



VRIJE  
UNIVERSITEIT  
BRUSSEL



# KENNEN STRUCTUUR VAN COMPUTERPROGRAMMA'S 1

Ilyan Vermeulen Kaïs  
Boutaleb

Matisse Van  
Wassenhove

2024-2025

## Table of Contents

1	Hoofdstuk 1: Inleiding & Basisbegrippen .....	2
2	Hoofdstuk 2: Eenvoudige Procedures .....	2
4	Hoofdstuk 4: Soorten Procedures en Soorten processen.....	7
5	Hoofdstuk 5: Lijsten en Symbolische Data.....	10
6	Hoofdstuk 6: Hogere Orde Procedures .....	12
7	Hoofdstuk 7: Imperatief programmeren.....	16
8	Hoofdstuk 8: ADT.....	17
9	Hoofdstuk 9 .....	19
10	Hoofdstuk 10 .....	21
11	Hoofdstuk 11 .....	22
12	Hoofdstuk 12 .....	23
13	Hulpvolle figuren .....	25
14	Mogelijke vragen + extras .....	26

# 1 Hoofdstuk 1: Inleiding & Basisbegrippen

## Wat is een compiler en een interpreter/evaluator?

Een compiler/vertaler is een programma dat een programma (bron code) in 'e'en stap volledig omzet naar machine-code (in binair). De computer zal de machine-code uitvoeren.

Een interpreter of evaluator ('tolk' in het Nederlands) is een computerprogramma dat een code lijn per lijn uitvoert. De evaluator is zelf meestal gecompileerd.

## Wat zijn S-expressies?

De syntax van Scheme. S-expressies, van *symbolic expressions*, hebben 2 regels:

Elke atomaire expressie is een geldige expressie.

Een lijst van geldige expressies tussen haakjes is een geldige expressie.

Voorbeelden van atomaire expressies zijn "Strings", # t, # f, 2.5, 'symbol, #\a, ...

Een lijst van expressies is dan  $(exp_1 exp_2 exp_3 \dots exp_n)$ .

## Hoe werken de 3 verschillende notaties voor rekenkunde (prefix, infix, postfix)?

<b>Infix</b> Operator staat tussen operanden Operatoren hebben een voorrang Haakjes moeten worden gebruikt om voorrangsregels te omzeilen	$(1 + 2) \cdot 3 + \sin 5$
<b>Prefix</b> Operatoren worden gevolgd door hun operanden Voorrangsregels zijn onnodig	+ * + 1 2 3 sin 5
<b>Postfix</b> Operatoren staan meteen na hun operanden Voorrangsregels zijn niet nodig De expressies van links naar rechts lezen. onthoud de getallen en de eerste operator dat je ontmoet voer je uit op deze getallen.	1 2 + 3 * 5 sin +

## REPL terminologie begrijpen: read, eval, print. Verschil tussen expressies en waarden begrijpen.

De REPL (Read, Eval, Print, Loop) is het proces waarmee een evaluator expressies uitvoert. De REPL "consumeert" telkens 'e'en expressie: de syntax wordt nagekeken (*read*), haar waarden worden berekend (*eval*) en deze wordt vervolgens op het scherm in leesbare vorm gezet (*print*).

## Het verschil begrijpen tussen syntax fouten en runtime fouten.

*Syntax error* gebeurt wanneer de read fase fout loopt. De syntax van de expressie is dus niet correct.

*Runtime error* gebeurt wanneer een fout in de eval fase loopt. Dit betekent dat de semantiek van de expressie incorrect is (ze heeft geen betekenis).

# 2 Hoofdstuk 2: Eenvoudige Procedures

## Het concept 'nesting' kunnen uitleggen in de context van geneste expressies

In programmeertalen is "nesting" het concept van expressies binnen expressies te plaatsen. Bijvoorbeeld,  $(+ (\cdot 2 3) 4)$  is een geneste expressie waarbij  $(\cdot 2 3)$  binnen de  $+$ -procedure ligt. In principe kan nesting willekeurig diep gebeuren.

## Begrijpen hoe samengestelde geneste uitdrukkingen (mechanisch) geëvalueerd worden.

Een samengestelde uitdrukking, ter verduidelijking, is een expressie van de vorm  $(+ a b)$ . Zo'n uitdrukking wordt geëvalueerd door eerst de onderdelen te evalueren en dan de samenstelling te evalueren.

Bijvoorbeeld, stel  $(+ 1 2)$ . Eerst wordt  $+$  geëvalueerd als  $< \text{procedure} : + >$ , 1 als de atomaire waarde 1, en 2 als de atomaire waarde 2. Dan wordt  $(+ 1 2)$  geëvalueerd.

**OPGELET:** De onderdelen worden in een niet-gespecificeerde volgorde geëvalueerd. Verschillende evaluatoren voor verschillende talen worden uitgevoerd op hun eigen wijze (van links naar rechts, van rechts naar links, etc.) Dit betekent dat bij geneste defines op een zelfde scope kunnen niet op elkaar afhangen omdat je niet kan weten wanneer ze worden geëvalueerd. Bij de taal Racket evalueert racket van links naar rechts. We kunnen dit zien door via de trace en gepaste parameters.

## Terminologie over variabelen, bindingen en omgeving kunnen uitleggen.

Om expressies te versimpelen, geven we een betekenisvolle naam aan deze expressies. Een expressie met een naam noemen we een *variabele*. De variabele wordt dan gebonden aan de waarde van de expressie in kwestie. Al deze *bindingen* worden opgeslagen in de *globale omgeving* binnen de computer.

## Algemene formaat van de eerste vorm van define kennen.

Een *special form* is een samengestelde uitdrukking waarvan de component die in de eerste positie staat een gereserveerd 'sleutelwoord' is. Consulteer de documentatie van R5RS om alle special forms te vinden.

*define* is een *special form* met de volgende syntax:

(define < naam > <        >)

Dit is de eerste vorm van *define*, die gebruikt wordt om variabelen te definiëren. Met < naam > bedoelen we een naam. Dit wordt een *placeholder* genoemd. Met < > bedoelen we een expressie. De waarde van deze uitdrukking zal gebonden zijn aan < naam >. Vanaf nu zal het symbool een expressie voorstellen. Uitdrukking en expressie zullen verder in dit document door elkaar gebruikt worden.

## Weten wat ariteit, unair, binair, ternair betekent.

Een procedure verwacht een aantal operanden. Het aantal operanden noemen we de *ariteit* van de *procedure*. Stel een procedure (define (func  $a_1 a_2 \dots a_{n-1} a_n$ )), dan heeft procedure func een ariteit van  $n$ .

Ariteit	Naam
1	Unair
2	Binair
3	Ternair

## Weten wat booleans zijn + werking van and/or/not kennen.

*Boolean* is een datatype met slechts twee waarden, waar en vals, respectievelijk geschreven als *#t* en *#f* in Scheme. Met de special forms *and*, *or* en *not* kan je dan met booleans "rekenen".

Syntax
(and < expr <sub>1</sub> >        ...        < expr <sub>n</sub> >)
(or    < expr <sub>1</sub> > ... < expr <sub>n</sub> >) (not < expr >)

'and' geeft #t terug wanneer beide expressies waar zijn.

'or' geeft #t wanneer ten minste één expressie waar is.

'not' geeft het tegenovergestelde van het resultaat van de booleane expressie.

'and' en 'or' stoppen met evalueren vanaf dat een conditie fout of waar komt voor 'and' en 'or' respectievelijk. Dit betekent dat ook al is een element in die getest moet worden syntactisch fout zou zijn, of een runtime error zou moeten opleveren, zolang dit element na de stop conditie te voor komt

zal er geen error geleverd worden. Deze eigenschap kan geëxploiteerd worden bij sommige procedures.

## Weten wat relationele operatoren zijn.

*Relationele operatoren* kijken na of een wiskundige relatie geldt voor hun argumenten.

`= > >= < <=` : de relationele operatoren.

De relationele operatoren zijn predicaten.

## Weten wat een predicaat is.

Procedures die een boolean teruggeven noemen we een *predicaat*. Bij conventie eindigt de naam van een predicaat met '?'.  
Type-predicaten

## Weten wat types zijn. De type-predicaten kennen.

Alle expressies hebben een type om te bepalen of een procedure uitgevoerd kan worden op een gegeven input.

De type-predicaten geven #t terug als het gegeven object van het gevraagde datatype is.

## Tweede vorm van define kennen + kunnen onderscheiden van de eerste vorm.

`(define (< naam > < parameter >))`

Dit is de tweede vorm van de define special form. Aan de hand van deze vorm kan men procedures definiëren.

## Terminologie procedures kennen: argumenten, ...

`(define (kwadraat x) (* x x))` is een *procedure definitie*. De *x* noemen we de *formele parameter* of kortweg de *parameter*.

`(kwadraat 10)` noemen we dan de *procedure oproep* of *toepassing*. De formele parameter heeft de waarde 10 gekregen; we noemen 10 de *actuele parameter* of *argument*.

