Hoofdstuk 1

Inleiding

Primitieve Gegevens

Primitieve datawaarden zijn datawaarden die niet afbreekbaar zijn (in dezelfde programmeertaal)

boolean: `#t, #f`

symbol: `'apple`

number: `3, 0+4i, 3.1415, 22/7`

character: `#\a, #\newline, #\space`

procedure: `#<primitive:sin>`

Samengestelde Gegevens

Samengestelde data is data die met behulp van een dataconstructor geconstrueerd wordt. De samenstellende delen zijn primitieve data waarden of – op hun beurt – samengestelde data. Samengestelde data is dus data die afbreekbaar is (in dezelfde programmeertaal)

heet ook "datastructuur"

pair: `(1 . 2)`

`cons`

string: `"hello world"`

`make-string`

vector: `#(1 2 3)`

`make-vector`

Dataconstructoren

Dataconstructor:
Geheugen reserveren + Onderdelen initialiseren

Procedurele Constructoren

```
(make-string 10)
                    (cons 1 2)
        (make-vector 110)
```

Letterlijke Constructoren

```
                    #(2 1 "hi!")
"Madam, I'm Adam"
```

Accessoren, Mutatoren&Operaties

Accessoren

```
(String-ref s 10)

(car c)     (vector-ref v 7)
```

Mutatoren

```
(vector-set! v 10 "new")

                (set-car! c ())
```

Operaties

```
(reverse l)
                        (+ 2 110)

     (member? l element)
```

Datatypes

Elke datawaarde behoort tot een verzameling met een zekere naam: het datatype van de waarde

```
vector          number
      pair
              procedure
string
          boolean
```

Het datatype definieert ook welke operaties toepasbaar zijn op zijn waarden

```
(+ #t #f)
```

Algoritmen

Een algoritme is een algemene en berekenbare procedure om een probleem op te lossen

```
(define (fac n)
  (define (fac-rec n)
    (if (= n 0)
      1
      (* n (fac-rec (- n 1)))))
  (if (odd? n)
    (error "It's odd!")
    (fac-rec n)))
```

Abu Abdullah Muhammad ibn Musa al-Khwarizmi ("degene van Khwarizmi")

Eigenlijk gewoon een Scheme procedure

Om de code van een bankkaart te kraken bel je het orakel van Pluto!

Primitieve Algoritmen

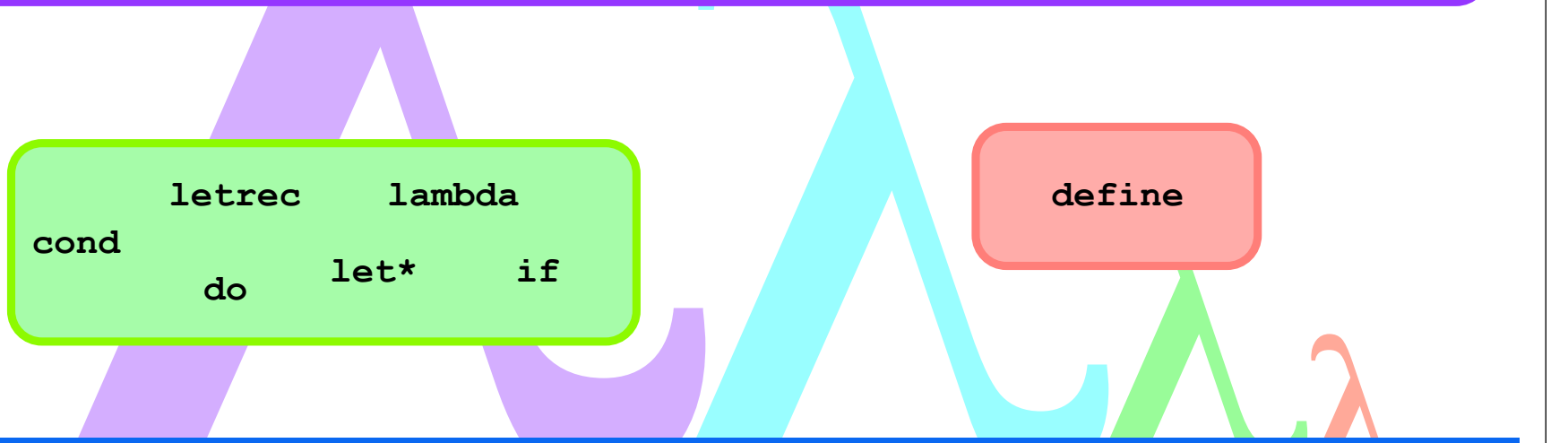
Een primitief algoritme is een algoritme dat niet afbreekbaar is (in dezelfde programmeertaal)

```
+    eq?      member?
        car
   -                sin
```

Sommige haakjes!

