AT Computing
Arnhemsestraatweg 33
6881 ND Velp
The Netherlands

Booleaanse Algebra

Versie: 1c

Inhoudsopgave

1 | Inleiding

Alle computertalen kennen “Booleaanse logica”, een systeem waarmee je combinaties bouwt van uitspraken die elk apart **waar** of **onwaar** zijn. Zo’n uitspraak is bijvoorbeeld: “*deze disk is voor meer dan 90% bezet*”. Andere voorbeelden zijn: “*op dit moment zijn er precies 12 mensen ingelogd*”, of “*deze file is al meer dan een week niet meer gewijzigd*”. Wanneer duidelijk is welke disk, welke computer of welke file je aanwijst, kun je van de bewering (laten) uitzoeken of ze **waar** of **onwaar** is.

Een **waar/onwaar** conclusie gebruik je typisch in een **if/then/else** situatie, om te besluiten welke stappen je gaat nemen. Zo’n conclusie kan dus dienen als de aansturing van een soort spoorweg-wissel:

```
ALS (deze bewering waar is)
DAN    gaan we déze ... stappen nemen
ANDERS gaan we díe ... stappen nemen
```

In de praktijk is het vaak wat ingewikkelder, en heb je combinaties van meerdere beweringen nodig: **ALS** deze disk voor meer dan 90% vol zit, **EN ALS** deze logfile al meer dan een week niet gewijzigd is, **DAN** gooien we de file weg, **ANDERS** zoeken we een andere oplossing.

In bovenstaand voorbeeld zitten **twee** beweringen die elk voor zich **waar/onwaar** kunnen zijn. Maar het stappenplan wordt gestuurd door een bepaalde combinatie van die twee: een **EN**-combinatie. Met die **EN** bouw je uit de twee aparte **waar/onwaar** uitkomsten één eindconclusie, en die bestuurt uiteindelijk de spoorweg-wissel.

Er blijken strikte afspraken nodig voor dit bouwen van een eindconclusie uit een verzameling aparte conclusies. Zonder afspraken krijg je verschillen in aanpak, dus verschillen in uitkomst. George Boole, een Engelse wiskundige, ontwierp rond 1845 een stelsel afspraken dat bruikbaar en volledig was, en geen interne tegenstrijdigheden had. Dit noemen we de **Booleaanse Logica**. De oorsprong van zijn systeem ligt in de **predicatenleer**, die de Griekse wijsgeer Aristoteles al rond 340vC heeft uitgewerkt.

In de rest van dit verhaal concentreren we ons op die Booleaanse Logica, en laten de **DAN/ANDERS** stappen vaak weg.

We concentreren ons op de manier waarop de **ALS….** één uitkomst moet leveren, **waar** of **onwaar**, zodat de spoorweg-wissel ons daarmee naar de gewenste vervolgstap kan leiden.

In plaats van **waar/onwaar** wordt in computerjargon vaak **true/false** gebruikt, en natuurlijk ook vaak `1` (= waar) en `0` (= onwaar).

2 | De ‘and’ operatie

In de inleiding zagen we al een **EN**-operatie aan het werk; de Engelse naam is **AND**. Daarmee kun je **twee** beweringen koppelen en een eindconclusie krijgen. Laten we de twee beweringen A en B noemen, en laten we aannemen dat we zowel van A als van B elk **apart** hebben uitgezocht of ze waar of onwaar is. Daarna bekijken we wat de eindconclusie “**A én B**” is:

In de programmeertaal C, en op veel andere plaatsen, wordt een `&` (of een `&&`) gebruikt als **AND**. Dat gaan we in de rest van dit verhaal ook doen.

Je ziet bovenstaande tabel ook wel eens als volgt opgeschreven:

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

en vrij vaak gebruikt men zelfs de volgende, sterk gecondenseerde notatie:

&	0	1
0	0	1
1	0	1

Boven de horizontale streep neemt u de uitkomst die u van bewering A had uitgezocht, en tegen de linkerkantlijn de uitkomst van bewering B. Op het kruispunt vindt u de uitkomst van A&B

3 | De ‘or’ operatie

Behalve de **AND** is ook de **OR** (Nederlands: “**of**”) een belangrijke Booleaanse operatie om twee beweringen te koppelen. Laten we de twee beweringen opnieuw A en B noemen, en laten we aannemen dat we zowel van A als van B elk **apart** hebben uitgezocht of ze waar of onwaar is. Daarna bekijken we de eindconclusie “**A or B**”:

In de programmeertaal C wordt een `|` (of een `||`) gebruikt als **OR**. Dat gaan we in de rest van dit verhaal ook doen.

