Wat moet ik kennen / kunnen uitleggen?		Wat moet ik kunnen (doen)?
Hoofdstuk 1		
Wat is een compiler en een interpreter/evaluator ?		DrRacket downloaden en opstarten. DrRacket in R5RS zetten en eenvoudige berekeningen kunnen doen in de REPL.
Wat zijn S-expressies ?		
Hoe werken de 3 verschillende notaties voor rekenkunde (prefix, infix, postfix) ?		
REPL terminologie begrijpen: read, eval, print. Verschil tussen expressies en waarden begrijpen.		
Het verschil begrijpen tussen syntaxfouten en runtime fouten.		
Hoofdstuk 2		
Het concept 'nesting' kunnen uitleggen in de context van geneste expressies.		Willekeurig diepe berekeningen uit de wiskunde omzetten in geneste prefix expressies.
Begrijpen hoe samengestelde geneste uitdrukkingen (mechanisch) geevalueerd worden.		
Terminologie over variabelen, bindingen en omgeving kunnen uitleggen.		Variabelen kunnen aanmaken met samengestelde expresies als bindings-expressies. Variabelen kunnen gebruiken in geneste expressies.
Algemene formaat van de eerste vorm van define kennen.		
Weten wat ariteit, unair, binair,... betekent.		Een voorbeeld van elk van hen kunnen geven.
Weten wat booleans zijn + werking van and/or/not kennen.		Waarheid kunnen uitrekenen met booleaanse expressies.
Weten wat relationele operatoren zijn.		Vergelijkingen kunnen maken van 2 rekenkundige uitdrukkingen.
Weten wat een predicaat is.		
Weten wat types zijn. De type-predicaten kennen.		
Tweede vorm van define kennen + kunnen onderscheiden van de eerste vorm.		Een eenvoudige wiskundige functie kunnen definieren in Scheme.
Terminologie procedures kennen: argumenten, ...		
Verschil tussen substitutiemodel en omgevingsmodel kennen om functie-oproepen te verklaren.		Het substitutiemodel kunnen toepassen op uitdrukkingen met procedure-oproepen van zelfgeschreven procedures erin.
Wat is procedurele abstractie? Waarom is het nuttig/nodig?		Het omgevingsmodel kunnen toepassen op uitdrukkingen met procedure-oproepen van zelfgeschreven procedures erin.
Wat is hergebruik? Waarom is het nuttig/nodig?		De tracer kunnen aanzetten voor zelfgeschreven procedures.
Verschil tussen Racket en R5Rs begrijpen.		Prentjes aan mekaar plakken + zelf prentjes-operatoren opzoeken in de Racket documentatie.
Begrijpen dat rekenen met getallen, booleans, strings en prentjes in Scheme exact dezelfde principes volgt.		
Hoofdstuk 3		
Werking van if en cond begrijpen.		Wiskundige functies met meerdere gevallen omzetten ("accolades") naar een Scheme definitie met cond of if erin.
Subtiel verschil met of zonder "else" begrijpen.		Lokale variabelen en procedures kunnen gebruiken om procedures leesbaarder te maken.
Lokaliteit van lokale procedures en formele parameters begrijpen.		Een beschrijving van een eenvoudig herhalingsproces (zoals de wortel-procedure van Newton) om kunnen zetten naar een zelf-oproepende procedure.
Lokaliteit kunnen uitleggen in termen van het omgevingsmodel.		
Tracing van recursieve procedures begrijpen.		
Algoritme voor vierkantswortel kennen.		
Algoritme voor verdubbeltijd berekenen kennen.		
Aan de hand van het omgevingsmodel kunnen uitleggen hoe een recursieve procedure de verschillende zelf-oproepen uit mekaar kan houden.		
Variabelen in programmatekst kunnen classificeren als vrij of gebonden (statisch).		
Verschil tussen 2 soorten "gebonden" begrijpen.		
Weten wat een blokgestructureerde programmeertaal is. Weten wat scope-niveau's zijn.		
Verschil tussen lexicale en dynamische scoping in programmeertalen kunnen uitleggen.		Herhalingen met eenvoudige display/newline/read interacties kunnen programmeren.

