SYSTEMATISCH LEREN PROGRAMMEREN

DEEL 1: TOP-DOWN PROGRAMMERING

HET SYSTEMATISCH ONTWERPEN

VAN SEQUENTIËLE ALGORITMEN

OVER EENVOUDIGE GEGEVENSSTRUCTUREN

EN DE REALISERING DAARVAN

IN ELAN.

C.H.A. KOSTER

November 17, 1998

Voorwoord

Dit leerboek is bestemd voor al diegenen die op systematische wijze willen leren programmeren: naast Informatica studenten in het wtenschappelijk onderwijs ook bijvak-studenten, leraren in het voorbereidend onderwijs en studenten aan hogescholen, maar ook rijpere scholieren, hobbyisten en personen werkzaam in de informatica industrie.

Vanwege de grote spreiding in de doelgroep is het leerboek zo ingericht dat het slechts een beperkt beroep doet op wiskundige kennis. Hoewel juist de wiskunde een bron kan zijn van leerrijke en nuttige voorbeelden zijn alle voorbeelden zo gekozen dat ze enkel elementaire kennis van de wiskunde vooronderstellen. Het boek is evenmin gericht op enigerlei specifiek toepassingsgebied van de informatica; door opgaven uit een bepaald toepassingsbied te geven kan een docent zelf een dergelijke specialisering bereiken.

Het centrale thema van alle programmeermethoden is het beheerst gebruik van abstractie. In dit leerboek wordt vooral een beroep gedaan op het linguïstisch abstractievermogen van de lezer, meer dan op het wiskundig abstractievermogen. Linguïstische abstracties kunnen echter worden gezien als pre-wiskundige abstracties, die in een wat later stadium (als voldoende kennis van de logica en de discrete wiskunde is opgebouwd) geformaliseerd en gepreciseerd kunnen worden. Vandaar dat onderwerpen als correctheidsbewijs en formele specificatie pas in een tweede deel ([KOS91]) aan bod komen.

De programmeertaal die in dit boek gebruikt wordt, de onderwijstaal Elan, kan worden beschouwd als een dochter van de taal ALGOL68, speciaal ontworpen voor het onderwijzen, aanleren en oefenen van de systematische programmering. De gevolgde didactische methode is zeker ook voor andere programmeertalen toepasbaar, voor zover daarin de noodzakelijke concepten aanwezig zijn. Maar alleen Elan is ontworpen als de belichaming van een aantal systematische programmeermethoden:

- Top-Down programmering
- recursieve programmering
- Bottom-Up programmering
- modulaire programmering.

Dit eerste deel behandelt uitgebreid de Top-Down programmering, die het hart uitmaakt van vrijwel iedere systematische programmeermethode. Het maakt gebruik van het feit dat verfijningen in Elan geen papier-en-potlood techniek zijn, maar een centraal taalmiddel, ondersteund door een bijzondere programmeeromgeving. Het gaat ook uitgebreid in op de mogelijkheden die het gebruik van recursie biedt. Het tweede deel houdt zich bezig met de Bottom-Up en modulaire programmering.

Deze ietwat ongebruikelijke, wat dogmatische indeling wordt mede gemotiveerd door een verschil in doelgroep: het tweede deel richt zich op de meer professionele gebruiker die zich op systematische wijze wil bekwamen in het ontwerp en de realisering van niettriviale programma's. Het tweede deel is daarom ook beduidend formeler in zijn benaderingswijze.

De auteur heeft gepoogd zeer zorgvuldig en consistent te zijn in de invoering van begrippen en nederlandse termen, met betrekking tot zowel de programmeertaal als het programmeerproces zelf. Voor zover mogelijk wordt standaard terminologie gebruikt, maar jargon wordt vermeden. De begrippen en termen die hier ingevoerd worden zijn niet alleen van toepassing op Elan maar kunnen worden gebruikt om het programmeren in alle programmeertalen te begrijpen. Wie systematisch programmeren met Elan heeft geleerd moet in staat zijn het geleerde ook met andere programmeertalen toe te passen, en snel een nieuwe programmeertaal te verwerven.

Het boek is zodanig ingericht dat het geschikt is voor zelfstudie. Wel moet er op worden gewezen dat daarnaast omvangrijke praktisch oefenen noodzakelijk is om bepaalde vaardigheden aan te leren en om door midddel van een confrontatie met de "echte" computer de toepasbaarheid en beperkingen van de aangeboden methoden aan den lijve te ondervinden. Voor dit praktische werk dient een computer met een Elan implementering beschikbaar te zijn.

