Hoofdstuk 8

Programmeertalen

In hoofdstuk 7 heb je kennis gemaakt met het begrip algoritme en voorbeelden daarvan bij routinehandelingen. We bespraken de basisstructuren waaruit elk algoritme is opgebouwd: het na elkaar uitvoeren van opdrachten, de voorwaardelijke keuze en de herhalingsstructuur. In dit hoofdstuk bouwen we daarop voort. We bekijken nu vooral hoe algoritmen moeten worden beschreven als we ze door computers willen laten uitvoeren. Aan de orde komt dus hoe algoritmen kunnen worden omgezet in computerprogramma's.

8.1 Programmeertalen

Elke digitale computer is opgebouwd uit miljoenen schakelaars. Deze kunnen in twee standen staan: aan of uit. Daarmee stellen we een bit voor, het digitale cijfer 0 of 1. Een computer bestaat dus uit een hoop bits en kan uitsluitend met nullen en enen werken.
In hoofdstuk 4 en 6 hebben we de basisinstructies gezien die de centrale processor van de computer kan verwerken. Deze instructies zijn zeer kleine, eenvoudige bewerkingen. Ieder programma, in welke programmeertaal dan ook, wordt uiteindelijk vertaald in deze basisinstructies. Voorbeelden van deze instructies staan in tabel 1.

Invoeren	Voer een cijfer in
Onthouden	Onthoud een cijfer
Optellen	Tel twee cijfers op
Uitvoeren	Voer een cijfer uit
Vergelijken	Bepaal of een cijfer groter is dan een ander
Kiezen	Kies een serie instructies
Herhalen	Herhaal een serie instructies

Tabel 1 Basisinstructies voor elke computer

Computers werken alleen met nullen en enen. Ze moeten dus ook hun instructies in het geheugen opslaan in de vorm van lange reeksen bits.
De taal waarin elke computer de primitieve instructies in het geheugen opslaat, heet *machinetaal*. In de eerste computers, ruim vijftig jaar geleden, voerden de programmeurs alle *machine-instructies* met de hand in door lange rijen schakelaars in te stellen. Vervolgens drukten ze op een knop om de instructie in de eerstvolgende variabele op te slaan. Daardoor werd een deel van het geheugen stap voor stap gevuld met instructies, een voorbeeld staat in tabel 2.

geheugenplaats	**machine-instructie**
8000	01011011011101110101101010111010
8004	00011110000010010010101010101110
8008	11101111101110101110111010111010

Tabel 2 Machinetaal

Het invoeren van die nullen en enen is een heidens karwei. Omdat het zo lastig is, zijn er andere manieren gezocht om programma's te beschrijven en in te voeren. De programmeertalen die na de machinetaal zijn bedacht en gerealiseerd, hebben allemaal een eigenschap gemeen: het zijn pogingen om computerinstructies die uiteindelijk toch in machinetaal in het computergeheugen moeten staan, voor ons mensen beter leesbaar en begrijpelijk te maken. Het liefst zouden we de machine-instructies in onze eigen taal willen schrijven of inspreken en ze automatisch laten vertalen in de machinetaal.

De eerste verbetering was de lange rijen bits te verdelen in groepjes van vier bits. Daarmee kunnen we in het hexadecimale talstelsel werken. We moeten de binaire machine-instructies in groepjes van vier bits dan vertalen in zestien 'cijfers'. Dat zijn de cijfers 0 tot en met 9 en de letters A tot en met F. Een voorbeeld staat in tabel 3.

reeks bits:	01101000111100000100111010011100101000011									
reeks bytes:	01101000		11110000		01001110		10011100		10100011	
reeks 4-bits:	0110	1000	1111	0000	0100	1110	1001	1100	1010	0011
hexadecimaal:	5	8	F	0	4	E	9	C	A	3

Tabel 3 Machinetaal in vier formaten

Ondanks gebruik van het hexadecimale talstelsel bleef het coderen van de instructies ingewikkeld en lastig, maar het typwerk was minder inspannend en dit beperkte het aantal invoerfouten.
De volgende stap was de *assemblertaal*, waarin elke machine-instructie door een afkorting, in het Engels: *mnemonic*, wordt beschreven. Ook kon je namen voor getallen en adressen in het geheugen gebruiken. Een *assemblervertaalprogramma* zet elke assembler-instructie om in de bijhorende rij hexadecimale of binaire getallen. In tabel 4 zie je een voorbeeld van assembler-instructies met bijbehorende hexadecimale machine-instructies.