## Verschil tussen substitutie model en omgevings model kennen om functie-oproepen te verklaren.

Om een procedure oproep uit te rekenen voor een zelfgeschreven procedure reken je eerst de argumenten uit. Dan pak je de body en *substitueer* je de argumenten in de parameters en reken je de body verder uit.

## Wat is procedurele abstractie? Waarom is het nuttig/nodig?

*Procedurele abstractie* is het concept om aan een procedure een aangepaste naam te geven zodat complexere procedures simpele namen gebruiken om zo het programma helder te houden. Dit maakt de *implementatie* van de code ook onbelangrijk.

Dit is van belang bij zeer grote en complexe programma's. Stel dat we dit niet zouden doen, dan zou men een procedure die het kwadraat van een getal berekent "g\$+@?t45 y" kunnen noemen. Bij programma's met honderden procedures wordt dit onmogelijk te volgen.

## Wat is hergebruik? Waarom is het nuttig/nodig?

Code hergebruiken betekent deze slechts 'e'en keer schrijven, 'e'en keer debuggen en onderhouden.

## Verschil tussen Racket en R5RS begrijpen.

R5RS is een kleine taal met weinig concepten en beperkte toepasbaarheid.

Type-predicaten
number?
boolean?
real?
integer?
procedure?
string?
image?
zero?
null?
pair?
symbol?
vector?

Racket is een grote taal met veel concepten. We zeggen van zo'n taal dat ze *batteries included* is, wat betekent dat een programmeertaal alles bevat dat nodig is om vrij wel elk mogelijk programma aan te maken. Met racket kan je bijvoorbeeld files met Racket code importeren uit een *bibliotheek/library*. Alle definities van de bibliotheek worden aan de omgeving toegevoegd.

## Begrijpen dat rekenen met getallen, booleans, strings en prentjes in Scheme exact dezelfde principes volgt.

Net als getallen zijn prentjes, strings en booleans waarden die we aan een procedure kunnen meegeven. Procedures zijn type afhankelijk maar de wijze waarop ze functioneren is dezelfde.

## 3 Hoofdstuk 3: Complexe Procedures: Beslissingen, Recursie en Lokale Definities

### Werking van if en cond begrijpen.

Syntax
(if < Test <sub>1</sub> > < waar <sub>2</sub> > < vals <sub>3</sub> >)
De eerste uitdrukking moet een boolean zijn en noemen we de <i>test</i> . De tweede waarde wordt gerunt als de test waar is en noemen we het <i>consequent</i> . Het derde wordt gerunt als de test vals is en noemen we het <i>alternatief</i> .
(cond(< Test <sub>1a</sub> >< expr <sub>1b</sub> >) < Test <sub>2a</sub> >< expr <sub>2b</sub> > ... (<Test <sub>(n-1)a</sub> >< expr <sub>(n-1)b</sub> >) (else < expr <sub>n</sub> >))
cond neemt een lijst van <i>clausules</i> . Elke clausule is een koppel uitdrukkingen die 'e'en voor 'e'en getest worden. De expressie van de eerste clausule waarvan de test waar is wordt uitgevoerd. De cond procedure wordt stopgezet wanneer de expressie van de ware clausule uitgevoerd werd. Als geen procedure waar is wordt de else statement uitgevoerd. else is optioneel en als afwezig maar toch geen test waar is geeft cond dan void terug.

### Subtiel verschil met of zonder "else" begrijpen.

Zie laatste punt vorige onderdeel.

### Lokaliteit van lokale procedures en formele parameters begrijpen.

Een *lokale variabele* is een variabele die enkel zichtbaar is binnen de procedure. Dit betekent dat deze niet in de globale omgeving terecht komt.

De formele parameter hier is dan een placeholder binnen de expressie van de procedure voor de actuele parameter. Deze is niet gebonden aan een waarde binnen de globale omgeving. Wanneer de procedure opgeroepen wordt zal de formele parameter gebonden worden aan de gegeven waarde.

### Lokaliteit kunnen uitleggen in termen van het omgevingsmodel.

In de context van het atomaschrift model, een lokale procedure maakt een nieuwe pagina aan met zijn eigen variabelen. De nieuwe pagina kan de voorgaande pagina's bindingen wel gebruiken, maar niet andersom. Elke lokale procedure oproep krijgt een apart blad.

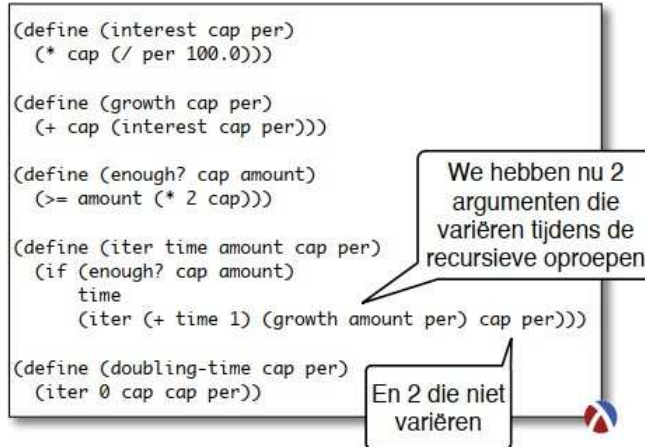
### Tracing van recursieve procedures begrijpen.

Een *recursieve procedure* is een procedure dat zichzelf oproept. De trace van zo'n procedure lijkt toren, met de basis de eerste oproep en de top de laatste oproep.

### Algoritme voor vierkantswortel kennen.

$$y = \sqrt{x} \iff y^2 = x$$
$$y_n = \frac{y_{n-1} + \frac{x}{y_{n-1}}}{2}$$

### Algoritme voor verdubbeltijd berekenen kennen.



[4]

**Aan de hand van het omgevingsmodel kunnen uitleggen hoe een recursieve procedure de verschillende zelf-oproepen uit mekaar kan houden.**

Elke oproep van een recursieve procedure maakt een nieuw blad aan en het resultaat van deze procedure wordt vervolgens gegeven aan het voorgaande blad.

**Variabelen in programmatekst kunnen classificeren als vrij of gebonden (statisch).**

Een *gebonden variabele* is de formele parameter van een procedure.

Een *vrij variabele* is een variabele gebonden binnen de globale omgeving.

**Verschil tussen 2 soorten “gebonden” begrijpen.**

*Statisch gebonden* betekent dat een variabele  $x$  nog geen waarde heeft en gebonden is aan een procedure. Syntactisch (de formele parameter).

*Dynamisch gebonden* waarde toevoegen aan een variabele tijdens de runtime. bv. (define  $x$  5)

**Weten wat een blokgestructureerde programmeertaal is. Weten wat scope-niveau's zijn.**

Een *blokgestructureerde programmeertaal* is het concept waarbij een opgeroepen lokale procedure zijn eigen 'blok'/bladzijde (atoma model). De *scope*/zichtbaarheid van deze blok bevat elke blok/blad er rond. (zie dia 30)

**Verschil tussen lexicale en dynamische scoping in programmeertalen kunnen uitleggen.**

*Lexical scope*: De waarde van een variabele wordt bepaald door de plaats in de programmacode waar de variabele gedefinieerd staat (t.t.z. door haar lexicale context). Je zoekt de betekenis van variabele op door “omhoog te lezen”.

*Dynamic scope*: De waarde van een variabele hangt af van de plaats in de “trace” van het programma waarin de variabele gedefinieerd stond. Je zoekt de betekenis van een variabele op door achterstevoren te lezen in de procedures die opgeroepen werden.

### Lexicale scoping kunnen uitleggen aan de hand van het omgevingsmodel.

Bij lexicale scoping worden de variabelen van boven naar onder toegevoegd aan de globale omgeving. Bij geneste procedures betekent dit dat de buitenste procedure eerst voorkomt en de binnenste procedure de vorige procedure's scope in zijn eigen scope heeft.

## 4 Hoofdstuk 4: Soorten Procedures en Soorten processen

### Tracing begrijpen en recursieve procedures.

Er zijn verschillende vormen van recursie: *lineair recursief proces*, *staartrecursie*, *lineair iteratief proces*, ...

Aan de hand van de vorm van de tracing kan je zien welk soort proces uitgevoerd wordt.

Bij lineair recursief proces heb je eerst een afdaling in de recursie tot dat de stop conditie waar is, en dan gebeurt er het tweede deel dat we het terugkeren of *backtracken* uit de recursie noemen.

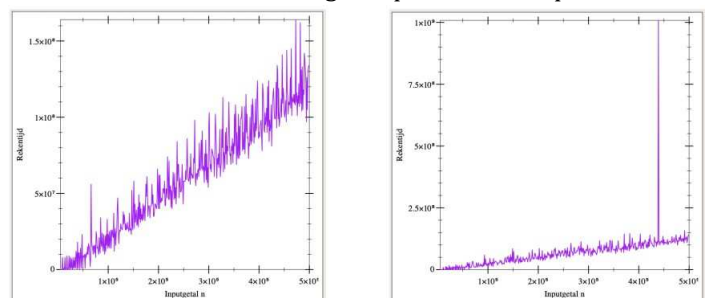
Bij lineair iteratief moet je evenveel procedure oproepen maken maar is backtracking niet nodig. Het blijft dus op hetzelfde niveau.

