! balance (- balance 20))
Mary    (set! balance (- balance (/ balance 2)))
```

- Hva er noen mulige sluttsaldoer,
  - hvis `set!`-uttrykkene evalueres i (en eller annen) rekkefølge?
  - hvis `set!`-uttrykkene evalueres samtidig (ingen rekkefølge)?
- Hva er kritiske punkter med tanke på parallelitet og tidsdimensjonen?

Veldig kort om parallelitet (forts.)

Fra SICP ex. 3.38:

- Suppose that Peter, Paul, and Mary share a joint bank account that initially contains $100.
- Concurrently, Peter deposits $10, Paul withdraws $20, and Mary withdraws half the money in the account.

```
Peter   (set! balance (+ balance 10))
Paul    (set! balance (- balance 20))
Mary    (set! balance (- balance (/ balance 2)))
```

- Hva er noen mulige sluttsaldoer,
  - hvis `set!`-uttrykkene evalueres i (en eller annen) rekkefølge?
  - hvis `set!`-uttrykkene evalueres samtidig (ingen rekkefølge)?
- Hva er kritiske punkter med tanke på parallelitet og tidsdimensjonen?

Veldig kort om parallelitet (forts.)

- For å synkronisere rundt bruk av felles ressurser brukes *semaphores* (aka *mutex* eller *lock*), først diskutert av Edsger Dijkstra i 1965.
- Før kritisk seksjon, få tak i semaphoren (*acquire*).
- Ved utgang fra kritisk seksjon, slippe (*release*).
- Prosesser blokkeres under venting på semaphoren.

```
(define (make-account balance)
  (let ((mutex (make-mutex)))
    (define (withdraw  amount)
      (mutex 'acquire)
      (set! balance (- balance amount))
      (mutex 'release))
    ... ))
```



Repetisjon: Likhet

- `eq?` tester om argumentene som sammenliknes er ett og samme objekt i minnet.
- `equal?` brukes når vi kun bryr oss om ekvialens i verdier.

```
? (define ping '(1 2))
? (define pong '(1 2))
? (define pang ping)
```

Repetisjon: Likhet

- `eq?` tester om argumentene som sammenliknes er ett og samme objekt i minnet.
- `equal?` brukes når vi kun bryr oss om ekvialens i verdier.

```
? (define ping '(1 2))
? (define pong '(1 2))
? (define pang ping)
? (equal? ping pong) →
```

Repetisjon: Likhet

- `eq?` tester om argumentene som sammenliknes er ett og samme objekt i minnet.
- `equal?` brukes når vi kun bryr oss om ekvialens i verdier.

```
? (define ping '(1 2))
? (define pong '(1 2))
? (define pang ping)
? (equal? ping pong) → #t
```

Repetisjon: Likhet

- `eq?` tester om argumentene som sammenliknes er ett og samme objekt i minnet.
- `equal?` brukes når vi kun bryr oss om ekvialens i verdier.

```
? (define ping '(1 2))
? (define pong '(1 2))
? (define pang ping)
? (equal? ping pong) → #t
? (eq? ping pong) →
```

Repetisjon: Likhet

- `eq?` tester om argumentene som sammenliknes er ett og samme objekt i minnet.
- `equal?` brukes når vi kun bryr oss om ekvialens i verdier.

```
? (define ping '(1 2))
? (define pong '(1 2))
? (define pang ping)
? (equal? ping pong) → #t
? (eq? ping pong) → #f
```

Repetisjon: Likhet

- `eq?` tester om argumentene som sammenliknes er ett og samme objekt i minnet.
- `equal?` brukes når vi kun bryr oss om ekvialens i verdier.

```
? (define ping '(1 2))
? (define pong '(1 2))
? (define pang ping)
? (equal? ping pong) → #t
? (eq? ping pong) → #f
? (eq? ping pang) →
```

Repetisjon: Likhet

- `eq?` tester om argumentene som sammenliknes er ett og samme objekt i minnet.
- `equal?` brukes når vi kun bryr oss om ekvialens i verdier.

```
? (define ping '(1 2))
? (define pong '(1 2))
? (define pang ping)
? (equal? ping pong) → #t
? (eq? ping pong) → #f
? (eq? ping pang) → #t
```

Repetisjon: Likhet

- `eq?` tester om argumentene som sammenliknes er ett og samme objekt i minnet.
- `equal?` brukes når vi kun bryr oss om ekvialens i verdier.
- Merk at symboler og den tomme lista `'()` er konstanter.

```
? (define ping '(1 2))
? (define pong '(1 2))
? (define pang ping)
? (equal? ping pong) → #t
? (eq? ping pong) → #f
? (eq? ping pang) → #t
? (eq? '() '()) → #t
? (eq? 'foo 'foo) → #t
```

Mer likhet

- `set!` endrer hvilken verdi variabelen er bundet til: selve verdien i seg selv endres ikke.

```
? (define ping '(1 2))

? (define pang ping)

? (eq? ping pang) → #t
```

Mer likhet

- `set!` endrer hvilken verdi variabelen er bundet til: selve verdien i seg selv endres ikke.

```
? (define ping '(1 2))
? (define pang ping)
? (eq? ping pang) → #t
? (set! ping 42)
```

Mer likhet

- `set!` endrer hvilken verdi variabelen er bundet til: selve verdien i seg selv endres ikke.

```
? (define ping '(1 2))
? (define pang ping)
? (eq? ping pang) → #t
? (set! ping 42)
? ping → 42
```

Mer likhet

- `set!` endrer hvilken verdi variabelen er bundet til: selve verdien i seg selv endres ikke.

```
? (define ping '(1 2))
? (define pang ping)
? (eq? ping pang) → #t
? (set! ping 42)
? ping → 42
? pang →
```

Mer likhet

- `set!` endrer hvilken verdi variabelen er bundet til: selve verdien i seg selv endres ikke.

```
? (define ping '(1 2))
? (define pang ping)
? (eq? ping pang) → #t
? (set! ping 42)
? ping → 42
? pang → (1 2)
```

Mer likhet

- `set!` endrer hvilken verdi variabelen er bundet til: selve verdien i seg selv endres ikke.
- Prosedyrer kalles på *verdien* av et uttrykk.
- I `zapp` er `x` en lokal variabel som bindes til verdien som gis som argument (før vi binder den til en ny verdi med `set!`).

```
? (define ping '(1 2))

? (define pang ping)

? (eq? ping pang) → #t

? (set! ping 42)

? ping → 42

? pang → (1 2)

? (define (zapp x)
    (set! x 42)
    x)
```

Mer likhet

- `set!` endrer hvilken verdi variabelen er bundet til: selve verdien i seg selv endres ikke.
- Prosedyrer kalles på *verdien* av et uttrykk.
- I `zapp` er `x` en lokal variabel som bindes til verdien som gis som argument (før vi binder den til en ny verdi med `set!`).

```
? (define ping '(1 2))

? (define pang ping)

? (eq? ping pang) → #t

? (set! ping 42)

? ping → 42

? pang → (1 2)

? (define (zapp x)
    (set! x 42)
    x)

? (zapp pang) → 42
```

Mer likhet

- `set!` endrer hvilken verdi variabelen er bundet til: selve verdien i seg selv endres ikke.
- Prosedyrer kalles på *verdien* av et uttrykk.
- I `zapp` er `x` en lokal variabel som bindes til verdien som gis som argument (før vi binder den til en ny verdi med `set!`).

```
? (define ping '(1 2))

? (define pang ping)

? (eq? ping pang) → #t

? (set! ping 42)

? ping → 42

? pang → (1 2)

? (define (zapp x)
    (set! x 42)
    x)

? (zapp pang) → 42

? pang →
```

Mer likhet

- `set!` endrer hvilken verdi variabelen er bundet til: selve verdien i seg selv endres ikke.
- Prosedyrer kalles på *verdien* av et uttrykk.
- I `zapp` er `x` en lokal variabel som bindes til verdien som gis som argument (før vi binder den til en ny verdi med `set!`).

```
? (define ping '(1 2))

? (define pang ping)

? (eq? ping pang) → #t

? (set! ping 42)

? ping → 42

? pang → (1 2)

? (define (zapp x)
    (set! x 42)
    x)

? (zapp pang) → 42

? pang → (1 2)
```

Mer likhet

- `set!` endrer hvilken verdi variabelen er bundet til: selve verdien i seg selv endres ikke.
- Prosedyrer kalles på *verdien* av et uttrykk.
- I `zapp` er `x` en lokal variabel som bindes til verdien som gis som argument (før vi binder den til en ny verdi med `set!`).

```
? (define ping '(1 2))

? (define pang ping)

? (eq? ping pang) → #t

? (set! ping 42)

? ping → 42

? pang → (1 2)

? (define (zapp x)
    (set! x 42)
    x)

? (zapp pang) → 42

? pang → (1 2)

? (zapp 1) → 42
```

Enda mer likhet: strukturdeling

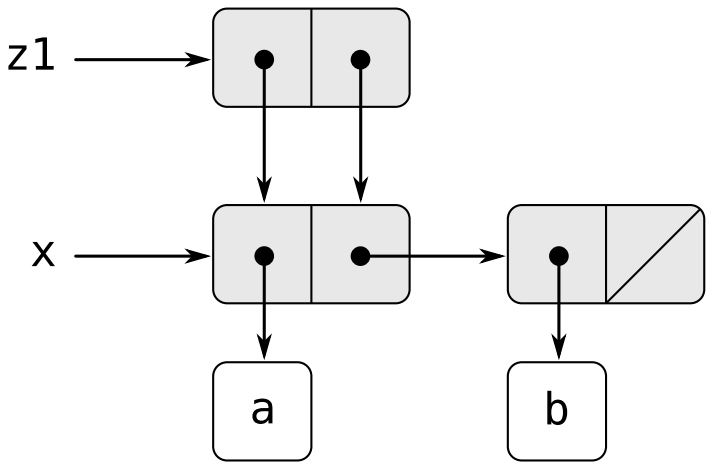
```
? (define x (list 'a 'b))
? (define z1 (cons x x))
? (define z2 (cons (list 'a 'b) (list 'a 'b)))
? z1 →
? z2 →
```

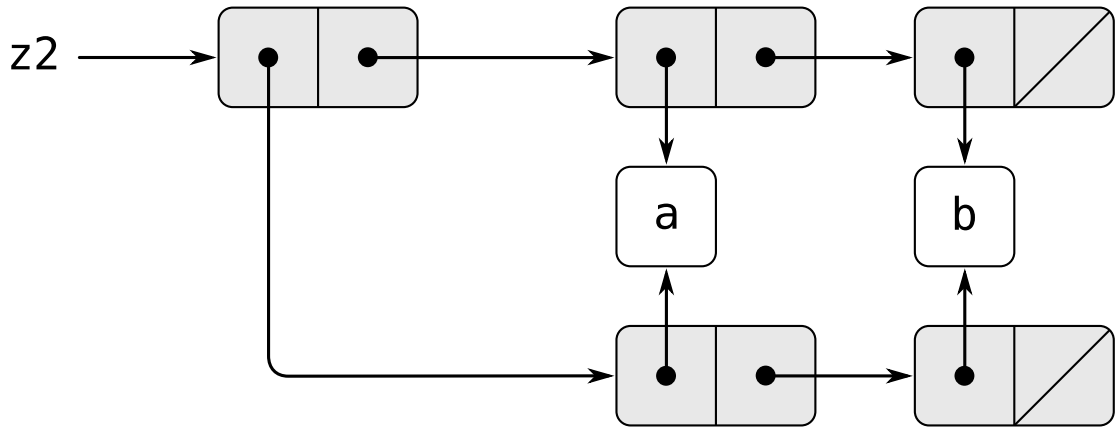
Enda mer likhet: strukturdeling