Assembler-instructie	**Machine-instructie**
LOAD REGISTER1 POORT3	FE AE 03
MOVE REGISTER 1 MEM2525	38 AE 2525
MOVE MEM2600 REGISTER 2	39 BA 2600
ADD REGISTER 1 MEM2602	72 AA 2602
COMPARE REGISTER 2 MEM2602	87 AB 2602
JUMP MEM8000	07 8000
REPEAT 10, MEM8100	A1 0A 8100

Tabel 4 Voorbeelden van assembler-instructies en bijbehorende hexadecimale machine-instructies

Achter elk codewoord van de assemblertaal gaat een vaste reeks enen en nullen schuil. Elke naam van een adres in het geheugen, met getallen of instructies, wordt ook vertaald in een reeks bits (hexadecimale cijfers).
De volgende stap was dat er *hogere programmeertalen* werden ontwikkeld, waarbij de opdrachten in onze eigen taal konden worden ingevoerd. Deze worden 'hogere' programmeertalen genoemd. De manier waarop zulke programma's verwerkt worden is in principe hetzelfde als bij assemblertaal.
Een *vertaalprogramma* (Engels: *compiler*) moet elke opdracht omzetten (vertalen) in een vaste reeks nullen en enen. De taal bestaat uit codewoorden die een opdracht voor de computer beschrijven. Bijvoorbeeld PRINT, wat wil zeggen dat de computer iets in de monitor moet laten zien. Of de opdracht: SOM:=EERSTE+TWEEDE om twee getallen in het geheugen van de computer op te tellen en het resultaat op een derde plek in het geheugen te onthouden. Elke instructie in een hogere programmeertaal wordt in feite omgezet in een serie assembler-instructies (met bijbehorende machine-instructies).

Hogere taal	**Assemblertaal**	**Machinetaal**
SOM:=EERSTE+TWEEDE	MOVE REGISTER1 MEM2500	38 AE 2500
	ADD REGISTER1 MEM2501	72 AA 2501
	MOVE MEM2502 REGISTER1	39 AA 2502

Tabel 5 Voorbeelden van programma-instructies en vertaling naar machinetaal.

Net als bij de machinetaal en de assemblertaal wordt een programma van boven naar beneden afgewerkt. Elke instructie wordt van links naar rechts gelezen.
Natuurlijk moet een programmeertaal exact volgens de specifieke regels van die taal gebruikt worden. We spreken daarom wel van de *codeerregels* van een programmeertaal. Niet alleen hebben codewoorden een vaste betekenis, ook is het gebruik van leestekens geregeld. De betekenis van de codewoorden ligt vast in de *semantiek* van de taal, dat is de leer van de betekenis der woorden. De grammatica ligt vast in de *syntaxis* van de taal, dat is de leer van het gebruik van zinsdelen, de volgorde van de woorden en de zinsopbouw, inclusief het gebruik van leestekens.
De syntaxregels van een programmeertaal kunnen worden beschreven in syntaxdiagrammen en in BNF-notatie. In hoofdstuk 2 zijn daar voorbeelden van te vinden.
Als we een programmeertaal vergelijken met een menselijke taal zoals Engels, Frans of Nederlands vallen er twee verschillen op. Een programmeertaal heeft maar enkele tientallen vaste woorden, de *codewoorden*. Je kunt ze makkelijk in een lijst zetten. In tabel 6 staan de codewoorden van de taal Pascal.

and	div	file	in	of	record	type
array	do	for	label	or	repeat	until
begin	downto	function	mod	packed	set	var
case	else	goto	nil	procedure	then	while
const	end	if	not	program	to	with

Tabel 6 Lijst met alle 35 codewoorden van de programmeertaal Pascal.

Elk codewoord heeft een nauwkeurig bepaalde vaste betekenis, die elke computer die de programmeertaal 'kent', op precies dezelfde manier 'verstaat'. Kortom, elk codewoord leidt tot exact dezelfde actie in elke computer.
Bij de taal die mensen gebruiken om met elkaar te communiceren, is dat anders. Het woordenboek der Nederlandse taal bevat wel 300.000 woorden, waaronder synoniemen en homoniemen. Het bijzondere is dat dezelfde woorden bij verschillende mensen tot verschillende interpretaties (gedachten en acties) kunnen leiden.
Een voorbeeld van een eenvoudige programmeertaal is *Basic*. Basic is ongeveer in 1970 bedacht. Bij Basic worden de programmaregels genummerd. Dat is handig om binnen een programma naar voren of terug te kunnen springen om keuze en herhaling te programmeren. Deze sprongopdracht is GOTO. In een Basicprogramma betekent GOTO 10, dat er na uitvoering van de voorgaande regels wordt gesprongen naar de instructie in programmaregel nummer 10. Het programma gaat dan verder met de uitvoering van regel 10 en de daaropvolgende regels. Een voorbeeld staat in programma 1, REM staat voor commentaar.

```
10 REM Dit is een Basicprogramma, dat nooit stopt
20 PRINT "Hallo allemaal"
30 GOTO 10
```

