De opkomst van Recursie

Een pluralistische kijk

2×7.5 ECTS

Ruben Groot Nibbelink
3345246

Bas Peters
3248895

K. van Oudheusden

Cognitieve Kunstmatige Intelligentie
Departement Geesteswetenschappen

21 augustus 2015

Abstract

In deze scriptie beschrijven we hoe het begrip *recursie* door de jaren heen bestudeerd, veranderd en geïmplementeerd is in computerprogramma's, beginnende eind jaren vijftig van de vorige eeuw. Hiertoe bestuderen we de visie van verscheidene actoren op specifieke momenten in de geschiedenis, namelijk E.W. Dijkstra, F.L. Bauer en K. Samelson rond 1959 en G. van den Hove en C. Rinderknecht omstreeks 2014. We contrasteren de verschillende zienswijzen, kijken hoe deze overeen komen en verschillen en plaatsen de visies in historische context. Aan de hand van implementaties in de `DRAMA`-programmeertaal worden de verschillen in de aanpakken van Dijkstra en Bauer & Samelson geïllustreerd.

Keywords: Recursie, Stack, Dijkstra, Bauer, Samelson, Subroutine, DRAMA, Van den Hove, Rinderknecht, Duplicatie, ALGOL-60.

Inhoudsopgave

[S]: geschreven door beide studenten.
[B]: geschreven door Bas Peters.
[R]: geschreven door Ruben Groot Nibbelink.

1 [S] Inleiding

Binnen de hedendaagse computerwetenschappen is recursie een bekend begrip. Het wordt veel toegepast in het oplossen van allerlei computerproblemen die betrekking hebben op recursief gedefinieerde datastructuren, zoals lijsten, *stacks* en zoekbomen. Toch lijkt een antwoord op de vraag "Wat is recursie?" nog niet zo makkelijk te formuleren. In deze scriptie benaderen we deze vraag door deze op te delen in deelvragen van de vorm: "Wat betekende recursie voor actor X in jaar Y?"

$$
\begin{aligned}
\textit{Recursie} \quad = \quad & \textit{recursie volgens Dijkstra rond 1960} \quad + \\
& \textit{recursie volgens Bauer en Samelson rond 1960} \quad + \\
& \textit{recursie volgens andere actoren rond 1960} \quad + \\
& \textit{recursie volgens Van den Hove rond 2014} \quad + \\
& \textit{recursie volgens Rinderknecht rond 2014} \quad + \\
& \vdots
\end{aligned}
$$

We gaan op zoek naar de overeenkomsten en verschillen van deze zienswijzen, en hoe de visies van vroeger passen binnen de theorieën van nu.

We vergelijken de opvattingen van Edsger Dijkstra, Friedrich Bauer en Klaus Samelson tijdens de ontwikkeling van de ALGOL-60 programmeertaal rond 1960. Allereerst verplaatsen we ons in Dijkstra; wat betekende recursie voor hem en wat kunnen we concluderen over zijn opvattingen als we kijken naar de rol van recursie in de hedendaagse informatica? Vervolgens bestuderen we het werk van Bauer en Samelson, wiens benadering van recursie in deze tijdsgeest haaks op die van Dijkstra stond. Wat vonden zij van recursie en welke rol kreeg recursie toebedeeld binnen hun implementatie van ALGOL-60? Aan het einde van deze scriptie wordt een vergelijking gegeven van deze twee opvattingen door deze in de assembleertaal `DRAMA` te implementeren. Er wordt onderzocht hoe de verschillen op het niveau van machinecode tot uiting komen.

Ook de standpunten van verscheidene andere actoren in dezelfde tijdsgeest — Edgar Irons, Wallace Feurzeig en Henry Gordon Rice — worden onderzocht en kort toegelicht, en de contrasten met de theorieën van Bauer, Samelson en Dijkstra geduid.

De afgelopen jaren hebben Gauthier van den Hove en Christian Rinderknecht theorieën over recursie ontwikkeld, en in dit onderzoek contrasteren we deze twee theorieën. Daarnaast proberen we het werk van Dijkstra en Bauer & Samelson binnen deze hedendaagse kaders te plaatsen.

Omdat we recursie op een pluralistische manier benaderd hebben, zal er ook meer begrip ontstaan tussen de verschillende vakgebieden onderling voor de manier waarop met recursie wordt omgegaan. Wanneer men deze scriptie heeft gelezen zal er een beter beeld zijn ontstaan van de verschillende standpunten ten opzichte van recursie.

2 [B] Edsger W. Dijkstra

Met de ontwikkeling van de programmeerbare computer halverwege de twintigste eeuw ontstond ook de ambitie een universele programmeertaal te ontwikkelen die op verschillende computers gebruikt kon worden. De ontwikkeling van één van deze talen, ALGOL-60 genaamd, bracht uiteenlopende ideeën teweeg over de rol die recursie binnen deze taal moest worden toebedeeld. In dit hoofdstuk kijken we naar de Nederlandse fysicus Edsger W. Dijkstra, die in de jaren vijftig en zestig een belangrijke rol speelde in de introductie van recursie binnen het programmeren. In de eerste paragrafen wordt een beeld geschetst van de tijdsgeest waarin dit zich afspeelde en wordt eveneens duidelijk waarom Dijkstra's ideeën over ALGOL-60 in deze tijd zo controversieel waren. Vervolgens wordt aan de hand van enkele voorbeelden uitgelegd hoe Dijkstra het gebruik van de *stack* voor zich zag. Wat zijn de verschillen tussen Dijkstra's implementatie uit 1960 en de *stack* zoals die vandaag de dag gebruikt wordt? En waarom vond hij het zo belangrijk dat zijn implementatie van ALGOL (in tegenstelling tot implementaties van anderen) wel in staat moest zijn met recursie om te gaan? Door zijn motieven en ideeën te bestuderen, zal inzicht worden verschaft omtrent deze vragen en zal duidelijk worden wat recursie rond 1960 precies voor Dijkstra betekende.

2.1 Historische context

Tijdens de tweede wereldoorlog werd van wetenschappers en technologen verwacht dat zij zich volledig inzetten voor hun vaderland, hetgeen zorgde voor een enorme impuls aan technologische ontwikkelingen. Na de oorlog zette deze stroomversnelling zich voort. ALGOL-60 moest de universele hogere-orde programmeertaal worden en compilers werden ontwikkeld om deze taal om te zetten naar de "natuurlijke taal" van de computer: machinetaal.

Computers waren in de jaren vijftig schaars, en bovendien erg beperkt vergeleken met de computers die we vandaag de dag gewend zijn. De meeste programmeurs (zoals Christopher Strachey, Friedrich Bauer en Klaus Samelson) namen dan ook de machine als uitgangspunt bij het ontwikkelen van hun compiler. De reden hiervoor was vooral van economische aard, want men beschikte meestal niet over de financiële middelen om nieuwe computers te bouwen. Het motto was dan ook: hoe efficiënter de code van een programma, hoe groter de kans op resultaat. Om dit voor elkaar te krijgen moesten aan ALGOL-60 restricties worden opgelegd die ervoor zorgden dat de machine geen overbodig werk hoefde te doen. Er werd gekeken naar waartoe de computer in staat was en op basis daarvan moest de programmeertaal ontwikkeld worden.

2.2 Dichotomie

Dijkstra, op dat moment werkzaam bij het Mathematisch Centrum Amsterdam, bekeek ALGOL-60 juist vanuit een meer linguïstisch standpunt. Zijn focus lag vooral op de algemeenheid, elegantie en leesbaarheid van het programma. In tegenstelling tot eerdergenoemde Duitsers[1] vond Dijkstra dat een ALGOL-programma zoveel mogelijk moest lijken op de natuurlijke taal met dus zo min mogelijk (onnodige) restricties. Dat de computers in die tijd bepaalde aspecten van de taal nog niet aankonden, zag hij als een probleem van tijdelijke aard; de computers zouden met de tijd immers steeds sneller en krachtiger worden. In een paper over een ALGOL translator (die hij overigens samen met Jaap Zonneveld ontwierp) schreef hij over deze principekwestie het volgende:

> The reason of principle is that as a scientific Institute we would rather devote our time to the development of a programming technique which we expect to be realised in the near future than a technique for which this is not so. [9, p.2]

Deze insteek staat dus haaks op die van Bauer en Samelson, die ALGOL-60 juist wilden aanpassen aan de machines van de jaren vijftig. Ook geheugentechnisch leidde dit tot een interessant contrast. Waar Bauer en Samelson tijdens het compileren van het programma al precies wilden weten hoeveel geheugen het programma in zou gaan nemen, werd dit op Dijkstra's manier pas (dynamisch) *at run-time* vastgesteld. Bovendien zagen de Duitsers weinig nut in recursieve procedures, terwijl Dijkstra ervan overtuigd was dat recursieve constructies (net als in de wiskunde en taalkunde) ook in de informatica thuishoorden. Deze tegenovergestelde uitgangspunten worden door Edgar G. Daylight omschreven als een "dichotomie tussen specialisatie en generalisatie" [5, p.4] en deze begrippen worden in zijn artikel 'Dijkstra's Rallying Cry for Generalization' als volgt toegelicht:

> Specialization refers to language restrictions, static (compile-time) solutions, and the exploitation of machine-specific facilities – in the interest of efficiency. Generalization refers to general language constructs, dynamic (run-time) solutions, and machine-independent language design – in the interest of correctness and reliability. [5, p.4]

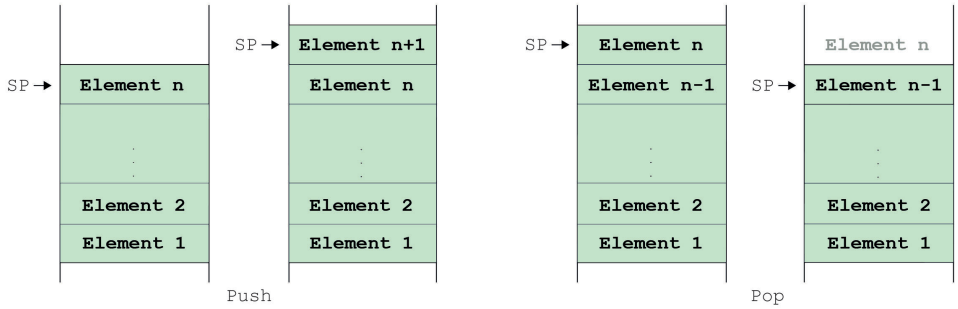
In zijn artikel 'Recursive programming' beschreef Dijkstra een programmastructuur waarbinnen het mogelijk is een *subroutine* (Dijkstra's benaming voor een procedure) aan te roepen, terwijl één of meerdere instanties van deze subroutine nog niet zijn beëindigd. Hierbij benadrukte hij het feit dat

[1] Strachey was overigens een Brit, maar hij zal in dit onderzoek niet uitgebreid aan bod komen.

er wat betreft de aangeroepen subroutine geen restricties opgelegd worden, hetgeen impliciet betekent dat dit net zo goed een “incarnatie” [8, p.317] kan zijn van de subroutine waarbinnen deze aangeroepen wordt. In de volgende alinea’s zal duidelijk worden hoe Dijkstra deze dynamische structuur precies voor ogen had en waarom algemeenheid hierin zo’n belangrijke rol speelde.

2.3 Stack

Dijkstra deelde de computer voor het gemak op in twee delen; het geheugen (waarin de *stack* opgeslagen is) en de *arithmetic unit* (het gedeelte van de machine waarin de berekeningen worden uitgevoerd). Eind jaren vijftig was men reeds bekend met de *stack*, een dynamische manier om elementen op te slaan in het geheugen van de computer. In het kort komt deze structuur neer op twee methoden: met de *push*-methode kan men elementen op de *stack* zetten en met de *pop*-methode worden de elementen van de *stack* afgehaald (zie figuur 1). In de processor van de computer wordt een speciaal register gereserveerd voor de *stack pointer*, deze wijst ten alle tijden naar het bovenste element van de *stack*. Dit is ook het enige element dat toegankelijk is, het is dus niet mogelijk elementen dieper in de *stack* te gebruiken (dit principe wordt *LIFO*, of *Last In First Out* genoemd). Bij het pushen van een element, wordt de waarde van de *stack pointer* met één verhoogd en het element wordt zodoende op de eerste vrije plek in het geheugen opgeslagen. Wanneer een element gepopt wordt, hoeft slechts de waarde van de *stack pointer* met één verlaagd te worden zodat dit element niet meer tot de *stack* behoort. In appendix A staat een uitgebreide omschrijving van hoe de *stack* precies werkt.



Figuur 1: De push- en popmethode van een *stack*.

Dijkstra had zijn eigen opvatting over de *stack* en hoe deze tijdens uitvoering van een programma gebruikt zou moeten worden. Het *pushen* van numerieke informatie werkt vrij eenvoudig, de waarde kan in dit geval rechtstreeks uit het geheugen worden gehaald en op de *stack* worden geplaatst. Wordt er in het programma echter een subroutine aangeroepen, dan beschrijft Dijkstra de ontwikkeling van de *stack* als volgt:

1. De eerste elementen die op de *stack* worden geplaatst, zijn de parameters van de aangeroepen subroutine. Om na voltooiing van de subroutine door te kunnen gaan met het programma worden de eerste van deze parameters gebruikt voor de *link*, hierin bevinden zich de gegevens die nodig zijn om de samenwerking tussen het geheugen en de *arithmetic unit* goed te laten verlopen. Hierin staat bijvoorbeeld het *return adress* (het adres in het geheugen waarnaar teruggekeerd moet worden na voltooiing van een procedure) en andere informatie over de toestand van de *arithmetic unit*. Na de *link* worden eventuele andere parameters opgeslagen, zoals functieargumenten. Het aantal parameters is constant en ligt bij aanroep van de subroutine dus al vast.

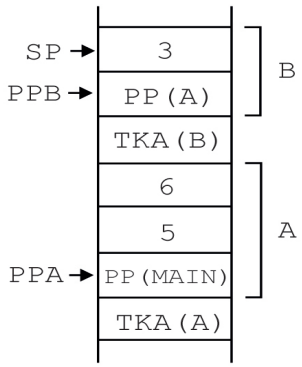
2. Bovenop de parameters komen de *local variables*, dit zijn variabelen die alleen in de subroutine zelf gebruikt mogen worden. Deze worden dan ook direct van de *stack* verwijderd zodra de subroutine beëindigd is. Net als het aantal parameters staat het aantal lokale variabelen al vast op het moment dat de subroutine aangeroepen wordt. Hierdoor kan tijdens de uitvoering verwezen worden naar variabelen die "dieper" in de *stack* liggen opgeslagen. In de volgende paragraaf zal uitgelegd worden waarom dit nodig is en hoe dit precies bewerkstelligd wordt.

3. Tot slot worden de *most anonymous intermediate results* op de *stack* geplaatst. Binnen een subroutine kunnen natuurlijk nog andere subroutines aangeroepen worden en daarom moeten de tussenresultaten bewaard worden totdat de functie helemaal voltooid is. Ook tussenresultaten van aritmetische expressies moeten overigens op de *stack* opgeslagen worden. Deze tussenresultaten komen dus bovenop de laatst toegevoegde lokale variabele.

2.4 Random access

Het ontstaan van ALGOL-60 betekende ook de introductie van de *block*-structuur in de programmeerwereld. Een *block* is in feite een stuk code dat bestaat uit één of meerdere statements en declaraties. Binnen zo'n *block* kunnen verschillende variabelen en procedures gedeclareerd worden en het begin en einde van het *block* geven syntactisch gezien aan binnen welk bereik, of *lexical scope* deze gebruikt kunnen worden. Wordt er binnen zo'n *block* een procedure aangeroepen, dan ontstaat er een genest *block* die ook binnen deze *scope* valt (aangezien we het over aanroepen hebben, spreken we in dit geval van een *dynamic scope*). Variabelen gedeclareerd in het buitenste *block* zijn hierdoor dus ook zichtbaar in het geneste *block*.

Volgens het *Last In First Out*-principe, is alleen het bovenste element van de *stack* toegankelijk. Dit is problematisch wanneer men binnen een procedure (bijvoorbeeld subroutine A) een andere procedure (subroutine B) wil



Figuur 2: Lokale variabelen in een *stack*.

aanroepen (in het geval van recursie is dit overigens één en dezelfde subroutine). Subroutine B komt dan logischerwijs boven subroutine A te staan in het geheugen. Om toegang te krijgen tot een lokale variabele van A, zouden dan eerst alle elementen die boven deze variabele staan van de *stack* verwijderd moeten worden. Deze zouden allemaal opgeslagen moeten worden in registers, om ze daarna ook weer terug te kunnen zetten. Om dit probleem te omzeilen bedacht Dijkstra een manier om *random access* te krijgen tot elementen dieper in de *stack*. Bij aanvang van elke subroutine-aanroep wordt daarom een *parameter pointer* in een van de registers opgeslagen, die verwijst naar de positie in het geheugen (het adres) waar dat *block* begint. Deze *parameter pointer* wordt vervolgens meegegeven aan het geneste *block* (*block* B in ons voorbeeld). Wanneer in *block* B een lokale variabele uit *block* A nodig is, dan kan de positie van deze variabele aan de hand van de *parameter pointer* van *block* A berekend worden. Bij voltooiing van *block* B komt de *stack pointer* op de plaats van de *parameter pointer* van B (dit is dus de plek waar de *stack pointer* stond voordat subroutine B werd aangeroepen) en wordt *block* B dus van de *stack* verwijderd. Deze methode om toegang te krijgen tot lokale variabelen uit andere subroutines is overigens ook van toepassing op globale variabelen. Omdat via elke *parameter pointer* ook de waarde van de *parameter pointer* van de omvattende (aanroepende) subroutine achterhaald kan worden, kan door middel van een aaneenschakeling van *parameter pointers* toegang verkregen worden tot de globale variabelen in de functie.

Ter illustratie is een voorbeeld uitgewerkt waarbij subroutine A twee lokale variabelen (de waarden 5 en 6) bevat (zie figuur 2). Binnen deze subroutine wordt subroutine B aangeroepen, die zelf de waarde 3 als lokale variabele heeft. De terugkeeradressen (TKA) zijn noodzakelijk om na een subroutine-aanroep terug te kunnen springen naar de juiste regel in het programma. TKA(A) verwijst dus naar het adres waar het programma naar moet terugkeren, zodra subroutine A voltooid is. In sectie B.8 is te

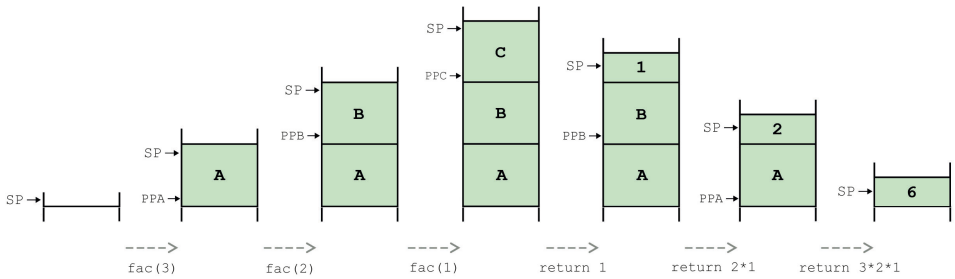
zien hoe subroutine B gebruik kan maken van de lokale variabelen in A door de *parameter pointer* van A te gebruiken die in de *link* van B is opgeslagen.

