# IN2040: Funksjonell Programmering

## *Mer om verditilordning og muterbare data.*

Martin Steffen

Universitetet i Oslo

Uke #8 (11.10.2022)

# De siste ukene:

# De siste ukene: destruktive operasjoner


- `set!` endrer verditilordningen til et symbol.
- `set-car!` og `set-cdr!` endrer tilordninger innenfor et par.

# Repetisjon (fra tidligere): par som prosedyrer

```
? (define (cons x y)
    (lambda (m)
      (cond ((eq? m 'car) x)
            ((eq? m 'cdr) y))))

? (define (car p)
    (p 'car))

? (define (cdr p)
    (p 'cdr))

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

- Konstruktoren returnerer et prosedyreobjekt som innkapsler car og cdr-verdiene som lokale variabler.
- Selektorene kommuniserer med prosedyreobjektet via *message passing*.

# Repetisjon (fra tidligere): par som prosedyrer

```
? (define (cons x y)
    (lambda (m)
      (cond ((eq? m 'car) x)
            ((eq? m 'cdr) y))))

? (define (car p)
    (p 'car))

? (define (cdr p)
    (p 'cdr))

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

? foo → #<procedure>

? (car foo) → 1

? (car (cdr foo)) → 2
```

- Konstruktoren returnerer et prosedyreobjekt som innkapsler car og cdr-verdiene som lokale variabler.
- Selektorene kommuniserer med prosedyreobjektet via *message passing*.

# Repetisjon (fra tidligere): par som prosedyrer

```
? (define (cons x y)
    (lambda (m)
      (cond ((eq? m 'car) x)
            ((eq? m 'cdr) y))))

? (define (car p)
    (p 'car))

? (define (cdr p)
    (p 'cdr))

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

? foo → #<procedure>

? (car foo) → 1

? (car (cdr foo)) → 2
```

- Konstruktoren returnerer et prosedyreobjekt som innkapsler car og cdr-verdiene som lokale variabler.
- Selektorene kommuniserer med prosedyreobjektet via *message passing*.
- Hvordan få med mutatorer i abstraksjonsbarrieren?

# Vi legger til mutatorene i grensesnittet

- Tidligere eksempel på egendefinert muterbar datatype: bankkonto.
- Klarte oss med innkapsling + verditilordning med `set!`. Samme her:

# Vi legger til mutatorene i grensesnittet

- Tidligere eksempel på egendefinert muterbar datatype: bankkonto.
- Klarte oss med innkapsling + verditilordning med `set!`. Samme her:

```
? (define (cons x y)
    (lambda (m)
      (cond ((eq? m 'car) x)
            ((eq? m 'cdr) y)
            ((eq? m 'set-car!) ...)
            ((eq? m 'set-cdr!) ...))))

? (define (set-car! p v)
    ...)

? (define (set-cdr! p v)
    ...)

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

? (set-car! foo 42)

? (car foo) → 42
```

# Vi legger til mutatorene i grensesnittet

- Tidligere eksempel på egendefinert muterbar datatype: bankkonto.
- Klarte oss med innkapsling + verditilordning med `set!`. Samme her:

```
? (define (cons x y)
    (lambda (m)
      (cond ((eq? m 'car) x)
            ((eq? m 'cdr) y)
            ((eq? m 'set-car!) (lambda (v) (set! x v)))
            ((eq? m 'set-cdr!) (lambda (v) (set! y v))))))

? (define (set-car! p v)
    ((p 'set-car!) v))

? (define (set-cdr! p v)
    ((p 'set-cdr!) v))

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

? (set-car! foo 42)

? (car foo) → 42
```

# Et lite sidespor: Aritet

- Ariteten til en prosedyre refererer til antall argumenter den tar.
- Vi har allerede sett prosedyrer som kan ta
  - null argumenter (*niladic function*),
  - ett argument (unær / *monadic*),
  - to eller flere argumenter (*polyadic*):
    - to (binær / *dyadic*),
    - tre (ternær / *triadic*)
    - osv
  - et variabelt antall argumenter ($n$-ær / *variadic*).

```
? (+) → 0
? (+ 1 2) → 3
? (+ 1 2 3) → 6
? (+ 1 2 3 4 5 6 6 7 8 9 10) → 61
```

- Vi kan også definere våre egne $n$-ære prosedyrer.

# Variadiske prosedyrer

```
(define (sum . args)
  (define (recurse list)
    (if (null? list)
        0
        (+ (car list)
           (recurse (cdr list)))))
  (recurse args))

? (sum 1 2 3) → 6
```

- Prikk-notasjon: parameterene etter prikken er valgfrie og samles i en liste, her `args`.

# Variadiske prosedyrer

```
(define (sum . args)
  (define (recurse list)
    (if (null? list)
        0
        (+ (car list)
           (recurse (cdr list)))))
  (recurse args))

? (sum 1 2 3) → 6
```

- Prikk-notasjon: parameterene etter prikken er valgfrie og samles i en liste, her `args`.
- Hvorfor trengs egentlig den indre rekursive prosedyren?

# Variadiske prosedyrer

```
(define (sum . args)
  (define (recurse list)
    (if (null? list)
        0
        (+ (car list)
           (recurse (cdr list)))))
  (recurse args))

? (sum 1 2 3) → 6
```

- Prikk-notasjon: parameterene etter prikken er valgfrie og samles i en liste, her `args`.
- Hvorfor trengs egentlig den indre rekursive prosedyren?

```
(define (sum . args)
  (if (null? args)
      0
      (+ (car args)
         (sum (cdr args)))))

? (sum 1 2 3) → ???
```

- Hva vil skje her?
- Hva er verdien til parameteret `args` i 1. og 2. oppkalling?

# Variadiske prosedyrer

```
(define (sum . args)
  (if (null? args)
      0
      (+ (car args)
         (apply sum (cdr args)))))

? (sum 1 2 3) → 6
```

- En annen versjon, med `apply`.

# Variadiske prosedyrer

```
(define (sum . args)
  (if (null? args)
      0
      (+ (car args)
         (apply sum (cdr args)))))

? (sum 1 2 3) → 6
```

- En annen versjon, med `apply`.

```
? (apply + '(1 2 3))
→  6

? (cons + '(1 2 3))
→ (#<procedure:+> 1 2 3)
```

- `apply` anvender en prosedyre på en liste av argumenter.

# Variadiske prosedyrer

```
(define (sum . args)
  (if (null? args)
      0
      (+ (car args)
         (apply sum (cdr args)))))

? (sum 1 2 3) → 6
```

- En annen versjon, med `apply`.

```
? (apply + '(1 2 3))
→  6

? (cons + '(1 2 3))
→ (#<procedure:+> 1 2 3)
```

- `apply` anvender en prosedyre på en liste av argumenter.

```
(define (sum . args)
  (apply + args))
```

- Hvis vi ikke hadde ønsket å illustrere rekursjonen.

# Mer om prikk-notasjon, for de spesielt interesserte

```
? (cons 1 (cons 2 3)) → (1 2 . 3)

? (cons 1 (cons 2 (cons 3 '())))
→ (1 2 3)
```

- `'(1 2)` = cons-kjeden `'(1 . (2 . ()))`
- Er `cdr` et par eller `()` sløyfer Scheme punktum og parentes: " `. (`".

# Mer om prikk-notasjon, for de spesielt interesserte

```
? (cons 1 (cons 2 3)) → (1 2 . 3)

? (cons 1 (cons 2 (cons 3 '())))
→ (1 2 3)

? '(1 . (2 . 3)) →
```

- `'(1 2)` = cons-kjeden `'(1 . (2 . ()))`
- Er `cdr` et par eller `()` sløyfer Scheme punktum og parentes: " `. (`".
- Scheme kan også lese slik "dot-notasjon".

# Mer om prikk-notasjon, for de spesielt interesserte

```
? (cons 1 (cons 2 3)) → (1 2 . 3)

? (cons 1 (cons 2 (cons 3 '())))
→ (1 2 3)

? '(1 . (2 . 3)) → (1 2 . 3)
```

- `'(1 2)` = cons-kjeden `'(1 . (2 . ()))`
- Er `cdr` et par eller `()` sløyfer Scheme punktum og parentes: " `. (`".
- Scheme kan også lese slik "dot-notasjon".

# Mer om prikk-notasjon, for de spesielt interesserte

```
? (cons 1 (cons 2 3)) → (1 2 . 3)

? (cons 1 (cons 2 (cons 3 '())))
→ (1 2 3)

? '(1 . (2 . 3)) → (1 2 . 3)

? '(1 . (2 . (3 . ()))) →
```

- `'(1 2)` = cons-kjeden `'(1 . (2 . ()))`
- Er `cdr` et par eller `()` sløyfer Scheme punktum og parentes: " `. (`".
- Scheme kan også lese slik "dot-notasjon".

# Mer om prikk-notasjon, for de spesielt interesserte