```
? (define x (list 'a 'b))
? (define z1 (cons x x))
? (define z2 (cons (list 'a 'b) (list 'a 'b)))
? z1 → ((a b) a b)
? z2 → ((a b) a b)
```

Enda mer likhet: strukturdeling

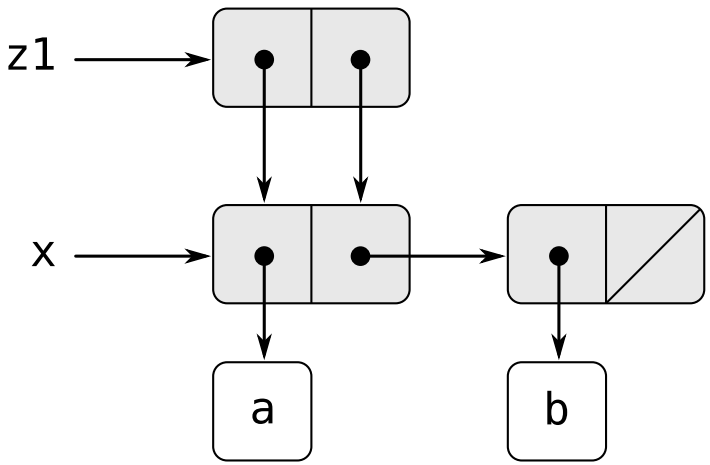
```
? (define x (list 'a 'b))
? (define z1 (cons x x))
? (define z2 (cons (list 'a 'b) (list 'a 'b)))
? z1 → ((a b) a b)
? z2 → ((a b) a b)
```

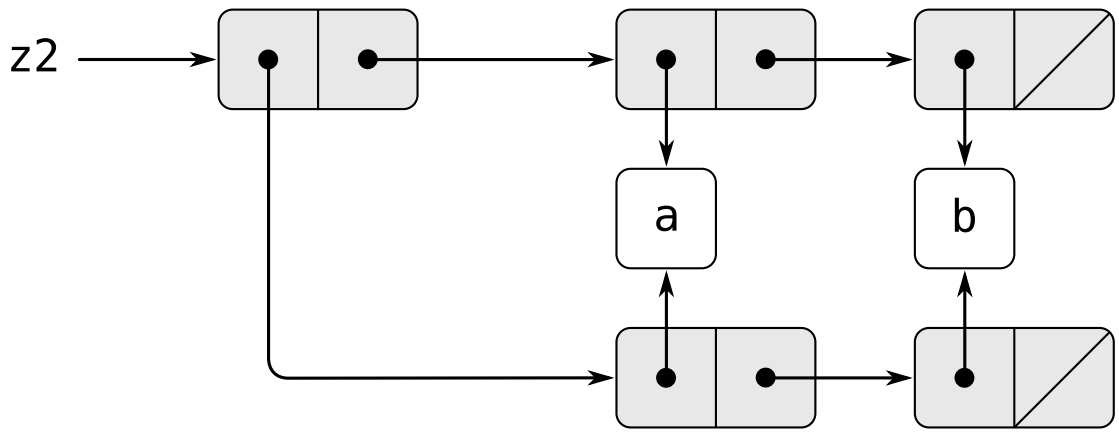




Enda mer likhet: strukturdeling

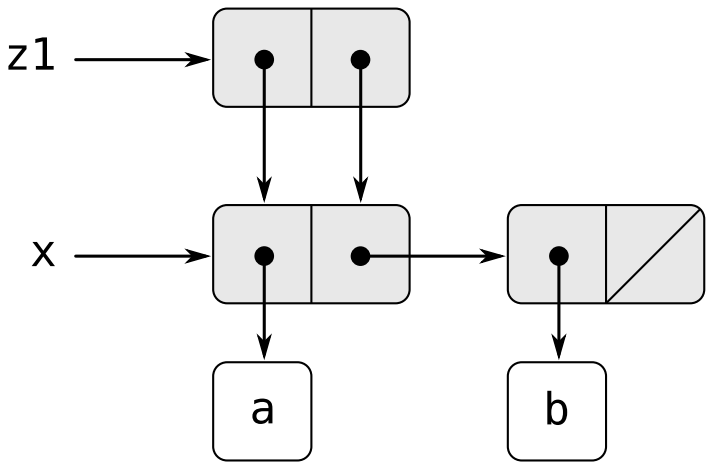
```
? (define x (list 'a 'b))
? (define z1 (cons x x))
? (define z2 (cons (list 'a 'b) (list 'a 'b)))
? z1 → ((a b) a b)
? z2 → ((a b) a b)
? (eq? (car z1) (cdr z1)) →
? (eq? (car z2) (cdr z2)) →
```

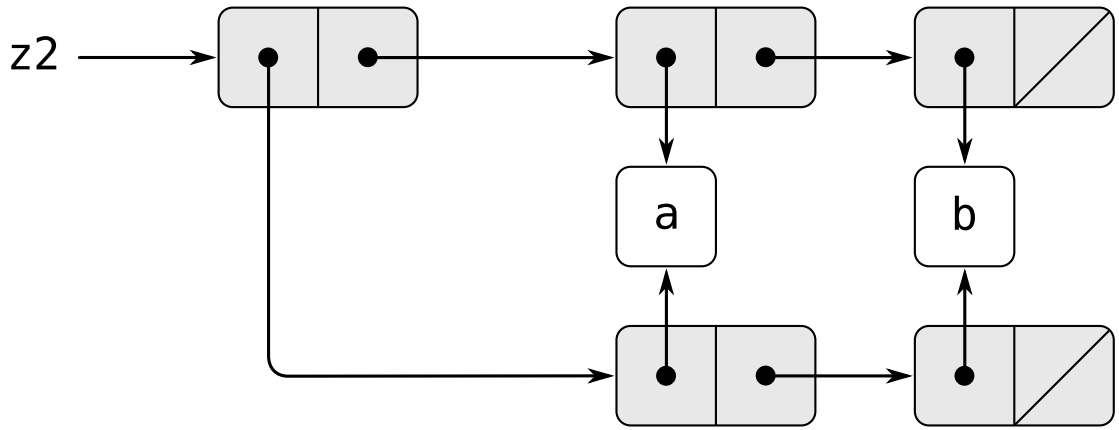




Enda mer likhet: strukturdeling

```
? (define x (list 'a 'b))
? (define z1 (cons x x))
? (define z2 (cons (list 'a 'b) (list 'a 'b)))
? z1 → ((a b) a b)
? z2 → ((a b) a b)
? (eq? (car z1) (cdr z1)) → #t
? (eq? (car z2) (cdr z2)) → #f
```





Tilbake til destruktive operasjoner



- Vi har sett at vi kan tilordne nye verdier for symbolske variabler.
- Har snakket om at vi må tenke på variabler som *steder* for å lagre verdier snarere enn *navn* på verdier.
- Generaliserer naturlig til komponenter i komplekse datastrukturer.
- Datastrukturer som tillater at vi endrer verdiene de lagrer kaller vi muterbare data.
- Eksempel: par og lister.

Destruktive listeoperasjoner

- Fra før kjenner vi konstruktorene og selektorene: `cons`, `car` og `cdr`.
- Nå kan vi legge til to mutatorer: `set-car!` og `set-cdr!`

Destruktive listeoperasjoner



- Fra før kjenner vi konstruktorene og selektorene: `cons`, `car` og `cdr`.
- Nå kan vi legge til to mutatorer: `set-car!` og `set-cdr!`

```
? (define foo (cons 'hoopy 'frood))
? foo →
```

Destruktive listeoperasjoner

- Fra før kjenner vi konstruktorene og selektorene: `cons`, `car` og `cdr`.
- Nå kan vi legge til to mutatorer: `set-car!` og `set-cdr!`

```
? (define foo (cons 'hoopy 'frood))
? foo → (hoopy . frood)
```

Destruktive listeoperasjoner

- Fra før kjenner vi konstruktorene og selektorene: `cons`, `car` og `cdr`.
- Nå kan vi legge til to mutatorer: `set-car!` og `set-cdr!`

```
? (define foo (cons 'hoopy 'frood))
? foo → (hoopy . frood)
? (set-car! foo 42)
? foo →
```

Destruktive listeoperasjoner



- Fra før kjenner vi konstruktorene og selektorene: `cons`, `car` og `cdr`.
- Nå kan vi legge til to mutatorer: `set-car!` og `set-cdr!`

```
? (define foo (cons 'hoopy 'frood))
? foo → (hoopy . frood)
? (set-car! foo 42)
? foo → (42 . frood)
```

Destruktive listeoperasjoner

- Fra før kjenner vi konstruktorene og selektorene: `cons`, `car` og `cdr`.
- Nå kan vi legge til to mutatorer: `set-car!` og `set-cdr!`

```
? (define foo (cons 'hoopy 'frood))
? foo → (hoopy . frood)
? (set-car! foo 42)
? foo → (42 . frood)
? (set-cdr! foo 24)
```

Destruktive listeoperasjoner



- Fra før kjenner vi konstruktorene og selektorene: `cons`, `car` og `cdr`.
- Nå kan vi legge til to mutatorer: `set-car!` og `set-cdr!`

```
? (define foo (cons 'hoopy 'frood))
? foo → (hoopy . frood)
? (set-car! foo 42)
? foo → (42 . frood)
? (set-cdr! foo 24)
? foo →
```

Destruktive listeoperasjoner



- Fra før kjenner vi konstruktorene og selektorene: `cons`, `car` og `cdr`.
- Nå kan vi legge til to mutatorer: `set-car!` og `set-cdr!`

```
? (define foo (cons 'hoopy 'frood))
? foo → (hoopy . frood)
? (set-car! foo 42)
? foo → (42 . frood)
? (set-cdr! foo 24)
? foo → (42 . 24)
```

Destruktive listeoperasjoner



- Fra før kjenner vi konstruktorene og selektorene: `cons`, `car` og `cdr`.
- Nå kan vi legge til to mutatorer: `set-car!` og `set-cdr!`

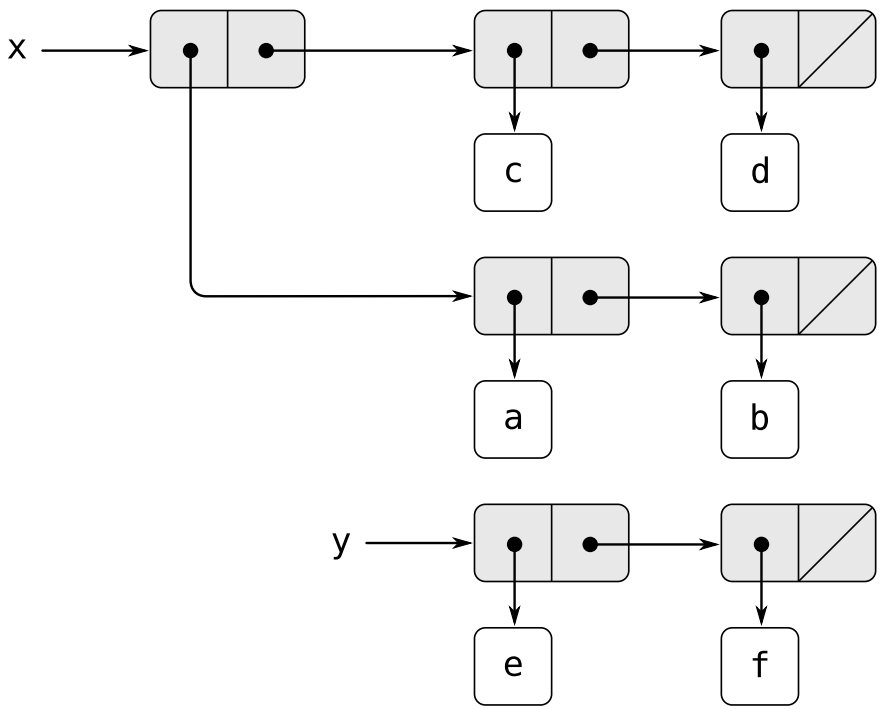
```
? (define foo (cons 'hoopy 'frood))
? foo → (hoopy . frood)
? (set-car! foo 42)
? foo → (42 . frood)
? (set-cdr! foo 24)
? foo → (42 . 24)
```

- `cons` lager *nye* par.
- `set-car!` og `set-cdr!` *endrer* verditilordninger innenfor par.
- Kan være hensiktsmessig for å minimere bruk av minne (plass): lar oss gjenbruke cons-celler.
- Åpner også nye muligheter for listebaserte datatyper.

cons

```
? x → ((a b) c d)

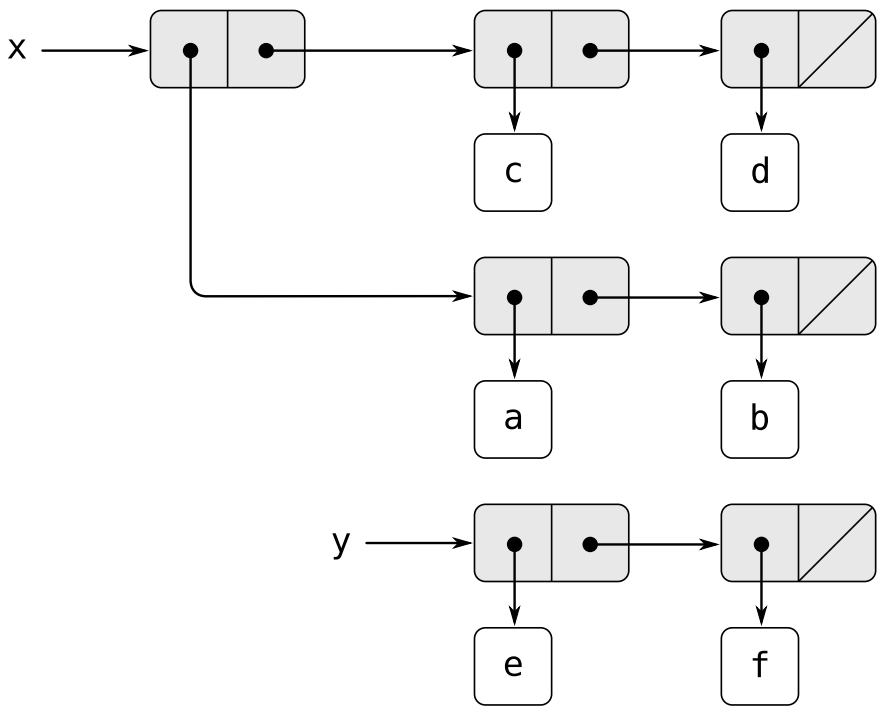
? y → (e f)
```



cons

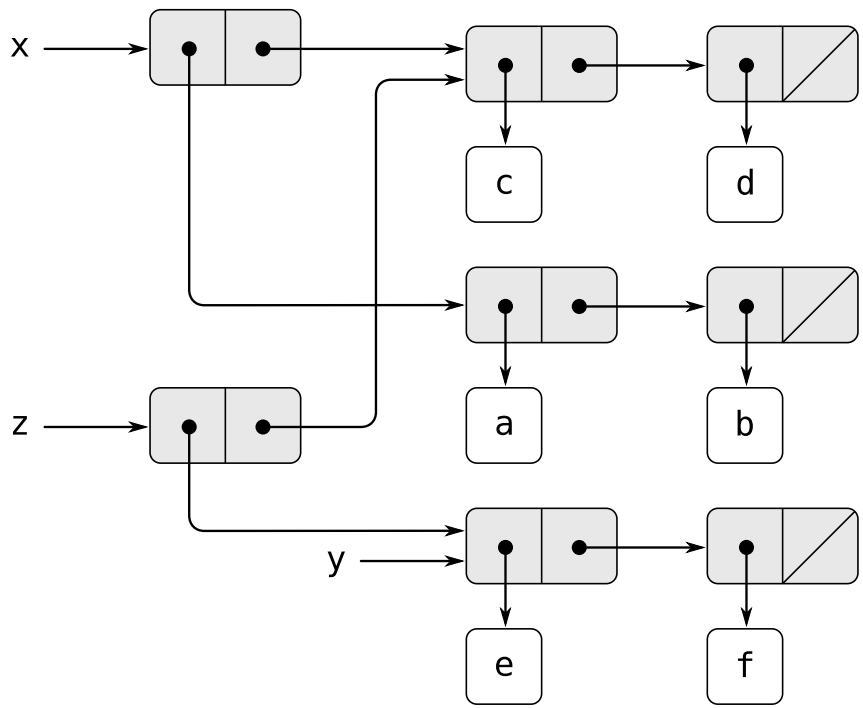
```
? x → ((a b) c d)