2.5 Recursieve procedures

Laten we nu kijken wat er volgens de Dijkstra-implementatie precies in de *stack* gebeurt, wanneer er een recursieve procedure aangeroepen wordt. Als voorbeeld nemen we de functie `fac(n)` die de faculteit van een getal n berekent. In pseudocode van een hogere-orde imperatieve programmeertaal ziet de functie er als volgt uit:

```
fac(n) {
    if (n<=1) return 1;
    else return n * fac(n-1);
}
```

In figuur 3 wordt als voorbeeld de aanroep `fac(3)` gebruikt. Omdat de details ten gunste van overzichtelijkheid in de figuur zijn weggelaten, zullen deze kort worden toegelicht. Bij elke subroutine-aanroep wordt een *block* voor deze subroutine (in de afbeelding is dit *block* A voor `fac(3)`, *block* B voor `fac(2)` en *block* C voor `fac(1)`) op de *stack* geplaatst. De onderste elementen van zo'n *block* worden gebruikt voor de *link*. Hierin bevindt zich onder andere een *return adress*, zodat de computer weet naar welke regel in het programma teruggesprongen moet worden na voltooiing van deze subroutine. Daarnaast wordt de toestand van de *arithmetic unit* in de *link* opgeslagen, net als de *parameter pointer* van het *block* en de *parameter pointer* van het *block* van waaruit de subroutine aangeroepen wordt. De *parameter pointer* van het *block* zelf wordt gebruikt om lokale variabelen binnen dat *block* te gebruiken. De *parameter pointer* van het omvattende *block* wordt gebruikt om toegang te kunnen krijgen tot globale variabelen en eventueel lokale variabelen uit andere subroutines die op dat moment actief zijn. Hoe dit precies in zijn werk gaat is beschreven onder het kopje *random access*. Bovenop deze *link* wordt in elk *block* de functiewaarde als parameter



Figuur 3: Grafische representatie van de *stack* bij aanroep van `fac(3)`.

opgeslagen en daar weer bovenop eventuele lokale variabelen die in dat *block* gedeclareerd zijn.[2] Wanneer de waarde van `fac(3)` berekend is, wordt het *return adress* uit de *link* van het eerste *block* gebruikt om met de waarde 6 terug te keren naar het juiste regelnummer van het *main*-programma. De *stack* wordt na voltooiing van de gehele procedure `fac(3)` dus weer precies hetzelfde achtergelaten als voorafgaand aan de aanroep.

2.6 Verschillen tussen Dijkstra en DRAMA

In hoofdstuk 7 gebruiken we de assembleertaal `DRAMA`, dat staat voor **D**ecimale **R**eken**A**utomaat met **M**eerdere **A**ccumulatoren. We hebben hiervoor gekozen omdat we op deze manier per instructie kunnen kijken wat er precies in het geheugen en in de registers van de computer verandert. Onder de subsectie *D - Faculteit* (programma 14) is een uitleg te vinden van dezelfde procedure-aanroep (`fac(3)`), maar dan uitgevoerd in `DRAMA`.

Als we deze twee implementaties naast elkaar leggen dan zien we dat er vrij veel overeenkomsten zijn. De manier waarop Dijkstra in 1960 zijn *stack* implementeerde, lijkt veel op de *stack* zoals die vandaag de dag gebruikt wordt. Toch zijn ook enkele interessante verschillen op te merken. Ten eerste is er in `DRAMA` de mogelijkheid om de `SBR`-instructie te gebruiken. Hiermee wordt in één stap een subroutine aangeroepen, het terugkeeradres op de *stack* geplaatst en de *parameter pointer* verplaatst. Omdat de waarde van de actuele *parameter pointer* in de `DRAMA`-simulator opgeslagen wordt in een van de registers (`R8`) hoeft deze niet ook nog eens in de *link* van het betreffende *block* opgeslagen te worden. Ook het terugkeeradres hoeft niet in de *link* opgeslagen te worden. Hierdoor zal de *stack* niet zulke extreme proporties aannemen als bij Dijkstra het geval was. Om de Dijkstra-implementatie trouw te blijven en zo algemeen mogelijk te werk te gaan, worden in onze implementaties onnodig veel *stack-accesses* gebruikt. Zo wordt er bijvoorbeeld regelmatig een element op de *stack* gezet, om deze er in de volgende instructie alweer af te halen. Op de manier waarop de *stack* tegenwoordig gebruikt wordt, wordt dit vermeden waardoor deze programma's veel efficiënter zijn. De conservatieve aanpak van Dijkstra is ook terug te zien in de manier waarop subroutines aangeroepen worden. Omdat Dijkstra elke procedure behandelde als een (potentieel) recursieve procedure, werd bij voorbaat allerlei informatie in de *link* opgeslagen die misschien helemaal niet nodig zou zijn bij uitvoering van het programma. In `DRAMA` is het terugkeeradres het enige dat automatisch op de *stack* wordt gezet bij aanroep van een subroutine. In dit contrast wordt mooi duidelijk hoe belangrijk recursie voor Dijkstra was, en hoe dit ten koste ging van de efficiëntie.

[2] In dit voorbeeld wordt geen gebruik van lokale variabelen gemaakt.

3 [R] Friedrich L. Bauer & Klaus Samelson

In deze sectie bespreken we de zienswijze van Friedrich Bauer en Klaus Samelson op programmeren in de tijd dat de specificatie van ALGOL-60 werd vastgesteld, rond 1960. Aan de hand van het artikel 'Sequentielle Formelübersetzung' [15], dat later in het Engels werd uitgebracht als 'Sequential Formula Translation' [16] ontrafelen we hun gedachten over de *stack*, het aanroepen van subroutines en hoe recursie binnen deze visie past.

3.1 Sequential Formula Translation

Een belangrijk onderscheid dat er gemaakt moet worden wanneer we de ideeën van Dijkstra willen vergelijken met die van Bauer & Samelson, is het verschil tussen *compile-time* en *run-time*. Waar Dijkstra er voor koos om zoveel mogelijk bewerkingen van het programma *at run-time* — tijdens de uitvoer van het programma — te verrichten, kozen Bauer & Samelson ervoor — zoals de meeste programmeurs halverwege de vorige eeuw — om zoveel mogelijk zaken *at compile-time* al zo te verwerken dat het object-programma[3] dat overbleef snel en efficiënt was, zodat het minder tijd kostte om dit te laten interpreteren door een *interpreter* (het uitvoeren). Een begrijpelijk idee, daar computers in die tijd nog niet erg snel waren en het compileren van de programmatekst maar één keer gedaan hoefde te worden, waarna het gegenereerde object-programma meerdere malen kon worden uitgevoerd.

In 'Sequential Formula Translation' beschrijven Bauer & Samelson een techniek om, met behulp van de *stack*, een aritmetische expressie (*formula*) uitgedrukt in de ALGOL-taal om te zetten (*translation*) naar objectcode, door middel van een stappenplan (*sequence*). Deze techniek werd gebuikt om snelle objectprogramma's te genereren waarna de computer *at run-time* alleen nog maar stap voor stap het recept af hoefde te lopen, en geen rekening meer hoefde te houden met eventuele haakjes of voorrangsregels in de formule. Bij uitvoer van dit programma werd ook nog een simpele *stack* gebruikt om de getallen en waarden van variabelen bij te houden, met operators — en hun volgorde — hoefde *at run-time* echter geen rekening meer te worden gehouden.

De posities op de *stack* waar *at run-time* de getallen werden geplaatst worden in het artikel aangeduid met η_1 voor de eerste plek op de *stack*, en η_i voor de i-de plek op de *stack*. De formule $a \times (b + c)$ werd vervolgens omgeschreven[4] in de volgende instructies:

[3] Het programma dat gegenereerd wordt na het compileren van de, door de programmeur geschreven, programmatekst.

[4] Voor het verkrijgen van deze vertalingen hebben we een `JAVA`-programma geschreven dat een expressie als invoer verwacht en een lijst instructies als uitvoer geeft, zie appendix C.

	Instructie	Omschrijving
i	$a \Rightarrow \eta_1$	Waarde van a wordt op de *stack* geplaatst.
ii	$b \Rightarrow \eta_2$	Waarde van b wordt op de *stack* geplaatst.
iii	$c \Rightarrow \eta_3$	Waarde van c wordt op de *stack* geplaatst.
iv	$\eta_2 \Rightarrow \eta_2 + \eta_3$	Plek twee op de *stack* krijgt waarde van de som van de bovenste twee getallen op de *stack*: $b + c$.
v	$\eta_1 \Rightarrow \eta_1 \times \eta_2$	Plek één van de *stack* krijgt de waarde van het product van de bovenste twee getallen op de *stack*: $a \times (b + c)$

De *stack* wordt hier dus enkel gebruikt om — *at compile-time* — de volgorde van bewerkingen vast te stellen, en om — *at run-time* — getallen en waardes van variabelen op te slaan, niet om — zoals Dijkstra dat wel deed — informatie over subroutines in op te slaan, zoals terugkeeradressen.

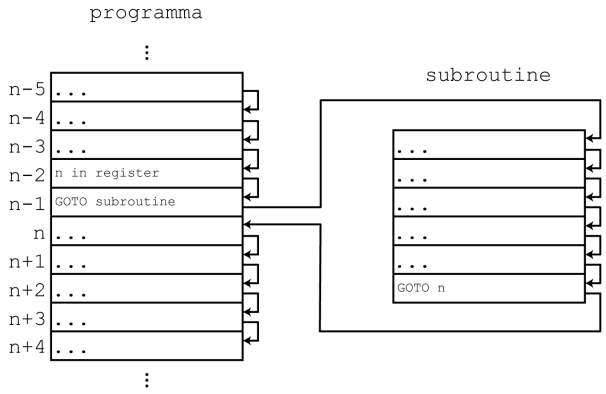
3.2 Subroutines

In de jaren vijftig en de vroege jaren zestig van de vorige eeuw waren computers een schaars goed. Deze machines waren niet zo maar te koop zoals dat nu het geval is en slechts enkele universiteiten beschikten over de middelen en kennis om zelf een computer te ontwikkelen. Doordat echte standaarden nog niet bestonden waren er maar weinig machines die op dezelfde manier werkten. Onderzoekers die zelf niet de beschikking hadden over een computer maakten regelmatig reizen naar universiteiten die dat wel hadden, om de werking van deze apparaten te leren van de mensen die ze hadden ontwikkeld en de mensen die er op dat moment mee werkten. Kennis over programmeertechnieken werd verspreid door zulk soort reizen, boeken en artikelen geschreven over dit onderwerp.

Maurice Wilkes en zijn team aan de University of Cambridge, Mathematical Laboratory ontwikkelden en bouwden eind jaren veertig van de vorige eeuw de EDSAC,[5] een van de eerste computers die met een digitaal geheugen werkte. Maurice Wilkes en David Wheeler — die in die tijd studeerde onder Wilkes — ontwikkelden het concept van een subroutine, een stuk code dat kan worden aangeroepen vanuit andere delen van het programma. In het — zeer invloedrijke — boek *The Preparation of Programs for an Electronic Digital Computer* [19, p.171-174] beschrijven Wilkes en collegae hun bevindingen op het gebied van programmeren op de EDSAC, waar ze ook zogenaamde *open-* en *closed* subroutines introduceren en toelichten.

Wanneer Bauer & Samelson in 'Sequential Formula Translation' het *procedure statement* en het aanroepen van subroutines behandelen, hebben zij het ook over een onderscheid tussen *open-* en *closed* subroutines, hetgeen ons doet vermoeden dat zij het aanroepen van subroutines op eenzelfde manier behandelden als Wilkes en consorten.

[5] EDSAC: Electronic Delay Storage Automatic Calculator [18].



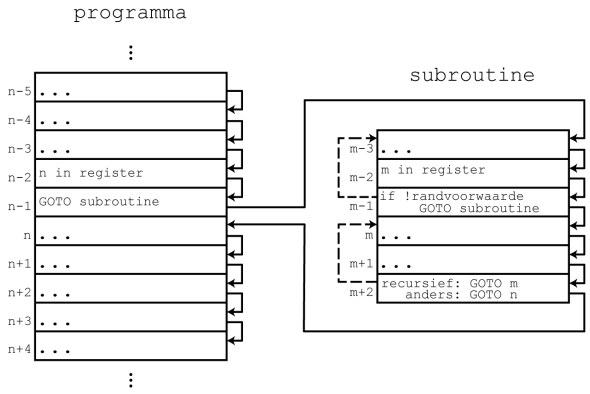
Figuur 4: Het normale verloop van een programma bij een subroutineaanroep.

3.2.1 Aanroep en terugkeer

Tegenwoordig wordt er met afkeur gekeken naar het aanpassen van de code van je programma terwijl deze wordt uitgevoerd [3, p.205]. Echter, in de tijd dat Wheeler het aanroepen van- en het terugkeren uit subroutines uitwerkte en het gebruik van de *subroutine library* in zwang kwam, was het de enige manier om je programma te laten terugkeren naar het punt waar uitvoering verder ging na afronding van de subroutine.

Programmeurs schreven programma's die sprongen naar de start van de subroutine door allereerst het adres van de regel code waar het programma erna verder moest gaan, op te slaan in een van de registers beschikbaar in de computerarchitectuur. Het eerste wat er dan gebeurde in de subroutine was het overschrijven van de laatste instructie door een sprong-instructie naar het adres opgeslagen in het register. Na afloop van de subroutine werd dan teruggesprongen naar de plek waar uitvoer van het programma hervat moest worden. Zie bijvoorbeeld het `DRAMA`-programma in tabel 6. Een illustratie van het verloop van een programma bij een subroutine-aanroep wordt gegeven in figuur 4.

Vanuit een subroutine konden ook weer andere subroutines worden aangeroepen. Het adres van de code waar uitvoering hervat moest worden na afsluiting van de aangeroepen subroutine, werd dan opgeslagen in het register. Hierna sprong men naar de tweede subroutine. Hier werd de terugkeersprong overschreven naar een sprong naar het adres opgeslagen in het register.



Figuur 5: Het verloop van een programma bij een recursieve subroutineaanroep.

3.2.2 Recursie

Recursie door een subroutine zichzelf aan te laten roepen is met bovenstaand omschreven aanpak onmogelijk. De laatste sprong, die overschreven wordt door een terugkeersprong, zal bij een tweede of verdere aanroep van dezelfde subroutine immers overschreven worden door een nieuwe terugkeersprong. Hierdoor zal er nooit uit de subroutine teruggekeerd kunnen worden naar het stuk code waar deze subroutine initieel werd aangeroepen. Recursie door een procedure zichzelf te laten aanroepen is dus onmogelijk.

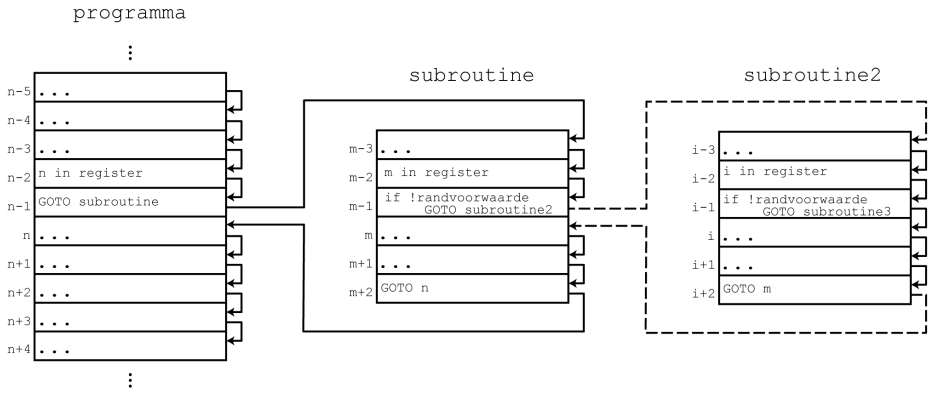
De verloop van een programma die verwacht wordt bij een recursieve aanroep wordt geïllustreerd in figuur 5. De gestippelde pijlen zijn de pijlen voor recursieve aanroep en terugkeer. Daar de laatste sprong in de subroutine telkens wordt overschreven door een sprong naar m, zal het programma nooit hervat kunnen worden op regel n.

3.2.3 Duplicatie

Dit probleem kan omzeild worden door de (recursieve) subroutine een aantal malen te kopiëren in het geheugen. Hierdoor hoeft de subroutine niet zichzelf aan te roepen, maar kan telkens gesprongen worden naar de volgende kopie van de originele subroutine. Hier zijn echter wel een aantal belangrijke nadelen aan verbonden, waar Dijkstra in de inleiding van zijn artikel 'Recursive Programming' ook al naar verwijst:

> ... the storage allocations will, in general, occupy much more memory space then they ever need *simultaneously* ... [8]

Ten tijde van het opschrijven van het programma, of bij het compileren van de programmatekst, is vaak nog niet duidelijk hoe vaak een recursieve



Figuur 6: Het verloop van een programma bij een recursieve subroutineaanroep, wanneer gebruik wordt gemaakt van recursie door middel van duplicatie.

aanroep bij uitvoer plaats zal vinden. Het is dus nodig om de subroutine zo vaak te kopiëren als het maximale aantal recursieve aanroepen dat je verwacht nodig te zullen hebben. Het is duidelijk dat dit ervoor zorgt dat je programma veel meer kopieën van de subroutine zal bevatten dan je in de meeste gevallen nodig zult hebben. Kies je ervoor om je subroutine minder vaak te dupliceren, zodat er economischer met de — in de jaren vijftig nog erg schaarse — geheugenruimte wordt omgegaan, dan zal het programma niet werken wanneer je dieper de recursie in gaat dan door het aantal kopieën wordt toegestaan.

Het verloop van een subroutine die gebruik maakt van duplicatie wordt aangegeven in figuur 6. Er wordt hier een recursieve aanroep met diepte van twee gegeven. Eerst wordt het origineel (subroutine) aangeroepen, vervolgens roept het origineel een kopie van zichzelf aan (subroutine2). Vanuit deze subroutine wordt teruggekeerd naar de originele subroutine en wanneer deze wordt afgesloten wordt het omvattende programma hervat.

Het moge duidelijk zijn dat het voor Bauer, Samelson en hun ALCOR-groep problematisch was om recursie te verwezenlijken in hun implementatie van de ALGOL-60 programmeertaal, en nog steeds de efficiëntie te bewerkstelligen die voor hen — en voor de groeiende computer-industrie — zo essentieel was.

4 [R] Andere actoren omstreeks 1960

Al voordat Bauer en Samelson hun invloedrijke artikel uitbrachten, maar vooral nadat Dijkstra in 1960 zijn artikel [8] publiceerde en in het echt liet zien dat zijn techniek werkte door als één van de eersten met een werkende

ALGOL-60 implementatie te verschijnen, waren ook anderen al bezig met verscheidene vraagstukken omtrent subroutines en hun aanroep, en hoe recursie hier in paste.

In deze tijd werden computers voornamelijk gebruikt om problemen uit de numerieke wiskunde te ontraadselen, en hoewel er nog geen problemen bekend waren die slechts op te lossen waren door recursieve algoritmen (deze volgden later pas), begonnen steeds meer mensen het nut van een implementatie te zien die recursieve aanroepen van subroutines toe stond.

In Bauer & Samelson en Dijkstra hebben we twee tegengestelde extremen gevonden in hun houding ten opzichte van recursie; de meeste wetenschappers stonden er echter een stuk gematigder tegenover. In deze sectie bespreken we kort het standpunt van een aantal anderen ten opzichte van recursie en de implementatie van recursieve subroutines.

