Robo

Een Introductie tot Programmeren

Arvid Halma
Universiteit van Amsterdam

15 augustus 2007

"*Kijk! Ik heb een robot met een eigen willetje geprogrammeerd.
Hij doet helemaal niet wat ik wil!*"
– Dick Huyser

Inhoudsopgave

Voorwoord v

1 Introductie 1

1.1 Computers en Instructies 1

1.2 Basisinstructies 3

1.3 Aan de slag met RoboMind 4

1.3.1 De monitor 5

1.3.2 De afstandsbediening 5

1.3.3 Het scriptvenster 6

1.4 Programmeertalen 8

1.5 Spaties 9

1.6 Commentaar 10

2 Bewegen 13

2.1 Argumenten 13

2.2 Een route uitstippelen 14

2.3 Botsingen 15

3 Uitbreidingen van de taal 17

3.1 De herhaal-lus 17

3.1.1 Oneindige herhaling 21

3.2 Waarnemen en beslissen 22

3.2.1 Zien 22

3.2.2 Condities 24
3.2.3 Vind de witte stip 24
3.2.4 Logische expressie 27
3.3 De herhaalZolang-lus 30
3.4 Meer basisinstructies: verven en grijpen 31
3.4.1 Verven 31
3.4.2 Grijpen 33
3.4.3 Een puzzel 34
3.5 Zelf instructies definiëren 35
3.5.1 Naamgeving 36
3.5.2 Procedure definitie 37
3.5.3 Argumenten 38

4 Interessante programma's 43
4.1 Lijnvolger 43
4.2 Doolhofdoorkruiser 44

Appendix A: Basisinstructies 47

Appendix B: Programmeerstructuren 49

Voorwoord

In dit boek maak je op de eerste plaats kennis met programmeren, maar ook met andere belangrijke facetten die binnen informatica, kunstmatige intelligentie en wiskunde een rol spelen. Dit klinkt meteen wel als heel droge kost, die je maar beter aan de ergste nerds kunt voorschotelen. Toch gaan we ons alleen met de leuke dingen bezig houden en is het ook nog eens niet heel ingewikkeld. Zo zal je bijvoorbeeld te weten komen hoe je altijd met een paar simpele regels uit elk mogelijk doolhof komt en hoe je door domweg stippen te tekenen en te bekijken elke gewenste berekening kunt maken.

Om ervoor te zorgen dat je snel aan de slag kan met de onderwerpen waar het om draait, is er een nieuwe programmeertaal geïntroduceerd genaamd Robo. Het is een kleine taal met een overzichtelijk aantal regels, toegespitst op het programmeren van een robot. Toch zijn mogelijkheden te over om interessante programma's te maken. Ook vormen de principes die je tegenkomt de kern voor de meeste andere programmeertalen.

Zoals Brian Kernighan en Dennis Richie, de makers van de populaire programmeertaal 'C', al aangaven, kan je een programmeertaal alleen leren door er in te programmeren. Voor Robo is dit niet anders. Ook al zijn de meeste opdrachten niet lastig, het is aan te raden vanaf het begin je ideeën echt uit te proberen.

Voor wie?

De tekst is geschreven voor mensen die geen enkele ervaring hoeven hebben met programmeren. Dit betekent dat we in het begin wat tijd moeten spenderen aan de introductie van sommige begrippen. Voor degenen die al enige programmeer ervaring hebben opgedaan komen er andere interessante zaken aan bod. Zij zullen wat sneller kunnen beginnen met het maken van ingewikkelder programma's om puzzels op te lossen.

Opbouw

Dit boek begint met een iets algemenere introductie om duidelijk te maken wat "het programmeren" nu eigenlijk inhoud. Vervolgens maken we kennis met de mogelijkheden van de robot, en de taal waarin we opdrachten kunnen geven. Onderwerpen die niet direct noodzakelijk zijn voor het programmeren van de robot, maar een iets algemener fenomeen beschrijven, staan in losse stukken genaamd Verdieping. Deze mag je dus overslaan.

In ontwikkeling

Zowel de software als deze handleiding blijven in ontwikkeling. Dat betekent dat de inhoud nog niet uitgekristalliseerd is en dat er onderwerpen kunnen ontbreken. Tips en commentaar zijn daarom van harte welkom. Stuur die naar *arvid@robomind.net*

Ik wil bij deze Albert Stoop nogmaals hartelijk danken voor de aangedragen verbeteringen van deze introductie.

– *Arvid Halma, mei 2005* (herziene versie: augustus 2007)

Hoofdstuk 1

Introductie

Nadat we op orde hebben wat er precies bedoeld wordt met instructies, maak je kennis met RoboMind, het enige gereedschap dat je nodig hebt om je eigen robot te programmeren.

1.1 Computers en Instructies

Instructies

Wanneer je een computer iets wilt laten doen, zal je instructies moeten geven voor wàt hij exact moet doen. Deze opdrachten kunnen erg voordehandliggend zijn. Zo kun je bijvoorbeeld aangeven dat je klaar bent voor het versturen van e-mail door op 'verzenden' te drukken. Of je kunt de opdracht geven om een tekstbestand te openen door op het bijbehorende icoontje te klikken. In beide gevallen geef je de instructie door met de muis ergens op te klikken.

Sommige andere instructies moeten worden uitgeschreven. Denk bijvoorbeeld aan het intikken van een adres in een webbrowser. De computer leest de door jouw geschreven regel, bijvoorbeeld *www.robomind.net*, en kan dan de juiste webpagina voor je laden.

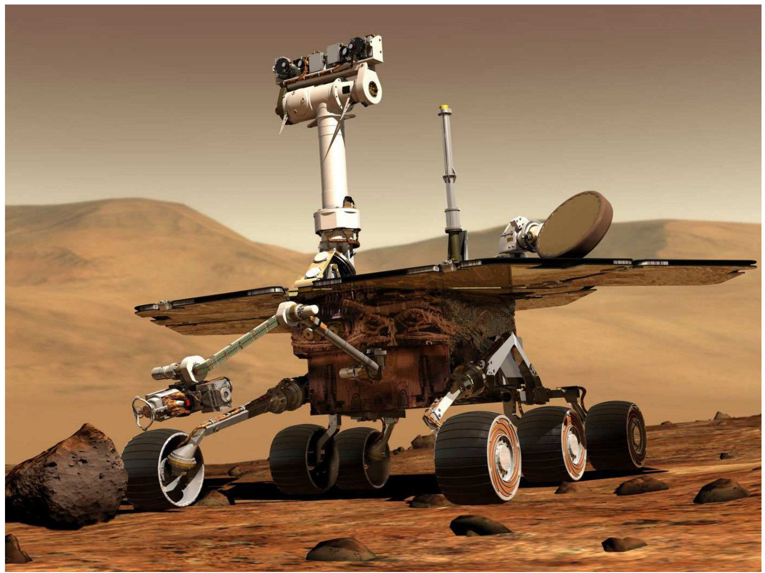
Er zijn nog tal van andere mogelijkheden om je bedoelingen kenbaar te maken, zoals met behulp van een joystick of gesproken opdrachten. Het belangrijkst is dat je inziet dat het in principe gaat om het geven van opdrachten, ofwel instructies.

Wanneer je een reeks instructies meer dan eens wilt uitvoeren, of je kunt de opdrachten niet zelf geven, wordt het tijd om het geven van instructies te automatiseren. Van de verschillende mogelijkheden die er zijn om instructies te geven, blijkt het vaak het handigst om alle opdrachten uit te schrijven. Deze geschreven instructies noem je een programma.

Programma

Het automatiseren van het geven van zulke instructies mag je programmeren noemen. In een programma noteer je precies *wat* de computer *wanneer* moet uitvoeren in een reeks instructies. Deze instructies die uitgevoerd moeten worden schrijf je uit in een tekst die de computer kan lezen.

De ruimtevaartorganisatie NASA heeft een aantal jaren terug een robot naar Mars gestuurd om daar het terrein te verkennen. Door de grote afstand waarop de robot moest opereren was het niet praktisch deze volledig door een mens aan te laten sturen via een afstandsbediening. Ook worden er robots gemaakt die geheel autonoom, dat wil zeggen op eigen houtje, een route kunnen uitstippelen tussen obstakels door. Doordat je soms van te voren niet weet hoe het terrein er uitziet, moet je de instructies in het programma zó kiezen dat de robot er het beste van maakt op het moment dat hij er is.



Figuur 1.1: Mars Explorer

Het soort problemen dat de programmeurs van NASA tegenkwamen, zullen wij ook behandelen. We zullen programma's schrijven voor een gesimuleerde robot die in een mogelijk onbekende omgeving handelingen moet uitvoeren.

1.2 Basisinstructies

Het noteren van instructies in een tekst gaat volgens stricte regels. Zo moet je bijvoorbeeld weten welke instructies je kunt gebruiken bij het omschrijven van de opdrachten. Deze instructies verschillen van situatie tot situatie. Bij het programmeren van een videorecorder zou je instructies willen geven als: *"neem het programma op van de VPRO dat om acht uur begint"*. Bij het programmeren van een robot zouden deze instructies echter nergens op slaan. Een robot zou je liever iets willen vertellen als: *"Als je ziet dat er links niets in de weg staat, beweeg dan naar links"*. Hier zouden *"kijk naar links"* en *"beweeg naar links"* basisinstructies kunnen zijn.

Basisinstructies

Per domein zijn er een vast aantal basisinstructies die je mag gebruiken bij het schrijven van een programma. Je kunt alleen deze instructies *direct* gebruiken bij het schrijven van een programma.

Ook al zou het niet een heel rare opdracht zijn voor een robot om een huis te bouwen, het is onwaarschijnlijk dat *"bouw een huis"* een basisinstructie is. Als je de robot dit toch wil laten doen zul je de opdracht *"bouw een huis"* moeten uitdrukken in termen van basisinstructies. Die zouden bijvoorbeeld kunnen zijn: *"vind een open plek"*, *"bekijk de bouwtekening"*, *"verzamel materiaal"*, enzovoort.