Samengestelde Algoritmen

Samengestelde algoritmen zijn algoritmen die met behulp van een algoritmische constructoren geconstrueerd wordt. De samenstellende delen zijn primitieve algoritmen of – op hun beurt – samengestelde algoritmen. Samengestelde algoritmen zijn dus afbreekbaar (in dezelfde programmeertaal)



Procedurele Abstractie

Procedurele abstractie bestaat erin een (betekenisvolle) naam te geven aan een samengestelde procedure

Verdeelt programmeurs in 2 groepen: gebruikers en implementator

```
(define (fib n)
  (define (iter n a b)
    (if (= n 0)
      b
      (iter (- n 1) b (+ a b))))
  (iter n 0 1))
```

Gebruikerscode leesbaar en korter. Implementatorcode makkelijk te onderhouden

```
(define (fib n)
  (if (< n 2)
    1
    (+ (fib (- n 1)) (fib (- n 2)))))
```

Proceduretypes

Een procedure werkt op invoer van een zeker datatype en produceert waarden van een zeker datatype. De combinatie van deze types is het proceduretype van de procedure.

```
sin     (number → number)
```

```
car     (pair → any)
```

```
append     (pair pair → pair)
```

```
(argtype-1 ... argtype-n → result-type)
```

Een complex voorbeeld

```
(argtype-1 ... argtype-n → result-type)
```

```
(define (zero f a b epsilon)
  (define c (/ (+ a b) 2))
  (cond ((< (abs (f c)) epsilon) c)
        ((< (* (f a) (f c)) 0) (zero f a c epsilon))
        (else (zero f c b epsilon))))
```

```
zero    ( (number → number) number number number → number )
```

Zo'n proceduretype is geen Scheme!

Data Abstractie

Procedurele abstractie bestaat erin een (betekenisvolle) naam te geven aan een samengestelde procedure

Data abstractie bestaat erin een (betekenisvolle) naam te geven aan samengestelde data

Verdeelt programmeurs in 2 groepen: gebruikers en implementators

Gebruikerscode leesbaar en korter. Implementatorcode makkelijk te onderhouden

Data abstractie wordt verwezenlijkt m.b.v. ADT's ...

Abstracte Data Types (ADT's)

Een ADT bestaat uit een naam voor een datatype en een aantal proceduretypes die de procedures beschrijven waarmee data elementen van het ADT gemanipuleerd kunnen worden. Dit zijn proceduretypes voor de constructoren, de accessoren, de mutators en de operaties.

Verdeelt programmeurs in 2 groepen: gebruikers en implementators

Een voorbeeld

```
ADT complex
        new          ( number number → complex )
        complex?     ( any → boolean )
        +            ( complex complex → complex )
        -            ( complex complex → complex )
        /            ( complex complex → complex )
        *            ( complex complex → complex )
        modulus      ( complex → number )
        argument     ( complex → number )
        real         ( complex → number )
        imag         ( complex → number )
```

Implementators kiezen voor een representatie en een implementatie

Procedurele ADT Implementatie

```
(library
 (complex)
 (export new complex? real imag + - / * modulus argument)
 (import (rename (except (rnrs base (6)) complex?)
                 (+ number+) (* number*) (/ number/) (- number-)))

 (define complex-tag 'complex)
 (define (get-real c)
   (cadr c))
 (define (get-imag c)
   (caddr c))

 (define (new r i)
   (list complex-tag r i))

 (define (complex? any)
   (and (pair? any)
        (eq? (car any) complex-tag)))

 (define (real c)
   (get-real c))

 (define (imag c)
   (get-imag c))
```

Niet-geëxporteerde namen zijn privé aan de implementatie

Refereer naar R6RS voor de import-clause

Procedurele ADT Implementatie

```
(define (+ c1 c2)
  (define real (number+ (get-real c1) (get-real c2)))
  (define imag (number+ (get-imag c1) (get-imag c2)))
  (new real imag))

(define (* c1 c2)
  (define real (number- (number* (get-real c1) (get-real c2))
                        (number* (get-imag c1) (get-imag c2))))
  (define imag (number+ (number* (get-real c1) (get-imag c2))
                        (number* (get-imag c1) (get-real c2))))
  (new real imag))

(define (- c1 c2)
  (define real (number- (get-real c1) (get-real c2)))
  (define imag (number- (get-imag c1) (get-imag c2)))
  (new real imag))
```

Procedurele ADT Implementatie

```
(define (/ c1 c2)
  (define denom (number+ (number* (get-real c2)
                                  (get-real c2))
                         (number* (get-imag c2)
                                  (get-imag c2))))
  (define real (number+ (number* (get-real c1)
                                 (get-real c2))
                        (number* (get-imag c1)
                                 (get-imag c2))))
  (define imag (number- (number* (get-imag c1)
                                 (get-real c2))
                        (number* (get-real c1)
                                 (get-imag c2))))
  (new (number/ real denom) (number/ imag denom)))

(define (modulus c)
  (sqrt (number+ (number* (get-real c) (get-real c))
                 (number* (get-imag c) (get-imag c)))))

(define (argument c)
  (atan (get-imag c) (get-real c))))
```

Gebruik van het ADT

```
(import (prefix (a-d examples complex (1)) complex:)
        (rnrs base (6))
        (rnrs io simple))

(define cpx1 (complex:new 1 4))
(define cpx2 (complex:new 5 3))

(display (complex:+ cpx1 cpx2))(newline)

(display (complex:* cpx1 cpx2))(newline)

(display (complex:+ cpx1 cpx2))(newline)

(display (complex:* cpx1 cpx2))(newline)

(display (complex:real cpx1))(newline)

(display (complex:imag cpx2))(newline)

(display (complex:modulus cpx1))(newline)

(display (complex:argument cpx2))
```

