Denkende Machines

Computers, Rekenen, Redeneren

Jan van Eijck, Jan Jaspars, Jan Ketting, Marc Pauly

Najaar 2001

Informatie in Context

onder redactie van ...

Denkende Machines is ontwikkeld met subsidie van het Spinoza *Logic in Action* project van Johan van Benthem. De samenstelling werd begeleid door een klankbord van leraren en β-cultuurverspreiders bestaande uit Johan van Benthem (hoogleraar logica), Wim Berkelmans (directeur *Stichting Vierkant voor Wiskunde*), Christian Bokhove (docent informatiekunde), Anne Kaldewaij (informaticus en universiteitsbestuurder), Jef Kolle (docent wiskunde), Peter Lammers (docent economie), Jan de Ruijter (docent wiskunde), Hubert Slings (redacteur *Tekst in Context*), Marco Swaen (docent lerarenopleiding wiskunde). Bij de voorbereiding waren ook Saskia de Vries en Suzanne Bogman van AUP betrokken, en bij de eindredactie Arnout van Ommen, Errit Petersma en Jaap Wagenaar. Voorlopige versies van de tekst werden elektronisch ter beschikking gesteld, via de *Informatica in het Voortgezet Onderwijs* website, `http://www.informaticavo.nl`, aan, en/of van commentaar voorzien door: G. Alberts (Amsterdam), K.J. Buijs (Tilburg), P. van Cauwenberghe (Maastricht), P. Cools (Den Bosch), P. Geelen (Nijmegen), P. Hage (Doetinchem), P. Harmsen (Hoorn), A.P. Hartsuijker (Enschede), A. van Kan (Culemborg), H. van Keulen (Zwolle), E. Kerstges (Rijswijk), R. Koortens (Heemstede), F. Kuipers (Rijswijk), G. de Mooij (Delft), J.C. Mulders (Den Bosch), R. den Ouden (Spijkenisse), F. Peeters (Ede), J. Tolboom (Groningen), R. Vanderhoek (Zierikzee), T. Velzeboer (Amstelveen), A. Verheijen (Roermond), L.J.C. Warrink (Leeuwarden), E.P. Wijnvoord (Den Haag), H.E.N. de Wolde (Geleen).

Elektronische ondersteuning op internet

De website die hoort bij dit boek is te vinden op het volgende adres:

`http://www.science.uva.nl/denkendemachines/`

Deze site, gemaakt door Jan Jaspars, bevat interactieve animaties en programma's die de belangrijkste onderwerpen uit het boek illustreren. Een aantal van die programma's kan bovendien gebruikt worden bij het maken van de opgaven. Naast het speciaal voor het boek ontwikkelde materiaal geeft de website ook verwijzingen naar andere sites over de geschiedenis en theorie van computers. Bovendien kan de lezer via de site reacties en commentaar op het boek kwijt.

De reeks *Informatie in Context* is ontwikkeld voor gebruik in de Tweede Fase. Met behulp van aanpassingen in de dosering en docering is de reeks op tal van manieren inzetbaar: zowel voor HAVO 4/5 als voor VWO 4/5/6, zowel voor klassikale behandeling als voor zelfstudie, zowel voor integrale bestudering als voor gedeeltelijk gebruik, zowel voor vrijblijvende kennismaking als voor toetsing met behulp van vragen en opdrachten op verschillende niveaus.
Denkende Machines is het eerste deel in de reeks *Informatie in Context*.

Inhoudsopgave

Hoofdstuk 1

Rekenen met Machines

1.1 De Geschiedenis van de Computer

De geschiedenis van de computer, opgevat als *elektronische digitale reken- en redeneermachine*, is tamelijk kort. Elektronische digitale rekenmachines werden voor het eerst gebouwd aan het eind van de Tweede Wereldoorlog. Computers in de moderne zin van het woord bestaan dus nog niet veel langer dan een halve eeuw.

De computer is een *machine*: een mechanisch werktuig voor rekenen en redeneren. *Elektronisch* slaat op het gebruik van vacuümbuizen (later transistors, nog later chips) voor het schakelen. De machine heet *digitaal* omdat hij informatie verwerkt die gecodeerd is met behulp van cijfers (Engels: digits, van het Latijnse woord voor *vinger*, het allereerste rekenhulpmiddel). Computers cijferen meestal in het binaire talstelsel, hetgeen wil zeggen dat ze alleen de cijfers 0 en 1 gebruiken. Over talstelstels komen we verderop nog uitvoerig te spreken.

Opdracht 1.1 *In de technische specificatie van computerapparatuur kun je tegenkomen: '256 kleuren' of '64 MB RAM geheugen'. Waar komen die getallen 256 en 64 vandaan?*

Opdracht 1.2 *Tegenover digitaal staat analoog. Geef voorbeelden van digitale en analoge apparaten (in ruime zin).*

Een computer is meer dan een rekenmachine: je kunt er veel meer mee doen dan simpelweg optellen, aftrekken, vermenigvuldigen en delen. Overigens is rekenen op zichzelf ook weer veel meer dan optellen, aftrekken, vermenigvuldigen en delen. *Rekenen en redeneren* is bedoeld om dit *meer* te vatten. In dit boek willen we laten zien wat die twee activiteiten met elkaar te maken hebben.

De digitale computer mag dan nog maar zo'n vijftig jaar oud zijn, hij heeft vele voorvaderen. Dit hoofdstuk geeft een kijkje in de prehistorie van het mechanisch rekenen. Waar je de geschiedenis van de computer laat beginnen is een beetje willekeurig. In 1945 werd aan de universiteit van Pennsylvania in de Verenigde Staten de ENIAC (Electronic Numerical Integrator and Computer) in gebruik genomen: de eerste elektronische digitale reken- en redeneermachine. Maar je zou het begin ook rond 1938 kunnen leggen, toen elektrische schakelingen (relais-schakelingen) de bouwstenen gingen vormen. Of honderd jaar eerder, toen voor het eerst werd geschreven over *programma* en over wat we nu de processor of CPU (Engels: *Central Processing Unit*) noemen.

> Het woord *computer* is afgeleid van het Latijnse *computare* (rekenen of berekenen). In het Engels betekent *to compute* nog steeds *berekenen*. In Nederland spraken de computerpioniers rond 1950 van *rekenautomaten*. Aad van Wijngaarden, de grondlegger van de informatica in ons land, sprak graag van *rekentuig*. Tegenwoordig slaat *rekenmachine* echter eerder op de rekenapparaatjes die je in je zak steekt; in het Engels heten zulke apparaatjes *calculators*.

Bij een computer denken we tegenwoordig aan het van kleurenbeeldscherm, luidsprekers, toetsenbord en joystick voorziene apparaat waar je teksten op kunt verwerken, muziek op kunt componeren, spelletjes op kunt spelen. Het hart van dat multimedia-apparaat wordt echter nog steeds gevormd door een buitengewoon snelle en krachtige machine voor rekenen en redeneren. We zitten er dus niet zover naast als we de geschiedenis van de computer opvatten als geschiedenis van de rekenmachine.

Op verschillende plaatsen in Europa en Nederland zijn mechanische optelmachines uit het begin van de zeventiende eeuw te bewonderen. Heel oude apparaatjes zijn te vinden in Teyler's museum in Haarlem en museum Boerhaave in Leiden. Voor recentere computergeschiedenis kun je terecht bij het Computermuseum van de Universiteit van Amsterdam. Al die museummachines zijn zeker te beschouwen als voorlopers van onze computer. Maar die voorlopers hadden zelf ook weer voorlopers.

1.2 Hulpmiddelen bij het Rekenen

1.2.1 De Abacus

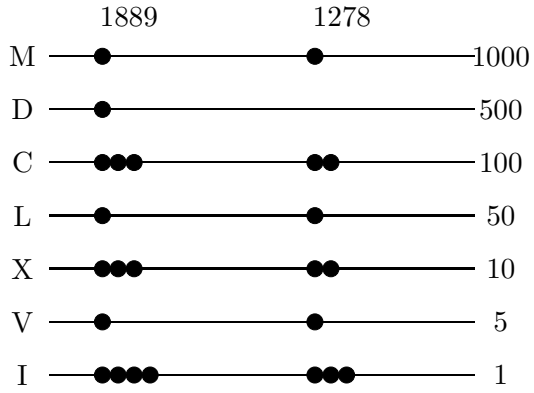
Lang voor het begin van onze jaartelling maakte de mens bij het rekenen al gebruik van hulpmiddelen. De bekendste daarvan is de *abacus* of het *telraam*. In Europa werd de abacus (meervoud: abaci) gebruikt door de Grieken en de Romeinen, maar ook in China was men ermee bekend.

> Primitieve uitvoeringen van de abacus bestonden uit in het zand getekende of in steen gekraste lijnen die een schema vormden. De oorsprong van het woord *abacus* is het Arabische *abq*, dat *stof* of *zand* betekent. Je kon een getal representeren door op bepaalde plaatsen in zo'n schema steentjes te leggen. De schema's werden later ook in stof geweven. Zo'n wollen lap heette in het Latijn *burra*; hieraan is ons woord *bureau* ontleend. Het woord *calculatie* voor *berekening* is ontleend aan het werken met de abacus, want *calculi* betekent letterlijk *(kiezel)steentjes*.

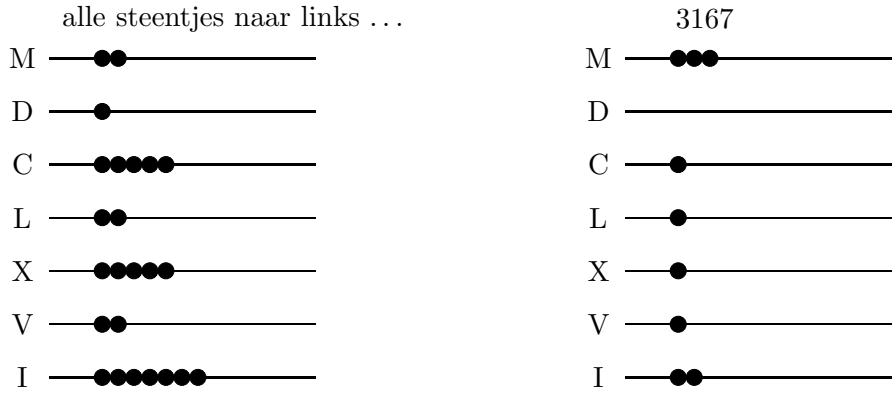
De Romeinse abacus was gebaseerd op het talstelsel dat ze gebruikten. Dat stelsel kende naast symbolen voor eenheden, tientallen, honderdtallen, ... (I, X, C, ...) ook aparte symbolen voor 5, 50, 500, ... (V, L, D, ...). Elk geheel getal groter of gelijk aan 1 kan met behulp van deze symbolen geschreven worden. Het getal 783 bijvoorbeeld wordt geschreven als DCCLXXXIII. In woorden: 1 keer 500 (D) en 2 keer 100 (CC), 1 keer 50 (L) en 3 keer 10 (XXX), en 3 keer 1 (III). In plaats van IIII schreven de Romeinen IV, in plaats van XXXX schreven ze XL, enzovoort. LXXIX staat dus voor 79.

Opdracht 1.3 *De Romeinse getalnotatie ontstond na de bloeiperiode van de klassieke Griekse wetenschap. Zoek uit welke notatie de oude Grieken gebruikten om getallen weer te geven.*

De volgende Figuur geeft representaties op een Romeinse abacus van de getallen 1889 (door de Romeinen genoteerd als MDCCCLXXXIX) en 1278 (in Romeinse notatie: MCCLXXVIII).



Optellen gaat nu in twee stappen: schuif eerst alle steentjes naar links, en doe vervolgens conversie (ook wel: *overdracht*).



De conversiestappen gaan als volgt.

- Haal 5 steentjes op de I-lijn weg, en plaats een extra steentje op de V-lijn. Er blijven 2 steentjes op de I-lijn liggen, en er liggen nu 3 steentjes op de V-lijn.
- Haal 2 steentjes op de V-lijn weg, en plaats een extra steentje op de X-lijn. Er liggen nu 6 steentjes op de X-lijn.
- Haal 5 steentjes op de X-lijn weg, en vervang ze door een extra steentje op de L-lijn. Er liggen nu 3 steentjes op de L-lijn.
- Haal 2 steentjes op de L-lijn weg, en plaats een extra steentje op de C-lijn. Er liggen nu 6 steentjes op de C-lijn.
- Haal 5 steentjes op de C-lijn weg, en vervang ze door een extra steentje op de D-lijn. Er liggen nu 2 steentjes op de D-lijn.
- Haal de 2 steentjes op de D-lijn weg, en vervang ze door een extra steentje op de M-lijn. Klaar.

Het resultaat 3167, of MMMCLXVII, kan nu worden afgelezen.

Later ontstonden verschillende modellen gebaseerd op het verschuiven van knopen, schijven of balletjes op staafjes. In deze vorm kennen wij het telraam hier alleen nog als speelgoed, maar in meerdere landen wordt de abacus tot de dag van vandaag gebruikt, onder andere in winkels en op markten. De 'moderne' vorm is die van de *soroban*, die in de veertiende eeuw in Japan werd ontwikkeld.

Opdracht 1.4 *Op de abacus op Figuur 1.1 zijn twee getallen gerepresenteerd. Welke? (Het schijfje dat niet op een lijn ligt, ligt niet verkeerd.)*

De *soroban*, de oosterse abacus, bestaat uit twee delen, één onderste deel met vier 'aarde' kralen per staaf, en een bovenste deel met op iedere staaf één 'hemel' kraal. Het aantal staafjes kan variëren; meer staafjes maakt het werken met grotere getallen mogelijk. De hemel kralen schuiven van boven naar beneden, de aarde kralen van beneden naar boven. Het rekenen geschiedt op de plaats waar hemel en aarde elkaar raken.
Op een soroban wordt het getal 783 weergegeven als in Figuur 1.2. Bij een uitvoering met staafjes en kralen moet je je voorstellen dat de abacus plat op de grond ligt.

Elk staafje bestaat uit twee delen. Het bovenste, korte stukje heeft betrekking op 5, 50, 500, ... , het onderste deel op 1, 10, 100, ... Het staafje helemaal rechts geeft de eenheden aan, het staafje daarnaast de tientallen, vervolgens de honderdtallen, enzovoort.

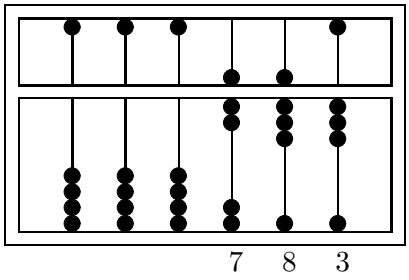
In de situatie van Figuur 1.2 representeert het staafje geheel rechts het getal 3: 0 keer 5 (bovenste steentje niet naar het midden) en 3 keer 1 (drie van de onderste steentjes naar het midden). Op het tweede staafje van rechts staat 80: 1 keer 50 (bovenste steentje naar het midden) en 3 keer 10. Op het derde staafje van rechts 700: 1 keer 500 en 2 keer 100.
Op een soroban gaat bijvoorbeeld de optelling $783 + 54$ als volgt. Stel 783 in op de abacus (zie Figuur 1.2). Tel eerst de eenheden bij elkaar op. Schuif hiertoe eerst het vierde en laatste steentje van het staafje geheel rechts naar het midden. Nu staat op de abacus $783 + 1 = 784$. Schuif dan de vier steentjes weg van het midden en schuif het steentje op het korte deel van het staafje naar het midden: 785. Schuif vervolgens twee van de onderste steentjes naar het midden: $783 + 4 = 787$. Hetzelfde doen we met de tientallen. Om 50 bij 787 op te tellen zouden we een steentje op de positie van de vijftigtallen op het tweede staafje naar het midden willen schuiven. Dat gaat niet, want er is maar één. De oplossing is dat we in feite representeren: $787 - 50 + 100$. Dus: steentje voor de vijftigtallen weg van het midden, en een van de steentjes voor de honderdtallen naar het midden. Hierna staat op de abacus het antwoord 837 (zie Figuur 1.3).

1.2.2 Posities en Overdracht bij Gebruik van het Telraam

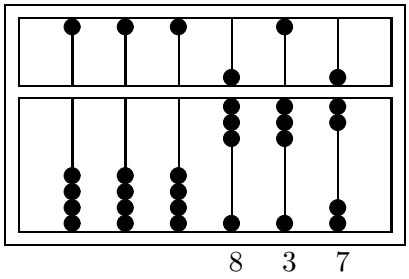
Met behulp van een abacus kon je twee getallen optellen op een manier waarin we 'onze' methode van optellen herkennen. Op school hebben wij geleerd om de twee getallen onder elkaar te zetten en achtereenvolgens van rechts naar links op te tellen: eerst de eenheden, dan de tientallen, enzovoort. Als de som van twee cijfers boven de 9 komt, moeten we '1 onthouden' en bij de volgende kolom optellen. Als we meer dan twee getallen bij elkaar optellen, kan het voorkomen dat we '2 onthouden' of '5 onthouden' of '31 onthouden'. Dat 'onthouden' wordt ook wel *overdracht*, of in het Engels *carry* genoemd. Om twee (grote) getallen op te tellen, hoeven we dus alleen te weten wat de som is van twee cijfers onder de tien en moeten we het principe van

Figuur 1.1: Methodenstrijd: abacus versus Arabische notatie op papier.

Figuur 1.2: Soroban met representatie van het getal 783.

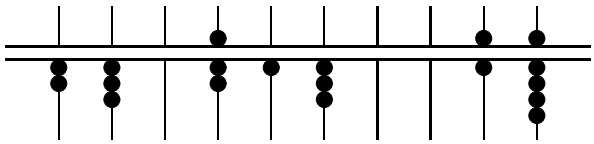


Figuur 1.3: Soroban met representatie van het getal 837.

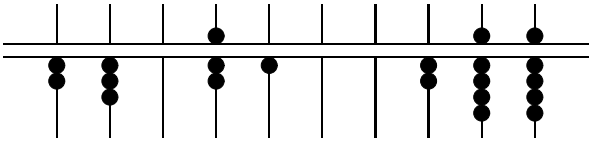


de overdracht naar een volgende kolom kennen. Op het telraam doen we dat door kralen 'over te brengen' naar de volgende staaf.

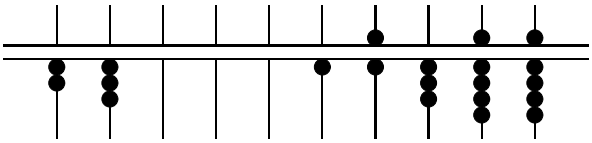
Net zo voor vermenigvuldigen: om 23 met 713 te vermenigvuldigen, vermenigvuldig je eerst 23 met 3, voor de eenheden in 713, met resultaat 69. Op de soroban is dit weergegeven als volgt, met het tussenresultaat 69 geheel rechts.



De drie kralen voor de eenheden in 713 kunnen nu weg. Vervolgens vermenigvuldigen we 23 met de tientallen in 713. Dit geeft 23(0). De 0 tussen haakjes geeft aan dat we met tientallen te maken hebben. Bij het optellen bij 69 slaan we dus de staaf voor de eenheden geheel rechts over.



De kraal voor de tientallen in 713 is nu ook afgewerkt en kan weg. Nu nog 23 met 7 vermenigvuldigen voor de honderdtallen, en resultaat 161(00) optellen bij het getal rechts, rekening houdend met het feit dat het om honderdtallen gaat door te beginnen bij de derde staaf van rechts. Daarmee zijn ook de honderdtallen afgewerkt, en kunnen we het resultaat 16399 rechts aflezen.



Overdracht vindt steeds plaats wanneer je een volgende spijl nodig hebt. Rekening houden met het feit dat je tientallen of honderdtallen berekent doe je door spijlen over te slaan.

In feite reken je op een telraam met positienotatie: je hebt spijlen voor de eenheden, de tientallen, de honderdtallen, enzovoorts. Dit strookt in het geheel niet met de Romeinse getalnotatie die juist niet positioneel is. Elk symbool uit het Romeinse talstelsel heeft een vaste waarde, onafhankelijk van de positie in de Romeinse notatie voor een getal. Zo heeft elke X in LXXIX de waarde 10.

1.2.3 Positionele Getalnotatie

In tegenstelling tot de Romeinen gebruiken wij bij het opschrijven van een getal voor de tientallen geen ander symbool dan voor de eenheden of voor de honderdtallen. Alleen de *positie* bepaalt het verschil in waarde, net zoals op het telraam. Zo staat de eerste 3 in het getal 3438 voor 3000, de tweede voor 30. 25 is verschillend van 205. In 25 staat de 2 voor 2×10, in 205 voor 2×100, en in 152 voor 2×1.

Positionele getalnotatie maakt essentieel gebruik van het symbool voor 0. De Romeinen hadden zo'n symbool niet. De tegenhanger van 0 op het telraam is een lege spijl. Het symbool voor 0 is binnen verschillende culturen op verschillende tijdstippen uitgevonden, net als de positionele getalnotatie. Zo beschikte de Maya cultuur over een representatie voor 0 en over positionele notatie voor getallen. Binnen onze eigen cultuur zijn deze verworvenheden overbracht uit India, via Arabische karavaanroutes: het is een Hindoe-Arabische uitvinding. Vandaar dat we nu nog spreken over $0, 1, 2, \ldots$ als *Arabische cijfers*.

Pas na 1200 begint het positiestelsel met Arabische cijfers, waaronder de 0, in Europa het Romeinse stelsel te verdringen. Rekenen op papier met het positiestelsel heeft als voordeel ten opzichte van rekenen op de abacus dat *tussenresultaten* worden vastgehouden. Als je op de abacus een rekenfout maakt, zit er niets anders op dan de berekening in zijn geheel over te doen. Bij het rekenen op papier kunnen de tussenresultaten worden gecontroleerd. Maar tot in de zestiende eeuw blijft papier in Europa een duur luxeartikel. Simon Stevins tractaat *De*

Thiende (1585) was internationaal een cruciaal geschrift in de verbreiding en acceptatie van de positionele notatie.

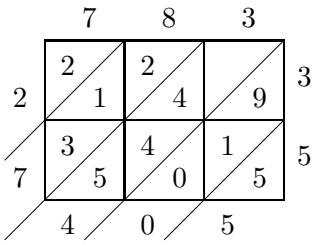
Opdracht 1.5 *In onze maatschappij en in ons taalgebruik zijn nog sporen zichtbaar van andere talstelsels dan het tientallige. Geef daarvan voorbeelden (kijk ook naar andere landen).*

1.2.4 Een Methode van Vermenigvuldigen

Hoe je met een telraam kunt vermenigvuldigen hebben we hierboven gezien. Voor 1500 bestonden er ook verschillende methoden om te vermenigvuldigen met pen en papier, methoden die essentieel gebruik maakten van de positie notatie. Een daarvan was de *gelosia* methode (genoemd naar de diagonale raamspijlen die destijds in Italië in de mode waren). De methode was zeker al in de twaalfde eeuw in India bekend en werd door de Arabieren mee naar Europa gebracht.

Om bijvoorbeeld 783×35 te berekenen werden de getallen 783 en 35 boven en naast een getekend diagram gezet. In elk vakje zette men het product van de bij die kolom en rij horende cijfers. Vervolgens werden de cijfers op de diagonalen opgeteld, te beginnen rechtsonder. Als de som van de diagonaalgetallen groter dan negen is, vindt overdracht plaats naar de volgende diagonaal.

Figuur 1.4: Een methode van vermenigvuldigen: 35×783.



Met behulp van deze methode is het vermenigvuldigen teruggebracht tot opzoeken in de tafels der vermenigvuldiging en kunnen optellen.

Opdracht 1.6 *Maak een plaatje zoals in Figuur 1.4 voor de vermenigvuldiging van* 47 *met* 369.

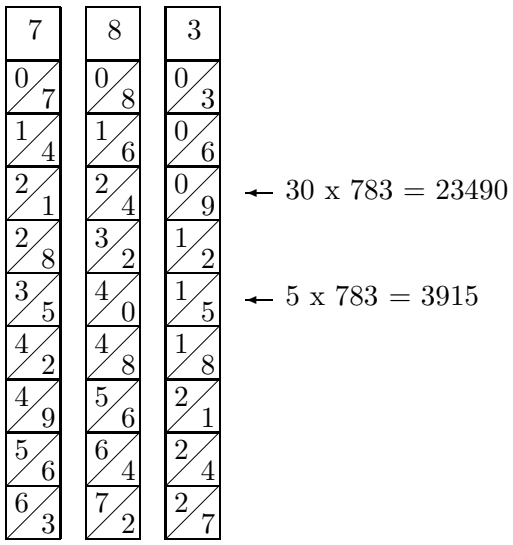
Opdracht 1.7 *Wat hebben bij de gelosia methode de getallen op de diagonaal met elkaar gemeen? (Hint: maak op papier de vermenigvuldiging* 35×783 *op de manier zoals je het op school geleerd hebt.)*

Opdracht 1.8 *Een handig foefje om een ingewikkelde vermenigvuldiging te controleren is de zogenaamde negenproef (Engels: casting out the nines). Om* $35 \times 783 = 27405$ *te controleren, tel je de cijfers van elk van de drie getallen bij elkaar op, en beschouw je hun rest bij delen door negen. Dus:* 35 *geeft* $3 + 5 = 8$, 783 *geeft* $7 + 8 + 3 = 18 - 18 = 0$, *en* $27405 = 18 - 18 = 0$. *De vermenigvuldiging moet dan nog steeds kloppen, tenminste wanneer je weer naar de rest kijkt*

bij delen door negen. Toepassen op het voorbeeld geeft: $8 \times 0 = 0$*, en dat klopt nog steeds. Nog een voorbeeld:* $127 \times 9721 = 1234567$*. De negenproef levert:* $1 \times 1 = 1$*, en ook dat klopt. De negenproef is niet onfeilbaar: een fout in de volgorde van de cijfers komt niet aan het licht. Heb je een idee waar de werking van de negenproef op berust?*

1.2.5 De Vermenigvuldigingsmethode van John Napier

Figuur 1.5: Vermenigvuldigen van 783 en 35 met de methode van Napier.



Een idee van de Schot John Napier (1550–1617) maakte kennis van de tafels overbodig. Napier stelde voor om staafjes te gebruiken met de tafels van vermenigvuldiging erop. Het uiterlijk van deze staafjes was gebaseerd op de gelosia methode. Om 783×35 te berekenen leg je de stokjes voor de tafels van 7, 8 en 3 naast elkaar (zie Figuur 1.5). Op de derde rij kun je nu 783×3 berekenen volgens de gelosia methode. Zet daar een 0 achter en je hebt 783×30. Op de vijfde rij kun je 783×5 berekenen. Tel beide resultaten bij elkaar op en je hebt 783×35. Je zou de staafjes van Napier (de Nepersche staafjes, zoals ze werden genoemd) kunnen zien als een simpel geheugen: je hoeft de tafels immers niet meer zelf te onthouden.

In de website bij dit boek vind je een *applet* met Nepersche staafjes. Manipuleren van tafels was de eerste aanzet tot mechanisering (Schickard), en schoolrekenen is niks anders dan het aanleren van 'tafelmanieren'.

1.2.6 Logaritmen en de Rekenlineaal

Napier was ook (samen met Henry Briggs, 1561–1630) de uitvinder van het rekenen met logaritmen. Een logaritme x van getal c met grondtal a is een waarde voor de exponent x die voldoet aan $a^x = c$. Notatie: $x = {}^a\log c$. Bij voorbeeld: $10^3 = 1000$, dus ${}^{10}\log 1000 = 3$. Omdat

geldt dat $a^x \cdot a^y = a^{x+y}$ kunnen we logaritmen gebruiken om vermenigvuldigen te herleiden tot optellen: ${}^a\log(xy) = {}^a\log x + {}^a\log y$.

Om logaritmen beschikbaar te hebben stelde men tabellen samen, bijvoorbeeld een boek met de waarden van de logaritme van alle natuurlijke getallen tot 10.000 elk tot in 6 of 15 decimalen uitgerekend. Het was een geweldige opgave om de tabellen met weinig rekenfouten en drukfouten gedrukt te krijgen. Een veelgebruikte was de van het werk van Briggs afgeleide en door de gebroeders Vlacq in Gouda gedrukte tafel.

Als illustratie bekijken we ${}^2\log 1$ tot en met ${}^2\log 1024$. Vermenigvuldigen in de bovenste rij correspondeert met optellen in de onderste rij: 8 maal 32 geeft 256, versus 3 + 5 = 8.

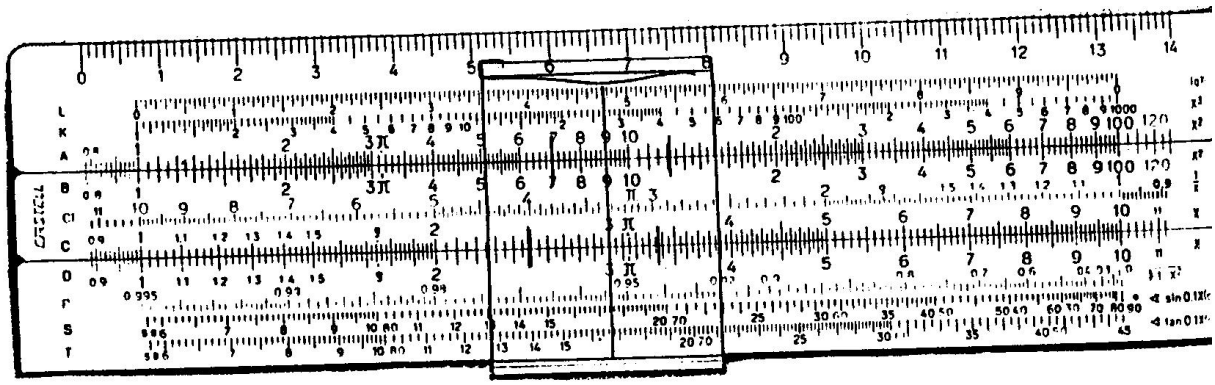
natuurlijke getallen	1	2	4	8	16	32	64	128	256	512	1024
logaritmen (basis 2)	0	1	2	3	4	5	6	7	8	9	10

Nog een voorbeeld, voor basis 10. De logaritmen zijn nu bij benadering, maar zoals je zelf gemakkelijk kunt nagaan zijn de waarden voor $5 \times 6 = 30$ en $5 \times 8 = 40$ nog nauwkeurig genoeg.

1	2	3	4	5	6	8	10	20	30	40	60	100
0	0,301	0,477	0,602	0,699	0,778	0,903	1	1,301	1,477	1,602	1,778	2

Om twee getallen x en y met elkaar te vermenigvuldigen, werden in een tabel ${}^{10}\log x$ en ${}^{10}\log y$ opgezocht. Die waarden werden dan bij elkaar opgeteld en met de uitkomst kon in de tabel dan het resultaat xy worden bepaald. De logaritme leidde rond 1620 tot een nieuw hulpmiddel bij het rekenen, van veel grotere praktische betekenis dan de rekenstaafjes: de rekenliniaal. De rekenliniaal vervangt de logaritmentafel en bestaat uit twee naast elkaar schuivende linialen met een logarithmische schaalverdeling. De handzame rekenliniaal – even groot als een gewone liniaal – bleef tot de komst van de rekenmachine in de jaren zeventig van de twintigste eeuw een veelgebruikt instrument. De uitkomsten waren weliswaar alleen bij benadering juist, want er is een grens aan de precisie van het mechaniek, maar je kon er snel mee uit de voeten. Een rekenlineaal is een voorbeeld van een analoog rekenhulpmiddel: de uitkomsten worden afgelezen van een glijdende schaal.

Figuur 1.6: Rekenlineaal.

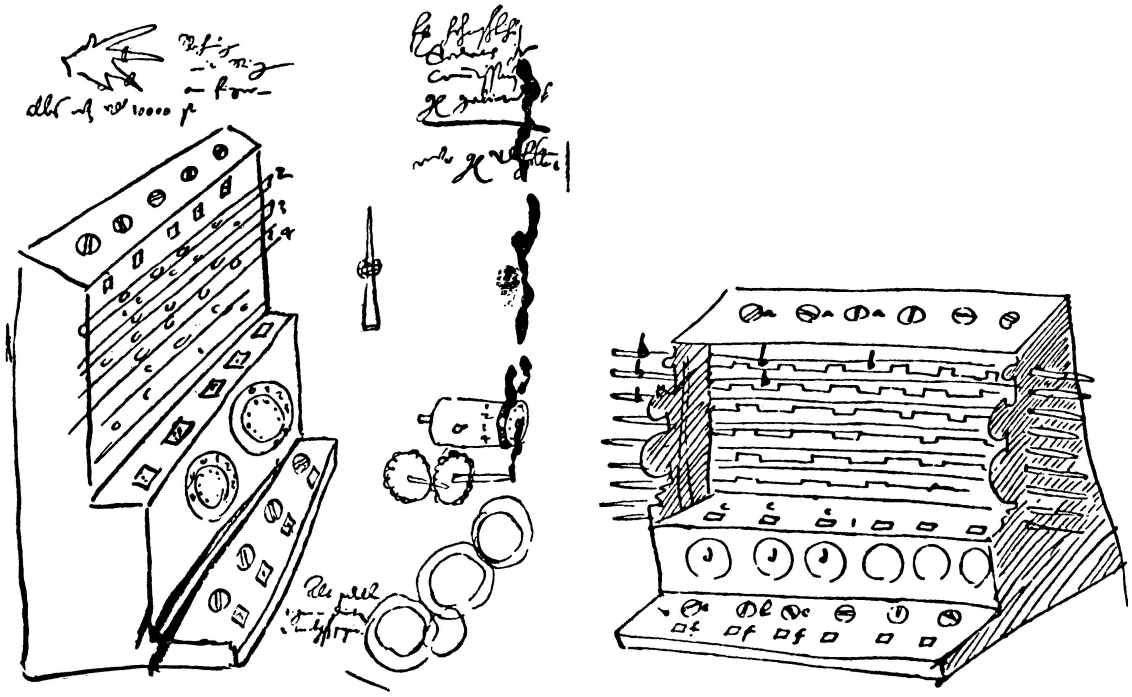


Opdracht 1.9 *Hoe kun je met behulp van een logaritmentafel (of een rekenlineaal) een deling uitvoeren? En kun je er ook mee worteltrekken? Hoe?*

1.3 Antieke Rekenmachines

1.3.1 De Rekenmachine van Wilhelm Schickard

Figuur 1.7: De schets van zijn machine die Schickard toestuurde aan Kepler.



De stokjes van Napier zijn een *mechanisch* hulpmiddel bij het rekenen, maar ze vormen nog geen rekenmachine. Algemeen wordt het apparaat dat werd ontworpen door de Duitse universele geleerde Wilhelm Schickard (1592–1635) als de eerste rekenmachine beschouwd.

> In 1623 schrijft Schickard aan de astronoom Johannes Kepler dat hij erin geslaagd is de staafjes van Napier onder te brengen in een rekenmachine die automatisch kan vermenigvuldigen. 'U zou in lachen uitbarsten als u zou zien hoe [de machine] overdraagt van de ene kolom van tientallen naar de volgende, en hoe hij bij het aftrekken automatisch leent uit de vorige kolom.' Schickard beschreef zijn machine uitvoeriger in een brief uit 1624. Die brief bevat een beschrijving en tekeningen van de rekenmachine, maar Schickard schrijft ook dat een tweede exemplaar dat voor Kepler was bedoeld bij een brand verloren is gegaan. Ook het eerste exemplaar is overigens nooit teruggevonden. Waarschijnlijk was Schickards ontwerp niet bekend bij ontwerpers van rekenmachines die na hem leefden. De brief aan Kepler is pas rond 1950 ontdekt en op grond van de beschrijving en tekeningen is enkele jaren daarna een werkend exemplaar gebouwd.