Op dit moment is het nog onduidelijk waar de grenzen liggen. Waarom zou *"verzamel materiaal"* nou wèl een basisinstructie zijn? Je kan deze namelijk ook weer opdelen in deelopdrachten als *"ga naar de winkel"* en *"koop hout"*. Om te bepalen welke instructies we 'simpel genoeg' noemen, kunnen we niet gewoon even logisch nadenken, omdat je niet weet waar je zou moeten stoppen. We spreken daarom gewoon af wat de basisinstructies zijn voor een bepaald probleem.

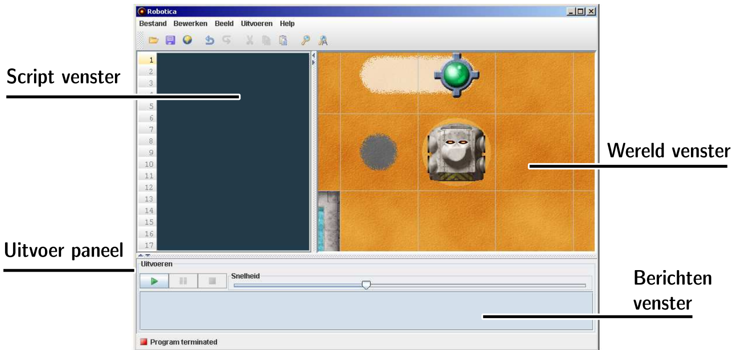
AFSPRAKEN

Zo is er in RoboMind ook een aantal basisinstructies afgesproken. Een daarvan is bijvoorbeeld *"rechts()"*[1]. Als we zo meteen in een programma *"rechts()"* opnemen, zal de robot daadwerkelijk naar rechts draaien. Omdat wij ons aan de afspraak houden, hoeven we niet verder uit te leggen hoe hij dit dan zou moeten doen.

[1] De haakjes achter 'rechts' horen er echt bij. Later wordt duidelijk waarom dit nuttig is.

1.3 Aan de slag met RoboMind

Laten we maar eens kijken wat de basisinstructies voor onze robot zijn. Om dat te doen moet je RoboMind opstarten. Je krijgt het volgende scherm te zien dat uit een aantal onderdelen bestaat.



Figuur 1.2: Overzicht van RoboMind

Scriptvenster	In dit tekstveld kun je de opdrachten omschrijven die de robot moet uitvoeren.
Monitor	Hier zie je hoe de robot er aan toe is in de omgeving. Omdat dit de gehele omgeving van de robot betreft, noemen we dit voor het gemak maar 'de wereld'.
Uitvoerpaneel	Wanneer je hebt besloten wat de robot moet doen, kan je hier de robot starten en stoppen.
Berichtenvenster	Mochten de opdrachten niet helemaal correct zijn genoteerd, dan word je hiervan op de hoogte gesteld in het Berichtenvenster. Ook meldt de robot hier zo nu en dan gebeurtenissen.

Tabel 1.1: De verschillende onderdelen van RoboMind.

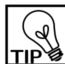
1.3.1 De monitor

Momenteel is de monitor het belangrijkst. Je ziet hierin de robot en de omgeving waarin hij verkeert. Je kunt de omgeving bekijken door op dit venster te klikken, de muis ingedrukt te houden en deze te bewegen. Als je de muis loslaat, spring je weer terug zodat de robot weer in het midden van het beeld staat. Onder het menu Beeld staan drie belangrijke opties:

Zoom in	Vergroot het beeld, zodat je alles van dichterbij bekijkt.
Zoom uit	Verklein het beeld, zodat je alles verderaf ziet.
Volg robot	Als deze optie aan staat, blijft de robot in het midden van het beeld. Zo niet, dan kan je de omgeving bekijken door in het venster te slepen met de muis.

Tabel 1.2: De verschillende onderdelen van RoboMind.

Je kunt ook in- en uitzoomen door aan het scrollwieltje van de muis te draaien als je boven de monitor staat.



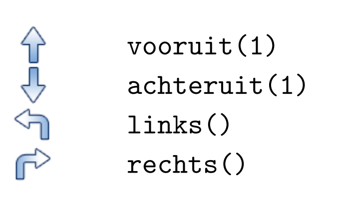
1.3.2 De afstandsbediening

Een van de manieren om in RoboMind instructies uit te voeren, is met behulp van de afstandsbediening.

Open de afstandsbediening: Uitvoeren ▷ Afstandsbediening. Je kunt de robot bewegen door op de pijltjesknoppen te klikken. Probeer dit uit. Iedere keer dat je op vooruit of achteruit klikt, zal de robot één vakje verplaatsen. Als je op links of rechts draait, draait de robot 90 graden. Je merkt dat het even duurt voordat de robot klaar is met bewegen. Wanneer je snel verschillende opdrachten geeft, worden deze allemaal onthouden en achtereenvolgens uitgevoerd.

Telkens wanneer je een opdracht uitvoert zie je onderin een woord verschijnen. Dit is de overeenkomende tekstuele instructie met de knop waarop je klikte. Zo krijg je wanneer je op vooruit drukt, onderin `vooruit(1)` te zien. Probeer na te gaan dat de knoppen de volgende instructies aan de lijst van instructies in de afstandsbediening toevoegen:

Figuur 1.3: De afstandsbediening



⇧	`vooruit(1)`
⇩	`achteruit(1)`
↰	`links()`
↱	`rechts()`

Tabel 1.3: Basisinstructies om te bewegen

Zo meteen schrijven we ons eerste programma en zullen daar de bovenstaande geschreven instructies voor gebruiken.

Je kan de knop ◎ gebruiken om de robot weer naar de startpositie terug te brengen. De overige knoppen op de afstandsbediening zijn nu niet erg van belang. Het kan overigens geen kwaad om er alvast een beetje mee te spelen, ook al komen ze pas later aan bod.

1.3.3 Het scriptvenster

We hebben tot dusver de muis gebruikt om de robot opdrachten te geven door te klikken. De andere manier is door instructies te schijven in het scriptvenster.

Ga naar het scriptvenster en tik vervolgens dit onderstaande programma over:

```
1 vooruit(1)
2 rechts()
3 vooruit(1)
4 rechts()
5 vooruit(1)
6 rechts()
7 vooruit(1)
8 rechts()
```

Dit is je eerste programma! Klik nu op Uitvoeren ▷ Uitvoeren in het menu. De robot zal de door jouw geschreven instructie lezen en achtereenvolgens uitvoeren. Als het goed is, rijdt de robot nu een klein vierkantje rechtsom. Als de laatste instructie is uitgevoerd, stopt de robot.

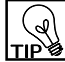
Als er een bericht in het berichtenvenster verschijnt, heb je ergens een foutje gemaakt. Doordat je niet precies een bekende opdracht schreef, weet de robot niet wat je bedoelt. Dit kan heel iets onbenulligs zijn als het vergeten van een letter of een haakje. In het berichtenvenster verschijnt een foutmelding waarin de robot probeert te raden wat je fout hebt gedaan. Jammer genoeg is hij hier vrij slecht in. Ook al klopt er echt iets niet in je programma, de suggestie is soms wat cryptisch.

FOUTMELDING

Een andere belangrijke opmerking over de precieze notatie van instructies betreft hoofdlettergevoeligheid . Dit betekent dat twee woorden zelfs als verschillend worden beschouwd, ook al verschillen ze alleen in hoofdlettergebruik. Het is duidelijk dat “fiets” en “robot” verschillende woorden zijn; de letters zijn simpelweg verschillend. Maar met hoofdlettergevoeligheid zijn bijvoorbeeld “robot”, “Robot”, en “roBOt” ook allemaal anders. In het geval van onze instructies, als je “Vooruit(1)” schrijft in plaats van “vooruit(1)” zal de instructie niet worden begrepen omdat je de opdracht met een hoofdletter begon. Het is gebruikelijk dat programma’s hoofdlettergevoelig moeten worden geschreven.

HOOFDLETTERGEVOELIGHEID

Je kunt dit programma opslaan, Bestand ▷ Opslaan, bijvoorbeeld onder de naam “eersteprogramma”. Er wordt automatisch de “.irobo” extensie aan toegevoegd zodat je later nog weet dat het om een robo programma ging. Je kunt deze later weer openen om het nogmaals uit te voeren of om het uit te breiden met extra instructies, Bestand ▷ Openen.



1.4 Programmeertalen

Op dit moment hebben we vier basisinstructies tot onze beschikking, namelijk `vooruit(1)`, `achteruit(1)`, `links()` en `rechts()`. Deze instructies kunnen we in willekeurige volgorde achter elkaar zetten en gebruiken in een tekst. We hebben dan een programma. Er komt echter nog meer kijken bij het programmeren dan af te spreken welke instructies je mag gebruiken. Tot nu toe werden de instructies gewoon van boven naar beneden uitgevoerd.

Programmeertaal

Een programmeertaal bestaat uit een aantal afgesproken basisinstructies *en* regels over hoe deze instructies aan elkaar mogen worden gevoegd.

CONTROLESTRUCTUREN

In een programmeertaal beschik je vaak ook over manieren om de uitvoervolgorde (in het Engels: flow of control) aan te passen. Dit zijn controlestructuren in de taal die ervoor zorgen dat je efficiënter kan omschrijven wat er moet worden uitgevoerd. Tot nu toe was het sneller om de afstandsbediening te gebruiken dan om het programma over te typen. Met ingewikkelder opdrachten wordt het interessanter ze te noteren in een tekst.

Om het wat duidelijker te maken is hier een voorbeeld van de herhaal-controlestructuur. Deze structuur komt later nog uitgebreid aan bod en dient nu slechts als voorbeeld.

```
1 herhaal(4){
2     vooruit(1)
3     rechts()
4 }
```

Je kunt dit programma weer overschrijven in het scriptvenster en uitvoeren. Je zult zien dat de robot precies hetzelfde doet als in je eerste programma, terwijl je minder instructies hebt hoeven schrijven!

Wat gebeurt hier? Met `herhaal(4)` geef je aan dat de robot vier keer de instructies tussen accolades, {...}, moet uitvoeren. Let op `herhaal(4)` is *geen* basisinstructie van de robot, maar behoort tot de regels om instructies samen te voegen tot een programma.