Programma 1 Basicprogramma met een oneindige herhaling door een sprongopdracht

Een nadeel en gevaar van het gebruik van regelnummering en de GOTO-instructie is dat grote programma's ondoorzichtig worden door de vele regelnummers en de sprongen erin. Het groot programma is niet meer te overzien. Als we de programma-instructies proberen te volgen, lijkt het wel of we een berg spaghetti proberen te ontrafelen. Het programma wordt feitelijk onleesbaar voor mensen. We noemen zulke programma's ongestructureerd.
In latere versies van Basic en andere programmeertalen zijn verbeteringen aangebracht die het gebruik van regelnummers en spronginstructies beperken of zelfs onmogelijk maken.
De basis voor goed gestructureerde (begrijpelijke) programma's is, dat elk algoritme zoveel mogelijk wordt verdeeld in *subroutines* of *procedures*. Een programma is dan goed

gestructureerd en overzichtelijk. Daardoor is de werking van het algoritme gemakkelijk leesbaar en goed te begrijpen.
Goede voorbeelden van talen die met procedures werken, zijn *Pascal* en *C++*. Deze programmeertalen werken niet met regelnummering. Dat hoeft ook niet, omdat elk deelprogramma of procedure een naam krijgt. Moet een procedure worden uitgevoerd, dan wordt dat deelprogramma opgeroepen via zijn naam.

Recent ontwikkelde programmeertalen zijn grafisch georiënteerd. Dat wil zeggen dat procedures worden voorgesteld door plaatjes. Het programmeren lijkt daarmee nog een stap eenvoudiger geworden. Bij Basic, Pascal en C++ kun je makkelijk zondigen tegen de syntaxis-regels. Bij grafisch georiënteerde talen is de kans daarop minder groot, ook omdat je dan werkt met programmabibliotheken. Bekende voorbeelden van talen waarin je visueel kunt programmeren zijn, Visual Basic, Delphi en Java. Op deze grafisch georienteerde talen komen we terug in hoofdstuk 3

8.2 Programma's verwerken

Programmeertalen zijn in de loop der tijd steeds makkelijker te gebruiken geworden. Daardoor wordt de afstand van die taal tot de machinetaal steeds groter. Nieuwe programmeertalen hebben steeds weer nieuwe mogelijkheden. Vertaalprogramma's worden daardoor steeds groter.
Ook de vertaalde programma's zelf worden steeds groter. Voor één enkele opdracht in een programmeertaal, bijvoorbeeld voor het tekenen van een lijn, zijn soms wel honderden machinetaal-instructies nodig. Dat zien we ook aan de toename in omvang van de programma's in de laatste jaren. Als een vertaald programma in 1970 ongeveer 1000 machine-instructies lang was, zou een grafisch programma dat hetzelfde doet nu na vertaling misschien wel 100.000 machine-instructies opleveren.

Hoewel de programmeertalen eenvoudiger te hanteren zijn, is het heel goed mogelijk dat mensen (programmeurs) bij het programmeren fouten maken. Juist door het gebruik van steeds grotere programmabibliotheken met allerlei bouwstenen neemt de kans op fouten toe. De programmeur kent de betekenis of het voorgeschreven gebruik van de bibliotheek niet precies. Er zijn drie soorten fouten: syntactische, semantische en logische fouten.
Syntactische fouten zondigen tegen de grammatica van de taal. In Basic bijvoorbeeld kun je een aantal opdrachten op één regel zetten, maar je moet ze wel van elkaar scheiden met dubbele punten. Gebruik je een puntkomma dan is dat een syntactische fout.
Ook typfouten horen bij deze soort. De opdracht PRINT betekent dat er iets moet worden afgedrukt. Schrijf je PRNT, dan geeft het vertaalprogramma een foutmelding, omdat het niet in de lijst met codewoorden voorkomt.
Syntaxfouten herkent het vertaalprogramma altijd. Bij de vertaling en uitvoering van het programma krijg je dan een melding dat er in een bepaalde regel een syntaxfout zit.
Semantische fouten zondigen tegen de betekenis van codewoorden. Ook het gebruik van bouwstenen of procedures waarvan we betekenis, bedoeling of gebruik niet precies kennen, leidt tot semantische fouten. Wanneer je in Java '=' (wordt gelijk aan) schrijft, terwij je '==' (is gelijk aan) bedoelt, dan maak je een semantische fout.
Semantische fouten worden soms wel en soms niet door het vertaalprogramma herkend.
Logische fouten zijn de lastigste. Bij logische fouten werkt het programma vaak wel, maar het doet niet wat het zou moeten doen. Om ze te voorkomen moet het algoritme goed in elkaar zitten. Bij een programma voor het omzetten van graden Celsius in graden Fahrenheit moet je de juiste formule gebruiken. Doe je dat niet, dan geeft het programma foute antwoorden.
Semantische fouten en logische fouten zijn soms zichtbaar, omdat je aan de werking en uitvoer van een programma kunt zien dat de resultaten fout zijn. Vaak zijn deze fouten niet direct duidelijk en dat is gevaarlijk, omdat de gebruiker van het programma veronderstelt dat het programma correct werkt.
Aan het voorkomen van semantische en logische fouten moet je daarom tijdens het programmeren veel aandacht besteden. Hier geldt: voorkomen is beter dan genezen, want

vaak zijn de symptomen van de ziekte - de verborgen fouten - nauwelijks zichtbaar en doen ze zich pas voor na jarenlang gebruik van het programma. Het stellen van de diagnose - het opsporen van de fouten - en het vinden van de juiste medicijnen - het herstel van de fouten - is dan een zeer lastig of misschien zelfs onmogelijk. De patiënt overlijdt; het programma moet worden vervangen door een nieuw programma.