Het niet-mechanische deel van het apparaat bestond uit draaibare cilinders met op elke cilinder de tafels van 1 tot en met 9 zoals op de stokjes van Napier. De cilinders werden zo gedraaid dat de voor de vermenigvuldiging benodigde tafels zichtbaar waren. Door middel van schuiflatjes werden de niet-benodigde rijen afgedekt. Zie Schickards schets in Figuur 1.7. Het andere deel bestond uit een constructie van tandwielen voor mechanisch optellen. Schickards tekeningen tonen in elkaar grijpende tandwielen. Aan de buitenkant waren zes knoppen zichtbaar waarmee een getal ingevoerd kon worden. Met de meest rechtse knop werden de eenheden ingesteld, met de knop daarnaast de tientallen, enzovoort. Op een cilinder die met een knop verbonden was, stonden rondom de cijfers 0 tot en met 9. Een van die cijfers was zichtbaar door een opening in het apparaat. Het meest vernuftige aan het apparaat was de mechanische overdracht in het geval dat de som van de cijfers groter dan 9 was. In zo'n geval was een knop precies een keer rondgedraaid en de tandwielconstructie zorgde er dan voor dat de knop links ernaast een cijfer verder draaide.

Als je 100722 met 4 wilt vermenigvuldigen op Schickards machine ga je als volgt te werk. Eerst zet je de Napier cilinders in de goede stand voor vermenigvuldigen met 100722. Dit gebeurt door aan de knoppen bovenop het apparaat te draaien tot de cijfers zichtbaar worden in een venster. Vervolgens zorg je dat het gedeelte van de tafel voor vermenigvuldigen met 4 zichtbaar wordt, door de vierde schuif open te zetten. Je ziet nu verschijnen:

0 / 4	0 / 0	0 / 0	2 / 8	0 / 8	0 / 8

Deze uitkomst kan nu worden opgeteld bij het getal in de accumulator (het stelsel van tandwielen voor het optellen) door draaien aan de ronde knoppen onderaan het apparaat (zie weer de schets).

> De tandwielen waarmee Schickard werkte werden al enkele honderden jaren gebruikt in klokken. Bij de bouw van rekenmachines waren dan ook vaak klokkenmakers betrokken. Overdracht gaat in Schickards machine met behulp van tandrad overbrenging. Probleem: 1 bij 99999999 optellen koppelt *acht* tandwielen aan elkaar (overdracht van 1 in 10 naar tweede tandwiel, van 1 in 100 naar derde tandwiel, van 1 in 1000 naar vierde tandwiel, enzovoort. Het mechaniek kon die krachten niet goed aan.

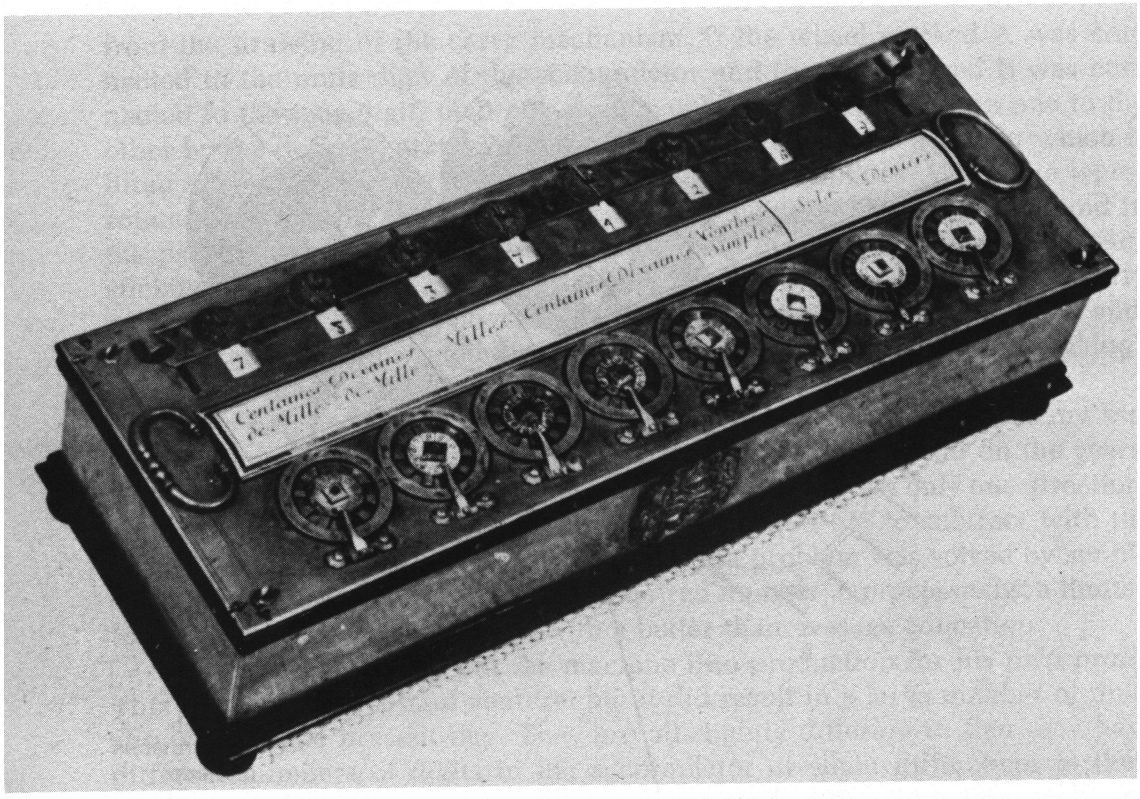
Opdracht 1.10 *Probeer te bedenken hoe je met behulp van tandwielen overdracht kunt regelen. Maak eventueel een schets van de tandwielconstructie.*

1.3.2 De Pascaline

De eerste rekenmachine die bewaard is gebleven is van Blaise Pascal, Frans filosoof, wiskundige en literator (1623–1662). Van het apparaat, de Pascaline, zijn enkele tientallen exemplaren gemaakt, de oudste daterend van 1642. De Pascaline bestond uit een mechanisch gedeelte dat wat bediening betreft overeen kwam met de machine van Schickard. Het mechaniek zat echter wat verfijnder in elkaar.

> Er waren twee versies van de Pascaline. De ene was een algemene optelmachine voor het tientallig stelsel: op elke positie die hoort bij een knop is een van de cijfers 0 tot en met 9 af te lezen. Het andere model was specifiek toegesneden op het Franse muntstelsel uit die tijd. Pascal senior was belastingontvanger, en zijn zoon Blaise heeft het rekenwerk van zijn vader

Figuur 1.8: Pascaline.



willen verlichten. De toenmalige Franse *livre* was 20 *sous* waard en een *sou* 12 *deniers*. Op de meest rechtse positie liep de belasting Pascaline daarom van 0 tot en met 11, en op de positie daarnaast van 0 tot en met 19.

Naast gewone tandwielen gebruikte Pascal tandwielen van een speciale vorm, waarbij de mechanische overdracht geregeld werd doordat een tandwiel na een ronde een palletje bewoog dat op zijn beurt een ander tandwiel een positie verder draaide. Dat werkte beter dan overdracht met gewone tandwielen, zoals bij Schickard. Net als Schickards machine maakte de Pascaline dus gebruik van een soort kilometerteller-constructie. Als je wegrijdt bij kilometerstand $\boxed{5}\boxed{5}\boxed{0}$, en je rijdt 9 kilometer, dan beweegt alleen het laatste schijfje naar de stand 9: $\boxed{5}\boxed{5}\boxed{9}$. Na nog een kilometer rijden gaat het laatste schijfje naar 0 en neemt het voorlaatste schijfje mee naar 6: $\boxed{5}\boxed{6}\boxed{0}$. Je hebt nu 10 bij 550 opgeteld.

Pascal heeft geprobeerd zijn rekenapparaat in productie te brengen om eraan te verdienen. Dat werd geen succes. De apparaten die bewaard zijn gebleven werken geen van alle echt goed, en waarschijnlijk hebben ze wel nooit goed gefunctioneerd: het mechaniek was zo delicaat dat de uiterste zorg in het gebruik geboden was.

De Pascaline had nog een specifiek nadeel dat te maken had met de constructie. Door het palletjes-mechaniek voor de overdracht kon de machine maar één kant op draaien, met als

gevolg dat je er mee kon optellen maar niet aftrekken. Aftrekken kon alleen door gebruik te maken van 9-complementen.

1.3.3 Aftrekken door Optellen van 9-complementen

Stel, je wilt 90817 aftrekken van 3456789, op een machine zoals de Pascaline, die alleen optelt. Dan kun je in plaats daarvan 1 plus het 9-complement van 0090817 optellen bij 3456789, en vervolgens de eerste 1 weglaten (dat wil zeggen, 10000000 aftrekken). Het 9-complement van een getal krijg je door elk cijfer te vervangen door het complement (verschil) van 9. Het 9-complement van 0090817 is niets anders dan het getal $9999999 - 90817$. Het 9-complement van 0090817 is dus 9909182, en 1 daarbij opgeteld geeft 9909183. We hebben nu:

$$\begin{array}{r} 3456789 \\ -90817 \\ \hline 3365972 \end{array} \qquad \begin{array}{r} 3456789 \\ +9909183 \\ \hline 13365972 \end{array}$$

De truc die hierachter zit:

$$3456789 - 90817 = 3456789 + (9999999 - 90817) + 1 - 10000000.$$

De 9-complement notatie stelt ons in staat om gebruik van negatieve getallen geheel te vermijden. Zulke trucs worden nog steeds gebruikt voor het representeren van negatieve getallen in de computer.

De reductie van aftrekken tot optellen gaat natuurlijk met 9-complementen omdat we werken met decimale getallen. 9-complementen worden daarom ook wel decimale complementen genoemd. Computers rekenen binair, en daarom worden voor de representatie van negatieve getallen binaire complementen gebruikt. We komen er in paragraaf 2.3.2 op terug.

Opdracht 1.11 *Geef het 9-complement van* 3456789. *Laat zien hoe je dit 9-complement kunt gebruiken om* $10000000 - 3456789$ *uit te rekenen met alleen optellen.*

1.3.4 De Rekenmachines van Gottfried Leibniz

De volgende stappen in het mechanische rekenen werden gezet door Gottfried Leibniz, ook weer een soort universeel genie (1646–1716). In 1673 demonstreerde hij een machine die kon vermenigvuldigen, gebaseerd op het gegeven dat vermenigvuldigen herhaald optellen is. Maar interessanter is zijn latere idee om het vermenigvuldigen aan te pakken. Hij bedacht een systeem met tandwielen met verschillende diameter en tandwielen met verschillende aantallen tanden.

> Dit werkt als de derailleur van een racefiets. Wanneer een wielrenner de trapas één keer rond trapt gaat zijn achterwiel in het algemeen vaker dan één keer rond. Bij fietsen met 42 tandjes voor en 14 tandjes achter vermenigvuldigt de renner in feite met 3. Vijf volledige slagen van de trappers, bijvoorbeeld, corresponderen met $5 \times 3 = 15$ omwentelingen van het achterwiel.

Opdracht 1.12 *Maak een tabel van de overbrengverhoudingen voor een derailleur met voor twee kettingwielen, met 42 en 38 tandjes, en achter zes kettingwieltjes, met respectievelijk 12, 13, 14, 15, 17, en 20 tandjes. Werk tot op drie decimalen nauwkeurig. (Als je kunt programmeren zou je hier ook een* programma *voor kunnen schrijven.)*

Als de diameter van het voorste tandwiel twee keer zo groot is als de diameter van het achterste tandwiel, zal één keer rondraaien van het voorste tandwiel het achterste twee keer doen rondraaien. Om met Leibniz' machine twee cijfers met elkaar te vermenigvuldigen moesten eerst de juiste tandwielen voorgeschakeld worden, waarna door middel van een draai aan een slinger het mechaniek in beweging werd gezet.

Een ander ontwerp was de getrapte tandwielcilinder. Dit was een cilinder waarop zich naast elkaar tandwielen met verschillende aantallen tanden bevonden. Door de cilinder over zijn lengteas te verschuiven kon een gewenste vermenigvuldiging ingesteld worden. Met de mechanische betrouwbaarheid bleef het nog lang tobben. De rekenmachientjes hadden vooral een functie van sier en verwondering.

1.3.5 Tabellen Maken met de Differentiemethode

Machines als die van Pascal en Leibniz waren technisch niet volmaakt, maar er was geen markt die op verdere ontwikkeling aandrong. Tot het begin van de negentiende eeuw zijn er geen wezenlijke veranderingen te bespeuren op het gebied van de mechanisering van het rekenen. In feite konden wetenschappers, handelaars en anderen die regelmatig moesten rekenen beter uit de voeten met tabellen en rekenlinialen.

Die tabellen gingen op verschillende gebieden (sterrenkunde, zeevaart) een steeds grotere rol spelen. Zowel omvang als betrouwbaarheid maakten dat hier vele uren in gestoken moesten worden. In het begin van de negentiende eeuw startte de Franse regering een project om goniometrische tabellen (tabellen voor belangrijke hoekfuncties zoals *sinus* en *cosinus*) aan te passen aan een nieuwe hoekeenheid die een kwart cirkel verdeelde in 100 graden in plaats van 90. Die nieuwe indeling is trouwens geen blijvertje gebleken.

Bij het maken van goniometrische tabellen werd gebruikgemaakt van de *differentiemethode*, die bepaalde wiskundige berekeningen terugbrengt tot optellen. Om te laten zien hoe de differentiemethode, of methode van verschillen, werkt beschouwen we een voorbeeld, de formule $f(x) = x^2+3x+1$. We zijn geïnteresseerd in een tabel van de waarden $f(0), f(1), f(2), f(3), \ldots$.

Die waarden zijn te berekenen met alleen optellen. Dit gaat als volgt. Beschouw de *eerste verschilfunctie* voor f, dat wil zeggen de functie g gegeven door $g(x) = f(x+1) - f(x)$. Dit kunnen we vereenvoudigen door f uit te schrijven:

$$\begin{aligned} g(x) &= (x+1)^2 + 3(x+1) + 1 - (x^2 + 3x + 1) \\ &= (x^2 + 2x + 1) + (3x + 3) + 1 - (x^2 + 3x + 1) \\ &= 2x + 4. \end{aligned}$$

Het blijkt dat $g(x)$ een slagje eenvoudiger is dan $f(x)$: de term met x^2 is weggevallen. We zeggen: de veelterm $f(x)$ is van de tweede graad, de veelterm $g(x)$ is van de eerste graad.

We kunnen nu ook de tweede verschilfunctie voor f bekijken, dat wil zeggen de verschilfunctie van de eerste verschilfunctie van f, ofwel $g(x+1) - g(x)$. Als we deze functie h noemen, kunnen we $h(x) = g(x+1) - g(x)$ weer vereenvoudigen door g uit te schrijven:

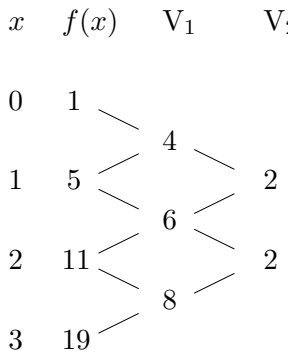
$$h(x) = 2(x+1) + 4 - (2x+4) = (2x+6) - (2x+4) = 2.$$

De tweede verschilfunctie van f is constant. Dit kunnen we nu gebruiken om de tabel voor f uit te rekenen. Immers, omdat $h(x) = g(x+1) - g(x)$, krijgen we dat $g(x+1) = g(x) + h(x) = g(x) + 2$,

want $h(x) = 2$. Dus als we $g(x)$ hebben, krijgen we $g(x+1)$ simpelweg door 2 bij $g(x)$ op te tellen. Omdat $g(x) = f(x+1) - f(x)$, krijgen we dat $f(x+1) = f(x) + g(x)$. Dus, als we $f(x)$ en $g(x)$ hebben, krijgen we $f(x+1)$ door $f(x)$ en $g(x)$ bij elkaar op te tellen. Kortom, we kunnen de tabel voor $f(0), f(1), f(2), f(3), \ldots$ construeren vanuit de beginwaarden $f(0)$ en $g(0)$, met alleen optellen. Dat gaat als volgt.

$$\begin{aligned}
f(0) &= 0^2 + 3 \cdot 0 + 1 = 1 \\
g(0) &= 2 \cdot 0 + 4 = 4 \\
f(1) &= f(0) + g(0) = 1 + 4 = 5 \\
g(1) &= g(0) + h(0) = 4 + 2 = 6 \\
f(2) &= f(1) + g(1) = 5 + 6 = 11 \\
g(2) &= g(1) + h(1) = 6 + 2 = 8 \\
f(3) &= f(2) + g(2) = 11 + 8 = 19
\end{aligned}$$

In de volgende tabelvorm, met functiewaarden, eerste verschillen (V_1) en tweede verschillen (V_2), blijkt duidelijk hoe een tabel van waarden van $f(x)$ gemaakt kan worden uit de beginwaarden 1, 4 en 2, door herhaald optellen.



De waarde in de kolom van de tweede verschillen blijft constant. Het volgende verschil in kolom V_1 is 8+2=10 en dan is $f(4) = 19 + 10 = 29$.

In het algemeen geldt dat voor een n-de graads veelterm (of *polynoom*), dat wil zeggen een functie van de vorm

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

de n-de verschillen constant zijn. Waardentabellen voor polynomen kunnen dus altijd met de differentiemethode worden geconstrueerd.

Opdracht 1.13 *Maak een verschiltabel voor de functie* $f(x) = x^2 - 3x + 7$.

De methode van verschiltabellen kan ook worden gebruikt om het volgende getal in een waardenreeks te raden. Als de waardenreeks begint met $0, 1, 4, 9, 16$, dan zie je meteen dat de volgende waarde 25 moet zijn, omdat je *ziet* dat het gaat om waarden voor de functie $f(x) = x^2$. Maar ook als je dit niet *ziet*, kun je er met de verschilmethode achterkomen wat de volgende waarde is. De verschilwaarden van de reeks $0, 1, 4, 9, 16$ zijn immers $1, 3, 5, 7$, en de verschillen van *die*

reeks zijn $2, 2, 2$. Het tweede verschil is dus constant. De volgende waarde voor de eerste verschilreeks wordt dus $7 + 2 = 9$, en daarmee hebben we de volgende waarde in de oorspronkelijke rij: $16 + 9 = 25$. Dat kan een domme computer ook!

Het leuke is dat die methode ook werkt als je niet meteen ziet wat het volgende getal moet zijn. Neem de reeks $3, 6, 17, 42, 87, \ldots$. Als je meteen ziet wat er in deze reeks volgt op 87 ben je een bolleboos. Als je het niet ziet is er geen man overboord, want dan gaan we gewoon verschiltabellen maken, tot we zien dat de verschillen constant worden. Dit geeft:

$$\begin{array}{ccccccccc} 3 & 6 & 17 & 42 & 87 & \mathbf{158} & \cdots \\ & 3 & 11 & 25 & 45 & \mathbf{71} & \cdots \\ & & 8 & 14 & 20 & \mathbf{26} & \cdots \\ & & & 6 & 6 & \mathbf{6} & \cdots \end{array}$$

Het volgende getal in de reeks is dus 158.

Opdracht 1.14 *Kun je nu ook zeggen wat de definitie was van de reeks?*

1.3.6 De Verschilmachine van Charles Babbage

Charles Babbage (1791–1871) raakte hierdoor geïnspireerd en begon in 1822 aan de bouw van de *Verschilmachine* of *Difference Engine*. De machine, die zijn naam ontleende aan de differentiemethode, moest functies van de vorm $a_6x^6 + a_5x^5 + ... + a_1x + a_0$ (zesdegraads polynomen) kunnen berekenen.

De Verschilmachine was een tandwielmachine, gebaseerd op het optelmechaniek van zijn zeventiende-eeuwse voorgangers. Op draaibare assen werden de bij een gegeven functie horende beginwaarden met de hand ingesteld. Op de eerste as werden de functiewaarden afgelezen, op de tweede de eerste verschillen (V_1), op de derde de tweede verschillen (V_2), enzovoort. Babbages ontwerp had zeven assen en kon dus maximaal zesdegraads polynomen berekenen. Door aan een slinger te draaien werd de waarde op een verschil-as doorgegeven aan de as ernaast. Zo waren op de assen achtereenvolgens alle verschillen en functiewaarden (waar het natuurlijk om ging) af te lezen. In de praktijk kwam het niet verder dan een half afgebouwd model, met drie hoofdassen, maar Babbage had zijn aandacht al verlegd naar een volgend project. In de website bij dit boek vind je een *applet* met Babbage verschiltafels.

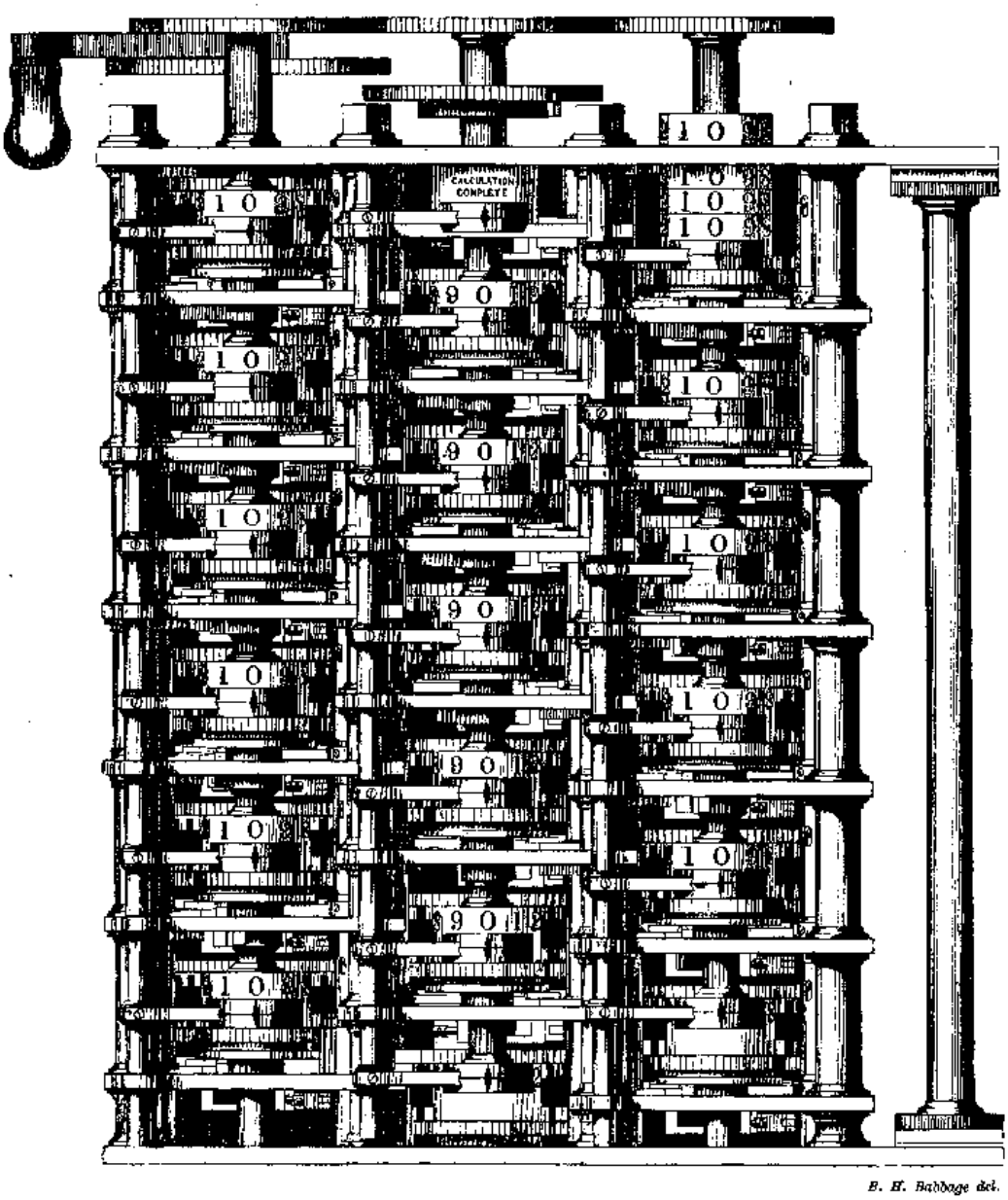
1.4 Van Mechanische Rekenmachine naar Computer

1.4.1 De Analytische Machine van Charles Babbage

Het nieuwe ontwerp van Babbage, de *Analytische Machine* of *Analytical Engine*, kwam niet veel verder dan de tekentafel, maar het is van groot belang, omdat het verscheidene ideeën bevatte die we in de moderne computers terugzien. Zo zou de machine een 'store' (geheugen) en een 'mill' (molen, of rekeneenheid) moeten hebben. De molen zou getallen uit het geheugen moeten kunnen halen en (tussen)resultaten weer in het geheugen zetten.

> Terwijl de Difference Engine bij gegeven beginwaarden een vaste opeenvolging van acties uitvoerde, lag dit bij de Analytical Engine niet vast. De volgorde waarin rekenkundige operaties werden uitgevoerd kon gestuurd worden door middel van ponskaarten. Babbage

Figuur 1.9: Gravure van de Verschilmachine van Babbage.



had dit idee afgekeken van Joseph Marie Jacquard. Deze had aan het eind van de achttiende eeuw een weefgetouw gemaakt waarvan het te weven motief werd bepaald door ponskaarten. Dit waren kaarten met op bepaalde plaatsen gaten, die al of niet een naald doorlieten. Babbage gebruikte de ponskaart niet alleen om de volgorde van rekenoperaties te sturen, maar ook om getallen in te voeren in het geheugen. Hier komt ook het begrip 'programma' in beeld: een serie ponskaarten vormt een combinatie van opdrachten die een gewenste berekening uitvoert.

Babbage was zijn tijd ver vooruit. Het zou tot de jaren dertig van de twintigste eeuw duren voordat zijn ideeën terugkwamen in computerontwerpen. Tot die tijd ontwikkelden de mechanische rekenmachines zich verder. Tandwielconstructies werden technisch degelijker, de invoer van getallen ging door middel van toetsen en de opkomst van elektriciteit maakte het mogelijk om delen van het mechaniek te vervangen door elektrische verbindingen.

1.4.2 Elektrische en Elektronische Schakelingen

Twee uitvindingen op het gebied van de elektriciteit zouden een belangrijke rol gaan spelen bij een volgende stap in de ontwikkeling van de rekenmachine: het relais en de elektronenbuis. Een relais is een elektronische schakelaar die, afhankelijk van het feit of er stroom door loopt, in een van twee mogelijke standen staat. De schakelaar bestaat uit een spoel met een ijzerkern, die magnetisch wordt als er stroom door de spoel loopt. Daardoor wordt een ander stukje metaal aangetrokken, zodat een verbinding gemaakt of verbroken wordt. Doordat er twee mogelijke standen zijn, spreken we van een binaire schakelaar, of simpelweg aan/uit schakelaar. De elektronenbuis is veelzijdiger, maar kan ook als binaire schakelaar gebruikt worden. Een voordeel van de elektronenbuis boven het relais is dat de eerste geen bewegende delen bevat: de schakelaar is puur elektronisch. De elektronenbuis schakelt daarom sneller, wat zeker bij intensief rekenwerk veel tijdwinst opleverde.

De nieuwe schakelaars luidden niet alleen het einde in van de tandwielconstructies, maar ook van het tientallig stelsel waarin de mechanische rekenmachines tot dan rekenden. De tandwielen kunnen gezien worden als 'schakelaars' met tien verschillende standen: bij elke stand hoort een van de cijfers 0 tot en met 9. Het gebruik van relais en elektronenbuis maakte dat het rekenwerk werd verricht in het tweetallig stelsel.

1.4.3 Elektrische en Elektronische Computers

Vanaf 1936 werd op verschillende plaatsen en door verschillende personen geëxperimenteerd met het gebruik van binaire schakelingen, relais en elektronenbuizen in rekenmachines. In 1936 bouwde Konrad Zuse in Duitsland de Z1, een rekenmachine met binaire, mechanische schakelaars. Het ontwerp van de Z1 bestond uit verschillende delen, voor invoer, rekenwerk, programmaopslag, geheugen en uitvoer. Vanwege de vele verbindingen tussen deze onderdelen meende Zuse dat decimale (10-plaatsige) schakelaars niet geschikt waren en hij koos daarom voor binaire schakelaars. Omdat relais te duur waren, gebruikte Zuse mechanische schakelaars. In latere modellen (Z2, Z3, Z4, ...) werden de mechanische schakelaars vervangen door buizen. In Amerika koos John Atanasoff om praktische redenen voor het rekenen met binaire getallen. Hij nam de elektronenbuis als bouwsteen.

De computers die Zuse, Atanasoff en anderen in die jaren bouwden, waren vaak enig in hun soort: er werd van ieder ontwerp maar één exemplaar gebouwd. Het waren ook omvangrijke

machines; de Z1 van Zuse nam een flink deel van zijn huiskamer in beslag en machines met duizenden buizen konden manshoog en meters breed zijn. Ze waren vaak ook ontworpen met het oog op een bepaalde toepassing, bijvoorbeeld het oplossen van stelsels vergelijkingen.

De Tweede Wereldoorlog leidde tot een sterke interesse in snelle computers voor militaire toepassingen. In dat kader werd in Amerika de ENIAC (Electronic Numerical Integrator and Computer) gebouwd. De machine, die bestond uit zo'n 18.000 buizen en 30.000 kilo woog, was klaar in 1945.
Hoewel de ENIAC veel sneller was dan andere computers uit die tijd, was het ontwerp in een aantal opzichten minder revolutionair dan de machines van Zuse en Atanasoff. De machine werkte dan wel met buizen, maar die stonden in dienst van decimaal rekenen. Zo werd bijvoorbeeld het cijfer 7 gerepresenteerd door een rijtje van tien binaire schakelaars waarvan de zevende in één stand stond en de overige in de andere stand.

Opdracht 1.15 *1. Als 10 vacuümbuizen worden gebruikt voor het opslaan van één cijfer, hoeveel buizen zijn dan nodig om getallen van* 0 *tot en met* 999 *op te slaan?*

2. Hoeveel procent van alle mogelijke combinaties met dit aantal buizen representeert daadwerkelijk een getal?

3. Hoeveel buizen zouden nodig zijn als de getallen binair worden opgeslagen, en men getallen van 0 *tot en met* 999 *wil kunnen opslaan?*

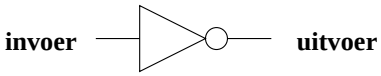
Nog voordat de ENIAC af was, werd al gewerkt aan zijn opvolger. Het ontwerp van de EDVAC (Electronic Discrete VAriable Computer) werd in 1945 beschreven door John von Neumann (1903–1957). Dit rapport wordt gezien als de theoretische basis voor het ontwerp van de moderne computer. Niet alleen werden de voordelen van binair rekenen erkend, maar ook werd voor het eerst voorgesteld het programma zelf in het geheugen van de computer op te slaan (het concept van *stored program*)). Tot dan werd (het uitvoeren van) een programma gestuurd van buitenaf door het *vooraf* instellen van bepaalde elektrische verbindingen. Aan het eind van de jaren veertig werden de eerste computers met programmaopslag gebouwd.

Al deze computers waren rekenmachines, met als functie het berekenen van (soms heel bepaalde) wiskundige formules. Een uitzondering hierop vormde de Colossus, een machine die in de Tweede Wereldoorlog in Engeland werd gebouwd voor het kraken van Duitse, gecodeerde communicatie. De machine was gebouwd met buizen, maar er werden logische bewerkingen mee uitgevoerd.

1.4.4 Transistors en Chips

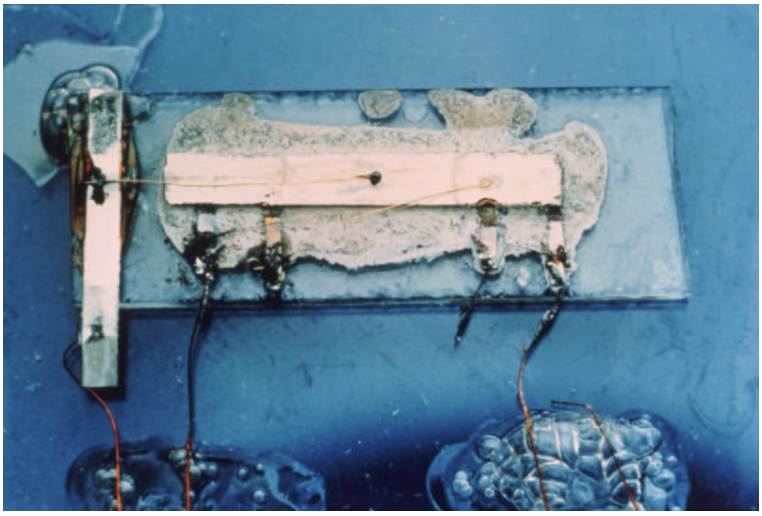
De uitvinding in de jaren veertig van de transistor als vervanger voor de vacuümbuis had een aantal grote voordelen. Een buis was een centimetershoge cilinder, een transistor was iets groter dan de kop van een lucifer. Buizen waren van glas en dus kwetsbaar, een transistor had een hard omhulsel.

Figuur 1.10: Omkeerschakeling: symbolische weergave.



Een transistor is eigenlijk niets anders dan een elektrisch bediende schakelaar. Door een tweetal transistoren te combineren kun je een zogenaamde *omkeerschakeling* realiseren die 'aan' omzet in 'uit' en andersom. Deze omkeerder is de basis van alle elektronische circuits. Zie Figuur 1.10. De omkeerschakeling wordt ook wel een *niet-poort* (Engels: *not-gate*) genoemd. In paragraaf 2.2.2 zullen we nog een paar andere schakelingen zien die met behulp van transistors kunnen worden gerealiseerd, en die samen met de omkeerschakeling worden gebruikt bij binair rekenen.

Figuur 1.11: Eerste IC, in 1958 gebouwd door Jack Kilby.



De volgende technische ontwikkeling kwam in de jaren zestig met de ontwikkeling van het IC (Integrated Circuit). Een IC is een complex van onderling verbonden transistoren, ondergebracht in één minuscuul plaatje. ICs vormen de bouwstenen van de huidige computers. ICs zijn rond 1958 uitgevonden door Jack Kilby (Texas Instruments) en Robert Noyce (Fairchild Electronics). Noyce overleed in 1990; Kilby kreeg voor zijn uitvinding in 2000 de Nobelprijs voor natuurkunde. Zie Figuur 1.11 voor een plaatje van de oer-chip.