Je ziet wel in dat deze extra regels je een hoop werk kunnen gaan besparen. De rest van dit boek is eigenlijk niets meer dan het introduceren van andere handige structuren in de Robo-programmeertaal.

Verdieping — WELGEVORMDHEID

De combinatie van basisinstructies en regels noem je de grammatica van een taal. In het Nederlands hebben we het ook over grammatica. We hebben het dan over de regels die de structuur van woorden en zinnen in een taal bepalen. Gelukkig zijn de regels bij een programmeertaal simpeler dan in een *natuurlijke* taal zoals het Nederlands of Engels.
Met een grammatica kun je programma's vormen. Als je je netjes aan de regels houdt die in de grammatica staan, begrijpt de robot wat hij moet doen. Programma's die op deze nette manier zijn gevormd, noem je *welgevormd*. Als je dit niet doet, is het ongedefinieerd wat er moet gebeuren en heb je met een incorrect programma te maken.

1.5 Spaties

Zo precies als je moet zijn bij het gebruik van hoofdletters, het gebruik van spaties, tabs en nieuwe regels maakt niet uit voor de uitvoer van het programma. In het Engels heten deze tekens: *white space*. De computer ziet geen enkel verschil tussen de volgende twee programma's:

```
1  achteruit(2)
2  rechts()
3  vooruit(1)
```

```
1          achteruit(2) rechts()
2
3  vooruit(1)
```

Alle opdrachten worden gewoon van links naar rechts en boven naar beneden gelezen. Op deze manier kun je het programma een beetje opmaken zodat het pret-

tiger leest. In het voorgaande voorbeeld zag het eerste programma er leesbaarder uit voor ons.

Zo was ook het programma dat de robot een vierkantje liet rijden opgemaakt voor de leesbaarheid. Ook al zijn de volgende programma's voor de computer hetzelfde:

```
herhaal(4) { vooruit(1) rechts() }
```

```
herhaal(4){
    vooruit(1)
    rechts()
}
```

Het is gebruikelijk om het programma op de tweede manier op te maken. Op deze manier is het wat duidelijker welke instructies herhaald worden. De tabs of spaties aan het begin van de regel om `vooruit(1)` en `rechts()` in te laten springen, wordt ook wel *indentatie* genoemd.

INDENTATIE

1.6 Commentaar

Het is vaak handig een programma te voorzien van commentaar. Daarmee wordt bedoeld dat je in het programma een stukje gewone tekst kunt schrijven dat niet bedoeld is om uit te voeren, maar dat alleen voor ons een handige notitie is. Om aan te geven dat we zo'n gewoon stukje tekst willen invoeren, schrijf je een hekje, `#`, gevolgd door jouw opmerking. De computer zal pas weer verder gaan met het lezen van de instructies op de volgende regel. Zo zou je je eerste programma van het volgende commentaar kunnen voorzien:

```
1  # Dit is mijn eerste programma.
2  # Maar nu voorzien van (niet al te zinnig) commentaar.
3
4  vooruit(1)
5  rechts()
6  vooruit(1)
7  rechts()
8
9  # We zitten nu op de helft
10 vooruit(1)
11 rechts()
12 vooruit(1)
13 rechts()    # draai de laatste keer naar rechts
```

Het wordt in het algemeen erg gewaardeerd als je een programma van zinnig commentaar voorziet, ook al doet het programma het ook zonder commentaar. Zo weet je later ook nog wat een programma doet.

Let even op. In het onderstaande programma zijn twee merkwaardige zaken aan de hand.

```
1  vooruit(1)
2  # rechts()
3  vooruit(1)
4  rechts()
5
6  # En dan nu
7    de laatste opdrachten
8  vooruit(1)
9  rechts()
10 vooruit(1)
11 rechts()
```

Op regel 2 staat `# rechts()`. De robot kent de opdracht `rechts()`, maar zal deze toch niet uitvoeren. Omdat de regel begint met # zal de instructie gezien worden als gewone tekst die dus niet uitgevoerd hoeft te worden. Het is niet een fout in het programma, zolang je maar doorhebt wat er wel en niet uitgevoerd zal worden.

Op regel 8 staat wèl echt een fout. Omdat deze regel niet begint met #, zal de computer denken dat er dus geldige opdrachten te lezen moeten zijn. Dus zal de computer denken dat je de instructies "de", "laatste" en "opdrachten" wilt uitvoeren. Dit klopt natuurlijk niet, en er zal een foutmelding verschijnen in het berichtenvenster. Ook deze regel moet dus beginnen met een hekje.

Hoofdstuk 2

Bewegen

Je hebt inmiddels al gezien dat de robot kan bewegen. In dit hoofdstuk kijken we iets preciezer naar deze mogelijkheden.

2.1 Argumenten

De instructie die je hebt gebruikt om vooruit te gaan was `vooruit(1)`. De robot bewoog dan precies een vakje in de richting waarin hij stond. Waarschijnlijk vermoed je al dat je ook `vooruit(2)` of `vooruit(15)` kan zeggen. Dit vermoeden is juist. De robot zal proberen precies zoveel stappen vooruit te gaan, als je in de instructie opgeeft. We zeggen dat je aan de instructie een *argument* kan meegeven. Dit argument moet een geheel getal zijn.

ARGUMENT

Het is natuurlijk een stuk korter om `vooruit(3)` op te schrijven, dan `vooruit(1) vooruit(1) vooruit(1)`. Beide manieren leveren uiteindelijk hetzelfde gedrag van de robot op.

Bij de instructie om achteruit te gaan moet je op dezelfde wijze een argument meegeven, bijvoorbeeld: `achteruit(3)`. Je zou overigens ook een negatief getal kunnen gebruiken: `vooruit(-3)`.

De instructies `rechts()` en `links()` nemen geen argument. Je kunt blijkbaar niet opgeven hoeveel de robot in een keer moet draaien. De robot zal altijd 90 graden in de juiste richting draaien. Als je de robot een heel rondje wilt laten draaien zal je vier keer `links()` of `rechts()` moeten opgeven.

Je kunt je afvragen waarom je dan toch nog haakjes achter bijvoorbeeld `links()` moet zetten. Bij `vooruit()` heeft dit zin, maar je mag helemaal niets opgeven bij

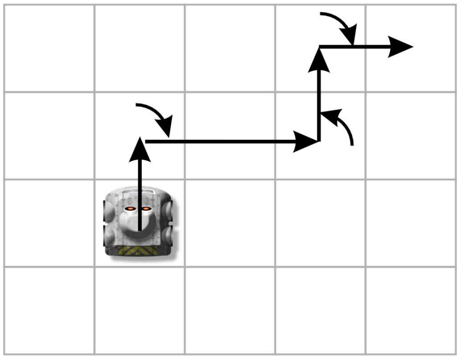
links(). Het ziet er inderdaad overbodig uit. Toch moet je ze schrijven om expliciet aan te geven. Het is ervoor om extra duidelijk aan te geven dat je instructies hebt mèt en zònder argumenten, maar dat het *beide* instructies zijn.

2.2 Een route uitstippelen

Om een beetje gevoel te krijgen voor de bewegingen van de robot is hier een voorbeeld. Stel dat we te maken hebben met het volgende programma:

```
1 vooruit(1)
2 rechts()
3 vooruit(2)
4 links()
5 vooruit(1)
6 rechts()
7 vooruit(1)
```

We kunnen nu, zonder dit programma uit te voeren, op papier bepalen wat er zal gaan gebeuren. Hieronder zie je de situatie. De pijlen geven aan waar de robot naartoe gaat. Als je alle stappen precies uitvoert, zie je dat de robot rechts boven zal eindigen. Probeer na te gaan dat dit het geval is.

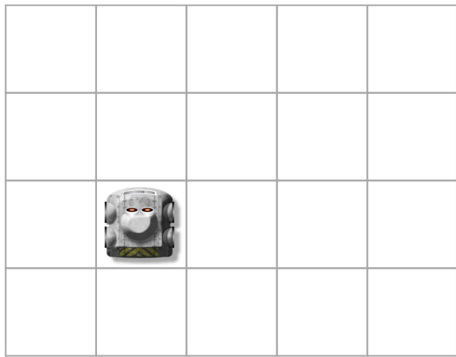


Figuur 2.1: De route van de robot

Je kunt het programma vervolgens controleren door het over te typen en te kijken of de robot doet wat je verwacht.

Probeer nu zelf de route uit te tekenen die de robot zal maken bij het volgende programma :

```
1 vooruit(2)
2 rechts()
3 vooruit(2)
4 rechts()
5 vooruit(3)
6 links()
7 vooruit(1)
```

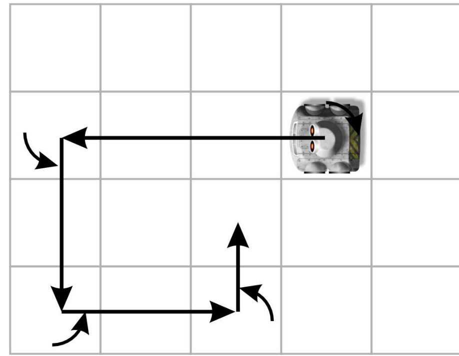


Figuur 2.2: Teken de route van de robot in

Andersom kunnen we kijken naar een route en daarbij alle opdrachten in een programma opschrijven. Probeer het programma te achterhalen bij de volgende route:

2.3 Botsingen

Bij het geven van de opdrachten moet je rekening houden met de omgeving. Als je de robot de opdracht geeft: `vooruit(100)` en er stond een muur dichtbij, dan zal het hem niet lukken 100 stappen vooruit te gaan. De robot botst dan tegen de muur, meldt dit in het berichtenvenster en gaat verder met de volgende opdracht. De route die je in gedachte had, hoeft nu niet meer te kloppen!



Figuur 2.3: De route van de robot

Verdieping — COMMUTATIVITEIT

De volgorde waarin je instructies laat uitvoeren maakt uit voor het resultaat. Als je bijvoorbeeld `vooruit(1) rechts()` uitvoert zal je op een andere positie eindigen dan wanneer je `rechts() vooruit(1)` uit zal voeren.