```
? (cons 1 (cons 2 3)) → (1 2 . 3)

? (cons 1 (cons 2 (cons 3 '())))
→ (1 2 3)

? '(1 . (2 . 3)) → (1 2 . 3)

? '(1 . (2 . (3 . ()))) → (1 2 3)
```

- `'(1 2)` = cons-kjeden `'(1 . (2 . ()))`
- Er `cdr` et par eller `()` sløyfer Scheme punktum og parentes: " `. (`".
- Scheme kan også lese slik "dot-notasjon".

# Mer om prikk-notasjon, for de spesielt interesserte

```
? (cons 1 (cons 2 3)) → (1 2 . 3)

? (cons 1 (cons 2 (cons 3 '())))
→ (1 2 3)

? '(1 . (2 . 3)) → (1 2 . 3)

? '(1 . (2 . (3 . ()))) → (1 2 3)

? (+ 1 2) → 3
```

- `'(1 2)` = cons-kjeden `'(1 . (2 . ()))`
- Er `cdr` et par eller `()` sløyfer Scheme punktum og parentes: " `. (`".
- Scheme kan også lese slik "dot-notasjon".
- Husk at kode = lister.

# Mer om prikk-notasjon, for de spesielt interesserte

```
? (cons 1 (cons 2 3)) → (1 2 . 3)

? (cons 1 (cons 2 (cons 3 '())))
→ (1 2 3)

? '(1 . (2 . 3)) → (1 2 . 3)

? '(1 . (2 . (3 . ()))) → (1 2 3)

? (+ 1 2) → 3

? (+ . (1 . (2 . ()))) → 3
```

- `'(1 2)` = cons-kjeden `'(1 . (2 . ()))`
- Er `cdr` et par eller `()` sløyfer Scheme punktum og parentes: " `. (`".
- Scheme kan også lese slik "dot-notasjon".
- Husk at kode = lister.

# Mer om prikk-notasjon, for de spesielt interesserte

```
? (cons 1 (cons 2 3)) → (1 2 . 3)

? (cons 1 (cons 2 (cons 3 '())))
→ (1 2 3)

? '(1 . (2 . 3)) → (1 2 . 3)

? '(1 . (2 . (3 . ()))) → (1 2 3)

? (+ 1 2) → 3

? (+ . (1 . (2 . ()))) → 3

? (define (sum . args)
    ...)
```

- `'(1 2)` = cons-kjeden `'(1 . (2 . ()))`
- Er `cdr` et par eller `()` sløyfer Scheme punktum og parentes: " `. (`".
- Scheme kan også lese slik "dot-notasjon".
- Husk at kode = lister.

# Mer om prikk-notasjon, for de spesielt interesserte

```
? (cons 1 (cons 2 3)) → (1 2 . 3)

? (cons 1 (cons 2 (cons 3 '())))
→ (1 2 3)

? '(1 . (2 . 3)) → (1 2 . 3)

? '(1 . (2 . (3 . ()))) → (1 2 3)

? (+ 1 2) → 3

? (+ . (1 . (2 . ()))) → 3

? (define (sum . args)
    ...)

? (sum 1 2 3)
```

- `'(1 2)` = cons-kjeden `'(1 . (2 . ()))`
- Er `cdr` et par eller `()` sløyfer Scheme punktum og parentes: " `. (`".
- Scheme kan også lese slik "dot-notasjon".
- Husk at kode = lister.

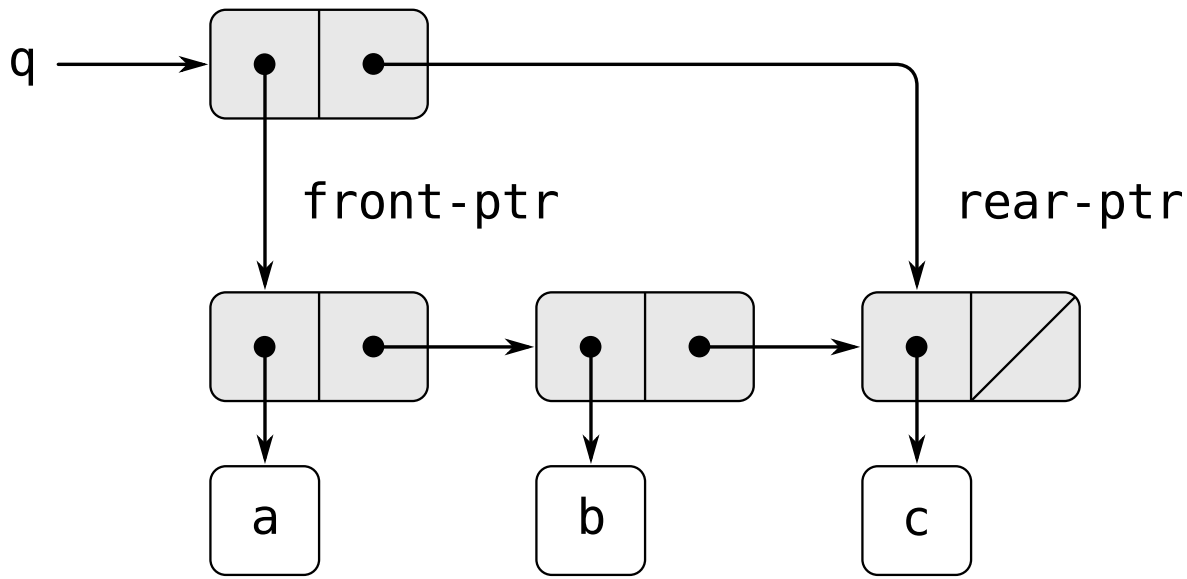
# Mer om prikk-notasjon, for de spesielt interesserte

```
? (cons 1 (cons 2 3)) → (1 2 . 3)

? (cons 1 (cons 2 (cons 3 '())))
→ (1 2 3)

? '(1 . (2 . 3)) → (1 2 . 3)

? '(1 . (2 . (3 . ()))) → (1 2 3)

? (+ 1 2) → 3

? (+ . (1 . (2 . ()))) → 3

? (define (sum . args)
    ...)

? (sum 1 2 3)

? (sum . (1 2 3))
```

- `'(1 2)` = cons-kjeden `'(1 . (2 . ()))`
- Er `cdr` et par eller `()` sløyfer Scheme punktum og parentes: " `. (`".
- Scheme kan også lese slik "dot-notasjon".
- Husk at kode = lister.
- Utnytter denne notasjonen til å skrive prosedyrer som tar et variabelt antall argumenter.

# Forrige gang: destruktive listeoperasjoner

- Lister som eksempel på muterbare data.
- Destruktiv listekonkatenasjon: `append!`
- Strukturdeling og sirkulære lister.
- Definerte køer som abstrakt (og muterbar) datatype.



# I dag og fremover

- Memoisering.
- Nok en abstrakt muterbar datatype basert på lister: tabeller.
- Mer moro med (prosedyre)navn, bindinger, og verditilordning.
- Relevant for oblig 3a.
- Strømmer og utsatt evaluering (neste to forelesninger)

# Memoisering

- Optimiseringsteknikk for å gjøre prosedyrer raskere.
- Enkelt eksempel på dynamisk programmering.

# Memoisering

- Optimiseringsteknikk for å gjøre prosedyrer raskere.
- Enkelt eksempel på dynamisk programmering.
- Unngår å utføre gjentatte beregninger:
- Lar prosedyrer "huske" sine tidligere returverdier for samme input-argumenter.
- Dersom en memoisert prosedyre kalles med samme argument flere ganger vil beregningen kun bli utført én gang, ved første kall, mens senere kall bare gjenbruker returverdien som er lagret fra tidligere.
- Kan gi store besparelser ved ressurskrevende beregninger.

# Memoisering (forts.)

- Oblig 3a: skal implementere støtte for å lage memoiserte versjoner av vilkårlige prosedyrer.
- Lagrer resultater i en lokal tabell (cache), indeksert på argumentene:
- For gjentatte kall med samme argumenter holder det å slå opp i tabellen.
- Skal i dag se på prekode som implementerer tabeller.
- Nok et eksempel på en listebasert muterbar datatype.

# Repetisjon: Fibonaccitall

- Eksponensjell vekst f.eks. antall forfarer til honningbier: den såkalte Fibonacci-rekken:
- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 \ldots$
- Bortsett fra de to første er hvert tall summen av de to foregående.

$$\text{fib}(n) = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{ellers} \end{cases}$$

# Repetisjon: Fibonaccitall

- Eksponensjell vekst f.eks. antall forfarer til honningbier: den såkalte Fibonacci-rekken:
- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 \ldots$
- Bortsett fra de to første er hvert tall summen av de to foregående.

$$\text{fib}(n) = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{ellers} \end{cases}$$

