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 (NIOC2023, gehouden op donderdag 30 maart 2023 jl. en georganiseerd door NHL Stenden Hogeschool). Bij elkaar bijna 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 op donderdag 27 maart 2025 in Zwolle en wordt dan georganiseerd door Hogeschool Windesheim. Kijk op www.nioc2025.nl voor meer informatie.

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.

Funktioneel programmeren als basis voor het programmeeronderwijs

S.M.M. Joosten, K.G. van den Berg
Universiteit Twente
faculteit Informatica/SETI
postbus217
7500 AE ENSCHEDE

Samenvatting

Aan de Universiteit Twente is de afgelopen vier jaar geëxperimenteerd met een inleidende kursus in het programmeren, op basis van funktioneel programmeren. Dit artikel gaat in op het waarom van deze aanpak, de doelstelling van het programmeeronderwijs, de opzet en inhoud van de kursusonderdelen en de evaluatie van de kursus.

1 Inleiding

Er is een groeiende belangstelling voor het funktioneel programmeren, vanwege de eenvoud en de orthogonaliteit van een aantal programmeertalen uit deze groep. Het ligt dan ook voor de hand dat een eerste kennismaking met het programmeren in een funktionele programmeertaal plaatsvindt. Deze aanpak is zo nieuw, dat er nauwelijks plaatsen in de wereld zijn aan te wijzen waar een vergelijkbare aanpak in het programmeeronderwijs wordt gevolgd (bijv. Bailes & Salzman 1989).

Tot nu toe was het inleidend programmeeronderwijs aan de fakulteit Informatika van de Universiteit Twente gebaseerd op imperatieve talen, i.c. Pascal en Modula-2. Een beslissing om over te gaan op funktioneel programmeren is ingrijpend, en mag dan ook niet lichtvaardig worden genomen. Er is uitvoerig en zorgvuldig geëxperimenteerd met inleidend programmeeronderwijs dat op funktioneel programmeren is geënt. Nu al voor het vierde jaar krijgt een klein aantal (30 à 40) Twentse eerstejaars studenten in de informatika zo'n kursus. Met ingang van het komende studiejaar (1991/92) is de kursus onderdeel van het kurrikulum voor alle eerstejaars informatika studenten.

Met dit artikel willen wij deze keuze motiveren. Tevens wordt een korte schets gegeven van het onderwijs, en de gang van

zaken rond de invoering ervan. De volgende vragen komen aan de orde:

- Wat is de doelstelling van inleidend programmeeronderwijs?
- Hoe is de kursus inhoudelijk opgezet?
- Hoe verhouden imperatief en funktioneel programmeren zich?
- Welke resultaten zijn in dit nieuwe programmeeronderwijs geboekt?

2 Doelstellingen van het programmeeronderwijs

Doel van het inleidend programmeeronderwijs is om de studenten vertrouwd te maken met het begrip algoritme. De student moet zelfstandig een algoritme kunnen ontwerpen. Na het met succes afronden van de kursus is de student in staat om

- een algoritme te ontwerpen, waarmee een praktisch probleem kan worden opgelost;
- te bewijzen dat een algoritme voldoet aan de specifikatie;
- een aantal standaard algoritmen te reproduceren en toe te passen (w.o. backtracking, combinatorische algoritmen op grafen, sorteren)

Hieraan zitten formele en praktische aspekten. Studenten moeten praktische problemen kunnen vertalen naar een algoritmische notatie, en tevens moeten zij bewijstechnieken kunnen toepassen om de korrektheid van een algoritme vast te stellen. Daarnaast moeten studenten vertrouwd raken met het ontwerp-aspekt. De hele kursus beslaat drie trimesters. Hierin wordt bijzonder veel waarde gehecht aan de samenhang tussen theorie en praktijk, die zijn weerslag vindt in een goed programma-ontwerp. In het navolgende komt onze visie op deze drie onderwerpen, *ontwerpen*, *theorie* en *praktijk*, in meer detail aan de orde.

2.1 Ontwerpen

Het ontwerpen van programmatuur valt uiteen in drie delen (zie Joosten 1989):

1. specifikatie
2. algoritmiek
3. implementatie

In het navolgende wordt elk van deze onderdelen besproken. Hierbij wordt een (klein) voorbeeld gehanteerd. Dit voorbeeld is niet essentieel, en kan bij een eerste lezing worden overgeslagen.