4.1 Edgar T. Irons & Wallace Feurzeig

Waar Dijkstra elke procedure zag als een recursieve procedure, en Bauer & Samelson recursie in hun implementaties van ALGOL-60 zoveel mogelijk probeerden te vermijden omwille van efficiëntie, gingen Edgar T. Irons en Wallace Feurzeig op zoek naar een oplossing in het midden. Ze wilden de efficiëntie van de Bauer & Samelson implementaties behouden, terwijl ze toch recursieve procedures wilden kunnen gebruiken wanneer ze dat nodig hadden. De oplossing die zij voor dit probleem bedachten was slechts weinig minder efficiënt dan de oplossingen van Bauer & Samelson wanneer er geen recursie geconstateerd werd, maar die — wanneer dat nodig was — procedures zichzelf recursief kon laten aanroepen.

Om dit mogelijk te maken introduceerden Irons & Feurzeig in hun artikel 'Comments on the implementation of recursive procedures and blocks in ALGOL 60' [10] de zogenaamde *Depth Counter* (C), een teller die elke procedure met zich meedraagt:

> C(proc) is a counter unique to the procedure referencing it, which indicates the current depth of recursion of the procedure at any time during the execution of a program. C(proc) is used also to determine at run time whether the procedure is involved in recursion [10, p.67].

Tijdens de start van de uitvoer van het programma bedraagt de waarde van de teller (C) voor elke procedure -1. Wanneer de procedure wordt aangeroepen wordt C met 1 verhoogt, en wanneer uit de procedure wordt teruggekeerd naar het gedeelte van het programma waaruit de procedure werd aangeroepen, wordt C met 1 verlaagt. De waarde van C bij een niet recursief aangeroepen procedure is dus 0.

De waarde van C bepaalt of bij het aanroepen van een procedure het mechanisme voor het mogelijk maken van recursieve procedures in werking

wordt gesteld. Is dit niet het geval dan behoudt het programma door de minimale *overhead* — slechts het verhogen en verlagen van een teller — zijn efficiëntie, maar kan het in gevallen wanneer dat noodzakelijk is (als C groter is dan 0) toch procedures recursief aanroepen — wat dan natuurlijk wél ten koste gaat van de efficiëntie — met behulp van de *stack*, zoals Dijkstra dat al beschreef.

4.2 Henry Gordon Rice

Henry Gordon Rice was een logicus en wiskundige die zich ook gebogen heeft over het vraagstuk van recursie. In 1960 schreef hij een brief die werd gepubliceerd in de 'Communications of the ACM' [12], waarin hij zijn standpunten op dat moment uiteen zette. Later, in 1965, verscheen van zijn hand het artikel 'Recursion and Iteration' [13]. In dit artikel liet hij zien dat het mogelijk, en vrijwel altijd beter is, om een recursieve procedure om te schrijven in een iteratieve procedure[6] die hetzelfde effect bereikt.

In zijn brief uit 1960 schrijft Rice het volgende:

> *When to recur?* Never if you can avoid it.

Hoewel recursie wiskundigen een elegante manier geeft om bepaalde functies uit te drukken, vervolgt Rice, heeft het eigenschappen die ervoor zorgen dat het voor computationele doeleinden niet altijd geschikt is. Recursie kan echter niet altijd uit de weg gegaan worden: bepaalde functies kunnen enkel uit de basisfuncties waarover een computer beschikt — denk aan optellen en aftrekken — gedefinieerd worden met behulp van recursie.

Als recursie dan niet volledig uit de weg kan worden gegaan, hoe moeten we er dan mee omgaan?

> *How to recur?* From the bottom up.

In plaats van te beginnen bij het n-de argument van een functie ($f(n)$), en het argument telkens te verlagen, is het vaak economischer om de functie te laten starten bij 0 ($f(0)$), en het argument telkens te verhogen tot je bij het n-de argument komt. De procedure moet nu ook n keer uitgevoerd worden, maar hoeft zich niet eerst vanuit het n-de argument naar het 0-de argument te bewegen.

Primitive Recursive Functions [12, p.L.13] zijn een klasse van functies waarvan Rice beweert dat deze altijd op een *bottom-up* manier te vertalen zijn.

4.2.1 Primitive Recursive Functions

De logicus Martin Davis definieert de klasse van *primitive recursive functions* in het boek *Computability, Complexity, and Languages* als volgt [4, p.42]:

[6] Een procedure die gebruikt maakt van *for*- en *while*-lussen.

Initial functions Neem de drie *initial functions*:

$$\begin{aligned} s(x) &= x + 1 \\ n(x) &= 0 \\ u_i^n(x_1, \ldots, x_n) &= x_i, 1 \leq i \leq n \end{aligned}$$

Deze functies vormen de basis waaruit we andere functies kunnen construeren.

Composition Als f een functie is met k variabelen en $g_1, \ldots, g_k$ zijn functies van n variabelen, en

$$h(x_1, \ldots, x_k) = f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$$

dan zeggen we dat h door middel van *composition* gemaakt kan worden uit f en $g_1, \ldots, g_k$.

Primitive recursion Als m een geheel, positief getal is en

$$\begin{aligned} h(0) &= m \\ h(t+1) &= g(t, h(t)) \end{aligned}$$

waarbij g een totale functie[7] is, dan zeggen we dat h door middel van *primitive recursion* gemaakt kan worden uit g.

Nu bestaat de klasse van *primitive recursive functions* uit:

- De *initial functions*.
- De functies die door middel van *composition* of *primitive recursion* gemaakt kunnen worden uit andere functies uit deze klasse.

In het artikel 'Recursion and Iteration' [13] zegt Rice over de klasse van *primitive recursive functions*:

> Despite its apparently modest beginnings, the class of primitive recursive functions is extremely rich and extensive (...). I would venture to say that all functions ever evaluated on computers so far have been primitive recursive, with one exception (...).

Rice laat met een simpel voorbeeld in `FORTRAN IV` zien hoe *primitive recursive functions* makkelijk in een iteratief programma te vangen zijn. Hoewel dit voorbeeld gemaakt is in `FORTRAN IV` gaat ditzelfde voorbeeld ook op voor andere imperatieve hogere-orde programmeertalen, zoals ALGOL -60.

[7] Een totale functie is een functie die voor elke invoer een uitvoer geeft.

Direct noemt Rice echter wel een uitzondering — De *exception* waar hij het over heeft: *General Recursive Functions*, de klasse van alle berekenbare functies, die niet allemaal uitgedrukt kunnen worden door slechts *primitive recursion* te gebruiken.

Edgar G. Daylight beschrijft in hoofdstuk 2.2 van het boek *The Dawn of Software Engineering* [6, p.30-31] een gevolg van Kleene zijn *Normal Form Theorem* [11]: Niet alleen *primitive recursive functions*, maar ook *general recursive functions* zijn altijd — met behulp van een *while*-loop — om te schrijven in wat we, vandaag de dag, een iteratief algoritme zouden noemen. Het is echter niet altijd voor de hand liggend hoe, of praktisch om, dit te implementeren, vooral niet op computers uit de jaren zestig [13, p.114].

De Ackermann-functie is het voorbeeld dat Rice gebruikt[8] om te laten zien dat er functies zijn die niet slechts door middel van *primitive recursion* uitgedrukt kunnen worden. De versie die Rice gebruikt is als volgt:

$$\begin{aligned}
&\textbf{if } m = 0 && ACKER(0, n) = n + 1 \\
&\textbf{if } m \neq 0, n = 0 && ACKER(m, 0) = ACKER(m - 1, 1) \\
&\textbf{if } m, n \neq 0 && ACKER(m, n) = \\
& && \quad ACKER(m - 1, ACKER(m, n - 1))
\end{aligned}$$

Deze functie — die duidelijk recursief is — werd door Rice omgeschreven in een iteratief FORTRAN-programma, wat $ACKER(5, 0)$ kan uitrekenen in zeer korte tijd,[9] terwijl hij verwacht dat het op de recursieve manier vele malen langer zal duren.

Uiteindelijk sluit Rice zijn artikel af met een zin waaruit direct duidelijk wordt wat zijn visie is op recursief gedrag van computerprogramma's:

> I would like to suggest that applying ingenuity to find ways to avoid recursion is just as much fun as finding ways to use recursion, and a lot more profitable. [13, p.115]

We kunnen concluderen dat Rice — als logicus — goed bekend was met de natuur van *primitive recursion* en *general recursion*, en de waarde van recursie voor de berekenbaarheidstheorie hoog inschatte. Echter, Rice — als wiskundige en programmeur — was geen voorstander van recursie in programmateksten. Het is immers altijd mogelijk om een recursieve procedure om te schrijven in een procedure die van recursieve aanroepen geen gebruik maakt — slechts van iteratie en *while*-lussen — en vrijwel altijd is dit vele malen efficiënter dan het gebruik van recursieve aanroepen in het programma.

[8] De Ackermann-functie werd ontworpen door Wilhelm Ackermann, een student van David Hilbert, met het doel om aan te tonen dat er functies bestaan die niet met alleen maar *primitive recursion* uit te drukken zijn [13, p.114].

[9] Minder dan 10 seconden op de UNIVAC 1107, een computer ontworpen in oktober 1962.

5 [R] Gauthier van den Hove

In het artikel 'On the Origin of Recursive Procedures' [17] gaat Gauthier van den Hove op zoek naar de beginselen van recursie in imperatieve, hogereorde programmeertalen. Hij vertelt van Dijkstra en Peter Naur, die allebei suggereerden dat recursie geïntroduceerd werd in de ALGOL-60 programmeertaal door het toevoegen van één zin in het Report [1]:

> Any other occurence of the procedure identifier within the procedure body denotes activation of the procedure.

Van den Hove onderzoekt of het inderdaad zo is dat, zonder deze zin, recursieve aanroepen van procedures niet mogelijk zouden zijn in ALGOL-60, zoals gespecificeerd in het ALGOL-Report.

5.1 Declaratie versus Activatie

Een belangrijk onderscheid dat Van den Hove maakt, is het verschil tussen een *procedure declaration* en een *procedure activation* [17, p.2]. Een *declaration* is de tekst van een procedure, hoe deze is opgeschreven in de woorden en syntax van een (hogere-orde) programmeertaal.

> ... it is a program text, (...) (that) specifies a number of steps by which a certain task (...) can be performed mechanically.

Een *activation* daarentegen is de aanroep van een procedure bij uitvoering van het programma. Een *activation* vind dus plaats *at run-time*.

> An *activation* of a procedure is an execution instance of that procedure.

Het is belangrijk om in het achterhoofd te houden dat, volgens de definities van Van den Hove, de *declaration* en de *activation* van een procedure niet over precies dezelfde voorstelling van een procedure gaan. Een programmatekst — die een *procedure declaration* kan bevatten — moet eerst worden gecompileerd, en het objectprogramma dat daarbij ontstaat moet eerst worden uitgevoerd, voordat je van een *declaration* naar de *activation* van een procedure kunt gaan.

Van den Hove lijkt het onderscheid tussen deze twee voorstellingen vaak niet te maken, waar hij het enerzijds heeft over programmateksten uit de wiskundige wereld, heeft hij het anderzijds over technologische artefacten als object-programma's. Deze object-programma's beschouwt Van den Hove ook als wiskundige objecten, terwijl er wel een aantal aannames over *compiler*-technologie en de uitvoering van deze programma's gemaakt moeten worden om er op deze manier over te kunnen redeneren — het verschil tussen

programma's die gebruik maken van een *call stack*[10] en programma's met een statische implementatie bijvoorbeeld. Desalniettemin denken we dat de voorbeelden en uitwerkingen die Van den Hove geeft, ook met dit onderscheid in acht genomen, wel degelijk lijken te kloppen.

5.2 Recursie

Waar een *procedure declaration* de definitie van een procedure is in de programmatekst, zo ook is een *recursive procedure declaration* een procedure van welke je uit de programmatekst kunt afleiden dat deze zichzelf kan — of zal — aanroepen. Wanneer Van den Hove spreekt over een *recursive procedure activation* dan heeft hij het over de activatie van een gecompileerde procedure — *at run-time* — terwijl een vorige activatie van dezelfde procedure nog niet is afgerond.

De vraag of een *recursive procedure activation* noodzakelijk volgt uit een *recursive procedure declaration*, of dat deze twee concepten ook los van elkaar kunnen bestaan, staat centraal in Van den Hove zijn artikel. Met een aantal wiskundig zeer interessante voorbeelden laat hij zien dat deze twee voorstellingen van recursie — statisch en dynamisch — niet noodzakelijk met elkaar verbonden zijn. Het is mogelijk om een *recursive procedure declaration* in de programmatekst te geven, zonder dat dit leidt tot de recursieve activatie van een procedure, gelijk ook kunnen er *recursive procedure activations* plaatsvinden tijdens uitvoer van het programma, zonder dat deze door *recursive procedure declarations* zijn gedefinieerd.

5.3 Recursie met duplicatie

Het lijkt er op dat Van den Hove uitgaat van recursie zoals Dijkstra het al voor ogen had — met behulp van *stack-based activation frames*. Het is dus interessant om te onderzoeken of de analyse die Van den Hove geeft van recursie zoals Dijkstra er mee om zou zijn gegaan, ook opgaat wanneer je recursie in combinatie met duplicatie gaat bekeken, zoals Bauer en Wilkes het zouden hebben aangepakt.[11]

Hiertoe moeten we allereerst bekijken wat dat doet met de definities van *recursive procedure declarations* en *recursive procedure activations*. Één enkele subroutine kan nog maar één keer worden aangeroepen voordat deze wordt afgesloten, verdere aanroepen van deze subroutine verwijzen naar

[10] De *call stack* is de *stack* die informatie over uitgevoerde subroutines bijhoudt — zie de Dijkstra implementatie.

[11] Hier wordt gebruik gemaakt van duplicatie in de programmatekst voordat die gecompileerd wordt. Hermann Bottenbruch [2, p.40-41] schreef over een techniek die subroutine's *at run-time* kopieert, alleen wanneer dat nodig is. Deze techniek wordt hier verder niet besproken, maar werd waarschijnlijk veelvuldig gebruikt door programmeurs in de jaren vijftig en zestig.

kopieën van deze subroutine. Daarom spreken we af dat een *procedure activation* recursief is wanneer dit de *activation* betreft, niet alleen van de originele subroutine, maar ook van een kopie van de originele subroutine, terwijl het origineel nog niet is voltooid.

Gelijk spreken we ook af dat een *procedure declaration* recursief is, niet alleen wanneer hij in de tekst naar zichzelf verwijst, maar ook als uit de programmatekst blijkt dat hij naar een kopie van zichzelf verwijst.

Daarnaast gebruiken we de volgende notatie om een functie en haar duplicaten[12] aan te duiden: $\texttt{fun}(\texttt{x}_1, \ldots, \texttt{x}_n)$ is een functie met n argumenten, en $\texttt{fun}_i(\texttt{x}_1, \ldots, \texttt{x}_n)$ met $1 < i \leq k$, waarbij k een positief geheel getal is dat het aantal duplicaten aan duidt, geeft de i-de kopie van deze functie aan, en waarbij $\texttt{fun}_1(\texttt{x}_1, \ldots, \texttt{x}_n)$ staat voor de originele functie.

Een duplicaat is in een hogere-orde programmeertaal een exacte kopie van de originele procedure. Wanneer deze kopie echter gecompileerd wordt tot een object-programma, dan zullen er kleine verschillen te zien zijn doordat elke procedure zijn eigen terugkeeradressen, variabelen en dergelijke gebruikt.

Fibonacci Van den Hove geeft de volgende uitwerking van een fibonacci functie:

integer procedure $fibonacci(n)$**; integer** n**;**
$fibonacci :=$ **if** $n = 0 \vee n = 1$ **then** n
  else $fibonacci(n-1) + fibonacci(n-2)$**;**

Volgens de conventie die net is afgesproken zouden we de procedure op onderstaande manier omschrijven:

integer procedure $fibonacci_1(n)$**; integer** n**;**
$fibonacci_1 :=$ **if** $n = 0 \vee n = 1$ **then** n
  else $fibonacci_2(n-1) + fibonacci_2(n-2)$**;**
$\vdots$
integer procedure $fibonacci_i(n)$**; integer** n**;**
$fibonacci_i :=$ **if** $n = 0 \vee n = 1$ **then** n
  else $fibonacci_{i+1}(n-1) + fibonacci_{i+1}(n-2)$**;**
$\vdots$
integer procedure $fibonacci_k(n)$**; integer** n**;**
$fibonacci_k :=$ **if** $n = 0 \vee n = 1$ **then** n
  else $fibonacci_{k+1}(n-1) + fibonacci_{k+1}(n-2)$**;**

[12]Wanneer we hier spreken over het aantal duplicaten, dan bedoelen we het origineel samen met al haar kopieën. Wanneer we spreken van het aantal kopieën dan bedoelen we het aantal kopieën exclusief het origineel.

Wanneer het maximale aantal kopieën bereikt is, zal de laatste kopie (met k aangeduid) nergens naartoe kunnen verwijzen — kopie $k+1$ bestaat immers niet — en zal het programma niet correct uitgevoerd kunnen worden. De precieze uitvoer in dit geval hangt ervan af hoe deze duplicaten gecompileerd zijn. In onze implementatie van recursie door middel van duplicatie in `DRAMA` (zie onder andere de programma's in tabel 10 en tabel 11) hebben we ervoor gekozen hier een stopcode op te nemen, waardoor het programma stopt wanneer geprobeerd wordt van meer duplicaten gebruik te maken dan dat er beschikbaar zijn.

Intuïtief is wel aan te voelen dat de werking van een programma dat gebruikt wordt van een aantal duplicaten kleiner of gelijk aan k goed zal verlopen. Alle duplicaten van de fibonacci-functie zijn immers hetzelfde als het origineel, slechts de verwijzing in de recursieve aanroep is anders.

Mutual recursion Wederzijdse recursie (of *mutual recursion*) is een speciaal geval van recursie waar een functie niet direct zichzelf aanroept, maar waar twee — of meerdere — functies elkaar aanroepen, voordat de vorige aanroepen zijn afgesloten. Er is als het ware sprake van indirecte recursie.

Van den Hove geeft het volgende voorbeeld van een functie die het n-de fibonacci getal uitrekent, gebruik makende van wederzijdse recursie:

```
integer procedure fibonacci(n); integer n;
begin
  integer procedure aux(n); integer n;
  aux := if n = 0 ∨ n = 1 then n
    else xua(n − 1) + xua(n − 2);
  integer procedure xua(n); integer n;
  xua := aux(n);
  fibonacci := aux(n)
end
```

Het is duidelijk te zien dat $aux(n)$ en $xua(n)$ op papier zelf niet recursief gedeclareerd zijn, maar dat de aux-functie de xua-functie aan roept, en dat het enige doel van de xua-functie het opnieuw aanroepen van de aux-functie is. Hier is dus sprake van wederzijdse recursie.

Omgeschreven in een functie die gebruik maakt van recursie door middel van duplicatie zou de functie er als volgt uit komen te zien:

```
integer procedure fibonacci_1(n); integer n;
begin
  integer procedure aux_1(n); integer n;
  aux_1 := if n = 0 ∨ n = 1 then n
    else xua_1(n − 1) + xua_1(n − 2);
                          ⋮
  integer procedure aux_i(n); integer n;
  aux_i := if n = 0 ∨ n = 1 then n
    else xua_i(n − 1) + xua_i(n − 2);
                          ⋮
  integer procedure aux_k(n); integer n;
  aux_k := if n = 0 ∨ n = 1 then n
     else xua_k(n − 1) + xua_k(n − 2);

  integer procedure xua_1(n); integer n;
  xua_1 := aux_2(n);
                          ⋮
  integer procedure xua_i(n); integer n;
  xua_i := aux_{i+1}(n);
                          ⋮
  integer procedure xua_k(n); integer n;
  xua_k := aux_{k+1}(n);

  fibonacci_1 := aux_1(n)
end
```

We zien hier weer dat recursie door middel van duplicatie geen problemen geeft voor correcte uitvoer van het programma. Voor elke *aux*-functie is er een corresponderende *xua*-functie, en andersom, totdat we de laatste van de duplicaties bereiken.