Object-gebaseerde Implementatie

```
(define (make-complex r i)
  (define (complex+ c)
    (make-complex (+ r (c 'real))
                  (+ i (c 'imag))))
  (define (complex* c)
    (make-complex (- (* r (c 'real))
                     (* i (c 'imag)))
                  (+ (* r (c 'imag))
                     (* i (c 'real)))))
  (define (complex- c)
    (make-complex (- r (c 'real))
                  (- i (c 'imag))))
  (define (complex/ c)
    (define denom (+ (* (c 'real)
                        (c 'real))
                     (* (c 'imag)
                        (c 'imag))))
    (define real (+ (* r (c 'real)) (* i (c 'imag))))
    (define imag (- (* i (c 'real)) (* r (c 'imag))))
    (make-complex (/ real denom) (/ imag denom)))
  (define (modulus)
    (sqrt (+ (* r r) (* i i))))
  (define (argument)
```

Object-gebaseerde Implementatie

```
(define (make-complex r i)
  ...
  (define (real)
    r)
  (define (imag)
    i)
  (lambda (message . args)
    (cond ((eq? message '+) (apply complex+ args))
          ((eq? message '-) (apply complex- args))
          ((eq? message '*) (apply complex* args))
          ((eq? message '/) (apply complex/ args))
          ((eq? message 'modulus) (modulus))
          ((eq? message 'argument) (argument))
          ((eq? message 'real) (real))
          ((eq? message 'imag) (imag))
          ((eq? message 'complex->list) (list 'complex r i))
          (else (error "Complex Number Message Not Understood")))))
```

De dispatcher encapsuleert privé procedures en data van de datastructuur

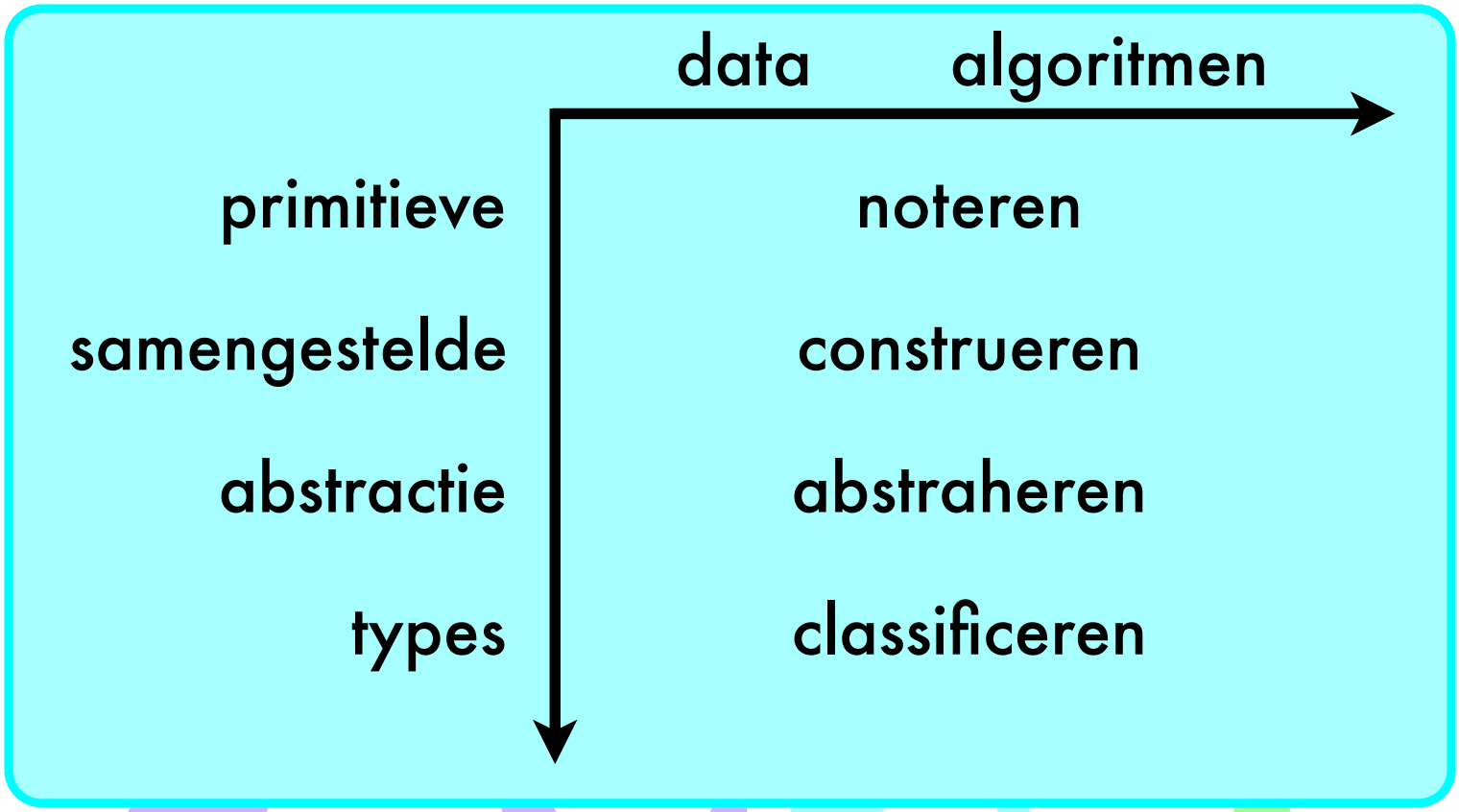
Gebruik van het ADT

```
(define cpx1 (make-complex 1 4))
(define cpx2 (make-complex 5 3))

(display ((cpx1 '+ cpx2) 'complex->list))(newline)

(display ((cpx1 '* cpx2) 'complex->list))(newline)

(display ((cpx1 '- cpx2) 'complex->list))(newline)

(display ((cpx1 '/ cpx2) 'complex->list))(newline)

(display (cpx1 'real))(newline)

(display (cpx2 'imag))(newline)

(display (cpx1 'modulus))(newline)

(display (cpx2 'argument))
```