Lexicale scoping kunnen uitleggen aan de hand van het omgevingsmodel.		Voorbeeld over algoritmisch tekenen kunnen uitleggen en kunnen toelichten hoe het voorbeeld zich verhoudt tot het algoritme om vierkantswortels te berekenen.
Hoofdstuk 4		
Tracing begrijpen van recursieve procedures.		Inductieve definities kunnen omzetten naar een recursieve Scheme procedure.
Weten wat een profile is. Begrijpen wat de X-as en de Y-as voorstellen.		Aan de trace kunnen zien of een proces recursief of iteratief is.
Weten waarom profiles zoveel aberraties hebben.		
Weten wat lineaire recursie is.		
Het verschil begrijpen tussen recursieve procedures die een recursief en iteratief process genereren.		Aan de code kunnen zien of een recursieve procedure een recursief dan wel iteratief process genereert.
Weten wat backtracking is. Weten waarom backtracking niet nodig is in een iteratief proces.		Kunnen zeggen of een recursieve oproep wel of niet in staartpositie staat.
Weten welke (iteratief/recursief) doorgaans betere profiles oplevert.		
Weten wat een accumulator is in een iteratief proces.		Een lineair proces geschreven met constructieve recursie kunnen omvormen naar staartrecursie, en omgekeerd.
Weten wat boomrecursie is.		
Weten wat een exponentieel proces is.		
Weten wat een recursieboom is.		
Weten wat een logaritmisch process is.		
Weten wat sublineariteit betekent.		
Het verschil tussen best-case en worst-case input begrijpen.		
Weten wat iteratieve code is. De algemene vorm van een do kennen.		Een eenvoudige iteratieve procedure met do kunnen opschrijven.
Weten wat het verband is tussen de profiles van een recursief process, een iteratief process (op basis van recursieve procedure) en een iteratief proces (op basis van een iteratieve procedure).		Een recursieve procedure (die iteratief proces genereert) om kunnen zetten naar een iteratieve procedure.
Helder het verband tussen procedures en processen kunnen toelichten met de juiste woordenschat.		Voorbeelden van recursief gegenereerde graphics kunnen toelichten a.d.h.v. de code.
Weten wat indirecte/mutuele recursie is.		Een mutueel recursief process kunnen schrijven gegeven een definitie.
Hoofdstuk 5		
De "wetten" van cons/car/cdr kennen.		Box-pointer diagrams kunnen tekenen van arbitraire lijsten / combinaties van cons
Weten wat een garbage collector is.		
Weten wat een datastructuur is.		
Weten wat een lijst is.		Op het zicht van een cluster van paren zien of het wel of geen lijst is.
Ingebouwde basisprocedures op lijsten kennen.		Eenvoudige procedures kunnen schrijven op lijsten die een iteratief of recursief proces opleveren.
		Correct cons / append gebruiken in recursieve procedures op lijsten
Weten waarom length en list-ref traag zijn. Begrijpen hoe lijsten (of paren in het algemeen) in het computergeheugen naar mekaar kunnen "wijzen".		Zien of een recursieve procedure op lijsten een recursie of iteratief process gaat opleveren. Op basis hiervan "de andere" kunnen schrijven.
Implementatie kennen van de basisprocedures: sum/square-all, reverse, flatten, member, length, append.		
Quote begrijpen voor alle geziene types.		
Weten wat een symbol is.		De juiste gelijkheidsprocedure toepassen waar nodig.
Weten wat een associatielijst is + assoc.		Zelf procedures op associatielijsten kunnen schrijven.
		Code van de GPS-kaarten kunnen toelichten.
		Code van de MiniLogo evaluator kunnen toelichten. Begrijpen hoe de toestand (tekening+turtle) iteratief door het proces "reist".
Hoofdstuk 6		
Weten wat eersterangsburgers zijn in programmeertalen.		Een hogere-orde procedure kunnen extraheren op basis van 2 or meerdere zeer op mekaar gelijkende procedures.

Weten wat het verschil is tussen een eerste-orde procedure en een hogere-orde procedure.		Hogere-orde procedures correct kunnen oproepen door inpluggen van de juiste procedures. Zowel met genaamde als met anonieme procedures.
