Stichting NIOC en de NIOC kennisbank

Stichting NIOC (www.nioc.nl) stelt zich conform zijn statuten tot doel: het realiseren van congressen over informatica onderwijs en voorts al hetgeen met een en ander rechtstreeks of zijdelings verband houdt of daartoe bevorderlijk kan zijn, alles in de ruimste zin des woords.

De stichting NIOC neemt de archivering van de resultaten van de congressen voor zijn rekening. De website www.nioc.nl ontsluit onder "Eerdere congressen" de gearchiveerde websites van eerdere congressen. De vele afzonderlijke congresbijdragen zijn opgenomen in een kennisbank die via dezelfde website onder "NIOC kennisbank" ontsloten wordt.

Op dit moment bevat de NIOC kennisbank alle bijdragen, incl. die van het laatste congres (NIOC2025, gehouden op donderdag 27 maart 2025 jl. en georganiseerd door Hogeschool Windesheim). Bij elkaar zo'n 1500 bijdragen!

We roepen je op, na het lezen van het document dat door jou is gedownload, de auteur(s) feedback te geven. Dit kan door je te registreren als gebruiker van de NIOC kennisbank. Na registratie krijg je bericht hoe in te loggen op de NIOC kennisbank.

Het eerstvolgende NIOC vindt plaats in 2027 en wordt dan georganiseerd door HAN University of Applied Sciences. Zodra daarover meer informatie beschikbaar is, is deze hier te vinden.

Wil je op de hoogte blijven van de ontwikkeling rond Stichting NIOC en de NIOC kennisbank, schrijf je dan in op de nieuwsbrief via

www.nioc.nl/nioc-kennisbank/aanmelden nieuwsbrief

Reacties over de NIOC kennisbank en de inhoud daarvan kun je richten aan de beheerder:

R. Smedinga kennisbank@nioc.nl.

Vermeld bij reacties jouw naam en telefoonnummer voor nader contact.

Een aanzet tot een didactiek van recursief programmeren

H. Koppelman
N.M. van Diepen
Faculteit Informatica
Universiteit Twente
Postbus 217
7500 AE Enschede

Samenvatting

Er worden drie uitgangspunten voor onderwijs in recursief programmeren besproken: procedurele abstractie, een vast implementatie-schema, en de reading-approach als instructie-strategie. Deze uitgangspunten worden uitgewerkt tot een aanzet tot een didactiek van recursief programmeren.

1 Inleiding

Recursief leren programmeren wordt door velen als moeilijk ervaren. Roberts (1986:1) merkt daarover op: "For the student (...) recursion appears to be obscure, difficult, and mystical. (...) recursion is an unfamiliar idea and often requires thinking about problems in a new and different way."
Dit artikel beschrijft een systematische aanpak om dat lastige concept recursie te onderwijzen.

Deze aanpak is gebaseerd op onderzoeksresultaten uit de literatuur, en op eigen ervaringen. Die ervaringen hebben we vooral opgedaan door studenten geselecteerde vraagstukken voor te leggen, en de uitwerkingen daarvan te analyseren (zie Koppelman e.a., 1989). Dit levert het volgende beeld op:

- de strategie om na te gaan of recursieve implementaties het gewenste effect hebben, is niet toereikend
- het ontbreekt aan een systematiek in het ontwerpen van recursieve implementaties
- er is weinig besef van procedurele abstractie
- er bestaat verwarring tussen recursie en iteratie

De aanpak die hier wordt voorgesteld, is bedoeld te voorzien in bovenstaande knelpunten.

In voorbeelden wordt als programmeertaal PASCAL gebruikt. De voorgestane benadering van onderwijs in recursie is echter te vertalen naar andere talen die recursie toestaan.

2 Uitgangspunten voor onderwijs in recursie

We noemen een aantal uitgangspunten die onzes inziens essentieel zijn voor onderwijs in recursie.