De object-gebaseerde benadering is korter, veiliger en vervuilt minder

De object-gebaseerde is niet altijd veiliger, is complexer en consumeert meer geheugen

Samenvatting Concepten



Opslagdatastructuren

Een datastructuur die dient om datawaarden op te slaan noemen we een opslagdatastructuur (Eng: storage data structure). De opgeslagen waarden noemen we records .

Een telefoonboek

Complex getal

Een bibliotheekcatalogus

Voorbeeld Opslagdatastructuur

```
(library
 (max-o-mem)
 (export new max-o-mem? read write!)
 (import (rnrs base (6))
         (rnrs mutable-pairs))

 (define max-o-mem-tag 'max-o-mem)

 (define (new << init)
   (list max-o-mem-tag << init))

 (define (max-o-mem? any)
   (and (pair? any)
        (eq? (car any) max-o-mem-tag)))

 (define (read mom)
   (caddr mom))

 (define (write! mom new-value)
   (define << (cadr mom))
   (define value (caddr mom))
   (if (<< value new-value)
     (set-car! (cddr mom) new-value))
   mom))
```

De max-o-mem implementatie hangt niet af van het soort waarden die opgeslagen worden!

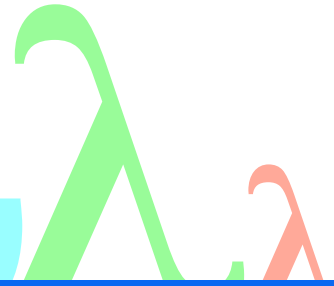
Enkel de grootste waarde wordt onthouden

Genericiteit

```
(library
 (max-o-mem)
 (export new max-o-mem? read write!)
 (import (rnrs base (6))
         (rnrs mutable-pairs))

 (define max-o-mem-tag 'max-o-mem)
```

Een generische datastructuur is een datastructuur die onafhankelijk is van het datatype van de elementen die in de datastructuur zitten. In Scheme wordt dit met een hogere-orde constructor bewerkstelligd.

```
    (caddr mom))

 (define (write! mom new-value)
   (define << (cadr mom))
   (define value (caddr mom))
   (if (<< value new-value)
     (set-car! (cddr mom) new-value))
   mom))
```



Generische ADT's

```
ADT max-o-mem< T >

new
    ( ( T T → boolean ) T → max-o-mem< T > )
max-o-mem?
    ( any → boolean )
write!
    ( max-o-mem< T > T → max-o-mem< T > )
read
    ( max-o-mem< T > → T )
```

ADT geparametrizeerd met het type T van de opgeslagen waarden

```
(define complex-mom (mom:new complex< (complex:new 0 0)))

(define number-mom (mom:new < 0))
```

```
complex-mom ∈ max-o-mem<complex>

number-mom ∈ max-o-mem<number>
```

Voorbeeld van Gebruik

```
(import (rnrs base (6)) (rnrs mutable-pairs (6)) (rnrs io simple)
        (prefix (a-d examples max-o-mem) mom:)
        (prefix (a-d examples complex (1)) complex:))

(define (greatest lst << init)
  (define max (mom:new << init))
  (define (iter lst)
    (mom:write! max (car lst))
    (if (not (null? (cdr lst)))
      (iter (cdr lst))))
  (iter lst)
  (mom:read max))

(define (complex< c1 c2)
  (define (square x) (* x x))
  (< (sqrt (+ (square (complex:real c1)) (square (complex:imag c1))))
     (sqrt (+ (square (complex:real c2)) (square (complex:imag c2))))))

(define integer-list (list 1 2 3 4 5))
(define complex-list (list (complex:new 1 0) (complex:new 0 1)
                           (complex:new 3 4) (complex:new 4 3)))

(display (greatest integer-list < 0))(newline)
(display (greatest complex-list complex< (complex:new 0 0)))(newline)
```

max ∈ max-o-mem<complex>

max ∈ max-o-mem<number>

Het Dictionary ADT

```
ADT dictionary< K V >

new
    ( ( K K → boolean ) → dictionary< K V > )
dictionary?
    ( any → boolean )
insert!
    ( dictionary< K V > K V → dictionary< K V > )
delete!
    ( dictionary< K V > K → dictionary< K V > )
find
    ( dictionary< K V > K → V U {#f} )
empty?
    ( dictionary< K V > → boolean )
full?
    ( dictionary< K V > → boolean )
```

key-value paren

associatief geheugen

records

sleutelvelden

satellietvelden

Dictionaries: Voorbeeld

```
(define english-dutch (new string=?))

(insert! (insert! english-dutch "algorithm" "algoritme")
                                "ADT" "Abstract Datatype")
(insert! english-dutch "key" "sleutel")
(insert! english-dutch "value" "waarde")
(insert! english-dutch "accessor" "uitlezer")
(insert! english-dutch "mutator" "overschrijver")
(insert! english-dutch "operation" "operatie")

(delete! english-dutch "mutator")
(if (find english-dutch "mutator")
  (error "It is still there!" "mutator"))
```

```
english-dutch ∈ dictionary<string string>
```

Performantie van Algoritmen

Mensen vinden computers goed als die snel zijn en weinig kosten. Geheugen kost geld.