De volgende notatie is weer equivalent met het tabelletje hierboven:

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

En in gecondenseerde notatie krijg je de volgende tabel

|	0	1
0	0	0
1	1	1

4 | De ‘not’ operatie

De derde belangrijke Booleaanse operatie is de **NOT**. In tegenstelling tot **AND** en **OR** werkt deze **NOT** slechts op één bewering (laten we die A noemen). De **NOT** operatie keert de uitkomst gewoon om:

In de programmeertaal C wordt voor de **NOT** een uitroepteken gebruikt. Dat nemen we in dit verhaal over. Je ziet bovenstaande tabel ook wel eens als volgt opgeschreven:

!	&
0	1
1	0

Prioriteit: Bij de **NOT** operatie maken we een speciale afspraak: wanneer hij ergens voorkomt dan heeft hij alleen betrekking op de enkele bewering die er vlak achter staat. Bijvoorbeeld:

ALS !A & B DAN ……

betekent: als A onwaar is, én tevens B waar is, dan….. De **NOT** heeft dus alleen betrekking op de A; je moet beginnen met de waarde van **NOT A** te bepalen. Als je die uitkomst hebt, verwerk je ‘m vervolgens in de **AND B**.

Laten we eens een voorbeeld in gewone-mensentaal bekijken:

- Bewering A luidt: “De bakkerswinkel is vandaag gesloten”
- Bewering B luidt: “Een collega is vandaag jarig”
- Elke ochtend, bij de eerste koffiepauze, wordt **!A&B** uitgerekend, en dan weet je of er getrakteerd zal worden: “**ALS** de bakkerswinkel vandaag **NIET** gesloten is, **EN** een collega is vandaag jarig, **DAN** krijgen we gebak bij de koffie”.

Als je vanuit je computer bij de website van de bakkerswinkel kunt komen, en als je ook nog het personeelsbestand met de geboortedata erbij haalt, dan kun je een programma schrijven dat elke morgen uitzoekt of je gebak kunt verwachten.

Merk wel op dat de Booleaanse vergelijking geen informatie geeft over **wíe** er gaat trakteren, en ook niet of je misschien zelfs twee gebakjes gaat krijgen. De Booleaanse vergelijking geeft alleen maar een kaal **WAAR/ONWAAR** antwoord op de vraag: “is er vandaag gebak?”. De uitkomsten van een Booleaanse expressie zijn (figuurlijk) altijd “zwart/wit” uitkomsten.

Dit systeem is erg gevoelig voor een **exacte** formulering van de beweringen waar je mee begint. Als je de beweringen een klein beetje verandert (bijvoorbeeld van bewering A maak je “De bakkerswinkel is vandaag open”) dan moet je de Booleaanse vergelijking ook meteen anders opbouwen.

5 | Haakjes

Net zoals in de “gewone” rekenkunde heb je bij het Booleaanse rekenen veel plezier van **haakjes**, om een bepaalde volgorde van verwerking af te dwingen. Het volgende voorbeeld demonstreert dat.

Stel je moet een verzameling files van je collega’s opruimen. Je maakt een programma dat de files stuk voor stuk bekijkt, en bij elke file deze beslissing neemt:

> ALS deze file van Jan is
>
> OF deze file van Marie is
>
> DAN laten we ‘m staan
>
> ANDERS gooien we ‘m weg

We schrijven het wat abstracter op:

- Bewering A luidt: “deze file is van Jan”
- Bewering B luidt: “deze file is van Marie”
- De beslissing gaat bij elke file als volgt:
  ALS A | B
  DAN doe niets
  ANDERS gooi weg

In dit voorbeeld kom je al gauw op de gedachte om de **DAN** en de **ANDERS** om te wisselen, zodat het weggooien in de **DAN**-tak komt, en de **ANDERS** tak overbodig wordt. Maar: als je de activiteit omkeert, moet je eerst de beslissing omkeren:

> ALS ! (A | B)
>
> DAN gooi weg

De haakjes zijn hier echt nodig. Zonder haakjes had er gestaan: ! A | B en daarbij geldt de prioriteitsregel uit de vorige paragraaf: een **NIET**-operatie heeft betrekking op de enkele bewering die er vlak achter staat. Zonder haakjes betekent het dus hetzelfde als: (! A) | B

6 | Prioriteiten

In de vorige paragraaf hadden we de vraag wat !A|B betekent: is het (!A)|B, of is !(A|B) de juiste betekenis. We hadden haakjes genomen om een eenduidige betekenis af te dwingen. Maar bij ingewikkelde expressies kan het aantal haakjes erg oplopen, en wordt de zaak snel onleesbaar. Daarom bestaan er **prioriteiten**.

