BASIC VERSUS PASCAL

H. J. C. OTTEN

In de microcomputerwereld is Basic een gevestigde hogere programmeertaal. Elke personal computer is uitgerust met een Basic in ROM. Naast Basic begint Pascal als hogere programmeertaal veld te winnen. Een voorbeeld daarvan is UCSD Pascal. Pascal stelt echter hogere eisen dan Basic aan een systeem. Waarom het de moeite waard is toch Pascal als programmeertaal toe te passen is het onderwerp van dit artikel.

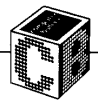
Zowel Pascal als Basic zijn aan universiteiten ontwikkeld om het onderwijs in programmeren te ondersteunen. Bij Basic was het uitgangsppunt een zo eenvoudig mogelijk te begrijpen taal maken. Bij Pascal lag de nadruk meer op een zo goed mogelijk gestructureerde taal om de studenten goede programmeertechnieken bij te brengen. De eenvoud van Basic heeft het tot de meest populaire taal voor personal computers gemaakt. De nadelen van de eenvoud zijn echter dat naarmate het programma groter wordt het slechter leesbaar en bij te houden is en de uitvoering ervan nogal traag.
Bovendien is gedwongen door de eenvoud iedereen Basic op zijn eigen wijze gaan uitbreiden waardoor er ontelbare Basic dialecten zijn ontstaan.
De gestructureerde opzet van Pascal levert heel wat beter leesbare programma's op en de rijkdom aan programma en data structuren beperken de aangebrachte uitbreidingen ook al omdat Pascal veel beter is gedefinieerd. De taal Pascal kent namelijk een standaard boek dat meteen de taal definieert: 'Pascal Manual and Report', geschreven door K. Jensen en N. Wirth. De laatste heeft de taal Pascal ontworpen.

Datastructuren

Een datatype dient ervoor informatie voor te stellen in een programma. Basic kent de datatypen real, integer en string. In Pascal zijn dat de real, integer en character. In UCSD Pascal is daar de string aan toegevoegd. Met deze eenvoudige datatypen zijn gestructureerde datatypen te construeren. In Basic hebben we het array. In Pascal kennen we naast het array ook de record en de set. Vooral de record is een krachtige structuur, verschillende datatypen kunnen daarin onder een noemer bij elkaar worden gebracht.
Pascal laat ook toe zelf datatypen te definiëren met de 'type' declaratie. De declaratie: type kaartkleur = (harten, klaveren, schoppen, ruiten) levert een nieuw datatype op dat net zo bruikbaar is als de andere datatypen. Basic heeft deze mogelijkheid niet. Pascal geeft de mogelijkheid informatie in de meest duidelijke vorm voor te stellen. Als we met speelkaarten werken, dan definiëren we gewoon het datatype speelkaart en werken verder met een speelkaart in het programma. In Basic is het vaak een kwestie van kunstgrepen die de duidelijkheid niet bevorderen.
Pascal onderscheidt zich van Basic als hogere programmeertaal door het pointertype, een dynamische variabele die in principe de datastructuur een onbeperkte mogelijkheid geeft en het programma een van te voren niet vastgelegd opneemvermogen voor informatie geeft. Met pointervariabelen kunnen de meest complexe datastructuren op een duidelijke manier worden gecreëerd, zoals gelinkte lijsten, bomen etc.

Programmastruren

Een programma bestaat in de eenvoudigste vorm uit een aantal opdrachten die na elkaar worden uitgevoerd. De sequentiële programmastructuur is dan ook de bouwsteen van een programma. In Basic is deze sequentiële structuur het duielijkst door het gebruik van regelnummers. Een regel, die een of meer statements bevat, is een afgesloten deel. In Pascal is deze beperking niet aanwezig, voor de uitvoer is het niet belangrijk of elke statement op een regel staat of over meerdere. Dit geeft de mogelijkheid de lay out van een programma bij Pascal zo in te delen dat de structuur van een programma duidelijk wordt.
Een programma wordt pas interessant als we gaan afwijken van de sequentiële structuur door terug te springen of vooruit door een aantal statements over te slaan. Meestal gebeurt dit aan de hand van een beslissing. In Basic hebben we het IF... THEN statement, in Pascal de uitgebreide IF... THEN... ELSE. (In sommige Basic implementaties ook.) De regelnummers in Basic dienen als eindpunt voor de sprong. In Pascal is iets dergelijks aanwezig maar er zijn andere en betere mogelijkheden. In Pascal kunnen we met BEGIN en END een aantal statements combineren tot een nieuw statement. Het statement achter THEN kan dan complexe vormen aannemen, in Basic kan dat alleen door een GOTO. De duidelijkheid is in Basic verloren, door de lay out in Pascal juist bevorderd door inspringen aan het begin van de regel.
De loop is een veel gebruikte programmastructuur. Een aantal statements wordt uitgevoerd tot aan een conditie is voldaan. Het testen van de conditie kan aan het begin van de loop gebeu-



ren (WHILE... DO) of aan het eind (REPEAT... UNTIL). Een veel gebruikte loop is de FOR...TO...DO loop, waarin variabele telkens met een vaste waarde wordt opgehoogd tot een eindwaarde is bereikt. In Pascal vinden we alle genoemde loop structuren, in Basic alleen de FOR loop. Loops in Basic worden dan ook met GOTO sprongen gemaakt, wat de duidelijkheid weer niet ten goede komt.