2.1 Procedure-aanroepen beschouwen op hun gespecificeerd effect

Er zijn twee manieren om tegen een procedure-aanroep aan te kijken. (Met procedures worden hier ook functies bedoeld.) In Roberts (1986) worden deze manieren in verband gebracht met de termen holisme en reductionisme (deze termen zijn afkomstig uit Hofstadter (1979)).
Kenmerkend voor het *reductionisme* is (zie Roberts, 1986:10): "a whole can be understood completely if you understand its parts, and the nature of their sum".
Vertaald naar een procedure-aanroep betekent dit: het effect van een aanroep kan begrepen worden door de *implementatie* van de procedure te bestuderen. Een procedure-aanroep wordt beschouwd als een manier om de besturing naar een ander deel van het programma te verplaatsen.
Het *holisme* wordt als volgt gekenschetst (zie Roberts, 1986:10): "the whole is greater than the sum of the parts". In het holistisch perspectief kan het effect van een procedure-aanroep begrepen worden uit de *specificatie* van de procedure. Een procedure-aanroep wordt beschouwd als een middel tot abstractie.

Wie vanuit een *reductionistisch* perspectief een recursieve implementatie probeert te begrijpen, neemt een voorbeeld van een aanroep en simuleert het proces van de achtereenvolgende recursieve aanroepen. De specificatie is niet direkt van belang.

Om een recursieve implementatie te begrijpen vanuit een *holistisch* perspectief, wordt de recursieve aanroep op zijn effect beschouwd. Dat effect staat beschreven in de specificatie van de betreffende procedure. Om een recursieve aanroep te kunnen begrijpen, is de implementatie dus niet van belang.

Het onderscheid tussen de beide benaderingen is het belang dat wordt toegekend aan de scheiding van specificatie en implementatie, dus aan de scheiding van het effect van een procedure en de manier waarop dat effect wordt bereikt.

In de hier gepresenteerde aanpak worden studenten vanaf het begin expliciet vanuit een *holistische* benadering onderwezen. Er is niets tegen een gegeven recursieve implementatie proberen te begrijpen vanuit reductionistisch perspectief. Het is echter niet toereikend alleen vanuit dit perspectief naar een implementatie te kunnen kijken. Bovendien biedt het reductionistisch perspectief geen houvast bij het *ontwerpen* van een recursieve implementatie. Beide aspecten, het begrijpen van een gegeven recursieve implementatie, en het ontwerpen van een recursieve implementatie, worden beter ondersteund vanuit het holistisch perspectief.

Het kost tijd en moeite om dit perspectief te verwerven. Beginnende programmeurs zijn niet van nature geneigd recursieve aanroepen op hun effect te beschouwen. Verwonderlijk is dit overigens niet. Het machinemodel dat impliciet of expliciet in inleidende programmeercursussen wordt aangereikt is een *proces*-model. Taalconstructies worden gezien in termen van processen, en niet in termen van effecten.

Het begrijpen en ontwerpen van een recursieve implementatie is in de hier gegeven aanpak gebaseerd op de specificatie van de procedure. Er moeten dus voor studenten duidelijke afspraken en regels zijn over de wijze van specificatie. In de voorbeelden in dit artikel is gekozen voor specificatie van procedures via pre- en postcondities.

2.2 Een vast implementatieschema

Een vaste structuur, in de vorm van een model van een recursieve implementatie, kan een belangrijk hulpmiddel zijn bij het (recursief) leren programmeren. Pirolli (1986:351) zegt hierover: "(...) it appears that a crucial problem faced by programming students is a lack of explicit instruction on the underlying planning structure for recursive functions." en: "Unfortunately, many textbooks do not provide instruction on the structure of recursive functions, or how such structures should be written. Rather, most textbooks focus on descriptions of the processes generated by recursive functions- that is, how recursive functions work." (zie Pirolli 1986:324)
Uit een experiment (zie Pirolli 1985) blijkt dat studenten die expliciete instructie krijgen over de structuur, het beter doen dan studenten die geïnstrueerd krijgen hoe recursie werkt. De conclusie is dan: "It seems that knowing the underlying structure and functionality of recursive programs facilitates the induction of skill needed for programming recursion more than knowledge of how such functions get

executed." (zie Pirolli 1986:325)

Wie een *recursief* algoritme nastreeft, moet twee problemen onderkennen en oplossen: het algemene probleem en het bijzondere probleem. Om het *algemene* probleem op te lossen, moet men proberen in het probleem een of meer deelproblemen te herkennen die van dezelfde aard zijn als het oorspronkelijke probleem, maar kleiner van omvang. Daarnaast is er een *bijzonder* probleem: een probleem dat zo'n kleine omvang heeft dat de oplossing eenvoudig en direkt (= niet-recursief) is.