### Weten wat een profile is. Begrijpen wat de X-as en Y-as voorstellen.

De term *profiling* betekent de tijd experimenteel te meten. We meten hoe lang een procedure loopt afhankelijk van verschillende invoeren. De X-as is het inputgetal  $n$ , de Y-as is de tijd.

### Weten waarom profiles zoveel aberraties hebben.

*Aberraties* zijn de pieken in de profile grafieken en de oorzaak ervan zijn de achtergrond processen binnen de computer zelf. Als er een heel



Figuur 1: Profile grafieken

grote piek voorkomt dan is dit een verwaarloosbare uitschieter.

### Weten wat lineaire recursie is.

Bij lineair constructief proces heb je eerst een afdaling in de recursie tot dat de stop conditie waar is. Dan gebeurt er het tweede deel dat we het terugkeren of *backtracking* uit de recursie noemen. Als binnen de body van een lineair proces een procedure opgeproeft wordt, dan noemen we dit een constructief recursief proces.

### Het verschil tussen recursieve procedures die een recursief en iteratief proces genereren.

Een recursieve procedure die een iteratief proces genereert doet dit door middel van staartrecursie (functie oproep in staartpositie).

Om een recursief proces te genereren wordt gebruikt gemaakt van constructieve recursie waarbij er nog een operatie wordt uitgevoerd op het resultaat.

We zien dan dat recursieve procedures meer geheugen nodig hebben terwijl iteratieve procedures een constant geheugen verbruik hebben.



Iteratieve processen zijn doorgaans een beetje sneller dan recursieve processen, zelfs indien beide processen hetzelfde uitrekenen en beide processen lineair zijn. Er is wat minder werk te doen voor de computer bij iteratieve processen.

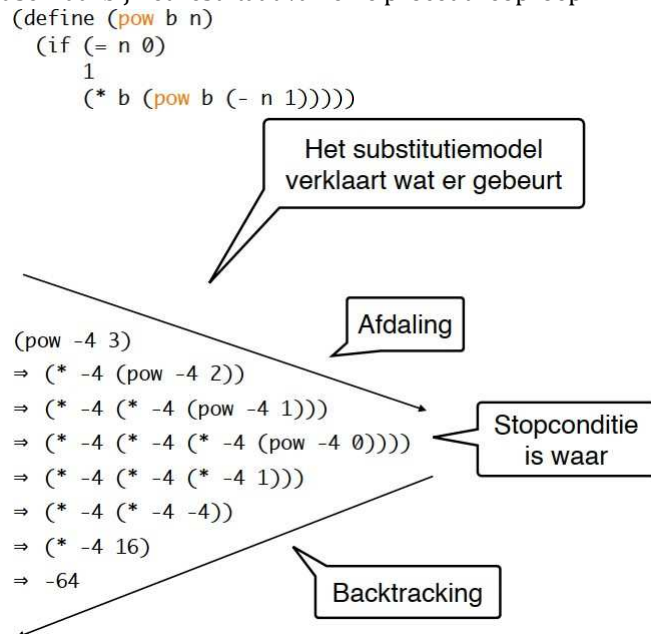
## Weten wat backtracking is. Weten waarom backtracking niet nodig is in een iteratief proces.

Bij lineaire recursie noemen we backtracking de fase waarbij het resultaat van elke procedureoproep opgebouwd wordt tot dat het resultaat van de eerste oproep teruggegeven wordt. In de context van het atoma model, de pagina's worden verwijderd en het antwoord naar de vorige pagina gestuurd, die deze gebruikt om zelf het antwoord naar de vorige pagina te sturen enz.

## Weten welke (iteratief/recursief) doorgaans betere profiles oplevert.

Iteratieve processen geven beter profiles.

Omdat in tegenstelling tot recursie, iteratieve processen hebben geen backtracking, en in de context van het atoma model worden er dus geen nieuwe bladen aangemaakt en verwijderd om een resultaat te bekomen. Omdat het aantal stappen bij recursie en iteratie gelijk zijn, maar recursieve processen met extra bladzijden moeten foefelen, heeft iteratie bijgevolg de betere profile.



Figuur 2: Constructieve recursie

do-loops (zie later) hebben een nog betere profile. Wanneer staartrecursie gebeurt zal Scheme intern een do-loop aanmaken om deze recursie uit te voeren. Meteen een loop aanmaken is equivalent met de tussenpersoon weg te nemen. De winst hierbij is wel verwaarloosbaar klein.

## Weten wat een accumulator is in een iteratief proces.

De *accumulator* is de parameter die het resultaat opbouwt. Het is 'e'en van de formele parameters van de procedure.

## Weten wat boomrecursie is.

Als een recursieve procedure zichzelf meer dan 'e'en keer oproept, dan zal het aantal procedure oproepen groeien zoals te zien in fig.4. Dit proces is meestal exponentieel.

## Weten wat een logaritmisch proces is.

Een logaritmisch proces is een procedure die uitgevoerd wordt volgens logaritmische tijd. Dit is de snelst mogelijke procedure. Zie fig.3.

## Weten wat sublineariteit is.

Bij fig.3 De curves onder de lineaire curve noemen we *sublineair*. De curves erboven noemen we *superlineair*.

## Het verschil tussen een best-case en worst-case input begrijpen.

Een best-case input is zo'n input die steeds de meest optimale processen gebruikt van de procedure die deze input opneemt. Wat er bedoeld wordt met processen hier is zo'n proces die de nodige rekenwerk verlaagt en dus de procedure versnelt.

De worst-case input in tegenstelling gebruikt de slechtste processen van de procedure.

## Weten wat iteratieve code is. De algemene vorm van een do kennen.

Iteratieve code heeft een vooraf gedefinieerde geheugen verbruik, in tegenstelling tot recursieve code waarvan het geheugen verbruik niet op voorhand gedefinieerd kan worden. Volgens het atoma model, geen nieuw blad moet bij een iteratief recursieve oproep worden opgemaakt.

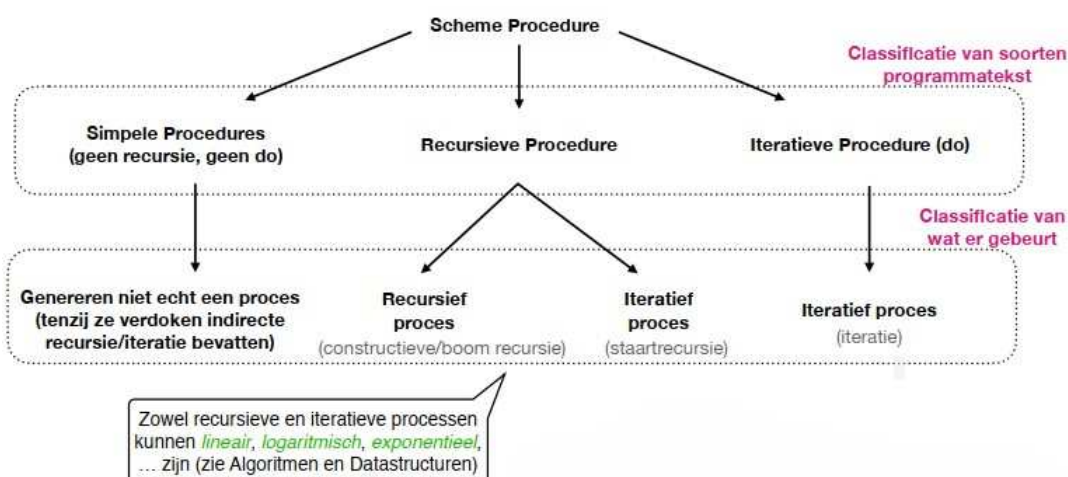
Er bestaat een special form *do*, en deze is een iteratieve procedure.

Syntax
<pre>(do  ((&lt; naam<sub>1</sub> &gt;&lt; init<sub>1</sub> &gt;&lt; update<sub>1</sub> &gt;) (&lt; naam<sub>2</sub> &gt;&lt; init<sub>2</sub> &gt;&lt; update<sub>2</sub> &gt;) ... (&lt; naam<sub>n</sub> &gt;&lt; init<sub>n</sub> &gt;&lt; update<sub>n</sub> &gt;)) (&lt; end-test &gt;&lt; result-exp<sub>1</sub> &gt; ... &lt; result-exp<sub>k</sub> &gt;) &lt; body-expr<sub>1</sub> &gt; ... &lt; body-expr<sub>m</sub> &gt;)</pre>
<p>De omgeving wordt tijdelijk uitgebreid met de n gegeven namen &lt; naam<sub>i</sub> &gt;.  &lt; naam<sub>i</sub> &gt; wordt gebonden aan &lt; init<sub>i</sub> &gt;.  &lt; end-test &gt; wordt dan uitgevoerd.  Als end-test #t, dan worden &lt; result-exp<sub>1</sub> &gt; tot k uitgevoerd.  Als end-test #f, dan worden alle body expressies 'e'en na 'e'en uitgevoerd.  Na dat de body-expr uitgevoerd zijn, worden de namen herbonden door de &lt; update<sub>i</sub> &gt; uit te voeren.  Als de 'do' eindigt, de tijdelijke uitbreiding van de omgeving wordt verwijderd.  Het resultaat wordt teruggegeven.</p>

## Weten wat het verband is tussen de profiles van een recursief proces, een iteratief proces (op basis van recursieve procedure) en een iteratief proces (op basis van een iteratieve procedure).