Om fouten te voorkomen, moet je programma's een goede structuur geven. Die structuren zijn al herkenbaar in de algoritmen die we in hoofdstuk 1 hebben bekeken. We bekijken deze algoritme nogmaals, maar dan in samenhang met een computerprogramma dat ze kan kan uitvoeren. We laten verschillende programmeertalen de revu passeren. De algoritmen voor de tekenopdrachten zullen we omzetten naar programma's in JavaLogo (een uitbreiding van de taal Java). De sorteeralgoritmen vertalen we in Pascal. Daarnaast bekijken we nog wat programma's in een eenvoudig te begrijpen programmeertaal die vroeger veel gebruikt werd: Basic.

We beginnen met een aantal kleine programma's in Basic en Pascal

8.3 Voorbeelden in Basic en Pascal

Programma 2 is een Basicprogramma dat een getal met drie vermenigvuldigt.

Programmastappen	**Basicprogramma**
Geef een getal	10 INPUT A
Bereken drie maal dat getal	20 LET B = 3*A
Druk de uitkomst af	30 PRINT B

Programma 2 Voorbeeld van een Basicprogramma.

Je ziet dat de programmaregels genummerd zijn. Meestal in tientallen, omdat dat bij het programmeren handig is. Moet je later een programmeerregel invoegen, dan kan dat zonder de regels andere nummers te geven. In het voorbeeldprogramma komen drie instructies voor. Natuurlijk moet je als programmeur weten wat die betekenen. Anders kun je niet precies begrijpen wat er binnen de computer gebeurt.

```
INPUT A
```

De *invoeropdracht* geeft aan dat er in het interne geheugen van de computer een variabele A wordt gereserveerd en dat daarin het getal wordt opgeslagen, dat de gebruiker van het programma invoert. Om aan te geven dat de gebruiker iets moet typen, verschijnt er op het scherm een vraagteken.

```
LET B = 3*A
```

De *rekenopdracht* bestaat uit een aantal verschillende acties. Ten eerste wordt er een variabele B gereserveerd. Ten tweede wordt de inhoud van variabele A opgehaald en vermenigvuldigd met 3. Tenslotte wordt de uitkomst van de berekening in B gestopt. Het codewoord LET mag in de nieuwe versies van Basic worden weggelaten.

```
PRINT B
```

De *uitvoeropdracht* geeft aan dat de inhoud van variabele B op het scherm moet komen.

Het voorgaande programma bevat uitsluitend de opdrachten die voor de computer nodig zijn. Zonder toelichting is het programma voor mensen niet eenvoudig te gebruiken, te lezen en te begrijpen.
Er komt zonder toelichting een vraagteken op het scherm. Als de gebruiker van het programma de programma-instructies niet kent, roept dat vraagteken alleen maar vragen op. De gebruiker heeft immers geen enkel idee wat er wordt gevraagd.
Ook het prog ramma zelf geeft weinig informatie. Je kunt programma's voor mensen beter leesbaar en bruikbaar maken door commentaar- of documentatie-regels toe te voegen. Ook

duidelijke namen voor variabelen dragen bij aan de leesbaarheid van een programma. Hieronder staat een beter leesbare en bruikbare versie van het programma ‘Drievoud’.

```
10 REM Dit programma berekent het drievoud van een getal.
20 PRINT “Ik bereken voor u het drievoud van een getal.”
30 PRINT “Typ achter het vraagteken een getal.”
40 INPUT Getal
50 LET Drievoud = 3 * GETAL
60 PRINT “Het drievoud van het getal is:”
70 PRINT Drievoud
80 END
```

Programma 3 Verbeterde versie van Basicprogramma 'Drievoud'.

REM Opmerking

Rem is de afkorting van het Engelse woord remark. Alles wat op een regel achter de *commentaaropdracht* REM zet, wordt niet uitgevoerd.

PRINT "tekst"

De *uitvoeropdracht* zet de tekst die hier tussen aanhalingstekens staat, letterlijk op het scherm. In het programma is deze opdracht driemaal gebruikt. In figuur 10 staat de interactie met de gebruiker.

END

Dit lijkt een overbodige opdracht. Als regel 70 is uitgevoerd, stopt het programma immers vanzelf. Toch is het nodig een Basicprogramma met de *END-opdracht* af te sluiten, zeker bij grote programma’s met subroutines. Bovendien wordt het programma zo duidelijker.

```
Ik bereken voor u het drievoud van een getal.
Typ achter het vraagteken een getal.
? 25
Het drievoud van het ingevoerde getal is:
75
```

De instructies van de programmeertaal Pascal zijn uitgebreider dan bij Basic. We gaan hetzelfde programma nog eens bekijken, maar nu in Pascal.