Kenmerken van een recursief algoritme zijn dus:

- de afbakening van het bijzondere en het algemene probleem
- de oplossing van het bijzondere probleem
- de oplossing van het algemene probleem

In een vaste structuur moeten deze onderdelen herkenbaar zijn, en moet duidelijk zijn hoe ze met elkaar samenhangen. In Pascal voldoet het volgende template hieraan:

```
IF   "probleem is bijzonder probleem"
THEN "oplossing bijzonder probleem"
ELSE "oplossing algemeen probleem"
```

Een recursief algoritme bij een gegeven probleem ontwerpen betekent dus een geschikte inhoud geven aan het template.

Een implementatie volgens het gegeven template is beter te begrijpen dan implementaties waarin de verschillende aspecten minder duidelijk zichtbaar zijn. Ter illustratie het volgende voorbeeld:

```
TYPE
  PosInt= 1..MaxInt;

PROCEDURE DrukSterretjesAf (N: PosInt);

(* pre:  true
   post: op de standaarduitvoer staan N asterisks
         afgedrukt
*)
```

Een implementatie die voldoet aan het template luidt:

```
BEGIN
IF    N = 1
THEN  Write ('*')
ELSE  BEGIN
      DrukSterretjesAf (N-1);
      Write ('*')
      END
END;
```

In deze implementatie zijn de oplossingen van het algemene en het bijzondere probleem van elkaar gescheiden en direkt terug te vinden. In de volgende, ook correcte, implementatie is dat niet het geval:

```
BEGIN
IF   N > 1
THEN DrukSterretjesAf (N-1);
Write ('*')
END;
```

De oplossingen van het algemene en het bijzondere probleem zijn in deze implementatie ook aanwezig, maar ze zijn met elkaar verweven. Deze tweede implementatie is daarom minder helder, en moeilijker te begrijpen dan de eerste.

2.3 De reading approach

In Van Merriënboer e.a. (1987) worden drie instructie-strategieën vergeleken voor het ontwerpen van inleidende programmeercursussen. De conclusie is dat één van deze strategieën, de reading approach, superieur is. In deze benadering worden studenten vanaf het begin geconfronteerd met niet-triviale programmeerproblemen. Echter, deze problemen worden gepresenteerd in combinatie met een volledige of gedeeltelijke implementatie in de vorm van programma's die op de juiste wijze zijn ontworpen, gestructureerd en gedocumenteerd. De taken van de studenten worden geleidelijk meer complex, vanaf het analyseren van programma's via het aanvullen en wijzigen, tot aan het ontwerpen van programma's.

Voordelen van deze strategie zijn: de complexiteit van gestelde taken is gemakkelijk te beheersen, er is een groot aantal problemen mogelijk, er is een grote variatie in opgedragen taken mogelijk, studenten ondervinden aan den lijve hoe nuttig documentatie en een goed programma-ontwerp zijn, en de complexiteit van problemen kan vanaf het begin hoog zijn.

Zoals gezegd is voor beginners het holistisch perspectief moeilijk te verwerven is. De verleiding is groot recursieve implementaties procesmatig na te

lopen. De vaardigheid op tijd te stoppen met analyseren, ofwel de vaardigheid om te abstraheren, moet geleerd worden. De beginfase van de reading-approach leent zich goed om die vaardigheid te leren. Dit kan worden bereikt door studenten een recursieve implementatie aan te bieden, en daar vragen over te stellen. Op deze wijze kan het abstraheren specifiek en efficiënt worden geoefend.

3 Een aanzet tot een didactiek van recursie

In het voorgaande zijn uitgangspunten genoemd voor onderwijs in recursie. Deze uitgangspunten worden hier geïntegreerd in een didactisch model. (zie Koppelman e.a. (1989) voor meer details en voorbeelden)

Uitgaande van de reading-approach ligt het voor de hand recursie in een aantal fasen te onderwijzen. De taken van de studenten veranderen daarin van het analyseren van een gegeven procedure, tot het ontwerpen van een implementatie bij een gegeven specificatie. We onderscheiden drie fasen, waarin de leerdoelen achtereenvolgens luiden:

1 het kunnen begrijpen van een gegeven recursieve implementatie, bij een gegeven specificatie.
2 een gegeven recursieve implementatie kunnen aanvullen en wijzigen, bij een gegeven specificatie
3 een recursieve implementatie kunnen ontwerpen bij een gegeven specificatie

Van belang in de reading approach zijn niet alleen de onderscheiden fasen zelf, maar ook de volgorde daarin. Onderwijs in programmeren begint volgens deze benadering door *volledige* programma's te geven, en daarover vragen te stellen.