1.4.5 Leibniz over Binair Rekenen

Hoewel de keuze voor het gebruik van het tweetallig stelsel in rekenmachines door de beschikbaarheid van relais en elektronenbuis voor de hand lag, was de gedachte erachter niet nieuw. De eerdergenoemde Leibniz schrijft al in 1679 over het binaire stelsel, in een ongepubliceerd manuscript in het Latijn. Naast een binaire representatie van de getallen 1 tot en met 100

beschrijft Leibniz het rekenen (optellen, aftrekken, delen en vermenigvuldigen) met binaire getallen. Later heeft hij het in een brief uit 1698 over de 'mooie orde' van het binaire rekenen. Leibniz was geïnteresseerd in mechanisering van rekenen en redeneren, en hij zag in dat binaire getalrepresentatie zeer geschikt is voor machinaal rekenen.

Leibniz beschrijft zo'n machine in detail. Het apparaat bestaat uit een bus met gaten erin, die geopend en gesloten kunnen worden. Geopend komt overeen met 1, gesloten met 0. Met behulp van kogeltjes, die al of niet door de gaten kunnen, kan door het verschuiven van de bus boven een onderstel met gootjes waarin de kogeltjes terecht kunnen komen, een rekenoperatie uitgevoerd worden. Het apparaat is nooit gebouwd en er is geen tekst van Leibniz bekend waarin hij erop terugkomt. Het zou meer dan 250 jaar duren voordat de eerste binaire rekenmachine gebouwd werd.

Leibniz was een veelzijdig man: jurist, diplomaat, wiskundige, technicus, filosoof. Als filosoof werkte hij aan het ontwikkelen van een universele taal als voertuig voor het menselijk redeneren. In geval van onenigheid over een of ander onderwerp zou deze taal de oplossing moeten geven. Door de strakke logica van zo'n taal zou het werken ermee een soort rekenen worden en Leibniz zelf schreef dan ook dat een ruzie tussen twee mensen opgelost kon worden als ze zouden zeggen: 'Laten we het uitrekenen.'

1.4.6 Binair Rekenen volgens George Boole

Rond 1850 kwam het onderwerp in één klap op een hoger plan dankzij het werk van George Boole. Zijn wiskundige analyse van de 'wetten van het denken' was baanbrekend. Voor beweringen gebruikte hij symbolen, bijvoorbeeld: x kan staan voor 'Het regent.' De waarheidswaarde van zo'n bewering kan 0 of 1 zijn. In het eerste geval is de bewering onwaar, in het tweede geval is zij waar. Boole liet zien hoe je waarheid en onwaarheid van beweringen die zijn samengesteld met de voegwoorden *en*, *of* en *niet* kunt uitrekenen (als je tenminste weet of de kleinste onderdelen, zoals 'Het regent', waar zijn of niet).

De rekenregels die hij daarvoor introduceerde lijken sterk op de regels van het gewone rekenen, met een paar interessante verschillen. Bij het gewone rekenen geldt $-(a \cdot b) = (-a) \cdot b = a \cdot (-b)$, maar bij het Boolese rekenen hebben we: $-(x \cdot y) = (-x + -y)$. Immers, 'Het is niet zo dat het regent of sneeuwt' komt op hetzelfde neer als 'Het regent niet en het sneeuwt niet', en dat is precies wat de wet uitdrukt.

$x + (y + z)$	$=$	$(x + y) + z$	$x \cdot (y \cdot z)$	$=$	$(x \cdot y) \cdot z$
$x + y$	$=$	$y + x$	$x \cdot y$	$=$	$y \cdot x$
$x + x$	$=$	x	$x \cdot x$	$=$	x
$x + (y \cdot z)$	$=$	$(x + y) \cdot (x + z)$	$x \cdot (y + z)$	$=$	$(x \cdot y) + (x \cdot z)$
$x + (x \cdot y)$	$=$	x	$x \cdot (x + y)$	$=$	x
$-(x + y)$	$=$	$-x \cdot -y$	$-(x \cdot y)$	$=$	$-x + -y$
$x + 0$	$=$	x	$x \cdot 0$	$=$	0
$x + 1$	$=$	1	$x \cdot 1$	$=$	x
$x + -x$	$=$	1	$x \cdot -x$	$=$	0
			$--x$	$=$	x

Opdracht 1.16 *Ga na welke Boolese wetten uit het bovenstaande lijstje overeenkomen met 'gewone' rekenregels (dat wil zeggen: welke regels gaan ook op voor het gewone rekenen, waar* $\cdot$

staat voor 'maal', + voor 'plus', en − voor 'negatief getal', of 'vermenigvuldigen met −1') en welke juist niet. In de gevallen waar verschil is: kun je iets zeggen over de 'gewone' rekenregel?

Boole gebruikte · voor 'en', + voor 'of' en − voor 'niet'. 1 staat voor een bewering die altijd waar is, en 0 voor een bewering die altijd onwaar is. Hier is een voorbeeld van een Boolese denkwet. Elke bewering is waar of onwaar, dat wil zeggen: voor elke x geldt: x is waar of zijn negatie is waar. Dit geeft de wet $x + -x = 1$.

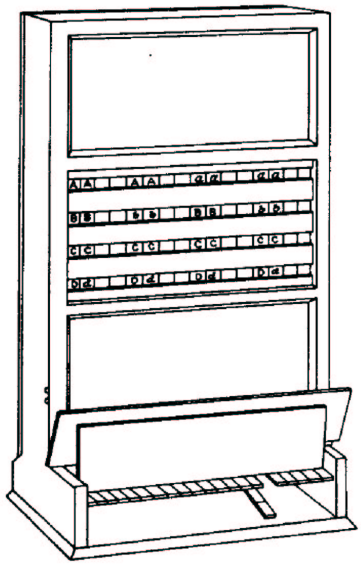
1.4.7 De Logische Piano van William Jevons

Omdat $x, y, \ldots$ alleen de waarde 0 of 1 kunnen aannemen, zijn de wetten van Boole ook te beschouwen als regels voor het rekenen met 0 en 1. Een redenering kon 'vertaald' worden in Booles symbolische systeem en vervolgens kon de juistheid van de redenering 'berekend' worden. In paragraaf 2.4.6 gaan we hier nader op in.

Booles werk maakt dus mechanisering van het redeneren mogelijk. Een van degenen die dit inzagen was William Jevons. In 1869 liet hij een machine bouwen waarmee redeneringen gemaakt en gecontroleerd konden worden. De machine was gebaseerd op de verzamelingenlogica van Boole, met een paar verbeteringen.

We gebruiken hoofdletters om verzamelingen aan te geven en we geven complementen aan met een streep, dat wil zeggen: $\bar{A}$ staat voor de dingen die *niet* in de verzameling A zitten (Jevons zelf gebruikte een iets andere notatie). Stel, A staat voor de verzameling 'voorwerpen van glas' en B voor de verzameling 'breekbare voorwerpen'. Dan staat $\bar{A}$ voor 'voorwerpen niet van glas' en $\bar{B}$ voor 'niet-breekbare voorwerpen'. Er zijn nu vier verschillende combinaties mogelijk.

AB: van glas en breekbaar	$A\bar{B}$: van glas en niet-breekbaar
$\bar{A}B$: niet van glas en breekbaar	$\bar{A}\bar{B}$: niet van glas en niet-breekbaar



Deze vier combinaties waren in de beginstand van de machine af te lezen. Door middel van toetsen zoals op een pianoklavier kunnen aannames worden ingevoerd, bijvoorbeeld 'Alles wat van glas is, is breekbaar'. Het mechaniek in de machine zorgde ervoor dat de combinatie $A\bar{B}$ werd geëlimineerd. Immers, onder de gegeven aanname is de combinatie ' van glas en niet-breekbaar' niet meer mogelijk. Door het invoeren van verdere aannames kunnen nog meer combinaties geëlimineerd worden. Jevons' machine kon logische problemen met maximaal vier verschillende verzamelingen (A, B, C en D) aan. Wegens het uiterlijk van de machine werd deze wel 'logische piano' genoemd.

Opdracht 1.17 *Hoeveel combinaties zijn mogelijk met de verzamelingen A, B en C? Welke combinaties vallen af onder de volgende aannames: (1) Alle A zijn B, en (2) Geen B is C ?*

Rekenen is vereenvoudigen. Rekenen met de Boolese wetten is dus een kwestie van Boolese termen in een aantal stappen in een zo simpel mogelijke vorm brengen. Hier zijn voorbeelden van regels die daartoe kunnen dienen.

$$\begin{array}{lcl} --x & \longrightarrow & x \\ -(x+y) & \longrightarrow & -x \cdot -y \\ -(x \cdot y) & \longrightarrow & -x + -y \\ x \cdot (y+z) & \longrightarrow & (x \cdot y) + (x \cdot z) \\ (y+z) \cdot x & \longrightarrow & (y \cdot x) + (z \cdot x) \\ (x \cdot y) \cdot z & \longrightarrow & x \cdot (y \cdot z) \\ (x+y) + z & \longrightarrow & x + (y+z). \end{array}$$

Hier is een voorbeeld van een Boolese berekening.

$$-(a + (b \cdot c)) \longrightarrow -a \cdot -(b \cdot c) \longrightarrow -a \cdot (-b + -c) \longrightarrow (-a \cdot -b) + (-b \cdot -c).$$

Opdracht 1.18 *Probeer na te gaan in welke zin de uitdrukkingen rechts van de pijlen eenvoudiger zijn dan de uitdrukkingen links.*

Opdracht 1.19 *Je zou de vereenvoudigingsregels nog kunnen uitbreiden met principes voor het vereenvoudigen van uitdrukkingen waarin* 1 *of* 0 *voorkomt. Hoe?*

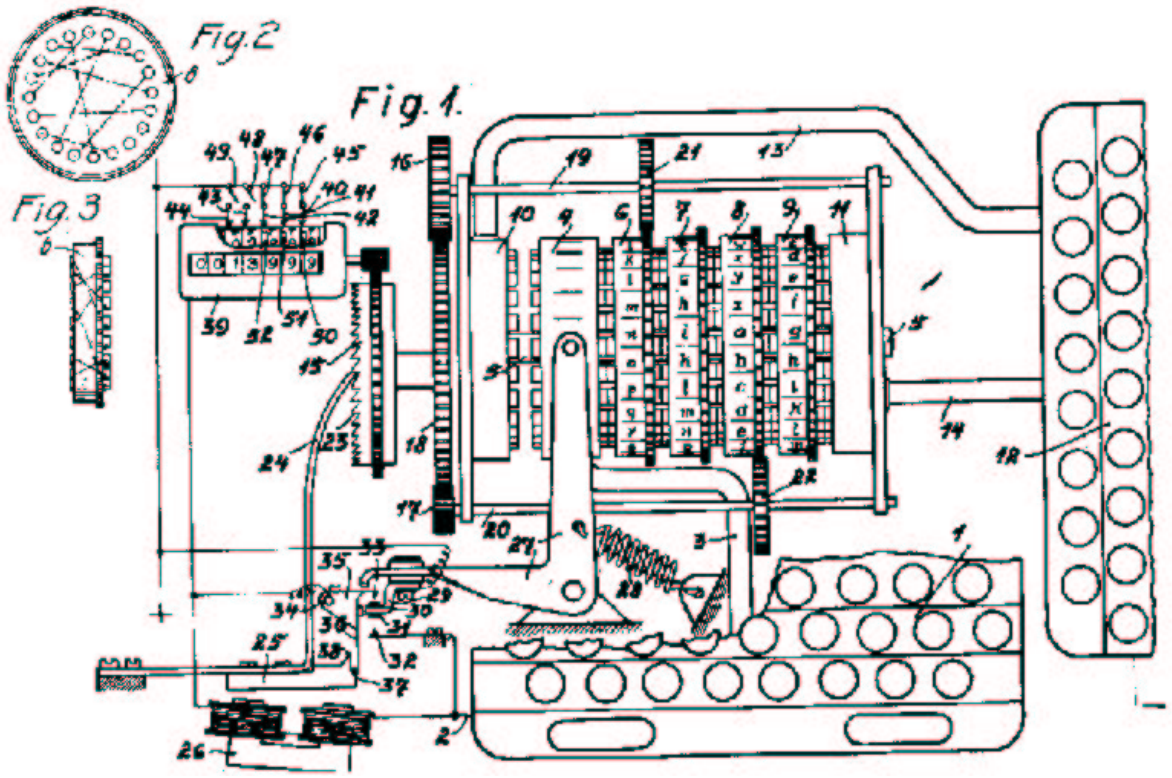
1.4.8 Boolese Algebra en Binaire Circuits

In 1938 toonde Claude Shannon aan dat er een directe relatie bestaat tussen Boolese algebra en (elektrische) circuits van binaire schakelaars. De twee standen van een schakelaar corresponderen met 0 ('open': geen stroom mogelijk) en 1 ('gesloten'), '$\cdot$' correspondeert met een serieschakeling en '$+$' met een parallelschakeling. Een Boolese formule kan zo 'vertaald' worden in een circuit en omgekeerd. De Boolese wet $x + 0 = x$ zegt dan: 'De parallelschakeling van schakelaar x en een altijd open schakelaar is equivalent met de schakelaar x'. Boolese algebra kan dus gebruikt worden bij het ontwerpen van circuits voor computers. Op de website bij dit boek vind je een *applet* met binaire telmachines, met vakjes in plaats van stapels.

1.4.9 Machinaal Kraken van Geheimschrift

In de Tweede Wereldoorlog gebruikten de Duitsers voor het coderen van hun radiomorseboodschappen een machine, de Enigma, die volgens een ingewikkeld patroon via een roterend schijvensysteem letters omzette in andere letters. De Enigma was een keuze-automaat: hoe de codering van een bepaald bericht eruitzag hing af van de keuze van een beginstand van de schijven; er waren eindig veel mogelijke beginstanden. Door logische analyse van berichten waarvan ze zowel een gecodeerde als een niet-gecodeerde versie te pakken hadden gekregen, hadden de Engelsen (geholpen door Poolse wiskundigen) uitgedokterd hoe de Enigma in elkaar zat (dat wil zeggen welke letteromzettingen de schijven afzonderlijk, in elk van hun mogelijke standen teweegbrachten).
Het ontcijferingsprobleem was nu: het vinden van de beginstand die de Enigma machine had gehad bij het coderen van de eerste letter van een bepaald bericht. Als die eenmaal gevonden

Figuur 1.12: Schematische beschrijving van de Enigma machine.



was kon het bericht worden gedecodeerd door het opnieuw door de Enigma machine met dezelfde beginstand te halen. Het omgekeerde ('de inverse') van de coderingsoperatie was dus die coderingsoperatie zelf (net zoals in de rekenkunde de inverse van de operatie 'vermenigvuldigen met -1' diezelfde operatie is: als je een getal n tweemaal met -1 vermenigvuldigt krijg je het oorspronkelijke getal n weer terug). Deze eigenschap van de Enigma code bleek de sleutel tot de ontcijfering.

De Engelse *Government Code and Cypher School*, waaraan onder anderen de wiskundige en AI-filosoof Alan Turing (1912–1954) verbonden was, ging het ontcijferingsprobleem machinaal te lijf, door een batterij machines te bouwen die in snel tempo alle mogelijke beginstanden van de Enigma probeerden, tot de goede stand gevonden was. Deze door Turing ontworpen machines, de zogenaamde 'bombes', waren *simpele automaten*: ze deden in zekere zin altijd hetzelfde. Het gevolg was dat, toen de Duitsers mechanische wijzigingen aanbrachten in hun Enigma machine, de 'bombes' in één klap waardeloos waren geworden en er nieuwe apparaten moesten worden gebouwd. Later ging men het codekraakproces uitvoeren op een machine waarvan het gedrag kon worden aangepast aan informatie over veranderingen van het Enigma mechaniek. Deze machine, de Colossus, kon worden geprogrammeerd: het was geen simpele automaat meer, maar een programmeerbare automaat. Hij had echter een andere architectuur dan de moderne computer: het programma was niet opgeslagen in een intern geheugen, maar het geheugen werd

alleen gebruikt voor het opslaan van gegevens en tussenresultaten van het programma dat werd afgewerkt.

1.5 De Moderne Computer en de Toekomst

1.5.1 Architectuur van de Moderne Computer

De kern van een moderne computer wordt gevormd door de centrale verwerkingseenheid, afgekort CVE (Engels: central processing unit, processor, CPU), plus het interne geheugen (Engels: core memory). Het interne geheugen bestaat uit vakjes die gevuld zijn met nullen en enen. Een geheugenvakje dat plaats biedt aan een enkele 0 of 1 wordt een *bit* genoemd, omdat er een '*bi*nary digi*t*' in kan worden opgeslagen. Zo'n binair cijfer heet zelf trouwens ook een *bit*. Het interne geheugen is onderverdeeld in stukjes, elk bestaande uit een aantal bits die samen een zogenaamde *geheugencel* vormen. Het aantal bits dat in een geheugencel bij elkaar staat verschilt per type computer; het wordt de *adreseenheid* van de computer genoemd. Elke geheugencel heeft een cijferaanduiding, het zogenaamde *adres* van de geheugencel. Iets anders is de *woordlengte* van een computer: dit is het aantal bits dat in één keer door de CVE kan worden verwerkt. De woordlengte van een microcomputer kan bij voorbeeld 32 of 64 zijn.

De cellen van het interne geheugen kunnen worden bereikt, zonder dat eerst andere geheugencellen hoeven te worden gepasseerd. Met andere woorden: alle cellen van het interne geheugen zijn direct toegankelijk (vandaar de benaming 'random access memory' voor het interne geheugen). De toegang tot de geheugencellen wordt geregeld met behulp van de geheugenadressen. Een stukje van het interne geheugen van een computer zou er zo uit kunnen zien:

1	2	3	4	5	6	7	8	9	10	11	10	13	14	15
a	a	p		n	o	o	t		m	i	e	s		

De getallen 1, 2, 3, ... boven de geheugencellen zijn de adressen. Het geheugen is in dit voorbeeld gevuld met een reeks schrijftekens: 'a', 'a', 'p', enzovoort. In feite zijn die schrijftekens zelf weer gecodeerd, bij voorbeeld 01100001 voor 'a', 00100000 voor het spatieteken, enzovoort. Hoe die codering er precies uitziet doet er voor de programmeur (meestal) weinig toe. De centrale verwerkingseenheid voert bewerkingen uit op de inhoud van bepaalde geheugencellen, en plaatst de resultaten van die bewerkingen weer in bepaalde andere geheugencellen, dit alles volgens de specificaties van het programma, dat *zelf* ook in het geheugen is opgeslagen (dat was nu juist de truc van de Von Neumann architectuur).

1.5.2 Betrouwbaarheid en Snelheid

De techniek waarmee mechanisering van rekenen en redeneren wordt gerealiseerd doet er eigenlijk niet toe: tandwielen, vacuümbuizen, transistors of microchips zijn allemaal manieren om processen van 'stapsgewijs vereenvoudigen' vorm te geven. Maar naarmate de processen die we vorm willen geven ingewikkelder worden, gaan betrouwbaarheid en snelheid er meer toe doen. Daarom kwam het mechanisch symboolverwerken dan ook pas in een stroomversnelling toen de techniek geen belemmeringen van betekenis meer opleverde: met de ontwikkeling van transistoren, geïntegreerde elektronische schakelingen, en microchips.

1.5.3 Biocomputers en Kwantumcomputers

De periode na de introductie in de jaren zestig van het IC als de bouwsteen van de computer wordt gekenmerkt door een verdere ontwikkeling van de technische prestaties van chips. Steeds grotere geheugens en rekencapaciteit en steeds hogere rekensnelheden hebben geleid tot de compacte geweldenaars die tot in alle hoeken van de maatschappij zijn doorgedrongen. Deze ontwikkeling zal zich zeker nog door blijven zetten.

Er wordt echter ook onderzoek gedaan naar andere bouwstenen dan chips. Het resultaat daarvan is dan niet een elektronische computer, maar een biocomputer of een kwantumcomputer. Bij een biocomputer kun je denken aan het programmeren met behulp van DNA-oplossingen in een reageerbuis. DNA, de drager van ons erfelijk materiaal, is opgebouwd met behulp van vier verschillende stikstofbasen: Adenine (A), Thymine (T), Guanine (G) en Cytosine (C), waarbij A steeds aan T is gekoppeld, en G steeds aan C. Informatie kan worden gerepresenteerd door DNA-strengen, die enkel en dubbel kunnen voorkomen. Het aan elkaar koppelen van enkele DNA-strengen tot dubbele (het zogenaamde hybridiseren) is een chemisch proces. Daarbij wordt, bij voorbeeld, AACGT gekoppeld aan TTGCA. De mogelijke uitkomsten van een berekening kunnen nu worden gerepresenteerd door enkele DNA-strengen, en de berekende oplossingen door gehybridiseerde strengen. Rekenen vindt hier dus plaats door DNA- 'verstrengeling' in een reageerbuis.

Bij een gewone computer is een bit de kleinste eenheid van informatie. De 1 of 0 van een enkele bit codeert het antwoord 'ja' of 'nee' op een ja/nee vraag. Een kwantumcomputer rekent niet met bits, maar met zogenaamde qubits ('quantum bits'). Qubits kunnen zich *tegelijkertijd* in twee verschillende toestanden 1 en 0 bevinden. Dit berust op een kwantumfysisch effect dat *superpositie van toestanden* wordt genoemd. Als je verschillende kwantumfysici vraagt hoe je je die superpositie precies moet voorstellen krijg je uiteenlopende antwoorden.

> 'Als je alleen wilt voorspellen waartoe kwantumcomputers in staat zullen zijn, dan heb je alleen de vergelijkingen nodig van de kwantummechanica. Maar als je zou willen verklaren hoe ze rekenen, dan moet je begrijpen dat de computer die je kunt zien en aanraken maar een klein facet is van veel groter ding, even echt, hoewel je het bestaan ervan slechts indirect kunt ontdekken, door het rekenwerk dat het voor ons doet. Het heeft de structuur van een heleboel computers van hetzelfde slag als het apparaat dat we zien, die allemaal tegelijk met verschillende berekeningen bezig zijn, waarbij die berekeningen elkaar beïnvloeden door kwantuminterferentie.
>
> In kwantumcomputers komen de effecten van kwantuminterferentie in het groot aan het licht. Maar de theorie houdt in dat de hele werkelijkheid zich zo gedraagt. Het hele universum dat we om ons heen zien is maar een miniscuul facet van een veel groter geheel, het multiversum, dat vele universa zoals het onze bevat die alleen door kwantuminterferentie op elkaar inwerken.'
>
> *Uit een inleiding in kwantum-rekenen op internet.*[1]

Als je hier een beetje duizelig van wordt ben je niet de enige. De meeste kwantumfysici zelf zijn opgehouden zich een voorstelling van de theorie te maken. Toptijdschriften uit de natuurkunde zoals de *Physical Review* zitten ook op deze lijn: het beleid is dat artikelen over de interpretatie van de kwantummechanica eenvoudigweg niet worden geplaatst.

[1] Zie `http://www.qubit.org`.

Hoe dan ook, volgens de kwantum mechanica hebben deeltjes op subatomair niveau de bijzondere eigenschap dat ze zich op hetzelfde moment in twee verschillende toestanden ('fasen') kunnen bevinden. Een atoomkern kan bijvoorbeeld tegelijkertijd twee verschillende kanten opdraaien, tenminste, zolang we niet proberen de fase te *observeren*. Waar een bit in één van twee standen kan staan (0 of 1), kan een qubit zich ook in beide standen tegelijk bevinden, tenminste, zolang we niet kijken. Zodra we kijken aggregeert de kwantumtoestand tot een gewone toestand. Een en ander betekent dat een qubit tegelijkertijd aan twee verschillende berekeningen kan meedoen, met alle gevolgen vandien voor de rekensnelheid. Je kunt een kwantumberekening niet volgen terwijl die aan de gang is. Maar het resultaat kan worden gezien als het resultaat van vele conventionele berekeningen tegelijk: de toestand van een qubit hangt immers af van *alle denkbare vorige toestanden van die qubit*.

Het onderzoek op het gebied van biocomputer en kwantumcomputer is nog zeer theoretisch, en loopt, net als in de tijd van dat Alan Turing speculeerde over het begrip 'mechanische berekenbaarheid', ver op de praktijk vooruit. Zo zijn er al programma's bedacht die, *als we een kwantumcomputer zouden hebben*, zeer efficiënt een getal van honderden cijfers in factoren kunnen ontbinden.

Hoofdstuk 2

Rekenen en Redeneren

2.1 Unair Rekenen

Met de tegenstelling niets/iets kun je al rekenen, bij voorbeeld door af te turven (achter elkaar zetten van streepjes):

$$\text{niets}, |, ||, |||, ||||, \ldots \ .$$

Zo turf je de score bij een spelletje biljart. Bij unair rekenen tel je op door middel van turven, en je trekt af door middel van uitgummen.

Vermenigvuldigen gaat door middel van kopiëren. Als je ||||| moet vermenigvuldigen met |||| vervang je elk streepje in het rijtje ||||| door een kopie van het rijtje ||||. Het resultaat: ||||||||||||||||||||.

Delen gebeurt door 'verdelen in blokjes, en turven hoe vaak je een blokje weggooit'. Voorbeeld: delen van ||||||||||||||||||||||| door ||||| geeft eerst verdelen van ||||||||||||||||||||||| in blokjes van |||||:

Eerste blokje (rijtje) weggooien geeft: | maal ||||| plus

Tweede rijtje weggooien geeft: || maal ||||| plus

Zo verder tot en met vierde rijtje weggooien, en we krijgen: |||| maal ||||| plus |||. De uitkomst is dus: |||| met rest |||. Voor mensen zijn de unaire representaties lastig te lezen, maar voor een

rekenende machine is unaire representatie uiteraard geen probleem. Wat unair rekenen onhandig maakt is dat de representaties voor de getallen erg lang worden. Je kunt dit vergelijken met een kaart maken die even groot is als het gebied dat je in kaart wilt brengen. Een grote sprong voorwaarts in het representeren van getallen werd gemaakt door het representeren van 'niets' met behulp van een symbool 0.

2.2 Binair Rekenen met Natuurlijke Getallen

2.2.1 Binaire Getalrepresentatie

Figuur 2.1: Decimale versus Binaire Representatie.

decimaal		binair
0	=	0
1	=	1
2	=	10
3	=	11
4	=	100
5	=	101
6	=	110
7	=	111
8	=	1000
9	=	1001
10	=	1010
11	=	1011
12	=	1100
13	=	1101
14	=	1110
15	=	1111
16	=	10000

Binair rekenen is gebaseerd op twee dingen: (1) representatie van het verschil tussen iets en niets, en (2) een methode om informatie beetje bij beetje (bitje voor bitje) over te dragen.

Punt (1) houdt in dat we werken met twee symbolen, 0 en 1, waarbij *niets* gerepresenteerd wordt als 0 en *iets* (een enkel ding) als 1. Dit punt kan aanleiding geven tot fraaie diepzinnigheden, zoals deze woorden van Leibniz: 'God, gerepresenteerd door het getal 1, schiep de wereld uit niets, gerepresenteerd door 0.' Punt (2) is iets minder diep, maar daarom niet minder belangrijk: het is niets meer of minder dan het principe van de positionele getalnotatie. De volgorde waarin nullen en enen worden opgesomd krijgt betekenis. 01 betekent iets heel anders dan 10. Bij unaire representatie speelt volgorde van streepjes zetten uiteraard geen rol. Het maakt bij het turven niet uit of je je volgende streepje vooraan of achteraan het rijtje zet. Alle streepjes zien er immers hetzelfde uit. In binaire representatie wordt 10 geschreven als 1010,

immers:

$$10_{10} = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 1010_2.$$

Figuur 2.1 geeft een nuttige tabel voor het omzetten van decimale naar binaire representatie. Hier is een procedure die je kunt volgen voor het omzetten van decimale naar binaire representatie.

> Kijk of het getal even is. Zo ja, schrijf dan een 0, zo nee trek 1 af van het getal en schrijf een 1. Deel vervolgens het (nieuwe) getal door 2 en herhaal de procedure, totdat je bij het getal 1 bent uitgekomen.

De cijfers die je hebt geschreven vormen *in omgekeerde volgorde* de binaire representatie van het getal waarmee je bent begonnen.

Hier is de procedure in actie, voor het voorbeeldgetal 435. We schrijven steeds het nieuwe verkregen binaire cijfer *voor* de binaire reeks die we al hadden.

435	1
217	11
108	011
54	0011
27	10011
13	110011
6	0110011
3	10110011
1	110110011

Het eindresultaat is 110110011, en inderdaad,

$$2^8 + 2^7 + 2^5 + 2^4 + 2^1 + 2^0 = 256 + 128 + 32 + 16 + 2 + 1 = 435.$$

Opdracht 2.1 *Geef een binaire representatie van het getal 2000 (decimaal).*

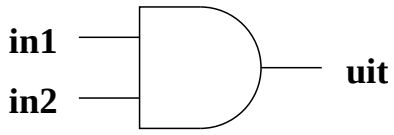
Wie opdracht 2.1 heeft uitgevoerd ziet dat er in het binaire systeem niets bijzonders is aan het getal 2000. De vrees voor de millenniumbug zat hem dan ook in iets heel anders. In de begintijd van het programmeren was geheugenruimte schaars, en bij de representatie van het jaartal 1970 werd daarom alleen 70 gerepresenteerd. Daarvoor was één byte voldoende: daarin kun je immers binaire representaties van de getallen 0 tot en met $2^8 - 1 = 127$ kwijt. Of bij representatie van één decimaal cijferteken per byte: je hebt nu een byte nodig voor de 7 en een byte voor de 0. Nog altijd een besparing van twee bytes. Onder deze geheugenbesparende representaties komt 2000 er hetzelfde uit te zien als 1900. In 1970 leek het jaar 2000 ver weg.

Opdracht 2.2 *Binair valt er pas iets te vieren in het jaar 2048. Waarom?*

Opdracht 2.3 *Een paar vragen om gevoel te krijgen voor het verschil in efficiëntie tussen de representaties.*

1. *Hoeveel tekens heb je nodig om je geboortejaar unair te schrijven? En in de binaire notatie? En decimaal?*

Figuur 2.2: Symbolische weergave van de EN-poort.



2. *Hoeveel tekens heb je nodig voor de unaire, binaire en decimale representatie van een natuurlijk getal n?*

Opdracht 2.4 *Een paar vragen om te oefenen met verschillende talstelsels.*

1. *Schrijf 435 in het drietallig stelsel.*

2. *In het zogenaamde hexadecimale stelsel (het zestientallige stelsel) hebben we naast* $0, 1, 2, \ldots, 9$ *nog vijf extra symbolen nodig. Het is gebruikelijk om hiervoor* $A, B, \ldots, F$ *te nemen. Welk (decimale) getal is* $4B7F$ *dan?*

3. *En wat is de hexadecimale representatie van* 435*?*

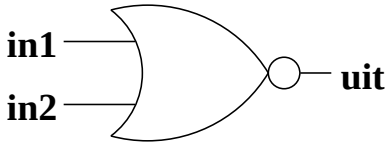
2.2.2 Circuits en Binair Rekenen

Een circuit is een elektronische schakeling met *ingangen* en *uitgangen*. De ingangen en uitgangen zijn draden die elektriciteit geleiden: bij de ingangen wordt elektrische spanning aangebracht, bij de uitgangen wordt elektrische spanning gemeten. De werking van een circuit wordt bepaald door wat er gebeurt als we een aantal van de ingangen onder spanning zetten. Welke uitgangen komen dan onder spanning te staan? Een eerste voorbeeld van een elektronische schakeling hebben we in paragraaf 1.4.4 gezien: de omkeerschakeling (Figuur 1.10).
Met behulp van transistoren kunnen andere schakelingen worden gebouwd. Figuur 2.2 geeft een symbolische weergave van de transitorschakeling die het logische gedrag van 'en' oplevert. Deze zogenaamde EN-poort heeft twee ingangen en een uitgang. De werking wordt bepaald door de volgende tabel:

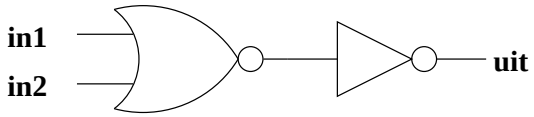
hoog	EN	hoog	=	hoog
hoog	EN	laag	=	laag
laag	EN	hoog	=	laag
laag	EN	laag	=	laag.

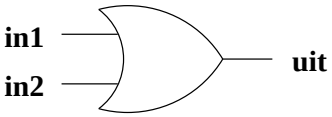
Hierbij staat 'hoog' voor 'hoog voltage' (meestal zo'n 5 volt) en 'laag' voor 'laag voltage' (bijvoorbeeld minder dan 2 volt). Een andere eenvoudige schakeling is de NOF-poort. Zie Figuur 2.3. Een NOF-poort kan worden gecombineerd met een omkeerschakeling tot een OF-poort. Zie Figuur 2.4. Hieronder zullen we zien hoe we dit soort schakelingen kunnen gebruiken voor rekenen en redeneren.

Figuur 2.3: Symbolische weergave van de NOF-poort.



Figuur 2.4: Implementatie van 'of' met een NOF-poort en een omkeerschakeling. Daarnaast de symbolische weergave van de OF-poort.





2.2.3 Circuits voor Optellen

Opdracht 2.5 *Bereken* $1011 + 10011$ *(binair) door de getallen onder elkaar te zetten en op te tellen.*

De tafel voor optellen van binaire cijfers ziet er als volgt uit.

$$\begin{aligned} 1+1 &= 10 \\ 1+0 &= 01 \\ 0+1 &= 01 \\ 0+0 &= 00 \end{aligned}$$

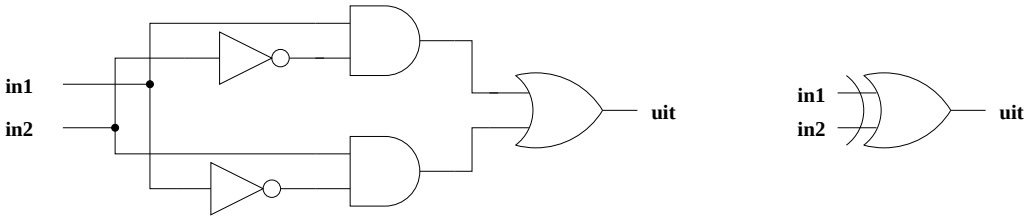
Het eerste cijfer van de uitkomst noemen we de 'overdracht' of 'carry' (de informatie die wordt overgedragen naar de volgende kolom), het tweede de eigenlijke som. (Als je grotere binaire getallen gaat optellen tel je de kolommen van rechts naar links op, en tel je steeds de overdracht op bij de volgende kolom.)

Je kunt hier op twee manieren naar kijken: je kunt de binaire 1 en 0 opvatten als getallen, maar je kunt ze ook opvatten als *waarheidswaarden*: 1 staat dan voor *waar*, en 0 voor *onwaar*. Deze tweede opvatting legt een direct verband tussen binair rekenen en de logica van beweringen, de zogenaamde propositielogica.

We zien dat de waarde in de somkolom wordt gegeven door de *exclusieve of* van de twee getallen (nu opgevat als waarheidswaarden). De uitvoer is *waar* precies dan als precies een van de twee invoeren *waar* is. Hier is een tabel voor de *exclusieve of* van twee waarheidswaarden, waarbij we $\oplus$ gebruiken voor 'of het een, of het ander, maar niet allebei'.

waar	$\oplus$	waar	=	onwaar
waar	$\oplus$	onwaar	=	waar
onwaar	$\oplus$	waar	=	waar
onwaar	$\oplus$	onwaar	=	onwaar

Figuur 2.5: Implementatie van exclusief 'of' met behulp van twee EN-poorten, twee omkeerders, en een OF-poort. Daarnaast de symbolische weergave van de XOF-poort.