Om te beginnen dient het probleem volkomen duidelijk te zijn beschreven. Dit wordt aangeduid met de term 'specifikatie'. De specifikatie dient zo precies mogelijk aan te geven wat verwacht wordt van een programma. De specifikatie kan verschillende vormen aannemen. Men kan algebraïsch te werk gaan, of verzamelingen gebruiken. Voorbeeld:

$$\text{primes} = \{n \in \mathbb{N} \setminus \{0\} | \forall m \in \mathbb{N} \setminus \{0,1,n\} : \ n \bmod m \neq 0\}$$

De priemgetallen worden hier uitgedrukt als een verzameling van natuurlijke getallen groter dan 0, die geen delers hebben anders dan 1 en zichzelf.

Deze specifikatie heeft nog niets te maken met een programma. Het is een zuiver wiskundige en konventionele specifikatie. Als de specifikatie niet exekuteerbaar is, zoals in het voorbeeld, dan wordt een initiëel algoritme gemaakt dat zo dicht mogelijk bij de specifikatie staat. Het volgende voorbeeld is een direkte vertaling van bovenstaande specifikatie naar een (initiëel) algoritme:

$$\text{primes} = [n| \ n \in [1 \ . \ . \ ]; \ \text{and}[n \bmod m \neq 0| \ m \in [2 \ . \ . \ n-1]] \ ]$$

In dit voorbeeld is overgegaan naar een lijst (met vierkante haken), in plaats van een verzameling. Ook moet een testkonditie zo worden geformuleerd dat met een eindig aantal vergelijkingen kan worden volstaan. Elk getal n moet dus voldoen aan de konditie dat alle getallen tussen 1 en n geen delers zijn van n.

Dan komt de tweede fase aan bod. Alle aspekten van algoritmische aard horen hierin thuis. Het is van belang om de oplossing ook efficiënt uit te rekenen. Onder 'efficiënt' verstaan we een relatieve beoordeling van twee algoritmen ten opzichte van elkaar. Het ene algoritme is efficiënter dan het andere wanneer het minder berekeningen of geheugen vraagt dan het andere. Hierbij is de absolute rekentijd of

geheugenbeslag dus niet van belang. Voor het voorbeeld, priemgetallen, bestaat het volgende algoritme, bekend als de zeef van Eratosthenes:

```
primes = 1: sieve [2 . . ]
         where
              sieve (m:N) = m: sieve [n| n ∈ N; n mod m ≠ 0]
```

In deze beschrijving wordt een funktie sieve gebruikt, die een lijst van nader te bekijken getallen als argument heeft. Het kleinste getal van deze lijst is m, en de lijst N bevat de overige getallen. Het resultaat van deze funktie is een lijst van alle priemgetallen gevonden in het argument, $(m:N)$, te weten m zelf en alle priemgetallen uit N. Die laatste rij getallen is bepaald door sieve toe te passen op alle getallen uit N waarvan m geen deler is.

Als specifikatie is deze formulering wellicht minder duidelijk dan de vorige. Er is echter wel sprake van een efficiënter algoritme. Het nader uitwerken van een algoritme betekent in de regel dat het programma wat uitdijt. Als formulering van het bekende zeef algoritme is het echter een zeer korte en duidelijke beschrijving. De vaardigheid om algoritmen in elkaar over te voeren zonder de betekenis te veranderen wordt *algoritmiek* genoemd.

Het is in de praktijk meestal niet voldoende dat een probleem goed gespecificeerd is en uitgewerkt met behulp van efficiënte algoritmen. Vaak zijn er allerlei randvoorwaarden, zoals absolute rekentijd en/of geheugenbeslag. Ook kan het zijn dat de omgeving waarin de oplossing moet worden gerealiseerd beperkingen kent. De keuze van hardware en/of implementatietaal is meestal niet vrij. De derde fase van het programma ontwerp is bedoeld om aan dit soort voorwaarden te voldoen. Men kan een imperatieve programmeertaal kiezen voor de uiteindelijke implementatie. Nodig is dit echter niet; implementatie op silicium is ook denkbaar, en soms voldoet het reeds verkregen funktionele algoritme. Met name de aspekten die te maken hebben met control flow komen pas in de implementatie fase aan de orde. Deze stap is lang niet altijd triviaal. Bij een keuze voor parallelle control flow, bijvoorbeeld, kan het zijn dat een beduidende hoeveelheid ontwerparbeid nog in het verschiet ligt. Desondanks zijn twee belangrijke aspekten afgehandeld: het beschrijven van het probleem, en het kiezen van de juiste algoritmen.