De drie fasen komen achtereenvolgens ter sprake.

3.1 Recursieve implementaties begrijpen

Studenten kunnen het eerstgenoemde leerdoel oefenen met opdrachten van de volgende vorm:

gegeven: een specificatie en een recursieve implementatie
gevraagd: na te gaan of de implementatie het gespecificeerde effect heeft

De gevraagde activiteit wordt in dit artikel aangeduid met de term *verificatie*. Daaronder wordt verstaan: het op informele wijze nagaan of een gegeven implementatie het gespecificeerde effect heeft, dus correct is.

Zoals gezegd is de natuurlijk neiging te verifiëren vanuit het reductionistisch perspectief. Het is nodig via expliciete oefeningen de holistische kijk aan te brengen. In de woorden van Ford (1984:213): "Students are taught to think of procedure invocations as indivisible operations, or in other words, as new primitive statements in the programming language. Once such a new statement is defined, it may be used just like any other statement in the language."

Een benadering van verificatie waarin deze gedachte centraal staat, is gebaseerd op *volledige inductie*. Hierbij moeten eerst het bijzondere en het algemene probleem worden geïdentificeerd. Vervolgens moet nagegaan worden:
1 of de oplossing van het bijzondere probleem juist is
2 of de oplossing van het algemene probleem juist is, onder de veronderstelling dat de recursieve aanroepen die daarin voorkomen het gespecificeerde effect hebben.
Bovendien moet nagegaan worden of die recursieve aanroepen betrekking hebben op "kleinere" problemen.

Ter illustratie volgt een toepassing.

Gegeven is de specificatie:

```
PROCEDURE DrukSterrenAf (N: PosInt);

(* pre:  true
   post: de standaarduitvoer bevat N asterisks, gevolgd
         door een punt
*)
```

Gevraagd: voldoet de volgende implementatie aan de specificatie?

```
BEGIN
IF   N=1
THEN Write ('*.')
ELSE BEGIN
     Write ('*');
     DrukSterrenAf (N-1);
     Write ('.')
     END
END;
```

1 het is eenvoudig in te zien dat het bijzondere probleem, het op de gewenste wijze afdrukken van één asterisk, juist is opgelost.
2 het algemene probleem, het op de gewenste wijze afdrukken van N asterisks, met N>1, is niet juist opgelost. Als we veronderstellen dat de recursieve aanroep het gespecificeerde effect heeft, hebben de

opdrachten:

```
Write ('*');
DrukSterrenAf (N-1);
Write ('.')
```

het volgende effect:

het afdrukken van één asterisk
het afdrukken van N-1 asterisks, gevolgd door een punt
het afdrukken van een punt

Het uiteindelijke effect is dus (op basis van de inductie-veronderstelling) dat N asterisks zijn afgedrukt, gevolgd door twee punten. Dit effect komt niet overeen met de specificatie, dus de implementatie is onjuist.

Bij het verifiëren spelen termen uit het template een rol. Studenten dienen het template dus in zoverre te beheersen dat ze in een gegeven implementatie de onderdelen ervan kunnen aanwijzen, en de betekenis van die onderdelen kunnen geven. Bovendien moeten ze weten wat verstaan wordt onder een "kleiner probleem".

Een type vraagstukken dat beslist *niet* past binnen onze aanpak, is het effect vragen van een aanroep van een niet-gespecificeerde recursieve procedure. Essentieel in onze aanpak is dat een aanroep van procedures worden gezien als een "new primitive statement", en van elk statement dient het effect bekend te zijn.

3.2 Een gegeven recursieve implementatie aanvullen en wijzigen

De tweede fase in de reading approach betreft het aanvullen en wijzigen van een programma. Voor het leren van recursie is dit te vertalen naar het invullen van (delen van) implementaties, zonder dat het derde leerdoel (zie 3.3) een rol speelt. Op voorhand ligt dus de oplossing van het probleem vast: de opdeling in bijzonder en algemeen probleem is bekend.