Als je even nadenkt, ben je dit al wel eens eerder tegengekomen bij berekeningen. Zo levert bij aftrekken $7 - 4 = 3$, maar $4 - 7 = -3$. De volgorde waarin je '7' en '4' zet, maakt uit voor het resultaat. Bij optellen is dit niet het geval. Zowel $5 + 3$ als $3 + 5$ leveren allebei '8' als resultaat. Ga zelf na of de volgorde bij vermenigvuldigen en delen uitmaakt.

Het ziet ernaar uit dat het uitvoeren van bewegingsinstructies in dit opzicht meer lijkt op aftrekken dan op optellen, omdat de volgorde wèl uitmaakt. Kunnen we dit iets algemener benoemen? Dat kan als we inzien dat bewegingsinstructies, maar ook rekenkundige operaties (zoals optellen en aftrekken) allebei een bepaald soort handelingen betreffen. Alle handelingen waarbij de volgorde *niet* uitmaakt, worden in de wiskunde *commutatief* genoemd. Als de volgorde van afhandeling *wel* uitmaakt, noem je het *niet-commutatief*.

Hoofdstuk 3

Uitbreidingen van de taal

In dit hoofdstuk breiden we onze uitdrukkingsmogelijkheden, oftewel *expressiviteit* uit. Dat betekent dat we sommige opdrachten op een makkelijkere manier kenbaar kunnen maken. Tot nu toe had je eigenlijk net zo goed de afstandsbediening kunnen gebruiken, als je de robot iets wilde laten doen. Nu er extra mogelijkheden worden toegevoegd aan de taal, zal blijken dat het maken van een programma vaak efficiënter is.

EXPRESSIVITEIT

3.1 De herhaal-lus

In de introductie werd al duidelijk dat er meer is dan het opsommen van instructies in een programma. Laten we nog eens kijken naar het programma om een vierkantje te tekenen.

```
1  herhaal(4){
2      vooruit(1)
3      rechts()
4  }
```

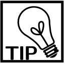
Hier werd gebruik gemaakt van de herhaal-lus. We noemen het een lus, omdat er een deel van het programma telkens opnieuw wordt uitgevoerd. Net zoals voor de opdrachten `vooruit(...)` en `achteruit(...)` gold, mag je bij `herhaal(...)` ook een willekeurig geheel getal als argument opgeven. De opdrachten tussen de accolades worden dan zo vaak herhaald als je hebt opgegeven. Het deel tussen tussen

accolades noem je een codeblok. In dit geval wordt dus `vooruit(1) rechts()`, `vooruit(1) rechts()`, `vooruit(1) rechts()`, `vooruit(1) rechts()` uitgevoerd. CODEBLOK

Een herhaal-lus kan bijna overal in een code voorkomen en zo vaak worden gebruikt als je wilt. In het volgende programma staan zowel basisinstructies als herhalingen:

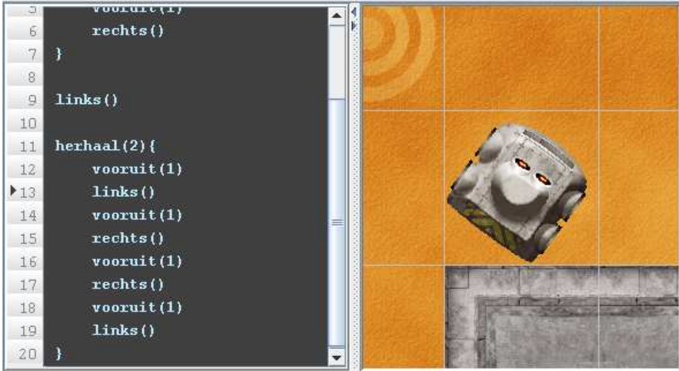
```
1  achteruit(2)
2  links()
3
4  herhaal(3){
5      vooruit(1)
6      rechts()
7  }
8
9  links()
10
11 herhaal(2){
12     vooruit(1)
13     links()
14     vooruit(1)
15     rechts()
16     vooruit(1)
17     rechts()
18     vooruit(1)
19     links()
20 }
```

Probeer dit programma uit en zorg ervoor dat je begrijpt hoe dit wordt uitgevoerd.

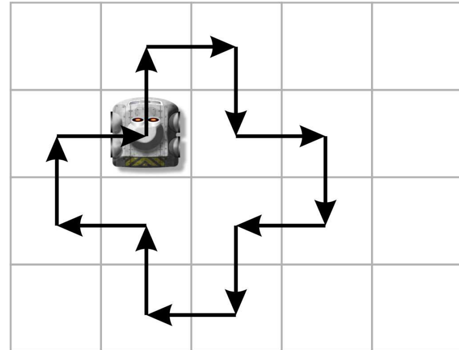


Aan de linkerkant van het scriptvenster staan regelnummers. Bij het uitvoeren van het programma zie je bij de regelnummers een pijltje dat aangeeft welke instructie op dat moment wordt uitgevoerd. Dit is handig om te zien of de robot doet wat je verwacht.

We kunnen het nog een beetje ingwikkelder maken. Stel dat je een herhaling wilt herhalen? Dit klinkt aanvankelijk wat vreemd, maar het kan in praktijk erg nuttig blijken. Hoe zou je bijvoorbeeld de volgende route willen programmeren?



Figuur 3.1: De huidige instructie zie je links bij de regelnummers.



Figuur 3.2: Een lastige route. Het draaien van de robot is hier niet meer aangegeven met pijltjes.

De route heeft een mooie symmetrische vorm. Meestal is dit een aanwijzing dat we hem vrij eenvoudig met herhaal-lussen kunnen programmeren. Als vuistregel kan je aannemen: hoe 'mooier' de vorm, des te simpeler is het programma. Onregelmatige routes, zoals het eerste voorbeeld in het hoofdstuk Bewegen, kunnen vaak niet zo makkelijk worden vereenvoudigd met herhaal-lussen. In dit geval gaat deze regel ook op.

Als we de hele route direct uitschrijven, wordt het een onoverzichtelijke bende:

```
# eerste stukje
vooruit(1)
rechts()
vooruit(1)
rechts()
vooruit(1)
links()

# tweede stukje
vooruit(1)
rechts()
vooruit(1)
rechts()
vooruit(1)
links()

# derde stukje
vooruit(1)
rechts()
vooruit(1)
rechts()
vooruit(1)
links()

# en het laatste stukje
vooruit(1)
rechts()
vooruit(1)
rechts()
vooruit(1)
links()
```

In elk stukje wordt er een deel van de route afgelegd. Zonder de regels commentaar, zou je al helemaal niet meer weten waar de robot was. In het programma valt het op dat alle stukjes er hetzelfde uitzien, dus kunnen we van de herhaal-lus gebruik maken:

```
1  herhaal(4){
2      vooruit(1)
3      rechts()
4      vooruit(1)
5      rechts()
6      vooruit(1)
7      links()
8  }
```

Zo, dat ruimt lekker op. Het programma is er een stuk korter op geworden, en doet nog steeds wat het moet doen. Als we nog iets preciezer kijken naar dit nieuwe programma, zien we dat er nog een herhaal-lus te gebruiken is. Er staat namelijk twee keer achter elkaar `vooruit(1) rechts()`. Dit kunnen we ook vervangen door een herhaling.

```
1  herhaal(4){
2      herhaal(2){
3          vooruit(1)
4          rechts()
5      }
6      vooruit(1)
7      links()
8  }
```

Dit levert niet zo'n grote winst op als de eerste keer. Het is ook bedoeld om te laten zien dat het kan. Hoe moeten we dit laatste programma nu eigenlijk lezen? De eerste herhaling geeft aan dat er vier keer een stuk herhaald moet worden. Dit stuk bevat zelf ook weer een herhaling en de instructies `vooruit(1) links()`. De instructie op regel 3 en 4 zullen dus in totaal acht keer worden uitgevoerd (4×2). Zorg ervoor dat je begrijpt dat alle zojuist besproken programma's hetzelfde doen.

3.1.1 Oneindige herhaling

Tot slot nog een speciaal gebruik van de herhaling. Als je geen argument meegeeft, zal de robot oneindig lang de opvolgende instructies blijven uitvoeren.

Nesting

Het laatste programma bevat een herhaalstructuur die zelf ook weer een herhaalstructuur bevat. Later zullen we andere structuren tegenkomen waarbij dit ook kan. In het algemeen noem je het geheel een *ingenestelde* structuur, wanneer je een *structuur in een structuur* tegenkomt. In het Engels noem je dit principe "nesting".

```
1 herhaal(){
2     vooruit(1)
3 }
```

Dit programma zal dus `vooruit(1)` alsmaar blijven herhalen. De robot zal dus vooruit rijden, net zolang tot hij ergens tegenop botst. Daarna zal hij blijven botsen. Je kan dit programma alleen stoppen door Uitvoeren ▷ Stoppen te gebruiken. De robot zal niet uit zichzelf ophouden. Het lijkt een beetje een vreemd gebruik van de herhaling; toch zal blijken dat deze herhaal-lus nuttig kan zijn.

Je kunt echt alleen de herhaal-lus ook zonder argument gebruiken. Bij de instructies `vooruit(...)` en `achteruit(...)` werkt dit niet.

3.2 Waarnemen en beslissen

Tot nu toe hebben we de robot alleen laten bewegen. De robot kan echter meer, namelijk: waarnemen. Dit zorgt ervoor dat de robot een stuk slimmer kan handelen dan dat we hiervoor hebben gezien. Hiervoor wordt de programmeertaal ook weer een beetje uitgebreid.

3.2.1 Zien

Zo kan de robot de vakjes links, voor en rechts van hem bekijken. Hij kan maar één vakje ver kijken. Ook al is dat niet veel, we hebben er toch wel iets aan. Probeer de volgende instructies eens uit:

```
1 linksIsObstakel()
2 voorIsObstakel()
3 rechtsIsObstakel()
```

Je zult zien dat de robot zijn hoofd draait in de juiste richting. Aangezien de robot altijd al zijn hoofd naar voren heeft gedraaid, zie je op regel 2 niets gebeuren.