Programmastappen	**Pascalprogramma**
	program drievoud (input, output);
	{Dit programma berekent het drievoud van een getal.}
	var getal: real;
	drievoud : real;
	begin
Zet tekst op het scherm	writeln(“Ik bereken voor u het drievoud van een getal.”);
	write(“Typ een getal: ”);
Geef een getal	read (getal);
Bereken drie maal dat getal	drievoud := 3 * getal;
	write(“Het drievoud van het getal is: ”);
Druk de uitkomst af	writeln(drievoud);
Einde programma	end.

Programma 4 Voorbeeld van Pascalprogramma 'Drievoud'

We lichten dit programma stap voor stap toe.

```
program naam
```

Elk Pascalprogramma heeft een naam, waarmee het programma begint.

```
(input, output);
```

Achter de naam staat tussen haakjes of het programma behoefte heeft aan invoer (input) door de gebruiker en of het iets uitvoert naar het beeldscherm of de printer (output). Elke Pascalopdracht wordt afgesloten met een puntkomma, waarmee de opdrachten van elkaar gescheiden worden.

```
{toelichting}
```

Commentaar zet je tussen accolades, vergelijk REM bij Basic. De regel wordt niet afgesloten met een puntkomma, omdat het geen opdracht is.

```
var getal : real;
```

Aan het begin van het programma worden de variabelen gedeclareerd. Var is een afkorting van variabele. De gebruikte variabelen heten getal en drievoud. Real betekent dat het om een kommagetal gaat.

```
begin
```

De uitvoering van het programma wordt gestart.

```
write("uitleg");
```

De tekst *uitleg* komt in het scherm. De volgende write-opdracht plaatst zijn uitvoer direct achter *uitleg*.

```
writeln("uitleg");
```

Ook nu komt de tekst *uitleg* in het scherm. Een hierop volgende write-opdracht plaatst zijn uitvoer nu echter op een nieuwe regel, omdat de letters 'ln' (=line) achter 'write' zijn toegevoegd.

```
read(getal)
```

De gebruiker voert iets in wat vervolgens wordt opgeslagen in de variabele getal. Omdat deze variabele bestemd is voor kommagetallen, moet de invoer ook daaruit bestaan.

```
drievoud:= 3*getal;
```

De variabele drievoud wordt gebruikt om driemaal de inhoud van variabele getal in op te slaan. Let op de dubbele punt := betekent in Pascal 'wordt'.

```
write (drievoud);
```

De inhoud van variabele drievoud wordt opgehaald en op het scherm gezet. De volgende write-opdracht zet zijn uitvoer direct achter dit getal.

```
End.
```

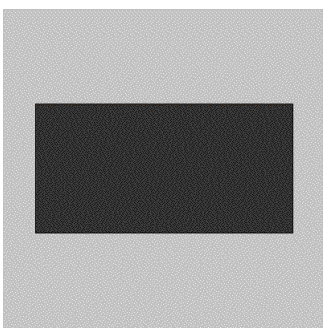
Het programma wordt beëindigd. Achter de laatste “End” moet is Pascal een punt.

Pascalopdrachten lijken op die van Basic, maar dat ze verschillen toch op onderdelen van elkaar. Een ander verschil is dat Pascal altijd vereist dat de gebruikte variabelen van tevoren zijn vastgelegd. Wat beide talen gemeen hebben, is dat je de taalregels exact moet volgen. Een kleine typfout heeft tot gevolg dat het programma niet doet wat je wilt. Je moet dus wel erg precies zijn, wil je ingewikkelde programma’s kunnen maken. Pascal heeft dan het voordeel dat je alle deelalgoritmen (procedures) die je nodig hebt, apart kunt ontwerpen en testen.
Door het programma of onderdelen ervan te *testen*, kom je er meestal wel achter wat je fout hebt gedaan bij het programmeren. Heel anders is het als je een denkfout hebt gemaakt of een verkeerde formule hebt gebruikt. Die logische fouten kun je niet zo eenvoudig achteraf constateren. Het is dan ook noozakelijk een probleem goed te analyseren, voordat je gaat programmeren.

Tenslotte kijken we naar een programma in JavaLogo, voor het uitvoeren van een tekenalgoritme.

```
hoogte = 200
breedte = 400

vulAan("zwart")
vooruit(hoogte)
rechts(90)
vooruit(breedte)
rechts(90)
vooruit(hoogte)
rechts(90)
vooruit(breedte)
rechts(90)
vulUit()
```

Algoritme rechthoek



Figuur 1 Rechthoek

```
import logotekenap.*;

public class Rechthoek extends TekenApplet
{
    double hoogte;
    double breedte;

    public void initialiseer()
    {   achtergrondkleur("lichtgrijs");
    }

    public void tekenprogramma()
    {   hoogte = 200;
        breedte = 400;
        stap(-breedte/2, -hoogte/2);
        vulAan("zwart");
        vooruit(hoogte);rechts(90);
        vooruit(breedte);rechts(90);
        vooruit(hoogte);rechts(90);
        vooruit(breedte);rechts(90);
        vulUit();
    }
}
```