? y → (e f)
```




```
? (define z (cons y (cdr x)))

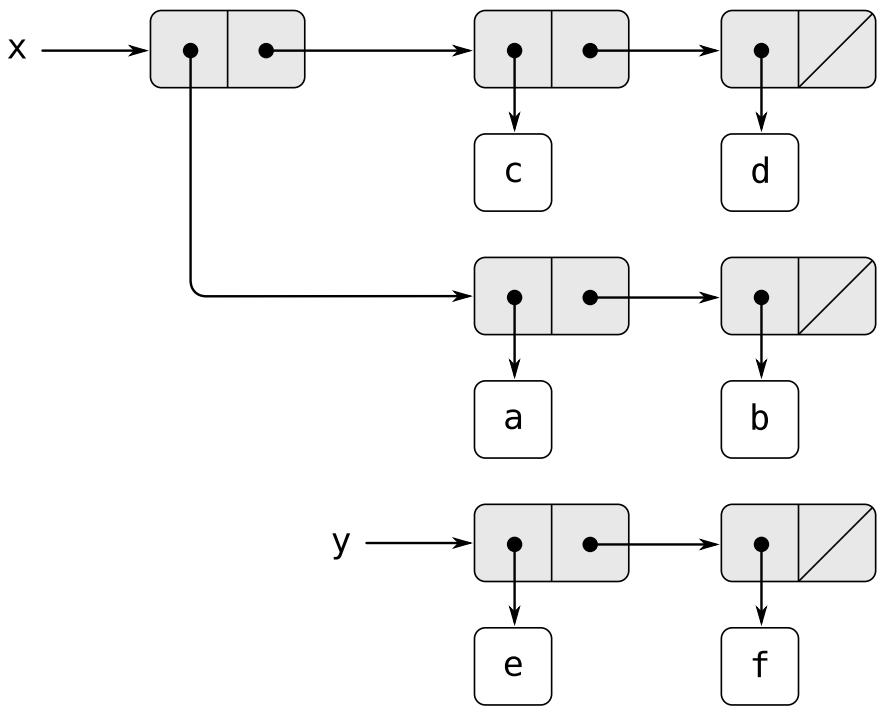
? z →
```



cons

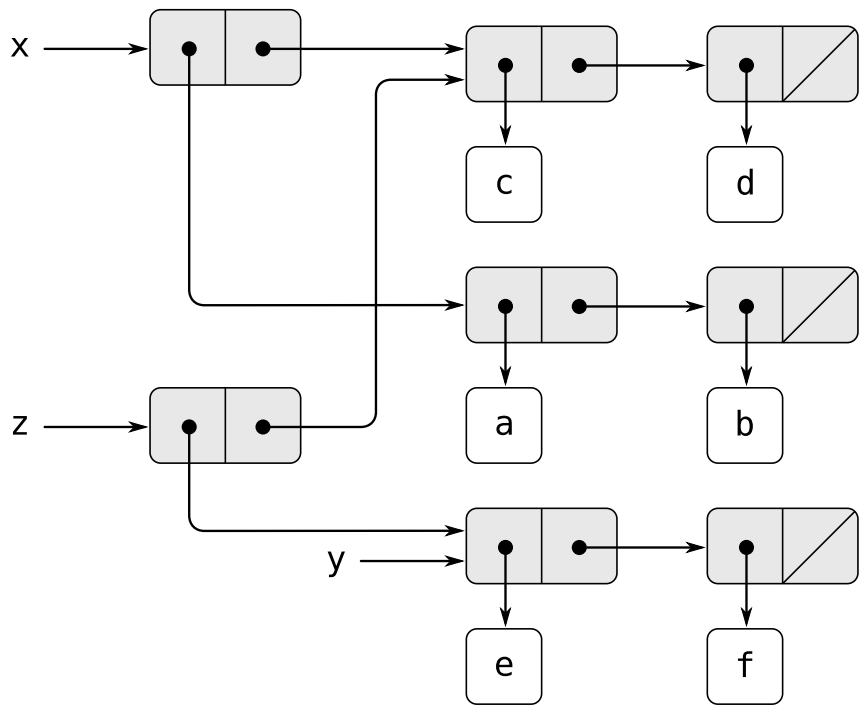
```
? x → ((a b) c d)

? y → (e f)
```




```
? (define z (cons y (cdr x)))

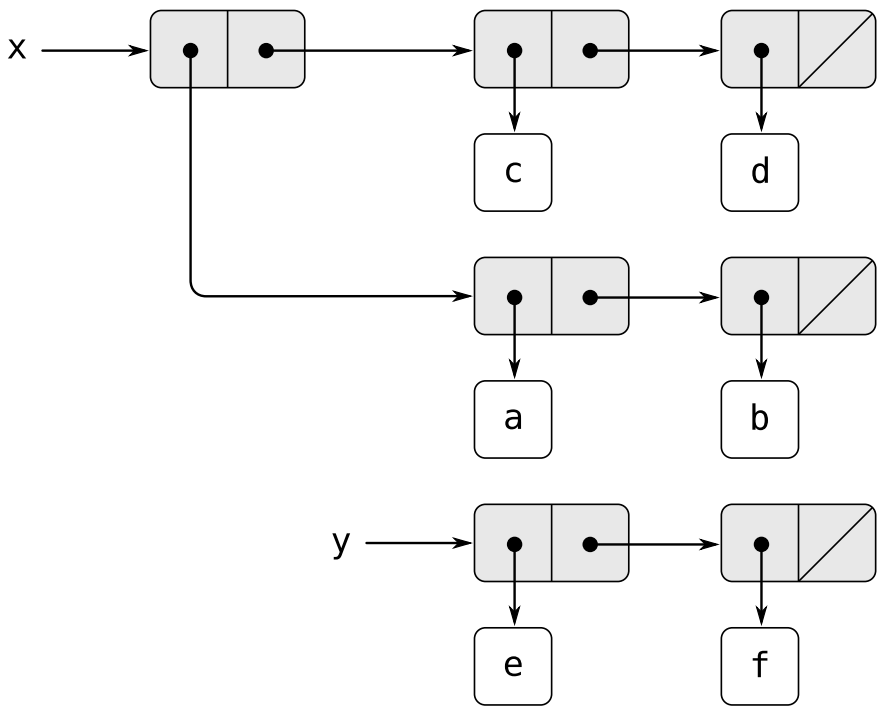
? z → ((e f) c d)
```



set-car!

```
? x → ((a b) c d)

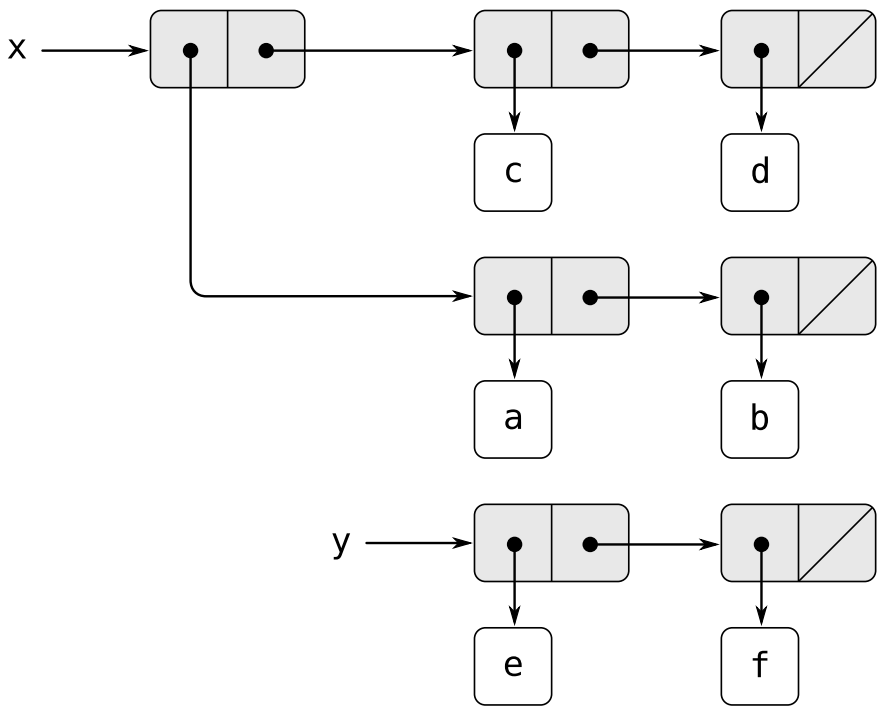
? y → (e f)
```



set-car!

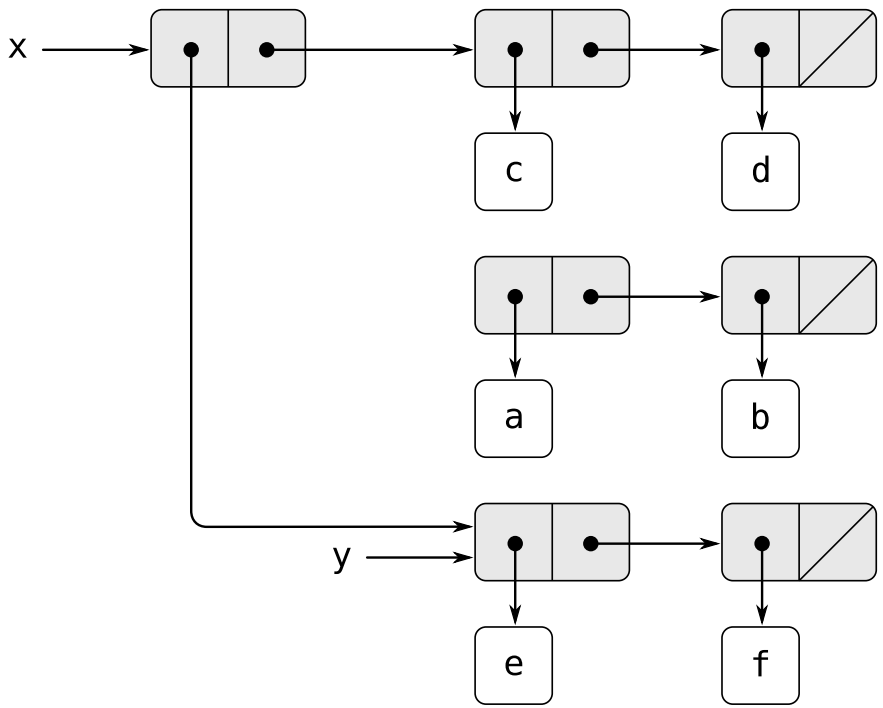
```
? x → ((a b) c d)

? y → (e f)
```




```
? (set-car! x y)

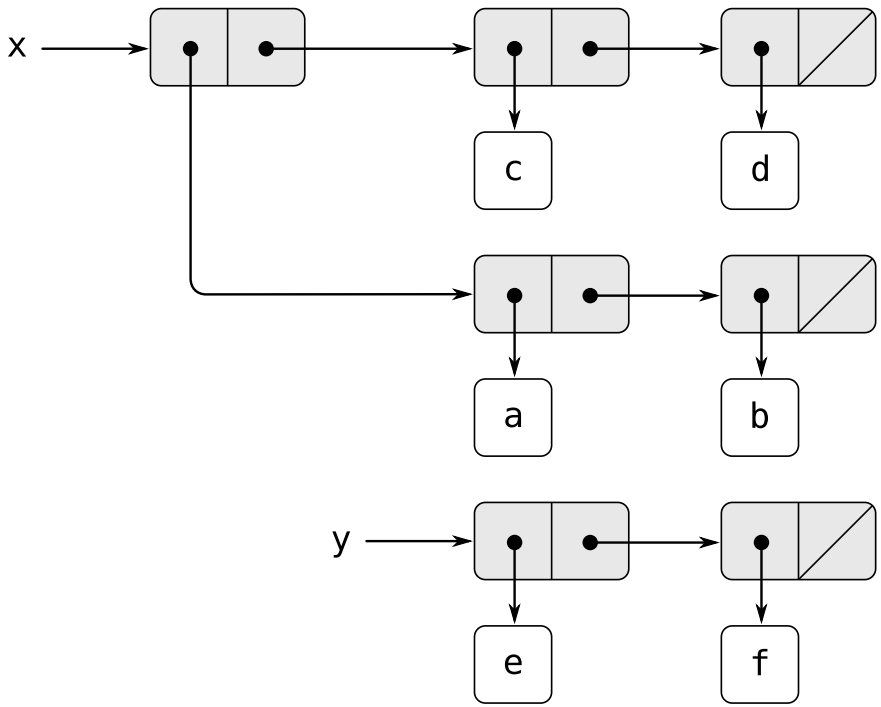
? x → ((e f) c d)
```



set-cdr!