```
? (define (fib n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (+ (fib (- n 1))
                   (fib (- n 2))))))
```

# Repetisjon: Fibonaccitall

- Eksponensjell vekst f.eks. antall forfarer til honningbier: den såkalte Fibonacci-rekken:
- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 \ldots$
- Bortsett fra de to første er hvert tall summen av de to foregående.

$$\text{fib}(n) = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{ellers} \end{cases}$$

```
? (define (fib n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (+ (fib (- n 1))
                   (fib (- n 2))))))

? (fib 5) → 5

? (fib 6) → 8

? (fib 7) → 13

? (fib 8) → 21

? (fib 9) → 34
```

# Repetisjon: Fibonaccitall

- Eksponensjell vekst f.eks. antall forfarer til honningbier: den såkalte Fibonacci-rekken:
- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 \ldots$
- Bortsett fra de to første er hvert tall summen av de to foregående.
- Før vi når basistilfellene fører hvert kall på `fib` til *to* nye rekursive kall.
- Gir opphav til en såkalt trerekursiv prosess.

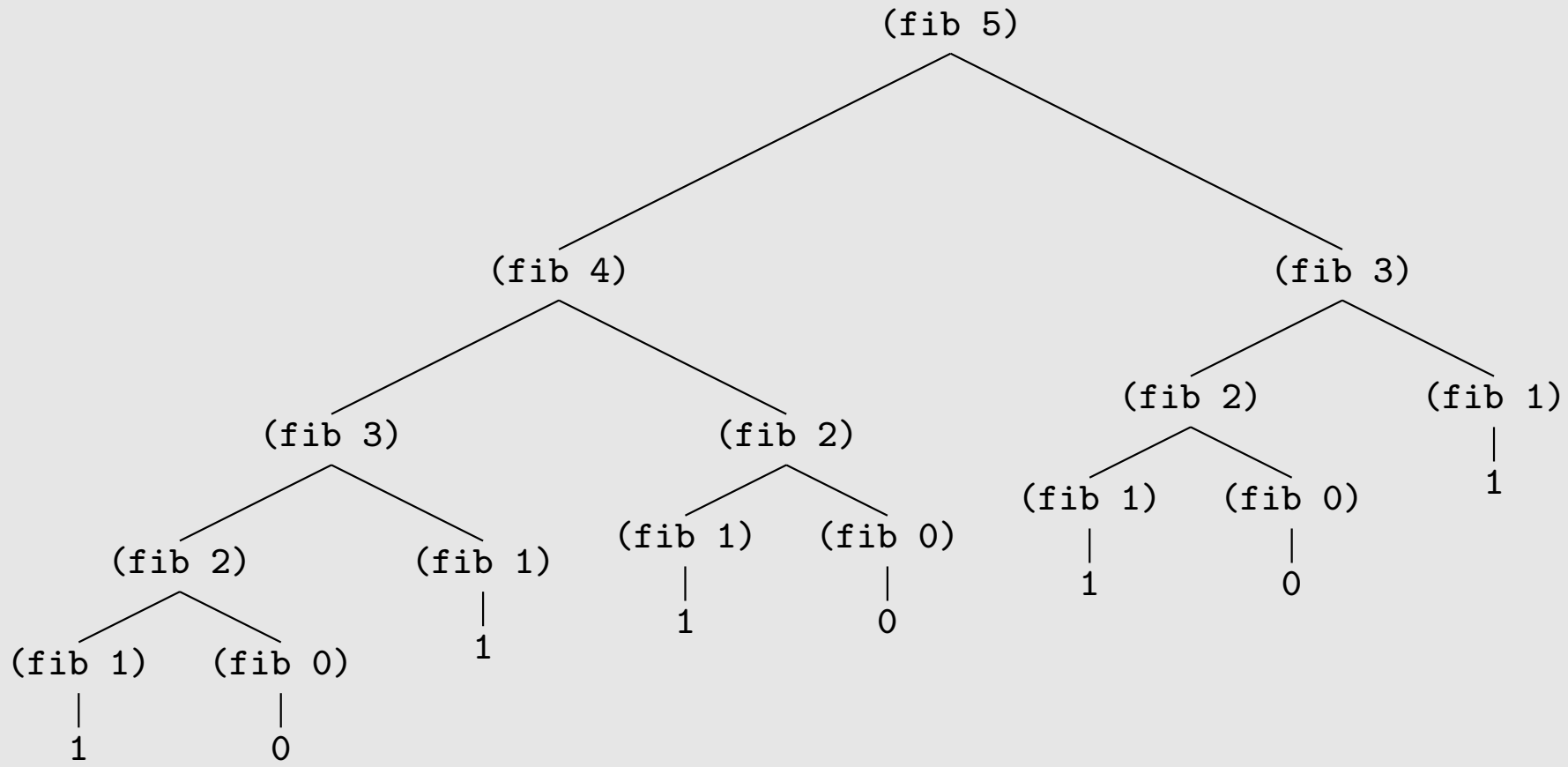
$$\text{fib}(n) = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{ellers} \end{cases}$$

```
? (define (fib n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (+ (fib (- n 1))
                   (fib (- n 2))))))
```

```
? (fib 5) → 5

? (fib 6) → 8

? (fib 7) → 13

? (fib 8) → 21

? (fib 9) → 34
```

# Repetisjon: `fib` som trerekursiv prosess

- Veldig mange dupliserte operasjoner i utregningen av fib:
- Eksponensiell vekst i tidsbruk; proporsjonal med antall noder.



# Hvor vi vil hen: kalleksempler med memoisering

## Testprosedyre (fibonacci)

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

# Hvor vi vil hen: kalleksempler med memoisering

## Testprosedyre (fibonacci)

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

```
? (fib 3)
⇝ computing fib of 3
  computing fib of 2
  computing fib of 1
  computing fib of 0
  computing fib of 1
→ 2
```

# Hvor vi vil hen: kalleksempler med memoisering

## Testprosedyre (fibonacci)

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

```
? (fib 3)
↝ computing fib of 3
  computing fib of 2
  computing fib of 1
  computing fib of 0
  computing fib of 1
→ 2
```

```
? (set! fib (mem 'memoize fib))
```

# Hvor vi vil hen: kalleksempler med memoisering

## Testprosedyre (fibonacci)

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

```
? (fib 3)
⇝ computing fib of 3
  computing fib of 2
  computing fib of 1
  computing fib of 0
  computing fib of 1
→ 2
```

```
? (set! fib (mem 'memoize fib))

? (fib 3)
⇝ computing fib of 3
  computing fib of 2
  computing fib of 1
  computing fib of 0
→ 2
```

# Hvor vi vil hen: kalleksempler med memoisering

## Testprosedyre (fibonacci)

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

```
? (fib 3)
⇝ computing fib of 3
  computing fib of 2
  computing fib of 1
  computing fib of 0
  computing fib of 1
→ 2
```

```
? (set! fib (mem 'memoize fib))

? (fib 3)
⇝ computing fib of 3
  computing fib of 2
  computing fib of 1
  computing fib of 0
→ 2

? (fib 3) → 2
```

# Hvor vi vil hen: kalleksempler med memoisering

## Testprosedyre (fibonacci)

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

```
? (fib 3)
⇝ computing fib of 3
  computing fib of 2
  computing fib of 1
  computing fib of 0
  computing fib of 1
→ 2
```

```
? (set! fib (mem 'memoize fib))

? (fib 3)
⇝ computing fib of 3
  computing fib of 2
  computing fib of 1
  computing fib of 0
→ 2

? (fib 3) → 2

? (set! fib (mem 'unmemoize fib))
```

# Hvor vi vil hen: kalleksempler med memoisering

## Testprosedyre (fibonacci)

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

```
? (fib 3)
↝ computing fib of 3
  computing fib of 2
  computing fib of 1
  computing fib of 0
  computing fib of 1
→ 2
```

```
? (set! fib (mem 'memoize fib))

? (fib 3)
↝ computing fib of 3
  computing fib of 2
  computing fib of 1
  computing fib of 0
→ 2

? (fib 3) → 2

? (set! fib (mem 'unmemoize fib))

? (fib 3)
↝ compute fib of 3
  compute fib of 2
  compute fib of 1
  compute fib of 0
  compute fib of 1
→ 2
```

# Assosiative tabeller

- En datastruktur for å assosiere *nøkler* med *verdier*.
- Kan implementeres med `cons`-celler i bunnen.
- Kan bygges på en datatype som heter *assosiasjonsliste*, eller bare alist:
- En liste av par: hver `car` er en nøkkel, "assosiert" med `cdr`-verdien.