Net zo kunnen we de uitkomst in de overdrachtskolom opvatten als het resultaat van een logische bewerking, namelijk *en*, oftewel *conjunctie*. We voeren weer een symbool in; $\wedge$ staat voor *en*. De tabel voor $\wedge$ ziet er als volgt uit.

waar	$\wedge$	waar	=	waar
waar	$\wedge$	onwaar	=	onwaar
onwaar	$\wedge$	waar	=	onwaar
onwaar	$\wedge$	onwaar	=	onwaar

Als we nu de twee binaire cijfers A_1 en A_2 noemen, en $\wedge$ gebruiken voor 'en', en $\oplus$ voor 'exclusief of', hebben we dus:

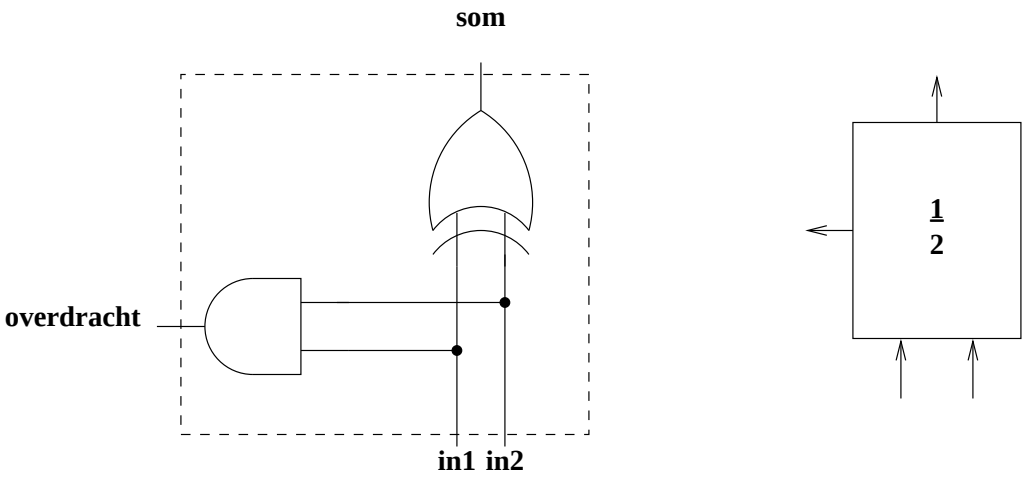
- de overdacht van A_1 en A_2 wordt gegeven door $A_1 \wedge A_2$,
- de eigenlijke som van A_1 en A_2 wordt gegeven door $A_1 \oplus A_2$.

De combinatie van een EN- en een XOF-poort (EN voor de overdracht, XOF voor de som) geeft ons een schakeling om binair op te tellen, tenminste, in de eerste kolom. Zo'n schakeling heet een halve optelschakeling (Engels: *half adder*). Een XOF-poort hadden we nog niet, maar zo'n poort kan worden samengesteld met de componenten die we al hebben. Zie Figuur 2.5 voor de implementatie van de XOF-poort. Het schema voor de halve optelschakeling wordt gegeven in Figuur 2.6. Hier is de tabel voor de halve optelschakeling.

invoer		uitvoer	
A_1	A_2	overdracht-uit	som
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

Om de volgende kolommen van twee binaire getallen op te kunnen tellen moeten we ook de overdracht van de vorige kolom kunnen verwerken. Een circuit hiervoor heeft een ingang voor het eerste binaire cijfer (voltage hoog = 1, voltage laag = 0), een ingang voor het tweede binaire cijfer, en een ingang voor de overdracht uit de vorige kolom. De volledige tabel ziet er als volgt uit.

Figuur 2.6: Implementatie van een halve optelschakeling met behulp van een EN-poort en een XOF-poort. Daarnaast een schematische weergave van de halve opteller.

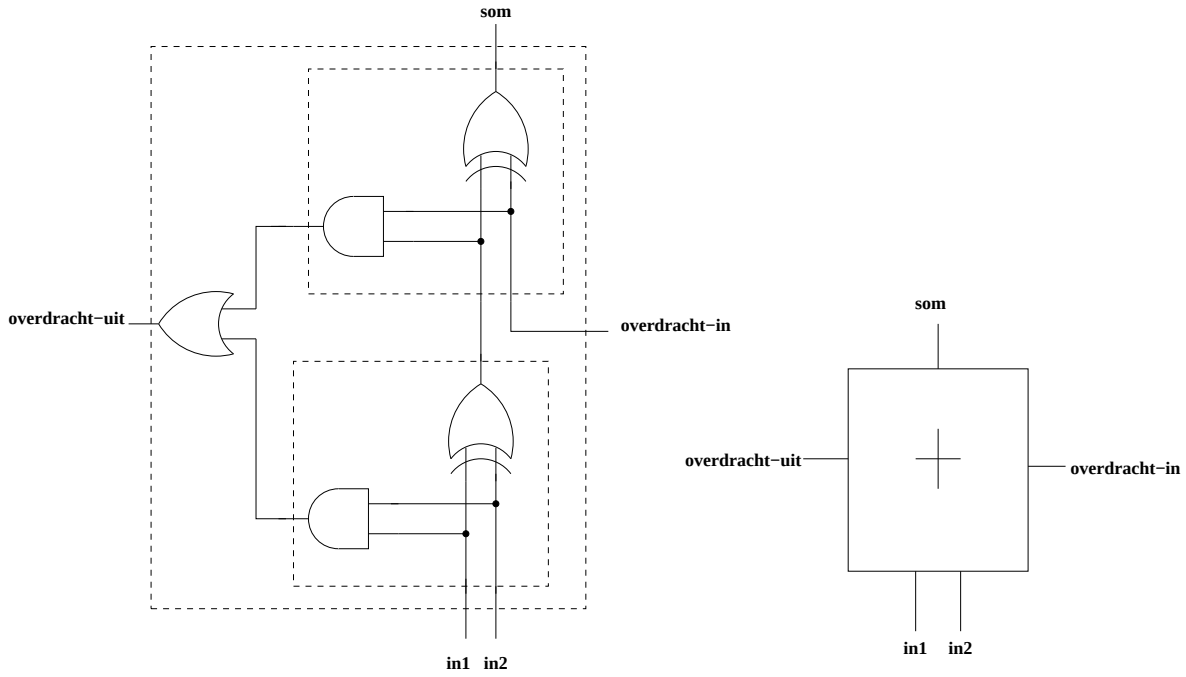


invoer			uitvoer	
A_1	A_2	overdracht-in	overdracht-uit	som
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

We zien dat de som van A_1 and A_2, gegeven overdracht C, wordt gegeven door $A_1 \oplus A_2 \oplus C$, waarbij $\oplus$ staat voor exclusief of. Merk op dat het in dit geval niet uitmaakt of we dit berekenen als $(A_1 \oplus A_2) \oplus C$, dat wil zeggen, eerst het exclusieve of van A_1 en A_2, en vervolgens het exclusieve of van het resultaat met C, of als $A_1 \oplus (A_2 \oplus C)$, dat wil zeggen, door eerst de exclusieve of van A_2 en C te nemen, en vervolgens het exclusieve of te nemen van het resultaat met A_1. Dit berust op een logische eigenschap van $\oplus$. Het resultaat van $A_1 \oplus A_2 \oplus A_2$ wordt wel de *pariteitsfunctie* genoemd: de pariteit van een aantal invoer bits is 1 als het aantal 1-en in de invoer *oneven* is, anders 0. Dit perspectief maakt meteen duidelijk waarom de volgorde van combineren er niet toe doet.

De overdracht-uit van de optelling wordt gegeven door $(A_1 \wedge A_2) \vee ((A_1 \oplus A_2) \wedge C)$. Hier staat $\vee$ voor het *inclusieve* of. De tabel voor inclusief of is als volgt.

Figuur 2.7: Implementatie van een volle optelschakeling met behulp van twee halve optellers. Daarnaast een schematische weergave van de volle opteller.



waar	$\vee$	waar	$=$	waar
waar	$\vee$	onwaar	$=$	waar
onwaar	$\vee$	waar	$=$	waar
onwaar	$\vee$	onwaar	$=$	onwaar

De functie voor de overdracht-uit wordt ook wel de *meerderheidsfunctie* genoemd. De meerderheidsfunctie voor n argument-bits geeft 1 als meer van $\frac{n}{2}$ van de argumenten de waarde 1 hebben, anders 0.

De schakeling die het bovenstaande bewerkstelligt (en het kan op meerdere manieren, want er zijn meerdere propositielogische formules die de bovenstaande waarheidstafel hebben) heet een *volle optelschakeling* (*full adder*).

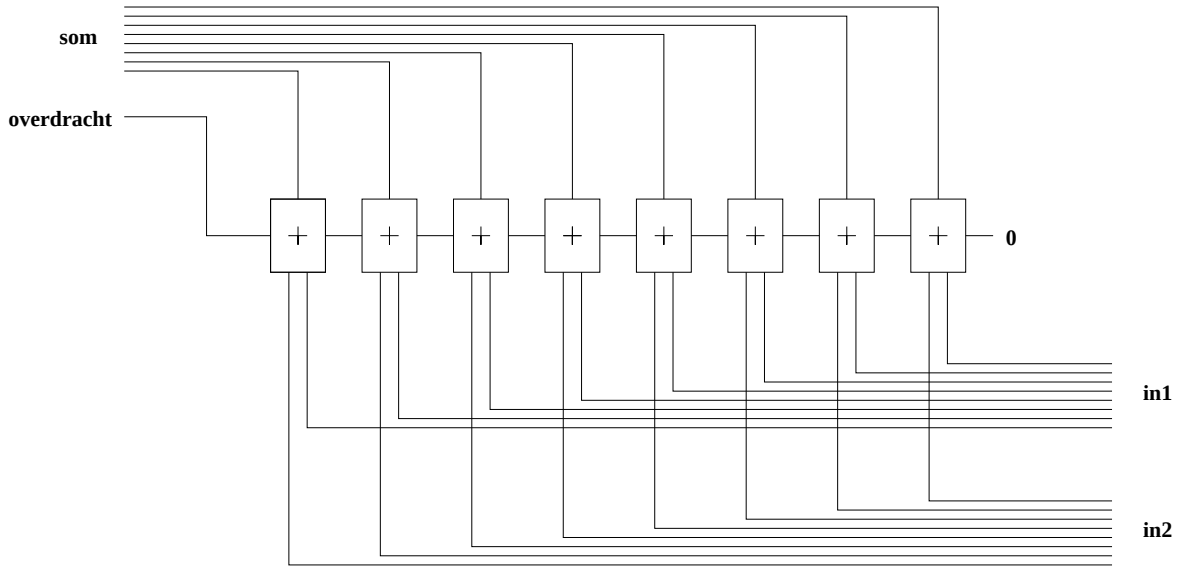
Opdracht 2.6 *Maak een tabel van waarheidswaarden voor* $A_1 \, noch \, A_2$.

Opdracht 2.7 *Nog weer een andere schakeling is die voor NEN.*

1. *Maak een waarheidstabel voor NEN.*
2. *Hoe denk je dat het symbool voor een NEN-poort eruitziet?*

Om in één keer twee bytes (twee binaire getallen van acht cijfers) bij elkaar op te kunnen tellen zetten we een halve optelschakeling en zeven volle optelschakelingen naast elkaar. Alternatief:

Figuur 2.8: Implementatie van een schakeling voor het bij elkaar optellen van twee bytes.



gebruik acht volle optelschakelingen, en zorg dat er een 0 signaal gaat naar de overdracht-in poort van de eerste: zie Figuur 2.8.

De overdracht-uit van de eerste opteller is de overdracht-in van de tweede, en zo verder, en de overdracht-uit van het geheel is de overdracht-uit van de laatste opteller. Als je meteen twee of vier bytes tegelijk wilt verwerken: gebruik 16 of 32 optellers, en klaar is Kees. Als je weet dat soort schakelingen meteen in het silicium wordt gebakken, zie je dat propositielogica alles te maken heeft met microchip ontwerp.

De optelschakeling die we hierboven hebben uiteengezet is niet de meest efficiënte schakeling voor optellen. Bij het berekenen van de overdracht stroomt de informatie van links naar rechts door het circuit, en voor, bij het optellen van twee getallen $a_7a_6a_5a_4a_3a_2a_1a_0$ en $b_7b_6b_5b_4b_3b_2b_1b_0$, de optelling van bits a_i en b_i ter hand kan worden genomen, moet worden gewacht op het resultaat van de overdracht van de optelling van a_{i-1} en b_{i-1}. Met wat ingewikkelder schakelingen valt dit nog wel iets te verbeteren.

2.2.4 Schakelingen voor Vermenigvuldigen

Vermenigvuldigen van twee niet-negatieve gehele getallen x en y kan worden gereduceerd tot herhaald optellen. Immers,

$$x \times y = \underbrace{x + \cdots + x}_{y \text{ maal}}.$$

Een en ander betekent dat we voor vermenigvuldiging strikt genomen geen aparte schakeling nodig hebben. Aparte schakelingen ontwerpen en implementeren voor operaties zoals vermenigvuldigen maakt het rekenen echter wel veel efficiënter.

Opdracht 2.8 *Bereken* 101001 : 11 *met behulp van een binaire staartdeling.*

De binaire tafel van vermenigvuldiging is buitengewoon simpel. De tafel laat zien dat binair vermenigvuldigen kan gebeuren met een EN-poort.

$$\begin{aligned} 1 \times 1 &= 1 \\ 1 \times 0 &= 0 \\ 0 \times 1 &= 0 \\ 0 \times 0 &= 0. \end{aligned}$$

Hier is een voorbeeld van binair vermenigvuldigen met de 'middelbare school' methode:

				1	1	0	1
				1	0	1	1
				1	1	0	1
			1	1	0	1	
		0	0	0	0		
	1	1	0	1			
1	0	0	0	1	1	1	1

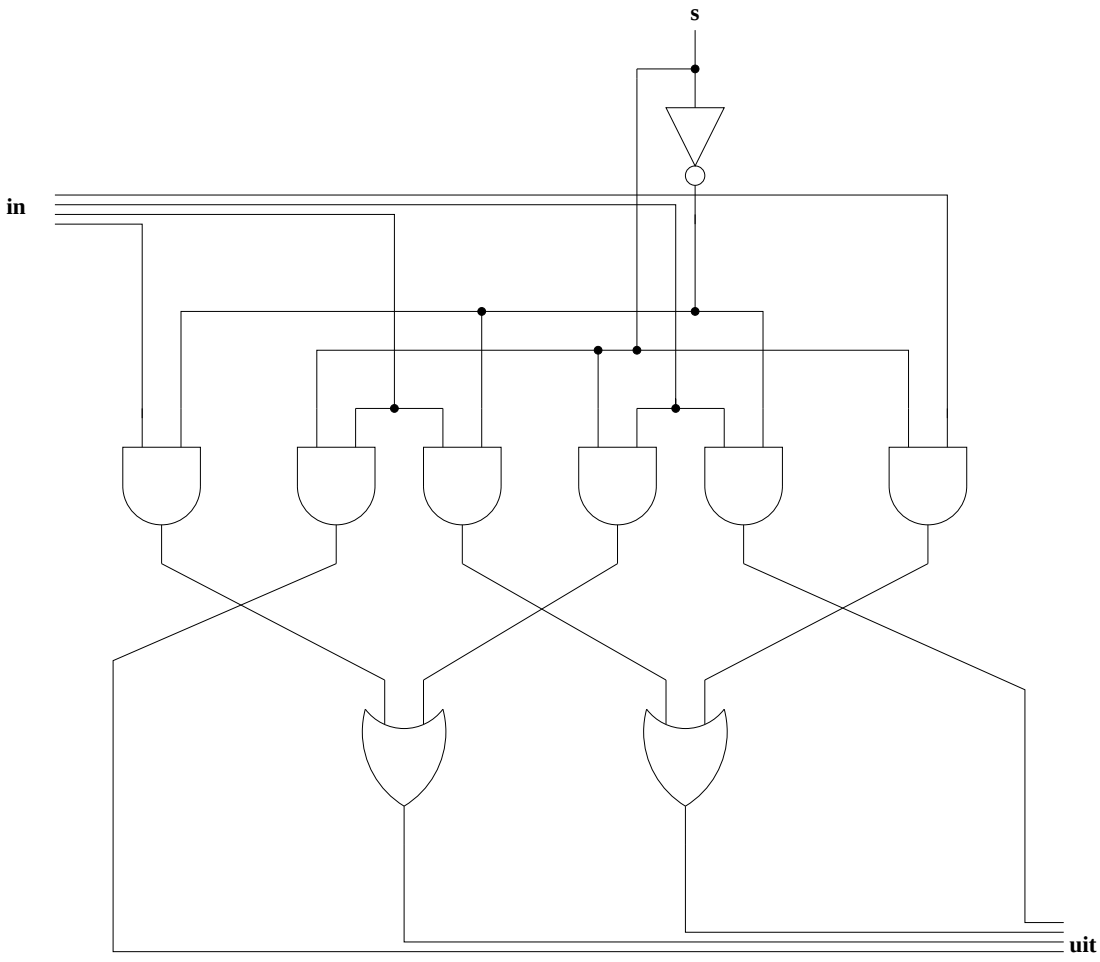
Het voorbeeld laat zien hoe binair vermenigvuldigen in feite uiteenvalt in bewerkingen voor schuiven naar links (over 0 posities als er met 1 wordt vermenigvuldigd, over 2 posities als er met 10_{bin} wordt vermenigvuldigd, over 3 posities als er met 100_{bin} wordt vermenigvuldigd, enzovoort), en vervolgens optellen van alle schuifresultaten.
Over één positie naar links schuiven is de binaire manier om met twee te vermenigvuldigen. Wat je hiermee in feite doet is een 0 achter het binaire getal zetten, en in het binaire stelsel is een 0 achter een getal zetten hetzelfde als vermenigvuldigen met twee (net zoals in het decimale stelsel een nul achter een getal zetten hetzelfde is als vermenigvuldigen met tien). Net zo: delen door twee correspondeert met over één positie verschuiven van alle bits naar rechts. Als het oorspronkelijke getal oneven was (1 in de meest rechtse positie) is er een rest 1.

Een en ander maakt duidelijk dat verschuiven over één positie naar links en naar rechts de basis vormt van binair vermenigvuldigen en delen. Deze operaties kunnen worden uitgevoerd met behulp van een schuifschakeling (Engels: *shifter*). Zo'n schuifschakeling zie je in Figuur 2.9. Om het schema niet nodeloos ingewikkeld te maken worden in de Figuur slechts vier bits verschoven. Bij echte implementaties worden in de praktijk meerdere bytes tegelijk verschoven. De schakeling uit de Figuur kan zowel naar rechts als naar links schuiven. Signaal 1 op invoerlijn **s** geeft aan dat er naar links wordt geschoven, signaal 0 op invoerlijn **s** zorgt voor schuiven naar rechts.

Opdracht 2.9 *Bedenk een schakeling voor schuiven naar links van een enkele bit, over één positie.*

Opdracht 2.10 *Bedenk een schakeling voor schuiven naar links van een hele byte, over één positie.*

Figuur 2.9: Implementatie van schakeling voor schuiven naar links of rechts.



2.3 Binair Rekenen met Negatieve Getallen

2.3.1 De Procedure voor Aftrekken

Aftrekken van binaire getallen gaat volgens de volgende tafel:

$$\begin{aligned} 1-1 &= 0 \\ 1-0 &= 1 \\ 0-1 &= ? \\ 0-0 &= 0. \end{aligned}$$

$0-1$ geeft een negatieve uitkomst, en bij ontstentenis van een afspraak om een negatief teken weer te geven hebben we op die plaats in de tafel voorlopig een vraagteken gezet. Als we weten dat er ergens in een hogere kolom een 1 staat weten we wel wat we krijgen: $10-1=1$, $100-1=11$, $1000-1=111$, enzovoort. Het laatste cijfer van de uitkomst is dan altijd een 1. Dit invullen in de tafel geeft:

$$\begin{aligned} 1-1 &= 0 \\ 1-0 &= 1 \\ 10-1 &= 1 \\ 0-0 &= 0, \end{aligned}$$

en we zien dat we (als we het lenen van een 1 uit een volgende kolom verwaarlozen) dit als een propositielogische formule kunnen weergeven: $A_1 \oplus A_2$.

Bij binair aftrekken moeten we in het geval $0-1$ altijd een 1 lenen van links. We trekken dan in feite 1 van (binair) 10 af, met uitkomst $10-1=1$. De 1 in 10 is geleend. Als in de kolom direct links een 1 staat komt daar dus na het aftrekken een 0 te staan. Als er een 0 stond wordt dat een 1, en moet er weer een 1 van links worden geleend, enzovoort. Kortom: alles gaat net als aftrekken in ons gewone tientallig stelsel: als er bij decimaal aftrekken een 1 links wordt geleend uit een kolom met cijfer 0 is de uitkomst $10-1=9$, en moet er weer een 1 van links worden geleend. Voorbeeld voor het decimale geval:

$$\begin{array}{r} 1000 \\ \underline{(-)\ 2} \\ 998 \end{array}$$

Voorbeeld (nu weer voor het binaire geval):

$$\begin{array}{r} 1110 \\ \underline{(-)\ 111} \\ 111 \end{array}$$

Merk op dat bij de berekening drie keer van links wordt geleend. Hier is een voorbeeld waarbij het lenen van een 1 links direct leidt tot twee volgende leningen links:

$$\begin{array}{r} 1000 \\ \underline{(-)\ 111} \\ 1 \end{array}$$

2.3.2 Binair Complement Representatie

De voorbeelden illustreren dat binair aftrekken lastiger is dan optellen, omdat het proces niet lokaal in termen van de operanden plus de overdracht kan worden beschreven (iets dat bij binair optellen wel kon), vanwege de complicatie van het mogelijkerwijs herhaald lenen.

Computerontwerpers hebben hier het volgende op gevonden. Eerst maken we een afspraak over de grootte van de registers voor het binaire rekenen. Neem aan: een byte, dat wil zeggen 8 posities. Als we één byte ter beschikking hebben schrijven we de negatieve getallen op als complementen van (binair) 1000.0000, dat wil zeggen complementen van $2^7 = 128$ decimaal. Een negatief getal $-n$, waarbij $n < 2^7$, wordt dus in zogenaamde binaire complementnotatie (*two's complement notation*) gereprenteerd als $2^7 - n$. De tekenkolom (in ons voorbeeld: de achtste kolom van rechts, ofwel, de meest linkse kolom als we acht bits schrijven) geeft aan of we met een complement te maken hebben. Een 1 in die kolom betekent: 2^7 aftrekken.

Hoe het complement eruit komt te zien hangt van de registergrootte af. Bij een registergrootte van twee bytes (16 binaire posities) geeft het 16de bit (van rechts af gerekend) aan of we met complementnotatie van doen hebben, en betekent een 1 in die positie dus: 2^{15} (dat wil zeggen decimaal 32768) aftrekken.

Een geheel getal in tekenloze binaire representatie wordt opgeslagen als een patroon van nullen en enen in een register, waarbij de posities worden gelezen als factoren van twee, als volgt.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Een geheel getal in binaire complementnotatie met 7 numerieke posities wordt net zo gepresenteerd, behalve dan dat de factor van twee in de 8ste kolom wordt *afgetrokken* in plaats van *opgeteld*.

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Binair complement over 7 numerieke posities geven we aan met COM_2^7. Als het achtste bit (van rechts af gerekend) op 1 staat betekent dat dat 2^7 moet worden afgetrokken. Een paar voorbeelden van complementen nemen.

$$\begin{array}{rcrcll} -1 & = & \mathrm{COM}_2^7\ 1 & = & 1111.1111 & (= \text{decimaal } 127 - 2^7 = -1) \\ -13 & = & \mathrm{COM}_2^7\ 1101 & = & 1111.0011 & (= \text{decimaal } 115 - 2^7 = -13) \\ -65 & = & \mathrm{COM}_2^7\ 100.0001 & = & 1011.1111 & (= \text{decimaal } 63 - 2^7 = -65). \end{array}$$

Complement nemen van een positief getal om het teken om te draaien gaat als volgt. Omdat (binair) $1000.0000 = 0111.1111 + 1$ kunnen we het binaire complement van een getal van hoogstens 7 bits vormen door dat getal eerst van 111.1111 af te trekken en er vervolgens 1 bij op te tellen. Het aftrekken is gewoon een kwestie van alle bits omdraaien (zie de tafel voor binair aftrekken). Om aan te geven dat we een complement nemen moeten we ook het tekenbit omdraaien. Het totaalrecept is dus: voeg nullen links toe tot de lengte gelijk is aan 8, draai alle bits om, tel vervolgens 1 bij het resultaat op.

Kunnen we nu ook omgekeerd door het complement te nemen van een negatief getal in complementnotatie het teken opnieuw omdraaien? $-n$ is gerepresenteerd als $-2^7 + (2^7 - n)$. Nogmaals het complement ten opzichte van 2^7 nemen levert: $-2^7 - 2^7 + (2^7 - 2^7 + n) = -2^8 + n$.

2^8 wordt gerepresenteerd door middel van een overdracht van 1 naar de negende kolom. Als we die overdracht *weggooien* krijgen we dus precies de goede uitkomst. Het recept is nu: alle bits omdraaien, vervolgens 1 optellen, en de overdracht naar de negende kolom weggooien. Voorbeelden:

$$\begin{array}{rcll} -(-1) & = & \mathrm{COM}_2^7\ 1111.1111 & = \quad 0000.0001\ (=1) \\ -0 & = & \mathrm{COM}_2^7\ 0000.0000 & = \quad 0000.0000\ (=0). \end{array}$$

Merk op dat binaire complementnotatie garandeert dat de representaties van 0 en van -0 hetzelfde zijn.

Opdracht 2.11 *Geef een tabel van decimale equivalenten van de binaire getallen* 0000 *tot en met* 1111*, onder de aanname dat binaire complementnotatie is gebruikt (over 4 posities).*

Opdracht 2.12 *Welk getal wordt gerepresenteerd door* 1010.1010*, gegeven dat binaire complementnotatie over 7 numerieke posities is gebruikt? Geef* COM_2^7 1010.1010*. Met welk decimaal getal correspondeert het resultaat?*

In feite hebben we dit werken met complementen al gezien toen we het hadden over de reductie van aftrekken tot optellen bij het werken met een telraam (zie paragraaf 1.3.3). Op een telraam neem je het decimale complement van een getal n door het decimale cijfer d van elke spijl te vervangen door $9-d$, en vervolgens 1 bij het resultaat op te tellen. Bij een telraam met 7 spijlen geeft dit $(9999999 - n) + 1 = 10000000 - n$.

2.3.3 Aftrekken = Optellen met Complementen

Om te demonstreren hoe aftrekken reduceert tot optellen met complementen gaan we nog even terug naar het gewone decimale stelsel. Decimale complementnotatie (zeg over lengte 4) van een getal wordt verkregen door dat getal van 9999 af te trekken en er vervolgens 1 bij op te tellen. Dus:

$$\begin{array}{rcl} \mathrm{COM}_{10}^4\ 4 & = & 9996 \\ \mathrm{COM}_{10}^4\ 347 & = & 9653 \\ \mathrm{COM}_{10}^4\ 1305 & = & 8695. \end{array}$$

Ook hier: het proces kan van links naar rechts worden uitgevoerd door elk cijfer van 9 af te trekken, en vervolgens bij het resultaat 1 op te tellen.

We kunnen nu aftrekken reduceren tot optellen van complementen door $A_1 - A_2$ te vervangen door $A_1 + \mathrm{COM}_{10}^4\ A_2$ en vervolgens 10000 van het resultaat af te trekken. Voorbeeld:

$$\begin{array}{rr} 78 & 78 \\ \underline{-23} & \underline{+9977} \\ 55 & 10055 \\ & \underline{-10000} \\ & 55 \end{array}$$

De rechtvaardiging is dat $A_1 - A_2 = A_1 + (B - A_2) - B$, voor willekeurige B. Door voor B nu het getal 10^4 te nemen krijgen we: $A_1 - A_2 = A_1 + (10^4 - A_2) - 10^4$.

Nog even kijken hoe dit werkt wanneer de uitkomst negatief wordt:

$$\begin{array}{rrr} 78 & & 78 \\ \underline{-2333} & & \underline{+7667} \\ -2255 & & 7745 \\ & \text{trek } 10.000 \text{ af:} & = -2255 \end{array}$$

Bij negatieve uitkomst krijgen we kennelijk de uitkomst in complementnotatie. Het werkt dus nog steeds. Complementnotatie geeft ons heel in het algemeen de mogelijkheid om een aftrekproces te reduceren tot complement nemen gevolgd door optellen.

Nog even verder in decimale notatie. Als we een negatief getal $-A$ met een absolute waarde kleiner dan 10^n representeren als $-10^n + (10^n - A)$, dan staat op positie $n+1$ (vanaf rechts gerekend) een 1. Als A een gewoon positief getal is kleiner dan 10^n, dan staat op positie $n+1$ vanaf rechts een 0. Dus: positie $n+1$ bevat bij decimale complementnotatie over n posities de tekeninformatie. Als op de tekenpositie een 1 staat moet dit gelezen worden als: trek 10^n af.

Hoe gaat het nu als we twee negatieve getallen (som van de absolute waarden kleiner of gelijk aan 10^3) bij elkaar willen optellen in decimale complementnotatie over drie posities? De negatieve getallen zijn dan gerepresenteerd als $-10^3 + (10^3 - A)$ en $-10^3 + (10^3 - B)$. Optellen geeft:

$$-10^3 + (10^3 - A) - 10^3 + (10^3 - B) = -2 \cdot 10^3 + (2 \cdot 10^3 - (A + B)).$$

We krijgen kennelijk de uitkomst in complement van 2×10^3 notatie.

Het mooie is nu dat hier bij binair rekenen gemakkelijk een mouw aan te passen valt. Binaire complementnotatie over n posities zou voor dit geval immers geven:

$$\begin{aligned} -2^n + (2^n - A) - 2^n + (2^n - B) &= \\ &= -2^{n+1} + (2^{n+1} - (A + B)) \\ &= -2^{n+1} - 2^n + (2^n - (A + B)). \end{aligned}$$

Dus: als we de overdracht naar de $n+2$-de positie weggooien (dit komt overeen met: een factor 2^{n+1} verwaarlozen) krijgen we precies de goede uitkomst in binaire complementnotatie met n numerieke posities. Let wel: dit gaat alleen op als de uitkomst nog binnen het te representeren bereik valt, dat wil zeggen: mits $A + B \leq 2^n$.

Bij representatie van gehele getallen in één byte (acht posities) kunnen we getallen van -128 $(= -2^7)$ tot en met 127 $(= 2^7 - 1)$ representeren. Voorbeelden van gewoon binair aftrekken versus optellen in binaire complementnotatie met deze representatie:

$$\begin{array}{rrr} 1.1011 & & 0001.1011 \\ \underline{-1.0100} & & \underline{+1110.1100} \\ 111 & & 1.0000.0111 \\ & \text{gooi overdracht naar pos 9 weg:} & = 0000.0111 \end{array}$$

$$\begin{array}{rrr} -1 & & 1111.1111 \\ \underline{-1} & & \underline{+1111.1111} \\ -10 & & 1.1111.1110 \\ & \text{gooi overdracht naar pos 9 weg:} & = 1111.1110 \end{array}$$

In binaire complementnotatie over 15 numerieke posities gebruiken we het meest significante bit (het bit helemaal links, dus op positie 16 vanaf rechts) voor de tekeninformatie: 1 betekent negatief getal in complementnotatie, 0 betekent: positief getal. De negatieve getallen die we kunnen representeren in twee bytes lopen van -2^{15} (binaire representatie 1000.0000.0000.0000) tot -1 (binaire representatie 1111.1111.1111.1111), en de niet-negatieve van 0 (binaire representatie 0000.0000.0000.0000) tot $2^{15}-1$ (binaire representatie 0111.1111.1111.1111).

Opdracht 2.13 *Wat is het kleinste getal dat gerepresenteerd kan worden in binaire complementnotatie, wanneer we per getal vier bytes ter beschikking hebben?*

2.4 Logica: Redeneren = Rekenen

Zoals wij in paragraaf 2.2 gezien hebben, kun je logische operaties als *EN*, *OF* en *NIET* gebruiken om circuits te bouwen waarmee je binair kunt rekenen. Je zou dus kunnen zeggen dat een circuit een soort van logische (reken)machine voorstelt. Een belangrijk verschil tussen circuits en de machinemodellen zoals de Turing machine en de registermachine die in het volgende hoofdstuk aan de orde komen is dat er bij een circuit altijd een bovengrens op de lengte van invoer en uitvoer zit. Kijk bijvoorbeeld naar de half-adder in Figuur 2.6. Dit circuit kan alleen getallen van 8 bits bij elkaar optellen. Een Turing machine daarentegen kan getallen van willekeurige lengte bij elkaar optellen.

Het rekenen met logische operaties kan echter ook op een andere manier worden bekeken. Laten we afspreken dat we de kleine letters $a, b, \ldots, r$ en $p_1, p_2, \ldots$ gebruiken als afkortingen voor de meest eenvoudige beweringen zoals 'Het regent' of 'Jan houdt van Marijke'. Met behulp van de logische operaties $\neg$ (niet), $\wedge$ (en), $\vee$ (of) en $\rightarrow$ (impliceert) kunnen we nu complexe beweringen als *logische formules* weergeven, zodat bijvoorbeeld de zin 'Als ik de loterij win, dan koop ik een auto of een huis' met de formule $l \rightarrow (a \vee h)$ correspondeert. Hierbij staat l voor 'Ik win de loterij', a voor 'Ik koop een auto' en h voor 'Ik koop een huis'.

2.4.1 Waarheidstabellen

Zoals al eerder vermeld, kun je de getallen 1 en 0 die de basis van het binaire rekenen vormen ook lezen als *waar* en *onwaar*, en onder deze interpretatie zijn de volgende waarheidstabellen van de logische operaties dan ook heel natuurlijk: ze geven aan hoe je de waarheid/onwaarheid van een complexe bewering kunt uitrekenen op basis van de waarheid/onwaarheid van eenvoudiger beweringen.

p	$\neg p$
0	1
1	0

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

p	q	$p \rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

Als we bijvoorbeeld naar de waarheidstabel van negatie kijken en 1 als 'waar' en 0 als 'onwaar' lezen, dan geeft de waarheidstabel van negatie aan dat $\neg p$ waar is als p onwaar is, en omgekeerd. De zin 'Het regent niet' is waar als de zin 'Het regent' onwaar is, en omgekeerd. Voor de conjunctie $p \wedge q$ aan de andere kant geeft de waarheidstabel aan dat deze alleen waar is als zowel

p als ook q waar zijn. De zin 'Het regent en het waait' zal onwaar zijn als het niet regent of als het niet waait, en dus is de zin alleen waar als de zinnen 'Het regent' en 'Het waait' allebei waar zijn. De disjunctie kun je op een soortgelijke manier uitleggen: 'Het regent of het waait' is alleen onwaar als het niet regent en niet waait. Dit houdt dus in dat, als het zowel regent als waait, de zin waar is.