In deze tweede, herziene druk zijn enige formuleringen aangescherpt en een groot aantal verschrijvingen gecorrigeerd. Enige voorbeelden werden toegevoegd. Het hoofdstuk over linguistische toepassingen werd uitgebreid en een hoofdstuk over Computergrafiek werd toegevoegd.

Op het moment worden zowel de tekst van dit boek als de Elanprogrammeeromgeving herzien. Naar uw open aanmerkingen ben ik zeer benieuwd.

Nijmegen, juni 1995

C.H.A. Koster

Chapter 1

Inleiding

Bij het programmeren willen we een computer zover krijgen dat hij voor ons iets doet wat wij willen. We moeten daartoe in de eerste plaats aan onszelf en vervolgens aan de computer duidelijk moeten maken wat er gedaan moet worden. Het blijkt telkens weer dat het knelpunt daarbij zit bij de mens, niet bij de computer: het is heel moeilijk precies te beschrijven wat je bedoelt. Je moet leren precies te beschrijven wat je wilt dat er voor je gedaan wordt. Zo'n beschrijving heet een **algoritme**[1].

Door de algoritme op te schrijven in een formalisme dat de computer kan uitvoeren (een **programmeertaal**) krijg je een door de computer uitvoerbaar algoritme (een **programma**). Dit dictaat gaat over het systematisch programmeren, gezien als het afleiden van algoritmen geschreven in een concrete programmeertaal (in dit geval Elan).

Bij het opstellen van algoritmen voor de computer kunnen we veel leren van andere beschrijvingen bedoeld om mensen te vertellen wat ze moeten doen: handleidingen, recepten, voorschriften. We zullen daarom enige algoritmen uit het dagelijkse leven nader bekijken, aan de hand waarvan belangrijke kenmerken van algoritmen duidelijk worden. Met name zal het belang van abstractie bij het opstellen van algoritmen blijken.

1.1 Algoritmen in het dagelijkse leven

De volgende voorbeelden, uit het leven gegrepen of geconstrueerd, tonen een aantal belangrijke kenmerken van algoritmen.

1.1.1 Telefoon met muntautomaat

Het eerste voorbeeld zou als handleiding voor het gebruik van een telefoon met muntautomaat in een telefooncel kunnen hangen.

gebruiksaanwijzing

1. Neem de hoorn van de haak
2. wacht op de kiestoon
3. werp muntstukken in de gleuf,
   minimaal een kwartje
4. kies het gewenste nummer.

Wanneer we deze algoritme wat nader beschouwen, vallen enige belangrijke punten op.

- Deze algoritme is een beschrijving die ten doel heeft het handelen van een mens te sturen. De beschrijving is bedoeld als *uitvoerbaar*, dit in tegenstelling tot bijvoorbeeld een vermelding, op een plaatje bevestigd aan de telefoon, dat het apparaat eigendom is van de PTT.
- De uitvoering van de algoritme verloopt in stappen. De opeenvolging van stappen bij een bepaalde uitvoering van de algoritme noemt men een door de algoritme beschreven **proces**. De uitvoerder van de algoritme, in dit geval de mens, noemt men de **processor**.
- Een stap in de uitvoering van de algoritme bestaat op zijn beurt weer uit de uitvoering van een of meerdere (andere) algoritmen, die in deze handleiding uitsluitend door een Xnaam als **neem de hoorn van de haak** worden aangeduid. Deze benoemde algoritmen worden blijkbaar als *elementair* opgevat, dat wil zeggen: enkel met hun naam aangeduid, niet op hun beurt in stappen uit elkaar geplozen. De gegeven algoritme is *samengesteld* uit elementaire algoritmen.
- Wie deze algoritme wil uitvoeren moet de taal kennen waarin de algoritme beschreven is. Hij moet met name de elementaire algoritmen kennen, en ze kunnen uitvoeren. Hij moet weten, wat de kiestoon is en hoe men kwartjes in de gleuf werpt.
- Van een algoritme wordt verlangd dat hij voldoende *nauwkeurig* is, d.w.z. dat zowel elke stap als de bedoelde volgorde van uitvoering niet voor verschillende interpretaties vatbaar zijn. Daartoe moet de algoritme op een welgekozen niveau van detaillering (**abstractieniveau**) geformuleerd worden.