```
? (define table '((a . 1) (b . 2) (c . 3)))
```

# Assosiative tabeller

- En datastruktur for å assosiere *nøkler* med *verdier*.
- Kan implementeres med `cons`-celler i bunnen.
- Kan bygges på en datatype som heter *assosiasjonsliste*, eller bare alist:
- En liste av par: hver `car` er en nøkkel, "assosiert" med `cdr`-verdien.

```
? (define table '((a . 1) (b . 2) (c . 3)))
? (assoc 'b table) → (b . 2)
```

# Assosiative tabeller

- En datastruktur for å assosiere *nøkler* med *verdier*.
- Kan implementeres med `cons`-celler i bunnen.
- Kan bygges på en datatype som heter *assosiasjonsliste*, eller bare alist:
- En liste av par: hver `car` er en nøkkel, "assosiert" med `cdr`-verdien.

```
? (define table '((a . 1) (b . 2) (c . 3)))
? (assoc 'b table) → (b . 2)
? (assoc 'e table) → #f
```

# Assosiative tabeller

- En datastruktur for å assosiere *nøkler* med *verdier*.
- Kan implementeres med `cons`-celler i bunnen.
- Kan bygges på en datatype som heter *assosiasjonsliste*, eller bare alist:
- En liste av par: hver `car` er en nøkkel, "assosiert" med `cdr`-verdien.

```
? (define table '((a . 1) (b . 2) (c . 3)))

? (assoc 'b table) → (b . 2)

? (assoc 'e table) → #f

(define (assoc key records)          ;; innebygd i scheme
  ...)
```

# Assosiative tabeller

- En datastruktur for å assosiere *nøkler* med *verdier*.
- Kan implementeres med `cons`-celler i bunnen.
- Kan bygges på en datatype som heter *assosiasjonsliste*, eller bare alist:
- En liste av par: hver `car` er en nøkkel, "assosiert" med `cdr`-verdien.

```
? (define table '((a . 1) (b . 2) (c . 3)))

? (assoc 'b table) → (b . 2)

? (assoc 'e table) → #f

(define (assoc key records)              ;; innebygd i scheme
  (cond ((null? records) #f)
```

# Assosiative tabeller

- En datastruktur for å assosiere *nøkler* med *verdier*.
- Kan implementeres med `cons`-celler i bunnen.
- Kan bygges på en datatype som heter *assosiasjonsliste*, eller bare alist:
- En liste av par: hver `car` er en nøkkel, "assosiert" med `cdr`-verdien.

```
? (define table '((a . 1) (b . 2) (c . 3)))

? (assoc 'b table) → (b . 2)

? (assoc 'e table) → #f

(define (assoc key records)           ;; innebygd i scheme
  (cond ((null? records) #f)
        ((equal? key (caar records)) (car records))
```

# Assosiative tabeller

- En datastruktur for å assosiere *nøkler* med *verdier*.
- Kan implementeres med `cons`-celler i bunnen.
- Kan bygges på en datatype som heter *assosiasjonsliste*, eller bare alist:
- En liste av par: hver `car` er en nøkkel, "assosiert" med `cdr`-verdien.

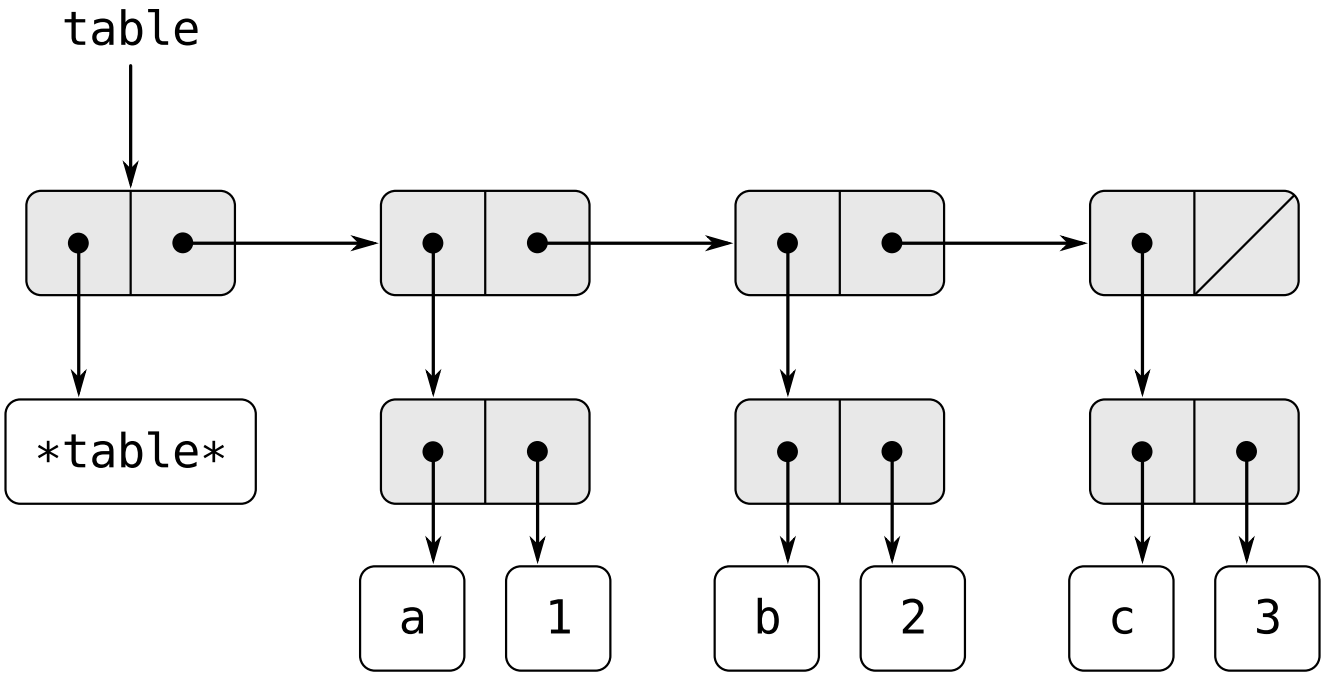
```
? (define table '((a . 1) (b . 2) (c . 3)))

? (assoc 'b table) → (b . 2)