Intermezzo Het nadeel van recursie door middel van duplicatie — het feit dat het veel geheugenruimte kost — wordt hier direct duidelijk. Er moeten immers niet k duplicaten van één enkele functie (de *aux*-functie) worden opgeslagen, ook de *xua*-functie moet k maal in het geheugen voorkomen. Het is duidelijk dat de benodigde geheugenruimte snel toeneemt wanneer er nog meer functies gebruikt worden in de wederzijds recursieve aanroepen, van elke functie moeten immers k duplicaten in het geheugen opgeslagen worden. Bij directe recursie zijn er $k - 1$ kopieën van deze functie, bij het bovenstaande voorbeeld zijn er $2 \cdot k$ functies en bij wederzijdse recursie met m functies zullen er $m \cdot k$ functies in het geheugen opgeslagen moeten worden.

Self application Wanneer een functie een nieuwe functie als argument vraagt, en dezelfde functie wordt als parameter meegegeven, dan is er sprake van *self application*. Als de nieuwe functie nogmaals uitgevoerd wordt is er dus weer sprake van recursie, daar de nieuw uitgevoerde functie dezelfde is als de aanroepende functie, en de aanroepende functie nog niet werd afgesloten voor de nieuwe functie werd uitgevoerd.

De volgende procedure is de procedure die Van den Hove gebruikt ter illustratie:

```
integer procedure fibonacci(n); integer n;
begin
  integer procedure aux(n, p);
    integer n; integer procedure p;
  aux := if n = 0 ∨ n = 1 then n
    else p(n − 1, p) + p(n − 2, p);
  fibonacci := aux(n, aux)
end
```

Procedure aux vraagt hier een procedure als argument, en voert deze procedure uit met $n - 1$ en $n - 2$ als argumenten. In de regel $fibonacci := aux(n, aux)$ is duidelijk te zien dat de procedure aux zichzelf meekrijgt als parameter. Bij uitvoer van dit programma zal dus sprake zijn van recursieve aanroepen daar de aux-procedure zichzelf aan zal roepen via de parameter p.

Dit programma is niet direct om te schrijven in een programma dat gebruik maakt van recursie door middel van duplicatie. De procedure die wordt meegegeven aan de eerste aanroep van aux, zal immers later worden uitgevoerd en zelf weer worden meegegeven als parameter; zie het onderstaande voorbeeld.

```
integer procedure fibonacci_1(n); integer n;
begin
  integer procedure aux_1(n, p);
    integer n; integer procedure p;
  aux_1 := if n = 0 ∨ n = 1 then n
    else p(n − 1, p) + p(n − 2, p);
                    ⋮
  integer procedure aux_i(n, p);
    integer n; integer procedure p;
  aux_i := if n = 0 ∨ n = 1 then n
    else p(n − 1, p) + p(n − 2, p);
                    ⋮
  integer procedure aux_k(n, p);
    integer n; integer procedure p;
  aux_k := if n = 0 ∨ n = 1 then n
    else p(n − 1, p) + p(n − 2, p);

  fibonacci_1 := aux_1(n, aux_2)
end
```

De $fibonacci_1$ procedure roept hier de aux_1-procedure aan met aux_2 als parameter. Vervolgens roept aux_1 de procedure meegekregen — aux_2 dus — aan, nogmaals met aux_2 als argument. Nu zal aux_2 in zijn uitvoering niet verwijzen naar het volgende duplicaat, maar naar zichzelf. Dit betekent niet dat recursie door middel van duplicatie niet kan werken met *self application*, we kunnen bijvoorbeeld weer gebruik maken van wederzijdse recursie om dit te repareren, getuige onderstaande programmatekst:

```
integer procedure fibonacci_1(n); integer n;
begin
  integer procedure aux_1(n, p);
    integer n; integer procedure p;
  aux_1 := if n = 0 ∨ n = 1 then n
    else p(n − 1) + p(n − 2);
                    ⋮
  integer procedure aux_i(n, p);
    integer n; integer procedure p;
  aux_i := if n = 0 ∨ n = 1 then n
    else p(n − 1) + p(n − 2);
                    ⋮
  integer procedure aux_k(n, p);
    integer n; integer procedure p;
  aux_k := if n = 0 ∨ n = 1 then n
    else p(n − 1) + p(n − 2);

  integer procedure xua_1(n); integer n;
  xua_1 := aux_2(n, xua_2)
                    ⋮
  integer procedure xua_i(n); integer n;
  xua_i := aux_{i+1}(n, xua_{i+1})
                    ⋮
  integer procedure xua_k(n); integer n;
  xua_k := aux_{k+1}(n, xua_{k+1})

  fibonacci_1 := aux_1(n, xua_1)
end
```

We zien hier dat recursie door middel van duplicatie ook werkt voor functies waar gebruik gemaakt wordt van *self application*, het is alleen niet zo triviaal als het slechts dupliceren van de functie die geappliqueerd moet worden. Door middel van wederzijdse recursie en een hulpprocedure is het echter toch nog mogelijk om ook *self application* in recursie te vangen op zo'n manier dat ook Bauer & Samelson het hadden kunnen implementeren.

5.4 Conclusie

Van den Hove geeft een goede wiskundige analyse van het concept recursie en hoe dit past binnen hogere-orde programmeertalen. Hij laat zien dat, wanneer je een programmeertaal ontwikkelt met een basis in de wiskunde, het bijna onmogelijk is om recursieve aanroepen van procedures (*recursive procedure activations*) buiten te sluiten, ook al doe je erg je best om te voorkomen dat statische recursie (*recursive procedure declarations*) in de

programmeertaal gebruikt mag worden.

Door het gebruik van *mutual recursion* kan een procedure die op papier recursief is, omgeschreven worden in een procedure die — met gebruik van een hulpprocedure — niet direct recursief is. Wanneer je ook *mutual recursion* verbiedt kunnen door middel van *self application* alsnog *recursive procedure activations* plaastvinden.

Van den Hove zijn theorie over recursie is echter gebaseerd op een implementatie van de programmeertaal met *stack based activation frames*, op de manier die Dijkstra gebruikt zou hebben om met recursie om te gaan. Uit deze vergelijking blijkt dat recursie geïmplementeerd op een manier die Bauer & Samelson ook hadden kunnen gebruiken — op een statische manier, door middel van duplicatie — ook binnen de theorie van Van den Hove valt, mits je de definities die hij geeft uitbreidt door kopieën van de procedure toe te laten. Hier wordt dus een disjunctie geïntroduceerd, het gaat over de procedure zelf *of* een kopie van deze procedure.

Een *recursive procedure activation* is nu niet alleen meer een *activation* van een procedure voordat een vorige *activation* van deze procedure is afgesloten, ook de *activation* van een duplicaat van deze procedure wordt nu een *recursive procedure activation* genoemd.

Gelijk ook wordt de definitie van een *recursive procedure declaration* aangepast: een *recursive procedure declaration* hoeft niet een procedure te zijn die in de programmatekst naar zichzelf verwijst, ook het verwijzen naar een duplicaat van de procedure wordt een *recursive procedure declaration* genoemd.

Doordat het noodzakelijk is om een disjunctie te gebruiken om een algemenere kijk op recursie ook te kunnen verklaren blijkt dat de theorie van Van den Hove heel mooi werkt voor een subset van de implementaties van hogere orde programmeertalen, namelijk die welke gebruik maken van *stack based activation frames*, maar dat het niet de meest elegante manier is om recursie in programmeertalen in zijn algemeen te beschrijven.

Het werk van Irons en Feurzeig, zoals beschreven in sectie 4.1, met een hybride vorm van subroutine aanroepen, zal wel goed door de theorie van Van den Hove ondervangen worden. De eerste maal dat een functie wordt aangeroepen is er geen sprake van recursie. De volgende malen dat deze functie wordt aangeroepen zal dit gebeuren zoals Dijkstra dit voorzag — door gebruik te maken van de *stack* om informatie over de subroutine in op te slaan en zoals Van den Hove zelf ook in zijn artikel aanhaalt.

In de volgende sectie bestuderen we het werk van Christian Rinderknecht, waar hij recursie analyseert met behulp van *call-graphs*, en zullen we zien of deze theorie ons andere inzichten geeft over het gebruik van recursie in hogere-orde programmeertalen — al dan niet met *stack based activation frames* of statische implementaties.

6 [B] Christian Rinderknecht

In ons onderzoek benaderen we recursie op een pluralistische manier; we verplaatsen ons in verschillende actoren (Bauer, Samelson, Dijkstra) en proberen vanuit hun interpretatie inzichtelijk te maken wat recursie precies voor hen betekende. Christian Rinderknecht geeft in zijn artikel 'A Survey on Teaching and Learning Recursive Programming' [14] een overzicht van verschillende didactische opvattingen op het gebied van recursief programmeren. Ideeën van meerdere actoren die recursie onderwijzen worden naast elkaar gezet, vergeleken en geïnterpreteerd. In dit hoofdstuk zal de nadruk liggen op het contrast met het reeds behandelde artikel van Gauthier van den Hove, die recursie vanuit een meer wiskundig standpunt bekijkt en zo tot één juiste definitie probeert te komen. Aangezien beide artikelen vrij recent zijn (Van den Hove zijn artikel werd gepubliceerd in 2014, dat van Rinderknecht eerder in hetzelfde jaar) is het interessant om te onderzoeken hoe zij het begrip recursie definiëren en stellen we onszelf de vraag: wat zijn de verschillen tussen deze definities en hoe verhouden die zich tot de meer gedateerde teksten van Dijkstra en Bauer?

Om het begrip recursie te definiëren maakt Rinderknecht net als Van den Hove de tweedeling tussen statische en dynamische recursie. Beide gebruiken hiervoor hun eigen woorden en begrippen, dit wordt in de eerste alinea uiteengezet. Vervolgens zal duidelijk worden op welke manier Rinderknecht de relatie tussen deze twee soorten recursie beschrijft en waarin deze beschrijving verschilt van die van Van den Hove.

6.1 Statisch versus dynamisch

Voordat Rinderknecht in gaat op de vraag wat recursie precies betekent, maakt hij eerst een duidelijk onderscheid tussen statische en dynamische criteria. Statische (syntactische) criteria hebben betrekking op de functie *definitie*, dit verwijst dus puur naar de functie zoals die statisch op papier (of in de code) staat. Een functie die in termen van zichzelf gedefinieerd wordt (in de *scope* van de functie komt precies dezelfde functie voor) kun je syntactisch gezien een recursieve functie noemen. Dynamische criteria daarentegen, hebben betrekking op de functie *aanroep*, ofwel het daadwerkelijke uitvoeren van de code. Deze tweedeling komt sterk overeen met het onderscheid dat Van den Hove maakt tussen de *procedure declaration* en *procedure activation*. Waar Van den Hove in zijn artikel 'On the origin of recursive procedures' laat zien dat een *recursive procedure declaration* niet per definitie ook een *recursive procedure activation* impliceert en andersom, wordt dit door Rinderknecht nog eens in eigen woorden benadrukt:

> It should be noted that the static and dynamic definitions of recursion may overlap, but are different in general, that is, if a

> function is recursive according to the syntactic criterion, it may not be recursive according to the dynamic criterion, and vice versa [14, p.90].

De manier waarop Rinderknecht aantoont dat syntactisch recursieve functies geen dynamisch recursieve functies impliceren, heeft veel weg van die van Van den Hove. Beide auteurs gebruiken een vorm van recursie, genaamd *staartrecursie*.

Een recursieve functie is staartrecursief als de recursieve aanroep de laatste instructie in de functie is. Om dit te verduidelijken kijken we naar een voorbeeld van zo'n staartrecursieve functie: `f(x, y) := f(x, g(y))`. Hoewel we deze functie syntactisch gezien niet staartrecursief mogen noemen (het gaat hier echter om zogenoemde staartaanroepen en aanroepen zijn dynamisch), is de functie *definitie* wel degelijk recursief. De waarde van `f(x, y)` is syntactisch gedefinieerd in termen van `f(x, g(y))`. Staartrecursieve procedures hebben de eigenschap dat ze altijd kunnen worden omgeschreven naar een niet-recursieve procedure, namelijk een *loop*:

> A loop is a segment of code syntactically distinguished and whose evaluation is repeated until a condition on the state of the memory is met. [14, p.93]

Dit komt omdat er geen lokale waarden onthouden hoeven te worden die na de recursie nog gebruikt moeten worden om het eindresultaat te berekenen. De recursieve aanroep wordt uitgevoerd en vervolgens wordt teruggekeerd naar de procedure waarbinnen deze werd aangeroepen, maar vervolgens hoeft er niets meer uitgevoerd te worden. Dit kan dus probleemloos vertaald worden naar een *loop*-constructie, waarbij na een iteratie ook direct naar het begin van de *loop* teruggesprongen wordt. Dit gegeven wordt door Rinderknecht gebruikt om te bewijzen dat statische recursie geen dynamische recursie impliceert. Met andere woorden, er bestaan functies die syntactisch gezien recursief zijn, maar tijdens uitvoering niet recursief blijken te zijn.

Wanneer Rinderknecht recursie echter volgens dynamische criteria gaat definiëren, zien we duidelijk een contrast met het werk van Van den Hove. Laatstgenoemde beschrijft een *recursive procedure activation* door middel van een *execution model*.

> An activation of a procedure is an execution instance of that procedure: it is created when the execution of the procedure begins, and terminated when the end of the procedure is reached. [17, p.2]

Waar Van den Hove dynamische recursie omschrijft door middel van *stack-based* implementaties (meerdere gelijktijdige *activations* van één en

dezelfde procedure worden in het geheugen opgestapeld), weet Rinderknecht het begrip te abstraheren van implementaties en uit te drukken als een eigenschap van een wiskundig object, de *call graph*.

6.2 Call graph

Een *call graph* is een grafische representatie van procedure-aanroepen tijdens uitvoering van een programma. Een knoop in zo'n diagram representeert een procedure (of functie, subroutine) en een pijl van knoop A naar knoop B houdt in dat procedure A procedure B aanroept. Rinderknecht maakt onderscheid tussen een *static call graph* en een *dynamic call graph*. Eerstgenoemde laat alle mogelijke aanroepen zien die in een programma voor kunnen komen, ongeacht de input die het programma krijgt. Later in dit hoofdstuk zal blijken dat zo'n *static call graph* ook gebruikt kan worden om te zien of een functie 'wederzijds' recursief is. Nu kijken we eerst naar recursie volgens dynamische criteria, waarvoor Rinderknecht gebruik maakt van de eigenschappen van een *dynamic call graph*:

> ...recursion is a reachable cycle, which means, in operational terms, that the control flow of calls returns to a vertex (a function) which was previously called. Here, the notion of *recursive definition* is not central, and it makes sense to speak of *recursive call* (a back edge closing a path). [14, p.90]

Een *dynamic call graph* beschrijft dus de *control flow* van één uitvoering van het programma, en er is sprake van dynamische recursie als in de *dynamic call graph* van zo'n uitvoering een lus te vinden is. Laten we kijken naar een voorbeeld in de functionele programmeertaal OCaml dat hij in zijn artikel aanhaalt:

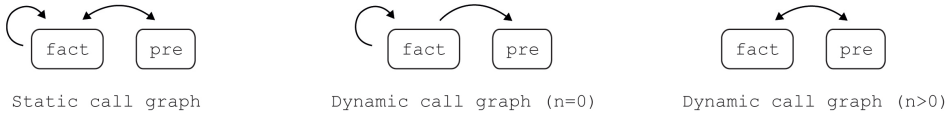
```
# let pre self n = if n = 0 then 1 else n * self(n-1);;
val pre : (int -> int) -> int -> int = <fun>
# let rec fact n = pre fact n;;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
```

We zien dat *fact* als argument een `int` neemt (de inputwaarde n) en de functie *pre* aanroept met als argument zichzelf en de waarde n.

Syntactisch gezien is vrij snel duidelijk dat de functie *fact* een recursieve functie is. In het rechterdeel van de definitie (*pre fact n*) komt immers dezelfde functie *fact* voor als in het linkerdeel. Bij de definitie van de functie *pre* is dit niet het geval. Deze neemt een functie van type `(int -> int)` toegepast op een `int`. Bij uitvoering van het programma zal dat de functie *fact* zijn. Vervolgens wordt gekeken of de waarde van n gelijk is aan 0. Is

dit het geval, dan geeft *pre* de waarde 1 terug. Is dit niet het geval, dan geeft *pre* het resultaat terug van de vermenigvuldiging van n met een `int` die berekend wordt door *fact(n-1)*.

In de rechter *graph* van figuur 7 is te zien dat er bij een inputwaarde groter dan 0, sprake is van 'wederzijdse' recursie. Dat wil zeggen, beide functies roepen *elkaar* aan. Hoewel de functie *pre* niet gedefinieerd is in termen van zichzelf (en om die reden dus syntactisch gezien niet recursief is) zien we aan de *dynamic call graph* dat *pre* en *fact* dynamisch gezien wederzijds recursief zijn. Waar Rinderknecht eerder reeds aantoonde dat statische recursie geen dynamische recursie hoeft te impliceren, bewijst hij op deze manier dat dit ook andersom niet het geval is.



Figuur 7: *Call graphs* van de faculteitsfunctie.

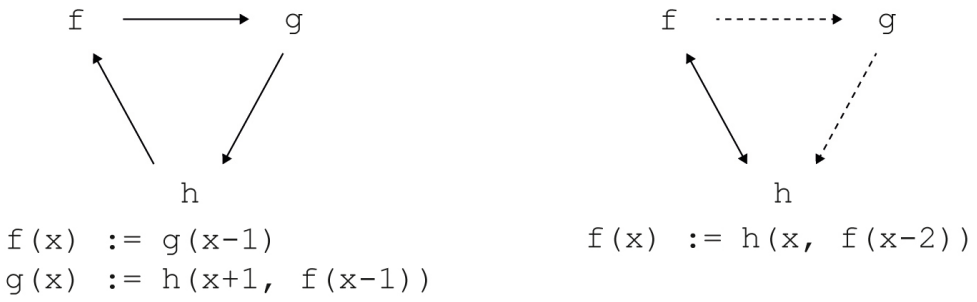
Verder laat Rinderknecht nog zien dat het ook mogelijk is om op een statische manier over wederzijdse recursie te kunnen spreken. Als voorbeeld worden de volgende twee wiskundige definities beschouwd:

```
f(x) := g(x-1)
g(x) := h(x+1, f(x-1))
```

Omdat er in de *static call graph* een pijl van f naar g en van g naar h gaat, kan door middel van transitiviteit (ook een wiskundig concept) ook een pijl van f direct naar h getrokken worden. Op deze manier laat Rinderknecht zien dat op een syntactisch niveau ook gesproken kan worden over een wederzijds recursieve relatie tussen f en h (zie figuur 8). De twee niet-statisch-recursieve definities zijn dus om te schrijven naar een equivalente definitie die wel statisch recursief is. Hoewel wederzijdse recursie een dynamische relatie is (de recursie is afhankelijk van aanroepen) is het op deze manier toch mogelijk om op een statische manier over wederzijdse recursie te kunnen praten.