De do-loop is de snelste procedure, gevolgd door iteratieve procedures zonder do-loop en ten laatste de recursieve procedure. Het verschil tussen de do-loop en andere iteratieve processen is echter verwaarloosbaar klein, als te zien is in fig.5.

## Helder het verband tussen procedures en processen kunnen toelichten met de juiste woordenschat.



## Weten wat indirecte/mutuele recursie is.

Indirecte recursie gebeurt wanneer een procedure een andere procedure oproept en deze vervolgens de eerste procedure oproept. Een willekeurig aantal procedures kunnen tussen de eerste en laatste procedure staan, zolang procedure 1 heropgeroep wordt en het proces herbegint.

## 5 Hoofdstuk 5: Lijsten en Symbolische Data

De “wetten” van cons/car/cdr kennen.

Syntax
(cons < value <sub>1</sub> >< value <sub>2</sub> >)
(cdr(cons < v <sub>1</sub> >< v <sub>2</sub> >))
(car(cons < v <sub>1</sub> >< v <sub>2</sub> >))
cons neemt een paar waarden op. consgeeft een pair of dotted pair op. car geeft de rechter waarde van cons (v <sub>1</sub> ). cdr geeft de linker waarde van cons (v <sub>2</sub> ).

Bijvoorbeeld (cons 5 10) print (5 10).

cons staat voor *construct*, car voor *contents adresss register*, cdr voor *contents decrement register*.

Wanneer cons opgeroepen wordt zal een de *memory manager* zoeken of er twee vrije geheugen hokjes naast elkaar staan.

Dr Racket kent enkel de taal Racket, geen R5RS. Dr Racket zal dus deze special forms omzetten naar hun Racket equivalenten: mcons, mcar en mcdr waarbij de m voor mutable staat. Zo simuleert Dr Racket andere Scheme talen.

### Weten wat een garbage collector is.

Wanneer een waarde niet meer nodig is, dan wordt deze verwijderd door de *garbage collector* om zo het computergeheugen vrij te laten. Deze waarden zijn de ongebonden/anoniem waarden die niet voorkomen binnen de omgeving. Bijvoorbeeld, als we de atomaire waarde 7 in de REPL tikken, dan zal de REPL 7 printen en deze waarde vervolgens wissen. Als 7 gebonden is aan *zeven*, dan is deze niet anoniem en wordt het niet verwijderd. Hoe dit gebeurt valt niet onder de scope van dit vak.

### Weten wat een datastructuur is.

Alles dat Scheme herkent als een bruikbaar object is een datastructuur. Voorbeelden zijn strings, getallen, lijsten, characters, booleans, enz.

### Weten wat een lijst is.

Aan de hand van cons kan je structuren maken die te voorstellen zijn door *box-and-pointer diagrammen*:

$$(\text{cons } < v >_1 (\text{cons } < v >_n (\text{cons} \dots (\text{cons } < v >_{n-1} < v >_n))))^1 \quad (1)$$

Op de tweede positie komt een nieuwe paar voor en deze reeks volgt dezelfde procedure om meer paren te kunnen binden met elkaar.

De lege lijst is een lijst.

Elke reeks paren die (via cdrs) aan mekaar hangen en die eindigen met een lege-lijst is ook een lijst.

## Ingebouwde basisprocedures op lijsten kennen.

Procedure	Betekenis
(length < list >)	Geeft lengte van lijst weer.
(list-tail < list > < number >)	Geeft de lijst weer vanaf < number >.
(list-ref < list > < number >)	Geeft element van < list > met adres < number >.
(reverse < list >)	Draait de lijst om.
(append < list <sub>1</sub> > < list <sub>2</sub> >)	Plakt < list <sub>2</sub> > aan het einde van < list <sub>1</sub> >.
(pair? < v >)	Type predicaat voor lijsten.

De ingebouwde lijstprocedures zijn niet-*destructief*. Dit betekent dat de input-lijst niet veranderd. Als ze een lijst teruggeven zou dit dan een nieuwe lijst zijn. Als een bestaande datastructuur wel zou veranderen, dan noemen we zo'n operatie wel destructief.

## Weten waarom length en list-ref traag zijn. Begrijpen hoe lijsten (of paren in het algemeen) in het computergeheugen naar mekaar kunnen "wijzen".

list-ref en length zijn twee dingen tegelijk aan het doen:

1. Het gebruikt cdr om door de lijst te bewegen.
2. Het telt hoeveel keer cdr gebruikt werd.

list-ref geeft dan het gevonden element eens de teller van het tweede proces gelijk is aan het gegeven getal in list-ref.

length gebruikt cdr tot het einde van de lijst en geeft dan haar teller weer.

Beide procedures zijn dus recursief iteratief.

## Implementatie kennen van de basisprocedures: sum/square-all, reverse, flatten, member, length, append.

```
(define (sum-all lst)
  (if (null? lst)
      0
      (+ (car lst) (sum-all (cdr lst)))))
-----
(define (reverse lst)
  (if (null? lst)
      '()
      (append (reverse (cdr lst)) (list (car lst)))))
-----
(define (flatten lst)
  (if (null? lst)
      '()
      (append (car lst) (flatten (cdr lst)))))
-----
(define (member? elm lst)
  (and (not (null? lst))
       (or (= elm (car lst))
           (member? elm (cdr lst)))))
-----
(define (my-append l1 l2)
  (if (null? l1)
      l2
      (cons (car l1) (my-append (cdr l1) l2))))
-----
(define (my-length lst)
  (define (my-length-iter l acc)
    (if (null? l)
        acc
        (my-length-iter (cdr l) (+ 1 acc)))))
```

```
(if (null? l)
    acc
    (my-length-iter (cdr l) (+ acc 1)))
(my-length-iter lst 0))
```

## Quote begrijpen voor alle gezien types.

Syntax
(quote < expr >) of ' < expr > expr kan alles zijn, van atomaire expressies procedures.

*quote* is een special form. De gegeven expr wordt niet geëvalueerd en wordt door de REPL zo geprint.

## Weten wat een symbol is.

Een symbol is een datatype wiens atomaire waarde een quote is. 'a is bijvoorbeeld een symbol. Enkel waarden die, als niet gequoteerd een variabele zouden kunnen zijn tellen als symbols. Waarden zoals '5 of '+ zijn dus geen symbolen.

## Weten wat associatielijst is + assoc.

Een *associatielijst* is een lijst van koppels. Een koppel heeft een *key* en een *value*. Voor degenen die python kennen, dit is de equivalent van een dictionary.

Er bestaan 2 procedures op zo'n lijsten: *assq* en *assoc*. Om het verschil tussen beide uit te leggen introduceren we respectievelijk 2 predicaten en procedures : *eq?* en *equal?*, krachtigere versies van = die enkel op numerieke waarden werkt, en *memq* en *member*.

*eq?* bekijkt of twee dingen gelijk zijn geheugenwijs. Stel bijvoorbeeld een variabele *vijf* en de atomaire waarde 5. (*eq? vijf 5*) print #t omdat *vijf* aan de atomaire waarde 5 gebonden is en bijgevolg zijn beide gelijk aan elkaar in het geheugen. In tegenstelling, voor een variabele *cel* gebonden aan (cons 1 2) zal (*eq? cel (cons 1 2)*) print #f omdat ook al zijn hun waarden gelijk, Elke instantie van cons een nieuw koppel in het geheugen aanmaakt. Ze zijn dus, geheugenwijze, niet gelijk. *equal?* in tegenstelling tot *eq?* kijkt of de waarden van de inputs gelijk zijn.

Voor *eq?* en *equal?* beide moeten de inputs waarde exact dezelfde zijn. 5 is bijvoorbeeld niet gelijk aan 5.0 voor deze predicaten, ook al zijn ze wiskundig hetzelfde.

*memq* gaat op zoek naar het eerste element in de gegeven lijst die gelijk is aan het de gegeven waarde. #f wordt teruggeven als het element niet gevonden wordt, anders wordt de lijst vanaf dat element teruggeven. (*memq 'b '(a b c)*) print (b c).

*member* doet exact hetzelfde als *memq* behalve dat de elementen met *equal?* worden vergeleken i.p.v. *eq?*.

*assq* is de equivalent van *memq*, behalve dat deze een key en een associatielijst opneemt en de value van de corresponderende associatie zal teruggeven. Als de koppel niet gevonden wordt zal #f gegeven worden. *assoc* doet exact hetzelfde als *assq* maar gebruikt *equal?* i.p.v. *eq?*.