De robot kan maar een beperkt aantal zaken onderscheiden. Zo ziet de robot alleen of ergens een obstakel staat of niet, of het wit of zwart is geverfd of dat er een baken staat. Telkens als je de robot wilt laten kijken, moet je aangeven in welke van deze dingen je geïnteresseerd bent. Voor elke combinatie van richting en ding is er een eigen instructie. De instructies zijn als volgt opgebouwd:

$$\left.\begin{array}{r}\texttt{links}\\ \texttt{voor}\\ \texttt{rechts}\end{array}\right\}\texttt{Is}\left\{\begin{array}{l}\texttt{Obstakel}\\ \texttt{Vrij}\\ \texttt{Baken}\\ \texttt{Wit}\\ \texttt{Zwart}\end{array}\right.$$

Bakens zijn groene dingen die zo nu en dan rondslingeren in de omgeving. We zullen zo zien wat we ermee kunnen doen, in plaats van ertegenaan botsen.

Hieronder vind je een tabel met alle instructies die met waarnemen te maken hebben.

links	voor	rechts
`linksIsObstakel()`	`voorIsObstakel()`	`rechtsIsObstakel()`
`linksIsVrij()`	`voorIsVrij()`	`rechtsIsVrij()`
`linksIsBaken()`	`voorIsBaken()`	`rechtsIsBaken()`
`linksIsWit()`	`voorIsWit()`	`rechtsIsWit()`
`linksIsZwart()`	`voorIsZwart()`	`rechtsIsZwart()`

3.2.2 Condities

Je hebt pas wat aan waarnemingsvermogen als je daardoor verschillende beslissingen kunt maken. Zo wil je bijvoorbeeld kunnen zeggen: “Als je links een witte stip ziet, draai dan naar links en ga erop staan”. Net zoals er een `herhaal(...){...}` constructie bestaat, is er ook een `als(...){...}` constructie. Bij deze als-constructie worden de instructies tussen accolades, {...}, alleen uitgevoerd als de voorwaarde tussen de haakjes, (...), geldt. Deze voorwaarde kan in ons geval `linksIsWit()` zijn. De eerder genoemde zin wordt dan het volgende programmaatje:

```
1  als(linksIsWit){
2          links()
3          vooruit(1)
4  }
```

Conditie

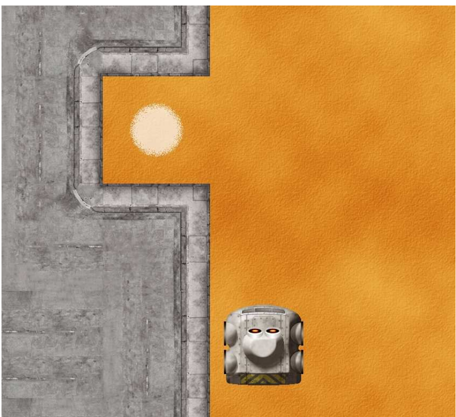
De voorwaarde tussen haakjes wordt ook wel de *conditie* genoemd. Als je helemaal chic wilt zijn, kun je het ook het *antecedent* noemen.

Dit voorgaande programmaatje was niet zo heel erg handig. We kunnen niet echt controleren of de als-constructie goed werkt. Als er namelijk links geen witte stip te zien is, zijn er geen instructies die hij kan uitvoeren. Daarom maken we een groter programma waarin de nieuwe constructie wel van pas komt.

3.2.3 Vind de witte stip

We zullen een programma gaan bekijken dat ervoor zorgt dat de robot een witte

stip zal vinden, ergens in een nis. Voor dit programma is een speciale omgeving gemaakt, namelijk *findSpot1.map*. Open deze kaart met Bestand ▷ Open kaart.



We weten van tevoren niet waar de nis met de witte stip zich bevindt. Natuurlijk kunnen we zelf naar de nis toerijden met de afstandsbediening, maar de volgende oplossing is een stuk slimmer:

```
1  herhaal(){
2      als(linksIsWit()){
3          # Er is links een witte stip
4          links()
5          vooruit(1)
6          einde
7      }
8      anders{
9          # Er is links geen witte stip
10         vooruit(1)
11     }
12 }
```

De robot kijkt naar links en controleert of hij een witte stip ziet. Als dit het geval is, draait hij naar links en gaat erbovenop staan. De robot is dan klaar met het uitvoeren van het programma. Als hij echter geen witte stip ziet, doet hij wat anders. Hij zal dan een stapje naar voren gaan.

In dit programma zie je twee nieuwe dingen. Ten eerste blijkt de als-structuur een uitgebreide variant te kennen, namelijk:

`als(...){...}anders{...}`

In deze uitgebreide vorm wordt altijd één van de twee codeblokken uitgevoerd. Als de conditie het geval blijkt te zijn, wordt alleen het deel tussen de eerste accolades uitgevoerd, *anders* alleen het deel tussen de tweede accolades. De als-constructie zonder het `anders{...}`-deel kan je nog steeds toepassen in programma's, als er niets is dat je wilt uitvoeren als de conditie niet opgaat.

Daarnaast staat er `einde` op regel 6. Als het programma bij dit woord aankomt nadat het `vooruit(1)` heeft uitgevoerd, zal het stoppen. Merk op dat achter `einde` geen haakjes staan. Dit is gedaan om aan te geven dat het geen directe instructie is voor de robot, maar meer een instructie die de uitvoer van het programma controleert. Zo'n instructie wordt een sleutelwoord, of in het Engels een *keyword*, genoemd. We zullen later nog meer sleutelwoorden tegenkomen.

SLEUTELWOORD

Om te laten zien dat dit programma ook nuttig is als de nis ergens anders in de muur zit, kan je de kaart *findSpot2.map* openen. De nis zit hier twee hokjes verder. Als je precies hetzelfde programma nogmaals uitvoert, vindt de robot de stip ook! Ook al wist de robot van tevoren niet waar de stip zou zijn, door onze slimme aanpak is het toch mogelijk de robot in zijn doel te laten slagen.

Verdieping — Reflexief Paradigma

De manier waarop wij onze robot hebben geprogrammeerd, valt onder het zogenaamde *reflexieve paradigma*. Bij elke robot zou je de bijbehorende programma's kunnen indelen in *waarneming* en *acties* die de robot uitvoert. In ons geval is dat zien en bewegen. De robot maakt zijn beslissingen door alleen informatie te gebruiken die hij direct kan waarnemen. Zo betekent `linksIsWit()`: kijk of je *op dit moment, op deze plek* witte verf op de grond ziet. Het betekent niet: als je vroeger ooit eens ergens iets wits zag, doe dan het volgende.

Deze directe manier van programmeren wordt in kunstmatige intelligentie het reflexieve paradigma genoemd. Het beslissen van wat je gaat doen is een directe reflex op de omgeving. Van alle mogelijke aanpakken lijkt dit nog de simpelste, ook al kan het verdraaid lastig zijn een goed reflexief programma te maken.

Er is wel eens gesuggereerd dat insecten zich gedragen als robots die volgens deze denkwijze zijn geprogrammeerd. Valentino Braitenberg beschrijft in zijn boekje “Vehicles”, oftewel “Voertuigen”, robotjes die hetzelfde gedrag als insecten vertonen aan de hand van een paar simpele instructies.

3.2.4 Logische expressie

Het kan voorkomen dat er geen waarneemingsinstructie bestaat als je een conditie voor een lastige situatie wilt beschrijven. Hoe kan je bijvoorbeeld aangeven dat beide zijden van de robot vrij moeten zijn om een bepaalde actie uit te voeren? Een oplossing is of twee `als`-structuren te nesten:

```
als(linksIsVrij()){
    als(rechtsIsVrij()){
        # doe de gewenste actie
    }
}
```

Dit kan echter ook iets mooier worden uitgedrukt als volgt:

```
als(linksIsVrij() en rechtsIsVrij()){
    # doe de gewenste actie
}
```

We kunnen blijkbaar gebruik maken van het sleutelwoord `en` om twee waarnemings instructies aan elkaar te knopen. Zo geven we preciezer aan wanneer de commando's van een `als`-structuur moeten worden uitgevoerd, namelijk als zowel de ene waarnemingsinstructie opgaat als de andere. We zouden het zelfs nog preciezer kunnen maken door nog een waarnemingsinstructie toe te voegen:

```
als(linksIsVrij() en rechtsIsVrij() en voorIsBaken()){
    # doe de gewenste actie
}
```

Een ander sleutelwoord dat we op soortgelijke wijze kunnen gebruiken is `of`. In dat geval moet minstens één, maar mogelijk ook beide, waarnemingsinstructie opgaan.

```
als(linksIsVrij() of rechtsIsVrij()){
    # doe de gewenste actie
}
```

Dit doet hetzelfde als hetvolgende script:

```
als(linksIsVrij()){
    # doe de gewenste actie
}
anders als(rechtsIsVrij()){
    # doe de gewenste actie
}
```

Tot slot hebben we het sleutelwoord `niet`, waarmee we een waarneming precies kunnen omkeren. Zo kunnen we hetvolgende uitdrukken:

```
als(niet linksIsVrij()){
    # doe de gewenste actie
}
```

Ga na dat de volgende twee scripts hetzelfde zouden doen:

```
als(linksIsObstakel()){
    # doe de gewenste actie
}
```

```
1 als(linksIsVrij()){
2     # doe niets
3 }
4 anders{
5     # doe de gewenste actie
6 }
```

We zullen deze mogelijkheden iets formeler herhalen. De conditie van `als`-structuren (en `herhaalZolang`-structuren) zijn zogenaamde logische expressies. Deze expressies zijn een voorwaarde die de waarden `waar` of `onwaar` aannemen, waarna naar het overeenkomende deel van de code kan worden gesprongen om dat uit te voeren. Een logische expressie kan een van de waarnemingsintructies zijn, zoals bijvoorbeeld `linksIsWit()`. Deze instructie kan `waar` zijn als er daadwerkelijk witte verf links van de robot is op het moment dat het wordt gevraagd. Als er links geen witte verf is, is de logische expressie `onwaar`. Waarnemingsinstructies kunnen ook worden samengevoegd met de sleutelwoorden `niet`, `en`, `of`. Deze sleutelwoorden worden ook wel *logische operatoren* of *booleaanse operatoren* genoemd.