2.2 Theorie

De theorie die studenten in deze kursus moeten leren komt globaal neer op het volgende lijstje:

- een funktionele programmeertaal (i.c. Miranda)
- de kernbegrippen uit funktioneel programmeren
- een imperatieve programmeertaal (i.c. Modula-2)
- de kernbegrippen uit imperatief programmeren
- programma transformaties en bewijstechnieken van funktionele programma's
- bewijstechnieken van imperatieve programma's

De kursus is theoretisch van aard, in de zin dat van de studenten een hoge mate van beheersing van formele technieken wordt gevraagd. Dit wordt dan ook uitgebreid geoefend. Hiermee wordt onder meer beoogd de studenten te oefenen in (wiskundig) abstrakt denken. De theorie in de kursus is echter niet het doel van studie, maar een hulpmiddel om betere programma's te ontwerpen.

Bij de programmatransformaties moet opgemerkt worden dat de theorie nog erg onvolledig is. Programmatransformaties in het domein van funktioneel programmeren bestaan wel, maar het is nog niet eenvoudig ze algemeen toe te passen. Transformaties van funktionele naar imperatieve programma's bestaan slechts op beperkte schaal. Hier wordt gewerkt met richtlijnen. Men kan bijvoorbeeld toewerken naar een rekursief funktioneel programma, en vervolgens een imperatief programma schrijven met hetzelfde rekursiepatroon.

2.3 Praktijk

Aan de praktische toepasbaarheid van de leerstof wordt veel aandacht besteed. Van elk theoretisch concept wordt de praktische relevantie getoond. Bijvoorbeeld: oneindige lijsten kunnen worden gebruikt om stromen van gebeurtenissen bij proceskommunikatie te modelleren. Een praktikum maakt dan ook in elk trimester deel uit van de kursus. Dat de theorie in dienst staat van de praktijk wordt bijvoorbeeld gedemonstreerd aan het praktikum van het eerste trimester. In de vierde week (!) van dit eerste trimester wordt aan de studenten gevraagd een programma te schrijven waarmee 'slimme' vragen over bestanden kunnen worden beantwoord. Een opdracht is bijvoorbeeld een bestand van voetbaluitslagen te ontwerpen, en vragen te beantwoorden zoals: welke clubs hebben een thuiswedstrijd verloren? Het feit dat studenten, vier weken na aanvang van de studie, al met dit soort problemen

werken motiveert enorm. Aan het eind van het eerste trimester hebben de studenten met verschillende soorten programmeerproblemen gewerkt.

3 Kursus opzet

Zoals al genoemd bestaat de hele kursus uit drie trimesters. In het navolgende wordt het programmeeronderwijs per trimester afzonderlijk belicht. Daarna komt het studiemateriaal aan de orde.

De werkvormen zijn in alle trimesters in grote lijnen gelijk: hoorkolleges (8×2 uur), werkkolleges (12×2 uur) en praktika (9×4 uur). De praktika zijn verplicht. Aan het einde van elk trimester wordt een tentamen afgenomen. Het derde trimester wordt tevens beoordeeld aan de hand van een individuele ontwerp-opdracht.

3.1 Funktioneel Programmeren

In eerste trimester leren de studenten het begrip *algoritme* kennen aan de hand van een funktionele programmeertaal. Aan het eind van dit trimester zijn de studenten in staat om:

- een algoritme in een funktionele taal op te schrijven;
- dit op verschillende manieren (bijvoorbeeld rekursief, met lijst-komprehensie of met standaardfunkties) te doen;
- een induktie bewijs te leveren bij een rekursieve definitie;
- een aantal praktische problemen te vertalen naar geschikte gegevensstrukturen met bijbehorende funkties.