We hadden al één zo'n prioriteitsregel gezien: "een ! (NOT) hoort altijd bij de enkelvoudige bouwsteen die er vlak achter staat". Met die regel mogen we !A|B gewoon opschrijven, en hoeven geen haakjes apart om de !A heen. De haakjes **mogen** wel, maar **moeten** niet.

De tweede prioriteitsregel is dat een & (AND) belangrijker is dan een | (OR). Bijvoorbeeld als je opschrijft A & B | C & D dan betekent dat: (A&B)|(C&D).

Populair gezegd betekent deze regel: "de AND kleeft harder dan de OR". Als er staat: A|B&C dan moet je dat uit elkaar proberen te trekken, en dan merk je dat de B&C het hardst aan elkaar kleven. Dus die reken je het eerst uit. Daarna pas combineer je de A| ermee.

Er zijn dus twee prioriteitsregels:

(1) De ! heeft hogere prioriteit dan de & en de |
(2) De & heeft hogere prioriteit dan de |

Alle combinaties die niet onder deze regels vallen, moeten stapsgewijs van links naar rechts worden afgehandeld.

Met deze twee prioriteitsregels hebben de Booleaanse operatoren hun eigen versie van: "Mijnheer Van Dalen Wacht Op Antwoord" zoals we die kennen van de rekenkundige operatoren. De Engelse vakterm is "precedence rules".

7 | Gedeeltelijke afwerking

In veel situaties waar Booleaanse expressies voorkomen, zal de afhandelings-software proberen om een “minimale evaluatie” te doen. Dat betekent dat een expressie wordt doorgerekend tot de uitkomst vaststaat, en dan stopt de verwerking. Het kan zijn dat de stop al plaatsvindt lang voordat de hele expressie doorgevlooid is. Dat kan weer neveneffecten hebben, waarop je moet letten.

We gaan terug naar het voorbeeld met het gebak bij de koffie, uit paragraaf *De ‘not’ operatie*, maar we hebben de A en B even omgewisseld om het effect wat duidelijker te krijgen.

- Bewering A luidt: “Een collega is vandaag jarig”
- Bewering B luidt: “De bakkerswinkel is vandaag gesloten”

We hadden daarbij een programma dat in het personeelsbestand kan zien of iemand jarig is, en dat ook via de website van de bakkerswinkel de openingstijden uitzoekt:

“**ALS** een collega vandaag jarig is EN de bakkerswinkel vandaag **NIET** gesloten is, **DAN** krijgen we gebak bij de koffie”.

Als op een dag de Internet-verbinding plat ligt. kan het programma dan toch nog vertellen of er wel/geen gebak zal zijn? Er zijn inderdaad dagen dat het programma, z nder Internet, een duidelijke uitspraak kan doen. Als namelijk **niemand** op die dag jarig is, is de website van de bakker ook niet nodig. Kijk maar in onderstaande AND-tabel, die we aan het voorbeeld hebben aangepast:

A blijkt onwaar	B is onbekend	eindconclusie **A AND B** is onwaar
A blijkt onwaar	B is onbekend	eindconclusie **A AND B** is onwaar
A blijkt waar	B is onbekend	eindconclusie **A AND B** is niet te bepalen
A blijkt waar	B is onbekend	eindconclusie **A AND B** is niet te bepalen

Software wordt vaak zo geschreven dat, ls het antwoord halverwege de rit al vaststaat, het verdere rekenwerk meteen wordt afgekapt.

8 | Tweewaardig of driewaardig?

In het voorafgaande voorbeeld dook opeens een nieuw fenomeen op: naast **waar** en **onwaar** kregen we een **niet te bepalen** situatie erbij. Zo'n "derde mogelijkheid" maakt het lastig als je toch een beslissing wilt laten volgen op de eindconclusie van je programma. Om bij het voorbeeld te blijven: ga je op een dag dat het Internet onbereikbaar is toch even in de kantine kijken? In de praktijk moet je vaak zo'n "derde" onbeslisbare mogelijkheid alsnog in de normale **true/false** einduitkomst erbij wringen.