```
? x → ((a b) c d)

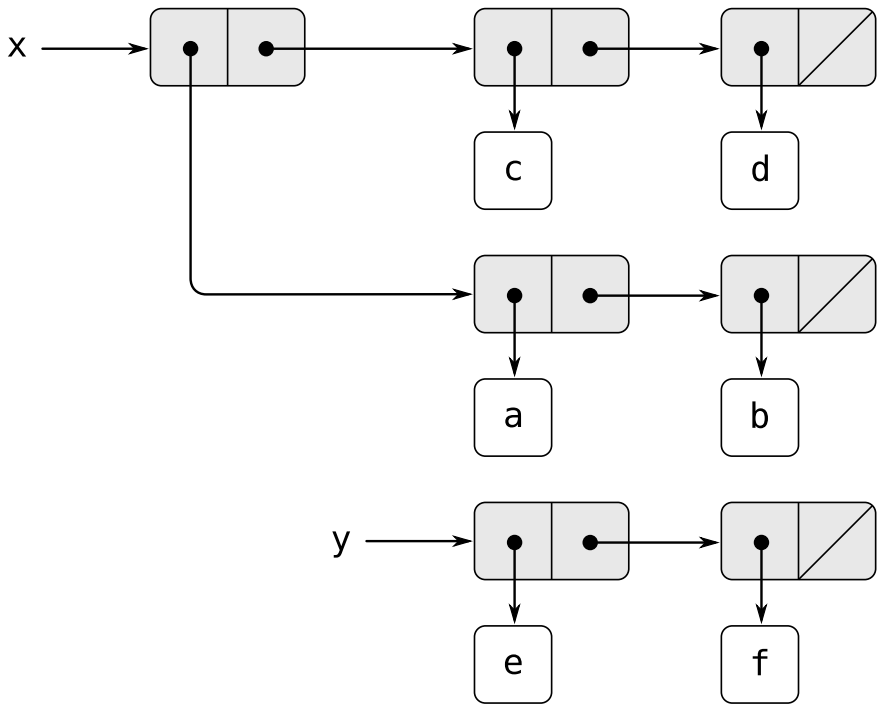
? y → (e f)
```



set-cdr!

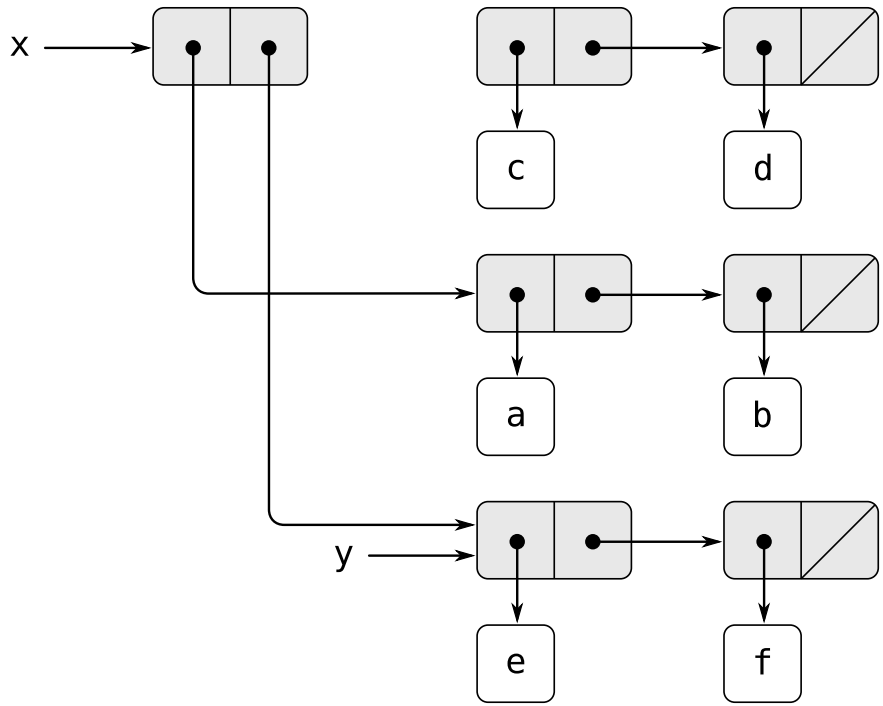
```
? x → ((a b) c d)

? y → (e f)
```




```
? (set-cdr! x y)

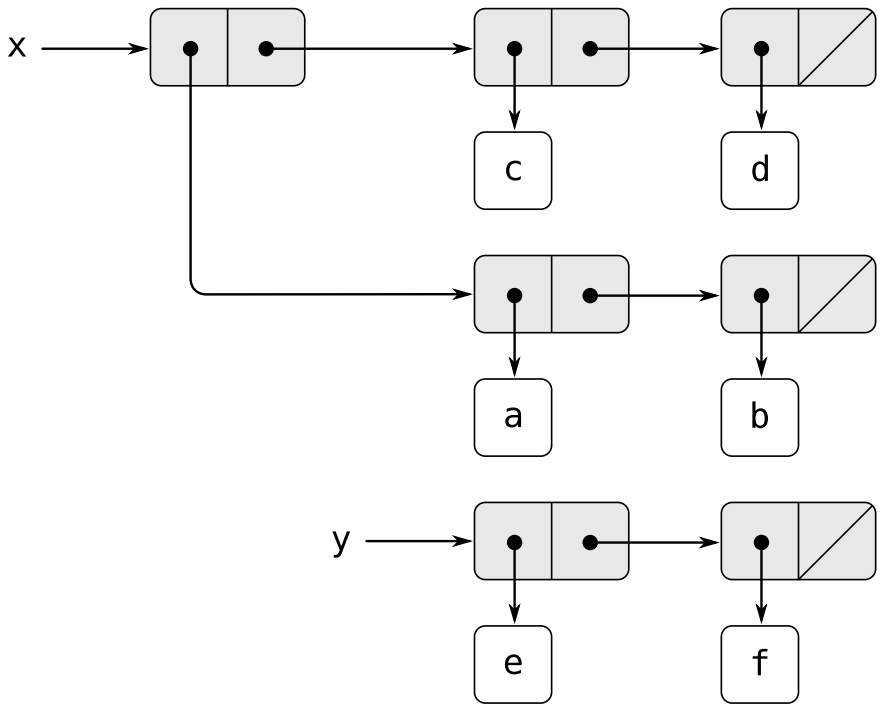
? x →
```



set-cdr!

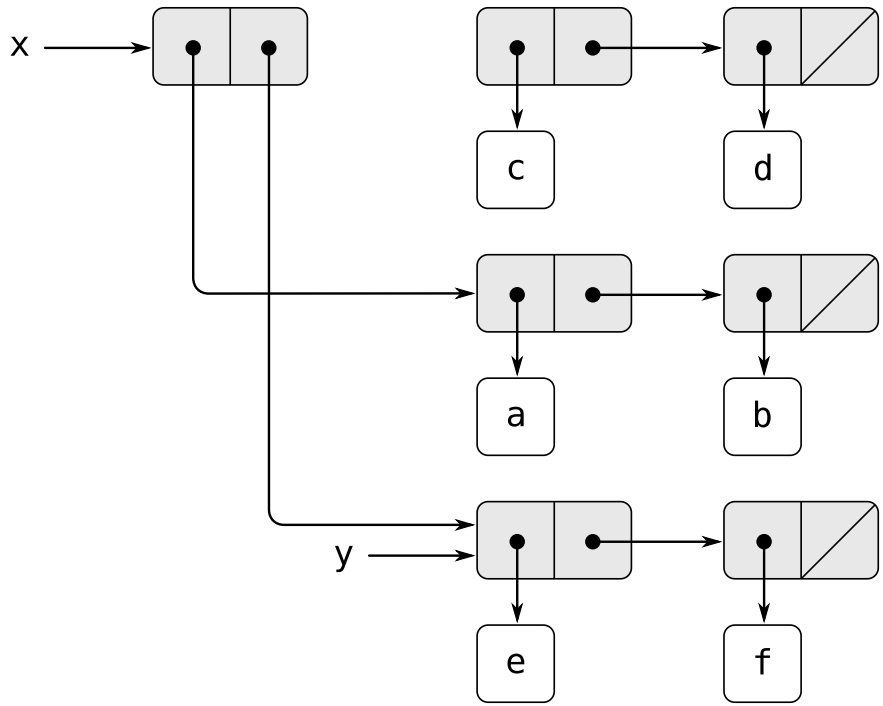
```
? x → ((a b) c d)

? y → (e f)
```




```
? (set-cdr! x y)

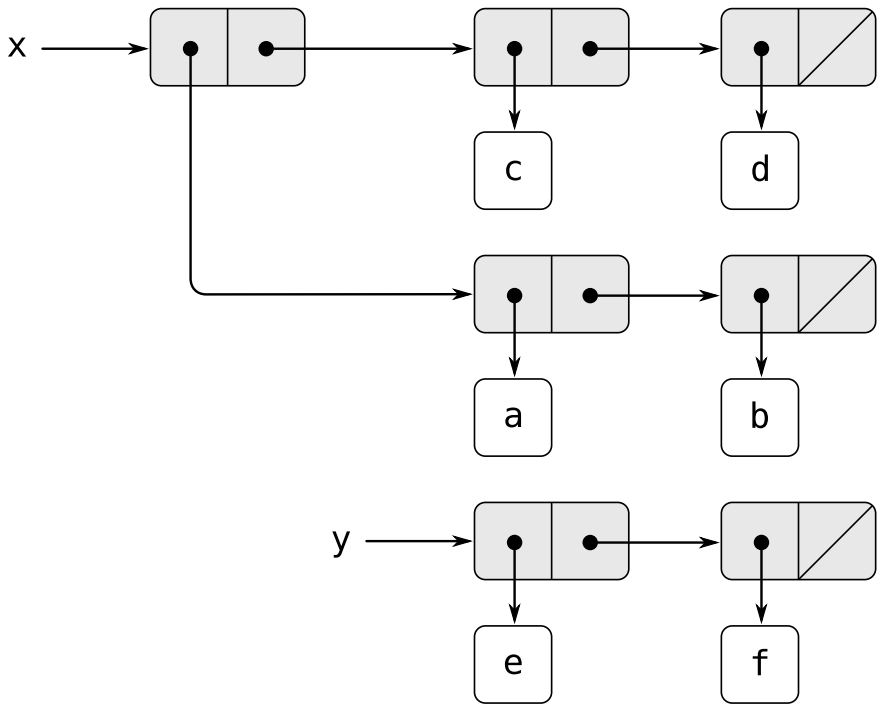
? x → ((a b) e f)
```



set-cdr!

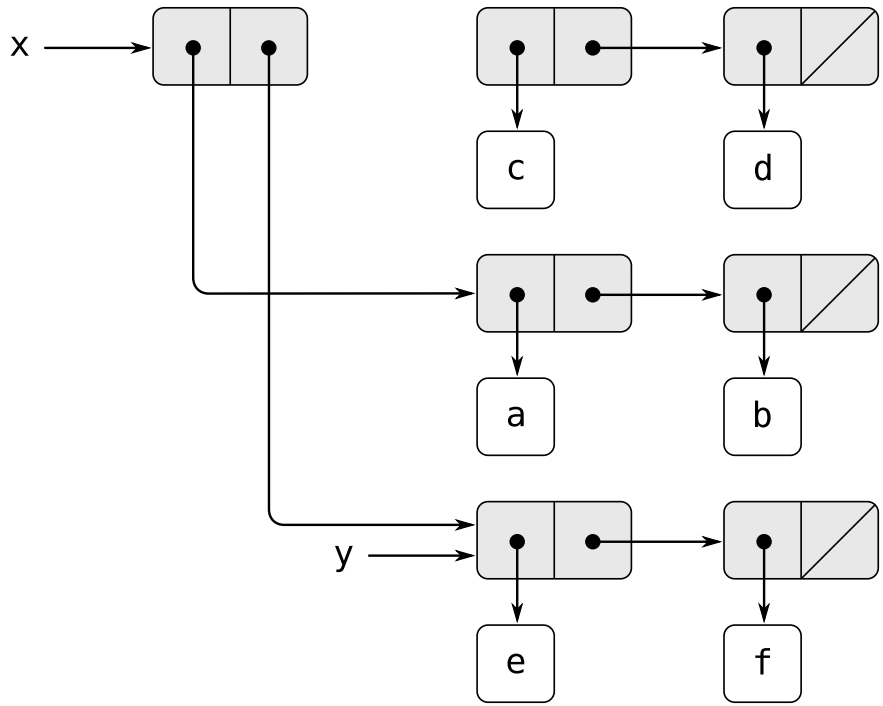
```
? x → ((a b) c d)

? y → (e f)
```




```
? (set-cdr! x y)

? x → ((a b) e f)
```



- Delstrukturen `(c d)` er ikke lenger lenket til noe → søppel (*garbage*).

Funksjonell konkatenasjon av lister

- `append` (innebygd) konkatenerer lister.
- Kopierer `foo` ved å `cons`'e elementene på `bar`.
- `append` er ikke destruktiv:
- `foo` og `bar` forblir uendret.