? (assoc 'e table) → #f

(define (assoc key records)           ;; innebygd i scheme
  (cond ((null? records) #f)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))
```

# Assosiative tabeller (forts.)

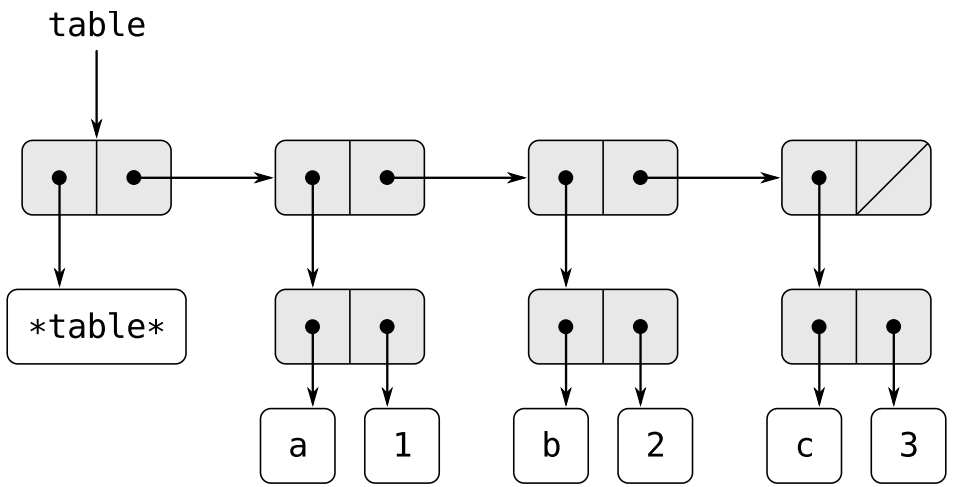
- Noen operasjoner kan forenkles ved å sette én ekstra `cons`-celle foran:
- Gir oss et fast sted å modifisere ved innsetting av nye oppslag.
- Mulighet for destruktive operasjoner som også støtter tomme lister.

# Assosiative tabeller (forts.)

- Noen operasjoner kan forenkles ved å sette én ekstra `cons`-celle foran:
- Gir oss et fast sted å modifisere ved innsetting av nye oppslag.
- Mulighet for destruktive operasjoner som også støtter tomme lister.
- En slik *headed list* kan også bruke første `car` som typemerkelapp.

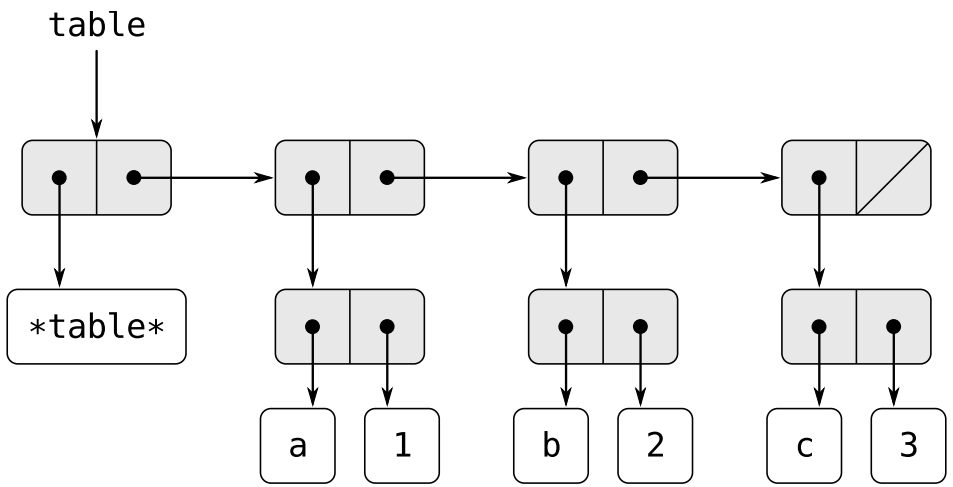


# Abstraksjonsbarriere



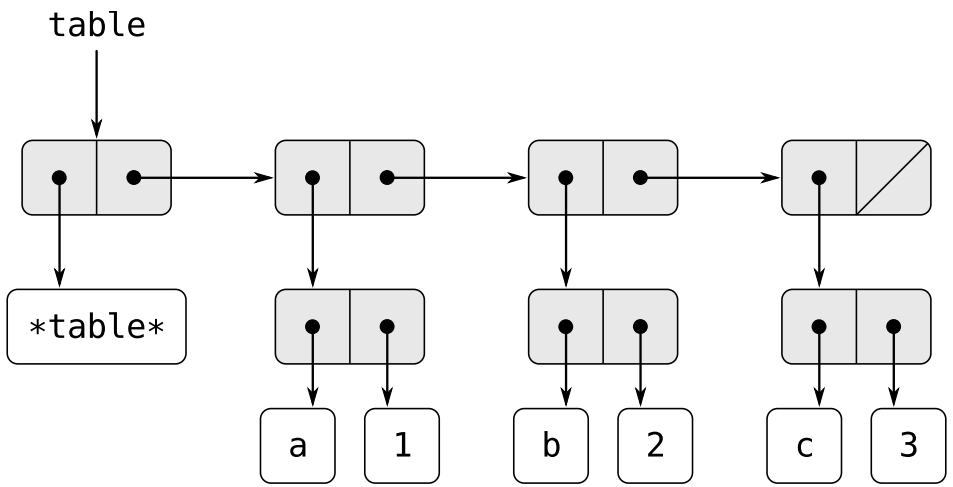

```
(define (make-table) (list '*table*))
```

# Abstraksjonsbarriere



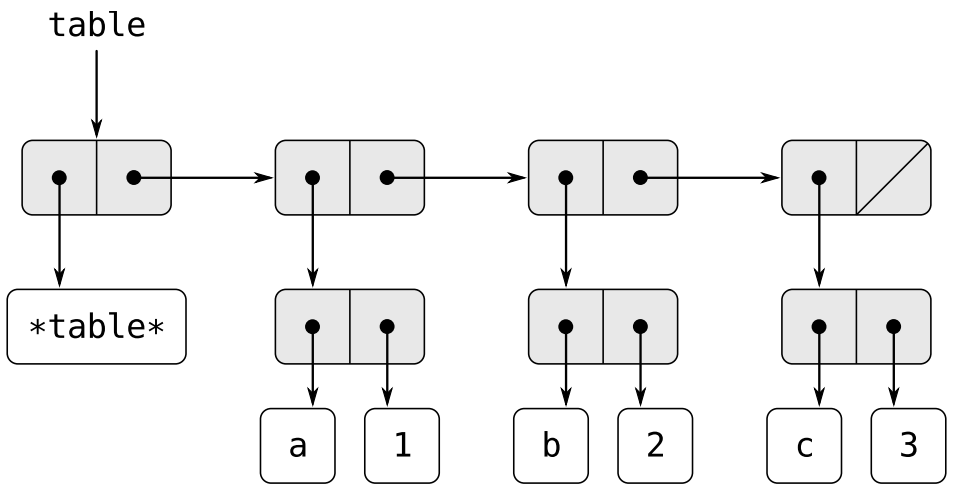

```
(define (make-table) (list '*table*))

(define (insert! key value table)
  ...)
```

# Abstraksjonsbarriere



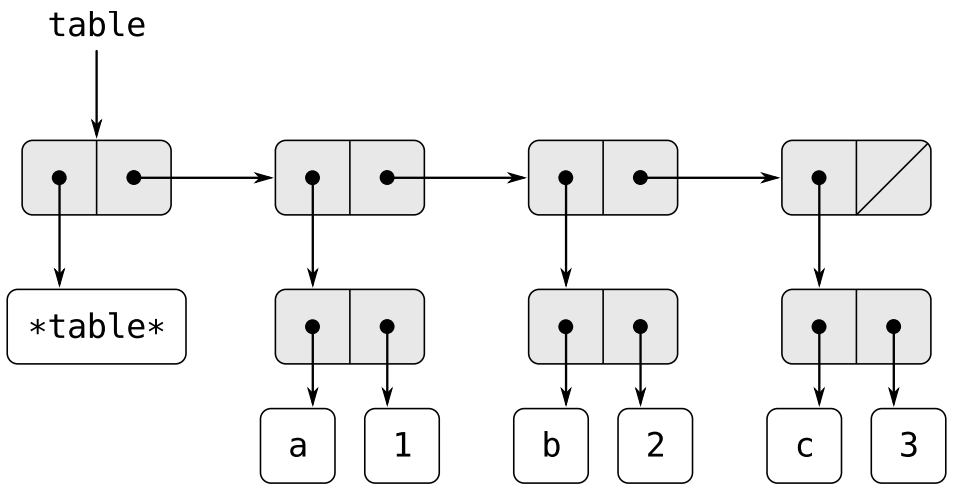

```
(define (make-table) (list '*table*))

(define (insert! key value table)
   (let ((record (assoc key (cdr table))))
    (if record
      (set-cdr! record value)
      (set-cdr! table (cons (cons key value) (cdr table))))))
```

# Abstraksjonsbarriere



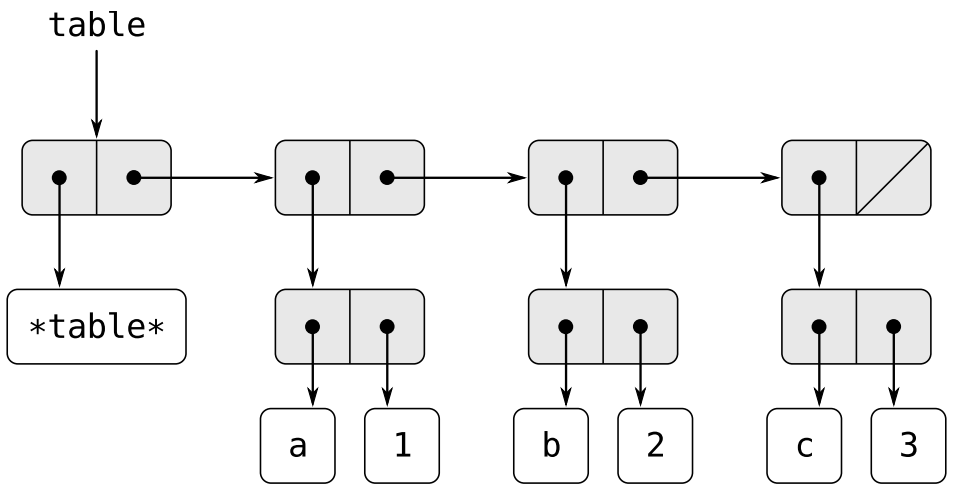

```
(define (make-table) (list '*table*))