Operatie	Alternatieve notatie	Aantal argumenten	Uitleg
`niet`	~	1	Keert de waarde van het argument om.
`en`	&	2	Alleen waar als beide argumenten waar zijn.
`of`	|	2	Waar als een van beide argumenten waar is.

Deze operatoren mag je ook anders noteren, maar doen dan nog precies hetzelfde. Zie hiervoor de kolom met de alternatieve notatie. De logische expressie `niet linksIsVrij() en rechtsIsWit()` is dus bijvoorbeeld hetzelfde als `~ linksIsVrij() &rechtsIsWit()`. Deze laatste notatie is misschien wat lastiger te onthouden, maar is iets korter en wordt daarom veel gebruikt in andere programmeertalen.

Het was bij nader inzien eigenlijk niet direct duidelijk wat we nu bedoelden met de expressie `niet linksIsVrij() enrechtsIsWit()`. Staat er dat "links niet vrij is en rechts wel"? Of staat er dat "het niet zo moet zijn dat zowel links als rechts vrij zijn"? Ze betekenen echt iets anders, dus willen we ook graag weten hoe de robot dit opvat. Het antwoord is dat sommige logische operatoren *sterker binden* dan andere, wat inhoud dat er eerder naar hun argumenten wordt gekeken. Dit is hetzelfde als met rekenkundige uitdrukkingen. Hoeveel is $3 + 4 \times 2$? We weten dat we eerst moeten vermenigvuldigen en daarna pas optellen. Daarom is het antwoord 11 en niet 14. Zo'n zelfde "Meneer van Dale"-regel voor $-$, $\times$, $+$ geldt ook

voor `niet`, `en`, `of` in dezelfde volgorde. `niet linksIsVrij() en rechtsIsWit()` betekent dus "links niet vrij is en rechts wel". Ook kan je bij logische expressies gebruik maken van haakjes om de volgorde te beïnvloeden. `niet(linksIsVrij() en rechtsIsWit())` betekent daarom "het niet zo moet zijn dat zowel links als rechts vrij zijn".

3.3 De herhaalZolang-lus

Omdat het in praktijk nogal vaak voorkomt dat je instructies alleen wilt blijven uitvoeren zolang een conditie geldt, is er een extra herhaal-structuur in de programmeertaal opgenomen. Stel je voor dat de robot net zolang vooruit moet rijden totdat hij een obstakel tegenkomt en dan moet omkeren. Dit programma zou dan uitstekend voldoen:

```
1  herhaal(){
2      als(voorIsVrij()){
3          vooruit(1)
4      }
5      anders{
6          doorbreekLus
7      }
8  }
9  rechts() rechts()
```

DOORBREEK LUS

Merk op dat hier op regel 6 een nieuw sleutelwoord staat: doorbreekLus. Dit sleutelwoord beïnvloedt net als `einde` de manier waarop het programma wordt uitgevoerd. Ook hier staan er geen haakjes achter het sleutelwoord.

Als `doorbreekLus` wordt uitgevoerd, zal de herhaling waarin het staat altijd stoppen. In plaats dat de robot helemaal stopt, zoals bij `einde` het geval is, gaat de robot verder met de instructies die achter de herhaling staan. In dit geval zal hij twee keer naar rechts draaien, zodat hij zich omdraait zoals we wilden.

Met de nieuwe herhaalZolang-lus kunnen we hetzelfde bereiken met:

```
herhaalZolang(voorIsVrij()){
        vooruit(1)
}
rechts() rechts()
```

Dit leest wel zo prettig. In plaats dat de herhaling altijd maar doorgaat, wordt het codeblok tussen de accolades zo vaak uitgevoerd als de conditie opgaat.

Verdieping — SYNTACTIC SUGAR

De herhaalZolang-lus is strikt gezien een beetje overbodig. We kunnen immers hetzelfde resultaat bereiken met de herhaal-lus en de als-constructie. In dat opzicht is het voor luie mensen die minder willen schrijven. Toch hebben de luie mensen wel gelijk dat het programma er met de herhaalZolang-lus een stuk leesbaarder op is geworden. Het is nu makkelijker te begrijpen wat er precies gebeurt.
In het algemeen wordt het fenomeen van (al dan niet overbodige) structuren in een programmeertaal *syntactic sugar* genoemd in het Engels. Dit betekent vrij vertaald dat de manier van programmanotatie, oftewel de tekst, er met de extra structuren iets smeuïger uit komt te zien, terwijl we er niet echt meer mee kunnen uitdrukken.

3.4 Meer basisinstructies: verven en grijpen

Als je de afstandsbediening er nog even bijpakt, Uitvoeren ▷ Afstandsbediening, zie je naast de bewegingsknoppen (de pijlen) nog een aantal knoppen. Die staan hieronder beschreven:

Je kunt hiermee alvast even spelen en proberen te achterhalen wat de gevolgen zijn van deze nieuwe instructies.

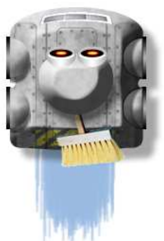
3.4.1 Verven

Met `verfWit()` en `verfZwart()` zet je de kwast van de robot op de grond, met respectievelijk witte en zwarte verf. Dit zorgt ervoor dat er een witte stip op de grond

`verfWit()`	zet de kwast met witte verf op de grond.
`verfZwart()`	zet de kwast met zwarte verf op de grond.
`stopVerven()`	haal de kwast van de grond.
`pakOp()`	probeer met de grijper een baken op te pakken.
`zetNeer()`	zet een eerder opgepakt baken neer.

Tabel 3.1: Meer basisinstructies: verven en grijpen.

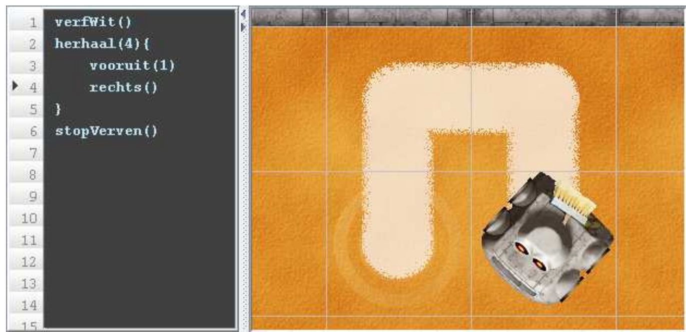
blijft staan. Als je na een van deze instructies gaat bewegen zie je dat je een streep achterlaat op de grond. Je zult een streep blijven trekken totdat `stopVerven()` wordt uitgevoerd. De kwast wordt dan weer opgeborgen.



Let op dat je de instructies met precies hetzelfde hoofdlettergebruik schrijft zoals hier staat. Dus bijvoorbeeld `verfWit()` met een kleine "v" en een hoofdletter "W". Dit was vanwege de hoofdlettergevoeligheid in de programmeertaal (zie pagina 7).

Met de kwast kunnen we simpele tekeningen maken. Het volgende programma tekent bijvoorbeeld een klein vierkantje.

```
verfWit()
herhaal(4){
        vooruit(1)
        rechts()
}
stopVerven()
```



Als je de kwast neerzet en meteen weer ophaalt, zal er een stip op de grond worden getekend.

```
verfWit()
stopVerven()
vooruit(1)
```

Je kunt nu proberen de robot je naam te laten schijven.

3.4.2 Grijpen

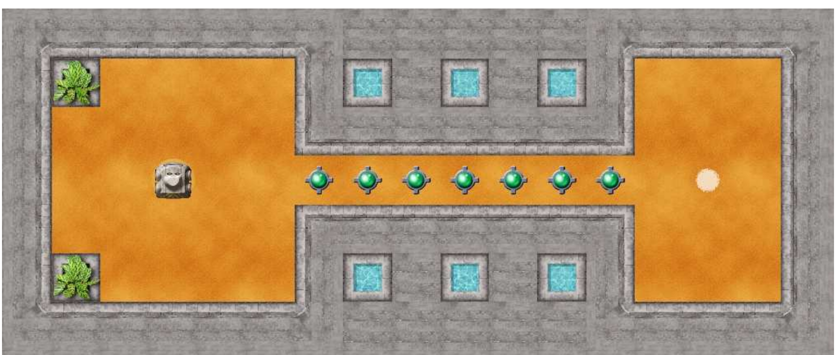
De robot beschikt tevens over een grijper waarmee hij dingen kan oppakken. Het enige wat in de omgeving opgepakt kan worden zijn bakens. Deze bakens doen helemaal niets, behalve in de weg zitten. Daarom is het handig dat je ze kunt oppakken en ze ergens anders neer kunt zetten.

Je kunt alleen bakens oppakken waar je recht voor staat, en alleen als je nog geen baken bij je hebt. De robot heeft slechts plaats om één baken tegelijkertijd

met zich mee te dragen. Als je een mooie plek hebt gevonden kan je het baken neerzetten op een vrije plaats, dus niet op een ander baken of op een muur.

3.4.3 Een puzzel

Laten we deze instructies maar eens gaan inzetten om een interessant probleem aan te pakken. Open daarvoor de kaart *passBeacons.map*.



De robot staat aan de linkerkant in een ruimte. Rechts is een ruimte met een witte stip. Het probleem is dat de robot op de witte stip moet komen te staan. Het was eenvoudig geweest als er niet zoveel bakens hadden gestaan in het gangetje dat de twee ruimten verbindt.

Als we dit met de hand moeten doen, dus met behulp van de afstandsbediening, dan zijn we wel even bezig. De manier die dan het snelst zou zijn, gaat als volgt:

1. rij naar het eerste baken,
2. pak dan telkens een baken op…
3. …zet het achter je neer,

4. …ga naar het volgende baken.

5. als er geen bakens meer zijn, rij het laatste stukje naar de stip.

Deze handelingswijze kunnen we gelukkig vertalen naar een programma:

```
1  # rij naar het eerste baken
2  rechts()
3  vooruit(2)
4
5  herhaalZolang(voorIsBaken()){
6          # pak een baken op en zet het achter je neer
7          pakOp()
8          rechts() rechts()
9          zetNeer()
10
11         # ga naar het volgende baken
12         rechts() rechts()
13         vooruit(1)
14 }
15
16 # er zijn geen bakens meer
17 # rij het laatste stukje naar de stip
18 vooruit(2)
```

Dit programma bespaart een hoop werk. Stel je voor dat de gang met bakens nog tien keer langer was geweest! Even goed nadenken over een programma weegt dan al gauw op tegen heel veel handwerk.