Nu de implicatie. Stel dat ik beweer 'Als het bliksemt, dondert het', dan kunnen we dit opschrijven als $p \to q$. Als jij wilt aantonen dat deze bewering onwaar is, moet je volgens de waarheidstabel een situatie vinden waar er bliksem is (p is waar), maar geen donder (q is onwaar). Verder is dit de enige manier om de onwaarheid van $p \to q$ aan te tonen (in alle andere rijtjes van de waarheidstabel staat immers een 1)! Als jij me bijvoorbeeld op een situatie wijst waar er donder is maar geen bliksem, volgt hieruit niet dat de zin 'Als het bliksemt, dondert het' onwaar moet zijn, want deze zin beweert alleen iets voor gevallen waar er wel bliksem is.

Met behulp van de waarheidstabellen kun je nu ook de waarheidswaarde (het waar of onwaar zijn) van ingewikkelder formules of beweringen uitrekenen. De zin 'Als de zon schijnt, gaat Joost naar het strand, anders naar het zwembad' kunnen we vrij makkelijk naar onze logische taal vertalen met de volgende vertaalsleutel.

a De zon schijnt.
b Joost gaat naar het strand.
c Joost gaat naar het zwembad.

De corresponderende formule is $(a \to b) \wedge (\neg a \to c)$, en de waarheidstabel kun je als volgt construeren.

a	b	c	$a \to b$	$\neg a$	$\neg a \to c$	$(a \to b) \wedge (\neg a \to c)$
0	0	0	1	1	0	0
0	0	1	1	1	1	1
0	1	0	1	1	0	0
0	1	1	1	1	1	1
1	0	0	0	0	1	0
1	0	1	0	0	1	0
1	1	0	1	0	1	1
1	1	1	1	0	1	1

Kort samengevat: in situaties met Joost bij zonneschijn aan het strand is de zin waar, in situaties zonder zon met Joost in het zwembad ook, en in alle andere situaties is de zin onwaar.

Hier is een wat ingewikkelder voorbeeld. Stel Saskia is met Jaap aan het kaarten, en in de loop van het spel doet zij de volgende bewering. 'Jaap heeft een aas als hij niet een heer of een boer heeft'. Wanneer is deze bewering waar? Merk op dat er een ambiguïteit in deze bewering zit, dat wil zeggen, er zijn twee mogelijke interpretaties.

- Volgens de eerste interpretatie wordt er beweerd dat Jaap een aas heeft als hij (1) geen heer heeft, of (2) een boer heeft.
- Volgens de tweede interpretatie heeft Jaap een aas als het niet het geval is dat hij een heer of een boer heeft.

Deze ambiguiteit komt tevoorschijn als je de zin naar onze logische taal vertaalt, met de volgende vertaalsleutel. a: Jaap heeft een aas, b: Jaap heeft een boer, en h: Jaap heeft een heer. Je zou de zin nu misschien als $\neg b \vee h \to a$ willen vertalen, maar deze formule is dan even ambigu

als de oorspronkelijke Nederlandse zin: net als de zin is ze voor tweeërlei lezing vatbaar. Een soortgelijk probleem ken je in feite al: de rekenkundige expressie $7+4*3$ kan op twee verschillende manieren worden geïnterpreteerd, als $(7+4)*3 = 33$ of als $7+(4*3) = 19$, en zoals je ziet: de twee lezingen geven een verschillende uitkomst. Voor rekenkundige bewerkingen biedt 'Meneer Van Dalen Wacht Op Antwoord' uitkomst. Bij logische formules lossen we het probleem op door haakjes te zetten die de gewenste interpretatie ondubbelzinnig vastleggen. Voor de eerste interpretatie is het resulaat dan $(\neg b \vee h) \rightarrow a$, terwijl de tweede interpretatie $\neg(b \vee h) \rightarrow a$ oplevert. Laten we eens de waarheidstabellen van deze twee formules bekijken.

a	b	h	$\neg b$	$(\neg b \vee h)$	$(\neg b \vee h) \rightarrow a$
0	0	0	1	1	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	1	1

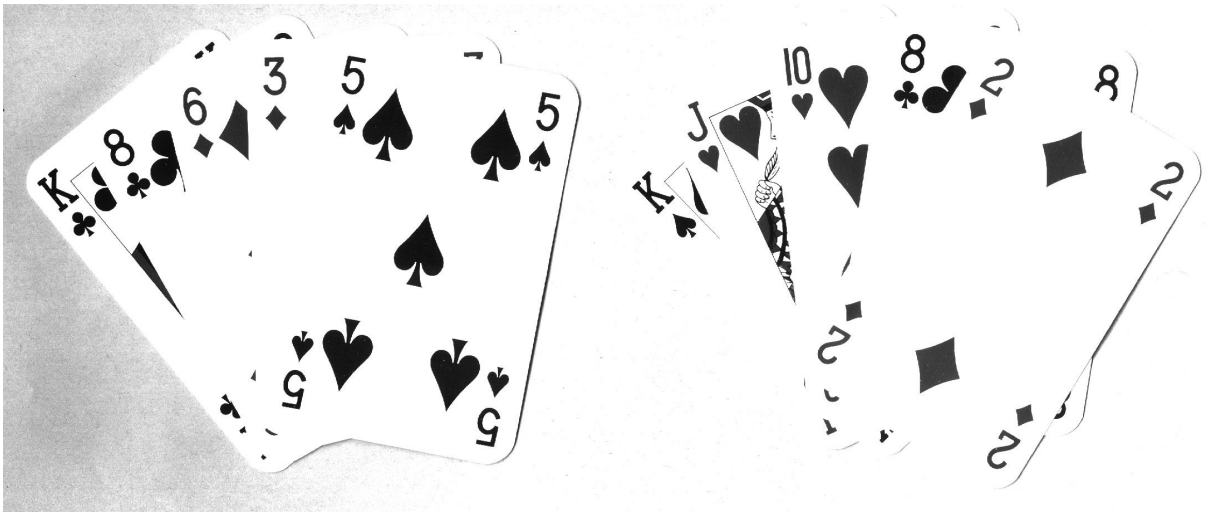
a	b	h	$b \vee h$	$\neg(b \vee h)$	$\neg(b \vee h) \rightarrow a$
0	0	0	0	1	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	0	1	1
1	0	1	1	0	1
1	1	0	1	0	1
1	1	1	1	0	1

Zoals je uit de waarheidstabellen kunt aflezen, de enige manier dat $\neg(b \vee h) \rightarrow a$ onwaar kan zijn is dat a, b en h alledrie onwaar zijn. Dus als Jaap geen boer, geen aas en geen heer heeft. $(\neg b \vee h) \rightarrow a$ aan de andere kant is ook onwaar als Jaap geen aas en geen boer, maar wel een heer heeft. De conclusie: zinnen in natuurlijke taal zijn vaak ambigu; door deze zinnen in een formele kunsttaal, zoals de taal van logische formules, te vertalen worden de verschillende mogelijke lezingen zichtbaar; met behulp van waarheidstabellen kun je vervolgens situaties bekijken waarin verschillende interpretaties verschillende waarheidswaarden tot gevolg hebben.

Opdracht 2.14 *Vertaal de volgende Nederlandse zinnen naar de propositielogica, en maak voor de resulterende formule een waarheidstabel. Beschrijf in woorden een situatie waarin de formule wel waar is en een situatie waarin de formule niet waar is. Voor het geval dat een zin ambigu is, leg dan de verschillende betekenissen uit en vertaal ze naar de propositielogica.*

1. *Jan heeft een auto gekocht, maar zijn vrouw noch hijzelf heeft een rijbewijs.*
2. *Piet doet zijn lichten aan als het donker is, maar niet als het regent.*
3. *Sandra rijdt naar zee als het niet regent of te warm is.*

Figuur 2.10: Twee kaarthanden waarbij $(\neg b \vee h) \rightarrow a$ onwaar is, maar $\neg(b \vee h) \rightarrow a$ waar.



4. *Kees doet zijn lichten niet alleen aan als het donker is, maar ook als het regent of mist.*

5. *Johan gaat met de auto, tenzij de auto kapot is of het mooi weer is.*

Formules zoals $p \rightarrow (q \vee r)$ zijn in feite niets anders dan een compacte representatie van een circuit, en de waarheidtafel voor zo'n formule geeft het gedrag van het circuit voor alle mogelijke invoeren weer. Het probleem is echter dat waarheidstabellen vrij snel te groot worden om ze goed te kunnen hanteren. Wij hebben al gezien dat je bij een formule met 1 propositieletter p een waarheidstabel van 2 regels krijgt, een regel voor het geval dat p waar is en een regel voor het geval dat p onwaar is. Bij twee propositieletters p en q kon je uit de waarheidtafels voor conjunctie et cetera aflezen dat er 4 regels nodig zijn. In het algemeen heb je voor een formule van n propositieletters een waarheidstabel van 2^n regels nodig. Voor 10 propositieletters zijn dit al 1024 regels.

2.4.2 De Prinses of de Tijger?

Je kunt waarheidstabellen en circuits niet alleen gebruiken om de waarheidswaarde van een ingewikkelde bewering uit te rekenen, je kunt er ook redeneerproblemen mee oplossen, dat wil zeggen, uitrekenen. Hier volgt een voorbeeld uit Raymond Smullyans bekende boek *De Prinses of de Tijger?* (Oorspronkelijke Engelse editie: Knopf, 1982).

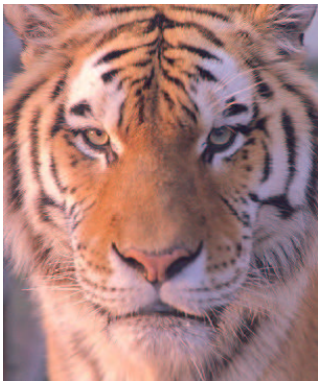
> De koning van een land ver weg beleeft er plezier aan om zijn gevangenen logica-puzzles te laten oplossen. Verder wil hij zijn twee dochters aan een man ten huwelijk geven die de kunst van het redeneren als een meester beheerst. Een gevangene heeft de keuze om een van twee deuren te openen. Achter elke deur zit of een prinses of een tijger. Aan de deuren hangen bordjes die de gevangene informatie geven:

deur 1

Achter deze deur zit een prinses,
en achter de andere deur een tijger.

deur 2

Achter een van de twee deuren
zit een prinses,
en achter de andere een tijger.



De koning vertelt de gevangene dat een van de twee bordjes de waarheid vertelt, maar het andere niet. Als de koning de waarheid spreekt en de gevangene liever met de prinses trouwt dan door de tijger wordt opgegeten, welke deur moet hij dan openen?

Probeer eerst eens zelf de oplossing te beredeneren voordat je verder leest.

Informele Redenering Precies een van de twee bordjes is waar. Stel dus dat het bordje op deur 1 waar is. Dan zit achter deur 1 een prinses en achter deur 2 een tijger te wachten. Dat maakt het bordje op deur 2 ook waar, maar dat kan niet, want de koning zei dat er maar een van de twee bordjes waar was. We hebben dus een tegenspraak. Hieruit kunnen we concluderen dat het bordje op deur 1 onwaar is en het bordje op deur 2 waar, dat wil zeggen: achter een van de deuren zit een prinses en achter de andere een tijger. Omdat het bordje op deur 1 onwaar is, kan het niet zo zijn dat de prinses achter deur 1 zit en de tijger achter deur 2, de enige mogelijkheid is dus dat de prinses achter deur 2 zit en de tijger achter deur 1. De gevangene doet er dus verstandig aan deur 2 te openen.

Formele Redenering Nu gaan we het probleem met behulp van onze logische notatie formaliseren en oplossen. Er zijn vier mogelijke situaties, want achter elke deur kan of een prinses of een tijger zitten. Laat p_1 staan voor 'Er zit een prinses achter deur 1', en laat p_2 staan voor 'Er zit een prinses achter deur 2'. De vier mogelijke situaties komen dan overeen met de vier mogelijke toekenningen van waarheidswaarden aan deze twee beweringen. De bewering op deur 1 kunnen we nu opschrijven als

$$d_1 : \quad p_1 \wedge \neg p_2,$$

terwijl de bewering op deur 2 te schrijven valt als

$$d_2 : \quad (p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_2),$$

dat wil zeggen: er is een prinses in kamer 1 of 2, en er is een tijger (en dus: geen prinses) in kamer 1 of 2. Laten we deze twee formules afkorten als respectievelijk d_1 en d_2. De koning zegt nu dat precies een van de twee bordjes waar moet zijn, dus $(d_1 \wedge \neg d_2) \vee (d_2 \wedge \neg d_1)$ of zonder afkortingen

$$\overbrace{((p_1 \wedge \neg p_2)}^{d_1} \wedge \neg \overbrace{((p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_2))}^{d_2})$$
$$\vee(\underbrace{((p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_2))}_{d_2} \wedge \neg \underbrace{(p_1 \wedge \neg p_2)}_{d_1})$$

is waar.

Nadat we het probleem naar de formele taal hebben vertaald, is de vraag nu: in welke situaties is deze formule waar? De waarheidstabel voor de formule geeft het antwoord:

p_1	p_2	d_1	d_2	$(d_1 \wedge \neg d_2) \vee (d_2 \wedge \neg d_1)$
0	0	0	0	0
0	1	0	1	1
1	0	1	1	0
1	1	0	0	0

Es is dus maar één situatie waarin de koning de waarheid vertelt, regel 2 van de waarheidstabel, de situatie waarin de tijger achter deur 1 zit en de prinses achter deur 2.

Zoals gezegd, geeft de waarheidstabel het gedrag van het circuit op alle mogelijke invoeren weer, en je kunt dus het circuit als hardware oplossing voor het prinses/tijger-probleem beschouwen. Probeer alle vier mogelijke stroom aan/uit toestanden voor de twee invoerdraden, en als er op de uitvoerdraad stroom is heb je de oplossing gevonden.

Opdracht 2.15 *Probeer met behulp van waarheidstabellen te ontdekken achter welke deur een prinses zit, gegeven de informatie:*

1.

deur 1

In minstens een van de twee kamers zit een prinses.

deur 2

In de andere kamer zit een tijger.

De koning vertelt je dat de bordjes beide waar of beide onwaar zijn.

2.

deur 1

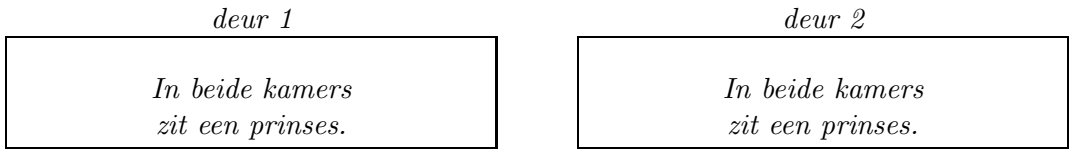
Een of twee van de volgende uitspraken zijn waar:
(a) Er zit een tijger in deze kamer.
(b) Er zit een prinses in de andere kamer.

deur 2

In de andere kamer zit een prinses.

De koning vertelt je dat de bordjes beide waar of beide onwaar zijn.

3.



De koning vertelt je dat, als er een prinses in kamer 1 zit, het gegeven op deur 1 waar is, anders niet. Verder zegt hij dat, als er een tijger in kamer 2 zit, het gegeven op deur 2 waar is, anders niet.

2.4.3 Bijzondere Patronen in Waarheidstabellen

Zoals we hebben gezien, kon de gevangene uit de gegevens concluderen dat een prinses achter deur 2 moet zitten, of preciezer, als de uitspraak van de koning $(d_1 \wedge \neg d_2) \vee (d_2 \wedge \neg d_1)$ waar is, dan moet ook p_2 waar zijn. Wij schrijven dit als

$$(d_1 \wedge \neg d_2) \vee (d_2 \wedge \neg d_1) \models p_2$$

en zeggen dat p_2 een *(logisch) gevolg* is van $(d_1 \wedge \neg d_2) \vee (d_2 \wedge \neg d_1)$. Algemeen gesproken is een formule B een logisch gevolg van formule A als B waar is in elke situatie waarin A ook waar is. In de waarheidstabel betekent dit dat in elke rijtje waar bij formule A een 1 staat ook bij formule B een 1 moet staan. In het geval van de prinses en de tijger kun je nagaan dat ook $\neg p_1$ uit de gegevens volgt, dat wil zeggen:

$$(d_1 \wedge \neg d_2) \vee (d_2 \wedge \neg d_1) \models \neg p_1,$$

en we hebben al eerder opgemerkt dat je uit de gegevens inderdaad kunt afleiden dat er een tijger achter deur 1 zit. Verder zie je aan hand van de waarheidstabel ook dat d_2 een logisch gevolg van de gegevens is; dus als de koning de waarheid zegt moet het bordje op deur 2 waar zijn. De formule d_1 aan de andere kant is geen logisch gevolg van de gegevens.

Opdracht 2.16 *Ga met waarheidstabellen na of de volgende beweringen kloppen.*

1. $p \vee (q \wedge r) \models (p \vee q) \wedge r$
2. $(p \rightarrow q) \wedge (p \rightarrow r) \models q \leftrightarrow r$
3. $(p \rightarrow q) \wedge (q \rightarrow r) \models \neg r \rightarrow \neg p$
4. $\neg(p \rightarrow (q \wedge r)) \models (r \rightarrow (p \wedge q)) \rightarrow \neg r$

De koning met de twee dochters die wij net tegenkwamen was heel aardig tegenover de gevangene, want hij gaf hem informatie over de bordjes waaruit de gevangene de goede deur kon concluderen. Stel echter dat de koning de gevangene liever niet als schoonzoon wil hebben. Als de gevangene nu aan de koning vraagt om hem over de bordjes in te lichten zegt de koning: 'Als het bordje op deur 1 waar is, dan is ook het bordje op deur 2 waar'. De gevangene die inmiddels iets van logica af weet, realiseert zich dat deze uitspraak kan worden weergegeven als $d_1 \rightarrow d_2$. Omdat hij het

nut van waarheidstabellen heeft begrepen, gaat hij zelfverzekerd aan de slag en produceert het volgende.

p_1	p_2	d_1	d_2	$d_1 \to d_2$
0	0	0	0	1
0	1	0	1	1
1	0	1	1	1
1	1	0	0	1

Wat valt hier nu uit te concluderen? De waarheidstabel laat zien dat de formule $d_1 \to d_2$, of helemaal uitgeschreven

$$(p_1 \wedge \neg p_2) \to ((p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_2)),$$

in elke situatie waar is. Zo'n formule noemt men in de logica een *tautologie*. In een waarheidstabel herken je een tautologie dus aan het feit dat er in elk rijtje van de kolom van de formule een 1 staat. Je kon in het voorbeeld zien dat de koning door het uiten van de tautologie $d_1 \to d_2$ geen enkele informatie geeft over de werkelijke situatie, omdat zijn uiting in *elke* situatie waar is; dit geldt voor tautologieën in het algemeen.

Opdracht 2.17 *Er is trouwens ook een verband tussen de noties van tautologie en logisch gevolg: B is een logisch gevolg van A precies dan als de formule $A \to B$ een tautologie is. Kun je beredeneren waarom dit zo is?*

Opdracht 2.18 *Ga met waarheidstabellen na welke formules tautologieën zijn.*

1. $p \to (q \vee p)$
2. $(p \to q) \to (p \to \neg r)$
3. $\neg(p \vee q) \leftrightarrow (\neg p \wedge \neg q)$

Als de gevangene beseft dat hij in feite geen enkele informatie van de koning heeft gekregen vraagt hij de koning opnieuw om hem iets over de waarheid of onwaarheid van de bordjes te vertellen. De koning bedenkt nu dat het toch wel billijk is de gevangene ten minste een kleine hint te geven, en hij zegt: 'De bordjes op de twee deuren zijn niet allebei waar.' Weer gaat de gevangene aan de slag: Als de koning de waarheid spreekt moet dus de formule $\neg(d_1 \wedge d_2)$ waar zijn. Hij maakt opnieuw een waarheidstabel (kijk nog maar even niet naar de laatste kolom).

p_1	p_2	d_1	d_2	$\neg(d_1 \wedge d_2)$	$\neg p_2 \to \neg p_1$
0	0	0	0	1	1
0	1	0	1	1	1
1	0	1	1	0	0
1	1	0	0	1	1

De enige situatie die de bewering van de koning uitsluit (0 in de waarheidstabel) is de situatie met een prinses achter deur 1 en een tijger achter deur 2. De beste strategie voor de gevangene zou dus zijn om deur 2 te openen. Het zou kunnen dat hij dan nog steeds een tijger ontmoet, maar in dat geval had er achter deur 1 ook een tijger gezeten. De twee formules $\neg p_2 \to \neg p_1$ en $\neg(d_1 \wedge d_2)$ zijn in feite *(logisch) equivalent*, dat wil zeggen: zij hebben dezelfde waarheidswaarde

in elke situatie. In een waarheidstabel kun je dit heel makelijk zien: in elk rijtje hebben de twee formules hetzelfde getal (0 of 1) staan.

Evenals de notie van logisch gevolg valt ook de notie van logische equivalentie uit te drukken in termen van een tautologie. Formules A en B zijn logisch equivalent precies dan als de formule $(A \to B) \wedge (B \to A)$ een tautologie is.

Opdracht 2.19 *Welke van de volgende tweetallen formules zijn logisch equivalent?*

1. $p \leftrightarrow (q \leftrightarrow r)$ *en* $(p \leftrightarrow q) \leftrightarrow r$
2. $p \wedge (q \vee r)$ *en* $(p \wedge q) \vee (p \wedge r)$
3. $\neg\neg(p \wedge q)$ *en* $p \wedge q \wedge r$
4. $p \to q$ *en* $\neg p \vee q$

2.4.4 Redeneren als Manipuleren van Informatietoestanden

We kunnen het proces van redeneren als volgt opvatten. Bij elke bewering a hoort de verzameling van alle situaties waar a waar is. Noem die verzameling van situaties A. Net zo: noem de verzameling van alle situaties waar a *niet* waar is $\bar{A}$. Zo'n verzameling kunnen we zien als een *informatietoestand*: de 'mogelijke toestanden van de wereld gezien onze huidige kennis'. Naarmate we meer informatie krijgen, krimpt de klasse situaties voor de beweringen tot nu toe (krimp in onzekerheid is kennisgroei). Of een conclusie geldig is hangt af van de eindtoestand die bereikt is na dynamische verwerking van de premissen. Hier is het geval van Contrapositie. We laten zien dat na verwerking van de informatie $a \to b$ en $\neg b$ we in een informatietoestand zijn aangeland waar $\neg a$ geldt.

begintoestand (geen informatie)	$\{AB, A\bar{B}, \bar{A}B, \bar{A}\bar{B}\}$
update met $a \to b$ leidt tot	$\{AB, \bar{A}B, \bar{A}\bar{B}\}$
update met $\neg b$ leidt tot	$\{\bar{A}\bar{B}\}$
test op $\neg a$ slaagt.	

De ongeldige variant met $a \to b, \neg a$ geeft toestand $\{\bar{A}B, \bar{A}\bar{B}\}$ waar $\neg B$ niet opgaat. Hier is nog een voorbeeld.

begintoestand	$\{ABC, A\bar{B}C, \bar{A}BC, \bar{A}\bar{B}C, AB\bar{C}, A\bar{B}\bar{C}, \bar{A}B\bar{C}, \bar{A}\bar{B}\bar{C}\}$
update met $a \vee b$	$\{ABC, A\bar{B}C, \bar{A}BC, AB\bar{C}, A\bar{B}\bar{C}, \bar{A}B\bar{C}\}$
update met $\neg a \vee c$	$\{ABC, A\bar{B}C, \bar{A}BC, \bar{A}B\bar{C}\}$

De conclusie $b \vee c$ volgt hier door directe inspectie van de eindtoestand. Maar de exclusieve disjunctie *òf b òf c* volgt niet, zoals blijkt uit de open mogelijkheden $ABC, \bar{A}BC$. Merk ook nog op dat de premissen samen sterker zijn (meer informatie bevatten) dan de conclusie $b \vee c$. Immers, gevallen als $AB\bar{C}$ zijn uitgesloten, hoewel compatibel met deze laatste bewering. (Een echt optimale conclusie is de conjunctie der premissen: $(a \vee b) \wedge (\neg a \vee c)$.) Verder gebruik van deze dynamische kijk: aan het aantal wegvallende mogelijkheden kun je precies meten hoe *informatief* een nieuwe bewering in een gegeven informatietoestand is.

2.4.5 Redeneren over Natuurlijke Getallen

De atomaire beweringen p, q, r, et cetera die we tot nu toe zijn tegengekomen hebben we geïnterpreteerd als beweringen over onze natuurlijke omgeving. We kunnen onze logische taal echter net zo gebruiken om complexe uitspraken over natuurlijke getallen te maken. In het programmeergedeelte van dit boekje zal dit later nog blijken.

De basisbeweringen die wij over natuurlijke getallen maken zijn vergelijkingen: $x = y$ betekent dat twee natuurlijke getallen hetzelfde zijn, $x < y$ betekent dat getal x kleiner is dan getal y, $x > y$ betekent dat getal x groter is dan getal y, en $x \leq y$ ($x \geq y$) dat x kleiner (groter) of gelijk is aan y. Als afkorting schrijven wij ook $x \neq y$ voor $\neg(x = y)$. Met behulp van deze beweringen kunnen wij in de taal van de propositielogica ingewikkelde beweringen opschrijven zoals de bewering *Als x kleiner dan 7 is en y groter of gelijk aan 11, dan is z gelijk aan x of verschilt van y.* Vertaald naar de propositielogica is dit:

$$((x < 7) \wedge (y \geq 11)) \rightarrow (z = x \vee z \neq y)).$$

Over natuurlijke getallen hebben wij de volgende equivalenties ($A \equiv B$ betekent dat A en B logisch equivalent zijn).

$$\begin{aligned} x = y &\equiv (x \leq y) \wedge (y \leq x) \\ x \leq y &\equiv y \geq x \\ x \leq y &\equiv (x < y) \vee (x = y) \\ \neg x \leq y &\equiv x > y \\ x \neq y &\equiv (x < y) \vee (y < x). \end{aligned}$$

Tenslotte kunnen wij in het bijzonder beweringen over het rekenen met 0 en 1 bekijken. In dat geval kun je de basisuitdrukkingen zoals $\leq$ en $>$ ook als tweeplaatsige logische operaties opvatten, die hun eigen waarheidstabellen hebben, net als $\vee$ en $\wedge$.

x	y	$x < y$
0	0	0
0	1	1
1	0	0
1	1	0

x	y	$x \leq y$
0	0	1
0	1	1
1	0	0
1	1	1

x	y	$x = y$
0	0	1
0	1	0
1	0	0
1	1	1.

Opdracht 2.20 *Welke formules uit de propositielogica corresponderen met $x < y$, $x \leq y$ en $x = y$? Preciezer, vind formules A_1, A_2 en A_3 uit de propositielogica die alleen x, y en de connectieven $\wedge$, $\vee$, $\neg$ en $\rightarrow$ bevatten, zodat $x = y$ logisch equivalent is met A_1, $x < y$ met A_2 en $x \leq y$ met A_3.*

2.4.6 De Rekenkunde van de Rede

In opdracht 2.19 kwamen wij al een aantal equivalente formules tegen, en we hebben gezien hoe je met waarheidstabellen de equivalentie van twee willekeurige formules kunt uitzoeken. Er is echter ook een meer rekenkundige manier om aan te tonen dat twee formules equivalent zijn, en we zullen deze methode eerst met een eenvoudig voorbeeld demonstreren.

Stel dat we al weten dat de formules $p \vee q$ en $q \vee p$ equivalent zijn (commutativiteit), en dat we ook weten dat $p \vee (q \vee r)$ en $(p \vee q) \vee r$ equivalent zijn (associativiteit). Deze twee

equivalenties kunnen we in termen van rekenkundige vergelijkingen opschrijven met variabelen voor de propositieletters

$$\begin{aligned} x \vee y &= y \vee x \\ x \vee (y \vee z) &= (x \vee y) \vee z, \end{aligned}$$

omdat de equivalenties voor alle mogelijke toekenningen van 0 en 1 aan x,y en z opgaan (dit is juist de betekenis van logische equivalentie). Stel, we willen nu weten of $p \vee (q \vee r)$ equivalent is aan $q \vee (p \vee r)$. Hiervoor hoef je nu niet meer een waarheidstabel te maken, want je kunt gewoon rekenkundig de vergelijking afleiden:

$$\begin{aligned} x \vee (y \vee z) &= (x \vee y) \vee z && \text{(associativiteit)} \\ &= (y \vee x) \vee z && \text{(commutativiteit)} \\ &= y \vee (x \vee z) && \text{(associativiteit)}. \end{aligned}$$

Hieruit volgt dan meteen de equivalentie van de twee genoemde formules. Voor de rest van dit hoofdstuk laten we de implicatie even buiten beschouwing. Dit is geen essentiële beperking, omdat je $A \to B$ toch door de equivalente formule $\neg A \vee B$ kunt vervangen; het maakt het volgende verhaal echter wat makkelijker.

Verder zullen wij de taal van de propositielogica ietsje uitbreiden. Zoals je kunt nagaan, zijn de formules $p \vee \neg p$ en ook $\neg(p \wedge \neg p)$ tautologieën. Wij introduceren nu het symbool $\top$ (lees 'waar') als formule die altijd waar is. Je kunt $\top$ dus beschouwen als een speciale propositieletter die altijd dezelfde waarde krijgt, namelijk 1. Dat betekent dat in een waarheidstabel $\top$ altijd de waarde 1 heeft, onafhankelijk van de waarden die aan de propositieletters worden toegekend. Hiernaast introduceren we ook het symbool $\bot$ (lees: 'onwaar') als formule die altijd onwaar is, dat wil zeggen: $\bot$ krijgt altijd de waarde 0 in een waarheidstabel. Als voorbeeld geven we hier de waarheidstabel voor de formule $(p \wedge \top) \vee (q \wedge \bot)$. Zoals blijkt is deze formule equivalent aan p.

p	q	$p \wedge \top$	$q \wedge \bot$	$(p \wedge \top) \vee (q \wedge \bot)$
0	0	0	0	0
0	1	0	0	0
1	0	1	0	1
1	1	1	0	1.

Het volgende stel vergelijkingen zou je als de basiswetten van de logische rekenkunde kunnen beschouwen:

$x \vee y = y \vee x$	$x \wedge y = y \wedge x$
$x \vee (y \vee z) = (x \vee y) \vee z$	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
$x \vee \bot = x$	$x \wedge \top = x$
$x \vee \neg x = \top$	$x \wedge \neg x = \bot$

Voor elke vergelijking kun je met waarheidstabellen aantonen dat de linker- en rechterkant inderdaad logisch equivalent zijn, en elke vergelijking die je met behulp van deze 10 wetten kunt afleiden correspondeert weer met een logische equivalentie. Maar wat deze 10 wetten zo bijzonder maakt is dat *elke* logische equivalentie uit deze 10 wetten kan worden afgeleid! Het bewijs van dit feit zullen we hier niet kunnen leveren, maar in de oefeningen kun je tenminste proberen enkele logische equivalenties uit deze 10 vergelijkingen af te leiden.

Opdracht 2.21 *Leid de volgende vergelijkingen af.*

1. $\neg\top = \bot$
2. $x \vee (y \wedge \neg y) = x$
3. $x \vee x = x$
4. $x \vee \top = \top$
5. $x \vee (x \wedge y) = x$
6. *Laat zien dat, als* $x \vee y = x \wedge y$*, dan* $x = y$*.*
7. *Laat zien dat, als* $x \wedge y = \bot$ *en* $x \vee y = \top$*, dan* $y = \neg x$*.*
8. *Laat zien dat* $\neg\neg x = x$*.*
9. *Laat zien dat* $\neg(x \vee y) = \neg x \wedge \neg y$ *en* $\neg(x \wedge y) = \neg x \vee \neg y$*.*

De rekenkundige manier om naar logische equivalentie te kijken is handig omdat het manipuleren van vergelijkingen vaak makkelijker is dan het construeren van een waarheidstabel. Om aan te tonen dat twee formules A en B *niet* equivalent zijn is de methode echter minder geschikt, want het feit dat je geen afleiding voor $A = B$ hebt kunnen bedenken hoeft nog niet te betekenen dat er ook werkelijk geen afleiding bestaat. In zo'n geval is een waarheidstabel het betere werktuig.

Zoals al eerder opgemerkt, is het uitrekenen van logische equivalenties belangrijk, onder andere voor het vereenvoudigen van circuits, maar met behulp van deze logische rekenkunde kun je ook tautologieën en logische gevolgen uitrekenen. Dit gaat heel gemakkelijk als je het volgende inziet. Een formule is een tautologie als hij logisch equivalent aan een tautologie is. Als formule A dus echt een tautologie is, dan moet je de vergelijking $A = \top$ kunnen afleiden. Iets ingewikkelder, als B een logisch gevolg is van A, dan moet de formule $\neg A \vee B$ een tautologie zijn (dit was immers equivalent aan $A \to B$), en dus moet $\neg A \vee B = \top$ afleidbaar zijn.

Opdracht 2.22 *Laat door afleidingen zien dat de volgende beweringen kloppen.*

1. $\neg(x \wedge \neg x)$ *is een tautologie*
2. $x \vee y \vee (\neg x \wedge \neg y)$ *is een tautologie*
3. $x \wedge y \models x$
4. $x \models x \vee y$
5. $x \wedge (y \vee z) \models (x \wedge y) \vee z$

2.4.7 De Mechanisering van het Redeneren

Met het blootleggen van de basiswetten van een logische rekenkunde zijn we al een heel eind onderweg naar de realisering van de droom van Leibniz, het reduceren van redeneren tot rekenen. Er ontbreekt echter nog één stap, het maken van een machine die deze logische rekenkunde hanteert en logische equivalenties kan toetsen. Wij gaan zo'n machine hier niet ontwerpen, maar we hebben in feite al een heel mechanische methode gezien om logische equivalentie te berekenen: de waarheidstabel. Je kunt een computer vrij makkelijk programmeren om gewoon alle mogelijke situaties te beschouwen, en te kijken of de twee formules in elke situatie dezelfde waarheidswaarde hebben. Zoals we al hebben opgemerkt, als er n verschillende propositieletters in de twee formules voorkomen moet de computer 2^n situaties checken, zodat deze procedure bij wat ingewikkeldere formules veel tijd kost. Het is tegenwoordig niet bekend of er een snellere methode bestaat, maar de meeste informatici geloven dat het niet echt sneller kan.

Ondanks de vrij hoge complexiteit van het probleem van de logische equivalentie lijkt de conclusie dat de droom van Leibniz uiteindelijk is uitgekomen. Met de propositielogica hebben wij een formele logische taal om redeneerproblemen en argumentaties af te beelden, en we hebben een mogelijkheid om de correctheid van deze argumentaties te verifiëren. Maar waarom discussiëren en twisten mensen dan nog steeds? Een reden is dat de logische taal die wij hier hebben gebruikt, de taal van de propositielogica, en heel eenvoudige taal is die niet toestaat om ingewikkeldere argumentaties weer te geven. Neem bijvoorbeeld de redenering:

> Alle mensen zijn sterfelijk, en Socrates is een mens.
>
> Dus Socrates is sterfelijk.