(define (insert! key value table)
   (let ((record (assoc key (cdr table))))
    (if record
      (set-cdr! record value)
      (set-cdr! table (cons (cons key value) (cdr table))))))
```

# Abstraksjonsbarriere



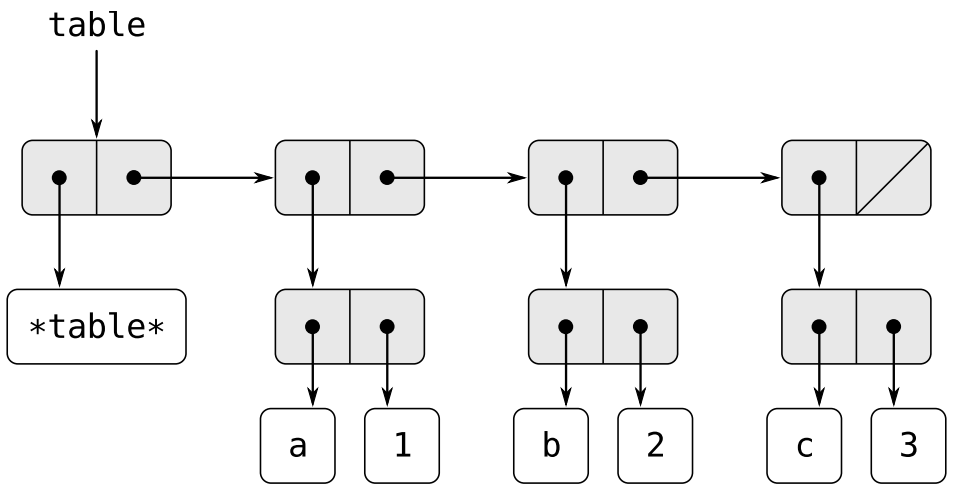

```
(define (make-table) (list '*table*))

(define (insert! key value table)
   (let ((record (assoc key (cdr table))))
    (if record
      (set-cdr! record value)
      (set-cdr! table (cons (cons key value) (cdr table))))))
```

# Abstraksjonsbarriere



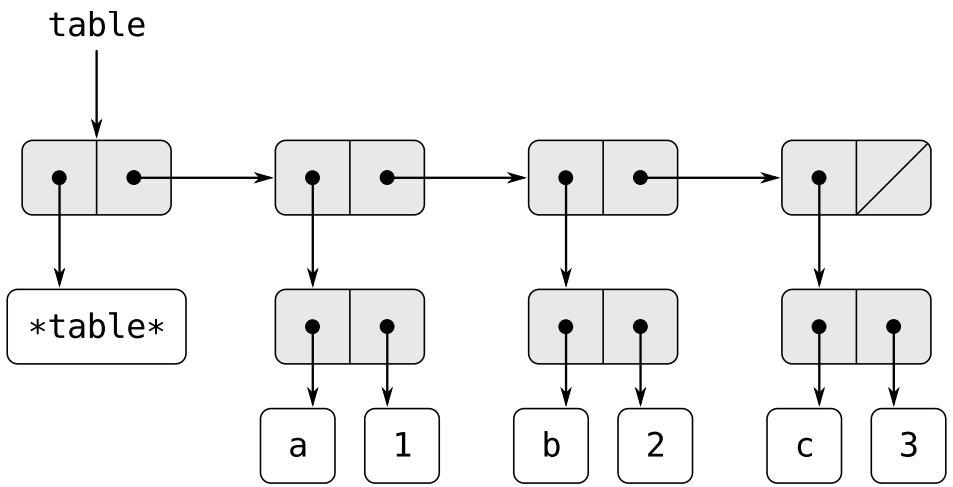

```
(define (make-table) (list '*table*))

(define (insert! key value table)
   (let ((record (assoc key (cdr table))))
    (if record
      (set-cdr! record value)
      (set-cdr! table (cons (cons key value) (cdr table))))))
```

# Abstraksjonsbarriere



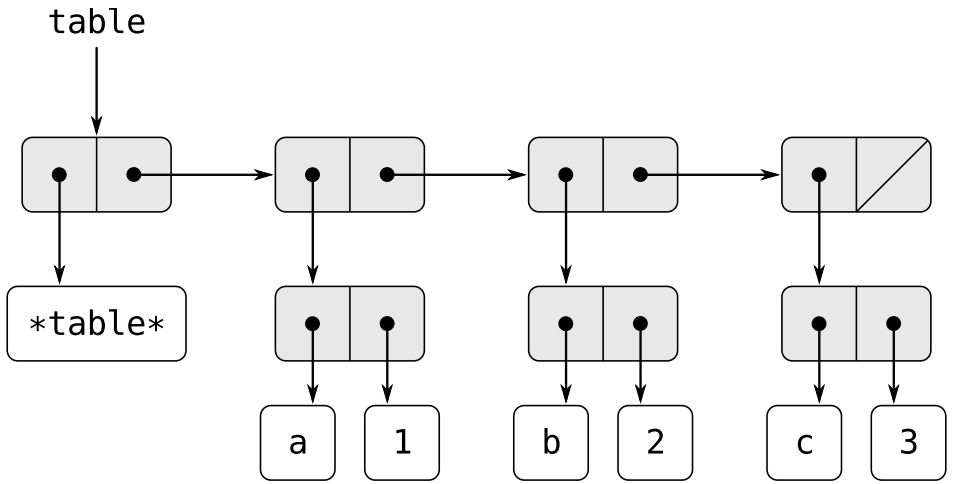

```
(define (make-table) (list '*table*))

(define (insert! key value table)
   (let ((record (assoc key (cdr table))))
    (if record
      (set-cdr! record value)
      (set-cdr! table (cons (cons key value) (cdr table))))))
```

# Abstraksjonsbarriere



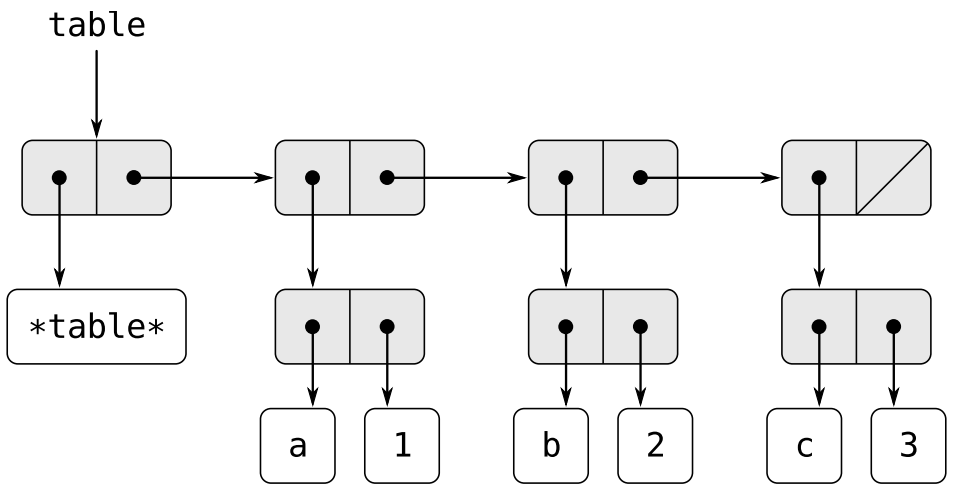

```
(define (make-table) (list '*table*))

(define (insert! key value table)
   (let ((record (assoc key (cdr table))))
    (if record
      (set-cdr! record value)
      (set-cdr! table (cons (cons key value) (cdr table))))))
```

# Abstraksjonsbarriere




```
(define (make-table) (list '*table*))

(define (insert! key value table)
   (let ((record (assoc key (cdr table))))
    (if record
      (set-cdr! record value)
      (set-cdr! table (cons (cons key value) (cdr table))))))
```

# Abstraksjonsbarriere




```
(define (make-table) (list '*table*))

(define (insert! key value table)
   (let ((record (assoc key (cdr table))))
    (if record
      (set-cdr! record value)
      (set-cdr! table (cons (cons key value) (cdr table))))))

(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (and record (cdr record))))
```

# Eksempel på bruk

- Tabell som lagrer dokumentasjonsstrenger for prosedyrer.
- Indekserer på prosedyrene selv ('førsteklasses objekter')!

```
? (define doc (make-table))
```

# Eksempel på bruk

- Tabell som lagrer dokumentasjonsstrenger for prosedyrer.
- Indekserer på prosedyrene selv ('førsteklasses objekter')!

```
? (define doc (make-table))
? doc →
```

# Eksempel på bruk

- Tabell som lagrer dokumentasjonsstrenger for prosedyrer.
- Indekserer på prosedyrene selv ('førsteklasses objekter')!

```
? (define doc (make-table))
? doc →  (*table*)
```

# Eksempel på bruk

- Tabell som lagrer dokumentasjonsstrenger for prosedyrer.
- Indekserer på prosedyrene selv ('førsteklasses objekter')!

```
? (define doc (make-table))
? doc →  (*table*)
? (insert! + "sums numbers" doc)
```

# Eksempel på bruk

- Tabell som lagrer dokumentasjonsstrenger for prosedyrer.
- Indekserer på prosedyrene selv ('førsteklasses objekter')!

```
? (define doc (make-table))
? doc →  (*table*)
? (insert! + "sums numbers" doc)
? (insert! cons "constructs pairs" doc)
```

# Eksempel på bruk

- Tabell som lagrer dokumentasjonsstrenger for prosedyrer.
- Indekserer på prosedyrene selv ('førsteklasses objekter')!