Programma rechthoek

Je ziet dat de tekenopdrachten uit het algoritme herkenbaar zijn in het programmadeel tussen de accolades na “public void tekenprogramma()”. De afzonderlijke opdrachten worden, net als in Pascal afgesloten met een punt-komma. Een andere overeenkomst met Pascal is dat de variabele moeten worden gedeclareerd. Dat gebeurt in de regels:

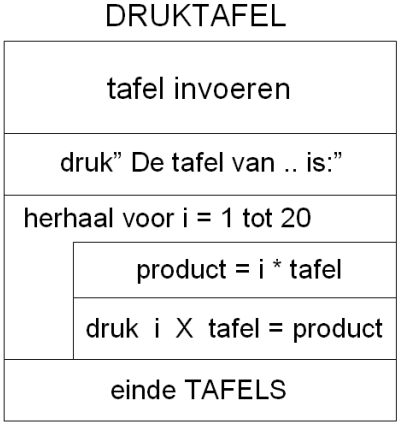
```
double hoogte;
double breedte;
```

Het woord ‘double’ betekent dat het hier gaat om komma-getallen, vergelijkbaar met ‘real’ in Pascal. Op de andere details van het programma gaan we niet in. In de digitale practica over JavaLogo wordt hier meer aandacht aan besteed.

8.4 Herhalingsstructuur in programeertalen

In algoritmen komen herhalingsopdrachten voor. Denk maar aan de voorbeelden uit het vorige hoofdstuk, zoals de algoritmen voor het maken van tekeningen. In deze paragraaf zien we hoe deze herhalingsopdrachten in verschillende programmeertalen worden gebruikt.

We kijken eerst naar een eenvoudig voorbeeld, het programma DrukTafel, dat tafels van vermenigvuldiging afdrukt. We bekijken twee versies: een versie in Basic (figuur 2) en een versie in Pascal (figuur 3). Elk programma drukt de tafel van een in te voeren getal. De tafel loopt door tot twintig.



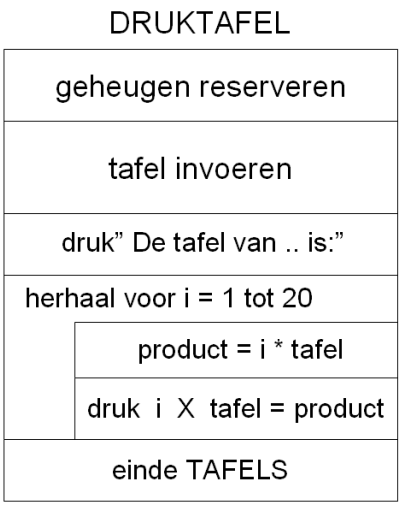

```
10 REM DRUKTAFEL
20 PRINT ”Geef getal tussen 1 en 10: ”
30 INPUT Tafel
40 PRINT ”De tafel van ”;tafel; ”is:”
50 FOR i = 1 TO 20
60    product = i * tafel
70    PRINT i,” X ”,tafel,” = ”,product
80 NEXT i
90 END
```

Figuur 2 Basicprogramma DRUKTAFEL

Toelichting bij de nieuwe Basicopdrachten:

```
PRINT i, tafel, ...
```

De komma geeft aan dat getallen die in de variabelen i, tafel en product zijn opgeslagen, in tabelvorm worden getoond. Na elke printopdracht wordt op een nieuwe regel begonnen.




```
program DrukTafel (input,output);
var i, product, tafel : integer;
begin
  writeln('Geef getal tussen 1 en 10');
  read(tafel);
  writeln('De tafel van ', tafel, ' is.');
  for i:=1 to 20 do
     begin
        product := i * tafel;
        writeln(i,' X ',tafel,' = ',product);
     end;
end.
```

Figuur 3 Pascalprogramma DrukTafel

Toelichting bij de nieuwe Pascalopdrachten:

for i:=1 to 20 do *opdracht*;

Dit is de bepaalde herhaling in Pascal. Je laat de teller i op 1 beginnen en met stapjes van 1 tellen tot en met twintig, andere getallen kan natuurlijk ook. Je hoeft niet zoiets als NEXT op te schrijven, Pascal doet dit zelf.

begin .... end;

In dit programma zitten er twee opdrachten binnen de herhaling. In Pascal moet je er dan 'begin ... end;' omheen te zetten. De twee opdrachten worden dan samen een 'blok' dat herhaald wordt. Binnen een blok moet je weer puntkomma's zetten tussen de opdrachten. Voor de leesbaarheid zet je de woorden begin en end op aparte regels en laat je ze iets inspringen. Met de opdrachten ertussen doe je hetzelfde. Zo kun je altijd snel zien bij welke 'begin' een 'end' hoort.