Een andere veel gebruikte programma structuur is de subroutine of procedure. Van subroutines maken we gebruik om een groepje veel gebruikte statements dat vaak wordt uitgevoerd in een subroutine te zetten en deze subroutine aan te roepen als we dat groepje statements nodig hebben. Dat bespaart een hoop dubbel werk.

In Basic springen we naar een subroutine door GOSUB, een sprong naar een regel waar de subroutine begint. Net als GOTO is dat weinig informatief over het doel van de subroutine.

In Pascal heten subroutines procedures en in plaats van een nummer kunnen we ze een suggestieve naam geven om het doel aan te geven. Dit is zo duidelijk dat procedures die maar eenmaal worden uitgevoerd ook zin hebben. Een programma bestaat dan uit een kort hoofdprogramma waarin de hoofdtaken door procedures zijn opgenomen. Zo'n hoofdprogramma is dan erg duidelijk door de goed gekozen procedure namen, de gelijkenis met spreektaal is soms frappant. Deze programma opbouw hangt samen met de top-down-programming methode waarbij een probleem wordt opgelost door het probleem in steeds kleinere stukjes op te delen. Pascal is daar uiterst geschikt voor, en programma's die op deze wijze zijn geschreven zijn snel te schrijven en goed te begrijpen. Wat een subroutine in Basic doet met de variabelen is bij aanroep van de subroutine niet duidelijk. In Pascal is elke procedure aan te roepen met een aantal parameters waar de procedure op werkt. Een procedure is op zich weer een programma met eventueel variabelen die alleen binnen die procedure bestaan. Wat een procedure doet is dus door de naam aan te geven en waarmee de procedure dat doet is door de parameters bij de procedure aanroep bekend.

Compiler versus interpreter

Het verschil tussen de gangbare Basic en Pascal implementaties is naast een taalverschil ook een verschil tussen interpreter (Basic) en compiler (Pascal). Het zijn beide oplossingen van het probleem een in een hoge programmeertaal geschreven programma door de computer te laten uitvoeren, maar de wijze van oplossen is geheel verschillend.

Bij een interpreter worden de programma opdrachten door een speciaal voor dat doel geschreven programma: de interpreter zelf, opdracht na opdracht ontcijferd en uitgevoerd. De interpreter zelf is in machinetaal geschreven, alle andere software kan in de interpretertaal worden geschreven. Een andere computer vereist alleen een andere interpreter, alle andere software blijft bruikbaar.

Een interpreter is interactief, omdat opdracht voor opdracht wordt uitgevoerd kan elke opdracht worden gevolgd en eventueel onderbroken. De opdrachten zijn in de hogere taal gegeven zodat wat de computer aan het uitvoeren is voor de gebruiker goed te volgen is. De interpreter is ook in staat fouten te ontdekken en zonodig de controle aan de gebruiker terug te geven.

Het interactieve karakter heeft als gevolg dat editing, het samenstellen van programmatekst, een onderdeel van de interpreter is.

De positieve kanten van een interpreter zijn als volgt samen te vatten. Ten eerste is een interpreter interactief en daardoor gebruikersvriendelijk en geeft een goede controle over de machine. Ten tweede, het in de programmeertaal geschreven programma, zoals een Basic programma, is het definitieve programma.

Nadelen van een interpreter zijn ook aanwezig. Het is een niet erg efficiënte oplossing, wat zich uit in een trage uitvoering, en een veel ruimte in beslag nemend programma. Daarnaast is een interpreterprogramma nodig, wat ook veel ruimte in beslag neemt omdat de hoge taal ver van de machinetaal afstaat.

Voor kleine machines is een interpreter toch een goede oplossing. Er wordt niet teveel geëist van achtergrond geheugens zoals floppy disk's etc. Meestal wordt de interpreter in ROM geplaatst, meteen na het opstarten kan worden gewerkt. De benodigde hoeveelheid geheugen is beperkt tot 8 Kbyte voor de interpreter in het algemeen. Gebruikers van een kleine machine zijn vaak niet erg ervaren zodat de gebruikersvriendelijkheid een voordeel is. Ook worden geen hoge snelheidseisen aan een klein systeem gesteld. Dit zijn de redenen dat de kleine personal computers met een Basic interpreter zijn uitgerust.