In onze logische taal kunnen we deze redenering alleen met $p \wedge q \models r$ weergeven, maar r is zeker geen logisch gevolg van $p \wedge q$. Dit laat zien dat de taal van de propositielogica te zwak is om sommige intuïtief geldige redenering uit te drukken. Er zijn talen waarin redeneringen van dit type wel goed kunnen worden weergegeven. Het probleem is echter dat er voor deze talen *geen* computerprogramma bestaat dat kan controleren of een formule een tautologie is. In paragraaf 3.4 gaan we hier nader op in.

Hoofdstuk 3

Modellen van Berekening

In het eerste hoofdstuk hebben we kennisgemaakt met allerlei verschillende soorten rekenapparatuur. Dit was nog maar een uiterst bescheiden greep uit wat er allemaal door de geschiedenis heen bedacht is om het rekenen te automatiseren. In dit hoofdstuk willen we laten zien hoe je, zonder je druk te maken over allerlei technische moeilijkheden aangaande de hardware en computerarchitectuur, op een tamelijk eenvoudige en heldere manier kunt begrijpen wat automatische rekenprocessen zijn. Hiervoor gebruiken we abstracte *machinemodellen* waarmee we zicht kunnen krijgen op welke problemen zich – met wat voor middelen dan ook – laten automatiseren.

In het begin van de twintigste eeuw werd door sommige vooruitdenkende wiskundigen nog verondersteld dat uiteindelijk alle rekenproblemen zich in computerprogramma's zouden laten vangen. Wiskundigen houden niet erg van losse veronderstellingen en zij stelden zich dan ook ten doel hun gevoel van optimisme wiskundig te funderen, met behulp van een bewijs. De vraag was natuurlijk hoe je zoiets kon bewijzen: er moest als eerste een wiskundige definitie gegeven worden van wat het betekent dat een probleem automatisch/mechanisch oplosbaar is. In de jaren dertig introduceerde de Britse wiskundige Alan Turing, de hoofdrolspeler in dit hoofdstuk, een voor iedereen bevredigend wiskundige formalisering van wat machinale berekenbaarheid behelst.

> Turing was niet de eerste met zo'n model. De Amerikaanse wiskundige Alonso Church had korte tijd voor Turing al een definitie gegeven die echter wel een stuk ingewikkelder was. Turing heeft overigens bewezen dat zijn definitie op hetzelfde neer kwam als die van Church.

Turing liet vervolgens zien dat er rekenproblemen zijn die niet automatisch kunnen worden opgelost. In dit hoofdstuk zullen we zijn eenvoudige en elegante model, de naar hem genoemde Turing machines, introduceren. We laten zien hoe je ze kan laten rekenen en vervolgens zullen we op een informele manier laten zien waarom er rekenproblemen zijn waar geen Turing machine voor te ontwerpen is.

3.1 Automaten

3.1.1 Simpele Automaten, Keuze-Automaten, Computers

Automaten zijn apparaten die eenmaal in gang gezet *zelf* een bepaalde handeling of serie handelingen uitvoeren. Het in gang zetten van een automaat gebeurt met het geven van een

zogenaamd *startsignaal*. Enkele voorbeelden van automaten: een koekoeksklok (het opwinden is het startsignaal); een cd-speler (het drukken op de daartoe bestemde knop is het startsignaal); een draaiorgel (het opzetten van het 'draaiboek' en het aanzetten van de motor vormen het startsignaal). Merk op dat 'herhaalbaarheid' in de definitie geen rol speelt: tijdbommen en warmtezoekende luchtdoelraketten zijn ook automaten. Een voorbeeld van een apparaat dat *geen* automaat is, is een muziekinstrument: om muziek te krijgen uit een viool moet je voortdurend met het ding in de weer zijn.

Bij een speeldoos of een koekoeksklok gebeurt er na het geven van het startsignaal steeds hetzelfde. Speeldozen en koekoeksklokken zijn voorbeelden van automaten van het allersimpelste type. Andere voorbeelden van simpele automaten zijn dieselmotoren, muizenvallen, stoommachines en stortbakken van wc's.

Naast de simpele automaten hebben we de automaten waarbij een zekere keuzemogelijkheid aanwezig is: de keuze-automaten. Voorbeelden zijn cd-spelers en wasautomaten. Bij een cd-speler kunnen we het soort muziek dat eruit komt beïnvloeden door het selecteren van het schijfje. Een en hetzelfde apparaat kan gebruikt worden om een sonate van Beethoven of een concert van Tina Turner ten gehore te brengen. Hoe het onderscheid tussen simpele automaten en keuze-automaten precies uitvalt is overigens voor een deel een kwestie van perspectief: een cd-speler is een keuze-automaat wanneer we veronderstellen dat er inderdaad meerdere cd's aanwezig zijn om uit te kiezen; een cd-speler samen met een enkel cd-schijfje is een simpele automaat. Een keuze-automaat biedt keuze uit n mogelijkheden, waarbij n een of ander geheel getal is dat groter is dan 1.

Ook een computer is een automaat, maar dan een met een uitzonderlijke mate van flexibiliteit. Waar de simpele automaat geen keuze bood en de keuze-automaat slechts een eindig aantal mogelijkheden had, biedt de computer in principe oneindig veel mogelijke keuzes. De computer dankt deze flexibiliteit aan het feit dat hij *programmeerbaar* is: in principe kunnen we een computer elke denkbare handeling of reeks van handelingen laten uitvoeren door het aanmaken van een toepasselijk programma om de handelingen te 'sturen'. We kunnen de klasse van automaten als volgt schematiseren:

Simpele automaten bieden één mogelijkheid.
Voorbeelden: speeldoos, koekoeksklok, muizenval.

Keuze-automaten bieden eindig veel mogelijkheden.
Voorbeelden: draaiorgel, cd-speler, wasautomaat.

Programmeerbare automaten bieden oneindig veel mogelijkheden.
Voorbeelden: computers ('programmaten') van allerlei slag.

Dat een computer oneindig veel mogelijkheden biedt, geldt overigens alleen voor de gebruiker die in staat is hem te programmeren. Het geldt bijvoorbeeld niet voor mensen die slechts met één programma, zoals een tekstverwerker, hebben leren omgaan. Zulke mensen gebruiken de computer in feite als een simpele automaat.

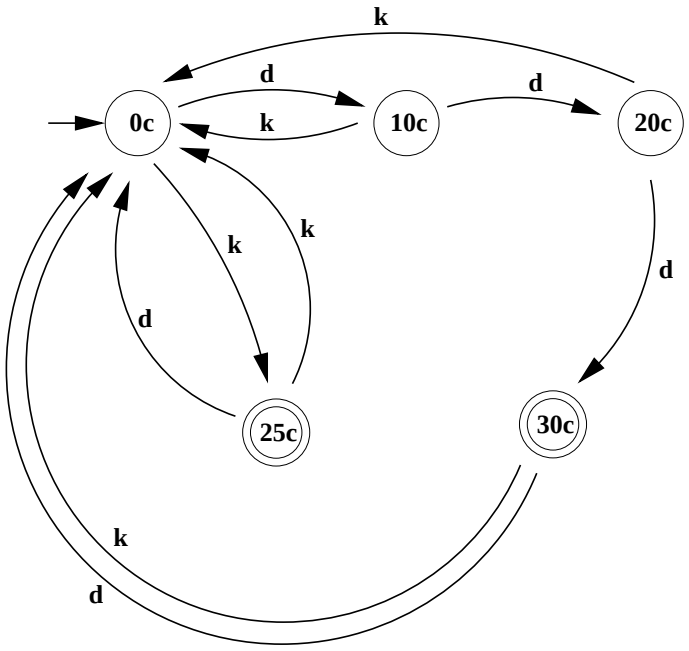
3.1.2 Keuze-automaten

Volgens Turing is een programma niet veel meer dan een rij van instructies die stuk voor stuk een bepaalde machine van een gegeven toestand overbrengen in een nieuwe toestand. Deze overgangen noem je ook wel *transities*. Een simpel voorbeeld hiervan is een koffieautomaat waarbij de

reeks van instructies, het programma, bestaat uit de achtereenvolgens ingeworpen munten. Een voorbeeld hiervan is gegeven in Figuur 3.1. Dit eenvoudige apparaat, dat nog uit het 'gulden' tijdperk stamt, accepteert dubbeltjes en kwartjes: in de tekening aangegeven met respectievelijk `d`-tjes en `k`-tjes. Het rekent 25 en 30 cent als voldoende bedrag om tot koffie uitschenken over te gaan: de *eindtoestanden* van deze automaat, terwijl 0 cent de evidente *begintoestand* is. Als er meer geld dan fl. 0,30 wordt ingeworpen, dan keert deze wat gebruiksonvriendelijke machine al het geld weer uit aan de klant. Hij maakt het zich wat dat betreft eenvoudig, maar hij moet evengoed wel het ingeworpen bedrag onthouden als dat niet meer is dan dertig cent. Kortom, de interne mechaniek moet zo ingericht zijn dat het apparaat onderscheid kan maken tussen vijf verschillende toestanden die de verschillende mogelijke ingevoerde bedragen van elkaar onderscheiden: fl 0.00, fl 0.10, fl 0.20, fl 0.25 en fl 0.30.

Zoals beloofd, stellen we ons in dit hoofdstuk op als wiskundigen. We maken ons dus niet meer druk over de mechanische verwerkelijking van dit apparaat, maar volstaan met het weergeven van de verschillende toestanden met de transities daartussen in een plaatje, de *procesgraaf*, zoals in Figuur 3.1. Dit geeft een volledig en ondubbelzinnig model van de werking van het apparaat dat ons voor ogen staat.

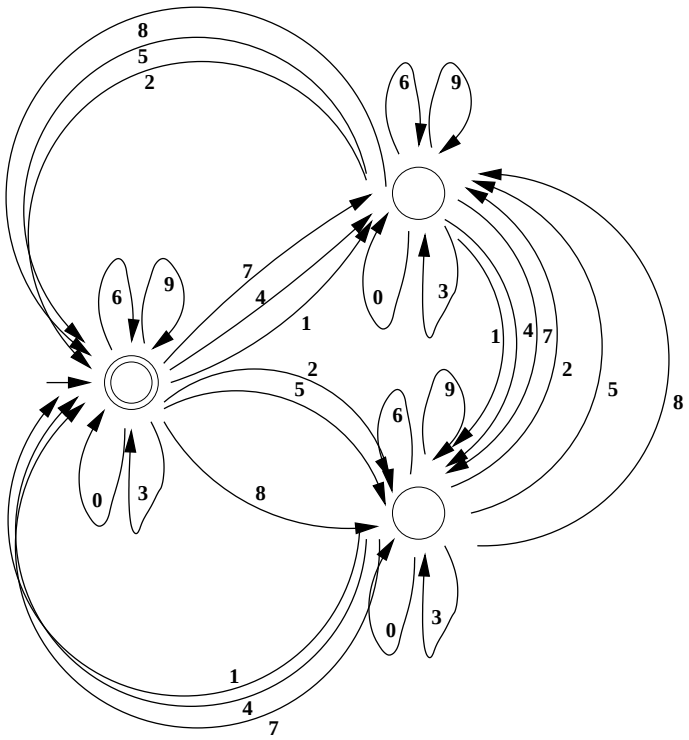
Figuur 3.1: De procesgraaf van een koffieautomaat.



Vanuit elke toestand vertrekt een transitiepijl voor elke mogelijke inworp. Zo brengt een dubbeltje, `d`, je van 10 cent naar 20 cent, terwijl vanuit dezelfde toestand de inworp van een kwartje, `k`, je weer terugzet op 0 cent. We zijn tenslotte over de 30 cent-grens heengegaan: de machine geeft het geld terug en brengt zichzelf weer in de begintoestand.

Een machine die in elke mogelijke toestand voor gegeven invoer gegarandeerd in een unieke nieuwe toestand terechtkomt heet een *deterministische automaat*. Vaak wordt ook gesproken

Figuur 3.2: De 'deelbaar-door-drie-automaat'.



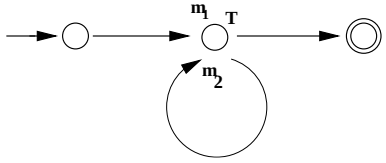
over *eindige* automaten om aan te geven dat het aantal toestanden eindig is. Automaten kunnen bepaalde berekeningen uitvoeren. De automaat uit Figuur 3.1 is daar een voorbeeld van. Deze berekent of er een juiste hoeveelheid geld in kwartjes en dubbeltjes is ingeworpen. Getallen zijn ook rijtjes van mogelijke invoer en dus kan je zo'n automaat ook berekeningen laten uitvoeren op gewone getallen. De automaat in Figuur 3.2 bekijkt of een ingevoerd getal deelbaar is door 3. De toestand met de dubbele cirkel is zowel begin- als eindtoestand. Als we nu bijvoorbeeld het getal 258 invoeren, dan wordt eerst de 2 ingelezen en van de begintoestand schakelt dit apparaatje over naar de toestand rechtsonder. Daarna wordt de 5 ingelezen en wordt naar de toestand rechtsboven overgestapt. Uiteindelijk, na het inlezen van de 8, komen we uit in de eindtoestand en daarmee *accepteert* of *herkent* deze automaat 258 als een getal dat deelbaar is door 3.

Opdracht 3.1 *Merk op dat de drievoud-automaat ook de lege getallenrij beoordeelt als deelbaar door* 3*. Hoe kan je deze automaat wijzigen zodat dit geval afgekeurd wordt?*

Opdracht 3.2 *Als we de drievoud-automaat veranderen door de toestand rechtsboven in Figuur 3.2 eindtoestand te maken, welke getallen worden dan herkend? En welke getallen worden herkend als we de toestand rechtsonder eindtoestand maken?*

Opdracht 3.3 *Van welke prettige eigenschap van drievouden maakt de drievoud-automaat gebruik?*

Figuur 3.3: De dikke lijn representeert het pad dat X aflegt. De originele string overbrugt dezelfde afstand maar met een extra lus met T als begin en einde.



Opdracht 3.4 *Vijfvouden zijn nog makkelijker te herkennen dan drievouden. Teken de procesgraaf van een automaat met twee toestanden die vijfvouden herkent.*

Opdracht 3.5 *Een automaat ontwerpen voor andere veelvouden kan iets lastiger zijn. Ze zijn echter altijd te maken. Teken de procesgraaf van een automaat voor viervoud-herkenning. Hint: Een viervoud van één cijfer moet een* 0*, een* 4 *of een* 8 *zijn. Een viervoud van minstens twee cijfers is altijd te herkennen aan de laatste twee cijfers: als het op een na laatste cijfer even is, dan moet het laatste cijfer een* 0*, een* 4 *of een* 8 *zijn. Als het op een na laatste cijfer oneven is, dan moet het laatste cijfer een* 2 *of een* 6 *zijn.*

Opdracht 3.6 *De mooie eigenschap van drievouden uit Opdracht 3.3 is een toevalligheid van het tientallig stelsel. De onderliggende wetmatigheid is echter niet van het talstelsel afhankelijk. In het unaire stelsel, waar de enige bewerking '+1' is, gedraagt elk getal zich zeer regelmatig. Teken een automaat die zevenvouden in het unaire stelsel herkent.*

3.1.3 Wat Kan een Keuze-automaat niet?

Ondanks het feit dat het automaatje uit Figuur 3.2 maar drie toestanden heeft, kan het onbeperkte invoer aan. Voor een willekeurig lang getal kan het bepalen of het een drievoud is. Dat stemt hoopvol. De vraag is dan ook of we automaten ook andere dingen kunnen laten berekenen, en dan liefst het soort saaie berekeningen die wij ook volledig 'op de automatische piloot' uitvoeren. Dat blijkt helaas niet het geval te zijn. Een automaat is hiervoor te 'vergeetachtig'. Het blijkt bijvoorbeeld niet mogelijk te zijn om een automaat te maken die een willekeurige correcte optelling kan herkennen.

Stel dat er wel zo'n optelautomaat zou zijn die elke uitdrukking van de vorm $getal_1$ + $getal_2$ = $getal_3$ 'nakijkt'. We voeren nu een correcte optelling van twee getallen in, n_1 en n_2, die zo groot zijn dat de lengtes van deze getallen samen groter is dan het aantal toestanden van de automaat. De verwerking van deze optelling zal meer toestandsovergangen vergen dan het aantal toestanden van de automaat, en zodoende moet er tenminste één toestand, zeg T, meerdere keren bezocht worden voordat de eindtoestand bereikt is. Laat m_1 het aantal tekens zijn dat ingelezen is bij het eerste bezoek aan T en m_2 het aantal ingelezen tekens bij de tweede keer. Laat nu X de rij symbolen zijn die we krijgen door uit de originele optelling het stuk van het $m_1 + 1$-ste tot en met het m_2-de teken weg te laten. Als we de invoer X aan dezelfde automaat geven, dan doorloopt deze dezelfde weg door de procesgraaf minus de T-lus. Daarmee wordt ook X door de automaat opgevat als een correcte optelling. Dat oordeel kan echter niet juist zijn. Als in het

weggelaten stuk een = of een + zat, dan is X zelfs niet eens een optelling, en als het weggelaten stuk alleen maar getallen bevat, dan verandert óf de waarde van een van de opgetelde getallen óf het resultaat en moet X dus een incorrecte optelling zijn. Kortom, zo'n automaat kan niet bestaan.

Opdracht 3.7 *Beredeneer op dezelfde manier waarom er geen automaat kan zijn die elke symbolenrij die begint met een aantal* 0*-en gevolgd door eenzelfde aantal* 1*-en herkent (en andere rijen afwijst).*

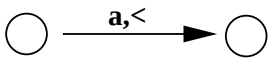
Opdracht 3.8 *Laat op een analoge manier zien dat er geen automaat is die kwadraten in het unaire getallenstelsel herkent (|-rijen van lengte* 0*,* 1*,* 4*,* 9*, etc.).*

3.2 Turing Machines

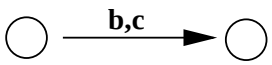
De beperking van een automaat in het nabootsen van rekenmechanismen is dat hij 'vergeet' hoe hij in een bepaalde toestand is gekomen. In het geval maken van een optelling is deze voorgeschiedenis van belang om de correctheid te bepalen, terwijl ze in het geval van de drievoudtest niet van belang is. Als we het getal 258 met de drievoud-automaat beoordelen, dan maakt het na het inlezen van het eerste symbool immers niet uit hoe we in de desbetreffende volgende toestand zijn gekomen. Het eerste symbool had net zo goed een 5 of een 8 kunnen zijn: het eindresultaat is hetzelfde en het antwoord blijft correct: ook 558 en 858 zijn deelbaar door drie.

Turing bedacht dat een machine een soortgelijk hulpmiddel zou moeten hebben als wij gebruiken bij het uitvoeren van berekeningen: *pen* en *papier*. Een Turing machine is een automaat uitgebreid met een oneindige hoeveelheid papier in de vorm van een aan beide kanten doorlopende band of tape die netjes in van elkaar gescheiden vakjes is verdeeld. Hiermee kan het apparaat informatie bijhouden en terugvinden. Met een speciaal mechaniek, de *kop* genoemd (vergelijk de kop van een cassetterecorder), kan het één symbool schrijven in het vakje waar hij staat, over de band heenschuiven in beide richtingen, en de inhoud van het vakje waar het staat ingelezen. Net als met de koppen van een cassetterecorder kun je er dus mee opnemen (op de band schrijven) en afspelen (van de band lezen).

De toestandsovergangen gaan nu vergezeld van een instructie voor de kop. Zo'n instructie kan een bewegingsinstructie zijn: *ga links* of *ga rechts*, of een schrijfinstructie: *schrijf symbool S op de band*, waarbij S een of ander gegeven symbool is.



Zo betekent de transitie in het plaatje hierboven: 'Als door de kop een a gelezen wordt, ga dan naar de volgende toestand en beweeg de kop naar links.'

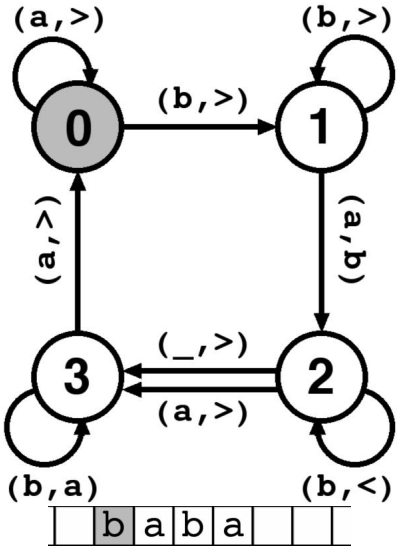


De transitie in het plaatje hierboven bevat een schrijfinstructie: 'Als door de kop een b gelezen wordt, ga dan naar de volgende toestand en schrijf een c op de band, op de plaats waar de kop zich bevindt.'

Een speciaal symbool dat bij Turing machines steeds gebruikt wordt is het liggende streepje _ voor lege invoer. Daarmee kunnen instructies gegeven worden voor situaties waar de kop op een positie staat waar niets te lezen valt. Bovendien kunnen met behulp van _ tekens van de band geveegd worden, door _ eroverheen te schrijven. Bij Turing machines worden geen specifieke eindtoestanden aangeduid. Een belangrijk verschil met automaten is dat niet voor elk gegeven invoer en elke toestand een transitie gegeven is. Zodra de leeskop op een symbool s staat en in de huidige toestand is geen transitie voor s gegeven, dan stopt de machine.

3.2.1 Een Sorterende Turing Machine

Figuur 3.4: Een alfabetisch sorterende Turing machine.



We beginnen met een simpel voorbeeld. In Figuur 3.4 is een Turing machine getekend. Bovenaan herken je de automaatachtige transitiegraaf. Daaronder is de tape getekend waarbij het donkere vakje de stand van de lees-schrijfkop aangeeft. De invoer is een rijtje van a's en b's en de machine moet dat rijtje alfabetisch sorteren: de a's voor de b's. De leeskop leest eerst een b en begint in toestand *0*. Hij volgt de transitie naar toestand *1* en beweegt de kop naar *rechts*. Nu komt hij een a tegen, volgt de transitie *a,b* naar toestand *2* en schrijft braaf een b op de huidige koppositie. De tabel van Figuur 3.5 geeft de gehele uitvoering van het sorteerprocédé. Het dakje ˆ geeft daarbij de stand aan van de lees-schrijfkop.

Opdracht 3.9 *Teken een Turing machine die rijtjes van* a*-tjes en* b*-tjes in zijn geheel een plek naar links over de band verschuift.*

Opdracht 3.10 *Teken een Turing machine die rijtjes van* a*-tjes van oneven lengte verlengen met een* a*-tje (en de overige ongemoeid laat).*

Opdracht 3.11 *Teken een Turing machine die rijtjes van* a*-tjes en* b*-tjes omdraait.*

Figuur 3.5: Tabel van de uitvoering van het sorteerprocédé.

stap	tape/kop	toestand	stap	tape/kop	toestand
0.	$\hat{b}aba$	0	10.	$abb\hat{b}$	2
1.	$b\hat{a}ba$	1	11.	$ab\hat{b}b$	2
2.	$b\hat{b}ba$	2	12.	$a\hat{b}bb$	2
3.	$\hat{b}bba$	2	13.	$\hat{a}bbb$	2
4.	$\hat{}bbba$	2	14.	$a\hat{b}bb$	3
5.	$\hat{b}bba$	3	15.	$a\hat{a}bb$	3
6.	$\hat{a}bba$	3	16.	$aa\hat{b}b$	0
7.	$a\hat{b}ba$	0	17.	$aab\hat{b}$	1
8.	$ab\hat{b}a$	1	18.	$aabb\hat{}$	1
9.	$abb\hat{a}$	1		STOP	

3.2.2 Een Kwadraterende Turing Machine

Van het sorteervoorbeeld hierboven raak je misschien nog niet overtuigd van de rekenkracht van Turing machines. Wellicht dat het volgende voorbeeld wat dat betreft meer tot de verbeelding spreekt: een kwadraterende Turing machine. Zijn interne transitiestructuur is gegeven in Figuur 3.6. Deze machine geeft voor een gegeven rijtje a-tjes van lengte n een a-rijtje van lengte n^2 terug. We gaan hier niet stap voor stap de machine doorlopen, maar geven je een schets van het algoritme. Hierbij wordt gebruikgemaakt van de volgende stelling.

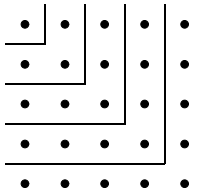
> Voor een gegeven natuurlijk getal n geldt dat zijn kwadraat gelijk is aan de som van de eerste n oneven getallen.

Je kunt dit inzien door $0^2 = 0$, $1^2 = 1$ en $2^2 = 4$ te beschouwen, en op te merken dat het verschil van de verschillen van opvolgende kwadraten constant is (zie hoofdstuk 1). In het geval van n^2 is deze constante gelijk aan 2.

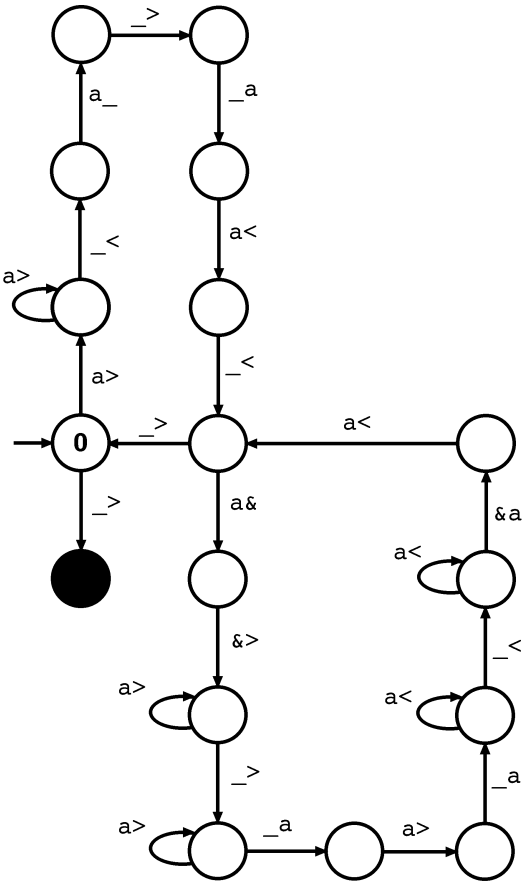
$$\begin{array}{l} 0^2 = 0 \\ 1^2 = 1 \quad \begin{array}{c} 1 \\ 3 \end{array} \quad 2 \\ 2^2 = 4 \end{array}$$

Je kunt al zien dat $2^2 = (1+2)+1 = 3+1 = 4$. Zo verdergaand is $3^2 = (3+2)+3+1 = 5+3+1 = 9$. Voor grotere getallen krijg je zodoende $n^2 = (2n-1) + (2n-3) + \ldots + 3 + 1$.

Opdracht 3.12 *Het volgende plaatje toont ook aan dat de som van de eerste n oneven getallen gelijk is aan n^2. Hoe?*



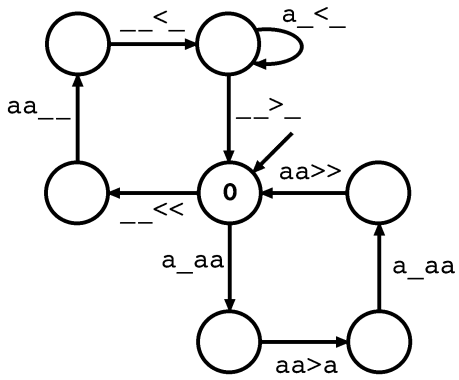
Figuur 3.6: De kwadraterende Turing machine.



De Turing machine uit Figuur 3.6 gebruikt de eerste (bovenste) lus van transities om een rijtje $a \ldots a$ van lengte n te herschrijven tot $a \ldots a \_ a$, wat overeenkomt door het beginrijtje maar dan met een lege plek geplaatst tussen de op een na laatste en de laatste a. Hierna wordt de onderste lus $n-1$ keer doorlopen waarbij steeds twee a-tjes worden geschreven achter de tweede rij van a-tjes. Er wordt met behulp van een hulpsymbool $\&$ steeds bijgehouden welke a in de eerste rij verdubbeld wordt. Dit levert dan uiteindelijk $a \ldots a\ a \ldots a$ als resultaat op de band op, waarbij de eerste rij bestaat uit $n-1$ a-tjes en de tweede uit $2n-1$ a-tjes. Nu wordt de hele procedure herhaald, totdat uiteindelijk geen a-tjes in de beginrij over zijn. Dit geeft aldus:

$$\begin{aligned}(2n-1) + (2(n-1)-1) + (2(n-2)+1) + \ldots + (2(n-(n-1))-1) &= \\ (2n-1) + (2n-3) + (2n-5) + \ldots + 1 &= n^2\end{aligned}$$

Figuur 3.7: Kwadraterende Turing machine voor twee tapes



a-tjes. Hieronder is de hele procedure gegeven voor het eenvoudige geval van vier a-tjes. Elke stap stelt de afwerking voor van een lus.

$\ldots aaaa \ldots$	begin		
$\ldots aaa\_a \ldots$	bovenste lus	$\ldots a\&\_aaaaaaaaaa \ldots$	onderste lus
$\ldots aa\&\_aaa \ldots$	onderste lus	$\ldots \&a\_aaaaaaaaaaaa \ldots$	onderste lus
$\ldots a\&a\_aaaaa \ldots$	onderste lus	$\ldots a\_aaaaaaaaaaaa \ldots$	bovenste lus
$\ldots \&aa\_aaaaaaa \ldots$	onderste lus	$\ldots \&\_aaaaaaaaaaaaaa \ldots$	onderste lus
$\ldots aa\_aaaaaaaa \ldots$	bovenste lus	$\ldots \_aaaaaaaaaaaaaa \ldots$	bovenste lus.

Opdracht 3.13 *Ontwerp een Turing machine die elk rijtje dat begint met* a*-tjes gevolgd door* b*-tjes en dan* c*-tjes aanvult met een* + *als het aantal* a*-tjes en* b*-tjes tezamen gelijk is aan het aantal* c*-tjes.*

3.2.3 Turing Machines met Meerdere Tapes

Zoals je wellicht al gemerkt hebt, kost het de kwadraterende Turing machine nogal wat tijd om zijn klus te klaren. Alleen al voor het kwadrateren van 9 moeten maar liefst 3432 toestandsovergangen gemaakt worden. Heel wat sneller is de Turing machine uit Figuur 3.7 die gebruikmaakt van twee tapes, ieder voorzien van een lees-schrijfkop. Zoals je ziet, zijn de overgangen nu voorzien van meer code. De eerste twee symbolen zijn condities en de laatste twee zijn weer instructies. Bijvoorbeeld $a\_>a$ betekent dat, als de kop van de eerste tape een a leest en de kop op de tweede tape op een lege positie staat, de eerste kop naar rechts moet bewegen en de tweede kop op zijn huidige positie een a moet neerzetten.

Opdracht 3.14 *Leg kort uit hoe de machine uit Figuur 3.2.3 een kwadraat berekent, bij voorbeeld aan de hand van de verwerking van drie* a*'s op de eerste band.*

Frankenstein is een roman uit 1818 van de Britse schrijfster Mary W. Shelley. Het boek waarschuwt tegen het te ver doorvoeren van industriële technologie. Het vertelt van een zekere dr. Frankenstein die in zijn wetenschappelijke enthousiasme het plan heeft opgevat de mens na te bouwen. Het plaatje hiernaast toont het resultaat (volgens de eerste verfilming uit de jaren dertig van de twintigste eeuw). Frankenstein's creatie pakt slecht uit. Ten prooi aan eenzaamheid slaat het nieuwe wezen aan het moorden. Hij eist dan van zijn schepper dat die een monstervrouwtje voor hem maakt. In eerste instantie zwicht de dokter voor zijn eigen moordend gedrocht en gaat aan het werk, maar gaandeweg realiseert hij zich dat een monsterpaartje ook zelf monstertjes kan gaan maken. Hij staakt zijn werkzaamheden, en de woede van het monster richt zich vervolgens op de arme dokter. Probeer de film een keer te gaan zien of de video te pakken te krijgen.

3.2.4 Turings These

Turing machines met dubbele tapes mogen dan een stuk handiger en sneller zijn dan de uitvoering met een enkele tape, in principe is er geen verschil in rekenkracht: voor elke Turing machine met dubbele tape kan er een monoversie gemaakt worden die precies hetzelfde resultaat berekent. Ook verdere uitbreidingen zoals Turing machines met een tweedimensionale tape waarover de kop ook verticaal kan bewegen leveren geen extra rekenkracht op. Sterker nog, er is in de vijfenzestig jaar na Turings uitvinding geen enkele machine bedacht die iets kan berekenen dat je niet ook op een geschikte Turing machine met een enkele tape kunt berekenen. Turing had dit zelf al voorzien: hij had het sterke vermoeden geopperd dat *elke rekenmachine* in essentie teruggebracht kan worden tot een Turing machine met één tape.

De juistheid van zo'n vermoeden is niet te bewijzen, je weet immers nooit wat voor machines er nog eens door mensen worden bedacht. Hoe dan ook, tot op de dag van vandaag is er niets bedacht dat de rekenkracht van een Turing machine essentieel te boven gaat. Wiskundigen en informatici zijn het er wel over eens dat de simpele Turing machine met één band, ondanks zijn inefficiëntie, ons een eenvoudig hanteerbaar model geeft om te bepalen welke problemen zich mechanisch laten oplossen.

Om deze vraag was het Turing ook te doen. Hij wou met zijn machines aantonen dat er problemen zijn die *niet* mechanisch beslisbaar of berekenbaar zijn. Hiervan geven we een paar voorbeelden in 3.4.1.

3.2.5 Turings Test

Turings scherpe begrenzing van het mechaniseerbare riep direct de aloude gedachte aan 'Frankenstein' op: 'Is het menselijke denken mechaniseerbaar? Turing publiceerde in 1950 een artikel in het filosofische tijdschrift *Mind*: 'Computing Machinery and Intelligence'. Dit artikel is een filosofische klassieker geworden. Turing voorspelt hierin dat mettertijd – Turing zegt zelf binnen 50 jaar – computers inderdaad zullen kunnen denken. Om het denkvermogen van een machine vast te kunnen stellen bedenkt Turing een truc.

Om zijn idee in te leiden bespreekt Turing eerst het volgende spel. Het simulatiespel is een spel voor drie personen, een man (M), een vrouw (V) en een vragensteller. De vragensteller zit in een andere kamer dan de man en de vrouw. Zijn opdracht is om door slimme vragen erachter te komen wie van de twee de man is en wie de vrouw. Hij weet aanvankelijk alleen dat hij te maken heeft met een persoon X en een persoon Y, een van de twee een man en de ander een vrouw. Na afloop van het spel moet hij kunnen zeggen wie X is en wie Y. M en V hebben elk ook een opdracht. Een van de twee (laten we aannemen dat het M is) heeft als opdracht om de vragensteller te misleiden; de ander (in ons geval dus V) heeft de opdracht om de vragensteller juist te helpen.