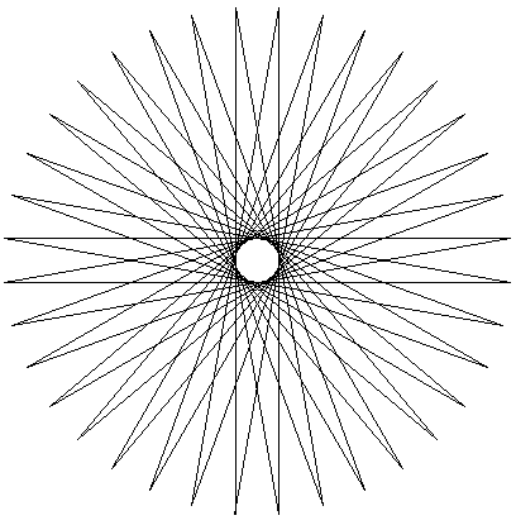
writeln(i, tafel, ..);

De komma geeft aan dat getallen die in de variabelen i, tafel en product zijn opgeslagen, in tabelvorm worden getoond

Tenslotte nog een voorbeeld van een JavaLogo-programma met herhalingsopdrachten. Je ziet dat syntax enigszins lijkt op die van Pascal, maar toch weer anders is. Bijvoorbeeld, de omlijsting van de opdrachten met 'begin' en 'end' zoals in Pascal gebeurt in Java niet. In Java worden daarvoor accolades gebruikt.

```
Herhaal 36 keer
{  vooruit(400)
   rechts(170)
}
```

Algoritme ster



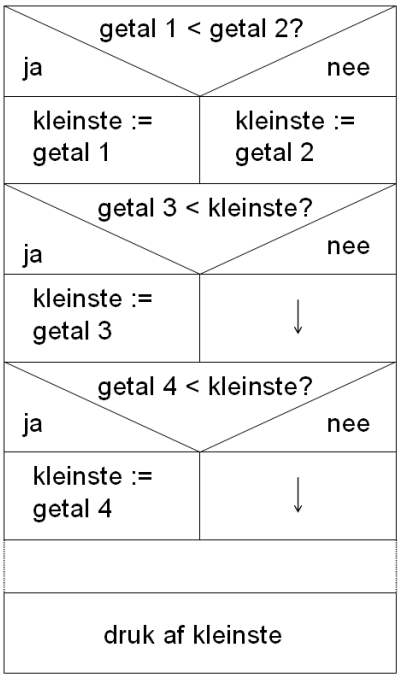
Figuur 4 ster

```
import logotekenap.*;

public class Ster extends TekenApplet
{
    public void tekenprogramma()
    {
        for(int i=0 ; i<36 ; i++)
        {   vooruit(400);
            rechts(170);
        }
    }
}
```

Programma ster

8.5 Keuzeopdrachten vertalen

In hoofdstuk 7 zagen we een algoritme voor het bepalen van het kleinste getal van een rij getallen. De structuurdiagram van dit algoritme zie je nogmaal hieronder




```
als getal 1 < getal 2

   dan onthoud getal 1 in kleinste

   anders onthoud getal 2 in kleinste

als getal 3 < kleinste

   dan onthoud getal 3 in kleinste


als getal 4 < kleinste

   dan onthoud getal 4 in kleinste

. . .
. . .

druk af kleinste
```

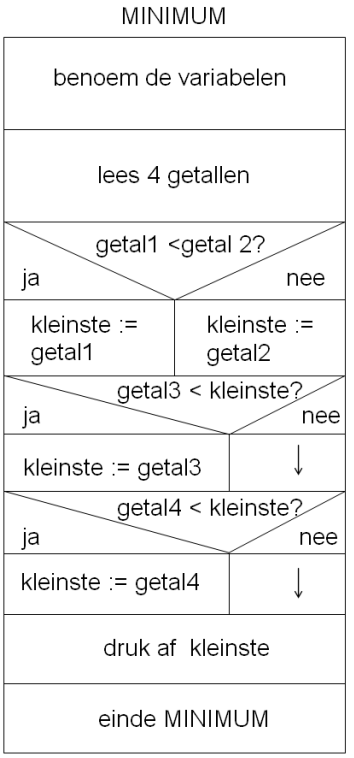
Figuur 5 Keuzestructuur ter bepaling van het kleinste getal.

Het bijzondere van dit algoritme is dat er geen geneste keuzeblokken voorkomen. De keuzen worden na elkaar (sequentieel) afgewerkt. Een andere bijzonderheid is dat er gebruik wordt gemaakt van een extra variabele: kleinste. Daarin wordt het tot dan toe kleinste getal bewaard. We bekijken het het bijbehorende Pascalprogramma eens nader.