De experimentele benadering

We zouden de snelheid van een algoritme kunnen meten met een stopwatch of met de `(time (algoritme))` expressie.

Nadelen: absoluut en niet algemeen

Snelheid Meten

We tellen het aantal computationele stappen die nodig zijn bij een invoer van grootte n.

Dit levert een functie op, $f_A(n)$, de performantiekarakteristiek van algoritme A

Deze methode is algemeen en relatief: we kunnen $f_A(n)$ en $f_B(n)$ op wetenschappelijke wijze vergelijken

Voorbeeld

```
(define (greatest l)
  (define (iter el l)
    (cond
       ((null? l) el)
       ((< el (car l)) (iter (car l) (cdr l)))
       (else (iter el (cdr l)))))
  (iter 0 l))
```

In het ergste geval is:
$f_{\texttt{biggest}}(n) = 6n + 2$

In het beste geval is:
$f_{\texttt{biggest}}(n) = 5n + 2$

Soorten Analyses

Bij een worst-case analyse onderzoeken we de performantiekarakteristiek van een algoritme en gaan we uit van de slechtst mogelijke invoer

Bij een best-case analyse onderzoeken we de performantiekarakteristiek van een algoritme en gaan we uit van de best mogelijke invoer

Bij een average-case analyse onderzoeken we de performantiekarakteristiek van een algoritme en gaan we uit van de gemiddelde invoer

Performantieschattingen: Motivatie

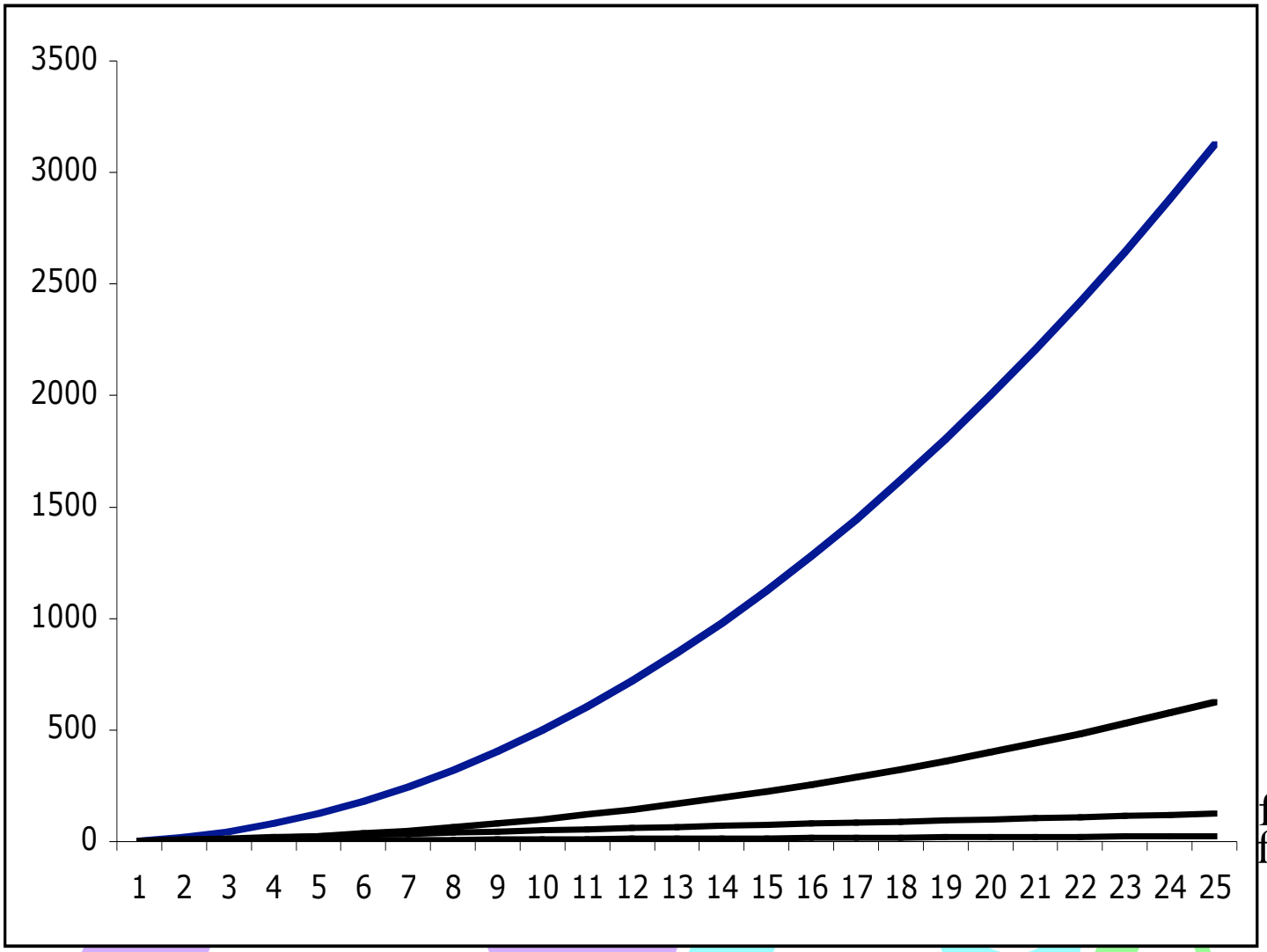
De ene primitieve operatie is de andere niet.