Als de vragensteller de stemmen van X en Y zou horen zou hij allicht de vrouwenstem van de mannenstem kunnen onderscheiden, en dat is flauw. Om het spel interessant te maken verloopt het contact daarom via een beeldscherm met toetsenbord. Als je het spel zelf wilt spelen en je vindt deze opzet te ingewikkeld, dan kun je ook een vierde persoon inschakelen die de antwoorden overbrengt. De vragensteller vraagt bij voorbeeld aan X: 'Wat voor haar heb je?', en krijgt dan het antwoord: 'Ik heb blonde vlechten, en de langste vlecht is zo'n 30 centimer lang.' De vragensteller weet dat dit antwoord zowel van M (die misleidt) als van V (die de waarheid spreekt) kan komen, en moet met slimmere vragen komen. Hoe hij ook wikt en weegt, hij zal nooit volledig overtuigende informatie inwinnen waarmee hij de man van de vrouw kan onderscheiden.

De Turing test heeft betrekking op de mens-machine versie van dit spelletje, waarbij de mens de rol van de hulpvaardige vrouw uit het bovenstaande spel speelt en de machine die van de misleidende man. Als het de meest pientere vragensteller niet lukt de mens van de machine te onderscheiden, dan mag volgens Turing vastgesteld worden dat de machine denkend door het leven gaat. Turing stelde overigens wel de eis dat de ondervraging zonder fysieke confrontatie zou moeten blijven.

> 'De vraag- en antwoord methode lijkt me geschikt om vrijwel elk gebied van menselijke activiteit te introduceren dat we willen bestrijken. We willen de machine geen strafpunten geven omdat het ding het er slecht zou afbrengen in een schoonheidswedstrijd, net zomin als we de mens willen afstraffen voor het feit dat hij het bij het hardlopen verliest van een vliegmachine. De omstandigheden van ons spel maken die handicaps irrelevant. De "getuigen" kunnen opscheppen wat ze willen over hun charmes, hun kunnen en hun wapenfeiten; de vragensteller mag geen praktische proef op de som eisen.'

Opdracht 3.15 *Inmiddels zijn de vijftig jaar voorbij die Turing als termijn gesteld had voor het volwassen worden van de computer. Zijn voorspelling is niet helemaal uitgekomen: computers*

vallen nog steeds door de mand in de Turing test. Wat denk je: is dit een kwestie van tijd, of zal het er nooit van komen? Beargumenteer je standpunt zo zorgvuldig mogelijk.

3.3 Programmeren

3.3.1 Registermachines

Figuur 3.8: De code van een kwadraterende registermachine.

0.	*x?*	*1*	*12*	*6.*	*z+*	*7*	
1.	*x-*	*2*		*7.*	*z+*	*8*	
2.	*z+*	*3*		*8.*	*x?*	*4*	*9*
3.	*x?*	*4*	*12*	*9.*	*y?*	*10*	*0*
4.	*x-*	*5*		*10.*	*y-*	*11*	
5.	*y+*	*6*		*11.*	*z+*	*9*	

De registermachine is een alternatief model voor mechanische berekening. Het levert niet meer rekenkracht, maar het model staat wat dichter bij de rekenmachines uit het eerste hoofdstuk. De aansturing lijkt op wat we vandaag de dag een computerprogramma noemen. Natuurlijk ontbeert zo'n simpel model alle technische toeters en bellen van de moderne computer: als we het aan het werk zien doet het meer denken aan een telraam dan aan de geavanceerde apparaten op school, thuis of op kantoor. Bij het redeneren over wat zo'n machine kan en wat niet is de eenvoud natuurlijk een voordeel.

Net zoals automaten en Turing machines verspringen de registermachines van toestand naar toestand. Alleen nu noemen we deze toestanden *programma-regelnummers*, om aan te geven waar de uitvoerende machine zich op een gegeven moment in het programma bevindt. De regels zelf bestaan uit programmacode geschreven in de bijzonder simpele taal *RePro*. Op elke regel staat een rekeninstructie of een test.

Een instructie bestaat uit de naam van een specifiek register (geheugenadres) gevolgd door een $+$ of een $-$ om aan te geven dat de machine 1 moet toevoegen of weg moet nemen uit het genoemde register. De laatste operatie heeft geen effect indien het register leeg is. Na de instructie volgt nog het regelnummer waar de machine vervolgens naartoe moet. Bijvoorbeeld *x+ 125* betekent dat de machine 1 moet toevoegen aan het register x en vervolgens naar regel *125* moet gaan. Een extra instructie is de *halt* instructie die ervoor moet zorgen dat de machine stopt. Tests bestaan wederom uit een registernaam met een vraagteken plus twee regelnummers met een verticale streep ertussen. De test betekent dat de machine moet controleren of het betreffende register iets bevat. Zo ja, dan gaat hij naar het eerstgenoemde regelnummer; in het andere geval gaat hij naar het tweede regelnummer.

Begin van de berekening van 3^2.

x	y	z	regel
●●●			0
●●●			1
●●			2
●●		●	3
●●		●	4
●		●	5
●	●	●	6
●	●	●●	7
●	●	●●●	8
●	●	●●●	4
●	●	●●●	5
	●●	●●●	6
	●●	●●●●	7
	●●	●●●●●	8
	●●	●●●●●	9
	●●	●●●●●	10
	●	●●●●●	11
●	●	●●●●●	9
●	●	●●●●●	10
●		●●●●●	11
●●		●●●●●	9
●●		●●●●●	0

Een voorbeeld *RePro*-programma is afgebeeld in Figuur 3.8. Deze Figuur bevat de code om het getal dat is opgeslagen in register x te kwadrateren en het resultaat vervolgens op te slaan in register z. De tabel hiernaast geeft het begin van de uitvoering van het programma, voor het geval waarin aan het begin van het programma x drie eenheden bevat en y en z leeg zijn. In de laatste stap die in de tabel wordt getoond, wordt het programma weer op regel *0* gezet, zodat de hele procedure zich zal herhalen. Het tussenresultaat is nu dat x twee eenheden bevat, y leeg is en z vijf eenheden bevat. Bij de verdere verwerking zullen de twee eenheden in x ook nog worden gekwadrateerd, en daarmee zal het eindresultaat in z inderdaad gelijk worden aan $2^2 + 5 = 9$.

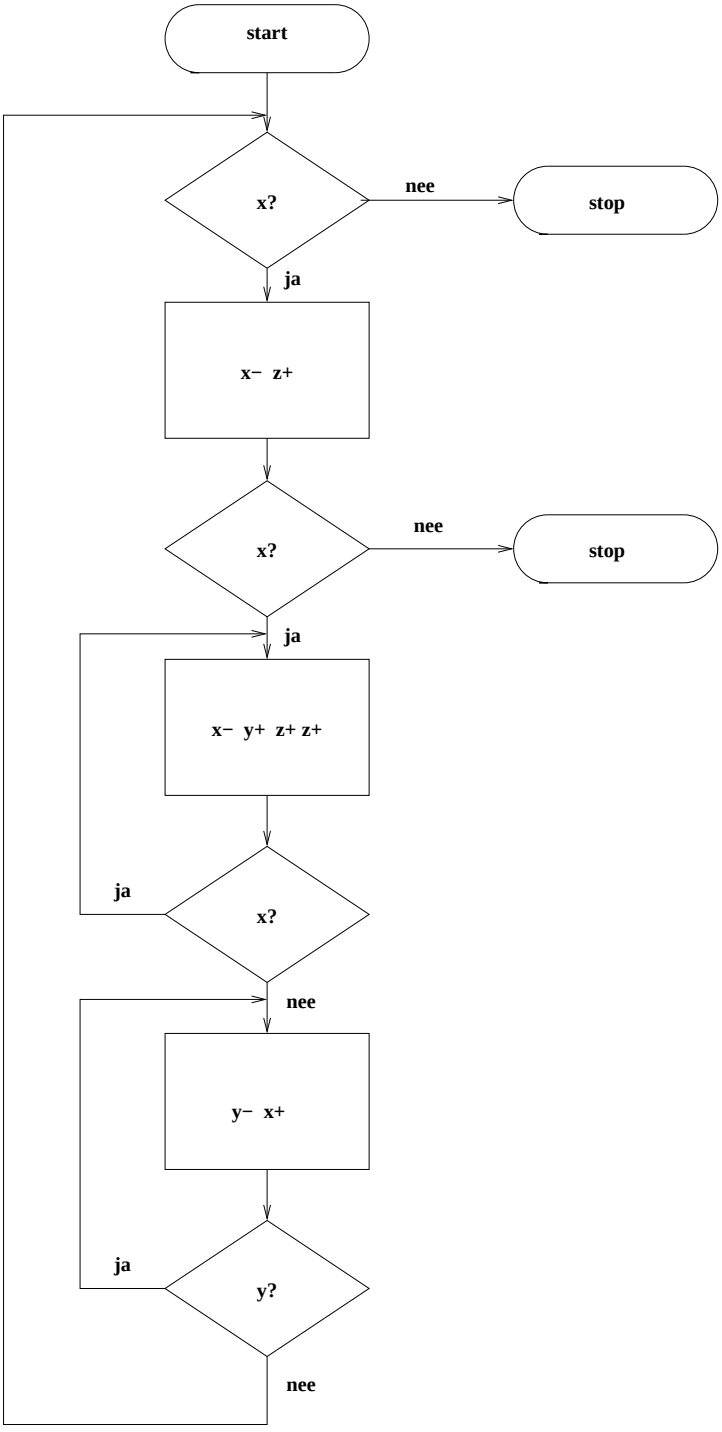
De registermachine waarop het *RePro*-programma uit Figuur 3.8 draait, werkt vrijwel op dezelfde manier als de kwadraterende Turing machine uit Figuur 3.6. Er is dan ook geen wezenlijk verschil in rekenkracht tussen Turing machines en registermachines: elk probleem dat berekenbaar is op een Turing machine is ook berekenbaar met een registermachine, en andersom. Omdat Turing machines meer symbolen aankunnen, is het wel noodzakelijk om problemen te hercoderen in een unair talstelsel voor je ze met een registermachine te lijf kunt gaan.

3.3.2 Stroomdiagrammen en Structuurdiagrammen

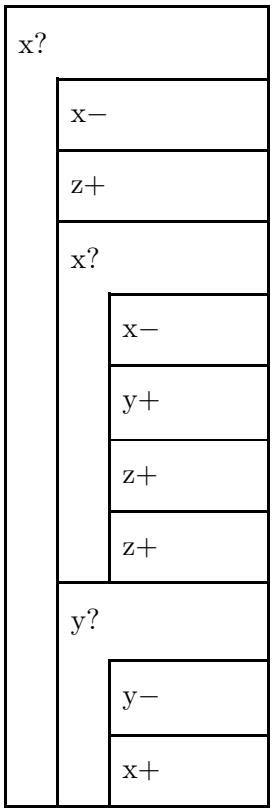
De code van een *RePro*-programma is niet zo gemakkelijk te doorgronden, en het is dus niet echt gemakkelijk om *RePro*-programma's te schrijven. Een *RePro*-programma staat in feite nog (te) dicht bij de operaties die de machine daadwerkelijk uitvoert. Daarom zijn er voor mensen beter hanteerbare programmeertalen bedacht om het leven van de programmeur te vergemakkelijken.

Als we het programma van Figuur 3.8 bekijken dan is niet in één oogopslag duidelijk hoe het algoritme werkt. Met name de expliciete regeltransities maken de code bijzonder lastig te doorgronden. Wat inzichtelijker wordt de zaak als we het algoritme schematiseren in een zogenaamd *stroomdiagram* zoals in Figuur 3.9. De rekeninstructies zijn in de rechthoeken als zogenaamde *transitie labels* weergegeven. De tests verschijnen in de ruitjes. Elke ruit heeft twee uitgangen: een 'ja' uitgang voor het geval dat de test slaagt en een 'nee' uitgang voor het geval dat hij faalt. De 'stop' doosjes representeren de eindtoestanden.

Figuur 3.9: Stroomdiagram voor het kwadratenalgoritme.



PSD — kwadratenalgoritme



Nog duidelijker zijn programmastructuurdiagrammen of PSD's. Zie de Figuur hiernaast. Een winkelhaak met een rechthoek geeft een herhalingslus, met de *voorwaarde* waaronder herhaald wordt in de linkerbovenhoek van de winkelhaak, en de instructies die herhaald worden in de rechthoek rechts onder.

3.3.3 Herhalingslussen en Keuzes

Een belangrijk onderdeel van de stroomdiagrammen zijn de lussen die staan voor herhaling van operaties. In een moderne programmeertaal worden deze als zogenaamde *while-lussen* geschreven (Engels voor lus: *loop*). Een programma-instructie *while (T)* { *P* } betekent dat het programma de instructie *P* moet blijven uitvoeren totdat de test *T* faalt. Indien *T* direct faalt bij het uitvoeren van deze regel, dan wordt *P* dus geen enkele maal uitgevoerd. Als we nu het registerprogramma van Figuur 3.8 noteren met behulp van dergelijke lusinstructies, dan krijgen we:

```
while (x?) { x-;z+;
             while (x?) { x-;y+;z+;z+ };
             while (y?) { y-;x+ }
           }
```

Je ziet ook het symbool *;* opduiken. Dit wordt gebruikt om instructies opeenvolgend te laten uitvoeren: *x+; y-* betekent: 'hoog *x* eerst op met 1 en verminder daarna *y* met 1'. We noemen

; het symbool voor *aaneenrijging*. We hebben op deze manier geen expliciete regelnummering nodig.

Naast de *while*-lussen gebruikt een moderne programmeertaal de simpelere *if*-constructie voor keuze die zich niet als een lus gedragen. Deze constructie heet ook wel *selectie* en heeft als algemeen formaat:

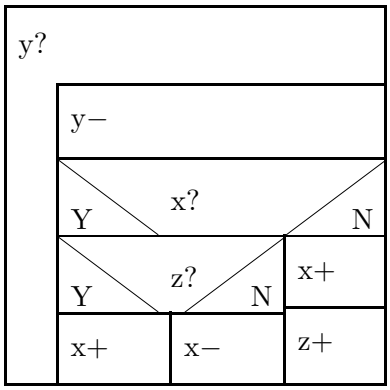
```
if (T) { P } else { Q }.
```

Het programma voert de instructie P uit indien de test T slaagt. In het geval dat C faalt, dan wordt Q uitgevoerd. Zouden we deze constructies aan *RePro* toevoegen, dan hebben we in principe geen expliciete regeltransities meer nodig, en daarmee is ook de nummering van regels overbodig geworden. Een programma stopt gewoon als het aan het einde van het programma gekomen is. Onze nieuwe programmeertaal noemen we *ImPro*. Deze stelt ons in staat om direct een stroomdiagram van een telraamberekening als programma op te schrijven.

Hieronder volgt een voorbeeld dat zowel een herhalingslus als een aantal selecties bevat. Het *RePro*-programma berekent het verschil tussen twee getallen x en y, ofwel de uitkomst van $|x - y|$, in z.

```
0.  y?  1  | 8
1.  y-  2
2.  x?  3  | 6
3.  z?  4  | 5
4.  x+  0
5.  x-  0
6.  x+  7
7.  z+  0
```

PSD — verschilalgoritme



Een PSD voor dit programma zie je hiernaast. De selectiestructuur is een rechthoek met een conditie in het midden en de code voor de twee mogelijke uitkomsten, 'ja' en 'nee', links en rechts daaronder. Het PSD leidt vervolgens gemakkelijk tot een *ImPro*-programma. Zie hieronder.

```
while (y?) {
              y-;
              if (x?) { if (z?) {x+} else {x-} }
              else    {x+;z+}
           }
```

Opdracht 3.16 *Dit programma werkt alleen goed als we beginnen met een leeg* z*-register. Kun je uitleggen waarvoor dit register gebruikt wordt in de programma's hierboven?*

Opdracht 3.17 *In veel programma's kom je ook de enkele* if*-constructie tegen:* if (T) {P}*. Het voert* P *uit als* T *slaagt, en doet anders niks. Hoe schrijf je* ImPro*-code die hetzelfde doet?*

Opdracht 3.18 *Net zo bestaan er constructies voor het geval van meer dan twee keuzes, bijvoorbeeld* if (T) {P} else if (S) {Q} else {R}*. Hoe doe je dit in* ImPro *na?*

Opdracht 3.19 *De constructie* repeat P until T *betekent:*

voer P *uit totdat* T *geldt.*

Hoe definieer je deze constructie in ImPro*?*

3.3.4 Functies, Toekenningen en Tests

Met *while-* en *if*-constructies wordt de transitiestructuur van een programma duidelijk. Toch blijft *ImPro* een beperkte taal, voornamelijk doordat we slechts over twee bijzonder simpele operaties beschikken en maar één soort test kunnen uitvoeren. Het zou een stuk handiger zijn als we complexere operaties die we vaak gebruiken slechts éénmaal hoeven te definiëren om ze vervolgens naar believen te kunnen aanroepen. Zo kwam je bij de *RePro-* en *ImPro*-opgaven reeds twee operaties tegen die je steeds weer moet implementeren: het legen en het kopiëren van een register. In de uitgebreide taal *ImPro+* kunnen we deze operaties in één keer vastleggen.

```
[EMPTY]() { while (Y) { Y- } }

[COPY](.) { Y := EMPTY(); while (X1?) { X1-; Y+ } }
```

EMPTY en *COPY* zijn functienamen waarmee in het vervolg van het programma het geassocieerde stukje programmatuur steeds weer kan worden aangeroepen. Met de vierkante haken geven we aan dat we een functie implementeren. Tussen de haakjes na de functieaanhef wordt met puntjes het aantal registers/variabelen gegeven waarop de functie opereert. Dit zijn de zogenaamde *argumenten* van de functie. Zo heeft de functie *EMPTY* helemaal geen invoer nodig, terwijl *COPY* één register als invoer neemt om deze vervolgens te kopiëren.

Een functie roepen we aan met behulp een *toekenning*: *x := FUNCTIENAAM(x1,...,xn).* Het resultaat van de functie toegepast op de gespecificeerde invoer *x1,...,xn* wordt toegekend aan *x*, ofwel in het *x*-register wordt de waarde van het resultaat opgeslagen. Dit laatstgenoemde register verliest daarmee evenwel zijn oude inhoud. Zo staat in de implementatie van *COPY* al een toekenning *Y := EMPTY()*. De variabele *Y* krijgt de waarde van het resultaat van de toepassing van de *EMPTY*-functie, dat wil zeggen: *Y* wordt geleegd.

De variabelen binnen de functie-implementatie worden bij afspraak met hoofdletters geschreven. Verder gebruiken we de letter *Y* steeds voor het resultaat, de uitvoer, van de functie. De variabelen *X1 ... Xn* worden gebruikt als referentie van de opeenvolgende invoerargumenten. Variabelen die met een andere hoofdletter geschreven worden bij een functie-implementatie zijn hulpvariabelen. Variabelen buiten de functies schrijven we met een kleine beginletter.

Hieronder tref je een uitbreiding van het kleine programmaatje hierboven aan. We definiëren optellen met behulp van een functie *PLUS* om vervolgens af te sluiten met het kwadratenalgoritme als hoofdonderdeel van het programma.

```
[PLUS](..) {
   Y := COPY(X1);
   while (X2?) { X2-; Y+ }
}

z := EMPTY();
while (x?) {
      x-;z+
      z := PLUS(z,PLUS(x,x))
}
```

De machine begint zijn berekening bij de eerste regel van het hoofdonderdeel en gebruikt de functie bij de aanroep. Functies hebben hun eigen gereserveerde registers. Als bijvoorbeeld *PLUS(x,x)* wordt aangeroepen, dan wordt eerst de waarde van *x* naar *X1* en *X2* gekopieerd en nadat de functie toegepast is, wordt het resultaat teruggegeven aan de variabele die de waarde *PLUS(x,x)* toegekend is. In dit geval veranderen de waarden van *X1* en *X2* tijdens de uitvoering van *PLUS*, maar dat heeft geen effect op de waarde van *x*, die blijft ongemoeid.

Opdracht 3.20 *Schrijf een* ImPro+*-programma met een functie* TIMES *voor vermenigvuldiging en completeer het programma, zodat voor twee registers* x *en* y *in een derde register* z *de waarde* x^y *wordt berekend.*

Functies kunnen we ook gebruiken om de testmogelijkheden van *ImPro* uit te breiden. Bij het implementeren van lussen en keuzes willen we vaak meer testen dan het leeg zijn van een register. Uitgebreide tests worden in *ImPro+* eigenlijk identiek aan functies gedefinieerd, alleen nu gaat het niet om de precieze inhoud van het resultaat in het uitvoerregister *Y*, maar over de vraag of dit register wel of niet inhoud heeft. Hieronder is een voorbeeld gegeven van een test waarbij nagegaan wordt of het eerste argument groter is dan het tweede argument.

```
[LARGER](..) {
   Y := EMPTY();
   while (X1?) {
      if (X2?) { X1-; X2- }
      else     { Y+;  X1 := EMPTY() }
   }
}
```

We zeggen dat de test slaagt indien Y niet leeg is na toepassing, en faalt indien Y leeg is. De volgende functie *MODULO* berekent de rest van het eerste argument na deling door het tweede met het behulp van een *LARGER*-test.

```
[MODULO] (..) {
   if ( LARGER(X2,X1)? ) {
      Y := COPY(X1);
   } else {
      Z := COPY(X2);
      while (Z?) { X1-; Z- };
      Y := MODULO(X1,X2);
   }
}
```

Opdracht 3.21 *De* MODULO*-functie werkt niet helemaal optimaal. Als het tweede argument leeg is krijgen we een onbevredigend resultaat. Wat gebeurt er? Hoe moeten we deze functie aanpassen als we een leeg resultaat willen krijgen voor het geval dat het tweede argument leeg is?*

Opdracht 3.22 *Schrijf een test* DIV *met twee invoerargumenten op basis van de functie* MODULO *die slaagt als het eerste argument een deler is van het tweede.*

Opdracht 3.23 *Schrijf een* ImPro+*-programma, gebruikmakend van de test* DIV*, dat berekent of een register een priemgetal representeert.*

Het lijkt erop dat het nieuwe kwadratenprogramma langer is geworden dan de *ImPro*-versie. De *ImPro+*-versie heeft echter twee belangrijke voordelen: het programma is opgedeeld in duidelijke deelprogramma's, en bovendien kunnen de gemaakte functies voortaan gebruikt worden in andere programma's.

In de praktijk maken programmeertalen gebruik van een bibliotheek van standaardfuncties die onzichtbaar voor de gebruiker wordt aangeroepen. Er wordt dan gebruikgemaakt van korte 'infix'-notaties (operatienaam tussen de argumenten) zoals $x+y$, $x*y$, $x\%y$ en $x>y$ in plaats van respectievelijk *PLUS(x,y)*, *TIMES(x,y)*, *MODULO(x,y)* en *LARGER(x,y)*. Vaak worden ook gespecialiseerde functie-bibliotheken bijgeleverd, met allerlei handige functies voor specifieke toepassingen. $y :=$ *COPY(x)* wordt doorgaans gewoon als $y := x$ geschreven, en in plaats van

y := EMPTY() schrijven we kortweg *y := 0*. Ook voor andere gehele getallen wordt gewoon de gebruikelijke decimale notatie gebruikt.

Ook Boolese connectieven vind je als functies terug in programmeertalen. De negatie bijvoorbeeld kunnen we simpel definiëren, als volgt.

```
[NOT](.) { if (X1?)  { Y := EMPTY() } else { Y := EMPTY(); Y+ }
```

In plaats van *NOT(x)* wordt vaak *!x* of $\sim x$ geschreven. Zo kun je een $\leq$-test maken door *NOT* en *LARGER* te combineren: *NOT(LARGER(x,y))*.

Opdracht 3.24 *Schrijf functies voor conjunctie en disjunctie (notatie vaak* x&y *en* x|y*, of,* x&&y *en* x||y*).*

Opdracht 3.25 *Schrijf een gelijkheidstest* EQUALS *(notatie vaak* x=y *of* x == y*) met behulp van* LARGER*.*

3.3.5 Datatypen

De registermachines werken slechts met één soort register waarvan de inhoud een niet-negatief geheel getal representeert. In de werkelijkheid zijn dit rijen van bits die die getallen op een binaire manier coderen. Voor de uitkomst van een test hebben we niet een volledig register nodig maar slechts een enkele bit. In de meeste programmeertalen wordt dan ook van de programmeur verwacht dat deze vertelt wanneer hij zo'n minimaal register nodig heeft. Dit verloopt via het typeren van een variabele. Voordat ermee gerekend wordt moet een variabele getypeerd worden: zo vind je vaak expressies van de vorm *x boolean* om aan te geven dat er slechts een bit voor *x* gereserveerd hoeft te worden. Zo vind je ook vaak de typering *x integer* om aan te geven dat *x* als een geheel getal verrekend moet worden dat eventueel negatief is. Deze zou in een binaire registermachine behandeld kunnen worden als een register met een extra bit om het teken te coderen: *0* voor negatief en *1* voor niet-negatief. Niet-gehele getallen kunnen we representeren met twee registers waarvan er één de positie van de komma aangeeft en de ander het getal dat we zouden krijgen als we de komma zouden vergeten.

Om een goede afwikkeling van functietoepassing te bewerkstelligen wordt bij de definitie van een functie ook verwacht van de programmeur dat hij aangeeft wat de types van de invoerargumenten zijn en natuurlijk ook wat het type van het resultaat zal zijn. Je kunt in een programma iets van de volgende vorm tegenkomen:

boolean [FUN](integer real)

Hierbij wordt aangegeven dat de functie *FUN* een Boolese resultaat (bit) geeft als zij aangeroepen wordt met als eerste argument een geheel getal (opgeslagen in een enkel register, met behulp van binair complement representatie) en als tweede argument een niet-geheel getal (opgeslagen in twee registers, 'eén voor de plaats van de komma, en één voor het getal zonder komma).

Voor het gemak maken wij hier alleen verschil tussen een bit en een volledig register, maar in de praktijk worden verschillende maten registers gebruikt. Zo heb je voor plaatsing van de

komma in een niet-geheel getal geen lang register nodig, en hetzelfde geldt voor het representeren van de toetsenbordsymbolen (Engels: *characters*, vaak afgekort als *char*). In veel programmeertalen is er ook gelegenheid voor de programmeur om zelf types te construeren om objecten van allerlei slag te definiëren. In programmeertalen als C en Java zijn de mogelijkheden hiertoe vrijwel onbeperkt.

3.4 Rekentijden

3.4.1 Berekenbaarheid

Toen Turing zijn algemene model van een rekenmachine ontwikkelde was zijn bedoeling om te laten zien dat *hoe algemeen je je begrip 'berekenbaar' ook kiest*, er altijd vragen zijn waarop een rekenmachine het antwoord schuldig moet blijven. Een beroemd voorbeeld is het *stop probleem* (Engels: *halting problem*). Hierbij gaat het om de vraag of er een programma geschreven kan worden dat kan bepalen of een ander gegeven programma eindigt bij uitvoering. Van sommige programmatuur, zoals klokken of besturingssystemen, willen we dat ze niet stoppen, maar de meeste programma's moeten juist *niet* 'hangen', en wekken ergernis wanneer ze dat onverhoopt *wel* doen. Een programma dat kan nagaan of een ander programma zou kunnen gaan 'hangen' zou behoorlijk wat geld opleveren. Ons standaard-voorbeeld van een programma dat hangt is het volgende:

```
x+; while (x?) {x+}
```

Om de stopvraag te preciseren: er wordt gevraagd om een programma *HALT*, dat bij invoer van een ander programma P als antwoord 'ja' geeft indien P eindigt en 'nee' indien P niet eindigt als P begint met alleen maar lege registers. We moeten hiertoe programma's coderen volgens een of andere vaste methode. Dat kan op een eenvoudige manier. In een programmeertaal zoals *RePro*, *ImPro* en *ImPro+* gebruiken we maar een eindig aantal verschillende symbolen. Zeg dat de taal n verschillende symbolen gebruikt. Dan kan elk programma in zo'n taal dus opgevat worden als een getal gerepresenteerd in het n-tallig stelsel.

HALT moet nu werken zoals een *ImPro+*-test: het geeft voor een stoppend programma gecodeerd in een vast gekozen invoerregister x een niet-lege inhoud op een eveneens vast gekozen uitvoerregister y. Stel dat we zo'n programma in *ImPro+* hebben. We noemen het programma *HALT*, en we schrijven vervolgens de volgende uitbreiding van *HALT*, het programma Q.

```
x := q;  HALT; if (y?) { z+; while (z?) { z+ } } else { z+;z- }
```

Hierbij is q de getalscode van het programma Q.

Opdracht 3.26 *Leg uit waarom we een fout resultaat krijgen wanneer we* HALT *uitvoeren op het programma* Q.

Kortom, een programma *HALT* dat voor elk programma bepaalt of het stopt of niet, startend met lege registers, kan niet bestaan. Wat hier direct uit volgt, en wat van groot belang is om te weten, is dat niet alles berekenbaar is.

Misschien vind je het Q programma nogal vreemd, omdat er in Q een verwijzing zit naar het programma zelf, in de vorm van de getalscode q. Er zijn echter ook problemen die er op het eerste oog veel onschuldiger uitzien, en die eveneens niet berekenbaar zijn. Een bekend voorbeeld is het volgende. Stel we hebben te maken met twee verschillende manieren om een eindig aantal symbolen met 0-en en 1-en te coderen. Bijvoorbeeld:

	a	*b*	*c*	*d*
eerste codering	*11*	*01*	*01110*	*1001*
tweede codering	*101*	*011*	*1010*	*01*

De vraag is nu of er een rijtje symbolen te geven is dat na codering volgens de twee verschillende methoden toch dezelfde rij van *0*-en *1*-en oplevert. In het geval hierboven kunnen we het positief beantwoorden, want *babcd* geeft voor beide coderingen hetzelfde resultaat, namelijk *011101011101001*. De vraag is nu of er een programma te schrijven valt dat voor een willekeurig alfabet van symbolen en voor twee willekeurige binaire coderingen voor dat alfabet berekent of er een gemeenschappelijk rijtje zoals in het geval hierboven te geven is. Het antwoord op die vraag is 'nee', en daarmee is dit zogenaamde *correspondentieprobleem van Post* (genoemd naar de logicus die het probleem verzonnen heeft) een onberekenbaar probleem. Als het gaat om het stopgedrag van programma's moeten mensen het trouwens ook nog wel eens laten afweten. Neem het volgende programma.

```
while (!(x = 1)) {
  if (x % 2 = 0) { x := x/2 }
  else { x := 3*x + 1 }
}
```

Starten met bijvoorbeeld $x = 9$ levert de rij

$$9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 4, 2, 1$$

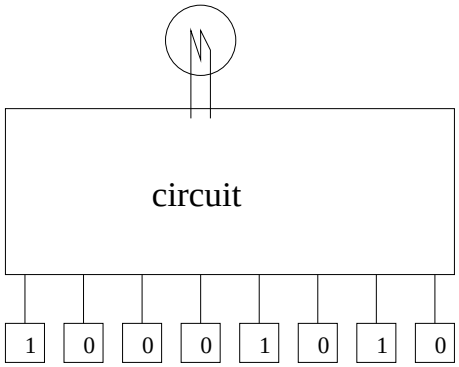
op. Indien dit programma voor verschillende positieve gehele getallen x wordt uitgevoerd blijkt het steeds te eindigen (soms met verrassend hoge tussenwaarden). Echter, tot op heden is er niemand in geslaagd te *bewijzen* dat dit programma altijd eindigt.

Opdracht 3.27 *Voor de regenachtige zaterdagmiddag: probeer dit programma eens na te rekenen voor $x = 27$ (een rekenmachine is handig).*

3.4.2 Tijdscomplexiteit

Ook al is een probleem berekenbaar volgens de Turing these, dan nog kan het computationeel onhanteerbaar zijn. De complexiteit van een probleem kan dusdanig zijn dat er alleen maar programma's te schrijven zijn die wel stoppen, maar pas na een enorme lange wachttijd. In de

Figuur 3.10: Het vervulbaarheidsprobleem: is het circuit zo dat het lampje bij de uitgang gaat branden voor een bepaalde stand van de bits, of blijft het altijd uit?



praktijk wordt de meetlat dan ook lager gelegd dan de berekenbaarheidsgrens van Turing, en worden algoritmen, de rekenmethodes die ten grondslag liggen aan een programma, ook kritisch bekeken op zogenaamd *combinatorisch explosief* gedrag.

Een goed voorbeeld van combinatorische explosie is de waarheidstabellenmethode voor propositielogica uit het vorige hoofdstuk. Als we willen weten of een propositie vervulbaar is, dan moeten we de hele tabel doorrekenen en kijken of in de laatst in te vullen kolom een *1* voorkomt. We kunnen dezelfde methode voor een circuit toepassen: zo'n circuit correspondeert met een propositie en de bits met de propositieletters. De vervulbaarheidsvraag is dan of een lampje dat we bij de uitgang van het circuit hangen gaat branden voor een of andere instelling van de bits, zoals dat getekend is in Figuur 3.10. De waarheidstabellenmethode gaat netjes alle mogelijkheden af, en dat levert bijvoorbeeld in Figuur 3.10 in het ergste geval dat we $2^9 = 512$ keer de bits moeten instellen en kijken of het lampje gaat branden.

De combinatorische explosie van deze methode bestaat eruit dat met maar een enkel bitje extra de rekentijd twee keer zo lang wordt. Zo moet je voor een circuit met 50 bits meer dan een biljard mogelijkheden doorrekenen (om precies te zijn 1125899906842624, en dat is inderdaad wat meer dan 10^{15}), en in het geval 100 bits meer dan een biljard keer een biljard (meer dan 10^{30}). In plaats van 'explosief' wordt in de complexiteitstheorie, een discipline binnen de theoretische informatica, zo'n algoritme *exponentieel* genoemd, omdat de verhouding tussen rekentijd en de omvang van de invoer exponentieel is. In de tabel hieronder staat deze verhouding in de kolom helemaal links. Als de invoeromvang met een enkele stap toeneemt, dan wordt de wachttijd een bepaalde constante factor langer.

Exponentieel	Polynomiaal	Lineair
tijd $\sim$ constante$^{\lvert\text{invoer}\rvert}$	tijd $\sim \lvert\text{invoer}\rvert^{\text{constante}}$	tijd $\sim$ constante $\cdot \lvert\text{invoer}\rvert$

De opdracht voor programmeur of informaticus is nu om algoritmen te ontwerpen om de variabele invoeromvang uit de exponent te krijgen. Een voor de praktijk zeer acceptabele wachtverhouding

tussen rekentijd en invoeromvang is de *polynomiale*. Daarin zijn de rollen van invoeromvang en tijdsfactor omgedraaid: zie de middelste kolom in de tabel.

Wanneer bij een polynomiaal algoritme de constante 2 is spreken we over een *kwadratisch* algoritme. Bij constante 3 spreken we over *kubisch*. Je gaat het verschil tussen een polynomiaal en een exponentieel algoritme goed merken als de invoeromvang een redelijke grootte krijgt, kijk maar naar de volgende tabel, die het verschil illustreert tussen polynomiale en exponentiële groei. Exponentiële groei zien we ook bij kettingbrieven of ketting-emails, en bij piramidespelen. Wie vooraan in de ketting of bovenaan in de piramide zit heeft kans op grote winst; de sukkels die later instappen gaan met mathematische zekerheid de boot in. Niet zo gek dus dat het opzetten van een piramidespel in Nederland strafbaar is.

n	n^2	n^3	2^n
2	4	8	4
3	9	27	8
4	16	64	16
5	25	125	32
6	36	216	64
7	49	343	128
8	64	512	256
9	81	729	512
10	100	1000	1024
11	121	1331	2048
12	144	1728	4096
13	169	2197	8192
14	196	2744	16384
15	225	3375	32768
16	256	4096	65536
17	289	4913	131072
18	324	5832	262144
19	361	6859	524288
20	400	8000	1048576
21	441	9261	2097152

Nog mooier wordt het als we problemen met een lineaire methode kunnen aanpakken. De groei van de invoeromvang is dan recht evenredig met de wachttijd.