```
? (define foo '(1 2))
? (define bar '(3 4))
? (define baz (append foo bar))
```

Funksjonell konkatenasjon av lister

- `append` (innebygd) konkatenerer lister.
- Kopierer `foo` ved å `cons`'e elementene på `bar`.
- `append` er ikke destruktiv:
- `foo` og `bar` forblir uendret.

```
? (define foo '(1 2))
? (define bar '(3 4))
? (define baz (append foo bar))
? baz → (1 2 3 4)
? foo → (1 2)
? bar → (3 4)
```

Funksjonell konkatenasjon av lister

- `append` (innebygd) konkatenerer lister.
- Kopierer `foo` ved å `cons`'e elementene på `bar`.
- `append` er ikke destruktiv:
- `foo` og `bar` forblir uendret.

```
? (define foo '(1 2))

? (define bar '(3 4))

? (define baz (append foo bar))

? baz → (1 2 3 4)

? foo → (1 2)

? bar → (3 4)

(define (append x y)
```

Funksjonell konkatenasjon av lister

- `append` (innebygd) konkatenerer lister.
- Kopierer `foo` ved å `cons`'e elementene på `bar`.
- `append` er ikke destruktiv:
- `foo` og `bar` forblir uendret.

```
? (define foo '(1 2))

? (define bar '(3 4))

? (define baz (append foo bar))

? baz → (1 2 3 4)

? foo → (1 2)

? bar → (3 4)

(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
```

Konkatenasjon av lister (forts.)

- Aliasing / strukturdeling: append-resultatet har verdien til `bar` som delstruktur.
- Umerkelig hvis vi holder oss til funksjonell programmering.

En nærmere titt på append

```
? (define foo '(1 2))
? (define bar '(3 4))
? (define baz (append foo bar))
? baz → (1 2 3 4)
? foo → (1 2)
? bar → (3 4)
```

Konkatenasjon av lister (forts.)

- Aliasing / strukturdeling: append-resultatet har verdien til bar som delstruktur.
- Umerkelig hvis vi holder oss til funksjonell programmering.

En nærmere titt på append

```
? (define foo '(1 2))
? (define bar '(3 4))
? (define baz (append foo bar))
? baz → (1 2 3 4)
? foo → (1 2)
? bar → (3 4)
? (set-car! foo "ping")
? (set-car! bar "pong")
```

Konkatenasjon av lister (forts.)

- Aliasing / strukturdeling: append-resultatet har verdien til `bar` som delstruktur.
- Umerkelig hvis vi holder oss til funksjonell programmering.

En nærmere titt på append

```
? (define foo '(1 2))
? (define bar '(3 4))
? (define baz (append foo bar))
? baz → (1 2 3 4)
? foo → (1 2)
? bar → (3 4)
? (set-car! foo "ping")
? (set-car! bar "pong")
? baz →
```

Konkatenasjon av lister (forts.)

- Aliasing / strukturdeling: append-resultatet har verdien til `bar` som delstruktur.
- Umerkelig hvis vi holder oss til funksjonell programmering.

En nærmere titt på append

```
? (define foo '(1 2))
? (define bar '(3 4))
? (define baz (append foo bar))
? baz → (1 2 3 4)
? foo → (1 2)
? bar → (3 4)
? (set-car! foo "ping")
? (set-car! bar "pong")
? baz → (1 2 "pong" 4)
```

Konkatenasjon av lister (forts.)

- Aliasing / strukturdeling: append-resultatet har verdien til `bar` som delstruktur.
- Umerkelig hvis vi holder oss til funksjonell programmering.
- Hvor mange nye `cons`-celler lager append gitt to lister med henholdsvis lengde m og n?

En nærmere titt på append

```
? (define foo '(1 2))
? (define bar '(3 4))
? (define baz (append foo bar))
? baz → (1 2 3 4)
? foo → (1 2)
? bar → (3 4)
? (set-car! foo "ping")
? (set-car! bar "pong")
? baz → (1 2 "pong" 4)
```

Konkatenasjon av lister (forts.)

- Aliasing / strukturdeling: append-resultatet har verdien til `bar` som delstruktur.
- Umerkelig hvis vi holder oss til funksjonell programmering.
- Hvor mange nye `cons`-celler lager append gitt to lister med henholdsvis lengde m og n?
- Den første listen 'kopieres', det kreves m `cons`-celler.

En nærmere titt på append

```
? (define foo '(1 2))
? (define bar '(3 4))
? (define baz (append foo bar))
? baz → (1 2 3 4)
? foo → (1 2)
? bar → (3 4)
? (set-car! foo "ping")
? (set-car! bar "pong")
? baz → (1 2 "pong" 4)
```

Destruktiv listekonkatenasjon

- Ny destruktiv versjon:
- (verdien til) `foo` endres.
- Lager ingen nye `cons`-celler.
- Antar at det første argumentet er en ikke-tom liste.

```
(define (append! x y)
  (if (null? (cdr x))
      (set-cdr! x y)
      (append! (cdr x) y)))
```

Destruktiv listekonkatenasjon

- Ny destruktiv versjon:
- (verdien til) `foo` endres.
- Lager ingen nye `cons`-celler.
- Antar at det første argumentet er en ikke-tom liste.

```
(define (append! x y)
  (if (null? (cdr x))
      (set-cdr! x y)
      (append! (cdr x) y)))

? (define foo '(1 2))

? (define bar '(3 4))

? (append! foo bar)

? foo →
```

Destruktiv listekonkatenasjon

- Ny destruktiv versjon:
- (verdien til) `foo` endres.
- Lager ingen nye `cons`-celler.
- Antar at det første argumentet er en ikke-tom liste.

```
(define (append! x y)
  (if (null? (cdr x))
      (set-cdr! x y)
      (append! (cdr x) y)))

? (define foo '(1 2))

? (define bar '(3 4))

? (append! foo bar)

? foo → (1 2 3 4)
```

Destruktiv listekonkatenasjon

- Ny destruktiv versjon:
- (verdien til) `foo` endres.
- Lager ingen nye `cons`-celler.
- Antar at det første argumentet er en ikke-tom liste.

```
(define (append! x y)
  (if (null? (cdr x))
      (set-cdr! x y)
      (append! (cdr x) y)))

? (define foo '(1 2))

? (define bar '(3 4))

? (append! foo bar)

? foo → (1 2 3 4)
```

Enda mer *aliasing*

```
? (append! foo bar)

? foo →
```

Destruktiv listekonkatenasjon

- Ny destruktiv versjon:
- (verdien til) `foo` endres.
- Lager ingen nye `cons`-celler.
- Antar at det første argumentet er en ikke-tom liste.

```
(define (append! x y)
  (if (null? (cdr x))
      (set-cdr! x y)
      (append! (cdr x) y)))

? (define foo '(1 2))

? (define bar '(3 4))

? (append! foo bar)

? foo → (1 2 3 4)
```

Enda mer *aliasing*

```
? (append! foo bar)

? foo → (1 2 . #0=(3 4 . #0#))
```

Destruktiv listekonkatenasjon

- Ny destruktiv versjon:
- (verdien til) `foo` endres.
- Lager ingen nye `cons`-celler.
- Antar at det første argumentet er en ikke-tom liste.

```
(define (append! x y)
  (if (null? (cdr x))
      (set-cdr! x y)
      (append! (cdr x) y)))

? (define foo '(1 2))

? (define bar '(3 4))

? (append! foo bar)

? foo → (1 2 3 4)
```

Enda mer *aliasing*

```
? (append! foo bar)

? foo → (1 2 . #0=(3 4 . #0#))

? bar →
```

Destruktiv listekonkatenasjon

- Ny destruktiv versjon:
- (verdien til) `foo` endres.
- Lager ingen nye `cons`-celler.
- Antar at det første argumentet er en ikke-tom liste.

```
(define (append! x y)
  (if (null? (cdr x))
      (set-cdr! x y)
      (append! (cdr x) y)))

? (define foo '(1 2))

? (define bar '(3 4))

? (append! foo bar)

? foo → (1 2 3 4)
```

Enda mer *aliasing*

```
? (append! foo bar)

? foo → (1 2 . #0=(3 4 . #0#))

? bar → #0=(3 4 . #0#) ;; sirkulær!
```

Destruktiv listekonkatenasjon

- Ny destruktiv versjon:
- (verdien til) `foo` endres.
- Lager ingen nye `cons`-celler.
- Antar at det første argumentet er en ikke-tom liste.

```
(define (append! x y)
  (if (null? (cdr x))
      (set-cdr! x y)
      (append! (cdr x) y)))

? (define foo '(1 2))

? (define bar '(3 4))

? (append! foo bar)

? foo → (1 2 3 4)
```

Enda mer *aliasing*

```
? (append! foo bar)

? foo → (1 2 . #0=(3 4 . #0#))

? bar → #0=(3 4 . #0#) ;; sirkulær!

? (list? foo) →
```

Destruktiv listekonkatenasjon

- Ny destruktiv versjon:
- (verdien til) `foo` endres.
- Lager ingen nye `cons`-celler.
- Antar at det første argumentet er en ikke-tom liste.

```
(define (append! x y)
  (if (null? (cdr x))
      (set-cdr! x y)
      (append! (cdr x) y)))

? (define foo '(1 2))

? (define bar '(3 4))

? (append! foo bar)

? foo → (1 2 3 4)
```

Enda mer *aliasing*

```
? (append! foo bar)

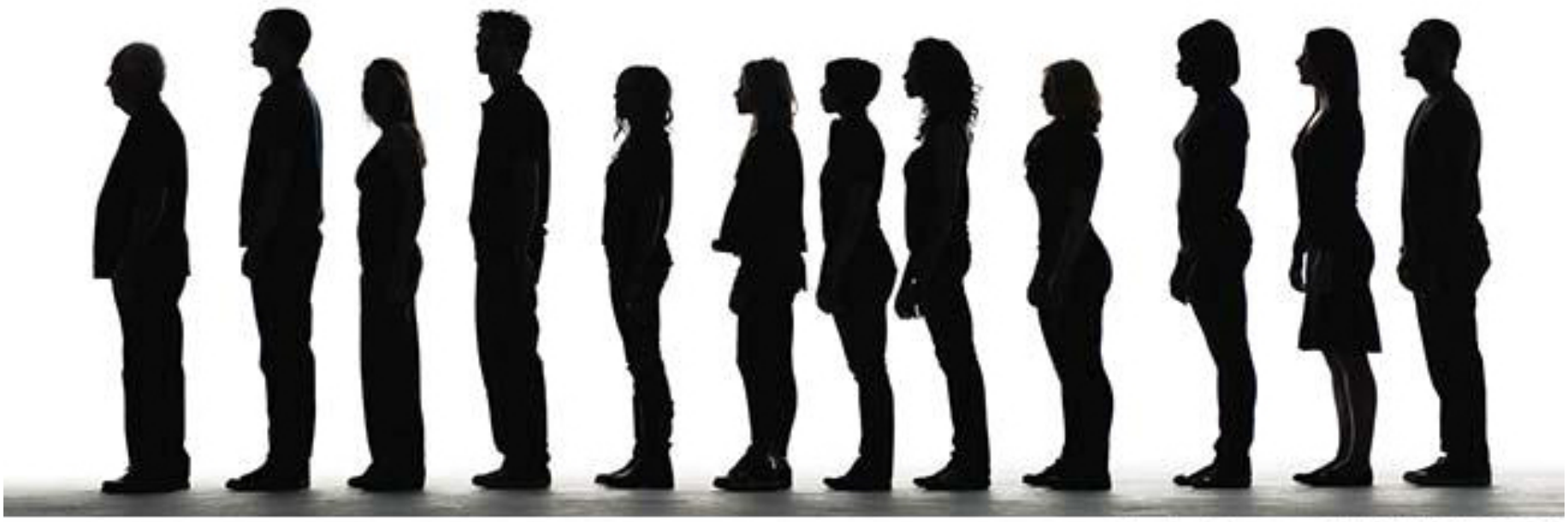
? foo → (1 2 . #0=(3 4 . #0#))

? bar → #0=(3 4 . #0#) ;; sirkulær!