Bijvoorbeeld: u moet een programma schrijven dat aan een examenkandidaat vertelt of hij/zij geslaagd is via het gemiddelde punt van een aantal proefwerken. U programmeert als volgt:

ALS ((*optelsom van alle punten/aantal proefwerken*) > 5) DAN geslaagd…

Hier ziet u de "derde mogelijkheid" over het hoofd! Dit programma krijgt namelijk problemen als een kandidaat de hele periode ziek is geweest. De correcte aanpak is:

ALS ((*aantal proefwerken* > 0) AND

((*optelsom van alle punten/aantal proefwerken*) > 5)) DAN geslaagd…

want dan hebt u een "onbesliste" uitkomst vermeden, en bent u terug bij een tweewaardig **true/false** eindresultaat.

Merk ook op dat in het bovenstaande de "gedeeltelijke afwerking" een essentiële rol speelt, om een deling door 0 te vermijden. Hoewel een AND in principe symmetrisch werkt (A&B = B&A) kunnen dergelijke randvoorwaarden in de praktijk je toch dwingen om een bepaalde volgorde aan te houden.

9 | De wetten van 'De Morgan'

In paragraaf *Haakjes* zagen we de expressie !(A|B)

Dat betekende daar: "ALS NIET (de file is van Jan OF de file is van Marie), DAN…".

In het normale spraakgebruik zeg je dat meestal een beetje anders:

"ALS de file NIET van Jan is EN de file NIET van Marie is, DAN…"

Merk op dat we hier van een OF naar een EN overspringen.

Eerste wet van De Morgan

Dat die beide uitspraken hetzelfde opleveren, is **de eerste wet van De Morgan**:

```
!(A | B) = !A & !B
```

Met de omwisselingen in het normale spraakgebruik moet je erg opletten. Kijk eens naar de twee volgende uitspraken; je moet ze zorgvuldig proeven:

(1)"ALS de file NIET van Jan is EN de file NIET van Marie is, DAN…"

(2)"ALS de file NIET van Jan is OF de file NIET van Marie is, DAN…"

De tweede uitspraak brengt je in de problemen. Een file is nooit van twee personen tegelijk: als hij van Jan is, is hij **dus** niet van Marie, en omgekeerd. Van de twee mogelijkheden "NIET van Jan" resp. "NIET van Marie" is er dus altijd minstens eentje **waar**. Maar bij de OR-operatie geldt: als er minstens eentje **waar** is, is de eindconclusie óók altijd **waar**. Dus de tweede uitspraak is volstrekt nutteloos: ze is **altijd waar**, voor elke file die je tegenkomt.

Uitspraak (2) van hierboven lijkt erg op de volgende uitspraak:

(3)"ALS de file NIET (van Jan OF van Marie) is, DAN…"

maar hier hebben we weer een duidelijk geval van wat we aan het einde van paragraaf *De 'not' operatie* al hebben gezegd: je moet je formuleringen exact neerzetten. Als je er daarna aan gaat sleutelen, bouw je eigenlijk een heel andere bewering. Dus krijg je een ander verhaal, dus mogelijk een andere uitkomst. Probeer maar eens uit de laatste uitspraak (3) de twee basis-beweringen los op te schrijven. zoals we dat eerder hebben gedaan: "uitspraak A is…" en "uitspraak B is…" Dan zie je dat hij wel hetzelfde klinkt als uitspraak (2), maar in feite gelijk is aan uitspraak (1).

Voorbeeld

Er is een exacte manier om te bewijzen dat de eerste wet van De Morgan inderdaad geldig is: je kunt gewoon alle mogelijke gevallen doorrekenen. Dat is meteen een nuttige oefening, en gaat als volgt: als je uitgaat van twee basis-beweringen A en B, dan zijn er vier combinaties van waar/onwaar mogelijk. In onderstaande tabel staan die aan de linkerkant van de vier regels:

A	B	A | B	!(A | B)	!A	!B	!A & !B
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Kolom drie is een tussenstap op weg naar kolom 4, en kolommen 5 en 6 zijn tussenstappen op weg naar kolom 7. De eerste wet van De Morgan is bewezen doordat kolommen 4 en 7 gelijk uitkomen.