Het aanvullen van een recursieve implementatie

Het implementatieschema voor recursieve implementaties bevat drie 'open plaatsen', namelijk de afbakening van het algemene en bijzondere probleem, de oplossing van het bijzondere probleem en de oplossing van het algemene probleem. Het ligt voor de hand steeds één van deze 'plaatsen' te laten (aan)vullen. De inhoud van de andere twee wordt voorzover nodig gegeven.

Het wijzigen van een recursieve implementatie

Er valt te denken aan vragen als: Is een ander bijzonder probleem denkbaar? Is de oplossing van het bijzondere probleem uniek? Blijft de implementatie correct als opdrachten van de oplossing van het algemene probleem van volgorde veranderen? Hoe moet de implementatie worden aangepast bij een licht gewijzigde probleemstelling?

3.3 Een recursieve implementatie ontwerpen

In het proces van het ontwerpen van een recursieve implementatie kan op verschillende manieren structuur worden gebracht.
In de eerste plaats door uit te gaan van het template.
In de tweede plaats door de invulling van het template in *stappen* te laten geschieden. Daarbij valt te denken aan de volgende stappen:

1 recursief aankijken tegen het datatype waarop de bewerkingen plaatsvinden
2 het algemene en het bijzondere probleem afbakenen
3 de oplossing bepalen van het bijzondere probleem
4 de oplossing bepalen van het algemene probleem

De strategie om een probleem op te lossen is in de probleemstelling deze stappen proberen te identificeren.
(De stappen zijn niet altijd onafhankelijk van elkaar en kunnen dus niet altijd strikt sequentieel worden doorlopen.)
In deze stappen zelf kan ook weer structuur worden gebracht.

1 recursief aankijken tegen het datatype

Om een probleem recursief op te lossen, moeten we het zodanig splitsen dat één of meer kleinere problemen van dezelfde aard ontstaan. Ideeën voor deze splitsing kunnen we in veel gevallen opdoen door recursief aan te kijken tegen het datatype waarop de bewerkingen plaatsvinden. Een array kan bijvoorbeeld recursief worden opgevat als bestaande uit een eerste element en een array, namelijk de rest van de oorspronkelijke array. Een "kleiner probleem van dezelfde aard" is dan het oorspronkelijke probleem, maar dan met betrekking tot de "rest-array".
Niet alleen arrays, maar bijvoorbeeld ook natuurlijke getallen, files, bomen en lijsten laten zich recursief beschouwen.

Als een datatype recursief wordt opgevat, is er ook altijd sprake van een *bijzonder* geval: het geval dat zo

"klein" is dat het niet meer recursief wordt opgevat. Een array die uit 1 element bestaat, kan als bijzonder geval worden beschouwd. (In principe kan het bijzondere geval worden *gekozen*. Het is bijvoorbeeld ook mogelijk een array met nul elementen als een bijzonder geval te beschouwen, of een array met twee elementen.)

2 het algemene en het bijzondere probleem afbakenen

In een recursief algoritme wordt onderscheid gemaakt tussen het algemene en het bijzondere probleem. Het resultaat van stap 2 is dat wordt vastgelegd (en ingevuld in het template) wat als bijzonder probleem wordt beschouwd. In termen van de voorgaande stap moet dus het bijzondere geval van het datatype worden omschreven.

3 het bijzondere probleem oplossen

In deze stap wordt de oplossing van het bijzondere probleem bepaald en ingevuld in het template. Die oplossing is meestal eenvoudig, en bevat geen recursieve aanroepen.

Soms is de oplossing van het bijzondere probleem wel zeer triviaal, namelijk als er *niets* hoeft te gebeuren. Te denken valt bijvoorbeeld aan het afdrukken van een lege lijst. Ook in een dergelijk geval is het wenselijk niet van het template af te wijken en als oplossing van het bijzondere probleem een commentaarregel, bijvoorbeeld: (* doe niets *), op te nemen.