? (list? foo) →  #f
```

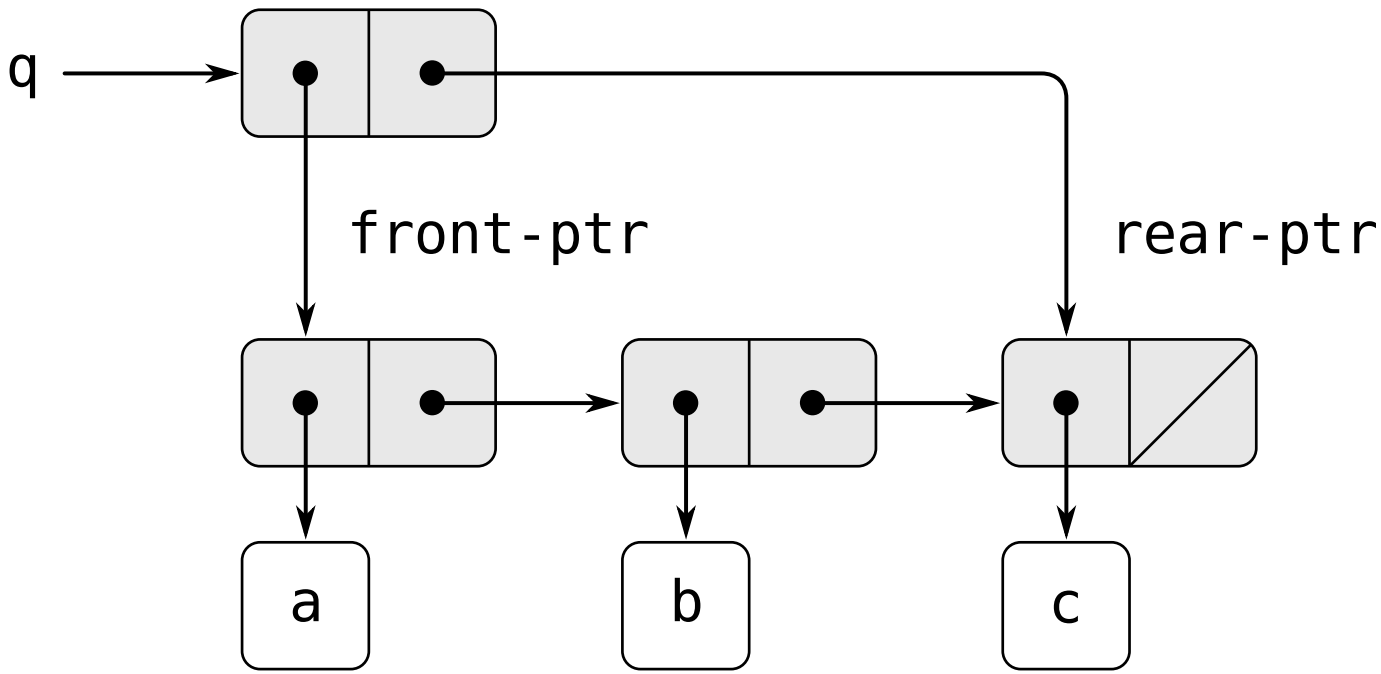
Egendefinerte datatyper

- Skal definere flere nye listebaserte muterbare datatyper.
- Motiv: bli mer fortrolig med bruk av destruktive listeoperasjoner + definere abstraksjonsbarrierer.
- Først ut: køer.



Køer

- En kø (*queue*) er en rettferdig datastruktur: *first in, first out* (FIFO).
- Elementer legges til på slutten, og kan kun fjernes fra fronten.
- Kan implementeres som liste, sammen med 'først'- og 'sist'-pekere.



- Kunne vi klart oss med *bare* en liste?

Køer: abstraksjonsbarriere (forts.)

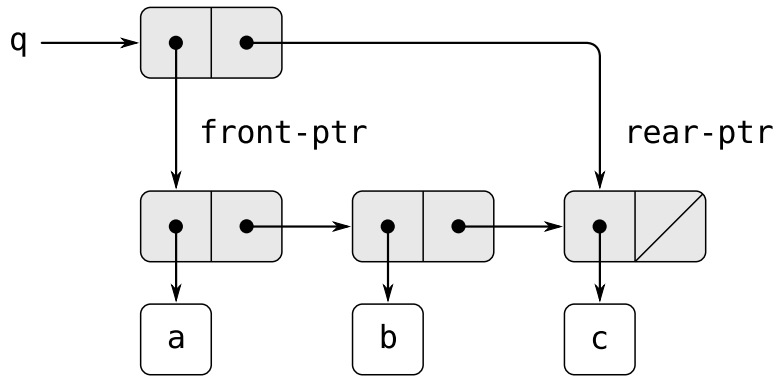


- `make-queue` returnerer en tom kø.
- `queue-empty?` tester om en gitt kø er tom.
- `queue-insert!` setter et nytt element inn i en kø.
- `queue-delete!` fjerner første element i køen og returnerer det.

Køer: abstraksjonsbarriere (forts.)

```
(define (make-queue)
    (cons '() '()))

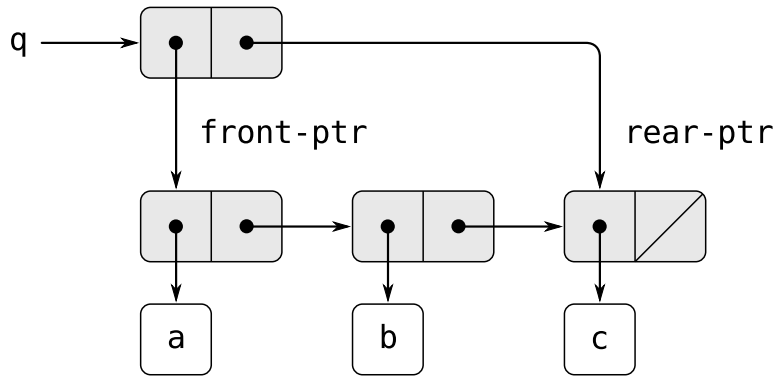
(define (queue-empty? queue)
  (null? (car queue)))
```



Køer: abstraksjonsbarriere (forts.)

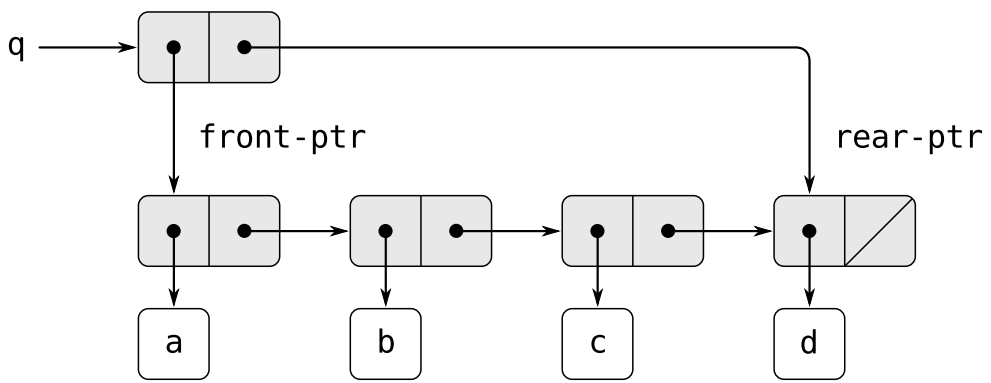
```
(define (make-queue)
    (cons '() '()))

(define (queue-empty? queue)
  (null? (car queue)))
```




```
(define (queue-insert! object queue)
  (let ((new (cons object '())))
    (if (queue-empty? queue)
      (set-car! queue new)
      (set-cdr! (cdr queue) new))
    (set-cdr! queue new)))

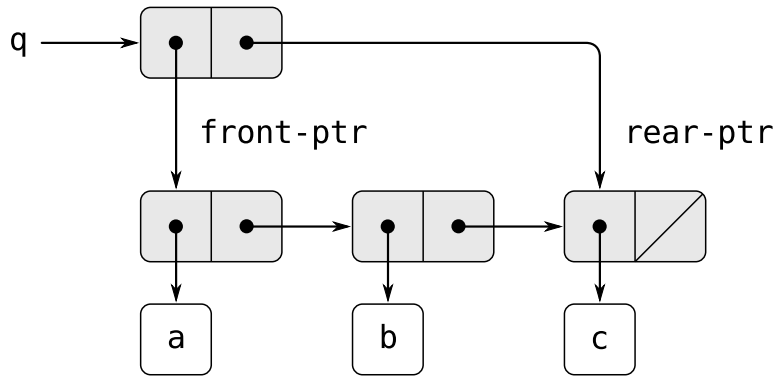
? (queue-insert! 'd q)
```



Køer: abstraksjonsbarriere (forts.)

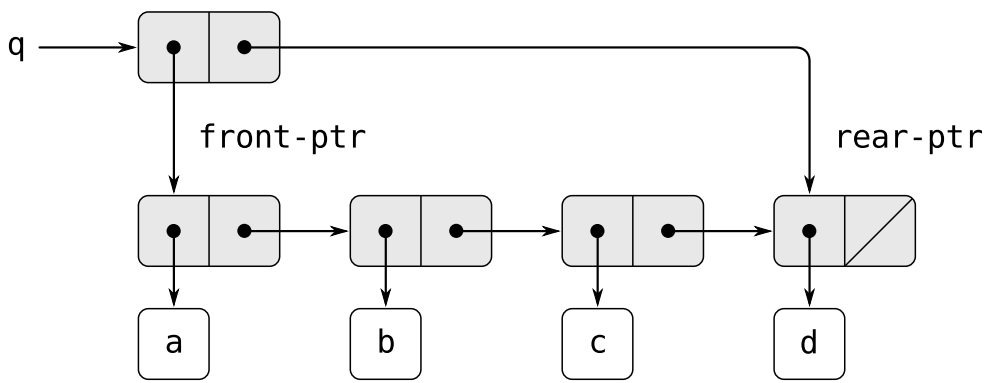
```
(define (make-queue)
    (cons '() '()))

(define (queue-empty? queue)
  (null? (car queue)))
```




```
(define (queue-insert! object queue)
  (let ((new (cons object '())))
    (if (queue-empty? queue)
      (set-car! queue new)
      (set-cdr! (cdr queue) new))
    (set-cdr! queue new)))

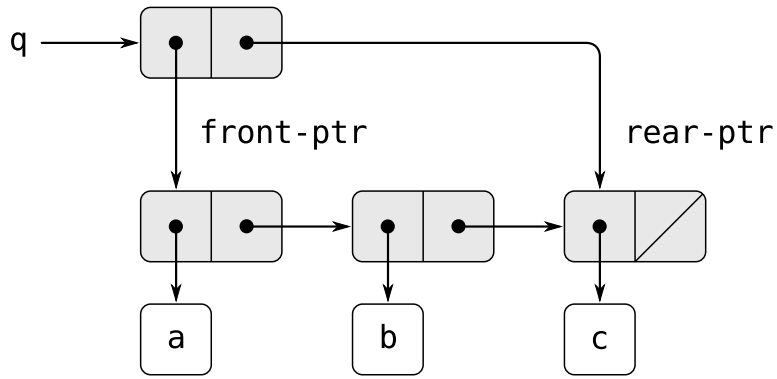
? (queue-insert! 'd q)
```



Køer: abstraksjonsbarriere (forts.)

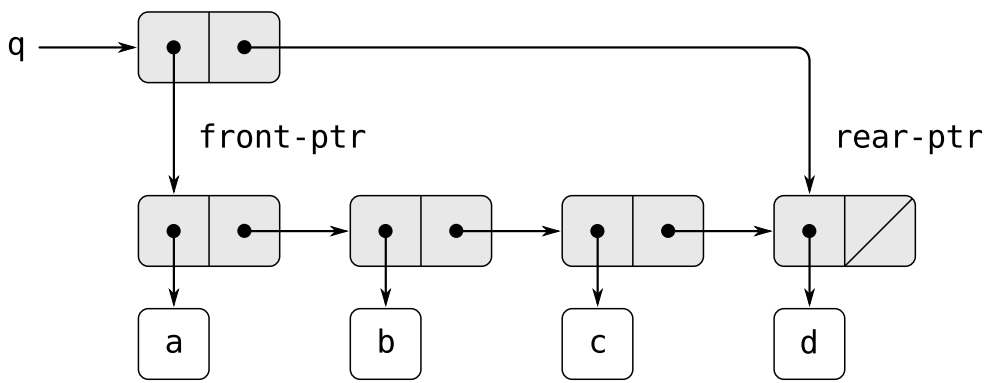
```
(define (make-queue)
    (cons '() '()))

(define (queue-empty? queue)
  (null? (car queue)))
```




```
(define (queue-insert! object queue)
  (let ((new (cons object '())))
    (if (queue-empty? queue)
      (set-car! queue new)
      (set-cdr! (cdr queue) new))
    (set-cdr! queue new)))

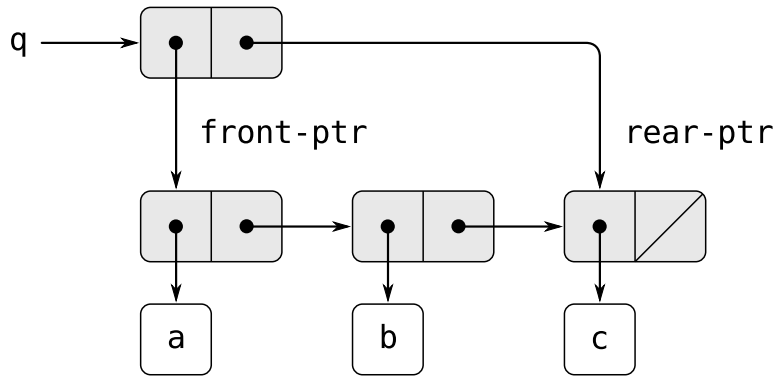
? (queue-insert! 'd q)
```



Køer: abstraksjonsbarriere (forts.)

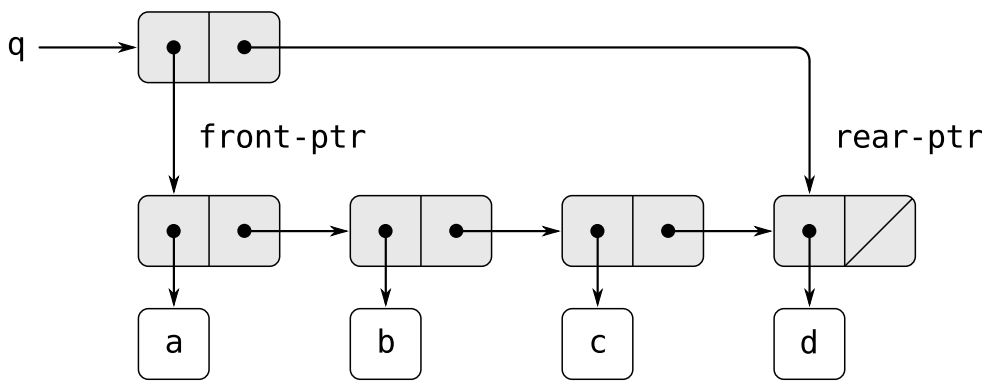
```
(define (make-queue)
    (cons '() '()))