Tweede wet van De Morgan

```
!(A & B) = !A | !B
```

Deze **tweede wet van De Morgan** heeft een mooie symmetrische gelijkenis met de eerste wet. We bewijzen 'm via de tabel-methode, en dan komen wat voorbeelden:

A	B	A&B	!(A&B)	!A	!B	!A | !B
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Het volgend voorbeeld begint met deze twee beweringen;

- Bewering A luidt: "Het stoplicht staat op groen"
- Bewering B luidt: "Het kruispunt is vrij"

De beslissing gaat dan als volgt:

ALS het stoplicht NIET op groen staat OF het kruispunt NIET vrij is

DAN moeten we stoppen, ANDERS mogen we doorrijden

Wanneer je dat omrekent volgens de tweede wet van De Morgan, krijg je:

ALS NIET (stoplicht op groen EN kruispunt vrij)

DAN moeten we stoppen, ANDERS mogen we doorrijden

dus dat komt inderdaad op hetzelfde neer. Het nut van de wetten van De Morgan is dat ze je helpen om datgene wat je opschrijft zo duidelijk mogelijk te krijgen, terwijl je toch zeker weet dat verschillende formuleringen exact hetzelfde betekenen. Vanaf het laatste voorbeeld hierboven kun je namelijk gemakkelijk doorredeneren naar de volgende, meest duidelijke uitspraak:

ALS (stoplicht op groen EN kruispunt vrij)

DAN mogen we doorrijden, ANDERS moeten we stoppen

Bij het bestuderen van een situatie ga je soms allerlei mogelijke combinaties proberen te doorzien, en slaag je er uiteindelijk in om bijvoorbeeld de eerste van bovenstaande beweringen op papier te krijgen. Maar die is toch nog ingewikkeld, dus je blijft twijfelen. Dan kunnen de wetten van De Morgan (plus een beetje routine) je helpen om de bewering stap voor stap om te bouwen, totdat je opeens een vorm bereikt waarvan je zegt: "Nu ziet het er heel logisch uit, dus dat geeft vertrouwen".

> Op dit punt hebben we de belangrijkste theorie achter de rug. Als u het allemaal maar taai vond, kunt u het beste hier stoppen en later nog eens van voren af aan beginnen. De volgende paragrafen brengen nog meer theorie, meer voorbeelden, en wat "speelgoed voor gevorderden".

10 | Bitwise Boolean

Voorbeeld: keyboard ASCII

Booleaanse rekentechnieken worden vaak gebruikt in combinatie met bitpatronen, bijvoorbeeld met ASCII-waarden van lettertekens. Het volgende voorbeeld komt uit de werking van een computer-keyboard:

- De ASCII-waarde van de hoofdletter 'G' is 0x47. In bits uitgeschreven is dat 0100.0111.
- De ASCII-waarde van de kleine letter 'g' is 0x67. In bits uitgeschreven is dat 0110.0111.
- De werking van een **Shift**-toets is (voor normale letter-toetsen) een bit-voor-bit operatie van "getypte letter" AND 1101.1111
  Want de getypte toets produceert een patroon met een mengsel van nulletjes en eentjes (die we maar allemaal als 'x' opschrijven omdat we het niet preciezer weten):
  `xxxx.xxxx` deze (niet nader bekende) bits komen van de toets
  `&&&&.&&&&` dit betekent dat je voor elke bit apart AND moet uitrekenen
  `1101.1111` dit patroon wordt door de Shift-toets geleverd
  `====.====`
  `xx0x.xxxx` AND een willekeurige bit met '1' en hij blijft ongewijzigd
  AND een willekeurige bit met `0` en de uitkomst wordt 0
  We zeggen wel: "trekt de bit omlaag naar 0"

Gebruik dit verhaal maar eens om aan te tonen dat Shift-g een G oplevert. Een complete lijst van dit soort operaties is:

- **willekeurige bit** AND `1` → die bit blijft ongewijzigd
- **willekeurige bit** AND `0` → die bit wordt naar `0` getrokken
- **willekeurige bit** OR `0` → die bit blijft ongewijzigd
- **willekeurige bit** OR `1` → die bit wordt naar `1` getrokken

Voorbeeld: UNIX umask en chmod

Dergelijke bitwise Boolean operaties kom je vaak tegen. Ze zitten bijvoorbeeld ook in het UNIX-`umask` mechanisme. Wanneer een gebruiker een nieuwe file aanmaakt (bijvoorbeeld met een `cp` commando) dan heeft de programmeur van dat `cp`-programma bepaald welke protectiemode de nieuwe file gaat krijgen. Maar de gebruiker kan in zijn `umask` instellen dat hij slechts bepaalde mode-bitjes vanuit dat programma opgedrongen wil krijgen. Een `umask` censureert dus.