6.3 Contrast met Van den Hove

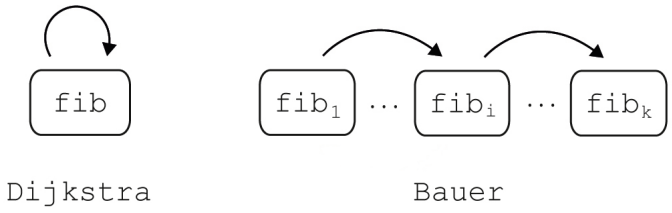
Als we kijken naar het artikel 'A Survey on Teaching and Learning Recursive Programming' van Rinderknecht, dan zijn er vrij veel overeenkomsten te ontdekken met het reeds eerder behandelde artikel van Van den Hove. Beide onderzoekers maken een duidelijk onderscheid tussen statische en dynamische recursie en beide omschrijven de relatie tussen deze twee soorten als disjunct; statische recursie impliceert geen dynamische recursie, en dynamische recursie impliceert geen statische recursie. In de manier waarop zij dit aanpakken zijn echter wel enige interessante contrasten te ontdekken.



Figuur 8: Transitiviteit toegepast op een *static call graph*.

Ten eerste valt op dat Rinderknecht in tegenstelling tot Van den Hove niet tot één juiste definitie van recursie probeert te komen. Recursie is een breed begrip dat op allerlei manieren geïnterpreteerd kan worden. Het doel van zijn paper is dan ook slechts om inzicht te verschaffen in de verschillende manieren waarop recursie in hogere-orde programmeertalen benaderd kan worden.

Ten tweede omschrijft hij de relatie tussen statische en dynamische recursie op een algemenere manier dan dat Van den Hove dat doet. Een *function definition* is recursief als deze is gedefinieerd in termen van zichzelf. Rinderknecht laat door middel van een *static call graph* zien dat als je daar transitiviteit op toepast, dat je dan kunt zien of de functie *mutual recursive* is. Van den Hove doet in principe hetzelfde, maar beperkt zich in zijn definitie wel tot *program text*. Ook op het gebied van dynamische recursie is dit verschil in abstractie duidelijk aanwezig. Waar Van den Hove een *recursive procedure activation* definieert aan de hand van *stack-based* implementaties in de stijl van Dijkstra, is Rinderknechts definitie van dynamische recursie abstracter, niet gebonden aan de eigenschappen van een computerarchitectuur.



Figuur 9: *Call graphs* van de twee `fibonacci`-implementaties.

Laten we ter illustratie de implementaties van de `fibonacci`-functie uit het hoofdstuk over Van den Hove bekijken. Waar Dijkstra dynamische recursie ziet als meerdere incarnaties van één en dezelfde procedure, worden in

de implementatie van Bauer telkens duplicaten van een procedure aangeroepen. In figuur 9 zien we een *dynamic call graph* van beide implementaties. In de implementatie van Dijkstra is eenvoudig te zien dat er sprake is van een lus, en volgens de definitie van Rinderknecht is er dus ook sprake van dynamische recursie. Kijken we naar de implementatie van Bauer, dan is er in de *dynamic call graph* geen lus te ontdekken en kunnen we dus niet spreken van dynamische recursie. Om recursie door middel van duplicatie ook in de theorie van Rinderknecht te laten passen, zullen we een soortgelijke disjunctie als die besproken in het vorige hoofdstuk over Van den Hove moeten introduceren. Er is dan niet alleen sprake van recursie bij een lus in de *dynamic call graph*, ook bij een aaneenschakeling van duplicaten (zoals te zien in de rechterafbeelding in figuur 9) zullen we spreken van recursie.

In zijn artikel haalt Rinderknecht verschillende voorbeelden aan in verschillende hogere-orde programmeertalen (zoals functionele en imperatieve programmeertalen). Daarnaast kan hij zelfs wiskundige functies als voorbeeld gebruiken (zie figuur 8) om recursie uit te leggen. Zijn definities van statische en dynamische recursie zijn gebaseerd op een eigenschap van een wiskundig object en zijn daarom (in tegenstelling tot de definitie van Van den Hove) niet gebonden aan een bepaald type implementatie.

7 [S] DRAMA

7.1 Subroutines

Het eerste wat opvalt wanneer je de implementatie van Bauer & Samelson op het niveau van machinecode gaat vergelijken met de implementatie van programma's op de manier van Dijkstra, is dat het Dijkstra meer werk, regels code, en meer geheugen kost om een subroutine aan te roepen. De Duitsers vonden het een kunst om alle eigenschappen van elke specifieke computer zo te gebruiken (of misbruiken) dat een programma zo efficiënt mogelijk was — maar dan alleen maar op dat systeem. Dijkstra gaf aan een minder efficiënt programma de voorkeur, als het maar zo algemeen was dat het op zo veel mogelijk systemen gebruikt kon worden. Waar Bauer & Samelson de *stack* slechts gebruikten om parameters voor de aangeroepen subroutine op te zetten, en het terugkeeradres naar de aanroepende subroutine in de accumulator te bewaren, gebuikte Dijkstra de *stack* om niet slechts de parameters, maar ook het terugkeeradres en de *parameter pointer* (en de rest van de *link*) op te bewaren.

Wanneer meer dan één subroutine wordt aangeroepen groeit bij Bauer & Samelson de *stack* alleen met het aantal parameters dat de subroutines meekrijgen, terugkeeradressen worden bewaard in de accumulator en direct gebruikt om de terugkeersprong aan te passen, waarna de accumulator voor andere doeleinden gebruikt kan worden. Bij Dijkstra groeit de *stack*, door alle extra boekhouding die hij doet omdat hij ervan uit gaat dat elke functie

in principe recursief is — of kan zijn — erg snel.

In de uitvoering van deze programma's zijn weinig voordelen te benoemen van de aanpak die Dijkstra voorstelt. De voordelen worden pas duidelijk wanneer je gaat kijken naar elegantie, generalisatie en leesbaarheid van de programma's.

We zien dat de niet-recursieve programma's van Bauer & Samelson korter zijn dan dezelfde soort programma's van Dijkstra, maar wanneer er recursie in het spel komt groeien de programma's van Bauer & Samelson erg snel, daar elke functie die recursief aangeroepen kan worden meerdere malen gedupliceerd moet worden. Vantevoren (*at compile-time*) is bovendien vaak nog niet duidelijk hoe diep de recursie dit maal zal zijn, dus er moet voor de zekerheid een groot — of in ieder geval ruim voldoende — aantal duplicaties van de procedure gemaakt worden. In deze (in potentie) recursieve programma's blijft de programmatekst in de stijl van Dijkstra even lang als wanneer het programma niet recursief zou zijn. Wel is de uitvoer van deze programma's bij Dijkstra veel trager en kost dit ook meer geheugen om de redenen die hiervoor al genoemd zijn.

7.2 De stack

Waar de *stack* bedoeld was als een stuk geheugen waar je maar aan één kant elementen kunt toevoegen en verwijderen, gebruikt Dijkstra de *stack*, door alle extra boekhouding, soms ook als een *Random-Acces Memory*. Waar Bauer & Samelson de *stack* voornamelijk zo gebruiken als hij bedoeld was — door er inderdaad alleen de meest recent toegevoegde elementen van af te halen — kost het Dijkstra nóg meer boekhouding om ook diep in de *stack* te graven naar de delen van het geheugen die hij op dat moment nodig heeft.

De reden dat Dijkstra toch voor deze aanpak gekozen heeft is om de blokstructuur van het programma, en de structuur van de *stack*, te waarborgen. Binnen de activatie van een *block* moet er immers ook toegang zijn tot variabelen uit andere *blocks* die op dat moment nog niet zijn afgesloten (zie ook sectie 2.4 in het hoofdstuk over Dijkstra).

Een van de dingen die Dijkstra en Bauer & Samelson gemeen hebben is dat ze er voor zorgen dat de *stack*, bij terugkeer uit een subroutine, niet meer elementen bevat dan toen de subroutine werd aangeroepen — op de uitkomst van de bewerking van de subroutine na.

7.3 Het geheugen

Waar Dijkstra probeert zoveel mogelijk informatie in de *stack* te bewaren, en daardoor ook alle extra boekhouding voor het graven in de *stack* nodig heeft, slaan Bauer & Samelson veel meer informatie op in het gewone geheugen van de computer. Het geheugengebruik — buiten de *stack* om — is bij Dijkstra dus kleiner dan bij de Duitsers.

7.4 [R] Kwantitatieve vergelijking

In deze sectie geven we een kwantitatieve analyse van de verscheidene manieren om een probleem op te lossen en kunnen we de verschillende manieren aan de hand van deze getallen vergelijken.

De verschillende programma's zijn beoordeeld aan de hand van vier eigenschappen:

Lengte Dit is de lengte van het `DRAMA`-programma in regels code. We verwachten dat de niet-recursieve programma's in de stijl van Bauer & Samelson doorgaans korter zijn dan die in de stijl van Dijkstra, maar wanneer recursie door middel van duplicatie in het spel komt zullen de programma's van Bauer & Samelson vlug groeien. Hoewel hier de lengte van een programma na het compileren wordt aangegeven, groeien de programma's van Dijkstra *at-runtime* natuurlijk vlug wanneer er recursieve aanroepen plaatsvinden. Over de gebruikte geheugenruimte *at-runtime* wordt hier niets verteld.

Aantal stappen Het aantal stappen dat een programma uit moet voeren om van een gegeven invoer bij het antwoord te komen. We verwachten dat het aantal stappen bij programma's in de stijl van Dijkstra over het algemeen groter zal zijn dan bij programma's in de stijl van Bauer & Samelson, daar Dijkstra voor de boekhouding in de *stack* meer werk moet verrichten.

Bewerkingen op stack Het aantal keer dat er een element op de *stack* wordt gezet, of dat er een element vanaf wordt gehaald. Ook hier verwachten we dat programma's in de stijl van Dijkstra een groter aantal *stack*-bewerkingen zal verrichten door de extra boekhouding.

Geheugenbewerkingen Waar programma's in de stijl van Dijkstra vrijwel alles bijhouden op de *stack*, verwachten we dat programma's in de stijl van Bauer & Samelson vaker gebruik zullen maken van de rest van het geheugen. Bijgehouden wordt hoe vaak een regel in het geheugen gewijzigd wordt (ook het wijzigen van de terugkeersprongen) en hoe vaak de waarde van een variabele uit het geheugen wordt gehaald.

7.4.1 Functie $\mathbf{a \times (b + c)^2}$

In tabel 1 zien we de scores van twee programma's op bovenstaande vier punten. In deze programma's wordt in de verschillende stijlen het antwoord op de functie $a \times (b + c)^2$ berekent, bij een invoer van $a = 3$, $b = 4$ en $c = 2$.

Wat hier voornamelijk opvalt is dat er niet erg veel verschil zit tussen beide programma's. De enkele geheugenbewerking in het programma in de stijl van Bauer & Samelson is volledig te verklaren door het feit dat de regel

	Bauer & Samelson functie	**Dijkstra** functie
Lengte (regels code)	27	30
Aantal stappen	25	29
Bewerkingen op *stack*	12	18
Geheugenbewerkingen	1	0

Tabel 1: Vergelijking van twee DRAMA-programma's (tabel 9, 12) om de functie $a \times (b + c)^2$ uit te rekenen, met $a = 3$, $b = 4$ en $c = 2$. Uitvoer: 108.

die zorgt voor terugkeer uit de subroutine moet worden overschreven. We zien al wel dat het programma in de stijl van Dijkstra — ook al bij zo'n simpel programma — significant vaker gebruik maakt van de *stack*.

7.4.2 Drie versies van faculteit

In tabel 2 worden drie programma's vergeleken die elk de faculteit van het getal (invoer) 5 berekenen. De faculteitsfunctie is gemakkelijk om te schrijven in een *loop*, maar ook op een voor de hand liggende manier recursief te berekenen. In het programma dat gebruik maakt van recursie door middel van duplicatie waren — in totaal — vijf versies van de faculteitsfunctie opgeslagen.

	Bauer & Samelson niet recursief	**Bauer & Samelson** recursie met duplicatie	**Dijkstra** recursief
Lengte (regels code)	21	73	23
Aantal stappen	49	63	64
Bewerkingen op *stack*	18	18	29
Geheugenbewerkingen	1	5	0

Tabel 2: Vergelijking van drie verschillende DRAMA-programma's (tabel 8, 10, 14) om de faculteit van getal (n) te berekenen, met invoer $n = 5$. Uitvoer: 120.

Wat hier direct opvalt is dat recursie door middel van duplicatie zorgt voor een sterke stijging van het aantal regels code. Waar de niet recursieve versie, en de versie met recursie in de stijl van Dijkstra niet veel meer dan 20 regels code nodig hebben, gebruikt de versie met recursie door duplicatie een gigantische 73 regels code.

We zien ook dat de beide versies die van recursie gebruik maken allebei ongeveer evenveel stappen nodig hebben om tot een antwoord te komen, terwijl de niet recursieve versie er minder nodig heeft.

Het aantal bewerkingen op de *stack* is voor beide implementaties in de stijl van Bauer & Samelson gelijk, daar zij de *stack* slechts gebruiken om getallen op te zetten. Dijkstra's versie gebruikt de *stack* veel vaker omdat

er in de recursieve versie veel subroutine aanroepen plaatsvinden, en de informatie hiervoor in deze versie wordt bijgehouden op de *stack*.

Dat verklaart ook het aantal geheugenbewerkingen: in de niet recursieve versie wordt er maar één keer een subroutine aanroepen, dus is er maar één geheugenbewerking nodig. Bij de versie die gebruik maakt van duplicatie worden er vijf subroutines aangeroepen, dus zijn er ook vijf geheugenbewerkingen nodig om het terugkeeradres te overschrijven. Ook hier gebruikt de recursieve versie in de stijl van Dijkstra geen geheugenbewerkingen.

7.4.3 Fibonacci met recursie

De vergelijking van twee programma's die een getal in de fibonacci-reeks berekenen — één in de stijl van Bauer & Samelson, die gebruikt maakt van recursie door duplicatie, de ander in de stijl van Dijkstra die gebruik maakt van recursie door middel van de *stack* — is te vinden in tabel 3. De (recursieve) fibonacci-procedure is in het programma met recursie door duplicatie in totaal vijf keer aanwezig (het origineel plus vier kopieën).

	Bauer & Samelson recursie met duplicatie	**Dijkstra** recursief
Lengte (regels code)	98	37
Aantal stappen	170	230
Bewerkingen op *stack*	31	133
Geheugenbewerkingen	34	0

Tabel 3: Vergelijking van twee `DRAMA`-programma's (tabel 11, 15) om het n-de getal in de fibonacci-reeks te berekenen, met invoer $n = 5$. Uitvoer: 5.

Bovenstaande verklaart direct het verschil in de lengte — gemeten in het aantal regels code — tussen de twee programma's. Daar de fibonacci-procedure in het programma in de stijl van Bauer & Samelson zelf al 17 regels bedraagt, en er hier in totaal 5 kopieën van in het programma staan, zorgen alleen deze duplicaten al voor 85 regels code. De fibonacci-procedure in het programma in de stijl van Dijkstra is weliswaar langer — 28 regels — maar doordat deze maar één keer voor komt blijft het totale aantal regels beperkt.

We zien dat recursie zoals Dijkstra dat voor ogen had inderdaad zorgt voor een minder efficiënt programma tijdens de uitvoering. De programma's in de vorige sectie waren nog betrekkelijk kort, maar om het 5-de getal in de fibonacci reeks uit te rekenen zijn al 15 subroutine aanroepen nodig. Het is duidelijk te zien in tabel 3 dat het extra werk dat Dijkstra moet verzetten om een subroutine aan te roepen zorgt voor een flinke stijging in het aantal stappen dat het programma moet doorlopen om tot een uitkomst te komen. Om dezelfde reden is het aantal keer dat, in de implementatie in de stijl van

Dijkstra, bewerkingen op de *stack* moeten worden uitgevoerd vele malen groter dan het aantal bewerking dat het andere programma gebruikt.

Het is daarentegen wel zo dat het programma in de stijl van Bauer & Samelson veel vaker getallen uit het geheugen moet halen — en regels in het geheugen moet aanpassen om terugkeeradressen te veranderen — waar het programma in de stijl van Dijkstra precies *nul* keer iets opvraagt uit of aanpast in het geheugen.

8 [S] Conclusie

Gauthier van den Hove en Christian Rinderknecht hebben de afgelopen jaren onderzoek gedaan naar het begrip recursie en wat dit precies inhoudt. In hun voetsporen tredende hebben wij het begrip op een pluralistische manier benaderd en de zienswijzen van verschillende belangrijke actoren op dit gebied met elkaar vergeleken en gecontrasteerd. Daartoe is het werk van Friedrich Bauer, Klaus Samelson en Edsger Dijkstra (allen rond 1960) bestudeerd, en is gekeken hoe hun opvattingen passen binnen de theorieën die Van den Hove en Rinderknecht hebben opgesteld.

Omdat computerprogramma's rond 1960 steeds complexer werden (en daardoor ook minder goed leesbaar) stelde Dijkstra zichzelf ten doel bij te dragen aan de ontwikkeling van een universele programmeertaal die elegant en simpel moest zijn. De taal moest zoveel mogelijk op de natuurlijk taal lijken en recursie mocht daarin — net als in de wiskunde en logica — uiteraard niet ontbreken. Dijkstra's implementatie was gebaseerd op een *call stack*, waarbij hij als algemeen geval de recursieve aanroep als uitgangspunt nam. Door deze (vrij conservatieve) opvatting van Dijkstra te vergelijken met een hedendaagse assembleertaal, vallen twee dingen op. Enerzijds zien we dat het concept van de *call stack* zoals Dijkstra dit destijds voor zich zag vandaag de dag nog steeds in grote lijnen gebruikt wordt bij het implementeren van recursie. Wat echter ook opvalt is de hoeveelheid overbodige boekhouding die vereist is om aan de door Dijkstra gewenste algemeenheid te voldoen. In zijn implementatie werd namelijk alle boekhouding tijdens het uitvoeren van subroutines op de *stack* bewaard, hetgeen (vooral in het geval van recursieve procedures) leidde tot zeer inefficiënte code.

Waar Dijkstra op zoek was naar elegantie, lag de nadruk bij Bauer & Samelson vooral op efficiëntie. Het gebruik van computers destijds, eind jaren vijftig, was duur en de gebruikte machines waren niet erg snel. Het was dus van belang dat de programma's die hiervoor ontworpen werden zo snel en efficiënt mogelijk waren. Daar recursieve aanroepen over het algemeen meer boekhouding vereisten en minder efficiënt waren in de uitvoering, waren Bauer & Samelson van mening dat recursie niet thuishoorde in programmeertalen.

Door in de assembleertaal `DRAMA` programma's te schrijven in de stijlen

van zowel Dijkstra als Bauer en Samelson, hebben we een kwantitatieve vergelijking van de verschillende aanpakken kunnen maken. Er wordt al snel duidelijk dat programma's waarin subroutines zichzelf niet aan kunnen roepen — en recursie dus niet gebruikt wordt — korter, sneller en efficiënter zijn als deze geschreven zijn in de stijl van Bauer & Samelson, vergeleken met de (niet-recursieve) programma's in de stijl van Dijkstra.

In plaats van de informatie over de aangeroepen subroutines op de *stack* te bewaren, kozen Bauer & Samelson voor een in die tijd meer gangbare aanpak. In de subroutine werd — wanneer deze werd aangeroepen — de laatste regel overgeschreven door een terugkeersprong naar het aanroepende deel van het programma, met als gevolg dat dezelfde subroutine niet meerdere malen kon worden aangeroepen zonder dat deze werd afgesloten. Een manier om deze belemmeringen te omzeilen was door de subroutines welke recursief aangeroepen zouden worden, meerdere malen in het geheugen te kopiëren, wat ten koste ging van de compactheid en efficiëntie van het programma.