4 het algemene probleem oplossen

De volgende fasen kunnen worden onderscheiden:

- de *decompositie*: het gegeven probleem zodanig splitsen dat hetzelfde probleem weer terugkomt, maar dan "kleiner"
- de *compositie*: een algoritme voor het oorspronkelijke probleem beschrijven, ervan uitgaande dat "kleinere" problemen opgelost kunnen worden via een geschikte recursieve aanroep

De decompositie wordt gesuggereerd in stap 1, waarin op recursieve wijze wordt aangekeken tegen het datatype waarop de bewerkingen plaatsvinden.
De compositie is gebaseerd op het *inductieprincipe*: de veronderstelling is dat kleinere problemen van dezelfde aard opgelost kunnen worden via recursieve aanroepen (zie Manber 1988). De overblijvende vraag is dan: hoe luidt de oplossing van het oorspronkelijke probleem? Het antwoord op deze vraag vormt de oplossing van het algemene probleem.

4 Het splitsen van problemen

Een recursieve oplossing van een probleem kan vaak worden afgeleid uit een recursieve kijk op het betreffende datatype. In het algemeen zijn er evenwel verschillende mogelijkheden om recursief naar hetzelfde datatype te kijken. Een array kunnen we bijvoorbeeld recursief opvatten als:

- een eerste element *gevolgd* door een array

of als:

- een laatste element *voorafgegaan* door een array

of als:

- een eerste element gevolgd door een array gevolgd door een laatste element.

Het is ook mogelijk een array te zien als een concatenatie van arrays. Een array kan bijvoorbeeld worden opgevat als:

- een concatenatie van twee (ongeveer) even lange arrays

of als:

- een concatenatie van een array van 1 element en een array

Er zijn uiteraard allerlei varianten denkbaar.

Bij al deze manieren hoort een verschillende decompositie, en een verschillende compositie.
Het lijkt zinvol studenten problemen op diverse manieren te laten decomponeren om ze de nodige flexibiliteit aan te leren.
Oefening met verschillende manieren om een datatype te splitsen is ook nuttig omdat verschillende problemen verschillende eisen kunnen stellen aan de recursieve kijk op het datatype. We geven daarvan een voorbeeld.

```
FUNCTION AantalCijfers (N: PosInt): Integer;

(* pre:  true
   post: AantalCijfers (N) = het aantal cijfers van N
*)
```

In dit geval wordt een geheel getal opgevat als een rij symbolen. Vaak vatten we een rij recursief op als een

eerste element *gevolgd* door een rest-rij. Bij deze kijk kunnen we voor het voorbeeld geen eenvoudige oplossing van het algemene probleem vinden. Het is niet eenvoudig het eerste cijfer van een geheel getal te bepalen. Een eenvoudige oplossing is wel te vinden door de rij cijfers recursief te beschouwen als een element *voorafgegaan* door een rij. Het laatste cijfer van een getal en de restrij van het getal laten zich gemakkelijk bepalen (via de operaties MOD 10 en DIV 10).

5 Recursie naast iteratie

Het is bij onderwijs in imperatieve talen gebruikelijk recursie in te voeren nadat de besturingsstructuur iteratie is behandeld. Verder wordt recursie vaak geïntroduceerd aan de hand van een probleem dat al in een eerder stadium iteratief is opgelost. Het gevaar bestaat dat studenten op deze wijze een verkeerd beeld van recursie vormen, en recursie aanvankelijk opvatten als een soort iteratie. Dit gevaar is des te groter omdat voor eenvoudige problemen het beeld nog toereikend is. Het schiet echter te kort bij ingewikkelde situaties. Implementaties met meer dan één recursieve aanroep zijn bijvoorbeeld moeilijk te begrijpen in termen van iteratie. Verder biedt deze misconceptie geen houvast bij het ontwerpen van recursieve algoritmen.

De misconceptie krijgt minder kans als vanaf het begin iteratieve en recursieve oplossingen nadrukkelijk van elkaar worden onderscheiden en de kenmerken van elk van beide expliciet worden gemaakt. Dit is te bewerkstelligen door van een aantal problemen zowel een iteratieve als een recursieve oplossing te laten zien. Op deze wijze kan duidelijk worden dat een verschillende kijk op hetzelfde probleem tot verschillende oplossingen kan leiden. Recursiviteit is dus niet zozeer een kenmerk van een probleem, maar veeleer van een oplossing.
Wie een oplossing met een *iteratie* nastreeft, moet op zoek gaan naar een aantal opdrachten dat herhaald uitgevoerd moet worden. Het grootste getal van een rij getallen bepalen we bijvoorbeeld door alle getallen langs te lopen, en de grootste-tot-dan-toe bij te houden.
Wie een recursief algoritme voor een probleem zoekt, moet op zoek gaan naar kleinere problemen van dezelfde aard. Het grootste van een rij getallen is dan het maximum van het eerste getal en de 'grootste van de rest'.