## 6 Hoofdstuk 6: Hogere Orde Procedures

### Weten wat eersterangsburgers zijn in programmeertalen.

Eersterangsburgers zijn structuren waarop alle operaties dat op andere structuren toegepast kunnen worden kan gebruiken. In Scheme zijn procedures eersterangsburgers; we kunnen dus procedures als argument geven. Bijvoorbeeld (+ 1 (- 2 3)), het verschil procedure staat binnen de optelling als argument.

### Weten wat het verschil is tussen een eerste-orde procedure en een hogereorde procedure.

Een procedure die een datawaarde neemt en een datawaarde teruggeeft noemen we een *eerste orde procedure*.

Een procedure die een procedure neemt of teruggeeft noemen we een *hogere orde procedure*. De exacte orde is irrelevant.

### Weten waarom hogere-orde procedures nuttig & gewenst zijn.

Vaak moeten er een kleine procedures aangemaakt worden om code te versimpelen, omdat er bijvoorbeeld een patroon te verschijn komt (zie procedurele abstractie en hergebruik). Deze kleine procedures kunnen dan als argument gegeven worden.

Als er een fout in het patroon verschijnt, dan moet je deze maar 'e'en keer debuggen i.p.v. alle procedures aan te passen (hergebruik).

Enkele hogere orde procedures worden vaak gebruikt en kom je eventueel van buiten te kennen. Je kan deze een maal opschrijven i.p.v. het idee opnieuw aan te maken. Programmeren gaat sneller

Code die hogere orde gebruikt is korter en gevat. I.p.v. de code te moeten bestuderen om te begrijpen hoe het werkt en wat het doet, kan je met kijkje begrijpen hoe het werkt en dus wat het doet; En dit maakt de code leesbaarder (=declaratief programmeren)

### De structuur van de lambda special form kennen. Het verschil kennen tussen een lambda en een procedure.

Syntax
(lambda (< naam > <sub>1</sub> < naam > <sub>2</sub> ... < naam > <sub>n</sub> ) )
Naamloze procedure Niet gebonden aan omgeving

Een lambda procedure is niet gebonden aan de omgeving. Na dat ze geëvalueerd wordt gaat deze meteen naar de garbage collector. Dit is in tegenstelling tot een procedure die wel gedefinieerd is binnen de omgeving met een naam.

Dit is een nuttig iets die het vervuilen van de omgeving met interne hogere orde procedures vermeid door ze anoniem te maken.

### Weten dat er eigenlijk maar 1 define bestaat en dat de eerste vorm de oer-vorm is.

(define < naam > (lambda(< parameters > ) ) ) = (define(< naam > (< parameters > ) ) )

Define definieert datastructuur op de omgeving. De tweede vorm van define is echt de eerste vorm waarbij lambda niet te zien is voor leesbaarheid. Syntax die te herleiden valt to een samenstelling van meer fundamentele constructies in een programmeertaal noemen we *syntactische suiker*.

### Het verschil tussen let, let\* en letrec. De algemene vormen kennen van deze 3 speciale forms.

Syntax
(let ((< naam > <sub>1</sub> < uitdrukking <sub>1</sub> > < naam > <sub>2</sub> < uitdrukking <sub>2</sub> > ... < naam > <sub>n</sub> < uitdrukking <sub>n</sub> >)) < body <sub>1</sub> > ... < body <sub>m</sub> >)

<p>Tijdelijke uitbreiding van oproepende omgeving voor <math>\langle \text{naam}_i \rangle</math>  <math>\langle \text{uitdrukking}_i \rangle</math> wordt in oproepende omgeving geëvalueerd  <math>\langle \text{naam}_i \rangle</math> wordt gebonden aan <math>\langle \text{uitdrukking}_i \rangle</math>  Uitdrukkingen in de body worden in tijdelijke omgeving uitgevoerd  Waarde van <math>\langle \text{body}_m \rangle</math> wordt teruggeven  De hele body gaat naar garbage collector</p>
<p>(let*     ) met     dezelfde syntax als let</p>
<p>Tijdelijke omgeving aangemaakt voor binding <math>\langle \text{naam} \rangle_1</math> en <math>\langle \text{uitdrukking} \rangle_1</math>  Tijdelijke omgeving aangemaakt aan eerste tijdelijke omgeving voor volgende binding  Zelfde proces voor <math>\langle \text{naam} \rangle_i</math> let* volgt dezelfde regels als let vanaf dit punt</p>
<p>(letrec     ) met     dezelfde syntax als let en let*</p>
<p>De zelfde regels als voor let* gelden  Mutuele (recursieve) afhankelijkheden zijn toegestaan  let maakt het mogelijk om</p>

### De algemene vorm van desugaring van let en let\* kunnen uitleggen.

Desugared
<pre>(let  (((&lt; naam &gt;<sub>1</sub> &lt; uitdrukking<sub>1</sub> &gt; &lt;       naam &gt;<sub>2</sub> &lt; uitdrukking<sub>2</sub> &gt;       ...       &lt; naam &gt;<sub>n</sub> &lt; uitdrukking<sub>n</sub> &gt;))       &lt; body<sub>1</sub> &gt;       ...       &lt; body<sub>m</sub> &gt;)</pre>
<pre>((lambda(&lt; naam &gt;<sub>1</sub> &lt; naam &gt;<sub>2</sub> ... &lt; naam &gt;<sub>n</sub>)   &lt; uitdrukking &gt;<sub>1</sub>   ...   &lt; uitdrukking &gt;<sub>m</sub>)   &lt; uitdrukking &gt;<sub>1</sub> ... &lt; uitdrukking &gt;<sub>n</sub>)</pre>
<p>(let*     ) met     dezelfde syntax als let</p>
<pre>((lambda(&lt; naam &gt;<sub>1</sub>) ((lambda(&lt;   naam &gt;<sub>1</sub>)   ...   ((lambda(&lt; naam &gt;<sub>n</sub>)     &lt; uitdrukking &gt;<sub>1</sub>     ...     &lt; uitdrukking &gt;<sub>m</sub>)     &lt; uitdrukking &gt;<sub>n</sub>)   ...))   &lt; uitdrukkingen &gt;<sub>2</sub>)   &lt; uitdrukkingen &gt;<sub>1</sub>)</pre>

### De half-intervalmethode kunnen uitleggen. Het verband met hogere-orde programmeren kunnen uitleggen.

Stel we zoeken een punt in een bepaalde interval. De half-interval methode houdt in dat interval te halveren en een test uit te voeren om te zien of het punt binnen de gehalveerde interval ligt. Als dat niet het geval is, dan ligt het punt in de andere helft en halveren we dat interval, en zo voort tot dat we dicht genoeg bij het punt zijn om deze terug te kunnen geven. Dit is een *benaderingsmethode*.

Gegeven een (wiskundige) functie die het punt definieert; deze functie moet dan gegeven worden aan de procedure die deze benaderingsmethode implementeert. Dit maakt van de functie 'en de benaderingsmethode een hogere orde procedure.

## Fixpunten kunnen uitrekenen. Het verband met hogere-orde programmeren kunnen uitleggen.

Een *fixpunt* of *dekpunt* van een functie is een waarde  $x$  waarvoor  $f(x) = x$ .

Je kan een *dekpunt* benaderen door de functie vertrekend van een startwaarde  $x_0$ , blijven toepassen op haar eigen output tot het resultaat geen significant verschil meer heeft met het vorig resultaat:  $f(f(f(\dots(f(x_0)\dots))))$ . We moeten de procedure die de functie bepaalt als argument geven aan deze benaderingsmethode, makend van beide procedures hogere orde procedures.

## Het algoritme van Newton kunnen formuleren als fixpunt vraagstuk. De *damping* verbetering er vervolgens kunnen aan toevoegen.

$$y = \sqrt{x} \implies y^2 = x \iff y = \frac{x}{y} \quad (2)$$

We zoeken dus een *dekpunt* van de functie  $f(y) = \frac{x}{y}$ . Als we starten met een punt  $y_0$ :

$$y_1 = \frac{x}{y_0} \implies y_2 = \frac{x}{y_1} = \frac{x}{\frac{x}{y_0}} = y_0$$

De output waarde zal oneindig tussen twee opeenvolgende benaderingen oscilleren. Om dit te vermijden gebruiken we *damping*, een standaard techniek in de numerieke wiskunde. I.p.v. de volgende waarde te gebruiken, nemen we het gemiddelde van die volgende waarde en de huidige waarde.

$$\implies y_1^{dempt} = \frac{y_0 + y_1}{2} = \frac{y_0 + \frac{x}{y_0}}{2} \quad (3)$$

$$f(y) = \frac{y + \frac{x}{y}}{2} \quad (4)$$

We zoeken dus een *dekpunt* van de functie (4).

## Weten hoe je een benaderingsmethode generisch kan maken door ze als procedure terug te geven uit een andere procedure.

```
(define (average-damp f)
  (lambda (y)
    (average y (f y))))
```

```
(define (sqrt5x)
  (fixpoint (average-damp (lambda (y) (/ x y)))))
```

## De implementatie en werking van map kennen.

```
(map "procedure" 'list)
```

Map gaat een voor een door de lijst en past de "procedure" toe op ieder element

## De implementatie en werking van filter kennen.

```
(define (filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst)) (cons (car lst) (filter pred (cdr lst))))
        (else
         (filter pred (cdr lst)))))
```

## Kunnen uitleggen wet een callback procedure is en wat het verband is met hogere-orde procedures.

Je geeft een call-back functie mee die zal worden opgeroepen door die hogere order procedure.  
bv (animate (lambda (time) ...image...))



