Hoofdstuk 0,2

De abstracties van “Les 6” als ADT beschreven

1ste Implementatie van "Breuken" (les 6)

In de eerste implementatie waren breuken gewoon **cons cellen**

```
(define (make-rat a b)
  (cons a b))

(define (numer c)
  (car c))

(define (denom c)
  (cdr c))
```

constructor

selectors of accessors

```
> (make-rat 1 2)
(1 . 2)
> (define p (make-rat 2 3))
> p
(2 . 3)
> (numer p)
2
> (denom p)
3
```

```
(define (rat+ p q)
  (make-rat
    (+ (* (numer p) (denom q))
       (* (numer q) (denom p)))
    (* (denom p) (denom q))))
```

```
(define (rat- p q)
  (make-rat
    (- (* (numer p) (denom q))
       (* (numer q) (denom p)))
    (* (denom p) (denom q))))
```

```
(define (rat* p q)
  (make-rat
    (* (numer p) (numer q))
    (* (denom p) (denom q))))
```

```
(define (rat/ p q)
  (make-rat
    (* (numer p) (numer q))
    (* (denom p) (denom q))))
```

```
> (define p (make-rat 2 3))
> (define q (make-rat 1 4))
> (rat+ p q)
(11 . 12)
> (rat* p q)
(2 . 12)
```

breuk is niet vereenvoudigd

2de Implementatie van "Breuken"

In de tweede implementatie waren breuken gewoon **cons cellen** met vereenvoudigde car en cdr.

```
(define (make-rat2 a b)
  (let ((g (gcd a b)))
    (cons (/ a g) (/ b g))))
```

breuken altijd in hun 'vereenvoudigde' vorm opslaan door teller en noemer te delen door hun grootste gemene deler

```
> (make-rat 3 12)
(1 . 4)
> (define p (make-rat 2 3))
> (define q (make-rat 3 12))
> (rat+ p q)
(11 . 12)
> (rat* p q)
(1 . 6)
```

operatoren blijven werken ook met deze nieuwe representatie

maar resultaten zijn bij constructie 'vereenvoudigd'

3

ADT rat

```
ADT rat

make-rat
   ( number number → rat )
rat+
   ( rat rat → rat )
rat-
   ( rat rat → rat )
rat*
   ( rat rat → rat )
rat/
   ( rat rat → rat )
```

Het ADT heeft een naam en de procedure-types van alle operaties die tot het ADT behoren.

Over de representatie van breuken (als cons cellen) en over de code van de implementatie wordt **bewust** niks gezegd.

"Verzamelingen" als ongeordende lijsten

In de eerste implementatie waren verzamelingen gewoon **ongeordende lijsten**

```
(define (create-set) '())

(define (empty? set) (null? set))

(define (element-of? el set)
  (cond
    ((empty? set) #f)
    ((equal? el (car set)) #t)
    (else (element-of? el (cdr set)))))

(define (insert el set)
  (if (element-of? el set)
      set
      (cons el set)))

(define (delete el set)
  (cond
    ((empty? set) set)
    ((equal? el (car set)) (cdr set))
    (else (cons (car set) (delete el (cdr set))))))
```

als een element al in de verzameling zit mag het niet een tweede keer toegevoegd worden

insert zal er voor zorgen dan elementen die in beide lijsten voorkomen niet dubbel in het resultaat terecht komen

hier mag cons gebruikt worden ipv insert omdat er geen dubbels in de input zitten en dus ook niet in de doorsnede

```
(define (union set1 set2)
  (if (empty? set1)
      set2
      (insert (car set1)
              (union (cdr set1) set2))))

(define (intersection set1 set2)
  (cond
    ((or (empty? set1) (empty? set2))
     (create-set))
    ((element-of? (car set1) set2)
     (cons (car set1)
           (intersection (cdr set1) set2)))
    (else (intersection (cdr set1) set2))))
```

Verzamelingen als geordende lijsten

In de tweede implementatie waren verzamelingen **geordende lijsten**

zorg dat een element dat wordt toegevoegd op de juiste plaats terecht komt

```
(define (insert el set)
  (cond
    ((empty? set) (list el))
    ((= el (car set)) set)
    ((< el (car set)) (cons el set))
    (else (cons (car set)
                (insert el (cdr set))))))
```

```
(define (element-of? el set)
  (cond
    ((empty? set) #f)
    ((= el (car set)) #t)
    ((< el (car set)) #f)
    (else (element-of? el (cdr set)))))

(define (delete el set)
  (cond
    ((empty? set) set)
    ((= el (car set)) (cdr set))
    ((< el (car set)) set)
    (else (cons (car set)
                (delete el (cdr set))))))
```

zoeken in een geordende lijst kan efficiënter

ook delete kan nu efficiënter

```
(define (intersection set1 set2)
  (cond
    ((or (empty? set1) (empty? set2))
     (create-set))
    ((= (car set1) (car set2))
     (cons (car set1)
       (intersection (cdr set1) (cdr set2))))
    ((< (car set1) (car set2))
     (intersection (cdr set1) set2))
    (else
     (intersection set1 (cdr set2)))))
```

```
(define (union set1 set2)
  (cond
    ((empty? set1) set2)
    ((empty? set2) set1)
    ((= (car set1) (car set2))
     (cons (car set1)
       (union (cdr set1) (cdr set2))))
    ((< (car set1) (car set2))
     (cons (car set1)
       (union (cdr set1) set2)))
    (else
     (cons (car set2)
       (union set1 (cdr set2))))))
```

ADT set

```
ADT set

create-set
   ( ∅ → set )
empty-set?
   ( set → boolean )
element-of?
   ( number set → boolean )
insert
   ( number set → set )
delete
   ( number set → set )
union
   ( set set → set )
intersection
   ( set set → set )
```

Over de representatie van verzamelingen (als een bepaald soort lijsten) en over de code van de implementatie wordt **bewust** niks gezegd.