In de `DRAMA`-programma's waar recursieve aanroepen van subroutines wel plaats kunnen vinden, is te zien dat door gebruik te maken van recursie door middel van duplicatie, de lengte van deze programma's erg snel toeneemt. De programmateksten die gebruik maken van recursie op de *stack* — zoals Dijkstra dat voorzag — blijven echter compact en wiskundig elegant. De snelheid van de programma's gemeten in het aantal bewerkingen tot er een uitkomst berekend is, valt echter altijd nadelig uit in de programma's in de stijl van Dijkstra.

Omdat recursie tegenwoordig vrijwel altijd geïmplementeerd wordt met een *call stack*, gebaseerd op de aanpak van Dijkstra, introduceert recursie door middel van duplicatie — op een statische manier — nog een interessante vraag: hoe past recursie door middel van duplicatie in de hedendaagse theorieën over recursie? Van den Hove laat met een aantal voorbeelden zien dat statische recursie (zoals in de programmatekst) en dynamische recursie (tijdens uitvoering van het programma) los van elkaar kunnen bestaan. Deze twee definities worden in deze scriptie uitgebreid, zodat ze niet alleen betrekking hebben op procedures die verwijzen naar zichzelf, maar ook op procedures die verwijzen naar duplicaten van zichzelf. Hieruit blijkt dat ook recursie door middel van duplicatie precies past binnen de voorbeelden die Van den Hove schetst. Echter, door introductie van de disjunctie in de definities van statische en dynamische recursie wordt de theorie wel verzwakt.

Om nog een vollediger beeld te krijgen van het begrip recursie, en dan met name het onderscheid dat Van den Hove maakt tussen statische en dynamische recursie, hebben we het werk van Rinderknecht bestudeerd. Daaruit bleek dat de definities van statische recursie redelijk overeen blijken te komen, maar constateerden we op het gebied van dynamische recursie een interessant verschil in de manier van aanpak. Waar Van den Hove dynamische recursie omschrijft door middel van *stack-based* implementaties in

de stijl van Dijkstra, veralgemeent Rinderknecht dit door gebruik te maken van *call graphs*. Door recursie op deze manier te definiëren in termen van een eigenschap van een wiskundig object, hebben we het verschil tussen de recursie van Dijkstra en de recursie door middel van duplicatie van Bauer en Samelson nog duidelijker kunnen illustreren.

Naast Bauer, Samelson en Dijkstra dachten ook andere wetenschappers eind jaren vijftig na over het nut van recursie. De meeste van hen hadden niet zo'n extreme mening als de eerdergenoemde actoren: Edgar Irons en Wallace Feurzeig ontwierpen een 'hybride' oplossing voor deze twee — schijnbaar tegengestelde — belangen. Hierdoor konden ze de efficiëntie van Bauer en Samelson behouden maar, wanneer dit nodig was, toch de elegante definities van recursie in hun programma's vangen. Tijdens uitvoer van het programma werd gekeken of een subroutine op een recursieve manier werd aangeroepen. Wanneer dit het geval was werd het mechanisme om deze recursieve aanroepen te kunnen behandelen in werking gesteld. Wanneer er geen sprake was van een recursieve subroutine aanroep, behandelden ze de subroutine op een efficiënte manier, zoals Bauer en Samelson.

Henry Gordon Rice daarentegen was als logicus goed op de hoogte van recursie en haar nut in de berekenbaarheidstheorie, maar was als programmeur falikant gekeerd tegen het gebruik van recursie in computerprogramma's. Mede door zijn logische achtergrond wist hij dat het altijd mogelijk was om een programma dat gebruik maakt van recursieve aanroepen om te schrijven in een programma dat van recursieve aanroepen geen gebruik maakt.

8.1 Implicaties

Door het multidisciplinaire karakter van het vakgebied dat we 'Kunstmatige Intelligentie' noemen, en het karakter van recursie binnen dit werkveld, is gedegen kennis van de verscheidene zienswijzen op dit begrip van belang. Uit ons onderzoek is duidelijk geworden dat er niet één overkoepelende theorie over recursie is die al haar aspecten kan verklaren. De onderlinge samenhang — en ook juist de punten waarop zij verschillen — tussen de geschetste kaders is verduidelijkt, waardoor communicatie tussen wetenschappers en geïnteresseerden uit verschillende vakgebieden gemakkelijker kan verlopen.

Ook voor studenten die net te maken krijgen met recursie en proberen te achterhalen wat dit begrip precies precies betekent, kan dit werk als uitgangspunt dienen om hun eigen mening over dit begrip te vormen.

8.2 Vervolgonderzoek

In dit onderzoek is vooral gekeken wat recursie betekende voor programmeurs — en later computerwetenschappers — van de jaren vijftig tot nu. Hierbij is ook kort het standpunt van de logicus Rice besproken. Het is interessant om vanuit deze logische hoek het begrip recursie verder te belichten

en te kijken hoe deze visie contrasteert met de zienswijzen die al besproken zijn.

Ook in de taalkunde wordt gebruik gemaakt van recursie, bijvoorbeeld in de definitie van contextvrije grammatica's. Door ook de visie op recursie van actoren — zoals Noam Chomsky — uit de taalkundige hoek te bestuderen zal een nog completer beeld over recursie ontstaan.

A [R] De stack

In 'Sequential Formula Translation' beschreven Bauer & Samelson een toepassing van de *stack* — zij noemen deze zelf *cellar* — in het uitvoerbaar maken van aritmetische expressies. Hiermee legden ze de fundamenten voor de technieken die Dijkstra later zou gebruiken voor zijn implementatie van het aanroepen van subroutines in de ALGOL-taal. Om deze zaken goed te begrijpen is het van belang de werking van de *stack* zelf ook te bevatten.

De *stack* is een deel van het geheugen waarbij elementen die opgeslagen worden niet vrij toegangelijk zijn, maar dat werkt volgens het zogenaamde *last in, first out*-principe. Slechts het element dat het meest recent aan de *stack* is toegevoegd kan er nu weer afgehaald worden om verder gebruikt te worden. Een veel gebruikte vergelijking is die met een stapel borden: wanneer je een reeks borden bovenop elkaar plaatst is het vreselijk onhandig om een bord uit het midden van de stapel te pakken, slechts het bord dat je als laatst op de stapel hebt gelegd — het bovenste bord dus — kun je weer van de stapel afhalen, waarna het bord dat daaronder ligt toegankelijk wordt.

Implementaties van een *stack* tegenwoordig maken gebruik van een aantal methoden en principes die verdere uitleg verdienen. Wanneer je een *stack* wilt gebruiken in een programma, dien je voor deze *stack* een aaneengesloten *block* geheugenruimtes te reserveren, minimaal zo groot als de maximale grootte die je verwacht nodig te hebben. De *stack pointer* (SP) verwijst naar het bovenste element op de *stack*, wanneer de *stack* leeg is zal de *stack pointer* de waarde 0 hebben. Zitten er elementen op de *stack* boven de *stack pointer* dan worden deze beschouwd als rommel en zijn niet meer toegankelijk.

Omdat je niet zomaar willekeurig toegang hebt tot elementen in de *stack*, zijn er twee[13] methodes die je helpen elementen op de goede plek, en van de goede plek terug, te krijgen:

Push De *push*-methode stelt je in staat om elementen aan de *stack* toe te voegen. Er wordt een element toegevoegd aan de *stack* op de plek waar de *stack pointer* naar wijst, daarna wordt de *stack pointer* met één verhoogd.

Pop De *pop*-methode verschaft toegang tot elementen die in de *stack* zitten, maar wel slechts tot het bovenste element. De *stack pointer* wordt allereerst met één verlaagd, het element waar de *stack pointer* nu naar wijst wordt terug gegeven.

In theorie heb je slechts toegang tot het bovenste element van de *stack*, in de praktijk bleek het echter handig — en soms zelfs noodzakelijk — om

[13] Sommige implementaties van de *stack* kennen nog een derde methode, de *peek* methode, welke toegang verschaft tot het bovenste element van de *stack* zonder de waarde van de *stack pointer* te veranderen.

elementen uit het midden van de *stack* op te vragen. Verschillende actoren gebruikten hiervoor verschillende technieken, bijvoorbeeld de *reference pointer* die Dijkstra in 1960 introduceerde.

B [R] DRAMA Programma's

Hieronder volgen enkele programma's die we hebben geschreven in DRAMA, zoals wij denken dat het door respectievelijk Bauer en Dijkstra gedaan zou zijn.[14] Programma's voorafgegaan door een B zijn in de stijl van Bauer geschreven, programma's vooraf gegaan door een D zijn in de stijl van Dijkstra gegeven. Enige voorkennis van DRAMA is nodig om deze programma's te beschrijven en begrijpen, al hebben we geprobeerd de werking duidelijk uit te leggen door middel van commentaar.

We hebben de DRAMA programma's gemaakt, geschreven en uitgevoerd in de DRAMA simulator van Tom Schrijvers van de KU Leuven, uit het jaar 2000. Informatie over de programmeertaal DRAMA en hoe deze te gebruiken hebben we gehaald uit het eerste deel van het studieboek 'Structuur en organisatie van computersystemen.' [7]

Programma's geschreven in DRAMA bestaan uit lijsten van zogenaamde *bevelen*, in de volgende sectie (tabel 4) is een lijst te vinden van de bevelen die door ons gebruikt zijn om de programma's te schrijven. De lijst met bevelen die samen het programma vormen wordt opgeslagen op de eerste plekken van hetzelfde geheugen dat later, bij uitvoer van het programma, gebruikt zal worden om variabelen, resultaten en ook bijvoorbeeld de *stack* bij te houden. Doordat programma's in dit geheugen worden opgeslagen is het mogelijk om het programma *at run-time* te wijzigen, hetgeen vandaag de dag als een slechte gewoonte wordt gezien [3, p.205].

De bevelen in letters worden door de vertaler omgezet in getallen van tien cijfers, weer opgedeeld in vijf velden (zie figuur 10):

- Twee cijfers voor de *functiecode* (fc), die de verschillende instructies onderscheiden.
- Twee cijfers voor een *modusveld* (modus), die aangeeft of het om een adres of een getal gaat.
- Eén cijfer om aan te geven welke *accumulator* (acc) gebruikt wordt in de instructie.
- Eén cijfer voor het *indexregister* (ind).
- Vier cijfers voor de *operand* (operand), waar een adres of heel getal opgeslagen wordt.

[14] Deze inleiding tot de DRAMA-programmeertaal werd geschreven door Ruben Groot Nibbelink, bij de programma's zelf staat vermeld door welke student dit programma geschreven werd.

Vooral de code van de `SPR` (de sprong) instructie is voor ons belangrijk, daar Bauer en Samelson de school van Wilkes volgden bij het aanroepen van een subroutine. Op de regel waar de terugkeerinstructie komt te staan wordt eerst door de programmeur een dummy instructie geplaatst. Deze regel wordt later door de *interpreter* overschreven met een spronginstructie naar de plek waar het programma verder gaat nadat de subroutine is voltooid. De `SPR`-instructie in `DRAMA` bestaat uit de tien cijfers `3221900000`, waarbij de laatste vier nullen verwijzen naar het regelnummer waarnaartoe gesprongen moet worden. Om de sprong naar de goede plek te laten verwijzen is het dus voldoende om het regelnummer waar naar gesprongen moet worden op te tellen bij eerdergenoemde code. Het resultaat van deze som is de code van de instructie die een sprong naar het regelnummer aanduidt.

Overzicht van DRAMA-instructies

In tabel 4 is een lijst[16] te vinden met door ons gebruikte bevelen in `DRAMA` en hun werking, `Rr` en `Rs` staan voor registers en `adres` voor een adres in het geheugen.

`DRAMA` kent een voorgedefinieerd subroutine bevel (`SBR adres`), dat ongeveer zo geïmplementeerd is als Dijkstra dat voorzien had. Wij hebben er echter in eerste instantie voor gekozen om dit bevel niet te gebruiken en deze handeling zelf te programmeren, zodat we de verschillen tussen Bauer en Dijkstra beter kunnen vergelijken. De grootste verschillen waar wij naar op zoek zijn zitten immers in de manier waarop subroutines worden aangeroepen, en hoe recursie daar in past. Later, wanneer we de verschillen in subroutineaanroepen hebben kunnen bestuderen, gebruiken we deze instructie wel omdat dit de leesbaarheid van de programma's ten goede komt.

De waarde van de spronginstructie wordt — in programma's in de stijl van Bauer & Samelson — opgeslagen onder de naam `sprc`, voor **spr**ong**c**ode. In `DRAMA` kunnen getallen van meer dan vier cijfers niet direct aan een optelinstructie worden meegegeven, grotere getallen moeten onder hun eigen naam worden opgeslagen.

[15] Figuur 1-3 uit [7, p.30].

[16] Dit is een deel van de volledige lijst [7, p.236-238] met `DRAMA`-bevelen uit 'Structuur en organisatie van computersystemen.'

	fc	**modus**	**acc**	**ind**	**operand**
aantal cijfers:	2	2	1	1	4

Figuur 10: Inwendige voorstelling van `DRAMA`-instructie.[15]

Instructie		Effect
HIA	Rr,Rs	Waarde van Rs wordt opgeslagen in Rr
HIA	Rr,adres	Waarde op geheugen[adres] wordt opgeslagen in Rr
HIA.w	Rr,getal	Waarde van getal wordt opgeslagen in Rr
BIG	Rr, adres	Berg waarde van Rr op in geheugen[adres]
HST	Rr	Haal van stack en stop in Rr (pop)
BST	Rr	Berg waarde van Rr op stack (push)
OPT	Rr,Rs	Tel waarde van Rs op bij Rr (Rr += Rs)
OPT.w	Rr,getal	Tel getal op bij Rr (Rr += getal)
AFT	Rr,Rs	Trek waarde van Rs af van Rr (Rr -= Rs)
AFT.w	Rr,getal	Trek getal af van Rr (Rr -= getal)
VER	Rr,Rs	Vermenigvuldig Rr met Rs (Rr *= Rs)
VER.w	Rr,getal	Vermenigvuldig Rr met getal (Rr *= getal)
VGL	Rr,Rs	Vergelijk Rr met Rs
VGL.w	Rr,getal	Vergelijk Rr met getal
SPR	adres	Spring naar adres
SPR	0(Rr)	Springt naar waarde van Rr als adres
VSP	vw,adres	Als aan voorwaarde vw is voldaan, spring naar adres
SBR	subroutine	Sprint naar de opgegeven subroutine en plaatst terugkeeradres op de stack
KTG		Keert terug uit subroutine, haalt terugkeeradres van de stack
LEZ		Vraag inputwaarde, stop in R0
DRU		Druk waarde van R0 af op scherm
STP		Stop uitvoer van programma

Tabel 4: Gebruikte instructies met hun betekenis.

B.1 [R] B - subroutineaanroep

Het programma in tabel 6 op pagina 56 was één van de eerste programma's die we schreven, om te testen hoe de subroutine aanroep op de manier van Bauer werkt. Het programma vraagt om één getal als input, en roept vervolgens twee keer de subroutine *pluseen* aan die het gegeven getal met één ophoogt. De output is het resultaat, wanneer deze twee subroutines voltooid zijn; ofwel: $output = input + 2$.

Er wordt gebruik gemaakt van één register, R0, en er wordt plek gereserveerd voor één variabele, *a*. In regel 3 wordt de subroutine voor de eerste keer aangeroepen, daarvoor wordt in regel 2 eerst het adres van het regelnummer waar na afloop van de subroutine het programma verder moet gaan — regel 4 — in de accumulator gestopt.

Het *main*-programma maakt in regel 3 de sprong naar de subroutine in regel 10, waar het regelnummer waar straks naar teruggesprongen moet wor-

den, wordt opgeteld bij de in `sprc` opgeslagen sprongcode. Vervolgens wordt in regel 11 de dummy-instructie, op de plek waar terugkeer uit de subroutine moet plaatsvinden, overschreven door de nieuw gevormde sprongcode.

Wanneer het programma dus voor de eerste keer aan het einde van de subroutine *pluseen* komt, is de code van deze regel overschreven door de code *3221900004*, en springt het programma terug naar de regel na de subroutineaanroep. Op eenzelfde manier wordt de tweede subroutine aangeroepen en verlaten, doch met andere regelnummers.

B.2 `[R]` B - Twee subroutine's

Hoe een subroutine zich gedraagt als hij wordt aangeroepen vanuit een andere subroutine laten we zien in programma 7 op pagina 57. Dit programma vraagt om twee getallen als input, de eerste wordt opgeslagen in variabele *a*, de tweede in variabele *i*. De subroutine *pluseen* is dezelfde subroutine als in het programma uit sectie B.1. De tweede subroutine is *plusx*, en roept de subroutine *pluseen* *i* maal aan. In feite berekent dit programma dus de som $output = input_a + input_i$, door het getal *a*, *i* maal met 1 te verhogen.

Om meerdere malen de subroutine *pluseen* uit te voeren wordt in de subroutine *plusx* gebruik gemaakt van een *while loop*, door telkens *i* met 1 te verlagen tot het de waarde 0 heeft, waarna de loop van het programma terugkeert naar het *main* gedeelte.

Het aanroepen van de subroutine's werkt op dezelfde manier als in sectie B.1 uiteen werd gezet, met één belangrijk verschil: wanneer het programma de eerste keer in het geheugen wordt geladen staat op het einde van beide subroutines een dummy-instructie. De eerste keer dat routine *pluseen* wordt aangeroepen wordt deze instructie overschreven door de terugkeerinstructie, de volgende keren wordt die terugkeerinstructie overschreven door de nieuwste terugkeerinstructie.

B.3 `[R]` B - Faculteit

De faculteitsfunctie is een klassiek voorbeeld van een functie waar je een compacte, elegante recursieve definitie voor kunt geven:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \times (n-1)! & \text{if } n > 0. \end{cases}$$

Of, omgeschreven in pseudocode van een hogere-orde programmeertaal:

```
int faculteit(int n) {
    if n == 0:
        return 1;
    else:
        return n * faculteit(n-1);
}
```

Aan dit stuk code is duidelijk te zien dat deze definitie van de faculteitsfunctie staartrecursief is, na de recursieve aanroep van de faculteitsfunctie zijn er geen andere instructies die nog moeten worden uitgevoerd. Zoals van den Hove [17, p.3] ook verteld in zijn artikel, is het mogelijk (en in sommige programmeertalen zelfs noodzakelijk) om een staartrecursieve functie om te schrijven in een *loop*.

Van deze mogelijkheid hebben we gebruik gemaakt om de faculteitsfunctie op de Bauer-manier te implementeren in programma 8 op pagina 58. In regel 2 wordt gesprongen naar de faculteit-subroutine; aan het einde van deze subroutine (op regel 18) is de instructie te vinden die wordt overschreven door de terugkeerinstructie. Deze instructie is alleen te bereiken via de voorwaardelijke sprong in regel 10, waar wordt gekeken of het getal in de accumulator (n) gelijk is aan 1, als dat niet het geval is wordt de body van de faculteitsfunctie doorlopen waar het rekenwerk plaatsvind en n met 1 verlaagd wordt.[17] Na het rekenwerk springt het programma terug naar het begin van de faculteitsfunctie, waar weer wordt gekeken of n gelijk is aan 1.