Het iteratieve en het recursieve algoritme van het

voorbeeld geven blijk van een verschillende kijk op de datastructuur, de rij getallen. In de *recursieve* optiek bestaat de verzameling uit een element, en de rest van de rij. Bij *iteratie* wordt de rij "ontvouwen", en zien we stuk voor stuk alle afzonderlijke elementen waaruit de rij bestaat.

Iteratieve en recursieve algoritmen stellen verschillende eisen aan het machinemodel dat nodig is om algoritmen te kunnen ontwerpen en begrijpen. In inleidende programmeercursussen is het model dat wordt aangeboden veelal gebaseerd op een model van een geheugen van een machine, en van de wijze waarop een machine met waarden in het geheugen manipuleert. Statements zijn instrukties aan de machine en expressies zijn rekenvoorschriften.
Voor het begrijpen van programma's waarin uitsluitend iteratie, selectie en dergelijke voorkomen, is dit beeld nog wel toereikend; voor het begrijpen van recursie schiet het echter tekort.
Een meer adequaat beeld ontstaat door te abstraheren van de machine. Taalconstructies worden beschouwd op hun effekt en niet op de manier waarop dat effekt tot stand komt. Van gebruikte keuzeopdrachten of herhalingsopdrachten, en vooral ook van procedures, ligt het effekt vast en moet het expliciet geformuleerd kunnen worden.

6 Afsluiting

We hebben een aanpak geschetst om recursief programmeren te onderwijzen. Een aantal aspecten van deze aanpak passen we toe in het onderwijs. Daarbij kan met name de mede door ons ontwikkelde COO-cursus Recursief programmeren worden genoemd.
We zijn op zoek naar verbeteringen en aanvullingen van onze aanpak. We besluiten met een aantal vragen die in dat kader interessant zijn om te onderzoeken:

- in welke volgorde heeft het (op onderwijskundige gronden) de voorkeur iteratie en recursie aan te bieden? In Kessler e.a. (1986) wordt de conclusie getrokken dat er bij de volgorde iteratie-recursie transfer optreedt. Mogelijk wordt echter in het betreffende onderzoek recursie door de proefpersonen begrepen in termen van iteratie. Bij de volgorde recursie-iteratie is geen transfer gemeten. Een mogelijk alternatief dat niet is onderzocht, is het gelijktijdig aanbieden van iteratie en recursie.
- een machinemodel gebaseerd op een procesmodel van het geheugen is niet toereikend voor het concept recursie. Uitgangspunt dient veeleer te zijn de

effectbeschrijving van taalelementen. De vraag is echter hoe zo'n model er precies uit moet zien.
- is de natuurlijke neiging datatypen als lijsten, arrays en files op te vatten als een eerste element gevolgd door het staartstuk en heeft dit tot gevolg dat het meer moeite kost algoritmen die een andere kijk vereisen te bedenken?

Gebruikte literatuur

Ford, G.A. (1984) An Implementation Independent Approach to Teaching Recursion. In: SIGCSE Bulletin 16, 1, 213-216
Hofstadter D.R. (1979) Godel, Escher, Bach: an Eternal Golden Braid. New York: Basic Books;
Kessler, C.M. & J.R. Anderson (1986) Learning Flow of Control: Recursive and Iterative Procedures. In: Human-Computer Interaction 2, 135-166
Koppelman, H. & N.M. van Diepen (1989), Onderwijs in recursie. In: Memoranda Informatica Universiteit Twente, 89-24
Manber, U. (1988) Using induction to design algorithms. In: Communications of the ACM 31, 11, 1300-1313
Merriënboer, J.J.G. van & H.P.M. Krammer (1987) Instructional strategies and tactics for the design of introductory computer programming courses in high school. In: Instructional Science 16, 251-285
Pirolli, P.L. (1985) Problem Solving by Analogy and Skill Acquisition in the Domain of Programming. Pittsburgh: Carnegie-Mellon University;
Pirolli, P.L. (1986) A Cognitive Model and Computer Tutor for Programming Recursion. In: Human-Computer Interaction 2, 319-355
Roberts, E.S. (1986) Thinking Recursively. New York: John Wiley & Sons;