Weten waarom hogere-orde procedures nuttig & gewenst zijn.		
De structuur van de lambda special form kennen. Het verschil kennen tussen een lambda en een procedure.		
Weten dat er eigenlijk maar 1 define bestaat en dat de eerste vorm eigenlijk de oer-vorm is.		Defines "van soort 2" om kunnen zetten naar defines "van soort 1" (zie hierboven)
		Het substitutiemodel kunnen toepassen op teruggegeven procedures
Het verschil kennen tussen let, let* en letrec. De algemene vormen kennen van deze 3 special forms.		let en let* kunnen gebruiken om subexpressies te extraheren.
De algemene vorm van desugaring van let en let* kunnen uitleggen.		Zelf eenvoudige let en let* expressies kunnen desugaren.
De half-intervalmethode kunnen uitleggen. Het verband met hogere-orde programmeren kunnen uitleggen.		
Fixpunten kunnen uitgerekenen. Het verband met hogere-orde programmeren kunnen uitleggen.		
Het algoritme van Newton kunnen formuleren als fixpunt vraagstuk. De demping verbetering er vervolgens kunnen aan toevoegen.		
Weten hoe je een benaderingsmethode generisch kan maken door ze als procedure terug te geven uit een andere procedure.		
De implementatie en werking van map kennen.		Zelf eenvoudige hogere-orde procedures kunnen schrijven op lijsten.
De implementatie en werking van accumulate kennen.		De deliberatiecode kunnen uitleggen.
De implementatie en werking van filter kennen.		De fixpunten(!) procedure van de logistieke functie kunnen uitleggen
Kunnen uitleggen wat een callback procedure is en wat het verband is met hogere-orde procedures.		
Hoofdstuk 7		
Algemene vorm van de set! kennen.		Eenvoudige groepen van display/newline maken met begin
Algemen vorm van begin kennen.		
De implicieite begins kennen en begrijpen.		
Kunnen uitleggen wat functioneel programmeren en imperatief programmeren is.		Staartrecursieve (i.e. iteratieve) procedures kunnen omzetten naar do en set!
Weten wat Turing-equivalentie betekent.		Zowel voor getallen als voor lijsten.
Kunnen uitleggen "welke x" door set! gemodifieerd zal worden als er verschillende x'en in de scope van een programma staan.		
set-car! en set-cdr! kennen en de werking utileggen.		Bestaande paren kunnen veranderen met set-car! en set-cdr!
Kunnen uitleggen wat aliassing is.		Aliasen in een lijst kunnen detecteren
		for-each kunnen gebruiken op lijsten
Verschil kunnen uitleggen tussen append en append! en tussen replace en replace!		Circulaire lijsten kunnen maken + aflopen met recursie en iteratie
Kunnen uitleggen waarom add! en delete! niet zomaar op gewone naakte lijsten geschreven kunnen worden (c.f. eerste paar).		cycles kunnen detecteren
Verschil tussen call-by-value en call-by-reference kunnen uitleggen.		
Kunnen uitleggen wat een headed list is en waarom het nuttig is.		add! en delete! of headed lists kunnen schrijven.
De vector-primitieven kennen, hun werking begrijpen en kunnen uitleggen waarom vectoren nodig zijn soms (i.p.v. lijsten)		Iteratieve procedures kunnen schrijven op vectoren (zowel met do als met staartrecursie)
Weten hoe je lijsten omzet naar vectoren. Wat is de moeilijkheid en hoe wordt het opgelost?		De code van de minilogo evaluator kunnen uitleggen en kunnen uitleggen wat het verband is met de concepten gezien in dit hoordstuk.
Weten wat een bitvector is en weten wat/hoe de operaties op bitvectoren geïmplementeerd worden.		Zelf een bitvector operatie kunnen toevoegen (bvb nand of xor).
Hoofdstuk 8		

Weten welke manieren er bestaan om name clashes tussen import clausules op te lossen en weten hoe je die moet gebruiken.		R7RS programma's kunnen schrijven die gebruik maken van libraries.