De elementen van de programmeertaal worden voornamelijk in het werkkollege en praktikum aangeleerd, door ze simpelweg toe te passen. Het hoorkollege vervult voornamelijk een strukturerende rol. Voor de studenten levert dit vaak een ahagevoel op: stof die ze al hebben toegepast op een ad-hoc manier wordt op het hoorkollege in samenhang gepresenteerd.

De leerstof omvat:

- het funktiebegrip, funktiedefinities, kompositie van funkties;
- het lijstbegrip en standaardfunkties op lijsten, met name primitieven voor herhaling over lijsten;

- rekursie en induktie;
- interaktief programmeren;
- top-down beschrijving van de oplossing van een groter programmeerprobleem.

In het praktikum pakken de studenten, behalve kleine oefenopgaven, ook realistische problemen aan. De eerder genoemde opgave met bestanden wordt op praktikum sessies 4 en 5 gemaakt. Op sessie 6 wordt gevraagd een aantal wijzigingen aan te brengen in een bestaand programma, dat kalenders uitrekent en opmaakt bij een gegeven jaartal. In de 7e week worden rekursieve krommen (hilbert krommen e.d.) op het beeldscherm geplot. Op het 8e praktikum maken de studenten een interaktieve dialoog.

De praktikumopgaven zijn groot (qua inhoud) en er zijn veel opgaven, vergeleken met de opgaven uit het 'oude' eerste trimester, Het blijkt in het praktikum dat deze manier van programmeren veel produktiever is dan de manier uit de oude kursussen programmeren. Opmerkelijk is dat de praktijk van het programmeren heel snel aan bod komt, terwijl een kursus funktioneel programmeren toch als theoretisch wordt gekenmerkt.

In het werkkollege is intensief geoefend met meer theoretische zaken, zoals het leveren van een induktiebewijs. Uit tentamenresultaten blijkt dat studenten dit aspekt goed beheersen.

3.2 Imperatief Programmeren

In het tweede trimester wordt imperatief programmeren gedoceerd, waarbij gebruik wordt gemaakt van de kennis van funktioneel programmeren. Als programmeertaal is Modula-2 gekozen, ondermeer omdat deze taal data-abstraktie goed ondersteunt door een scheiding van specifikatie en implementatie. Specifieke kenmerken van de vakinhoud zijn:

- Rekursie is volledig geïntegreerd in de leerstof. Het wordt als vanzelfsprekend beschouwd, vanwege de voorkennis van funktioneel programmeren.
- Procedures als parameter worden van meet af aan toegepast, omdat studenten vertrouwd zijn met hogere orde funkties.
- Er wordt veel gebruik gemaakt van funktieprocedures, waarvan het resultaat ook van een samengesteld type mag zijn.

- Standaardoperaties voor abstrakte datatype, zoals lijsten en bomen worden aangeboden in herbruikbare modules. Deze standaardoperaties komen, voor zover mogelijk, overeen met de standaardoperaties waarvan de studenten in het eerste trimester gebruik hebben leren maken.

Met de invoering van abstrakties door (standaard) funkties worden studenten minder belast met details van implementaties. Hiermee lossen studenten op een hoger abstraktienivo hun programmeerproblemen op. Ze nemen deze stijl over bij zelf te definiëren funkties.

De volgende stap is de implementatie van deze standaardoperaties. Hierbij doet men implementatie keuzes, zoals een keuze tussen rekursie of iteratie. Om het verschil met funktioneel programmeren aan te geven wordt in het begin van het tweede trimester nadruk gelegd op het geheugenmodel bij de semantiek van het assignment-statement, aangevuld met toestandsbeweringen. Studenten leren om expliciet de control flow te beschouwen en beslissingen te nemen over de geheugenrepresentatie van zaken die in de funktionele wereld abstrakt bleven. Na afloop van dit trimester zijn studenten in staat om:

- een algoritme in een imperatieve taal te schrijven;
- een imperatief programma te ontwerpen, uitgaande van een funktioneel programma;
- korrektheid van programma's aan te tonen met invariant en toestandsbeweringen;
- een aantal praktische problemen te vertalen naar gegevensstrukturen en bijbehorende operaties.

3.3 Programmeermethoden

In het derde trimester komt het ontwerpen opnieuw aan bod. Het vak valt in twee delen uiteen. Het eerste deel is een behandeling van programmeertechnieken, standaardmethoden en probleemsoorten. In het tweede deel wordt een strak geregisseerde opdracht uitgevoerd, waarbij een ontwerpmethode centraal staat.