```
? (define doc (make-table))
? doc →  (*table*)
? (insert! + "sums numbers" doc)
? (insert! cons "constructs pairs" doc)
? (insert! append "concatenates lists" doc)
```

# Eksempel på bruk

- Tabell som lagrer dokumentasjonsstrenger for prosedyrer.
- Indekserer på prosedyrene selv ('førsteklasses objekter')!

```
? (define doc (make-table))

? doc →  (*table*)

? (insert! + "sums numbers" doc)

? (insert! cons "constructs pairs" doc)

? (insert! append "concatenates lists" doc)

? doc →
(*table*
 (#<p:append> . "concatenates lists")
 (#<p:cons> . "constructs pairs")
 (#<p:+> . "sums numbers"))
```

# Eksempel på bruk

- Tabell som lagrer dokumentasjonsstrenger for prosedyrer.
- Indekserer på prosedyrene selv ('førsteklasses objekter')!

```
? (define doc (make-table))

? doc →  (*table*)

? (insert! + "sums numbers" doc)

? (insert! cons "constructs pairs" doc)

? (insert! append "concatenates lists" doc)

? doc →
(*table*
 (#<p:append> . "concatenates lists")
 (#<p:cons> . "constructs pairs")
 (#<p:+> . "sums numbers"))

? (lookup append doc) → "concatenates lists"
```

# Eksempel på bruk

- Tabell som lagrer dokumentasjonsstrenger for prosedyrer.
- Indekserer på prosedyrene selv ('førsteklasses objekter')!

```
? (define doc (make-table))
? doc →  (*table*)
? (insert! + "sums numbers" doc)
? (insert! cons "constructs pairs" doc)
? (insert! append "concatenates lists" doc)
? doc →
(*table*
 (#<p:append> . "concatenates lists")
 (#<p:cons> . "constructs pairs")
 (#<p:+> . "sums numbers"))
? (lookup append doc) → "concatenates lists"
? (lookup map doc) → #f
```

# Eksempel på bruk

- Tabell som lagrer dokumentasjonsstrenger for prosedyrer.
- Indekserer på prosedyrene selv ('førsteklasses objekter')!

```
? (define doc (make-table))
? doc →  (*table*)
? (insert! + "sums numbers" doc)
? (insert! cons "constructs pairs" doc)
? (insert! append "concatenates lists" doc)
? doc →
(*table*
 (#<p:append> . "concatenates lists")
 (#<p:cons> . "constructs pairs")
 (#<p:+> . "sums numbers"))
? (lookup append doc) → "concatenates lists"
? (lookup map doc) → #f
```

- Kunne pakket dette inn i et grensesnitt med `(describe append)` osv.

# Noen refleksjoner rundt tabeller

- I denne (naïve) implementasjonen, hvilken tidskompleksitet har det
  - å slå opp verdien til en nøkkel?
  - å legge inn et $\langle n\text{økkel}, v\text{erdi}\rangle$-par?

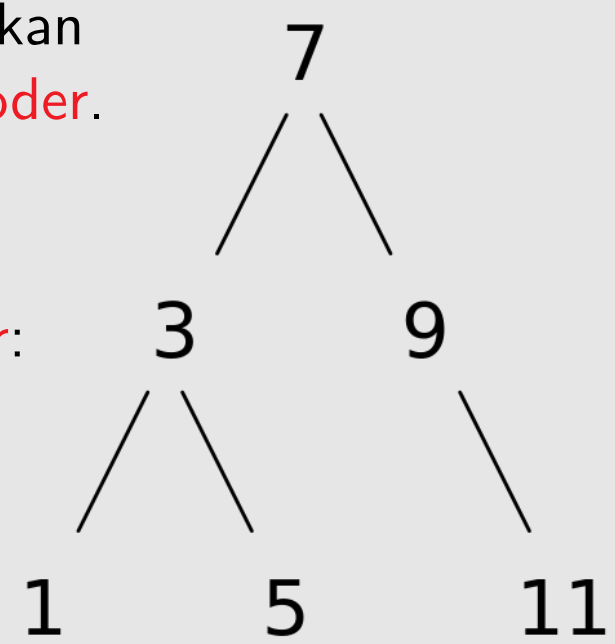
# Noen refleksjoner rundt tabeller

- I denne (naïve) implementasjonen, hvilken tidskompleksitet har det
  - å slå opp verdien til en nøkkel?
  - å legge inn et $\langle n\text{økkel}, v\text{erdi}\rangle$-par?
- Hvordan kan kompleksiteten forbedres?

# Noen refleksjoner rundt tabeller

- I denne (naïve) implementasjonen, hvilken tidskompleksitet har det
  - å slå opp verdien til en nøkkel?
  - å legge inn et $\langle$*n*økkel, *v*erdi$\rangle$-par?
- Hvordan kan kompleksiteten forbedres? Sorterte nøkler.

- Har tidligere sett hvordan mengder effektivt kan representeres som binærtrær med ordnede noder.
- Kan halvere søkerommet i hvert trinn.
- Kan enkelt utvides til å impementere tabeller:
- Elementene i mengden tilsvarer *nøkler*,
- men i tillegg lagrer hver node også en indeksert *verdi*.



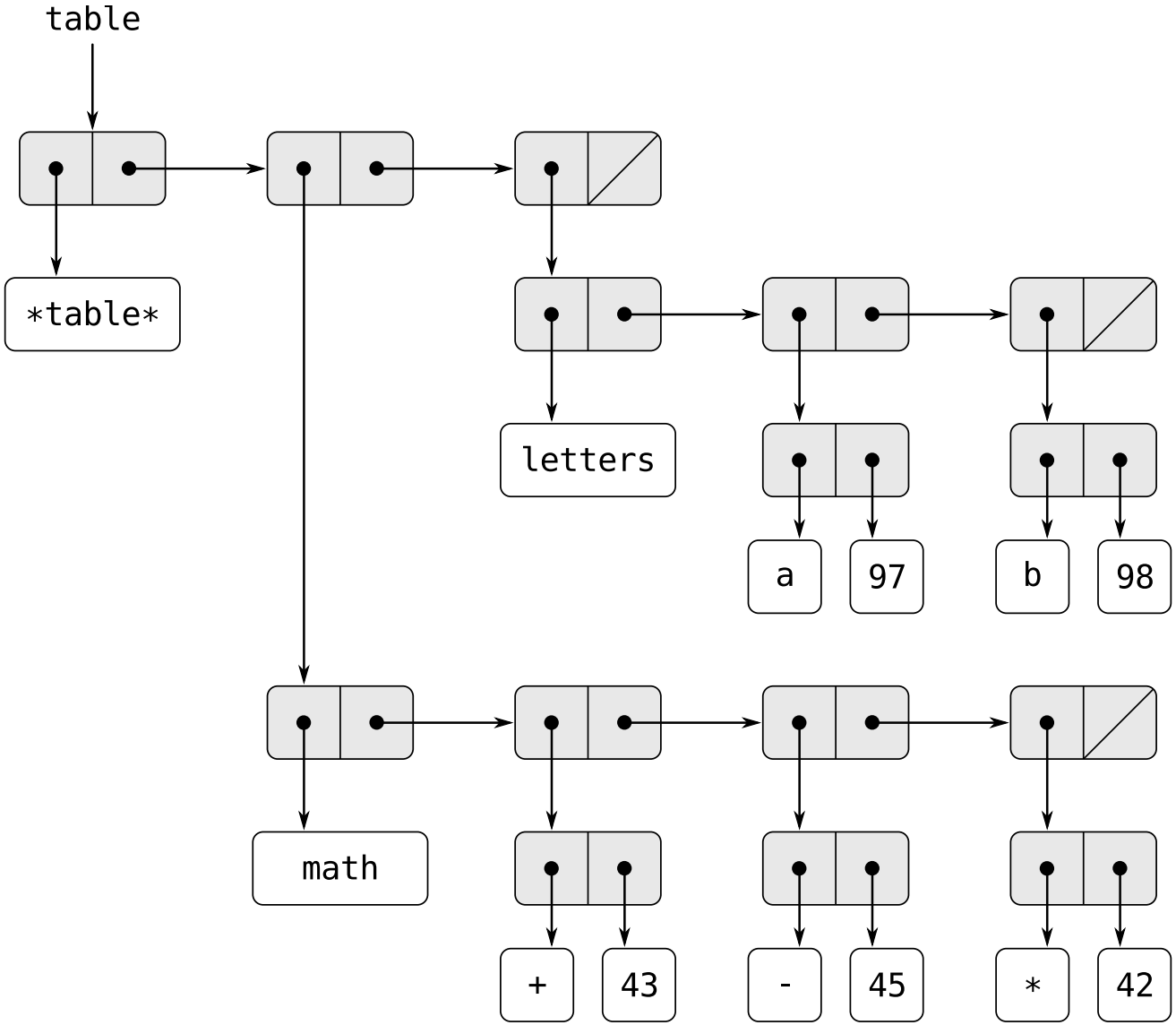
# Noen refleksjoner rundt tabeller (forts.)

- Med mange elementer kan det lønne seg å bruke en hashtabell isteden.
  - Common Lisp har hashtabeller som innebygd datatype, men ikke R5RS.
  - Men Scheme har støtte for fast dimensjonerte felt: *vector* ('array').
- Hva er restriksjonene på nøkler? Hva avgjør?

# Noen refleksjoner rundt tabeller (forts.)

- Med mange elementer kan det lønne seg å bruke en hashtabell isteden.
  - Common Lisp har hashtabeller som innebygd datatype, men ikke R5RS.
  - Men Scheme har støtte for fast dimensjonerte felt: *vector* ('array').
- Hva er restriksjonene på nøkler? Hva avgjør?
- Kan være hva som helst, f.eks prosedyrer, strenger, eller lister: `assoc` bruker `equal?`.
- Heller ingen restriksjoner på verdiene:
- Kan f.eks være nye tabeller eller alister: multidimensjonale tabeller.

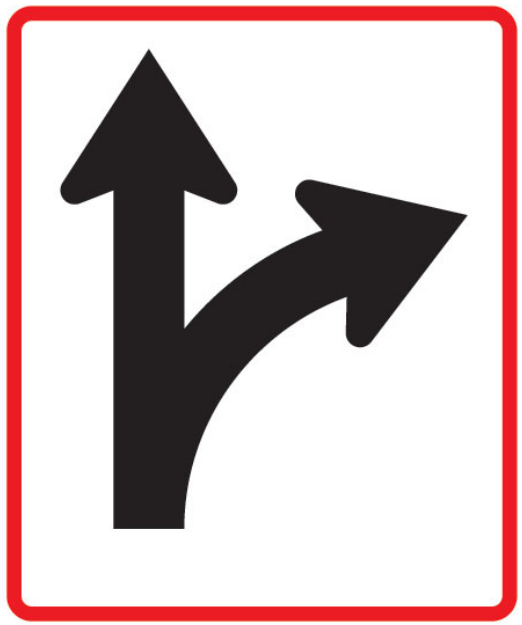
# En todimensjonal tabell




- Undertabellene er forsatt *headed lists* men nøkkelen selv er "hodet", trenger ikke dummy-verdien `'*table*`.

# Nok et sidespor...

- Forrige gang så vi en del eksempler som viste samspillet mellom verditilordning og navn, bindinger, og 'aliasing'.
- Tar oss tid til noen flere eksempler i dag i forbindelse med prosedyrenavn.
- Prosedyrenavn er bare vanlige navngitte variabler; kan dermed også tenkes på som tilstandsvariabler som kan manipuleres.
- Kan vise seg nyttig for å forstå memoiserings-oppgaven i oblig 3a.



# Bindinger, navn og prosedyrer (1:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)   ;; aliasing
```

```
? (foo 1 2) ⇝
```