(define (queue-empty? queue)
  (null? (car queue)))
```




```
(define (queue-insert! object queue)
  (let ((new (cons object '())))
    (if (queue-empty? queue)
      (set-car! queue new)
      (set-cdr! (cdr queue) new))
    (set-cdr! queue new)))

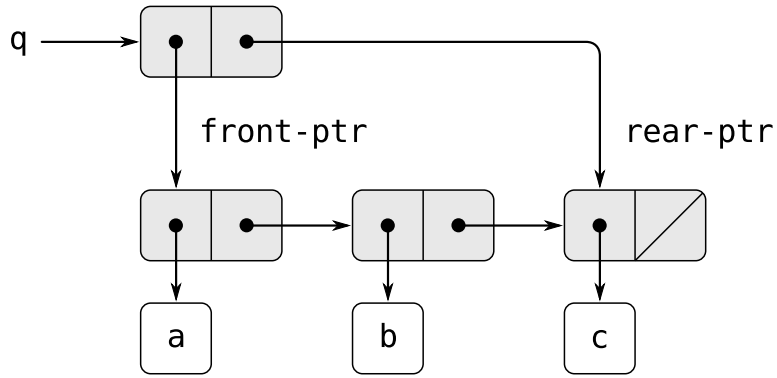
? (queue-insert! 'd q)
```



Køer: abstraksjonsbarriere (forts.)

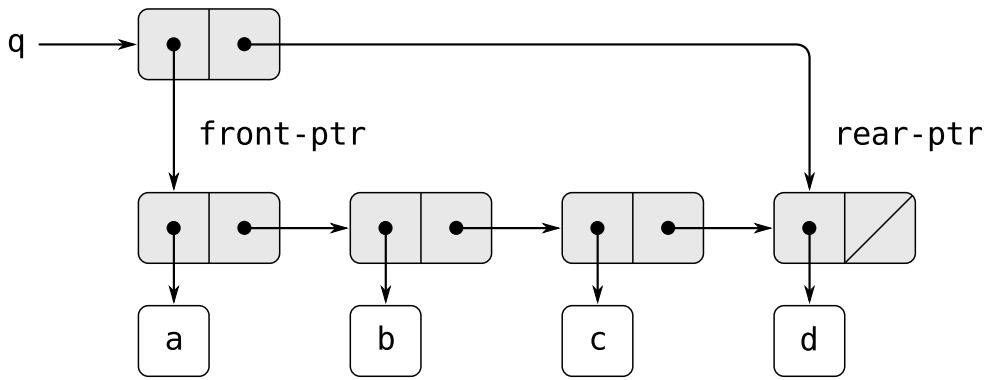
```
(define (make-queue)
    (cons '() '()))

(define (queue-empty? queue)
  (null? (car queue)))
```




```
(define (queue-insert! object queue)
  (let ((new (cons object '())))
    (if (queue-empty? queue)
      (set-car! queue new)
      (set-cdr! (cdr queue) new))
    (set-cdr! queue new)))

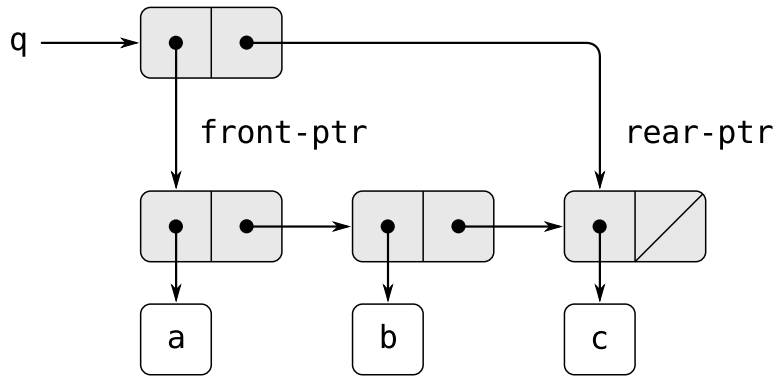
? (queue-insert! 'd q)
```



Køer: abstraksjonsbarriere (forts.)

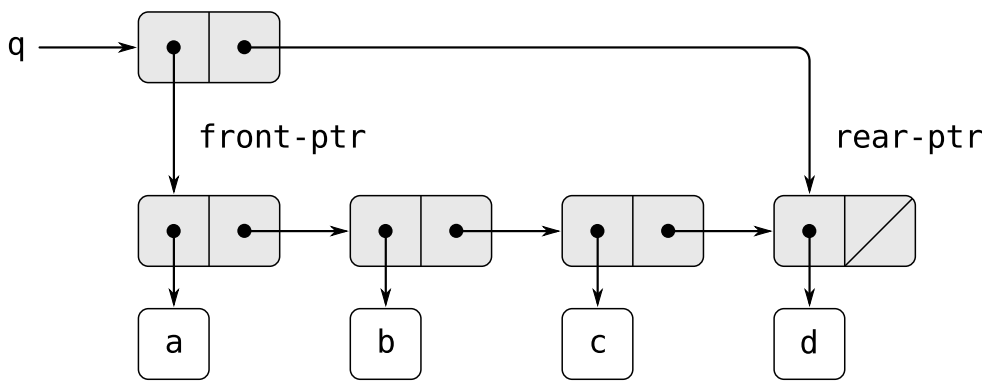
```
(define (make-queue)
    (cons '() '()))

(define (queue-empty? queue)
  (null? (car queue)))
```




```
(define (queue-insert! object queue)
  (let ((new (cons object '())))
    (if (queue-empty? queue)
      (set-car! queue new)
      (set-cdr! (cdr queue) new))
    (set-cdr! queue new)))

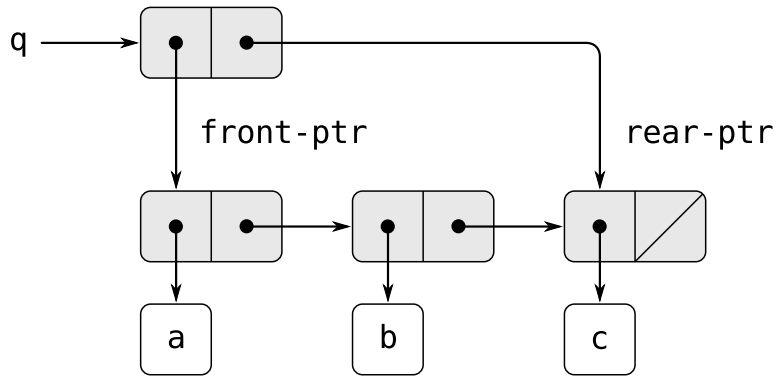
? (queue-insert! 'd q)
```



Køer: abstraksjonsbarriere (forts.)

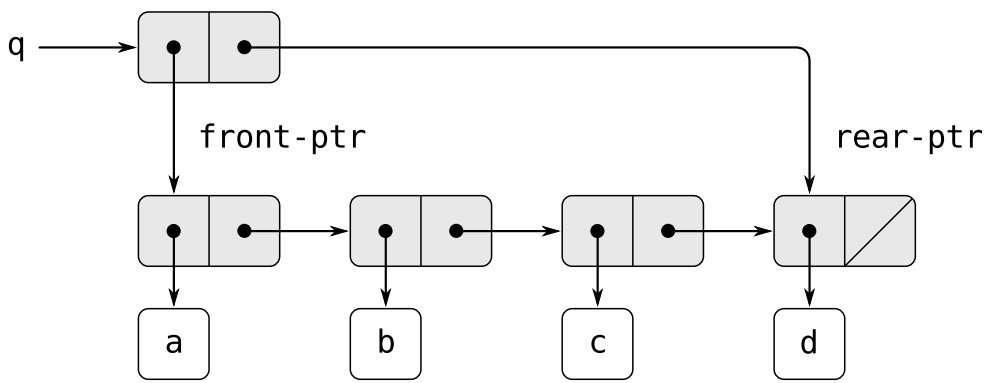
```
(define (make-queue)
    (cons '() '()))

(define (queue-empty? queue)
  (null? (car queue)))
```




```
(define (queue-insert! object queue)
  (let ((new (cons object '())))
    (if (queue-empty? queue)
      (set-car! queue new)
      (set-cdr! (cdr queue) new))
    (set-cdr! queue new)))

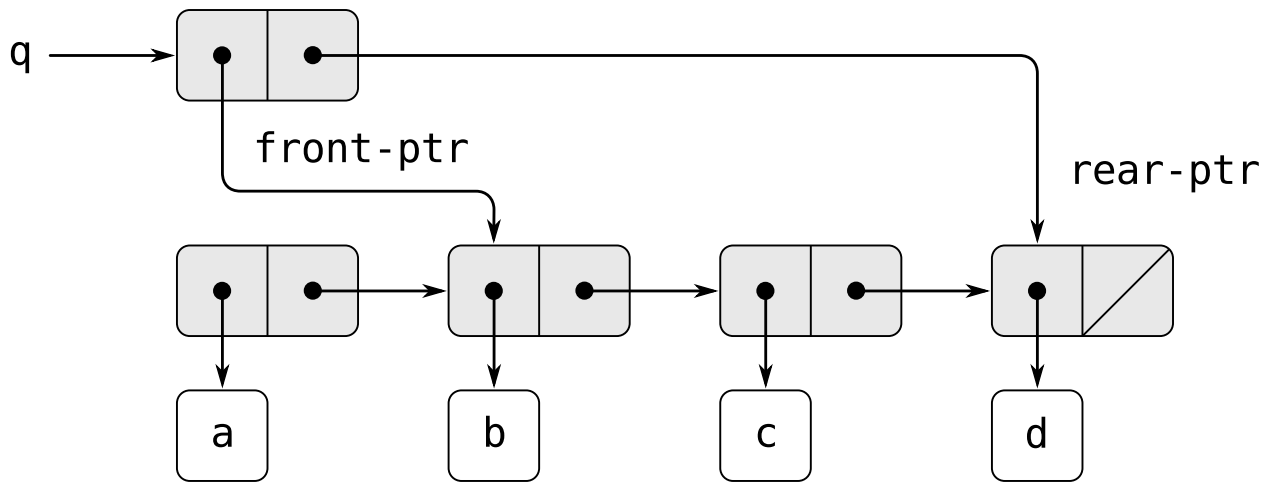
? (queue-insert! 'd q)
```



Køer: abstraksjonsbarriere (forts.)

```
(define (queue-delete! queue)
  (if (queue-empty? queue)
      "Error: empty queue."
      (let ((element (caar queue)))
        (set-car! queue (cdar queue))
        element)))
```

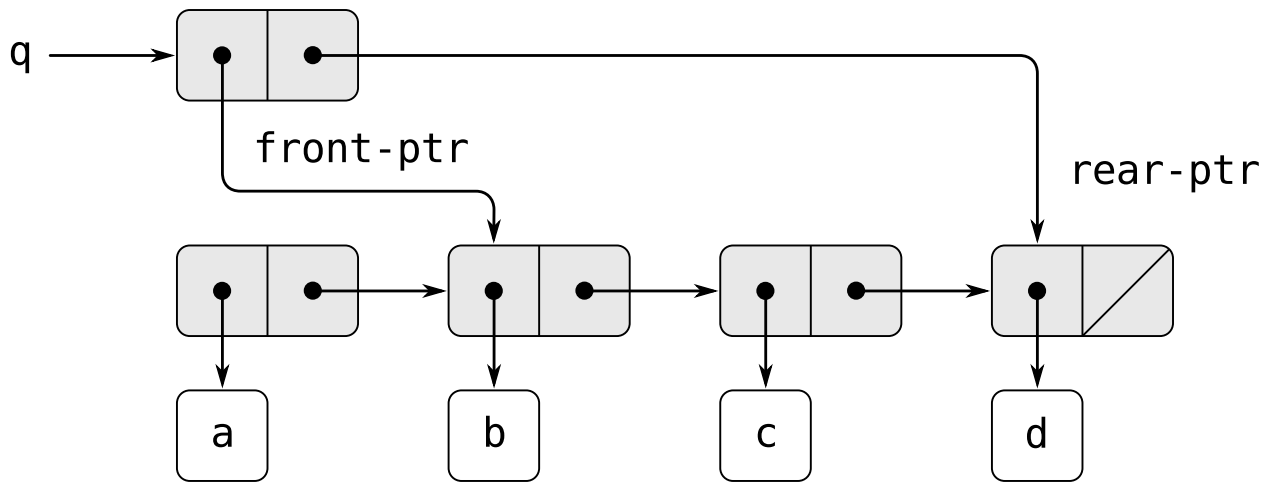
```
? (queue-delete! q) → a
```



Køer: abstraksjonsbarriere (forts.)