Het eerste deel valt uiteen in vier klusters, waarin steeds een probleemgebied of probleemsoort centraal staat.

1. backtracking en branch and bound;

2. patroonherkenning en ontleding;

3. bestanden en relaties;

4. grafen en relaties.

In elk kluster komen dezelfde aspekten aan de orde:

- standaardtechnieken en algoritmen;
- komplexiteitsoverwegingen;
- relatie tussen funktioneel- en imperatief programmeren;
- data abstraktie;
- dokumentatie.

Bij de opdracht wordt erop gelet dat de algoritmiek in het funktionele bereik wordt afgehandeld, en de uiteindelijke implementatie in een imperatieve taal wordt afgeleverd. De programmeermethode die in de opdracht gebruikt wordt is de methode zoals beschreven in sektie .

Na afloop van het trimester is een student in staat:

- een praktisch probleem in de vorm van een initiëel algoritme te specificeren;
- het initiële algoritme te transformeren naar een efficiënt algoritme;
- dit algoritme om te zetten naar een imperatieve implementatie;
- het ontwerp-proces te dokumenteren.

3.4 Studiemateriaal

Veel aandacht is besteed aan het ontwikkelen van goed studiemateriaal. Er is niet alleen gekeken naar het te kiezen leerboek, maar er is ook veel aandacht besteed aan alle mogelijke vormen van schriftelijke ondersteuning, zowel voor studenten als docenten. Als boek voor het eerste trimester wordt gebruikt "An introduction to functional programming" door Richard Bird en Philip Wadler (1988). In het tweede trimester wordt het boek van Koffman (1988) als naslagwerk gebruikt. Er zijn bundels met oefenopgaven samengesteld. Deze bundels bevatten ook uitwerkingen (zie Van der Hoeven 1988). Er zijn draaiboeken gemaakt van hoorkolleges, werkkolleges

en praktikumsessies, waarbij van iedere zitting de leerdoelen zijn vastgelegd. Deze draaiboeken zijn uitsluitend bestemd voor docenten. Ten behoeve van de hoorkolleges zijn transparanten gemaakt, die tevens in gebundelde vorm kunnen dienen als aanvullend diktaat. Voor de praktika zijn praktikum-handleidingen gemaakt. In het eerste trimester wordt geprogrammeerd in Miranda (zie Turner 1986) in een UNIX omgeving. Het tweede trimester maakt gebruik van Modula-2 in een MS-DOS omgeving (zie Jensen 1988). In het derde trimester worden beide gebruikt.

4 Imperatief vs. funktioneel programmeren

De kursussen in het programmeren die vandaag de dag op universiteiten worden gegeven zijn in de regel gebaseerd op imperatieve programmeertalen, zoals Pascal, C, Algol en dergelijke. Ook in Twente is de inleidende kursus in het programmeren altijd op imperatieve talen gebaseerd geweest, i.c. Algol 60, Algol 68, Pascal en Modula-2. In dit soort kursussen ligt de nadruk op de implementatie van algoritmen in de programmanotatie. Het is vaak moeilijk om expliciet aandacht te besteden aan de specifikatie- en ontwerpfase (zie Tromp 1989). Een formele aanpak van het ontwerpen van algoritmen wordt vaak onhaalbaar geacht. Docenten hebben dan ook vaak het gevoel dat de programmeren-kursus ontaardt in een programmeertaal-kursus. Minder bekend is hoe een inleidende kursus in het programmeren, gebaseerd op funktioneel programmeren, verloopt. Er zijn niet veel mensen die met deze benadering vertrouwd zijn, ondanks het feit dat de principes wel algemeen bekend zijn. Bekend is dat funktioneel programmeren elegant en wiskundig is, maar wat dat onderwijskundig voor gevolgen heeft is nauwelijks onderzocht. Daarnaast is er minder keuze aan beschikbare funktionele talen, die bovendien 'traag' zouden zijn. Voor sommigen is dat voldoende reden om funktioneel programmeren als onpraktisch van de hand te wijzen. Dit beeld willen wij wat nuanceren.