In de beginregels staat wat de naam van het programma is en welke variabelen er worden gebruikt. Bovendien ontbreken de regelnummers die zo kenmerkend voor Basic zijn. De nummers achter de regels horen niet bij het programma. Ze zijn toegevoegd om naar die regels te kunnen verwijzen.
Om de werking van het programma goed te begrijpen, is het handig om zelf na te gaan welke inhoud de variabelen krijgen tijdens de uitvoering van de programmaregels.
De keuzestructuur wordt in Pascal gecodeerd door de opdracht if-then-else. Het kan voorkomen dat het else-deel van de opdracht leeg is. Er zijn dan twee vormen mogelijk:

```
if  voorwaarde then opdracht
if  voorwaarde then opdracht1 else opdracht 2
```

In de eerste vorm geldt: Als aan voorwaarde is voldaan dan voer opdracht uit, anders doe niets.
In de tweede vorm geldt: Als aan de voorwaarde is voldaan dan voer opdracht1 uit, anders voer opdracht2 uit. Beide vormen komen voor in het programma Minimum.

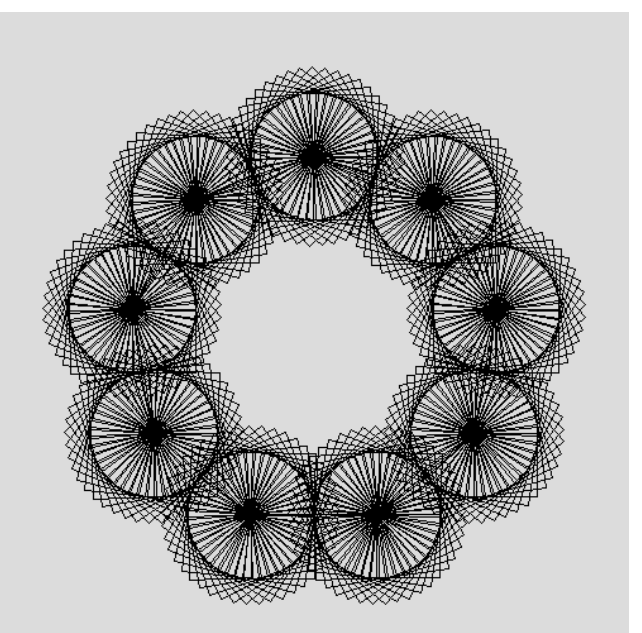



```
program MINIMUM(input,output); {1}
var getal1, getal2, getal3,    {2}
    getal4, kleinste: real;    {3}
begin                          {4}
          {inlezen van 4 getallen}
readln(getal1); readln(getal2);{5}
readln(getal3); readln(getal4);{6}
         {bepalen van het minimum}
if getal1 < getal2             {7}

   then kleinste:=getal1       {8}
   else kleinste:=getal2;      {9}

if getal3 < kleinste           {10}

   then kleinste:=getal3;      {11}

if getal4 < kleinste           {12}

   then kleinste:=getal4;      {13}
              {afdrukken minimum}
write(kleinste)                {14}

end.                           {15}
```

Figuur 6 Pascalprogramma Minimum.

8.6 Deeltaken: procedures en methoden

In hoofdstuk zag je dat het soms efficient is op in grotere algoritmen deeltaken te maken. Dat zagen we bij het tekenalgoritme van de krans.



Figuur 7 Krans

```
Herhaal 9 keer:
{   Maak rozet
    vooruit(100)
    links(40)
}

Maak rozet
Herhaal 45 keer
{   Maak vierkant
    rechts(8)
}

Maak vierkant
Herhaal 4 keer
{   vooruit(50)
    rechts(90)
}
```

Algoritme krans

Figuur 8 Structuurdiagram Krans met 9 rozetten

Verwerkt in een javaprogramma ziet het er zo uit:

```
public class Krans extends TekenApplet
{
    double zijde, afstand;

    public void tekenprogramma()
    {   zijde=50;afstand=100;
        penUit();stap(-50,-150);penAan();
        rechts(90);
        for(int i=0 ; i<9 ; i++)
        {   rozet(zijde);
            vooruit(afstand);
            links(40);
        }
    }

    void rozet(doubel z)
    {   for(int i=0 ; i<45 ; i++);
        {   vierkant(z);
            rechts(8);
        }
    }
```

```
   void vierkant(double z)
   {  for(int i=0 ; i<4 ; i++}
      {  vooruit(z);
         rechts(90);
      }
   }
}
```

Programma krans

Je hoeft niet alle details van het programma te begrijpen. In het digitale practicum gaan we hier dieper op in. Voor nu is het voldoende dat je de deeltaken ‘rozet’ en ‘vierkant’ herkend in de code. Zo’n codeblok voor een deeltaak noemen we in Java een ‘methode’. Verder kun je zien dat er meer getallen als variabele worden gebruikt. Dat maakt het makkelijker om de maten van de tekening later nog aan te passen.

Elke programmeertaal heeft de mogelijkheid om op een vergelijkbare manier deeltaken te maken. Het zijn altijd stukken code met een naam, waarin een verzameling opdrachten staat. Met behulp van die naam kunnen die opdrachten vanuit een andere plaats in het programma worden uitgevoerd. In Pascal heten deze deeltaken ‘procedures’, in Basic worden het ‘subroutines’ genoemd.