3.5 Zelf instructies definiëren

Naast het gebruik van basisinstructies heb je de mogelijkheid zelf instructies te definiëren. Dit is handig wanneer je vaker het zelfde rijtje instructies achter elkaar wilt uitvoeren. Laten we even het geval nemen dat je zo nu en dan een vierkantje wilt tekenen. Tot nu toe moest je, telkens wanneer je een vierkantje wilde tekenen, de herhaal-lus op de juiste plaats in het programma zetten.

```
# doe hier een paar instructies...
links()
achteruit(1)

# teken een vierkantje
verfWit()
herhaal(4){
   vooruit(2)
   rechts()
}
stopVerven()

# nog een paar instructies
rechts()
vooruit(3)

# teken nog een vierkantje
verfWit()
herhaal(4){
   vooruit(2)
   rechts()
}
stopVerven()
```

Omdat het tekenen van het vierkantje niet op een regelmatige basis hoeft terug te komen in het programma, kunnen we niet altijd van een herhaal-lus gebruik maken. Toch is het een beetje jammer dat er twee keer precies dezelfde instructies geschreven moeten worden om een vierkantje te tekenen.

3.5.1 Naamgeving

Je kunt zelf een instructie definiëren die een vierkantje tekent. Eerst moeten we een naam bedenken. In principe mag je elke naam kiezen die je wilt, maar je moet je aan een paar regels houden:

- Zo mag je alleen namen kiezen die bestaan uit een combinatie van letters en cijfers, dus niet bijvoorbeeld: "J%p&M@rie".
- Iedere naam moet beginnen met een letter.
- Net als bij de basisinstructies mogen er ook geen spaties in de naam voorkomen. Dus als we onze instructie "teken vierkant" willen noemen, mag dat niet.

De truc om woorden toch een beetje gescheiden te houden is door ieder woord met een hoofdletter te laten beginnen. Het is echter gebruikelijk om eigen instructies, net als basisinstructies, te laten beginnen met een kleine letter. Wij zullen onze eigen instructie dan ook "tekenVierkant" noemen.

3.5.2 Procedure definitie

Je moet vastleggen hoe je eigen instructie "tekenVierkant" moet worden uitgevoerd. Dit doe je met behulp van een procedure. De structuur van een procedure ziet er als volgt uit:

```
procedure jeEigenNieuweNaam(...){...}
```

Tussen de haakjes kan je aangeven of je gebruik wilt maken van argumenten, zoals bij de `vooruit(...)`-instructie. Dit zullen we even later gaan gebruiken, maar nu nog niet. Tussen de accolades schrijf je alle instructies op die nodig zijn.

Onze nieuwe instructie "tekenVierkant" wordt bijvoorbeeld als volgt gedefinieerd:

```
1
2  procedure tekenVierkant(){
3      verfWit()
4      herhaal(4){
5          vooruit(2)
6          rechts()
7      }
8      stopVerven()
9  }
```

Dit programma doet nog helemaal niets. Onthoud dat instructies in deze definitie *nooit automatisch word uitgevoerd*. Als we onze nieuwe instructie ook echt willen gebruiken, zullen we de *procedure moeten aanroepen*. Dit doe je in dit geval door `tekenVierkant()` nu als een basisinstructie tussen de andere instructies te plaatsen. Het eerdere programma kunnen we nu als volgt schrijven:

AANROEPEN

```
1  # doe hier een paar instructies...
2  links()
3  achteruit(1)
4
5  # teken een vierkantje
6  tekenVierkant()
7
8  # nog een paar instructies
9  rechts()
10 vooruit(3)
11
12 # teken nog een vierkantje
13 tekenVierkant()
14
15 # de definitie van "tekenVierkant"
16 procedure tekenVierkant(){
17     verfWit()
18     herhaal(4){
19         vooruit(2)
20         rechts()
21     }
22     stopVerven()
23 }
```

Op de regels 6 en 13 zie je de aanroep van de procedure. Op deze regels wordt er even naar de definitie gesprongen, worden alle instructies daarin uitgevoerd en wordt weer teruggesprongen waar hij gebleven was. Om het iets duidelijker te maken dat de instructies in de definitie worden hergebruikt, zijn de twee aanroepen met pijlen ingetekend:

Het is even wennen dat er zo door het hele programma wordt heengesprongen. Toch kan je wel zeggen dat het programma prettiger leest met onze eigen definities. Merk overigens op dat we onze eigen instructie ook een heel andere naam hadden kunnen kiezen. Bijvoorbeeld “tv”, van Teken Vierkant, of “PrinsWillemAlexanderVanOranjeNassau”. Je moet alleen even opletten dat je dit dan concequent doet. Dus zowel in de proceduredefinitie, als in de aanroepen moet je dan dezelfde naam kiezen.

3.5.3 Argumenten

De instructie `tekenVierkant()` tekent nu een vierkant dat twee hokjes breed en hoog is. Als we nu een vierkant willen tekenen dat 1 bij 1 is, of 3 bij 3, moeten we

```
# doe hier een paar instructies...
links()
achteruit(1)

# teken een vierkantje
tekenVierkant()

# nog een paar instructies
rechts()
vooruit(3)

# teken nog een vierkantje
tekenVierkant()

# de definitie van "tekenVierkant"
procedure tekenVierkant(){
    verfWit()
    herhaal(4){
        vooruit(2)
        rechts()
    }
    stopVerven()
}
```

dan telkens nieuwe procedures maken? Gelukkig hoeft dat niet. We kunnen de procedure *parametriseren*. Dat betekent dat we een argument kunnen meegeven in een procedureaanroep, zoals bijvoorbeeld `tekenVierkant(3)`. Hiervoor moet je in de proceduredefinitie een variabele opnemen. Deze variabele mag je ook een eigen naam geven, volgens dezelfde regels als bij het maken van eigen instructies. In de procedure definitie noemen we deze variabele de parameter, op het moment dat je een concrete waarde kiest in een aanroep noem je het een argument. Om niet nog meer verwarring te schoppen, een voorbeeld:

PARAMETRISEREN

PARAMETER

ARGUMENT

```
# teken een vierkant van 1 bij 1
# het argument van ‘tekenVierkant’ is ‘1’
tekenVierkant(1)

# teken een vierkant van 3 bij 3
# het argument van ‘tekenVierkant’ is ‘3’
tekenVierkant(3)

# de definitie van ’tekenVierkant’
# het heeft de parameter ‘maat’

procedure tekenVierkant(maat){
    verfWit()
    herhaal(4){
        vooruit(maat)  # hier wordt de parameter gebruikt
        rechts()
    }
    stopVerven()
}
```

Allereerst kijken we naar de verandering in de definitie van “tekenVierkant”. Deze heeft namelijk de parameter “maat” gekregen. Op het moment dat deze parameter voor het eerst genoemd wordt, dus tussen de haakjes op regel 12, mogen we de naam van de parameter bepalen. Hier is gekozen voor “maat”, maar we hadden net zo goed voor “grootte”, “sinterklaas” of “x” kunnen kiezen. Op regel 15 wordt de parameter gebruikt in de instructie `vooruit(...)`. We waren gewend dat hier een concreet getal stond, een ‘echt’ getal als 1, 7 of 42. In de definitie weten we op dit moment nog niet exact hoeveel we vooruit moeten en daarom gebruiken we de variabele “maat”. We willen namelijk dat hij precies zoveel stapjes vooruit gaat als later wordt opgegeven.

Op de regels 4 en 7 wordt de procedure “tekenVierkant(maat)” aangeroepen met verschillende argumenten, namelijk `tekenVierkant(1)` en `tekenVierkant(3)`. Als de eerste aanroep wordt uitgevoerd, weten we eindelijk wat “maat” precies is, namelijk 1. En dus wordt er op regel 15 eigenlijk `vooruit(1)` uitgevoerd. Als de hele procedure is uitgevoerd, zal `tekenVierkant(3)` worden aangeroepen. In dit geval is “maat” gelijk aan 3. Op regel 15 zal dus `vooruit(3)` worden uitgevoerd. Op deze manier kan je makkelijk meerdere vierkanten tekenen met een verschillende maat.

Omdat we meerdere argumenten mogen meegeven kunnen we bijvoorbeeld ook makkelijk een rechthoek definiëren. Dit kan er als volgt uitzien:

```
# definieer het tekenen van een rechthoek
procedure tekenRechthoek(breedte, hoogte){
    verfWit()
    herhaal(2)        {
        vooruit(hoogte)
        rechts()
        vooruit(breedte)
        rechts()
    }
    stopVerven()
}
```

We kunnen nu de rechthoek in de rest van het programma aanroepen met bijvoorbeeld `tekenRechthoek(2,3)`. Merk op dat hier twee argumenten worden meegegeven. De '2' komt hier overeen met 'breedte', en de '3' met 'hoogte'. Er zal dus een rechthoek worden getekend van 2 bij 3, gezien vanuit het oogpunt van de robot.

Een vierkant is eigenlijk een speciaal soort rechthoek. /een vierkant is namelijk een rechthoek waarbij alle zijden even lang zijn. Het leuke is dat we in de definitie van `tekenVierkant` gebruik kunnen maken van de algemenere procedure `tekenRechthoek`. Het volgende programma demonstreert deze mogelijkheid.

```
# voer de nieuwe definities uit door ze aan te roepen
tekenVierkant(2)
vooruit(1)
rechts()
tekenRechthoek(1, 3)


# definieer het tekenen van een vierkant
# in termen van een rechthoek!
procedure tekenVierkant(maat){
    tekenRechthoek(maat, maat)
}

# definieer het tekenen van een rechthoek
procedure tekenRechthoek(breedte, hoogte){
    verfWit()
    herhaal(2)        {
        vooruit(hoogte)
        rechts()
        vooruit(breedte)
        rechts()
    }
    stopVerven()
}
```

Dit hergebruik van eerder gedefinieerde procedures is een erg krachtig principe. Bij grotere programma is het erg prettig als je kunt werken met je eigen instructies.