## 7 Hoofdstuk 7: Imperatief programmeren

### Algemene vorm van de set! kennen.

(set! <naam> <uitdrukking>)

Evalueren door in de omgeving <uitdrukking> te evalueren, dan <naam> opzoeken in die omgeving en de binding te vervangen door de gevalueerde <uitdrukking>

### Algemene vorm van begin kennen.

(begin <uitdrukking> <uitdrukking>)

Wordt uitgevoerd door alle <uitdrukking> een na een te evalueren. De waarde van ene begin expressie is de waarde van haar laatste deexpressie

### De impliciete begins kennen en begrijpen.

Let, cond, define, do

**cond, define, do**

Plaatsen waar we tot nu toe 1 expressie hadden staan

The diagram illustrates the execution of four Scheme constructs: `let`, `define`, `cond`, and `do`. Each construct is shown in a box with a small icon of a person running, indicating the flow of execution. A speech bubble points to the first argument of each construct, stating 'Plaatsen waar we tot nu toe 1 expressie hadden staan' (Places where we have had 1 expression so far). The `let` snippet shows `(let ((x 1) (y 2)) (display x) (newline) (display y) (newline))`. The `define` snippet shows `(define (f) (display 1) (newline) (display 2) (newline))` followed by `(f)`. The `cond` snippet shows `(cond ((= 1 1) (display 1) (newline) (display 2) (newline)) (else (display 3) (newline) (display 4) (newline)))`. The `do` snippet shows `(do ((i 0 (+ i 1))) ((= i 10) (display 1) (newline) (display 2) (newline)))`. Execution flow arrows show the sequence of operations for each construct.

### Kunnen uitleggen wat functioneel programmeren en imperatief programmeren is.

Functioneel programmeren is functies schrijven die expressies 'uitrekenen'

Imperatief programmeren is rijen van instructies oplijsten die de computer een na een uitvoert, bevel om iets te doen + imperatief word vaak ook destructief genoemd omdat je vaak bv een lijst aan past maar niet de originele lijst bewaard.

### Weten wat Turing equivalentie betekent.

De taal families van programmeertalen heet programmeerparadigma's. In theorie zijn deze exact even krachtig Alles wat je met de ene kan programmeren kan je ook met de andere dit noemt Turing-equivalentie sommige talen zijn beter voor specifieke talen dit heet expressiviteit

### Kunnen uitleggen "welke x" door set! gemodificeerd zal worden als er verschillende x'en in de scope van een programma staan.

Zeer goed het punt van set! Begrijpen (atoma schriftjes + lijntjes model)

### Set-car! en set-cdr! Kennen en de werking uitleggen.

Set-car! en Set-cdr! maken geen nieuw paar, het vervangt dus de car of cdr door de gegeven waarde

### Kunnen uitleggen wat aliasing is.

Als op een bepaald moment 2 verschillende namen wijzen naar hetzelfde paar: aliases

## **Verschil kunnen uitleggen tussen append en append! En tussen replace en replace!**

Append past de lijst niet aan append! (maakt en nieuwe) past de gevraagde lijst aan (geen nieuwe) zelfde met replace de originele lijst bestaat niet meer

## **Kunnen uitleggen waarom add! en delete! niet zomaar op gewone naakte lijsten geschreven kunnen worden (c.f. eerste paar).**

Ze kunnen niet het eerste element aanpassen van de lijsten dus moet je ze een header geven.

## **Verschil tussen call-by-value en call-by-reference kunnen uitleggen.**

Call-by-value: Een parameter wordt gebonden aan een waarde

Call-by-reference: een parameter wordt gebonden aan een externe variable (ouderwets bv Pascal)

## **De vector-primitieven kennen, hun werking begrijpen en kunnen uitleggen waarom vectoren soms nodig zijn (i.p.v. lijsten).**

Een vector van grootte n bestaat uit n geheugenlocaties die in de computer naast mekaar opgeslagen worden je kan hier dus direct de value uit bepaalde vakjes halen

## **Weten hoe je lijsten omzet naar vectoren. Wat is de moeilijkheid en hoe wordt het opgelost?**

```
(define (from-list-to-vector lst)
  (define (calc-length-and-fill l idx)
    (if (null? l)
        (make-vector idx)
        (let ((v (calc-length-and-fill (cdr l) (+ idx 1))))
          (vector-set! v idx (car l))
          v))))
  (calc-length-and-fill lst 0))
```

Het lastige hier is de grote van de vector berekenen (de lengte van de lijst)

## **Weten wat en bitvector is en weten wat/hoe de operaties op bitvectoren geïmplementeerd worden.**

En bitvector is een vector gevuld met bits dus 1 of 0 om getallen te maken. Je kunt dit omzetten naar getallen of omgekeerd.

# **8 Hoofdstuk 8: ADT**

## **Weten welke manieren er bestaan om name clashes tussen import clauses op te lossen en weten hoe je die moet gebruiken.**

(only <clause> <name1> ... <namex>) => Alleen bepaalde procedures

(prefix <clause> <name>) => <name> plakken voor alle namen

(except <clause> <name1> ... <namex>) => alles behalve bepaalde procedures

(rename <clause> (<name1> <name'1>) (<name2> <name'2>) (<name3> <name'3>)) => hernoem sommige procedures

## **Weten wat een ADT is, waarom ADTs nodig zijn en wat de voordelen zijn van het denken in termen van ADTs.**

Een abstract data type, het maken van code en omzetten in een library voor gemakkelijker gebruik in grote projecten. Andere mensen kunnen deze ADTs ook gebruiken als je ze publiek stelt

### **Weten wat bedoeld wordt met ‘implementatie van een ADT’ en ‘representatie van het datatype’.**

De implementator kiest zelf voor een representatie (die hij zo goed mogelijk probeert weg te stopeen door bv libraries) en een implementatie

### **Weten dat we verzamelingen zowel geordend als ongeordend kunnen representeren en weten wat de voor- en nadelen zijn. Begrijpen dat veranderingen van representatie geen invloed zou mogen hebben op de functionaliteit van de operaties.**

In gesorteerde lijsten kun je bepaalde waarden veel sneller vinden + het resultaat zal vaak ook gesorteerd zijn

### **Weten dat we een matrix kunnen voorstellen als geneste vectoren en begrijpen hoe 2-dimensionale indexering werkt als combinaties van vectorref en vector-set!**

Slide 28 //

### **Weten wat record types zijn en wat de voordelen zijn. De algemene vorm van define-record-type kennen. Weten dat mutatoren optioneel zijn.**

Een record is net zoals een steekkaart met als een “veld”(key) “waarde”(value)

Mutatoren zijn optioneel want zij en accesoren worden automatisch gegenereerd door r7rs

```
(define-record-type <naam_type>
  (<naam_constructor> <naam_1> <naam_2> ... <naam_n>)
  <naam_test>
  (<naam_1> <naam_1a> <naam_1m>)
  (<naam_2> <naam_2a> <naam_2m>)
  ...
  (<naam_n> <naam_na> <naam_nm>))
```

Mutatoren zijn facultatief

### **Weten wat een dictionary is en wat de belangrijkste operaties zijn.**

ADT table is een geminiaturiseerde versie van ADT dictionary

Make-table

Table?

Insert!

Lookup

Number-of-elements

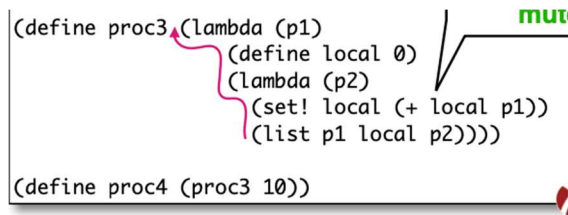
### **Weten wat testing is en wat een testing framework voor je kan doen.**

Grote programma's bestaan uit vele ADTs die samenkomen in een “hoofdprogramma”. Om complexiteit te beheersen gaat men al deze “units” apart één voor één testen. If testen schrijven veel werk... Dus hier kun je een testing framework gebruiken (een test library) die iemand anders ontwikkeld heeft

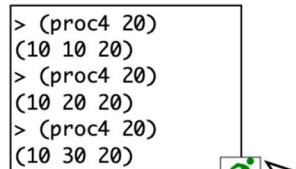
## 9 Hoofdstuk 9

Een voorbeeld kunnen geven van hoe 2 geneste procedures dezelfde omliggende variabelen kunnen lezen en schrijven.

```
(define proc3 (lambda (p1)
  (define local 0)
  (lambda (p2)
    (set! local (+ local p1))
    (list p1 local p2))))
(define proc4 (proc3 10))
```



```
> (proc4 20)
(10 10 20)
> (proc4 20)
(10 20 20)
> (proc4 20)
(10 30 20)
```



**Kunnen uitleggen (a.d.h.v. een vb.) waarom noch het substitutiemodel noch het atomaschriftmodel voldoende zijn om het gedrag van Scheme procedures helemaal te verklaren.**

#1 Atomaschriftjesmodel faalt -> tijdelijke omgevingsuitbreidingen mag je nie zomaar Gcen (als je een blad toevoegt en hierop krijg je een procedure als output dan word dit GC en werkt het niet meer

#2 Substitutiemodel faalt -> Zeer inefficiënt voor geneste procedures (zonder set!) dus het werkt sowieso niet zonder set!