B.4 `[R]` B - Functie

Om verschillen tussen de stijlen van Bauer en Dijkstra inzichtelijk te maken hebben we ervoor gekozen om de volgende arithmetische expressie op beide manieren uit te werken:

$$a \times (b + c)^2$$

Deze functie bestaat uit een aantal belangrijke elementen die beide scholen op een verschillende manier aanpakken. Zo is de volgorde van bewerkingen belangrijk (de haakjes om $b + c$ hebben voorrang), en wordt het kwadraat uitgerekend door een aparte subroutine.

In programma 9 op pagina 59 is de uitwerking op de manier van Bauer te vinden. Om deze code te krijgen hebben we eerst een vertaler geschreven in `JAVA` (te vinden in appendix C) die als input een arithmetische expressie vraagt, en deze vertaalt in een lijst assembleer-instructies, volgens de regels die Bauer en Samelson geven op pagina 78 [16]. Echter, voor de aanroep

[17] Hier wordt gebruik gemaakt van het feit dat de recursieve aanroep van de faculteitsfunctie gestopt kan worden als n gelijk is aan 1, daar bij $n = 0$ en $n = 1$ het gevonden resultaat nog twee keer vermenigvuldigd wordt met 1 en dus niet meer verandert.

van een subroutine werd geen prioriteit aangegeven, dus deze hebben we zelf zorgvuldig een plek gegeven in de bijgevoegde code. De output van de `JAVA`-vertaler is te vinden in tabel 5. De notatie η_i wordt gebruikt door Bauer en Samelson om het i-de element op de *stack* aan te duiden. De instructies in de output hebben we zelf verwerkt tot een `DRAMA`-programma (programma 9).

`Input`	`Output`
`a*(b+c)`	$a \Rightarrow \eta_1$ $b \Rightarrow \eta_2$ $c \Rightarrow \eta_3$ $\eta_2 \Rightarrow \eta_2 + \eta_3$ `**kwadraat`[18]`**` $\eta_1 \Rightarrow \eta_1 \times \eta_2$

Tabel 5: Input en Output van de `JAVA` vertaler.

B.5 `[R]` B - Faculteit met recursie door middel van duplicatie

Het programma in tabel 10 op pagina 60 laat zien hoe de faculteitsfunctie berekend kan worden door recursie door middel van duplicatie te gebruiken.

In deze implementatie van een programma om de faculteitsfunctie te berekenen is ervoor gekozen om van procedure `fac` in totaal vijf versies op te nemen, het programma geeft dus een output voor input $n \leq 5$. In regel 3 wordt de eerste versie van de `fac` subroutine aangeroepen — aangeduid met `fac1` — welke op zijn beurt de volgende oproept — `fac2` — enzovoort.

In de laatste kopie van de `fac`-subroutine — `fac5` — staat op de plek van de recursieve aanroep een stopcode. Mocht het programma dus worden aangeroepen met een invoer $n > 5$ dan zal het programma stoppen.

B.6 `[R]` B - Fibonacci

Het programma in tabel 11 op pagina 61 berekent het n-de getal van de reeks van fibonacci, waarbij het getal n als invoer wordt meegegeven. Dit programma maakt gebruik van recursie door middel van duplicatie, en er zijn dan ook 5 kopieën van de fibonacci functie (`fib1`...`fib5`) in het programma aanwezig. Dit betekent dat het programma uitvoer geeft voor invoer $0 \leq n \leq 5$. Wanneer er een groter getal wordt ingevoerd zal het programma stoppen omdat de sprongcode naar het volgende duplicaat in de laatste kopie (`fib5`) is vervangen door een stop-code.

De waarde van het huidige argument dat als parameter wordt meegegeven aan de volgende (recursieve) aanroep wordt opgeslagen in een — voor

[18] Deze regel komt niet uit de vertaler, maar hebben we zelf op deze plek toegevoegd.

elke procedure aparte — variabele. Wanneer de waarde van twee recursieve aanroepen is berekend worden deze in de vorige aanroep bij elkaar opgeteld en op de *stack* gezet.

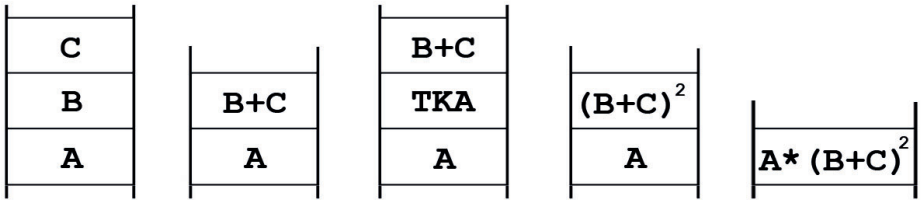
Wanneer de eerste aanroep van fibonacci wordt afgesloten staat op de *stack* slechts nog de uitkomst. Deze wordt van de *stack* gehaald en afgedrukt.

B.7 [B] D - Functie

In het programma in tabel 12 op pagina 62 is de uitwerking van een programma te vinden dat dezelfde functie, $a \times (b + c)^2$ in de stijl van Dijkstra berekent. Wat opvalt in deze implementatie is het feit dat de *stack* veel meer gebruikt wordt dan bij Bauer en Samelson. Waar de Duitsers het meeste werk al tijdens *compile-time* gedaan wilden hebben, gaf Dijkstra de voorkeur aan een meer dynamische oplossing waarbij de omvang van de *stack* pas bekend wordt tijdens uitvoering van het programma. Opmerkelijk is ook de hoeveelheid "overbodig" werk dat gedaan wordt, we hebben hiervoor gekozen om te laten zien hoe belangrijk algemeenheid voor Dijkstra was. De meest simpele instructies moesten keer op keer netjes uitgevoerd worden, zodat elke stap in het programma op dezelfde manier behandeld wordt. Op regel 9 en 10 van programma 12 is hiervan een voorbeeld te zien. Het resultaat van de optelling op regel 8 moet eerst als tussenresultaat op de *stack* worden geplaatst (regel 9). Op regel 10 wordt deze waarde echter direct weer van de *stack* gehaald om op regel 13 als parameter meegegeven te worden wanneer de subroutine *kwadraat* aangeroepen wordt. In figuur 11 is te zien hoe de *stack* zich ontwikkeld tijdens uitvoering van dit programma. Vergelijken we dit met de functie-implementatie van Bauer en Samelson dan vallen twee dingen op: Ten eerste is de code van de Duitsers aanzienlijk korter, ten tweede heeft het programma van Dijkstra veel meer stack-operaties nodig. Het spreekt voor zich dat de implementatie van Bauer en Samelson dan ook veel efficiënter is dan die van Dijkstra.

B.8 [B] D - Twee subroutines

Het volgende programma in Dijkstra-stijl, het programma in tabel 13 op pagina 63, heeft als doel om twee kenmerken te illustreren: Het laat zien hoe een subroutine (in het voorbeeld, subroutine B) zich gedraagt wanneer deze wordt aangeroepen vanuit een andere subroutine (subroutine A) en hoe lokale variabelen die binnen A gedeclareerd zijn, door middel van *random access* in subroutine B gebruikt kunnen worden. De functionaliteit van dit programma is verder niet bijster interessant en zou in pseudocode voor een imperatieve hogere-orde programmeertaal als volgt beschreven kunnen worden:



Figuur 11: De *stack* tijdens uitvoering van $a \times (b + c)^2$.

```
a() {
    int x = 5;
    int y = 6;
    b();
}
b() {
    int z = 3;
    print(z * x);
}
```

In de vorige voorbeelden hebben we gekozen om de springinstructie `SPR` te gebruiken bij de aanroep van een subroutine, om beter te kunnen illustreren hoe het terugkeeradres op de *stack* wordt gezet en hoe deze bij voltooiing van de subroutine er weer af wordt gehaald. In de voorbeelden die nog volgen maken we gebruik van de ingebouwde instructie `SBR`, dat staat voor subroutine. In principe doet deze instructie precies hetzelfde als wat wij eerder deden, met als verschil dat het regelnummer waarnaar gesprongen wordt (de regel direct volgend op de regel van de routine-aanroep) nu automatisch gegenereerd wordt en dus niet expliciet in de code staat. De `KTG`-instructie zorgt ervoor dat teruggekeerd wordt naar het regelnummer dat op dat moment in het bovenste element van de *stack* staat. Het is dus van belang dat na voltooiing van een subroutine, de *stack* er precies hetzelfde uitziet als toen deze aangeroepen werd. De reden dat we er daarom voor gekozen hebben deze `SBR`-instructie te gebruiken, is dat dit principe (de subroutine die elementen op de *stack* plaatst is zelf ook verantwoordelijk om ze er weer af te halen) perfect aansluit bij Dijkstra's opvattingen. Hiernaast komt het gebruik van deze instructie de leesbaarheid van de programma's ten goede.

Deze implementatie is vooral geschreven om te laten zien hoe door middel van *parameter pointers* gebruik kan worden gemaakt van elementen die niet

bovenop de *stack* liggen. In de sectie 'Random access' van het hoofdstuk over Dijkstra wordt uitgelegd hoe dit in zijn werk gaat. Het gebruik van registers in `DRAMA` behoeft nog enige uitleg; Het register `R9` wordt standaard gebruikt om de waarde van de *stack pointer* in te bewaren, het register `R8` zal altijd verwijzen naar de waarde van de *parameter pointer*.

Op regel 23 en 24 van dit programma is te zien hoe de *parameter pointer* van subroutine A (de waarde hiervan is opgeslagen in de *link* van subroutine B) gebruikt wordt om de juiste lokale variabele op te halen uit de *stack*. De *parameter pointer* van A verwijst altijd naar het eerste element van *block* A, en aangezien A geen functieargumenten heeft moet de eerste lokale variabele één plek boven de *parameter pointer* te vinden zijn. Na de vermenigvuldiging en het printen wordt *block* B van de *stack* gehaald, *block* A van de *stack* gehaald en is het programma voltooid.

B.9 `[B]` D - Faculteit

Er zijn een aantal problemen waarvoor geldt dat een recursieve definitie de meest elegante en leesbare oplossing is. Zo kan de fibonacci-functie (die het n-de getal in de fibonacci-reeks berekent) bijvoorbeeld op een mooie intuïtieve manier recursief gedefinieerd worden:

```
fib(int n) {
    if(n <= 1) return n
    else return fib(n-1) + fib(n-2)
}
```

Een ander klassiek voorbeeld is de faculteitsfunctie, waarvoor in B.3 reeds een pseudocode gegeven is. Omdat Bauer en Samelson geen gebruik wilden maken van recursie, moesten zij een alternatief verzinnen om de faculteit van een waarde n uit te rekenen. Zoals in sectie B.3 uitgelegd, wordt deze functie daarom omgeschreven naar een *loop*.

In tabel 14 is te zien dat met het recursieve gebruik van de *stack* het idee van deze functie bijna letterlijk in de implementatie vertaald kan worden. Er is eigenlijk maar één subroutine, en dat is *fac()*. Dit komt omdat *base*, waar de voorwaardelijke sprong naartoe gaat als $n \leq 1$, slechts een onderdeel is van de subroutine *fac()*. Er wordt geen terugkeeradres voor *base* op de *stack* gezet en daarom wordt gelijk weer naar regel 11 gesprongen om de vermenigvuldigingen uit te voeren. Na de laatste vermenigvuldiging ($tussenresultaat \times n$, in het geval van *fac(3)* dus 2×3) keert het programma terug naar de regel waar *fac()* voor het eerst aangeroepen werd (regel 3) om tenslotte het resultaat af te drukken.

B.10 `[R]` D - Fibonacci

Het programma in tabel 15 op pagina 64 berekent het n-de getal van de fibonacci-reeks, waarbij n als input wordt meegegeven. Dit programma

werkt met recursie in de stijl van Dijkstra door informatie over subroutine aanroepen op de *stack* te zetten.

Wanneer de fibonacci-subroutine (`fib`) wordt aangeroepen, en het argument dat wordt meegegeven kleiner of gelijk is aan 1 dan wordt direct naar het basisgeval gesprongen en de waarde van het argument (0 of 1) op de *stack* gezet. Wanneer het argument groter is dan één wordt het met één verminderd, wordt dit getal op de *stack* gezet met een terugkeeradres. Vervolgens wordt het argument weer met één verminderd en met een terugkeeradres op de *stack* gezet waarna de subroutine twee maal recursief wordt aangeroepen.

Dit vergt wel wat gegoochel met de volgorde van de waarden op de *stack*, vandaar dat er in regel 20-25 van het programma elementen in de *stack* op een andere volgorde worden gezet. Wanneer het programma terugkeert naar de plek waar `fib` voor de eerste keer werd aangeroepen staat alleen de uitkomst nog op de *stack*, deze wordt daar vanaf gehaald en uitgeprint.

B.11 Programma's

```
 0                LEZ
 1                BIG     R0,a        Berg dit getal op in variable a
 2                HIA.w   R0,4        Haal getal 4 in accumulator (terugkeeradres)
 3                SPR     pluseen     Spring naar subroutine pluseen
 4                BIG     R0,a        Berg getal in R0 op in variabele a
 5                HIA.w   R0,7        Haal getal 7 in accumulator (terugkeeradres)
 6                SPR     pluseen     Spring naar subroutine pluseen
 7                HIA     R0,a        Haal waarde van a in accumulator R0
 8                DRU                 Druk dit getal af
 9                STP                 Stop het programma
10   pluseen:    OPT     R0,sprc     Tel het getal in sprc op bij de waarde in R0
11                BIG     R0,15       Berg het verkregen getal op de plek van de terugkeersprong
12                HIA     R0,a        Haal de waarde van variabele a in R0
13                OPT.w   R0,1        Tel 1 op bij de waarde in R0
14                BIG     R0,a        Berg waarde van R0 op in variabele a
15                 STP                Dummy instructie, wordt in 11 overschreven door terugkeersprong
16      sprc:     3221900000          Waarde van de SPR-instructie
17         a:     RESGR   1           Reserveer geheugen voor variabele a
                  EINDPR
```

Tabel 6: Een DRAMA programma met subroutineaanroepen op de manier van Bauer en Samelson.

0		LEZ		Lees een getal (input)
1		BIG	R0,a	Berg dit getal op in variabele *a*
2		LEZ		Lees een getal (input)
3		BIG	R0,i	Berg dit getal op in variabele *i*
4		HIA.w	R0,6	Haal het terugkeeradres in accumulator
5		SPR	plusx	Spring naar subroutine *plusx*
6		HIA	R0,a	Haal de waarde van *a* in R0
7		DRU		Druk de waarde in R0 af
8		STP		Stop het programma
9	pluseen:	OPT	R0,sprc	Tel het getal in *sprc* op bij de waarde in R0
10		BIG	R0,14	Berg dit getal op de plek van de terugkeersprong
11		HIA	R0,a	Haal de waarde van *a* in R0
12		OPT.w	R0,1	Tel daar *1* bij op
13		BIG	R0,a	Berg deze waarde in variabele *a*
14		STP		Dummy instructie, wordt overschreven door terugkeersprong
15	plusx:	OPT	R0,sprc	Tel het getal in *sprc* op bij de waarde in R0
16		BIG	R0,26	Berg dit getal op de plek van de terugkeersprong
17	wh:	HIA	R0,i	begin while-loop: Haal waarde van *i* in accumulator
18		VGL.w	R0,0	vergelijk deze waarde met het getal *0*
19		VSP	KLG,endwh	is *i* kleiner of gelijk aan 0, spring dan uit de while loop
20		HIA.w	R0,22	Haal het terugkeeradres in de accumulator
21		SPR	pluseen	Spring naar subroutine *pluseen*
22		HIA	R0,i	Haal de waarde van *i* in de accumulator
23		AFT.w	R0,1	Trek daar *1* van af
24		BIG	R0,i	Berg dit getal op in *i*
25		SPR	wh	Spring naar het begin van de while loop
26	endwh:	STP		Dummy instructie, wordt overschreven door terugkeersprong
27	sprc:	3221900000		Numerieke waarde van de sprong-instructie
28	a:	RESGR	1	Reserveer geheugen voor variabele *a*
29	i:	RESGR	1	Reserveer geheugen voor variabele *i*
		EINDPR		

Tabel 7: Een programma met twee subroutines in de stijl van Bauer en Samelson.

```
 0          LEZ             Lees een getal
 1          HIA.w   R1,3     Haal terugkeeradres 3 in register 1
 2          SPR     fact    Spring naar subroutine 'fact'
 3          HST     R0      'Pop' van stack in R0
 4          DRU             Druk dit getal af
 5          STP             Stop het programma
 6   fact:  OPT     R1,sprc Tel het getal in sprc op bij waarde in R1
 7          BIG     R1,18   Berg het verkregen getal op de plek van de terugkeersprong
 8          BST     R0      Berg het getal in R0 op de stack (push)
 9   fact2: VGL.w   R0,1    Vergelijk dit getal met 1
10          VSP     GEL,18  Is het gelijk aan 1, spring dan naar de terugkeersprong
11          AFT.w   R0,1    Zo niet, trek 1 van dit getal af
12          BST     R0      Berg het verkregen getal op de stack (push)
13          HST     R2      Haal bovenste getal van de stack in register 2
14          HST     R3      Haal bovenste getal van de stack in register 3
15          VER     R3,R2   Vermenigvuldig deze twee getallen
16          BST     R3      Berg het verkregen getal op de stack
17          SPR     fact2   Spring naar het 'fact2'-gedeelte van de subroutine
18          STP             Dummy instructie, wordt in 7 overschreven door terugkeersprong
19   sprc:  3221900000      Waarde van de SPR-instructie
            EINDPR
```

Tabel 8: DRAMA programma voor de faculteitsfunctie in de stijl van Bauer en Samelson.