Een compiler is een vertaalprogramma. De programmatekst wordt door de compiler vertaald naar machinetaal. Het resultaat van dit vertaalproces is een machinetaalprogramma dat erg efficiënt met geheugen omgaat en natuurlijk erg snel is. Een goede compiler levert code af die nauwelijks onderdoet voor een in machinetaal geschreven programma. De compiler zelf is

een bijzonder groot programma, het vertalen van hoge naar machinetaal is niet eenvoudig. Bovendien moet zoveel mogelijk door de compiler worden gecontroleerd omdat het uitvoeren van de vertaalde code door de machine voor de gebruiker erg onduidelijk is, het verband met het programma en de vertaalde code is gering. Een compiler is maar een onderdeel van de programmatuur die nodig is. Eerst wordt de programmatekst samengesteld met een editor, dit levert de source. Deze source wordt door de compiler vertaald (gecompileerd) tot de code. Deze code kan door de machine worden uitgevoerd. Een compiler voert het programma niet uit zoals een interpreter doet. Een programma bestaat in twee vormen in de computer: de voor de gebruiker leesbare in hoge taal geschreven source en de vertaalde code die de machine kan uitwerken. Dit vereist meer achtergrondgeheugen. Bij het compileren is er erg veel geheugen nodig voor de compiler, die weer na het compileren vrij komt.
Voor het uitvoeren van de code is het gehele geheugen in principe beschikbaar.
De hier beschreven compiler is wel erg efficiënt maar ook erg onhandig. Het uitproberen van een programma is vrijwel onmogelijk en de eerste de beste fout laat de machine op hol slaan. In de praktijk werkt men dan ook meestal met een compiler-interpreter combinatie.
De compiler levert een tussencode op die door de interpreter kan worden verwerkt. De interpreter is veel compacter dan een Basic interpreter, want de tussencode staat vrij dicht bij de machine en niet bij de hoge taal. De compiler vertaalt immers de hoge taal naar de zo efficiënt mogelijke interpretertaal. Het is gebruikelijk om te spreken van een machinetaal waarvoor de compiler code levert. Deze machine wordt op de computer gesimuleerd door de interpreter. Deze compiler interpreter combinatie voegt de voordelen van compiler en interpreter samen tot een redelijk efficiënte oplossing, snel door de compiler en een goede controle door de interpreter. Overgaan op een andere computer vereist alleen het vertalen van de interpreter, ook de ingewikkelde compiler blijft bruikbaar.
Voor grotere systemen is de compiler interpreter combinatie de beste oplossing om een programma te ontwikkelen. Soms is het dan ook nog mogelijk een goed uitgetest programma tot machinetaal verder te compileren voor de snelheid. UCSD Pascal is een voorbeeld van zo'n compiler interpreter combinatie. Overzetten op een andere machine vereist alleen het herschrijven van de interpreter, zodat UCSD Pascal op vele machines al is geïmplementeerd zonder veel moeite.
De eisen die een compiler aan een systeem stelt zijn vrij hoog. Een goed en snel massageheugen is nodig, een programma is zowel als source en code aanwezig wat ruimte vraagt. Ook is naast de compiler een volledig operating system nodig met editor, interpreter, file management etc. Een minimale configuratie voor UCSD Pascal bestaat uit een microcomputer met 48K RAM en twee minifloppy disk drive's bijvoorbeeld.
Dat het combineren van compiler en interpreter een efficiënte oplossing is, blijkt wel als we zien dat het operating system geheel in de hoge programmeertaal is geschreven met alle voordelen zoals snelle programma ontwikkeling en gemakkelijk op peil te houden. Zo is UCSD Pascal geheel in Pascal geschreven, natuurlijk het ingewikkelde compiler programma ook. Alleen de interpreter voor de code is in machinetaal geschreven. Een hoge taal interpreter zoals Basic zou een hopeloos traag operating system opleveren, de Pascal versie is echter snel genoeg. Dat we bij deze Basic Pascal vergelijking interpreter tegen compiler zetten heeft een praktische oorzaak. We zien hoofdzakelijk Basic interpreters en Pascal compilers. Er zijn echter ook Basic compilers, maar die zijn aan grotere systemen voorbehouden.

Conclusie

Voor veeleisende problemen is Pascal een veel betere taal dan Basic. De duidelijke structuur die een Pascal programma uit zich zelf heeft en de grotere rijkdom aan data- en programmastructuren leveren beter leesbare en op peil te houden programma's op. Daarbij is Pascal een veel beter gedefinieerde taal dan Basic. Bij Basic is sprake van een babylonische spraakverwarring.
Voor kleine systemen is een Basic interpreter een goede oplossing die het programmeren in een hoge taal mogelijk maakt, omdat de eisen van een interpreter niet zo hoog zijn en een interpreter gebruikersvriendelijk is. Om het extreem te stellen: voor zogenaamde wegwerp programma's is Basic geschikt, serieuze programma's kunnen beter in Pascal worden geschreven.