		R7RS libraries kunnen schrijven met de juiste exports.
		DrRacket instellen in 'Determine language from source' mode voor #lang r7rs
		packages installeren via DrRacket
Weten wat een ADT is, waarom ADTs nodig zijn en wat de voordelen zijn van het denken in termen van ADTs.		Het proceduretype kunnen opschrijven van een gekende procedure.
Weten wat bedoeld wordt met 'implementatie van een ADT' en 'representatie van het datatype'		
Weten dat we verzamelingen zowel geordend als ongeordend kunnen representeren en weten wat de voor en nadelen zijn. Begrijpen dat verandering van representatie geen invloed zou mogen hebben op de functionaliteit van de operaties.		
Weten dat we een matrix kunnen voorstellen als geneste vectoren en begrijpen hoe 2-dimensionale indexering werkt als combinaties van vector-ref en vector-set!		
Weten wat record types zijn en wat de voordelen zijn. De algemene vorm van define-record-type kennen. Weten dat mutatoren optioneel zijn.		Record types kunnen gebruiken in samenwerking met libraries.
Weten wat een dictionary is en wat de belangrijkste operaties zijn.		ADT's implementeren door middel van de 3 geziene principes: getagde lijsten, getagde vectoren of record-types + getters en setters definiëren (bij de eerste 2)
Weten wat testing is en wat een testing framework voor je kan doen.		
Hoofdstuk 9		
Een voorbeeld kunnen geven van hoe 2 geneste procedures dezelfde omliggende variablen kunnen lezen en schrijven.		
Kunnen uitleggen (adhv een vb) waarom noch het substiutiemodel noch het atomaschrijftjesmodel voldoende zijn om het gedrag van Scheme procedures helemaal te verklaren.		
Wat is een omgeving?		
Hoe wordt een variabele opgezocht om uit te lezen en/of aan te passen met set!		Uitbreidingen van een omgeving kunnen tekenen. Procedure-objecten kunnen tekenen, met of zonder "naam pijl" uit de omgeving van definitie.
De 2 basisregels kennen en begrijpen om procedures te evalueren en op te roepen.		De 2 basisregels op systematische wijze kunnen toepassen op complexe voorbeelden met veel nesting en veel oproepen (gegeven de code, zelfs als die betekenisloos is vanuit praktisch standpunt).
Weten wat een closure is.		Weten waar de verschillende soorten pijlen in een omgevingsdiagram vandaan komen.
Weten waar de verschillende soorten pijlen in een omgevingsdiagram vandaan komen.		Omgevingsdiagrammen kunnen tekenen voor hogere orde procedures en voor de procedures teruggegeven door hogere orde procedures.
Werking van let, let* en letrec kunnen uitleggen a.d.h.v. het omgevingsdiagram van hun ontsuikering.		Correct kunnen aangeven welke variabelen gelezen en geschreven zullen worden (met set!) op basis van een omgevingsdiagram.
Weten wat een wrapper is.		Wrappers kunnen bouwen voor gegeven procedures.
Kunnen uitleggen wat memoization is, hoe de techniek werkt en wat de voordelen zijn.		Een gegeven procedure met 1 argument zelf kunnen memoizen.
Hoofdstuk 10		
Weten wat object-gericht programmeren is.		apply kunnen gebruiken op een procedure en een lijst
Weten wat encapsulatie is.		Procedures van variabele ariteit kunnen schrijven.
Kunnen uitleggen dat elk object een closure is met lokale state.		Een eenvoudig objecten-maker programmeren.
Kunnen uitleggen wat late-binding polymorfisme is.		Het bijhorende omgevingsdiagram kunnen tekenen.
Weten wat het verschil is tussen een bericht en een methode.		
Weten wat de interface van een object is.		
Het verschil kennen tussen monomorfe en polymorfe expressies.		
De verschillen tussen procedurele en OOP-gebaseerde ADT implementaties kennen.		Een ADT kunnen implementeren m.b.v. objecten en dispatchers.

Kunnen uitleggen waarom single-dispatch objecten soms geen goede programmeertechniek is.	
Weten wat getagde data is.	