Programma `cp` wil je geven:	`rw-rw-rw-`
Je `umask` accepteert slechts	`rwxr-x---` (d.w.z. `umask u=rwx,g=rx,o=`)
Resultaat via bit-wise AND:	`&&&&&&&&&`
Dus je krijgt:	`rw-r-----`

Je moet wel even in gedachten een kleine vertaling maken: **aan**wezigheid van een `rwx`-letter tellen we op elke plek als een `1`, en **af**wezigheid (dus een `-`) tellen we op die plek als een `0`.

Een ander voorbeeld vinden we in de werking van het `chmod`-commando, waarmee je mode-bitjes kunt toevoegen of weghalen, bijv. commando `chmod ugo+rw`:

Stel een file heeft:	`rw-r-----`
chmod voegt toe:	`rw-rw-rw-` (d.w.z. `chmod ugo+rw`)
Resultaat via bit-wise OR:	`|||||||||`
Dus je krijgt:	`rw-rw-rw-`

Weghalen van een bepaald bitpatroon is wat ingewikkelder. Stel dat je commandeert `chmod ugo-wx` dan is Booleaans weghalen hetzelfde als AND met het ge nverteerde patroon (“AND met de NOT-waarde van het patroon”):

Stel een file heeft:	`rwxr-x---`
chmod haalt weg:	`-wx-wx-wx` (d.w.z. `chmod ugo-wx`)
omgekeerd is dat:	`r--r--r--` ←
Resultaat via bit-wise AND:	`&&&&&&&&&`
Dus je krijgt:	`r--r-----`

Als we even terugkijken naar het vorige voorbeeld met de ASCII-toetscodes dan is de werking van de **Shift** eenvoudiger te formuleren als: “haal de `0010.0000` bit weg”.

11 | De 'xor' (exclusive-or) operatie

Naast de gewone OR kun je ook een XOR (exclusive-OR) operatie tegenkomen. Echte programmeertalen kennen die altijd, maar in eenvoudige taaltjes-met-Booleaanse-component (bijvoorbeeld in de UNIX-commando's `find` en `test`) wordt hij wel eens weggelaten. Je hebt 'm niet zo vaak nodig.

Dit zijn de spelregels:

In de programmeertaal C wordt een `^` (hoedje) gebruikt als **XOR**.

Je ziet bovenstaande tabel ook wel eens als volgt opgeschreven:

^	0	1
0	0	1
1	1	0

De **XOR**-operatie wordt erg veel gebruikt in encryptie-berekeningen. Een aardige mogelijkheid van **XOR** is ook dat je er twee waarden `a` en `b` onderling mee kunt omwisselen: `a=a^b; b=a^b; a=a^b;` Na afloop van deze drie stappen is `a` geworden wat `b` oorspronkelijk was, en omgekeerd.

12 | Oefenen

We bekijken nog een aantal oefeningen, om routine op te doen. We geven steeds een expressie, en de bedoeling is dat je er een tabel van mogelijkheden voor opstelt zoals we die ook bij het bewijs van de wetten van De Morgan hebben opgesteld. Voor theorie-liefhebbers zijn de voorbeelden belangrijke stof, maar in ons verhaal zijn ze alleen maar bedoeld als vingeroefeningen voor het rekenen, en om een expressie “op het oog” te leren doorgronden.

De & is overbodig

We hebben de operaties AND, OR en NOT gezien. Een populaire oefening is om aan te tonen dat een van de drie overbodig is. Bijvoorbeeld de AND:

A & B = ! (!A | !B)

Links staat een AND, in z’n eentje. Rechts staat iets waar helemaal geen AND in voorkomt. We gaan bewijzen dat beide kanten hetzelfde zijn. Als we dat klaarspelen dan kun je voor de rest van je loopbaan de AND dus missen: overal waar je nog een AND tegenkomt, vervang je ‘m door de constructie van de rechterkant…..

A	B	A&B	!A	!B	!A | !B	!(!A | !B)
0	0	0	1	1	1	0
0	1	0	1	0	1	0
1	0	0	0	1	1	0
1	1	1	0	0	0	1

Je ziet: kolom 3 en kolom 7 komen hetzelfde uit, dus het bewijs is geleverd. We kunnen de AND missen, maar of het er allemaal leesbaarder van wordt?

Zo’n bewijs door uitwerken van de tabel is volledig en sluitend. Maar het is ook nuttig om dergelijke vergelijkingen “op het oog” te leren bekijken. Wanneer je daar wat gevoel voor ontwikkelt, dan gaat een vergelijking al leven als je ‘m zorgvuldig leest. Zo begint de kweek van een wiskundeknobbeltje.