Hoofdstuk 4

Interessante programma's

Dit hoofdstuk wordt weldra uitgebreid.

4.1 Lijnvolger

De robot kan een lijn volgen met het volgende programma. Laat hem dit uitvoeren in *default.map*.

```
rechts()
vooruit(8)

herhaal()
{
        als(voorIsWit()){
                vooruit(1)
        }
        anders als(rechtsIsWit()){
                rechts()
        }
        anders als(linksIsWit()){
                links()
        }
        anders als(voorIsObstakel()){
                einde
        }
}
```

4.2 Doolhofdoorkruiser

De robot moet het baken aan de andere kant van het doolhof zien te vinden. De robot zal altijd uit een doolhof komen door de muur rechts te blijven volgen en komt zo bij het baken uit. Open hiervoor *maze1.map*.

```
herhaal(){
    als(rechtsIsObstakel()){
        als(voorIsVrij()){
            vooruit(1)
        }
        anders{
            links()
        }
    }
    anders{
        rechts()
        vooruit(1)
    }

    als(voorIsBaken()){
        # eureka!
        pakOp()
        einde
    }
}
```

Appendix A: Basisinstructies

Bewegen	`vooruit(n)`	Beweeg n stappen vooruit
	`achteruit(n)`	Beweeg n stappen achteruit
	`links()`	Draai 90 graden naar links
	`rechts()`	Draai 90 graden naar rechts
Verven	`verfWit()`	Zet de kwast neer met witte verf
	`verfZwart()`	Zet de kwast neer met zwarte verf
	`stopVerven()`	Berg de kwast weer op
Grijpen	`pakOp()`	Pak het baken recht voor je op
	`zetNeer()`	Zet het baken recht voor je neer

Zien

links	voor	rechts
`linksIsObstakel()`	`voorIsObstakel()`	`rechtsIsObstakel()`
`linksIsVrij()`	`voorIsVrij()`	`rechtsIsVrij()`
`linksIsBaken()`	`voorIsBaken()`	`rechtsIsBaken()`
`linksIsWit()`	`voorIsWit()`	`rechtsIsWit()`
`linksIsZwart()`	`voorIsZwart()`	`rechtsIsZwart()`

Appendix B: Programmeerstructuren

Herhalingen

`herhaal(`n`)`{...*instructies*...}
herhaalt de instructies tussen accolades precies *n* keer.

```
1  # een 2x2 vierkantje
2  herhaal(4)
3  {
4          vooruit(2)
5          rechts()
6  }
```

`herhaal()`{...*instructies*...}
blijft de instructies tussen accolades telkens herhalen.

```
1  # blijf alsmaar rechtdoor gaan
2  # (zal uiteindelijk blijven botsen)
3  herhaal()
4  {
5          vooruit()
6
7  }
```

`herhaalZolang(`*conditie*`)`{...*instructies*...}
herhaalt de instructies tussen accolades net zo lang totdat de conditie niet meer opgaat. De conditie moet een waarnemingsinstructie zijn (bijv. `voorIsVrij()`)

```
1  # blijf rechtdoor gaan, totdat je niet verder kunt
2  herhaalZolang(voorIsObstakel)
3  {
4          vooruit(1)
```

```
5
6   }
```

`doorbreekLus`
zorgt ervoor dat de lus waar deze instructie in staat wordt beëindigd, en er wordt verder gegaan met de eerste instructie na deze lus.

```
1   # blijf rechtdoor gaan, totdat je niet verder kunt
2   herhaal()
3   {
4           als(voorIsObstakel())
5           {
6                   doorbreekLus
7           }
8           anders
9           {
10                  vooruit(1)
11          }
12  }
```

Als-structuren

`als(`*conditie*`)`{...*instructies*...}
voert de instructies tussen accolades alleen uit als de conditie opgaat. De conditie moet een waarnemingsinstructie zijn (bijv. `voorIsVrij()`)

```
1   # als je links een witte stip ziet, maak hem zwart
2   als(linksIsWit())
3   {
4           links()
5           vooruit(1)
6           verfWit()
7           stopVerven()
8           achteruit(1)
9           rechts()
10  }
```

`als(`*conditie*`)`{...*instructies*...}`anders`{...*instructies*...}
voert de instructies tussen het eerste paar accolades alleen uit als de conditie opgaat, anders voert het alleen de instructies uit tussen het tweede paar accolades. De conditie moet een waarnemingsinstructie zijn (bijv. `voorIsVrij()`)

```
1   # als je links een witte stip ziet, maak hem zwart,
2   # rij anders een stukje door;
```

```
3   als(linksIsWit())
4   {
5           links()
6           vooruit(1)
7           verfWit()
8           stopVerven()
9           achteruit(1)
10          rechts()
11  }
12  anders
13  {
14          vooruit(1)
15  }
```

`als`(*conditie*){...*instructies*...} `anders als` (*conditie*){...*instructies*...}
is de makkelijkere notatie zonder extra accolades voor: `als`(*conditie*1){...} `anders` {`als` (*conditie*2){...}}. Het codeblok van `anders` wordt alleen uitgevoerd als zijn overeenkomende conditie het geval is. Deze constructie is met name nuttig wanneer er meerdere verschillende gevallen moeten worden gecontroleerd en uitgevoerd. In het voorbeeld volgt de robot een stapje een witte lijn op de grond.

```
1   als(voorIsWit()){
2           # alleen als voor is wit
3           vooruit(1)
4   }
5   anders als(rechtsIsWit()){
6           # alleen als rechts is wit
7           rechts()
8           vooruit(1)
9   }
10  anders als(linksIsWit()){
11          # alleen als links is wit
12          links()
13          vooruit(1)
14  }
15  anders{
16          # alleen als ALLE voorgaande condities niet het geval waren
17          rechts()
18          rechts()
19          vooruit()
20  }
```

Procedures

`procedure` $naam(par_1, par_2, \ldots, par_n)\{\ldots instructies \ldots\}$

definieert een nieuwe procedure met een zelf gekozen *naam*. De procedure kan beschikken over een aantal parameters die hier $par_1, par_2, \ldots, par_n$ worden genoemd. Dit zijn de namen van de variabelen die je wilt gebruiken in de instructies die tussen accolades staan.

```
1  # definieer het tekenen van een rechthoek
2  procedure rechthoek(breedte, hoogte)
3  {
4          verfWit()
5          herhaal(2)
6          {
7                  vooruit(hoogte)
8                  rechts()
9                  vooruit(breedte)
10                 rechts()
11         }
12         stopVerven()
13 }
```

$naam(arg_1, arg_2, \ldots, arg_n)$

is de aanroep van de procedure met dezelfde naam, en hetzelde aantal parameters. De argumenten $arg_1, arg_2, \ldots, arg_n$ zijn de concrete waarden voor de parameters in de procedure definitie.

```
1  vooruit(1)
2  rechthoek(3,2) # de aanroep gebruikt de bovenstaande definitie
3  vooruit(3)
4  rechthoek(1,4) # nog een aanroep, nu met andere argumenten
```

`retourneer`

zorgt ervoor dat de uitvoer van de huidige procedure wordt afgebroken. De uitvoer zal worden hervat bij de eerste insructie na de procedure-aanroep. Op deze manier is het mogelijk slehts een eerste deel van de procedure te laten uitvoeren.

```
1  piet()
2  # hier gaat de uitvoer verder nadat 'piet' is voltooid
3  links()
4  vooruit(1)
5
6  procedure piet(){
7      vooruit(5)
8      als(voorIsObstakel()){
9              # breek de uitvoer van deze procedure af
```

```
10             retourneer
11     }
12     vooruit(3)
13 }
```

Einde

`einde`

zorgt ervoor dat het hele programma direct stopt met de uitvoer als deze instructie wordt bereikt.

```
1  # stop na 5 stappen, of eerder als je rechts een baken ziet
2  herhaal(5)
3  {
4          vooruit(1)
5          als(rechtsIsBaken())
6          {
7                  einde # breek het programma af
8          }
9  }
10 # normaal einde van het programma
```

Index

aanroepen, 37
acties, 26
afstandsbediening, 5
als-constructie, 24
als. . . anders -constructie, 26
antecedent, 24
argument, 13, 39
automatiseren, 1
autonoom, 2

bakens, 33
basisinstructies, 3
berichtenvenster, 4, 7
bewegen, 13
botsingen, 15

codeblok, 18
commentaar, 10
commutatief, 16
conditie, 24
controle structuren, 8

doorbreekLus, 30

einde, 30
en, 27
expressiviteit, 17

flow of control, 8
foutmelding, 7

grammatica, 9
grijpen, 33

haakjes, 13
hekje, #, 10
herhaal-lus, 17
herhaalZolang-lus, 30
hoofd draaien, 23
hoofdlettergevoeligheid, 7

informatica, v
ingeneste structuur, 22
instructies, 1
instructies definiëren, 35

keyword, 26
kunstmatige intelligentie, v

logische expressies, 29
logische operatoren, 29

monitor, 4, 5

naamgeving, 36
Nasa, 2
natuurlijke taal, 9
neerzetten, 34
nesting, 22
niet, 28

of, 27
oneindige herhaling, 21
onwaar, 29
openen, 7
operatoren, 29
oppakken, 33
opslaan, 7

parameter, 39
parametriseren, 39
pijltjesknoppen, 5

procedure, 37
programma, 1
programmeertaal, 8

reflexief paradigma, 26
regelnummers, 18
Robo, v
RoboMind, 1

scriptvenster, 4
sleutelwoord, 26
spaties, tabs, nieuwe regels, 9
stoppen, 22
syntactic sugar, 31

tekenen, 32

uitdrukkingsmogelijkheden, 17
uitvoeren, 7
Uitvoerpaneel, 4
uitvoervolgorde, 8

verdieping, vi
verven, 32
volg robot, 5
voorwaarde, 24

waar, 29
waarnemen, 22
waarnemingsinstructie, 27
welgevormdheid, 9
white space, 9
wiskunde, v

zoom in, 5
zoom uit, 5
zoomen, 5