0		LEZ		Lees een getal (a)
1		BST	R0	Berg dit getal op de stack (push)
2		LEZ		Lees een getal (b)
3		BST	R0	Berg dit getal op de stack (push)
4		LEZ		Lees een getal (c)
5		BST	R0	Berg dit getal op de stack (push)
6		HST	R0	Haal het bovenste getal van de stack (pop)
7		HST	R1	Haal het bovenste getal van de stack (pop)
8		OPT	R0,R1	Tel de waarde van R1 op bij R0
9		BST	R0	Berg het verkregen getal op de stack (push)
10		HIA.w	R0,13	Haal getal *12* in de accumulator (terugkeeradres)
11		HST	R1	Haal de parameter voor de functieaanroep van de stack in register *1*
12		SPR	kwadraat	Spring naar de 'kwadraat' subroutine
13		BST	R1	Push het resultaat van de kwadraat-subroutine op de stack
14		HST	R0	Haal het bovenste getal van de stack (pop)
15		HST	R1	Haal het bovenste getal van de stack (pop)
16		VER	R0,R1	Vermenigvuldig deze getallen met elkaar
17		BST	R0	Berg het resultaat op de stack
18		HST	R0	Haal het bovenste getal (antwoord) van de stack
19		DRU		Druk dit getal af
20		STP		Stop het programma
21	kwadraat:	OPT	R0,sprc	Tel het getal in *sprc* op bij de waarde in R0
22		BIG	R0,24	Berg het verkregen getal op de plek van de terugkeersprong
23		VER	R1,R1	Vermenigvuldig het getal in R1 met zichzelf
24		STP		Dummy instructie, wordt in *22* overschreven door terugkeersprong
25	sprc:	3221900000		Waarde van de SPR-instructie
26		EINDPR		

Tabel 9: DRAMA programma in de stijl van Bauer en Samelson voor berekening van de functie $a \times (b + c)^2$.

```
 0          LEZ              Lees een getal
 1          HIA.w   R1,3     Haal terugkeeradres in register R1
 2          SPR     fac1     Spring naar de originele faculteitsubroutine
 3          HST     R0       Haal het resultaat van de stack
 4          DRU              Druk het resultaat af
 5          STP              Stop het programma
-------------------------------------------------------------------------------
 6  fac1:   OPT     R1,sprc  Tel het getal in sprc op bij waarde in R1
 7          BIG     R1,18    Berg het verkregen getal op de plek van de terugkeersprong
 8          VGL.w   R0,1     Vergelijk parameter met 1
 9          VSP     KLG,17   Als het 1 of 0 is, spring naar het basisgeval
10          BST     R0       Zo niet, berg hem op de stack
11          AFT.w   R0,1     Trek er één van af
12          HIA.w   R1,14    Haal terugkeeradres in register R1
13          SPR     fac2     Spring naar eerste kopie van de faculteitssubroutine
14          HST     R0       Haal bovenste getal van stack
15          HST     R1       Haal volgende getal van stack
16          VER     R0,R1    Vermenigvuldig met elkaar in R0
17          BST     R0       Berg het resultaat op in de stack
18          STP              Dummy instructie, wordt overschreven door terugkeersprong
-------------------------------------------------------------------------------
19  fac2:   OPT     R1,sprc  Tel het getal in sprc op bij waarde in R1
20          BIG     R1,31    Berg het verkregen getal op de plek van de terugkeersprong
21          VGL.w   R0,1     Vergelijk parameter met 1
22          VSP     KLG,30   Als het 1 of 0 is, spring naar het basisgeval
23          BST     R0       Zo niet, berg hem op de stack
24          AFT.w   R0,1     Trek er één van af
25          HIA.w   R1,27    Haal terugkeeradres in register R1
26          SPR     fac3     Spring naar tweede kopie van de faculteitssubroutine
27          HST     R0       Haal bovenste getal van stack
28          HST     R1       Haal volgende getal van stack
29          VER     R0,R1    Vermenigvuldig met elkaar in R0
30          BST     R0       Berg het resultaat op in de stack
31          STP              Dummy instructie, wordt overschreven door terugkeersprong
-------------------------------------------------------------------------------
32  fac3:   OPT     R1,sprc  Tel het getal in sprc op bij waarde in R1
 ⋮          ⋮                ⋮
        kopie van de originele functie, met eigen subroutine-aanroepen en terugkeerwaardes
 ⋮          ⋮                ⋮
44          STP              Dummy instructie, wordt overschreven door terugkeersprong
-------------------------------------------------------------------------------
45  fac4:   OPT     R1,sprc  Tel het getal in sprc op bij waarde in R1
 ⋮          ⋮                ⋮
        kopie van de originele functie, met eigen subroutine-aanroepen en terugkeerwaardes
 ⋮          ⋮                ⋮
57          STP              Dummy instructie, wordt overschreven door terugkeersprong
-------------------------------------------------------------------------------
58  fac5:   OPT     R1,sprc  Tel het getal in sprc op bij waarde in R1
59          BIG     R1,70    Berg het verkregen getal op de plek van de terugkeersprong
60          VGL.w   R0,1     Vergelijk parameter met 1
61          VSP     KLG,69   Als het 1 of 0 is, spring naar het basisgeval
62          BST     R0       Zo niet, berg hem op de stack
63          AFT.w   R0,1     Trek er één van af
64          HIA.w   R1,66    Haal terugkeeradres in register R1
65          STP              Laatste kopie, kan geen nieuwe kopie aanroepen.  Stop het programma.
66          HST     R0       Haal bovenste getal van stack
67          HST     R1       Haal volgende getal van stack
68          VER     R0,R1    Vermenigvuldig met elkaar in R0
69          BST     R0       Berg het resultaat op in de stack
70          STP              Dummy instructie, wordt overschreven door terugkeersprong
-------------------------------------------------------------------------------
71  sprc:   3221900000       Waarde van de SPR-instructie
            EINDPR
```

Tabel 10: Een programma dat de faculteit van een invoergetal berekent, met recursie door middel van duplicatie.

0		LEZ		
1		HIA.w	R2,3	Terugkeeradres in register R2
2		SPR	fib1	Spring naar eerste versie fibonacci subroutine
3		HST	R0	
4		DRU		Druk het resultaat af
5		STP		
6	fib1:	OPT	R2,sprc	
7		BIG	R2,22	Overschrijf terugkeeradres
8		VGL.w	R0,1	
9		VSP	KLG,21	$n \leq 1$, spring naar basisgeval
10		AFT.w	R0,1	Verminder argument
11		BIG	R0,fib1v	Sla waarde argument op fib1 variabele
12		HIA.w	R2,14	Terugkeeradres in register R2
13		SPR	fib2	Spring naar volgende kopie
14		HIA.w	R0,fib1v	Haal waarde argument uit fib1 variabele
15		AFT.w	R0,1	Verminder argument
16		HIA.w	R2,18	Terugkeeradres in register R2
17		SPR	fib2	Spring naar volgende kopie
18		HST	R1	
19		HST	R0	
20		OPT	R0,R1	Tel uitkomsten recursieve aanroepen bij elkaar op
21		BST	R0	
22		STP		Dummy-instructie, wordt overschreven door terugkeersprong
23	fib2:	OPT	R2,sprc	
				⋮
39		STP		
40	fib3:	OPT	R2,sprc	
				⋮
56		STP		
57	fib4:	OPT	R2,sprc	
				⋮
73		STP		
74	fib5:	OPT	R2,sprc	
				⋮
90		STP		
91	fib1v:	RESGR1		reserveer ruimte voor variabele in fib1
92	fib2v:	RESGR1		...
93	fib3v:	RESGR1		...
94	fib4v:	RESGR1		...
95	fib5v:	RESGR1		...
96	sprc:	3221900000		Waarde van de SPR-instructie
		EINDPR		

Tabel 11: DRAMA-programma dat het n-de getal in de fibonacci reeks uitrekent, met behulp van recursie door duplicatie. Vraagt n als invoer.

```
 0                LEZ             Lees een getal (a)
 1                BST     R0      Berg dit getal op de stack (push)
 2                LEZ             Lees een getal (b)
 3                BST     R0      Berg dit getal op de stack (push)
 4                LEZ             Lees een getal (c)
 5                BST     R0      Berg dit getal op de stack (push)
 6                HST     R0      Haal het bovenste getal van de stack (pop)
 7                HST     R1      Haal het volgende getal van de stack (pop)
 8                OPT     R0,R1   Tel deze getallen bij elkaar op in R0
 9                BST     R0      Berg dit getal op de stack (push)
10                HST     R0      Haal het bovenste getal van de stack (pop)
11                HIA.w   R1,15   Haal het terugkeeradres in register R1
12                BST     R1      Berg het terugkeeradres op de stack (push)
13                BST     R0      Berg parameter voor functieaanroep op stack
14                SPR     kwadraat Spring naar de kwadraat subroutine
15                HST     R0      Haal het bovenste getal van de stack
16                HST     R1      Haal het volgende getal van de stack
17                VER     R0,R1   Vermenigvuldig deze getallen in R0
18                BST     R0      Berg dit getal op de stack
19                HST     R0      Haal het bovenste getal van de stack
20                DRU             Druk dit getal af
21                STP             Stop het programma
22  kwadraat:     HST     R0      Haal het bovenste getal van de stack
23                VER     R0,R0   Vermenigvuldig dit getal met zichzelf (kwadraat)
24                HST     R1      Haal het terugkeeradres van de stack
25                BST     R0      Berg uitkomst van vermenigvuldiging op stack
26                SPR     0(R1)   Spring naar waarde in R1 (terugkeeradres)
```

Tabel 12: DRAMA programma in de stijl van Dijkstra voor berekening van de functie $a \times (b + c)^2$.

```
 0        HIA     R8,R9       Stel PP(main) in
 1        SBR     a           Spring naar subroutine a
 2        STP                 Beëindig het programma
 3   a:   BST     R8          Berg PP(main) op de stack (push)
 4        HIA     R8,R9       Stel PP(a) in
 5        HIA.w   R0,5        Haal de waarde 5 in accumulator
 6        BST     R0          En berg als lokale variabele op de stack
 7        HIA.w   R0,6        Haal de waarde 6 in accumulator
 8        BST     R0          En berg als lokale variabele op de stack
 9        SBR     b           Spring naar subroutine b
10        HIA     R9,R8       Stel PP(a) in als nieuwe stack pointer
11        HST     R8          Gooi block a van de stack (pop)
12        KTG                 Keer terug naar regel 2
13   b:   BST     R8          Berg PP(a) op de stack (pop)
14        HIA     R8,R9       Stel PP(b) in
15        HIA.w   R0 ,3       Haal de waarde 3 in accumulator
16        BST     R0          En berg als lokale variabele op de stack
17        HIA     R1,0(R8)    Haal PP(a) uit de link
18        HIA     R0,-1(R1)   En haal de eerste lokale variabele op
19        HST     R1          Haal lokale variabele 3 van de stack en zet in R1
20        VER     R0,R1       Vermenigvuldig 5 met 3
21        DRU                 Druk de waarde af
22        HIA     R9,R8       Stel PP(b) in als nieuwe stack pointer
23        HST     R8          En gooi block b van de stack (pop)
24        KTG                 Keer terug naar regel 10
```

Tabel 13: DRAMA-programma in de stijl van Dijkstra dat gebruik maakt van *random access*.

```
 0          HIA     R8,R9       Stel PP(main) in
 1          LEZ                 Lees een getal (n)
 2          SBR     fac         Spring naar subroutine fac
 3          HIA     R0,R1       Haal het tussenresultaat naar accumulator R0
 4          DRU                 En druk dit af
 5          STP                 Beëindig het programma
 6    fac:  BST     R8          Berg PP op de stack (push)
 7          HIA     R8,R9       Stel nieuwe PP in
 8          BST     R0          Berg n op de stack (push)
 9          VGL.w   R0,1        n ≤ 1?
10          VSP     KLG,base    Spring dan naar base
11          AFT.w   R0,1        Anders, verlaag n met 1
12          SBR     fac         En spring weer naar subroutine fac
13          HST     R0          Haal n van de stack
14          VER     R1,R0       Tussenresultaat * n
15          HST     R8          Stel nieuwe PP in
16          KTG                 Keer terug naar regel 3
17   base:  HST     R1          Zet 1 in het register voor het tussenresultaat
18          HST     R8          Stel nieuwe PP in
19          KTG                 Keer terug naar regel 13
```

Tabel 14: DRAMA-programma voor de faculteitsfunctie in de stijl van Dijkstra.

```
0         LEZ
1         HIA.w   R1,5      Terugkeeradres in register R1
2         BST     R1        Terugkeeradres op stack
3         BST     R0        Invoer op stack
4         SPR     fib       Spring naar fibonacci subroutine
5         HST     R0
6         DRU               Druk resultaat af
7         STP
8   fib:  HST     R0        Argument in register R0
9         VGL.w   R0,1
10        VSP     KLG,base  Als argument≤ 1 spring naar basisgeval
11        AFT.w   R0,1      Verminder argument
12        HIA.w   R1,27
13        BST     R1        Tweede terugkeeradres op stack
14        BST     R0        Argument op stack
15        AFT.w   R0,1      Verminder argument
16        HIA.w   R1,20
17        BST     R1        Eerste terugkeeradres op stack
18        BST     R0        Argument op stack
19        SPR     fib       Eerste recursieve aanroep
20        HST     R0        Uitkomst eerste recursie aanroep
21        HST     R1        Eerstvolgende argument
22        HST     R2        Eerstvolgende terugkeeradres
23        BST     R0
24        BST     R2
25        BST     R1        Sla op in volgorde:  uitkomst > terugkeeradres > argument
26        SPR     fib       Tweede recursieve aanroep
27        HST     R0
28        HST     R1
29        OPT     R0,R1     Tel uitkomsten recursieve aanroepen bij elkaar op
30        HST     R1        Terugkeeradres uit stack
31        BST     R0        Uitkomst op stack
32        SPR     0(R1)     Spring naar terugkeeradres
33  base: HST     R5        Terugkeeradres
34        BST     R0        Waarde argument (0 of 1) op stack
35        SPR     0(R5)     Spring naar terugkeeradres
          EINDPR
```

Tabel 15: DRAMA-programma dat n-de getal uit fibonacci-reeks berekent in de stijl van Dijkstra. Vraagt n als invoer.

C [R] JAVA-vertaler

Dit programma, geschreven in de JAVA-programmeertaal, geeft een vertaling van een aritmetische expressie naar een lijst met instructies op machinecode-niveau op de manier van Bauer en Samelson in 'Sequential Formula Translation' [16].

```
1  import java.util.*;
2
3  public class testmetstrings {
4
5      static Stack<String> operatorsCellar = new Stack<String>();
6      static String[] variabelen = new String[]{"XXX", "n1", "n2", "n3", "n4",
           "n5", "n6", "n7", "n8", "n9", "n10"};
7      static int varPointer = 0;
8
9      public static void main(String[] args) {
10       // De expressie die moet worden vertaald:
11       String expression = "a*(b+c)";
12
13       LinkedList<String> uitvoer = new LinkedList<String>();
14       computeExpression(expression, uitvoer);
15       System.out.println("Instructies:␣" + uitvoer.toString());
16     }
17
18     public static void computeExpression(String expression, LinkedList<
           String> machinecode){
19       System.out.println("Expressie:␣" + expression);
20       // Voeg "E" Toe op eind, deze zorgt voor stop.
21       expression = expression.concat("E");
22
23       String X = null;
24       int startIndex = operatorsCellar.size();
25
26       /* In deze loop worden de karakters in de expressie ombeurten
             doorgelopen.
27        * Wanneer het een spatie is, doe niks; wanneer het de eerste is, voeg
              toe aan cellar.
28        * Wanneer het een letter is, voeg toe aan output.
29        * Wanneer het een operator is wordt er op basis van de vier cases als
              beschreven in het artikel
30        * van Bauer en Samelson een keus gemaakt voor de vervolgstap.
31        */
32       for(int i = 0; i < expression.length(); i++){
33         X = Character.toString(expression.charAt(i));
34
35         if(!X.matches("␣")){
36
37           if(X.matches("[a-z]")){
38             varPointer ++;
39             machinecode.add(X + "␣=>␣" + variabelen[varPointer]);
```

```
40          } else if(X.matches("E")){
41            while (operatorsCellar.size() > startIndex) {
42              machinecode.add(variabelen[varPointer - 1] + "␣=>␣" +
                    variabelen[varPointer-1] + operatorsCellar.pop() +
                    variabelen[varPointer]);
43              varPointer -= 1;
44            }
45          } else if(startIndex == operatorsCellar.size()){
46            operatorsCellar.push(X);
47          } else {
48            String last = operatorsCellar.peek();
49            if (X == "(" && last == "(") {
50              operatorsCellar.push(X);
51            } else if(firstcase(last,X)){
52              machinecode.add(variabelen[varPointer - 1] + "␣=>␣" +
                    variabelen[varPointer - 1] + operatorsCellar.pop() +
                    variabelen[varPointer]);
53              varPointer -= 1;
54              operatorsCellar.push(X);
55            } else if(secondcase(last,X)){
56              operatorsCellar.push(X);
57            } else if(thirdcase(last,X)) {
58              operatorsCellar.pop();
59            } else if(fourthcase(last,X)) {
60              machinecode.add(variabelen[varPointer - 1] + "␣=>␣" +
                    variabelen[varPointer - 1] + operatorsCellar.pop() +
                    variabelen[varPointer]);
61              varPointer -= 1;
62              i--;
63              continue;
64            }
65          }
66        }
67      }
68    }
69
70    public static boolean firstcase(String fromStack, String current) {
71      boolean first = fromStack.matches("[\\+\\-]") && current.matches("
            [\\+\\-]");
72      boolean second = fromStack.matches("[\\*\\/]") && current.matches("
            [\\*\\/]");
73
74      return first || second;
75    }
76
77    public static boolean secondcase(String fromStack, String current) {
78      boolean first = fromStack.matches("\\(") && current.matches("
            [\\+\\-\\*\\/]");
79      boolean second = fromStack.matches("[\\+\\-]") && current.matches("
            [\\*\\/]");
80      boolean third = fromStack.matches("[\\+\\-\\*\\/]") && current.matches
            ("\\(");
81
82      return first || second || third;
```

```
83    }
84
85    public static boolean thirdcase(String fromStack, String current) {
86      boolean first = fromStack.matches("\\(") & current.matches("\\)");
87
88      return first;
89    }
90
91    public static boolean fourthcase(String fromStack, String current) {
92      boolean first = fromStack.matches("[\\*\\/]") && current.matches("
            [\\+\\-]");
93      boolean second = fromStack.matches("[\\+\\-\\*\\/]") && current.
            matches("[\\)e]");
94
95      return first || second;
96    }
97
98  }
```

Referenties

[1] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A., Rutishauser, H., Samelson, K., Vauquois, B., et al. Report on the algorithmic language ALGOL 60. *Numerische Mathematik 2*, 1 (1960), 106–136.

[2] Bottenbruch, H. Structure and use of ALGOL 60. *J. ACM 9*, 2 (Apr. 1962), 161–221.

[3] Brooks, R. R. *Introduction to Computer and Network Security: Navigating Shades of Gray*. CRC Press, 2013.

[4] Davis, M., Sigal, R., and Weyuker, E. J. *Computability, complexity, and languages: fundamentals of theoretical computer science*. Newnes, 1994.

[5] Daylight, E. G. Dijkstra's rallying cry for generalization: The advent of the recursive procedure, late 1950s–early 1960s. *The Computer Journal 54*, 11 (2011), 1756–1772.

[6] Daylight, E. G. *The Dawn of Software Engineering: From Turing to Dijkstra*. Lonely Scholar, Heverlee, 2012.

[7] Decker, B. D., and Verbaeten, P. *Structuur en organisatie van computersystemen*, 3 ed., vol. 1. Uitgeverij Acco, 2013.

[8] Dijkstra, E. W. Recursive programming. *Numerische Mathematik 2*, 1 (Dec. 1960), 312–318.

[9] Dijkstra, E. W. Algol-60 translation. *ALGOL Bulletin, supplement nr. 10* (1961).

[10] Irons, E. T., and Feurzeig, W. Comments on the implementation of recursive procedures and blocks in ALGOL 60. *Communications of the ACM 4*, 1 (1961), 65–69.

[11] Kleene, S. C. *Introduction to Metamathematics*. Van Nostrand, New York, 1952.

[12] Rice, H. G. Letters to the editor. *Communications of the ACM 3*, 9 (1960), L.12–13.

[13] Rice, H. G. Recursion and iteration. *Commun. ACM 8*, 2 (Feb. 1965), 114–115.

[14] Rinderknecht, C. A survey on teaching and learning recursive programming. *Informatics in Education 13*, 1 (2014), 87–119.

[15] Samelson, K., and Bauer, F. L. Sequentielle formelübersetzung. *Elektronische Rechenanlagen 1*, 4 (1959), 176–182.

[16] Samelson, K., and Bauer, F. L. Sequential formula translation. *Communications of the ACM 3*, 2 (Feb. 1960), 76–83.

[17] van den Hove, G. On the origin of recursive procedures. *The Computer Journal* (2014).

[18] Wilkes, M. V., and Renwick, W. The EDSAC (Electronic Delay Storage Automatic Calculator). *Mathematics of Computation 4*, 30 (1950), 61–65.

[19] Wilkes, M. V., Wheeler, D. J., and Gill, S. *The Preparation of Programs for an Electronic Digital Computer (Charles Babbage Institute Reprint)*. The MIT Press (Tomash), 1951 (1984).