Allereerst moeten we konstateren dat er wel degelijk toepassingen bestaan van funktioneel programmeren. Klassiekers op dit gebied zijn Darlington (1982), Turner (1984) en Henderson (1986). Er zijn voldoende voorbeelden van serieuze toepassingen, bijvoorbeeld op het gebied van bedrijfssystemen (zie Jones 1984), digitale ontwerpen (zie Hill 1986), vertalerbouw (zie Johnsson 1987), gegevensbanken (zie Steutel 1988) en op het gebied van communicerende processen (zie Turner 1987). Funktioneel programmeren kan dus

bepaald niet als onpraktisch van de hand worden gewezen. De literatuur vermeldt tevens dat het programmeerwerk relatief weinig tijd in beslag neemt. Ook is er consensus over het feit dat algoritmen duidelijk, eenvoudig en leesbaar kunnen worden opgeschreven. Dit zijn eigenschappen die ook in het programmeeronderwijs van belang zijn. Het is daarom zinvol serieus te kijken naar funktioneel programmeren ten behoeve van het inleidend programmeeronderwijs.

Funktionele programmeertalen kunnen een goede basis vormen voor algoritmiek. De orthogonaliteit van een aantal funktionele talen heeft tot gevolg dat geen kostbare onderwijstijd en programmeerarbeid verloren gaat aan het afrekenen met uitzonderingen in de taal. Daarnaast geldt de substitutie eigenschap. Wanneer twee expressies aan elkaar gelijk zijn, dan mogen ze elkaar op een willekeurige plek vervangen. Deze eigenschap geldt niet voor imperatieve talen, vanwege de zij-effekten van statements. Een andere nuttige eigenschap is 'luie' of vraag-gestuurde (demand driven) evaluatie. Als gevolg hiervan hoeft de programmeur zich niet bezig te houden met de volgorde waarin berekeningen worden uitgevoerd. De volgorde van berekeningen wordt uitsluitend bepaald door de afhankelijkheid van gegevens (data dependency). Het is bijvoorbeeld niet van belang of men bij een breuk eerst de teller of eerst de noemer uitrekent.Het quotiënt verandert er niet door. In imperatieve talen houdt de programmeur zich wel degelijk bezig met control flow. Dat wil zeggen: de programmeur specificeert precies de berekeningsvolgorde. Een verkeerde keuze kan betekenen dat het programma in een oneindige lus terecht komt, of dat men per ongeluk buiten de grenzen van een array indiceert. Dat is vervelend, omdat control flow niets te maken heeft met het op te lossen probleem, maar met de volgorde waarin de berekeningen plaatsvinden. In deze zin is control flow een implementatie detail.

Samenvattend kunnen we konkluderen:

- In tegenstelling tot wat sommigen beweren, is funktioneel programmeren wel degelijk praktisch toepasbaar.
- De eigenschappen van funktionele talen, zoals orthogonaal taalontwerp en de substitutie-eigenschap, zijn belangrijk tijdens het ontwerpen van algoritmen.
- Abstraktie wordt uitstekend ondersteund, mede door hogere orde funkties en polymorfie.

- Luie evaluatie betekent voor de programmeur dat control flow tijdens het programmeren buiten beschouwing mag blijven.

5 Invoering en evaluatie

Vanaf augustus 1987 is de opzet van dit programmeeronderwijs ontwikkeld en aangeboden aan eerstejaars studenten in de fakulteit Informatika van de Universiteit Twente. In 1987/88 is begonnen met een groep van 24 studenten die zich als vrijwilliger hadden opgegeven. Deze groep bleek uit gemiddeld betere studenten te bestaan. In 1988/89 werd gewerkt met twee representatief samengestelde groepen studenten (op basis van VWO examencijfers wis- en natuurkunde, totaal 48 studenten). In 1989/90 volgden twee willekeurig gekozen groepen (totaal 40 studenten) dit nieuwe programmeeronderwijs. Ook in het seizoen 1990/91 is deze aanpak gevolgd. Vanaf het komende studiejaar (1991/92) volgen alle groepen dit onderwijs.