Eén primitieve operatie kan soms trager zijn dan 2 primitieve operaties

Een paar extra primitieve operaties (onafhankelijk van n) maken het verschil niet.

Beter is te zeggen dat "$f_{\texttt{biggest}}(n) = an+b$ voor zekere a en b, relatief klein"
Of nog: "het algoritme is lineair"

Voorbeeld: lineair vs. kwadratisch



Andere Voorbeelden

n	$log(n)$	$\sqrt{n}$	n	$n.log(n)$	n^2	n^3	2^n
2	1	2	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	3	8	24	64	512	256
16	4	4	16	64	256	4096	65536
32	5	6	32	160	1024	32768	4294967296
64	6	8	64	384	4096	262144	1.85×10^{19}
128	7	12	128	896	16384	2097152	3.40×10^{38}
256	8	16	256	2048	65536	16777216	1.16×10^{77}
512	9	23	512	4608	262144	134217728	1.34×10^{154}
1024	10	32	1024	10240	1048576	1073741824	1.79×10^{308}

We hebben een techniek nodig die ons laat zeggen dat n, 2n, 10n+5 allen van de soort an+b zijn en dat bvb. n^2 en $3n^2+4n+3$ van soort n^2 zijn

Grote Θ

$$\Theta(g(n)) = \{ f \mid \exists\, c_1, c_2 > 0, n_0 \geq 0 : \forall\, n \geq n_0 : 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n) \}$$

$5n \in \Theta(n)$

$5n^2 \notin \Theta(n)$

$n/34 \in \Theta(n)$

$6n+2.3 \in \Theta(n)$

$1 \notin \Theta(n^2)$

$5n \notin \Theta(n^2)$

$10n^3+3n^2 \notin \Theta(n^2)$

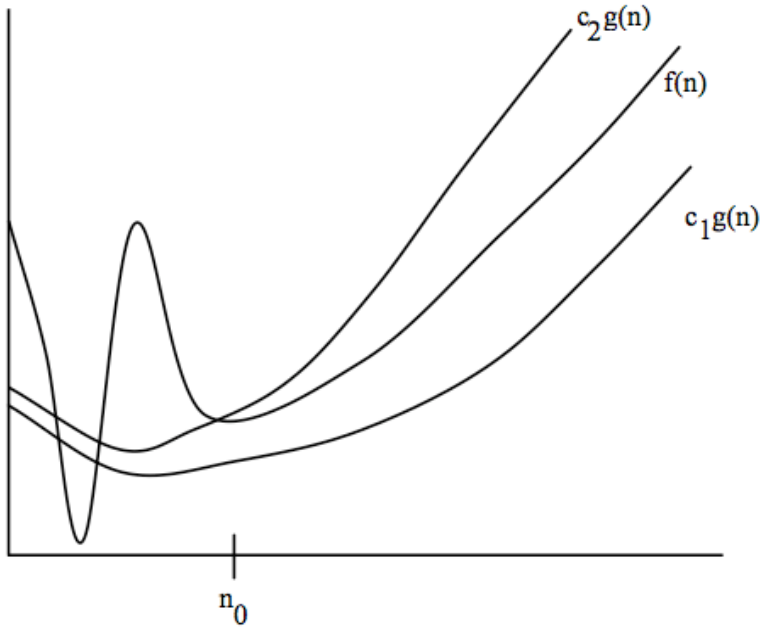


Figure 1.7: $f(n) \in \Theta(g(n))$

Grote O

$$O(g(n)) = \{ f \mid \exists\, c, n_0 \geq 0 : \forall\, n \geq n_0 : 0 \leq f(n) \leq c\, g(n) \}$$