Bovenstaande vergelijking kun je in feite al zien zitten als je ‘m vergelijkt met de eerste wet van De Morgan. Pak die er maar eens bij, en schrijf daar aan beide kanten een extra ! bij. Dan ben je al in de buurt van de vergelijking uit deze paragraaf.

De | is overbodig

Als je de AND niet wilt missen, wil je misschien wel de OR overboord doen. Op dezelfde manier als in de vorige paragraaf zien we dat: A | B = !(!A & !B)

A	B	A | B	!A	!B	!A & !B	!(!A & !B)
0	0	0	1	1	1	0
0	1	1	1	0	0	1
1	0	1	0	1	0	1
1	1	1	0	0	0	1

Het bewijs zit hier opnieuw in de gelijkheid van kolom 3 en kolom 7. Maar je had natuurlijk al de tweede wet van De Morgan genomen, en daarin de A en de B vervangen door !A resp. !B.

De ^ (xor) is helemáál overbodig

Die ontbreekt in de praktijk al geregeld, en dat illustreert zijn overbodigheid. Maar u kunt het ook formeel opschrijven:

A ^ B = (A | B) & ! (A & B)

Nu moet u eens op het oog proberen om te begrijpen wat hier staat door het langzaam van links naar rechts te lezen. We hadden de populaire definitie gezien als: “een van beide moet je hebben, maar dubbel-op is teveel”. Dat is precies wat hier ook staat: A OF B EN NIET “A EN B beide”.

13 | De ‘nand’ (not-and) operatie

In de beginjaren van het computertijdperk was behandeling van de NAND altijd een verplicht nummer. Wij voeren ‘m hier ten tonele als opnieuw een aardige oefening in het berekenen van tabelletjes. De definitie van een NAND is in feite een “not-and”; we gebruiken het symbooltje ¬&

A	B	A ¬& B
0	0	1
0	1	1
1	0	1
1	1	0

De reden dat de NAND zo populair was, is dat je zowel de AND, de OR en de NOT kunt uitvoeren met NAND-schakelingen. Dus een schakelblokje voor de NAND (twee signalen erin, één resultaat-signaal eruit) is een soort universele bouwsteen voor computerschakelingen. De volgende drie vergelijkingen moet u maar eens proberen te bewijzen:

!A	=	A ¬& A		
A & B	=	! (A ¬& B)	=	(A ¬& B) ¬& (A ¬& B)
A | B	=	! A ¬& ! B	=	(A ¬& A) ¬& (B ¬& B)

Merk op dat bij de laatste twee regels de overgang van het middendeel naar het rechterdeel feitelijk bestaat uit het toepassen van de !A vergelijking uit de eerste regel. U ziet dat in het rechterdeel **alles** met NAND wordt gedaan. Daar was het ons om begonnen.

14 | Rekenen met bits

De zéér dapperen onder u mogen tenslotte nog meegaan in een demonstratie om te zien hoe Booleaanse operaties de grondslag vormen voor het “echte” rekenwerk dat in een computer plaatsvindt.

Stel we hebben twee getallen, A en B. Die willen we optellen, dus we zoeken de uitkomst U=A+B. Om het eenvoudig te houden gaan we werken met getallen A en B van elk 4 bits lang. Hun afzonderlijke bits noemen we A_3,A_2,A_1,A_0 resp B_3,B_2,B_1,B_0 . De uitkomst kan maximaal 5 bits lang worden; die noemen we U_4,U_3,U_2,U_1,U_0.

We proberen dus Booleaanse formules te vinden voor elk van de vijf bits, die de uitkomst zijn van het **optellen** van twee reeksen van vier bits elk. Bijvoorbeeld: 7+3=10 wordt tweetallig: 0111+0011=01010. Het is verstandig om op dit punt zelf een aantal optellingen in bitjes te oefenen (bijv. 1+2=3 (0001+0010=0011), 2+3=5 (0010+0011=0101), 5+3=8 (0101+0011=1000)) zodat u wat gevoel krijgt voor het gedrag.

Uit uw schooltijd weet u nog wel dat bij optellingen vaak een “een-onthouden” meespeelt. In het Engels heet dat een “carry-over”, en daarvoor voeren we aparte bits C_3,C_2,C_1,C_0 in.

De laagste bit van de uitkomst is eenvoudig: $U_0=A_0 \hat{} B_0$. De laagste carry is ook eenvoudig: $C_0 =A_0 \& B_0$. Daarna gaan we één bit hogerop, en proberen we de formules voor U_1 en C_1 te vinden.