De invoering van vernieuwd programmeeronderwijs op basis van funktionele talen brengt een aantal problemen met zich mee. Eén ervan is dat er weinig studiemateriaal elders ontwikkeld is (vergelijk Bailes 1989, Savitch 1989). Er bestaat weinig materiaal om in een kursus te gebruiken. Alles moet in principe opnieuw worden overdacht. Zo'n proces kost veel tijd en inspanning, maar de lange duur heeft ook een aantal voordelen. De docenten krijgen de tijd om aan het idee te wennen, en zich het funktioneel programmeren eigen te maken. Een ander voordeel is dat er tijd ontstaat om metingen te doen aan het nieuwe onderwijs.

Er is uitgebreid aandacht besteed aan de evaluatie van het programmeeronderwijs in samenwerking met het Onderwijskundig Centrum, ondermeer door regelmatige gesprekken van docenten en studenten en door na afloop van elk trimester een uitgebreide enquête te houden over het betreffende vak, over het studiemateriaal, de werkvormen, tijdsbesteding, de vakinhoud, knelpunten, de moeilijkheidsgraad, enz. Bovendien werd door de studenten bijgehouden hoeveel tijd ze aan alle vakken besteedden (tijdschrijven), zowel in de experimentgroepen als in de kontrolegroepen. Verder werden de studieresultaten nauwkeurig gevolgd, ondermeer de korrelatie met andere eerstejaarsvakken (analyse en diskrete wiskunde) (zie van den Berg & Pilot 1989).

Op basis van de evaluaties is het onderwijs bijgesteld. Opvallend waren bijvoorbeeld de moeilijkheden in het eerste

jaar met het onderwerp typering. Door een andere aanpak is dit onderwerp volledig verdwenen uit de top-tien van moeilijke onderwerpen. De studenten vinden het vak in verhouding niet erg moeilijk. Het engelstalige en vrij formele studieboek wordt niet als een probleem ervaren. De tijdsbesteding is goed in overeenstemming met de norm (en ligt gunstiger dan voor de oude vakken programmeren). Over het algemeen vinden de studenten het een plezierig en nuttig vak.

Er zijn ook problemen met dit onderwijs. Met name de gebrekkige follow-up in de periode na de propaedeuse wordt door studenten als problematisch ervaren.

Parallel aan de evaluatie van het onderwijs is ook onderzoek verricht. In het eerste experimentjaar (seizoen '88/'89) zijn uitvoerige analyses gemaakt van hardopdenkprotokollen van studenten en docenten (zie Massink 1988). Ook zijn analyses gemaakt van fouten en miskoncepties van studenten in huiswerk- en tentamenuitwerkingen. In het tweede experimentjaar is een toets gehouden voor de groepen funktioneel programmeren en twee kontrolegroepen. Met behulp van deze toets is de programmeervaardigheid onderzocht (zie van den Berg, Massink & Pilot 1989). De belangrijkste verschillen werden gevonden bij het ontwerpen van een programma: de studenten funktioneel programmeren hadden signifikant vaker een uitwerking overeenkomstig het ontwerp dan de kontrolegroepen. Voor de vergelijking van de oplossingen (in Miranda voor de FP-groepen en in Pascal voor de kontrolegroepen) werd hiertoe een abstraktie van de oplossing gemaakt in de vorm van een programmagraaf.

Naast de harde gemeten resultaten bestaan er natuurlijk ook nog de nodige indrukken en ervaringen. Zo bleken de studenten het hogere abstraktie nivo in deze kursus heel goed aan te kunnen. Ook is er nagedacht over de rol van deze kursus in de selektieve propaedeuse. Bij de konventionele kursus had men het gevoel dat studenten met veel handigheid achter het toetsenbord (de zogenaamde hackers) relatief weinig moeite hadden met het programmeeronderwijs. In de nieuwe opzet vindt de selektie veel meer plaats op het vermogen te werken met abstrakties.

6 Afsluiting

Na vier jaar experimenteren heeft de faculteit Informatica besloten om het programmeeronderwijs voor haar eerstejaars studenten te herzien volgens de in dit artikel geschetste

lijn. Op basis van de evaluaties is vastgesteld dat de kwaliteit van het inleidend programmeeronderwijs hierdoor kan worden verbeterd, omdat in de nieuwe opzet

- studenten beter leren omgaan met abstraktie als ontwerphulpmiddel;
- studenten hun probleem formeel beter kunnen beschrijven;
- studenten een duidelijke ontwerpstrategie is aangeleerd;
- studenten formeler leren omgaan met het ontwerp van programmatuur.