### Wat is een omgeving?

Een omgeving  $O$  is een lijst van frames. Elke frame is een lijst van bindingen van namen aan waarden. Elke frame (behalve de frame die overeenkomt met de globale omgeving) heeft een parent frame. Een

### Hoe wordt een variabele opgezocht om uit te lezen en/of aan te passen met set!

Variabele wordt opgezocht in een omgeving  $O$  door te vertrekken in de laagste frame en dan naar boven te lopen tot de frame van de globale omgeving.

### De 2 basisregels kennen en begrijpen om procedures te evalueren en op te roepen.

#1 Als een (anonieme) procedure aangemaakt wordt, onthoudt scheme de parameters, de body + een verwijzing naar de omgeving waarin die aanmaak gebeurde.

#2 Als een procedure wordt opgeroepen maakt scheme een tijdelijke uitbreiding van haar omgeving van definitie. In deze uitbreiding worden parameter bindingen + locale definities gezet. De body wordt geëvalueerd in deze tijdelijke uitbreiding  $O$ .

### Weten wat een closure is.

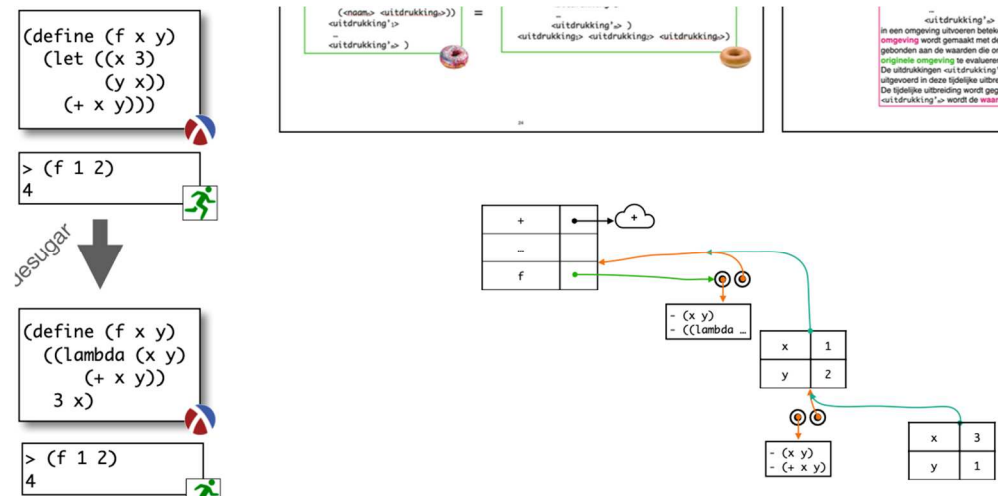
Een procedure-object heet technisch een closure. Het is de "afsluiting" van een lambda expressie zodat alle vrije variabelen toch gebonden worden aan haar body.

### Weten waar de verschillende soorten pijlen in een omgevingsdiagram vandaan komen.

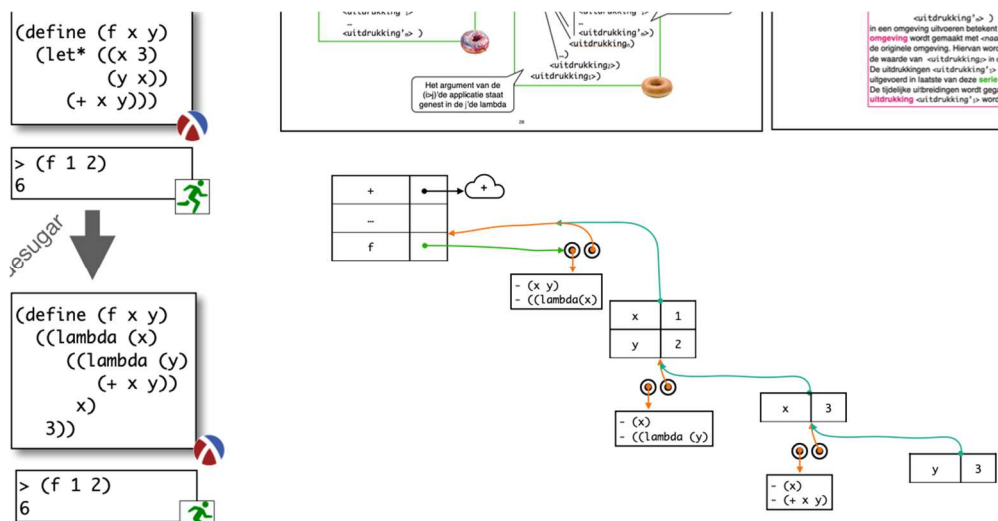
Het goed begrijpen van omgevingsdiagrammen

# Werking van let, let\* en letrec kunnen uitleggen a.d.h.v. het omgevingsdiagram van hun ontsuikering.

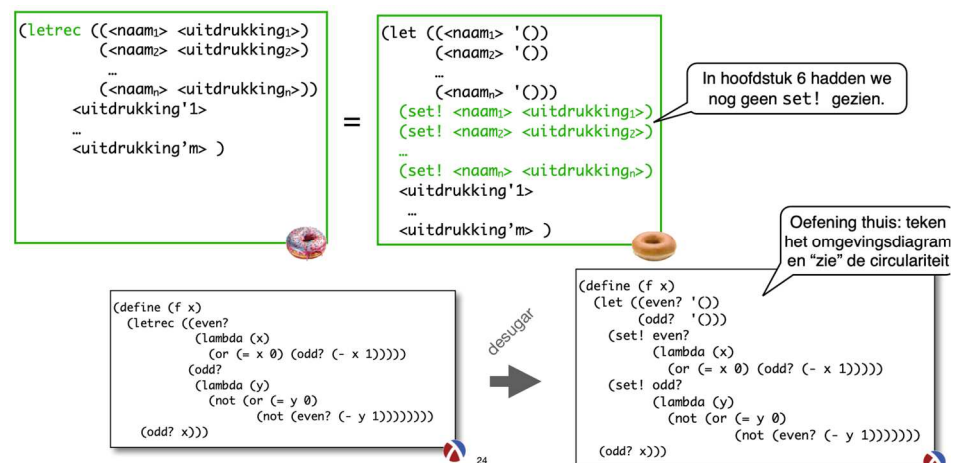
Let:



Let\*:



Letrec:



### **Weten wat een wrapper is.**

Een procedure die en andere procedure omhelst en als je de omhelende procedure oproept krijg word de originele ook opgeroepen (vaak gebruikt voor errors of bij te houden hoeveel keer een functie gecalld word)

### **Kunnen uitleggen wat memoization is, hoe de techniek werkt en wat de voordelen zijn.**

Als een proces overlappende subproblemen heeft zoals (fib n) dan kun je functies hun werk laten onthouden snelheidswinst op grote hoeveelheden is zeer groot

## **10 Hoofdstuk 10**

### **Weten wat object-gericht programmeren is.**

Programmas die berichten naar objecten sturen. De berichten worden door methoden afgehandeld.

### **Weten wat encapsulatie is.**

Een object encapsuleert zijn interne state ((viviane 'deposit) 100)

### **Kunnen uitleggen dat elk object een closure is met lokale state.**

Elk object is een nieuwe closure die ontstaan is door een andere oproep van make-account die state van ieder object is dus gescheiden

### **Kunnen uitleggen wat late-binding polymorfisme is.**

Polymorfe expressies: Kunnen iets totaal anders doen bij elke evaluatie

OOP leidt tot late-binding polymorfisme omdat het pas tijdens de runtiem duidelijk wordt welke procedure nu precies uitgevoerd zal worden, in gewone procedurele code zie je statisch welke procedures uitgevoerd zullen worden.

Slide 13 heel goed leren

### **Weten wat het verschil is tussen een bericht en een methode.**

Een bericht in OOP is '(bericht) een methode is code om op dezelfde berichten te reageren

### **Weten wat de interface van een object is.**

De code van bepaalde objecten is bijna identiek buiten op een paar woorden na

### **Het verschil tussen monomorfe en polymorfe expressies.**

In gewone (monomorfe) code zie je statisch welke procedures uitgevoerd zullen worden. Polymorfe expressies kunnen iets totaal anders doen bij elke evaluatie

### **De verschillen tussen procedurele en OOP-gebaseerde ADT implementaties kennen.**

Procedureel	OOP
Programmeerstijl = procedures oproepen	Programmeerstijl = berichten sturen
Datavoorstelling: vectoren, records, paren	Datavoorstelling: dispatchers (objecten)

Data hiding; m.b.v libraries	Data hiding: objecten encapsuleren
Geen automatische dispatching	Automatische dispatching (polymorf)
Goede ondersteuning in basisscheme	Beperkte ondersteuning in basisscheme

### **Kunnen uitleggen waarom single-dispatch objecten soms geen goede programmeertechniek is.**

Als er eisen zijn van 2 of meer argumenten en geen van allen als geprivilegieerde ontvanger van de boodschappen aangewen kan worden dan werkt single-dispatch niet.