```
(define (queue-delete! queue)
  (if (queue-empty? queue)
      "Error: empty queue."
      (let ((element (caar queue)))
        (set-car! queue (cdar queue))
        element)))
```

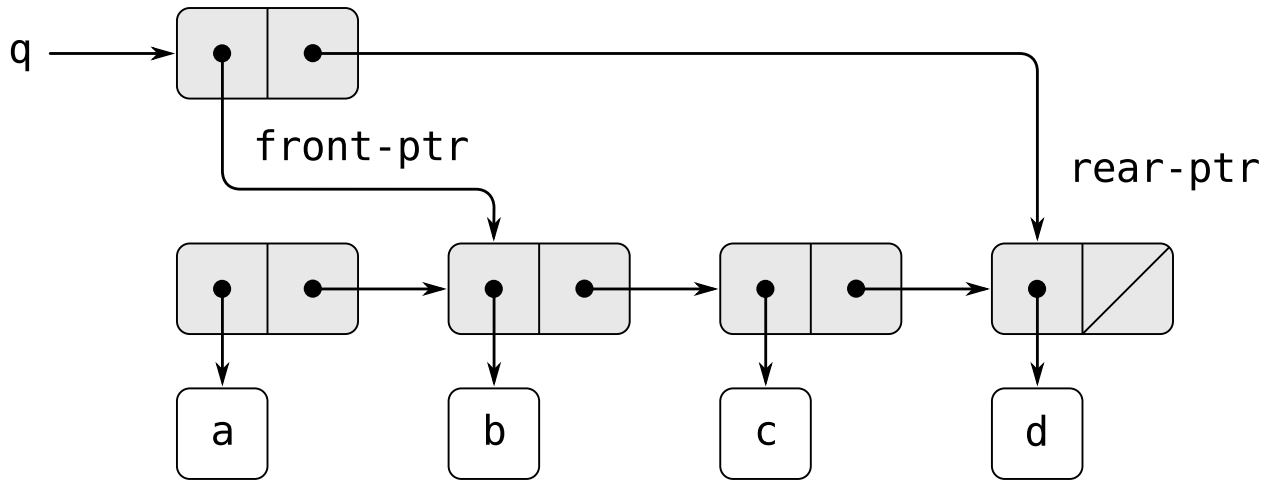
```
? (queue-delete! q) → a
```



Køer: abstraksjonsbarriere (forts.)

```
(define (queue-delete! queue)
  (if (queue-empty? queue)
      "Error: empty queue."
      (let ((element (caar queue)))
        (set-car! queue (cdar queue))
        element)))
```

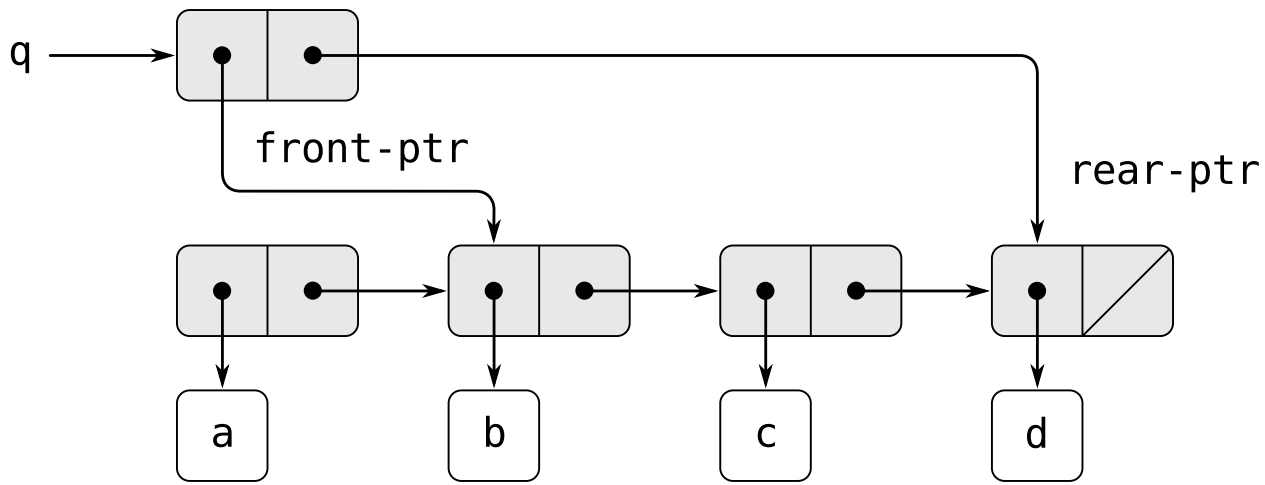
```
? (queue-delete! q) → a
```



Køer: abstraksjonsbarriere (forts.)

```
(define (queue-delete! queue)
  (if (queue-empty? queue)
      "Error: empty queue."
      (let ((element (caar queue)))
        (set-car! queue (cdar queue))
        element)))
```

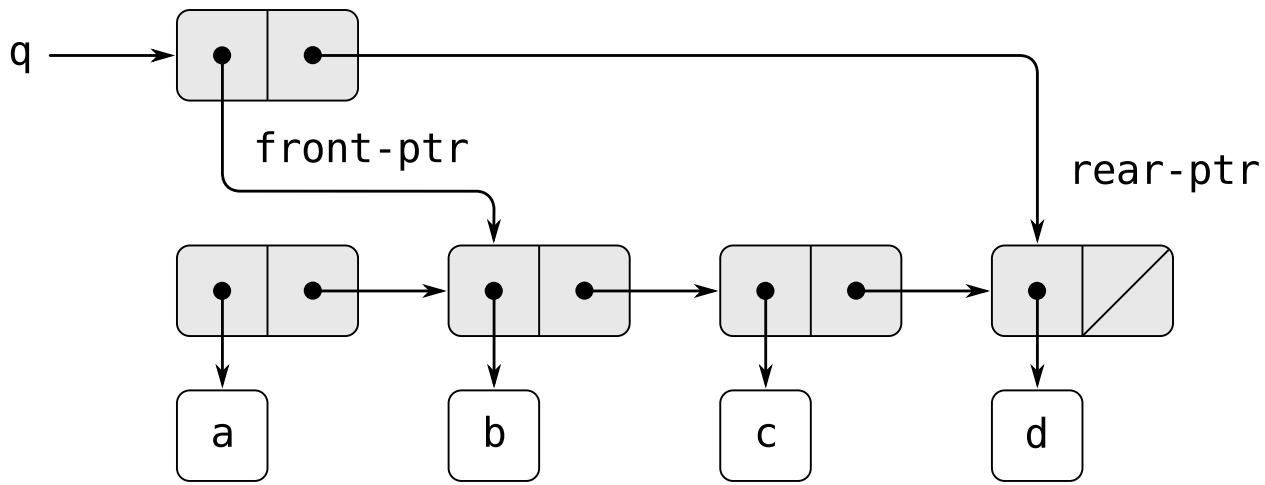
```
? (queue-delete! q) → a
```



Køer: abstraksjonsbarriere (forts.)

```
(define (queue-delete! queue)
  (if (queue-empty? queue)
      "Error: empty queue."
      (let ((element (caar queue)))
        (set-car! queue (cdar queue))
        element)))

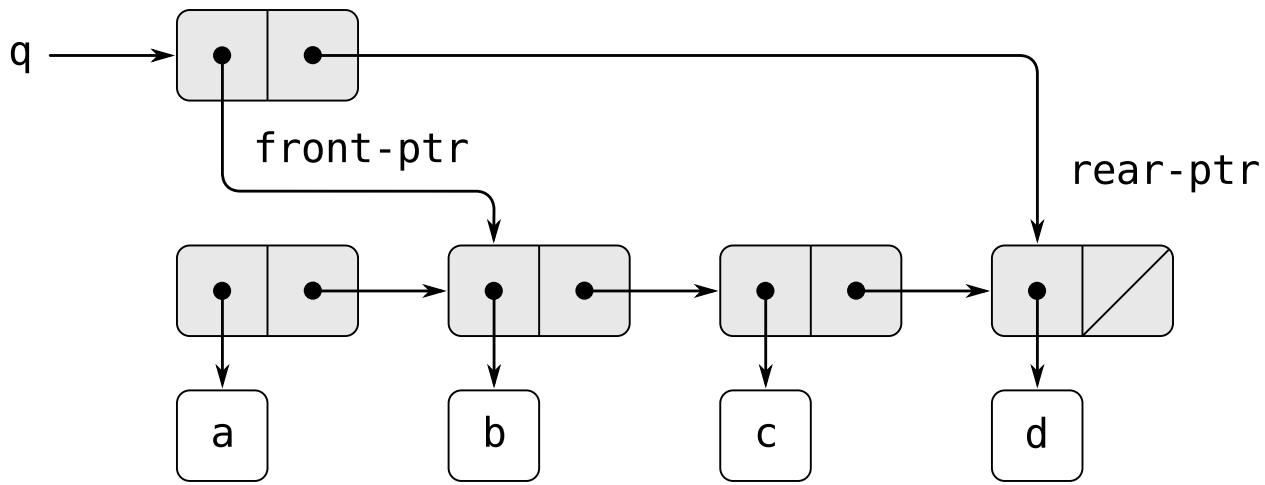
? (queue-delete! q) → a
```



Køer: abstraksjonsbarriere (forts.)

```
(define (queue-delete! queue)
  (if (queue-empty? queue)
      "Error: empty queue."
      (let ((element (caar queue)))
        (set-car! queue (cdar queue))
        element)))
```

```
? (queue-delete! q) → a
```



Vanlige prosedyrer eller unntak?

- "Special forms" eller vanlige prosedyrer?

```
? (define foo '(1 2))
? (set-car! foo 0)
? foo → (0 2)
```

```
? (set! foo 42)
? foo → 42
```

Vanlige prosedyrer eller unntak?

- "Special forms" eller vanlige prosedyrer?

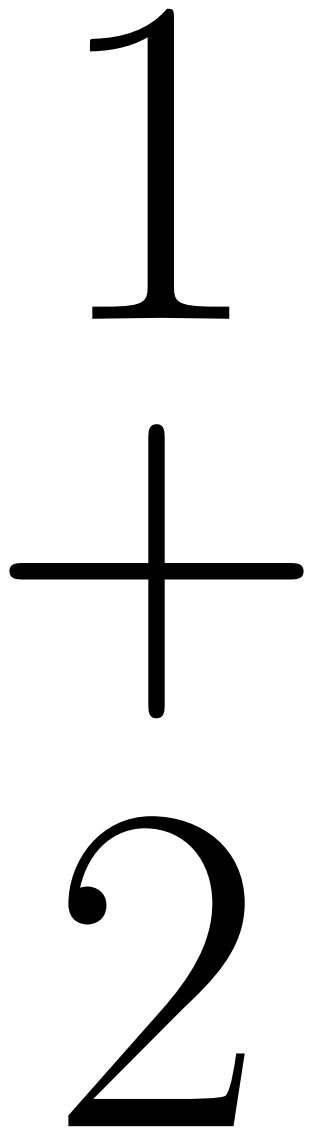
```
? (define foo '(1 2))
? (set-car! foo 0)
? foo → (0 2)
```

```
? (set! foo 42)
? foo → 42
```

- Kun `set!` er en "special form":
  foo evalueres ikke, vi ønsker å tilordne selve variabelen en *ny* verdi.
- `set-car!` og `set-cdr!` er vanlige prosedyrer:
  Det er verdiene i lista bundet til `foo` vi ønsker å endre, ikke `foo` selv.

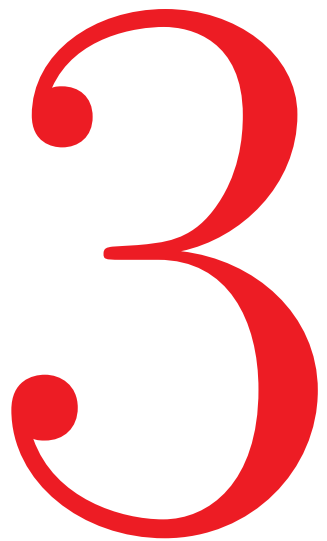
Frem til kapittel 3: Funksjonell programmering

- Frem til forrige uke hadde vi holdt oss til ren funksjonell programmering.
- Alltid samme resultat gitt samme argumenter.
- Beregninger utføres som funksjonelle transformasjoner av data (i stedet for sekvensielle endringer av tilstandsvariabler).
- En prosedyre kalles for sin returverdi alene (i stedet for bieffekter / *side-effects*).
- Semantikken til et uttrykk er uavhengig av hvor og når det brukes.
- I en forstand et *statisk* univers: forholdt oss ikke til *tid*.



Hva er skjedd i kapittel 3?

- Introduserer flere nye strategier for å organisere programmene våre.
- Fellesnevner: vi modellerer egenskaper som kan forandre seg med tid, tilstand.
- Første eksempel: objektorientering, implementert som prosedyrer:
- modellerer lokal tilstand med innkapsling + verditilordning.
- Destruktive operasjoner for å modifisere variabler og lister: muligheter for nye datatyper (muterbare).
- Trenger en ny modell for evaluering: omgivelsesmodellen.
- Videre: strømmer og utsatt evaluering.



Veien videre

- Neste forelesning(er):
  - Mer om destruktive listeoperasjoner.
  - Tabeller.
  - Memoisering.
  - Utsatt evaluering og strømmer.
- Oppgave (2b):
  - Omgivelsesdiagrammer.
  - Innkapsling og lokal tilstand.
  - Strukturdeling.
  - Abstrakte datatyper med mutatorer.
    - Stakker