Wanneer u de bovengenoemde eenvoudige oefeningetjes hebt gemaakt, dan kunt u in de volgende tabel controleren of u het gedrag van twee opgetelde bitjes, met een carry die vanuit de lagere positie kan meekomen, goed hebt doorzien:

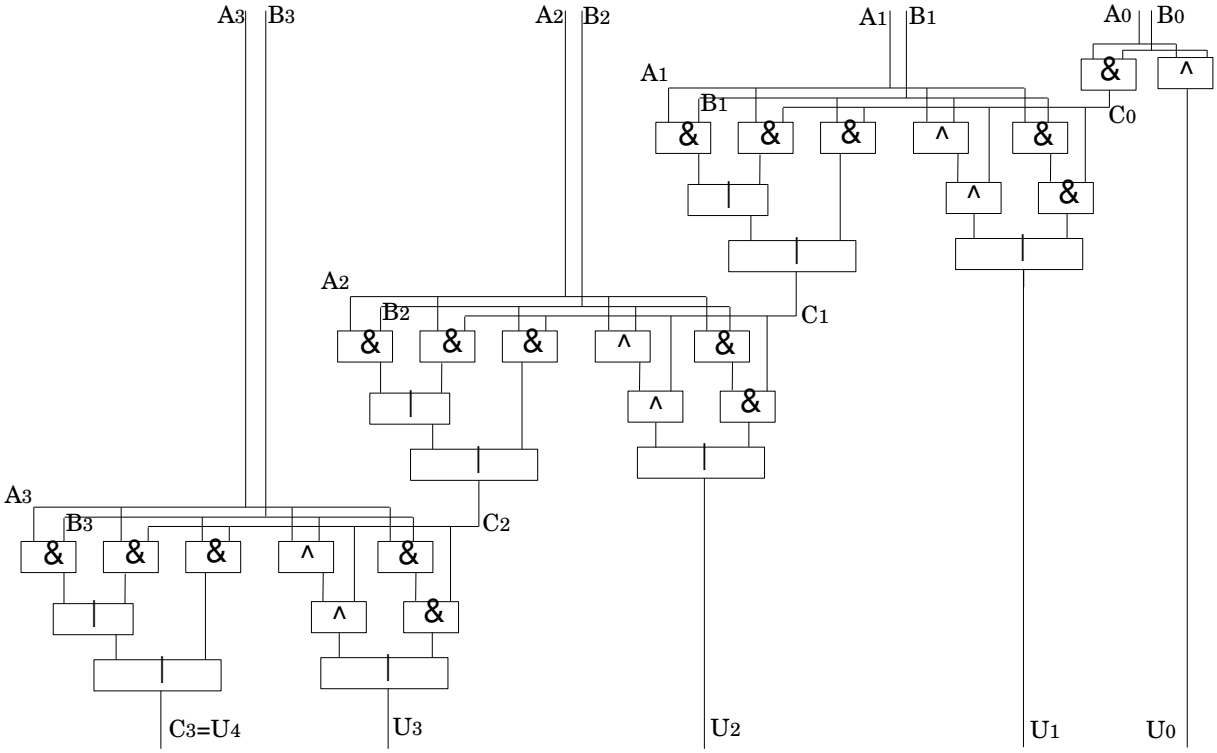
Wanneer u deze tabel kritisch bekijkt, dan ziet u de volgende structuur: U_1 wordt 1 bij één of drie aangeleverde 1-bits. Dat kun je opschrijven als:

$U_1 =(A_1 \hat{} B_1 \hat{} C_0)|(A_1 \& B_1 \& C_0)$

De carry C_1 wordt 1 bij minstens twee aangeleverde 1-bits. Een formule daarvoor is:

$C_1 =(A_1 \& B_1)|(A_1 \& C_0)|(B_1 \& C_0)$.

De formules voor U_2, C_2, U_3 en C_3 zien er hetzelfde uit als voor U_1 resp. C_1 . Tenslotte geldt dat $U_4 =C_3$, want de allerhoogste bit in een uitkomst kan alleen maar ontstaan als carry van de positie die er vlak onder zit. In onderstaande tekening stelt een T-splitsing van lijnen een verbinding voor, maar een kruispunt niet. Kruispunten zijn “ongelijkvloers”.



Als dit voorbeeld verslavend werkt dan mag u doorgaan met **vermenigvuldigen**, enzovoort. Na **worteltrekken** mag u ophouden, want dan hebt u waarschijnlijk geen leeg vel schrijfpapier meer in huis.