Opdracht 3.28 *Stel dat we twee algoritmen hebben voor het berekenen van een probleem. Het eerste gedraagt zich exponentieel waarbij de* constante *2 is, terwijl het tweede zich polynomiaal gedraagt waarbij de* constante *8 is. Bij welke invoeromvang ga je merken dat het eerste algoritme slechter is dan het tweede?*

3.4.3 Beroemdheid Kent Geen Tijd

Stel we hebben een gezelschap van N mensen bijeen waaronder zich een *beroemdheid* bevindt.[1] Die zouden we graag ontmoeten en om een handtekening vragen. Een beroemdheid binnen een

[1] Met dank aan dr. Anne Kaldewaij.

gezelschap herken je aan twee eigenschappen: ieder ander persoon in het gezelschap kent hem of haar, maar zelf kent hij of zij niemand in het gezelschap (of deze beroemdheid zichzelf wel kent doet er even niet toe). Hoe gaan we te werk? We beginnen als volgt. Om niet in de war te raken geven we alle leden van het gezelschap rugnummers, van 1 tot en met N. Vervolgens beginnen we bij persoon 1 met nagaan of we misschien meteen al om een handtekening kunnen vragen. Hiertoe gaan we de rij van 2 tot en met N af, en vragen we of 1 de personen kent die we opnoemen. Als we er op deze manier achterkomen dat 1 een van de $N-1$ anderen kent, dan valt 1 af en beginnen we opnieuw, nu met persoon 2. Als we uiteindelijk iemand tegenkomen die niemand kent, dan moet dat de beroemdheid zijn. In het ergste geval treffen we het zo dat juist persoon N de beroemdheid is en in dat geval hebben we $(N-1)^2$ tests moeten uitvoeren. Deze methode is dus polynomiaal, en wel: kwadratisch. De wachttijd hangt af van het kwadraat van het aantal personen in ons gezelschap (min 1).

Opdracht 3.29 *Hiernaast is een* ImPro+*-implementatie van een wat sneller algoritme gegeven voor een gezelschap van 1000 personen. We gaan ervan uit dat we al beschikken over een* KENT*-test die aangeeft of het eerste argument (rugnummer van gezelschapslid) het tweede argument kent. Teken eerst het corresponderende programmastructuurdiagram, en leg vervolgens uit wat het programma precies doet, en waarom het efficiënter is dan het algoritme hierboven. Leg ook uit waarom dit programma gegarandeerd de beroemdheid uitkiest.*
N.B. Het blijft een kwadratische methode.

```
x := 1;
f := 0;
while (x < 1001 & f = 0) {
    y :=x ;
    while (y < 1001) {
       if ( KENT(x,y) ) {
           y := 1001; x+;
       } else {
           if (y = 1000) {
               f+;
           } else {
               y+;
           }
       }
    }
}
```

Het algoritme uit de opgave hierboven is wat efficiënter, maar echt veel maakt het niet uit. Immers, de tijd die met het uitvoeren ervan gemoeid is verhoudt zich nog steeds kwadratisch tot de omvang van de invoer (N). Er bestaat echter wel een algoritme dat de wachttijd terugbrengt tot lineair.

Begin met twee personen, 1 en N. Laten we even aannemen dat $N = 1000$. Vraag aan 1 of hij/zij 1000 kent. Als dit niet het geval is, dan kan 1000 niet de beroemde persoon zijn. Dan strepen we 1000 af en gaan nu door met 1 en 999, dat wil zeggen: $N-1$. Als 1 wel 1000 kent, dan kan 1 niet de beroemdheid zijn en gaan we verder met 2 en 1000. Uiteindelijk houden we maar één persoon over, en dit *moet* de beroemdheid zijn. Dit kost ons in alle gevallen N tests.

Opdracht 3.30 *Schrijf een* ImPro+*-programma dat dit algoritme implementeert, weer met gebruikmaking van de functie* KENT *uit de vorige opdracht.*

De tijdscomplexiteit van een rekenprobleem zoals de hier besproken beroemdheidsdetectie is de rekentijd voor het snelste algoritme dat het probleem oplost. Het beroemdheidsprobleem is daarmee een lineair probleem.

3.4.4 Een Open Kwestie: P versus NP

We hebben gezien dat we het oorspronkelijke kwadratische algoritme voor het beroemdheidsprobleem konden vervangen door een lineair algoritme. De vraag rijst nu of er niet ook een beter (polynomiaal) algoritme bestaat voor het vervulbaarheidsprobleem van de propositielogica. Dit blijkt een bijzonder moeilijke en diepe vraag te zijn. De enige bekende algoritmen gedragen zich allemaal exponentieel, en het is tot op de dag van vandaag niet bekend of er een polynomiale oplossing bestaat. De tijdscomplexiteit van het vervulbaarheidsprobleem is daarmee onbekend.

Er bestaat wel een nauwkeurige benadering met betrekking tot zogenaamde *non-deterministische* algoritmen. Dit zijn algoritmen waarbij ook door de uitvoerder zelf keuzes gemaakt mogen worden (zonder een test). Een zeer simpel voorbeeld van zo'n non-deterministisch algoritme voor het vervulbaarheidsprobleem is de volgende methode:

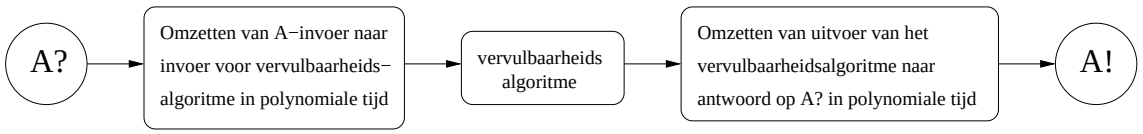
> Kies een instelling van de bits (zoals in Figuur 3.10) en kijk of het lampje brandt.

De non-deterministische complexiteit van een probleem is gedefinieerd als het snelste non-deterministische algoritme waarbij tijdens de uitvoering de meest optimale keuzes worden gemaakt. Als in het hierboven gedefinieerde vervulbaarheidsalgoritme met maximaal geluk gekozen wordt, dan zal in het geval van een vervulbare propositie een instelling gekozen worden waarbij het lampje gaat branden, want als het lampje in zo'n geval niet zou gaan branden, dan moet de uitvoerder niet optimaal geweest zijn in zijn keuze. Natuurlijk kost het de van zijn optimaal geluk overtuigde uitvoerder slechts polynomiaal veel tijd om op basis van dit algoritme met een juist antwoord te komen. Deze non-deterministische tijdscomplexiteit wordt aangeduid met NP, terwijl P staat voor deterministische polynomiale complexiteit.

Non-deterministische tijdscomplexiteit lijkt een tamelijk academisch en onpraktisch begrip. De praktijk leert immers dat non-deterministische keuzen (willekeurige keuzen) niet altijd gouden grepen hoeven te zijn. Het begrip wordt veel natuurlijker als we het als volgt omschrijven: problemen in NP zijn de problemen waarvoor de beste aanpak die tot op heden is bedacht neerkomt op systematisch alle mogelijkheden uitproberen, terwijl het controleren of een oplossing de goede is juist heel efficiënt kan gebeuren. Kijken of bij een bepaalde stand van de bits (schakelaars) in Figuur 3.10 het lampje gaat branden kost slechts polynomiaal veel tijd. De complexiteitskarakterisering NP wordt dan ook veel gebruikt in de theoretische informatica. Veel combinatorische lastige rekenproblemen blijken zich net zo te gedragen als het vervulbaarheidsprobleem. Ze behoren allemaal tot de NP-klasse, maar P-algoritmen ervoor zijn niet bekend. Daarmee onderscheiden ze zich van problemen waarvan aangetoond is dat ze zich alleen maar met een exponentieel algoritme laten oplossen.

Het vervulbaarheidsprobleem voor de propositielogica speelt een centrale rol in de NP-klasse. Het blijkt zo te zijn dat, als een P-algoritme voor dit probleem gevonden wordt, dat dan alle NP-problemen een P-algoritme moeten hebben. Elk probleem in NP laat zich namelijk vertalen naar het vervulbaarheidsprobleem, en die vertaling neemt maar polynomiaal veel tijd in beslag. Met een vertaling bedoelen we dat we een probleem A? kunnen omzetten, zodat het kan dienen als invoer van een vervulbaarheidsalgoritme, om vervolgens de uitvoer weer te vertalen naar een bevredigend antwoord op de vraag A?. Natuurlijk is voor deze omzetting van invoer en uitvoer weer rekentijd nodig, maar men is erin geslaagd te bewijzen dat voor elk NP-probleem een vertaling naar het vervulbaarheidsprobleem te geven is waarvan de rekentijd zich polynomiaal verhoudt tot de invoeromvang. In Figuur 3.11 is deze situatie weergegeven. Hieraan kun je zien

Figuur 3.11: Vertalen van en naar het vervulbaarheidsalgoritme.



dat, als er een polynomiaal algoritme voor het vervulbaarheidsprobleem gevonden wordt, elk NP-probleem polynomiaal op te lossen is. Met andere woorden, dan zou NP hetzelfde zijn als P. Een ander voorbeeld van een probleem in NP dat, net als propositionele vervulbaarheid, exemplarisch is voor de hele klasse, is het mijnenveegprobleem, bekend van het spelletje *minesweeper*, gemakkelijk te vinden op internet. Pas op: het spelletje is verslavend.

> Met het oplossen van het P- versus NP-probleem is veel geld te verdienen. Het *Clay Mathematics Institute* in de VS heeft een miljoen dollar uitgeloofd aan de eerste die hetzij bewijst dat P=NP of bewijst dat P≠NP, en dat bewijs gepubliceerd weet te krijgen in een gerenommeerd wiskundetijdschrift (dat wil zeggen: het bewijs moet voor wiskundigen overtuigend zijn). Hier is de omschrijving van het probleem, van de website van het *Clay Mathematics Institute*, `http://www.claymath.org/`.
>
> Het P- versus NP-probleem.
>
> Het is zaterdagavond en je arriveert op een groot feest. Je voelt je wat opgelaten, en je vraagt je af of je al iemand van de aanwezigen kent. De gastheer oppert dat je Rosa zeker moet kennen, de dame in de hoek naast het buffet. Je werpt een snelle blik, en je ziet dat de gastheer gelijk heeft. Zonder die suggestie van de gastheer had je alle aanwezigen in de hele kamer langs gemoeten om te zien of er een bekende tussen zat. Dit is een voorbeeld van het algemene verschijnsel dat het *genereren* van een oplossing voor een probleem veel meer tijd kost dan het *controleren* of een gegeven oplossing correct is. Net zo: als iemand je vertelt dat het getal 13717421 kan worden geschreven als het product van twee kleinere getallen, dan weet je waarschijnlijk niet of je dat moet geloven of niet. Maar als hij je vertelt dat het kan worden geschreven als 3607×3803 kun je gemakkelijk met een zakrekenmachine nagaan dat hij gelijk heeft. Een van de open problemen in logica en informatica is het vaststellen of er vragen zijn waarvan het antwoord efficiënt kan worden gecontroleerd (bijvoorbeeld met een computer), maar waarbij het vinden van een goede oplossing vanaf 0 (zonder dat je het antwoord weet) veel meer tijd vergt. Het lijkt erop dat er tal van dat soort vragen zijn. Maar tot op heden heeft niemand bewezen dat het oplossen van zulke problemen echt heel veel tijd kost. In theorie zou het zo kunnen zijn dat we eenvoudigweg nog niet ontdekt hebben hoe we ze snel moeten oplossen. Stephen Cook formuleerde het P- versus NP-probleem in 1971.

De P- versus NP-kwestie is een open vraag. In feite is dit het belangrijkste hete hangijzer van de theoretische informatica: veel combinatorisch lastige problemen die we graag efficiënt zouden willen aanpakken zitten in de NP-klasse. Aan het begin van de twintigste eeuw waren wiskundigen optimistisch over de mogelijkheden van het mechaniseren van het rekenen. Turing en Church lieten toen zien dat er fundamentele grenzen zijn aan berekenbaarheid. De stemming onder wiskundigen en informatici nu met betrekking tot het P- versus NP-probleem is heel wat pessimistischer. Vrijwel iedereen gelooft dat NP $\neq$ P, maar er zal een nieuwe Turing moeten opstaan om ons daar daadwerkelijk van te overtuigen, of, heel misschien, om het tegendeel te bewijzen.

Biografieën

Bronnen: *MacTutor History of Mathematics archive*:
`http://www-history.mcs.st-andrews.ac.uk/history/`
en de *Encyclopaedia Britannica*:
`http://britannica.com`

Simon Stevin (1548–1620)

Simon Stevin is een erflater van onze beschaving. In de grote bloeitijd van de Noord-Nederlandse cultuur, de Gouden Eeuw, schitterde hij als toegepast wiskundige, natuurkundige en waterbouwkundige. Hij werd in Brugge geboren, verhuisde in 1581 naar Leiden en kwam als ingenieur in dienst van het leger van prins Maurits, die dankbaar gebruikmaakte van Stevins uitzonderlijke vermogen om theoretisch inzicht praktisch toe te passen.

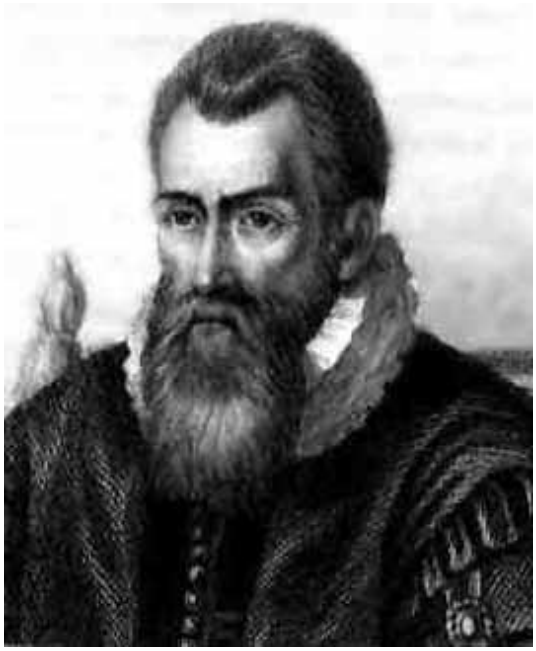
Stevin zag in dat het toenmalige verwarrende stelsel van maten en gewichten op de helling moest en stelde in zijn boek *De Thiende* voor om decimaal te gaan rekenen. Zijn notatie was nog niet optimaal, maar ze werd later door Napier gemodificieerd tot de bekende 'drijvende komma' of 'floating point' notatie, die algemeen ingang vond. Wat betreft het decimale stelsel

voor munten, maten en gewichten was Stevin zijn tijd te ver vooruit: dat werd pas ingevoerd tijdens de Franse Revolutie.

Stevin was de grondlegger van de hydrostatica. Hij toonde aan dat de druk die een kolom vloeistof uitoefent op een oppervlak afhangt van de hoogte van de kolom en de grootte van het oppervlak. Hij wist ook wat je in de praktijk van het oorlogvoeren met water kon doen in een polderland: hij bedacht de Hollandse Waterlinie. In 1586, drie jaar vóór Galilei, wist hij te melden dat dingen van verschillend gewicht even snel vallen.

Stevin is de bedenker van heel wat Nederlandse terminologie voor rekenen en redeneren. Hij schonk het Nederlands woorden als *wiskunde*, *optellen*, *aftrekken*, *vermenigvuldigen*. Hij schreef ook over politiek. Zijn tractaat *Het Burgherlick Leven*, uit 1590, is in 2001 opnieuw uitgegeven.

John Napier (1550–1617)



John Napier werd in 1550 geboren op het landgoed Merchiston bij Edinburgh in Schotland. Op dertienjarige leeftijd ging hij naar de universiteit van St. Andrews. Zonder zijn studie af te maken vertrok hij naar het vasteland van Europa. Aangenomen wordt dat hij aan universiteiten aldaar (waarschijnlijk Parijs en misschien ook in Holland en Italie) zijn kennis van wiskunde en klassieke literatuur opdeed. Maar belangrijker voor hem was theologie. In 1593 publiceerde de fanatieke protestant Napier een boek over het gevaar van een opkomend katholicisme in Schotland. Tussen het werk aan godsdienstige zaken door vond hij met moeite tijd om zich aan zijn hobby wiskunde te wijden.

Hoewel zijn theologische werk een zekere bekendheid genoot (zijn boek uit 1593 werd in meerdere talen vertaald), is Napier vooral bekend door zijn uitvinding van de logaritmen. In 1614 komt zijn boek *Mirifici logarithmorum canonis descriptio* ('Beschrijving van de wonderbare regels der logaritmen') uit. In het voorwoord schrijft hij dat in de wiskunde niets zo vervelend en zonde van de tijd is als vermenigvuldigen, delen en worteltrekken. Nu zijn logaritmen een 'gewoon'

onderdeel van de wiskunde, maar toen was het een methode om, onder andere, vermenigvuldigen te herleiden tot optellen. Om met behulp van logaritmen het vermenigvuldigen te vereenvoudigen was een tabel van logaritmen nodig en Napiers boek bevatte dan ook zo'n tabel. In dit werk gebruikte Napier de decimale notatie voor breuken – hij schreef bijvoorbeeld 0,25 in plaats van 1/4 – die door Simon Stevin in zijn verhandeling *De Thiende* uit 1585 was gepropageerd. Die notatie bleek in logaritmentafels buitengewoon handig.

Drie jaar later beschrijft Napier in zijn boek *Rabdologiae* ('Stokjeskunde') de staafjes met de tafels als hulpmiddel bij het vermenigvuldigen. In hetzelfde jaar, 1617, overlijdt John Napier in Edinburgh.

Wilhelm Schickard (1592–1635)

Wilhelm Schickard werd in 1592 geboren in Herrenberg, niet ver van Tübingen, in Duitsland. Tot 1613 studeerde hij aan de universiteit van Tübingen theologie en oosterse talen. Van 1614 tot 1619 was hij diaken en in die periode leerde hij Johannes Kepler kennen. In 1619 werd Schickard hoogleraar Hebreeuws. Schickard was een universeel geleerde: zijn onderzoeksgebied omvatte onder andere wiskunde, astronomie en landmeetkunde. In 1631 werd hij benoemd tot hoogleraar astronomie aan de universiteit van Tübingen.

In 1957 werd bekendgemaakt dat een brief uit 1623 van Schickard aan Kepler was gevonden waarin Schickard schrijft dat hij een machine heeft gebouwd waarmee op mechanische wijze berekeningen uitgevoerd kunnen worden. De brief bevat schetsen van het apparaat, maar de machine zelf is nooit teruggevonden. Een tweede exemplaar dat Schickard speciaal voor Kepler liet bouwen werd tijdens een brand verwoest. Na 1957 is op grond van de beschrijving en de schetsen een werkend model gebouwd. De machine kon optellen, aftrekken en, met wat kunst- en vliegwerk van de gebruiker, vermenigvuldigen en delen. In 1634 maakte de pest een einde aan

de levens van zijn vrouw en zijn drie dochters en in 1635 werd Schickard zelf ook het slachtoffer van deze ziekte.

Blaise Pascal (1623-1662)

De vader van Pascal was van plan zijn zoon tot zijn vijftiende weg te houden van de wiskunde. Toen de twaalfjarige Blaise echter op eigen houtje meetkundige stellingen begon te bewijzen ging zijn vader overstag en begon hij hem wiskundeonderricht te geven.

Blaise Pascal was geïnteresseerd in natuurkunde, wiskunde en religie. Tussen 1642 en 1645 ontwikkelde en bouwde hij de *Pascaline*, een van de eerste mechanische rekenmachines. Het apparaat was onder andere bedoeld om zijn vader, die belastingambtenaar was, te helpen bij het berekenen van de belastingen. Vanwege het ingewikkelde Franse muntenstelsel van die tijd (1 livre = 20 sous = 12 deniers) zaten er nogal wat technische haken en ogen aan de constructie van deze machine.

De natuurkundige studies van Pascal richtten zich onder andere op het vacuüm, waarvan hij het bestaan met experimenten probeerde aan te tonen. Er volgden felle discussies met onder andere René Descartes die de mogelijkheid van een vacuüm ontkende. De natuur had volgens Descartes een 'horror vacui', en Pascal zelf 'te veel vacuüm in zijn hoofd' (uit een brief van Descartes aan Christiaan Huygens).

In de wiskunde werkte Pascal aan meetkundige problemen en legde hij de grondslagen voor de kansberekening. Dat laatste onderwerp speelt ook een rol in Pascals opvattingen over religie, met name in de befaamde 'weddenschap van Pascal' die de rationaliteit van de geloof aan God moet aantonen. Als God niet bestaat, dan verliest men niets door in hem te geloven, maar als hij wel bestaat, dan verliest men alles door zijn bestaan te loochenen. Je kunt hoe dan ook dus maar het beste in God geloven. Zijn boek *Pensées* (*Gedachten*) bevat deze en andere beschouwingen over geloof en menselijk leed. Het laat Pascals diepe religiositeit zien die hij in een leven vol fysieke pijn en ziekte tot aan zijn dood behield. Pascal overleed al op 39-jarige leeftijd, aan een tumor die vanuit zijn maag uitzaaiingen veroorzaakte naar zijn hersenen.

Gottfried Wilhelm Leibniz (1646-1716)

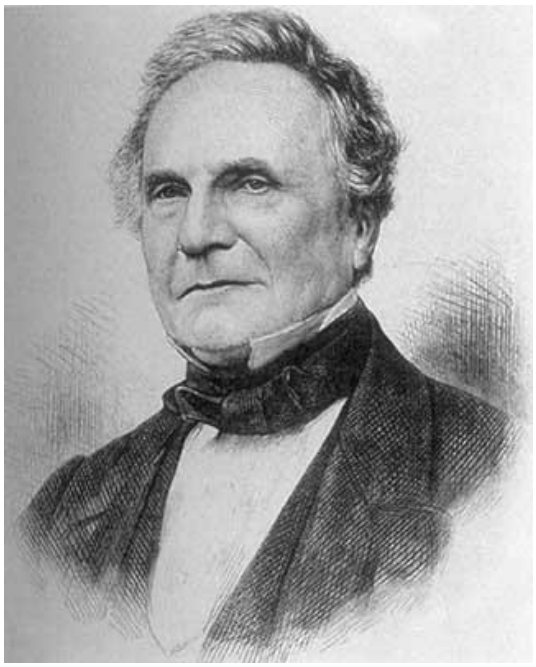
Onder de grote filosofen uit de geschiedenis zijn er maar een paar die met werken de kost moesten verdienen, en daar was Leibniz er één van. Misschien was dat de reden dat hij een echt manusje van alles was, en met meer dan 600 mensen briefwisselingen voerde.

Leibniz was niet alleen op zoek naar een universele taal, maar ook naar een rekenkunde van de rede. Deze rekenkunde zou niet met getallen maar met symbolen werken, en rekenkundige manipulaties zouden dan van oude tot nieuwe waarheden moeten leiden, van premissen tot conclusies. Dit idee van de mechanisering van het redeneren bracht Leibniz tot het voorstel voor een rekenmachine die hij in 1673 voor de leden van de Royal Society in Londen demonstreerde.

Hiernaast kan Leibniz ook als een van de ontwikkelaars van het binaire talstelsel worden beschouwd (1679), waarin 1 voor God en 0 voor de leegte stond. Zijn belangstelling voor het binaire stelsel is niet verwonderlijk in het licht van zijn grote filosofische theorie, de monadologie, en zijn algemene streven naar verzoening van elkaar bestrijdende partijen, bijvoorbeeld de katholieke en de protestantse kerken.

In zekere zin is Leibniz een ongelukkige figuur in de geschiedenis van de wetenschappen. Met zijn ideeën over een algebra van het denken verdwijnt hij geheel in de schaduw van Boole, die deze algebra veel later concrete wiskundige vorm gaf. En in de wiskunde gebeurde iets soortgelijks: in 1675 ontwikkelde Leibniz het differentiaal- en integraalrekenen, maar helaas tegelijkertijd met Isaac Newton. Die laatste gaat, vanwege zijn opzienbarende toepassingen van differentiaal- en integraalrekenen in de natuurkunde, meestal in zijn eentje met de eer strijken.

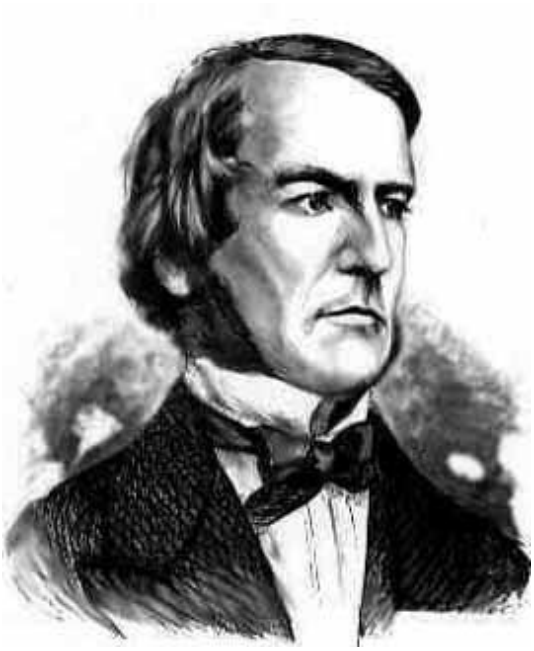
Charles Babbage (1791-1871)



Twee machines vormen het levenswerk van Charles Babbage, de *Difference Engine* en de *Analytical Engine*. Tussen 1819 en 1822 maakte hij de eerste Difference Engine waarmee hij de waarden van $n^2 + n + 41$ voor alle waarden van n kon laten berekenen. Volgens Babbage kon de machine zo'n 60 termen in 5 minuten berekenen. Met name het Engelse Astronomisch Gezelschap was zeer onder de indruk van de Difference Engine, vanwege het nut dat een dergelijke machine voor astronomisch onderzoek zou kunnen hebben. De overheid besloot dan ook geld in de ontwikkeling van een grote Difference Engine te steken, hoewel hier uiteindelijk niets van terecht kwam.

Al in 1834 had Babbage tekeningen voor de Analytical Engine gemaakt. Deze machine bevatte al alle componenten van een hedendaagse computer, zoals beschreven in paragraaf 1.4.1. Babbage heeft echter nooit een Analytical Engine kunnen bouwen, hoewel er toen al ingewikkelde programma's voor de Analytical Engine werden gemaakt, bijvoorbeeld om de reeks van Bernoulli getallen te berekenen. De ongepubliceerde tekeningen van de Analytical Engine werden pas in 1937 herontdekt.

George Boole (1815-1864)



George Boole, geboren op 2 november 1815 in Lincoln, Engeland, was op school al een uitblinker, niet zozeer in wiskunde, zoals je misschien zou kunnen denken, als wel in het Latijn. Toen hij twaalf was werd een door hem vertaalde ode van Horatius gepubliceerd. Er ontstond een discussie met de schoolmeester, omdat die niet wilde geloven dat een twaalfjarige jongen een stuk van zulke diepgang kon produceren.

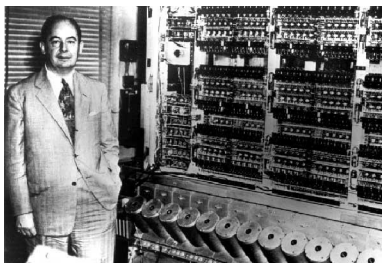
In 1854 publiceerde Boole een belangrijk boek met de lange titel 'An investigation into the laws of thought, on which are founded the mathematical theories of logic and probabilities' ('Een onderzoek naar de wetten van het denken, waarop de wiskundige theorieën van logica en waarschijnlijkheid gegrondvest zijn'). De logica werd hier op een geheel nieuwe manier benaderd, omdat logisch geldige uitspraken tot rekenkundige vergelijkingen werden gereduceerd. Deze vergelijkingen spelen ook vandaag de dag nog een belangrijke rol in verschillende wiskundige disciplines, en verder vormen ze het theoretische fundament van de circuits die je in elke computer tegenkomt.

Op 8 december 1864 overleed Boole aan de gevolgen van een lezing. Vanuit zijn huis moest Boole zo'n drie kilometer door de stromende regen naar de collegezaal lopen om aldaar in natte kleren zijn lezing te verzorgen. Zijn echtgenote probeerde de koorts die daarvan het gevolg was te bestrijden met middelen die op de oorzaak leken: zij gooide een paar emmers water over de in bed liggende Boole.

John von Neumann (1903-1957)

Johan (later John) von Neumann werd geboren in Boedapest. Al heel vroeg op school werd duidelijk dat hij een wiskundig genie was, met een heel bijzonder geheugen. Zo had hij bijvoorbeeld na een paar keer doorlezen een hele pagina uit het telefoonboek in zijn hoofd. Zijn wetenschappelijke carrière bracht hem in 1930 naar de Amerikaanse Princeton universiteit waar hij 1933 tegelijk met Albert Einstein professor werd.

Met Oskar Morgenstern is Von Neumann de grondlegger van de speltheorie. In hun boek uit 1944 (*De theorie van spelen en economisch gedrag*) wordt een poging gedaan een wiskundig model (het spel) voor economische interacties te construeren. Een spel kent aan elke speler een aantal mogelijke strategieën toe, en de uitkomst van het spel is dan afhankelijk van de door de spelers gekozen strategieën. Analyse van het spel kan bepaalde optimale strategieën opleveren, bijvoorbeeld strategieën die het best mogelijke resultaat opleveren ongeacht de door de tegenstander(s) gekozen strategie.



In 1944 kwam Von Neumann in contact met de groep die de ENIAC computer aan het bouwen was. Tijdens de bouw werd al overlegd over de architectuur van een opvolger. Deze discussies zetten Von Neumann aan tot het schrijven van de beroemde *First Draft of a Report on the EDVAC* (1945). Dit was de eerste beschrijving van de architectuur die bekend werd als 'Von Neumann architectuur', met zowel het programma als de data opgeslagen in het computergeheugen.

De EDVAC was bedoeld als de eerste computer volgens dit nieuwe concept. Dat lukte niet helemaal: door verloop in de groep liet de voltooiing van de EDVAC tot 1952 op zich wachten. Intussen waren in Groot-Brittannië al computers van het Von Neumann type operationeel.

Von Neumann deed theoretisch onderzoek naar kwantummechanica en naar het wezen van berekening (in de zogenaamde automatentheorie). Hij verkocht ook wel eens onzin. Zo stelde

hij voor om de poolkappen donker te verven. De gereduceerde reflectie van het zonlicht zou er dan voor zorgen dat het klimaat van IJsland op dat van Hawaii zou gaan lijken.

Alan Mathison Turing (1912–1954)

Alan Mathison Turing werd in 1912 geboren in Londen. Hoewel een buitenbeentje in het Engelse schoolsysteem, was hij van jongs af fanatiek en zelfstandig. Om zijn eerste kostschooldag, die viel tijdens een algemene staking, niet te missen fietste hij als veertienjarige in zijn eentje honderd kilometer. In 1931 werd hij als student aangenomen op King's College in Cambridge. Nog voor hij afstudeerde schreef hij in 1939 een baanbrekend artikel over het begrip 'berekenbaarheid'. In dit artikel wordt in zeer algemene termen een computer beschreven; dit theoretische model van een symbolen verwerkend apparaat werd later een Turing machine genoemd.

In 1939 kwam Turing terecht bij de Britse Government Code and Cipher School, waar hij een belangrijke bijdrage leverde aan het ontcijferen van Duitse geheime boodschappen. De Duitsers gebruikten voor hun communicatie de zogenaamde Enigma, een codeermachine die via een ingewikkeld proces van transcriptie via roterende schijven letters omzette. Om een Enigma codebericht te ontcijferen moest het nogmaals door de machine worden gehaald, met dezelfde beginstand. De Britten hadden een exemplaar van de machine in handen gespeeld gekregen, maar om de berichten te kraken moesten ze de Enigma beginstand weten, en er waren zo'n 159 000 000 000 000 000 000 mogelijkheden.
Met de hand uitproberen was onbegonnen werk, maar Turing had een briljant idee om het probleem te simplificeren, en daarna mechanisch te kraken, door machinaal alle mogelijke beginstanden te doorlopen tot de juiste stand gevonden was. Daarmee werden opmerkelijke successen geboekt, maar alles in het diepste geheim. Alans ouders, bij voorbeeld, hadden geen idee van de rol die hun zoon speelde in de oorlog.

Na de oorlog werkte Turing mee aan het ontwerp en de bouw van de ACE (Automatic Computing Machine), een van de eerste echte elektronische computers. Hij zou de voltooiing van dit project niet meer beleven. In 1952 werd Turing veroordeeld wegens homoseksuele praktijken (toen nog strafbaar in Groot-Brittannië). Hij kon kiezen tussen de gevangenis of hormooninjecties toegediend krijgen om zijn libido te beteugelen. Turing raakte depressief en verbitterd, en twee jaar later maakte de man die de oorlog had helpen winnen een einde aan zijn leven met het eten van een in cyanide gedoopte appel.

Flaptekst

De vraag 'Kunnen computers denken?' is volgens de Nederlandse informaticus Edsger Dijkstra ongeveer even interessant als 'Kunnen duikboten zwemmen?' Op die vraag krijg je dan ook in dit boek geen antwoord. De vraag wordt pas spannend wanneer je probeert te specificeren hoe een 'denkwedstrijd' tussen een mens en een computer er zou kunnen uitzien. Dit boek vertelt daar meer over.

Het basisstramien van de taken die een computer uitvoert blijkt zo eenvoudig te zijn dat de wiskundige en filosoof Alan Turing rond 1930 al een wiskundige formalisering kon geven van de essentie van mechanische berekenbaarheid. En als je eenmaal begrijpt hoe een Turing machine (Turings abstracte model van de computer) werkt begrijp je in feite van elke computer hoe hij werkt.

Over de auteurs

Jan van Eijck is filosoof, computationeel taalkundige en toegepast logicus. Hij werkt op het Centrum voor Wiskunde en Informatica en doceert aan de universiteiten van Utrecht en Amsterdam.

Jan Jaspars is freelance wiskundige. Hij doceert logica en informatica aan verschillende universiteiten, en ontwerpt educatieve software voor universitair en middelbaar onderwijs.

Jan Ketting is wiskundige en werkt op Instituut De Leeuw als docent en studiebegeleider van middelbare scholieren.

Marc Pauly is als informaticus verbonden aan de universiteit van Liverpool. Zijn werk slaat een brug tussen logica en speltheorie.

Informatie in Context is een reeks die speciaal wordt ontwikkeld voor het voortgezet onderwijs. De reeks biedt informatie over ontwikkelingen in de beta wetenschap die het gezicht van de moderne informatiemaatschappij mede bepalen, met aandacht voor de historische context. De serie is op vele manieren en op vele niveaus te gebruiken.

Website bij dit boek: `http://www.science.uva.nl/denkendemachines/`