$5n \in O(n)$

$5n^2 \notin O(n)$

$n/34 \in O(n)$

$6n+2.3 \in O(n)$

$1 \in O(n^2)$

$5n \in O(n^2)$

$10n^3+3n^2 \notin O(n^2)$

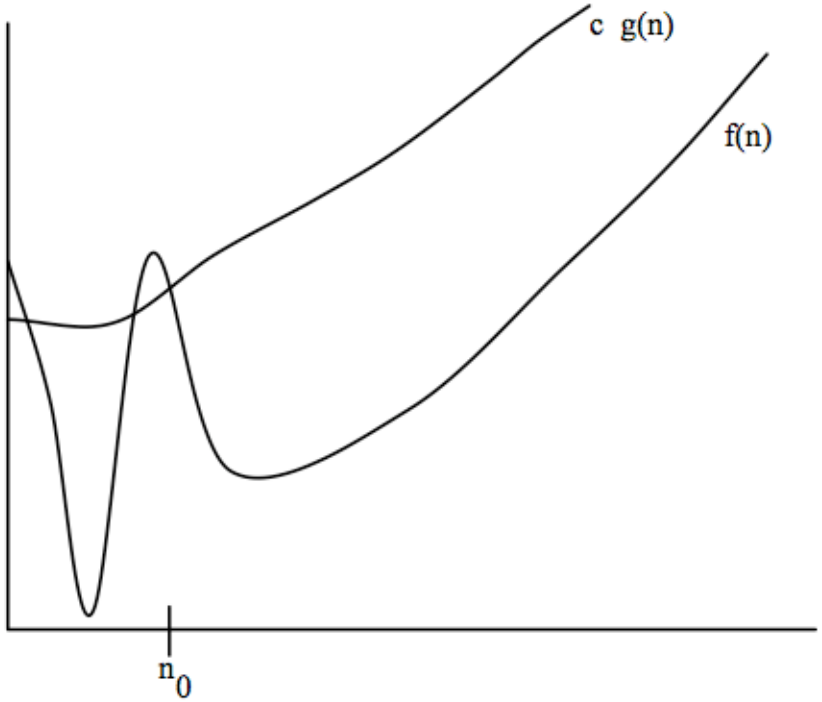


Figure 1.8: $f(n) \in O(g(n))$

Grote Ω

$$\Omega(g(n)) = \{ f \mid \exists\, c, n_0 \geq 0 : \forall\, n \geq n_0 : 0 \leq c\, g(n) \leq f(n) \}$$

$5n \in \Omega(n)$

$5n^2 \in \Omega(n)$

$n/34 \in \Omega(n)$

$6n+2.3 \in \Omega(n)$

$1 \notin \Omega(n^2)$

$5n \notin \Omega(n^2)$

$10n^3+3n^2 \in \Omega(n^2)$

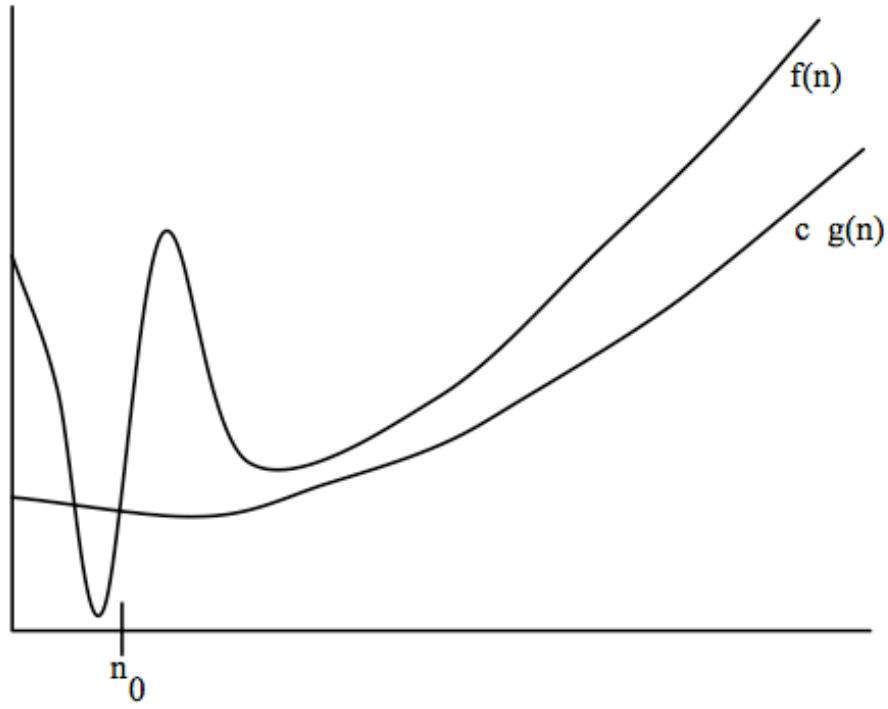


Figure 1.9: $f(n) \in \Omega(g(n))$

Eigenschappen + Opmerkingen

Voor alle functies f en g geldt:

$f \in \Theta(g)$ als en slechts als $f \in \Omega(g)$ én $f \in O(g)$

Let soms toch op:

- In zeldzame gevallen zitten er heel grote constanten in Θ, Ω en O-expressies verborgen
- Θ, Ω en O zijn enkel nuttig om functies van *verschillende* orde te vergelijken. Eén algoritme van O(n) kan toch véél sneller zijn dan een ander van O(n).

Rekenregels

- $O(c.f) = O(f)$ voor constanten c
  - Gevolg: $O(c) = O(1)$
  - Gevolg: $O(\log_a(n)) = O(\log_b(n))$
- Enkel dominante termen tellen mee:

  $O(t_1(n) + t_2(n)) = O (max(t_1(n),t_2(n)))$

  waarbij:

  $1 < \log(n) < \sqrt{n} < n < n.\log(n) < n^{k-1} < n^k < 2^n < n!$

Analyse van Scheme Expressies

$O(f_{\texttt{(set! v E)}}(n))$ $= O(1)+O(f_E(n)) = O(f_E(n))$

$O(f_{\texttt{(set-car! c E)}}(n))$ $= O(f_{\texttt{(set-cdr! c E)}}(n))$

$= O(f_E(n))$

$O(f_{\texttt{(lambda lst body)}}(n))$ $= O(1)$

$O(f_{\texttt{(define n E)}}(n))$ $= O(f_E(n))$

$O(f_{\texttt{(let ((v1 e1) (v2 e2) ... (vn en) b1 b2 .. bk)}}(n))$

$= O(\max_i(f_{ei}(n)) + \max_j(f_{bj}(n)))$

$= O(\max_{\{i,j\}}\{ f_{ei}(n), f_{bj}(n) \})$

$O(f_{\texttt{(begin e1 ... en)}}(n))$ $= O(f_{e1}(n) + \ldots + f_{en}(n))$

$= O(\max_i(f_{ei}(n)))$

$O(f_{\texttt{(cond ((c1 a1) ... (cn an)))}}(n))$ $= O(\max_i \{f_{ci}(n), f_{ai}\}(n)\})$

$O(f_{\texttt{(if c t e)}}(n))$ $= O(\max\{f_c(n), f_t(n), f_e(n)\})$

$O(f_{\texttt{(p a1 ... an)}}(n))$ $= O(1) + O(f_{ai}(n)) + \ldots + O(f_{an}(n))$

voor p ∈ { +, -, and, or, eq?, vector-ref, vector-set, car, cdr, ... }

$O(f_{\texttt{(f a1 ... an)}}(n))$ $= O(f_f(n)+f_{a1}(n)+ \ldots + f_{an}(n))$

`list->X, X->list, string-append,...` $\in O(n)$

De named let

```
(define (fac n)
  (let fac-loop
    ((result 1)
     (factor n))
    (if (= factor 0)
      result
      (fac-loop (* result factor) (- factor 1)))))
```

```
(define (fac n)
  (define (fac-loop result factor)
    (if (= factor 0)
      result
      (fac-loop (* result factor) (- factor 1))))
  (fac-loop 1 n))
```

Dus de performantiekarakteristiek zoeken is equivalent aan de analyse van een (recursieve) procedure.

Analyse van Scheme Procedures

Voor niet-recursieve Scheme procedures:
Bepaal O voor alle deelexpressies van de body en neem de som, t.t.z. het maximum.

```
(define (weird vector)
  (define len (vector-length vector))
  (if (odd? len)
    (a-linear-function len)
    (a-quadratic-function len)))
```

$f_{\texttt{weird}} \in O(n^2)$

Analyse van Scheme: Recursie

```
(define (fac n)
  (if (= n 0)
    1
    (* n (fac (- n 1)))))
```

Recurrentievergelijkingen

$f_{fac}(n) = 1 + f_{fac}(n-1)$

Vuistregel: voor een recursieve Scheme procedure bepaal je O(b(n)) voor de body en bepaal je O(r(n)) voor het aantal recursieve oproepen. De procedure is dan O(b(n).r(n))

$b_{fac}(n) \in O(1)$
$r_{fac}(n) \in O(n)$
Dus:
$f_{fac}(n) \in O(n)$

Voorbeelden

```
(define (fib1 n)
  (if (< n 2)
    1
    (+ (fib1 (- n 1)) (fib1 (- n 2)))))
```

$b_{fib1}(n) \in O(1)$

$r_{fib1}(n) \in O(2^n)$

Dus:

$f_{fib1}(n) \in O(2^n)$

Eigenlijk is:

$f_{fib1}(n) \in \Theta(\Phi^n)$

```
(define (fib2 n)
  (define (iter n a b)
    (if (= n 0)
      b
      (iter (- n 1) b (+ a b))))
  (iter n 0 1))
```

$b_{iter}(n) \in O(1)$

$r_{iter}(n) \in O(n)$

Dus:

$f_{iter}(n) \in O(n)$

Meerdere Parameters

```
(define (sum n m)
  (if (= n 0)
    m
    (+ 1 (sum (- n 1) m))))

(define (times n m)
  (if (= m 1)
    n
    (sum n (times n (- m 1)))))
```

$b_{sum}(n,m) \in O(1)$
$r_{sum}(n,m) \in O(n)$
Dus:
$f_{sum}(n,m) \in O(n)$

$b_{times}(n,m) \in O(n)$
$r_{times}(n,m) \in O(m)$
Dus:
$f_{times}(n,m) \in O(n.m)$

Analyse van Geheugenverbruik

Mensen vinden computers goed als die snel zijn en weinig kosten. Geheugen kost geld.

- De meeste algoritmen uit deze cursus zijn "in-place": ze consumeren geen extra geheugen naast het geheugen ingenomen door de invoer.
- Als algoritmen tóch extra geheugen vragen, meten we dat op dezelfde manier als tijd: Θ, Ω en O

`fib1` $\in \Theta(n)$
`fib2` $\in \Theta(1)$

Hoofdstuk 1

1.1 Data en Dataconstructoren

1.2 Algoritmen en Algoritmische Constructoren

1.3 Procedurele Abstractie en Proceduretypes

1.4 Data abstractie en ADT's
- 1.4.1 ADTs implementeren in Scheme
- 1.4.2 Abstractie vs. Genericiteit
- 1.4.3 Het Dictionary ADT

1.5 Het meten van performantie
- 1.5.1 Het meten van snelheid
- 1.5.2 Grote O
- 1.5.3 Grote O en Scheme Expressies
- 1.5.4 Grote O en Scheme Procedures
- 1.5.5 Het meten van geheugenverbruik