# Bindinger, navn og prosedyrer (1:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)   ;; aliasing
```

```
? (foo 1 2) ⇝
```

# Bindinger, navn og prosedyrer (1:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)   ;; aliasing
```

```
? (foo 1 2) ⇝
```

# Bindinger, navn og prosedyrer (1:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)   ;; aliasing
```

```
? (foo 1 2) ⇝
```

# Bindinger, navn og prosedyrer (1:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)   ;; aliasing
```

```
? (foo 1 2) ⇝
```

# Bindinger, navn og prosedyrer (1:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)   ;; aliasing
```

```
? (foo 1 2) ⇝
```

# Bindinger, navn og prosedyrer (1:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)   ;; aliasing
```

```
? (foo 1 2) ⇝
args = (1 2)
args = (2)
args = ()
```

# Bindinger, navn og prosedyrer (1:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)   ;; aliasing
```

```
? (foo 1 2) ⇝
args = (1 2)
args = (2)
args = ()

? (bar 1 2) ⇝
```

# Bindinger, navn og prosedyrer (1:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)   ;; aliasing
```

```
? (foo 1 2) ⇝
args = (1 2)
args = (2)
args = ()

? (bar 1 2) ⇝
args = (1 2)
args = (2)
args = ()
```

# Bindinger, navn og prosedyrer (2:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)
```

```
? (set! foo (lambda args (display "new foo called!")))

? (foo 1 2) ⇝
```

# Bindinger, navn og prosedyrer (2:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)
```

```
? (set! foo (lambda args (display "new foo called!")))

? (foo 1 2) ⇝
new foo called!
```

# Bindinger, navn og prosedyrer (2:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)
```

```
? (set! foo (lambda args (display "new foo called!")))

? (foo 1 2) ⇝
new foo called!

? (bar 1 2) ⇝
```

# Bindinger, navn og prosedyrer (2:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)
```

```
? (set! foo (lambda args (display "new foo called!")))

? (foo 1 2) ⇝
new foo called!

? (bar 1 2) ⇝
args = (1 2)
new foo called!
```

# Bindinger, navn og prosedyrer (2:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)
```

```
? (set! foo (lambda args (display "new foo called!")))

? (foo 1 2) ⇝
new foo called!

? (bar 1 2) ⇝
args = (1 2)
new foo called!
```

# Bindinger, navn og prosedyrer (3:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)
```

```
? (set! foo (let ((old-foo foo))              ;; let-over-lambda
              (lambda args (display "new foo called! ")
                (apply old-foo args))))

? (foo 1 2) ⇝
```

# Bindinger, navn og prosedyrer (3:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)
```

```
? (set! foo (let ((old-foo foo))              ;; let-over-lambda
              (lambda args (display "new foo called! ")
                (apply old-foo args))))

? (foo 1 2) ⇝
new foo called! args = (1 2)
new foo called! args = (2)
new foo called! args = ()
```

# Bindinger, navn og prosedyrer (3:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)
```

```
? (set! foo (let ((old-foo foo))              ;; let-over-lambda
              (lambda args (display "new foo called! ")
                (apply old-foo args))))

? (foo 1 2) ⇝
new foo called! args = (1 2)
new foo called! args = (2)
new foo called! args = ()

? (bar 1 2) ⇝
```

# Bindinger, navn og prosedyrer (3:3)

```
? (define foo
    (lambda args
      (display "args = ") (display args) (newline)
      (if (not (null? args))
          (apply foo (cdr args)))))

? (define bar foo)
```

```
? (set! foo (let ((old-foo foo))              ;; let-over-lambda
              (lambda args (display "new foo called! ")
                (apply old-foo args))))

? (foo 1 2) ⇝
new foo called! args = (1 2)
new foo called! args = (2)
new foo called! args = ()

? (bar 1 2) ⇝
args = (1 2)
new foo called! args = (2)
new foo called! args = ()
```

# Tilbake til memoisering

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

# Tilbake til memoisering

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))


? (set! fib (mem 'memoize fib))
```

# Tilbake til memoisering

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))


? (set! fib (mem 'memoize fib))

? (fib 3)
⇝ computing fib of 3
   computing fib of 2
   computing fib of 1
   computing fib of 0
→ 2

? (fib 3) → 2
```

# Tilbake til memoisering

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))


? (set! fib (mem 'memoize fib))

? (fib 3)
⇝ computing fib of 3
   computing fib of 2
   computing fib of 1
   computing fib of 0
→ 2

? (fib 3) → 2

? (set! fib (mem 'unmemoize fib))
```

# Tilbake til memoisering

```
(define (fib n)
  (display "computing fib of ")
  (display n) (newline)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))


? (set! fib (mem 'memoize fib))

? (fib 3)
⇝ computing fib of 3
   computing fib of 2
   computing fib of 1
   computing fib of 0
→ 2

? (fib 3) → 2

? (set! fib (mem 'unmemoize fib))
```

- Har i dag sett på mye som er nyttig for å implementere mem:
- Hva må vi holde rede på? Må kunne koble;
- argumenter ⇒ returverdier,
- memoisert versjon av prosedyre ⇒ opprinnelig versjon.
- Tabeller nyttig for begge deler.
- Har sett at både lister og prosedyrer kan være nøkler.
- Idiomet let over lambda; nyttig for innkapsling av variabler i et returnert lambda-uttrykk.

# Veien videre

- Strømmer
- En ny funksjonell tilnærming til tid og tilstand
- Realisering av data ved behov
- Uendelige datastrukturer
- Utsatt evaluering
- Skal begynne med litt repetisjon om sekvensoperasjoner.