Ook is vastgesteld dat na afloop van de kursus de studenten zich *niet* minder imperatieve programmeerkennis hebben eigengemaakt. Dit onderwijs vergt dan ook geen aanpassingen aan de rest van het curriculum.

Gebruikte literatuur

Bailes, P.A. & Salzman, E.J. (1989) *A Proposal for a Bachelor's Degree Program in Software Engineering*, Software Engineering Education. Lecture Notes in Computer Science 376, pp. 90-108, Springer-Verlag, Berlijn, ISBN 0-387-97090-8.

Berg, K.G. van den & Pilot, A. (1989) *Functioneel programmeren 1987/88. Evaluatie van een onderwijsexperiment*, Universiteit Twente, Enschede, Memorandum INF-89-6.

Berg, K.G. van den, Massink, M. & Pilot, A. (1989) *Experimentele vergelijking van het leren programmeren ondersteund door een functionele versus een imperatieve programmeertaal*, Voordracht Onderwijs Research Dagen, Leiden.

Bird, R.S. & Wadler, P. (1988) *Introduction to Functional Programming*, Prentice Hall, New York, ISBN 0-13-484197-2.

Darlington, J. et al. (1982) *Functional Programming and its Application*, Cambridge University Press.

Henderson, P. (1986) *Functional Programming, Formal Specification, and Rapid Prototyping*, IEEE Transactions on software engineering, Vol. SE-12. no. 2, pp 241-250.

Hill, S.A. (1986) *Simulating Digital Circuits in Miranda* UKC Computing Laboratory Report No 42, University of Kent at Canterbury.

Hoeven, G.F. van der (1988) *Studiemateriaal bij het experiment Functioneel Programmeren*, Universiteit Twente, Enschede, Memoramdum INF-88-30.

Jensen & Partners International (1988) *TopSpeed™ Modula-2*.

Jones, S.B. (1984) *A range of operating systems written in a purely functional style*, Technical Report 16, Univ. of Stirling, Scotland.

Johnsson, T. (1987) *Attribute Grammars as a Functional Programming Paradigm*, in: Kahn, G. (ed), Functional Programming Languages and Computer Architecture, Proc. Portland, Oregon, LNCS 274, pp. 154-173, Springer-Verlag, Berlin, ISBN 3-540-18317-5.

Joosten, S.M.M. (1989) *The use of functional programming in software development*, proefschrift, Universiteit Twente, fakulteit Informatika, ISBN 90 9002729 7.

Koffman, E.B. (1988) *Problem Solving and Structured Programming in Modula-2*, Addison-Wesley, ISBN 0-201-07828-7.

Massink, M., Van den Berg, K.G., Van Berne, H. & Pilot, A. (1988) *Oplossingsprocessen bij leren Functioneel Programmeren*, Universiteit Twente, Enschede, Memorandum INF-88-27.

Massink, M.(1988) *Onderwijsleerprocessen bij Functioneel Programmeren*, Verslag Doctoraalopdracht, Katholieke Universiteit Nijmegen/Universiteit Twente, Enschede.

Savitch, W.J. (1989) *Functional Programming in Pascal?*, Journal of Pascal, Ada & Modula-2, Vol. 8, No 5, pp. 35-41.

Steutel, D. & van Zonneveld, E.P. (1988) *Prototyping a Relational DBMS in a Functional Language* Verslag Doctoraalopdracht, Universiteit Twente, fakulteit Informatika.

Tromp, Th.J.M. (1989) *The Acquisition of Expertise in Computer Programming*, Proefschrift, Universiteit van Amsterdam.

Turner, D.A. (1984) *Functional programs as executable specifications*, Phil. Trans. R. Soc. Lond. A312, pp 363-388.

Turner, D.A. (1986) *An Overview of Miranda*, Sigplan Notices Vol 21, nr 12, pp 158-166.

Turner, D.A. (1987) *Functional Programming and Communicating Processes*, in: Parle 87, Parallel Architectures and Languages Europe, (eds. J.W. de Bakker e.a.), pp.54-74, Springer-Verlag Berlin, LNCS 258, ISBN 3-540-17943-7.