Het multiple-dispatch mechanisme kunnen uitleggen (versie 1.0) zonder coercions.	Zelf een ADT met 2 verschillende representaties kunnen programmeren en beide representaties aan een multiple-dispatch tabel toevoegen + gebruiken.
Weten wat coercions zijn en waarom ze nodig zijn bij multiple-dispatch.	
Het multiple-dispatch mechanisme kunnen uitleggen (versie 2.0) met coercions.	Klassen kunnen opzoeken in de Racket documentatie (c.f. GUI voor programmeerproject 1).
Hoofdstuk 11	
Begrijpen dat een geneste lijst als boom bekeken kan worden.	Boomrecursieve procedures kunnen schrijven op geneste lijsten. Toepassing van "het geneste lijsten patroon".
Verschillende definities van "boom" kennen en begrijpen waarom niet elke geneste lijst aan die definitie(s) voldoet.	Hogere orde procedures van lijsten kunnen herimplementeren op bomen.
Weten wat een familieboom is en wanneer familiebomen toepasbaar zijn om een situatie te modeleren.	Een gevraagde procedure op familiebomen kunnen implementeren volgens "het familiebomen patroon" (2 mutueel recursieve procedures).
Weten wat een binaire voorstelling van een alfabet met vaste en variabele lengte is en weten waarom dat laatste efficiënter kan zijn.	
De prefix-eigenschap kennen en kunnen duiden.	
Weten dat frequentie-studies van alfabetten de leiddraad zijn voor een voorstelling met variabele lengte.	Een boodschap kunnen coderen en decoderen, gegeven een Huffman boom voor het alfabet van de boodschap.
Weten hoe een Huffman-boom eruit ziet, wordt voorgesteld en opgebouwd wordt m.b.v. het geziene algoritme.	
Weten wat een expressieboom is. De expressieboom kunnen tekenen voor eenoudige rekenkundige expressies.	
Weten hoe expressiebomen voorgesteld worden in Scheme.	De procedures om expressies uit te rekenen en af te leiden kunnen uitbreiden met nieuwe operatoren.
Weten wat generatieve boomrecursie is.	
Kunnen uitleggen hoe de yield van een gegenereerde boom (tot zekere diepte) overeen komt met instructies voor de turtle.	
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 een accumulator bestaat.	
Het ADT stream kennen. De implementatie met eindige lijsten kennen.	Stromen kunnen maken en verwerken a.d.h.v. de procedures van het stream ADT. Kunnen uitleggen wat een stroomdiagram weergeeft. Zelf een stroomdiagram kunnen tekenen.
De implementatie van stream-append kennen.	Zelf procedures schrijven over stromen in de directe stijl (= nieuwe stroomoperatoren definieren).
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.	Cons/car/cdr/... gebruiken wanneer je met lijsten bezig bent, en het stream ADT wanneer je met stromen bezig bent.
De implementatie van stream-map kennen en stream-map kunnen oproepen met de juiste argumenten.	Gebruik kunnen maken van de gekende stroomoperatoren, d.w.z. directe stijl vermijden indien mogelijk en de juiste argumenten meegeven aan de stroomoperatoren.
Begrijpen dat sommige argumenten van de stroom operaties "configuratie-argumenten" zijn.	
De implementatie van stream-filter kennen en stream-filter kunnen oproepen met de juiste argumenten.	
Idem voor stream-accumulate.	
En stream-for-each	
Kunnen uitleggen dat 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.		Procedures schrijven die geneste stromen als input nemen en/of als output teruggeven.
De implementatie van matrices als geneste stromen kunnen uitleggen en de code kunnen toelichten. Ook zo voor de geziene operatoren uit de linearire 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 begrijpen in de REPL.		Procedures schrijven die over (mogelijks) oneindige stromen gaan.
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.		Een impliciete stroom kunnen definieren en procedures over schrijven.
De Zeef van Erastosthenes 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.		Ideeen voor het implementeren van stream-flatten/interleaved zelf kunnen toepassen.
Begrijpen dat real-time applicaties (zoals de MIVB) geschreven kunnen worden als stroomverwerkers die hun resultaat met een stroom accumulator laten zien.		