### **Weten wat getagde data is.**

Een naam geven aan data om dit bv te kunnen hergebruiken zodat er geen data duplicatie is.

### **Het multiple-dispatch mechanisme kunnen uitleggen (versie 1.0) zonder coercions**

Een assoc lijst (key value) in een assoc lijst steken kijk slide 32 - 36

### **Weten wat coercions zijn en waarom ze nodig zijn bij multiple-dispatch.**

Het dwingen van een getal om dit te veranderen in een ander getal bv van een integer een breuk te maken door /1 te doen

### **Het multiple-dispatch mechanisme kunnen uitleggen (versie 2.0) met coercions.**

Slide 42

## **11 Hoofdstuk 11**

### **Begrijpen dat een geneste lijst als boom bekeken kan worden.**

Een boom is: een atoom of een lijst met n subbomen (dus een lijst met daarin lijsten)

### **Verschillende definities van "boom" kennen en begrijpen waarom niet elke geneste lijst aan die definitie(s) voldoet.**

Een boom is: <ul style="list-style-type: none"> <li>• ofwel een atoom.</li> <li>• ofwel een lijst met n subbomen.</li> </ul>
Een Famillieboom bestaat uit een knoop smaen met $n \geq 0$ familliebomen (genaamd subbomen)
Een Huffmanboom bestaat uit bladeren van interne knopen <ul style="list-style-type: none"> <li>• Ieder blad bevat een symbool en gewicht</li> <li>• Iedere interne knoop heeft 2 kinderen. Het gewicht van de knoop is d som van de gewichten van de kinderen</li> </ul>
Een expressieboom bestaat uit bladeren van interne knopen: <ul style="list-style-type: none"> <li>• Een blad bestaat uit een symbool of een getal</li> </ul>

- Iedere interen knoop bevat een operator en een aantal kinderen dat gelijk is aan de ariteit van de operator

### **Weten wat een familieboom is en wanneer familieboomen toepasbaar zijn om een situatie te modelleren.**

Familieboomen worden gebruikt om allerlei hiërarchische vragen te beantwoorden (en letterlijke familieboom) + het goed begrijpen tot slide 20

### **Weten wat een binaire voorstelling van een alfabet met vaste en variabele lengte is en weten waarom dat laatste efficiënter kan zijn.**

Vaste lengte is wanneer ieder letter in het alfabet wordt gelinkt aan een binaire waarde dan kun je dit allemaal achter elkaar zetten. Bij variabele lengte krijgen letters die vaker voorkomen een lagere bitwaarde zodat je woorden of stukken tekst korter kunt schrijven.

### **De prefix-eigenschap kennen en kunnen duiden. ??**

Een voorstelling van variabele lengte moet voldoen aan de prefix-eigenschap geen enkele letter heeft een voorstelling die een prefix is van de voorstelling van een andere letter.

### **Weten dat frequentie-studies van alfabetten de leidraad zijn voor een voorstelling met variabele lengte.**

Dus voor iedere taal wordt er gemeten hoeveel en letter ongeveer voorkomt en dan wordt hieraan een percent gegeven.

### **Weten hoe een Huffman-boom eruitziet, wordt voorgesteld en opgebouwd wordt m.b.v. het gezien algoritme.**

Slide 27 - 33

### **Weten wat een expressieboom is. De expressie boom kunnen tekenen voor eenvoudige rekenkundige expressies.**

Slide 38

### **Weten hoe expressieboomen voorgesteld worden in Scheme.**

Neemt een expressie en dan de bladeren eronder zijn de symbolen of getallen slide 38

### **Weten wat generatieve boomrecursie is.**

In plaats van een boom af te lopen met boomrecursie gaan we een boom aflopen die we zelf aan het genereren zijn

### **Kunnen uitleggen hoe de yield van een gegenereerde boom (tot zekere diepte) overeen komt met instructies voor de turtle.**

De "yield" hier zijn de bladeren van de boom slide 46//

## **12 Hoofdstuk 12**

**Begrijpen dat(recursieve) procedures op allerlei input gezien kunnen worden als een rij van acties die die input geleidelijk aan transformeert, filtert en combineert.**

???



**Weten dat een (eindige) stroom uit een generator, verwerkers en accumulator bestaat.**

???

**Het ADT stream kennen. De implementatie met eindige lijsten kennen.**

???

**De implementatie van stream-append kennen.**

???

**Beide implementaties van stream-values kunnen uitleggen in en buiten het ADT.**

???

**Het verschil inzien tussen lijsten als concept in code en lijsten als implementatietechnologie van code die conceptueel over stromen gaat.**

???

**De implementatie van stream-map kennen en stream-map kunnen oproepen met de juiste argumenten.**

???

**Begrijpen dat sommige argumenten van de stroom operaties “configuratieargumenten” zijn.**

???

**De implementatie van stream-filter kennen en stream-filter kunnen oproepen met de juiste argumenten.**

???

**De implementatie van stream-accumulate kennen en stream-accumulate kunnen oproepen met de juiste argumenten.**

???

**De implementatie van stream-for-each kennen en stream-for-each kunnen oproepen met de juiste argumenten.**

???

**Kunnen uitleggen da code om (grote) files te verwerken kan geformuleerd worden als stroomgebaseerde code.**

???

**Weten wat een geneste stroom is en dat je stream-flatten kan gebruiken om geneste stromen tot stroom “plat te kloppen”. De implementatie van stream-flatten kennen.**

???

De implementatie van matrices als geneste stromen kunnen uitleggen en de code kunnen toelichten. Ook zo voor de geziene operatoren uit de lineaire algebra.

???

delay en force begrijpen. Hun algemene vorm kennen en begrijpen wat er gebeurt als je delay en force met set! gaat combineren.

???

De 2de implementatie van het stream ADT kennen en de werking van tail te begrijpen in REPL.

???

Weten wat een impliciet gedefinieerde stroom is en begrijpen hoe het mogelijk is dat je iets definieert (met define) dat ogenschijnlijk meteen naar zichzelf refereert.

???

De Zeef van Erastothenes kunnen uitleggen, zowel met woorden en voorbeelden als met code.

???

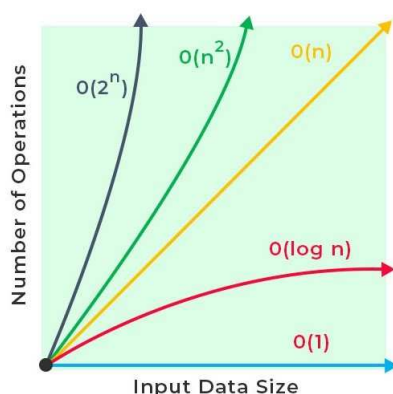
Het probleem van geneste oneindige stromen kunnen uitleggen en met een opgebouwde redenering tot de implementatie van stream-flatten/interleaved kunnen komen.

???

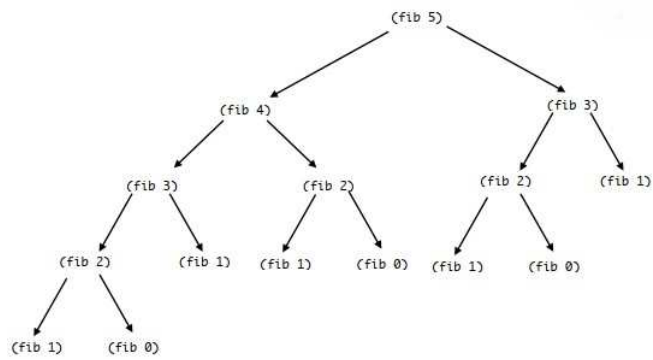
Begrijpen dat real-time applicaties (zoals MIVB) geschreven kunnen worden als stroomverwerkers die hun resultaat met een stroom accumulator laten zien.

???

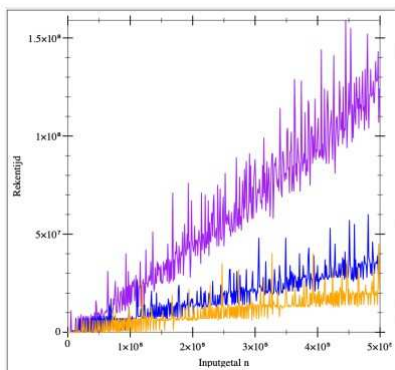
## 13 Hulpvolle figuren



Figuur 3: Tijd complexiteit



Figuur 4: Boomrecursie



Figuur 5: Recursie(paars), Iteratie(blauw), do-loop (geel)

## 14 Mogelijke vragen + extras

<https://assq.notion.site/Tentamen-Structuur-van-Computerprogramma-s-1-2024-13674775a13980bcb90ecc208d1fc9f5>