Om duidelijk te maken hoe men de hoorn van de haak neemt, kunnen we natuurlijk aangeven welke hand naar de hoorn moet worden uitgestrekt (daarbij verschil ma-

1. Dit merkwaardige woord heeft niets te maken met het woord *logarithme*. En al helemaal niets met ritme. Het is afgeleid van de naam van de centraal-aziatische geleerde *Al khowarizmi*, de schrijver van het boek met de titel *Kitab al Jabr w'al muqaballah*, waarvan op zijn beurt het woord *algebra* is afgeleid. In de oudheid stond de algoritme van Euklides bekend als "de algoritme".

kend tussen links- en rechtshandigen), met welke kracht (in Newton) de verschillende vingers de hoorn dienen te omvatten, en met welke snelheid dat uiteinde van de hoorn, waaraan niet de draad verbonden is, met deszelfs vlakke zijde tot op 1 mm van het rechteroor van de uitvoerende gebracht dient te worden (bij ontstentenis van het rechteroor neme men het linker). Maar dan zijn we er nog lang niet: we moeten definiëren, hoe en wanneer de individuele spieren van hand en arm ingespannen worden.

De beschrijving dreigt in een bodemloos moeras van steeds gedetailleerder uitvoerigheid te verzinken. Alleen een geschikte keuze van niet al te gedetailleerde elementaire algoritmen **(hoorn van de haak nemen, wachten op kiestoon)** en een formulering van de algoritme op het daardoor gedefinieerde niveau van abstractie, stelt ons in staat nauwkeurig te zijn.

Nauwkeurigheid heeft slechts zin met betrekking tot een bepaald niveau van abstractie.

- De algoritme, d.w.z. de tekst, moet *eindig* zijn, voor mensen zelfs kort, om begrepen te kunnen worden. De uitvoering van de algoritme (het proces) is daarentegen niet noodzakelijkerwijs eindig. Een handleiding voor het lopen in een tredmolen is daar een voorbeeld van.

1.1.2 Kookrecept

In kookboeken vindt men *recepten*, d.w.z. handleidingen voor het bereiden van gerechten, geschreven met een typisch taalgebruik en met typische notationele conventies.

> Kip á la Marseillaise.
>
> 2 jonge haantjes, 6 tomaten, 2 paprika's, 1/2 kopje witte wijn, 1/2 kopje bouillon, 1/2 citroen, zout, peper, tijm en een teentje knoflook.
>
> Verdeel de haantjes in stukken. De stukken zouten en peperen en in de in water gedrenkte Römertopf leggen. Voeg daaraan de fijn gesneden paprika's, de in vieren gesneden, geschilde en van hun kern ontdane tomaten en de knoflook u.d.k. toe. Roer daar de witte wijn en bouillon die met zout, peper, tijm en citroensap gekruid is doorheen. De gesloten Römertopf in de oven plaatsen en het gerecht bij een temperatuur van 225 graden in ca. 90 minuten gaar laten worden.

Voor een niet al te onervaren kok is dit recept een nauwkeurige, uitvoerbare en uiteraard eindige beschrijving van een proces, dat hij onder de naam "bereiding van kip á la Marseillaise" kent (zie [EXN70]).

De uitvoering van deze algoritme bestaat uit het stap voor stap uitvoeren van andere algoritmen (in stukken verdelen, kruiden) die werken op bepaalde objecten (haantjes, peper). Ook aan deze algoritme vallen weer enige belangrijke zaken op.

- Aan het begin van deze beschrijving worden de noodzakelijke ingrediënten opgenoemd, om vervolgens tot een gerecht verwerkt te worden. Meer in het algemeen: de algoritme beschrijft operaties op benoemde **objecten**, zoals bijvoorbeeld het schillen van tomaten en het fijn snijden van paprika's. Sommige objecten moeten voor de uitvoering van de algoritme aanwezig zijn *(invoer)*, andere staan als resultaat van de uitvoering van de algoritme ter beschikking *(uitvoer)* en weer andere bestaan alleen maar tijdelijk, tijdens het proces *(locale objecten)*.
- Objecten kunnen *elementair* zijn, d.w.z. in de algoritme met hun naam worden aangeduid en geen verdere structuur vertonen, of ze kunnen als *samengesteld* beschouwd worden, wanneer hun structuur voor de uitvoering van de algoritme van wezenlijk belang is. In het vorige voorbeeld bestond de telefoon uit een hoorn, een haak, een kiesschijf enz. Ook dit is een kwestie van abstractieniveau — voor de chemicus zal keukenzout er heel anders uitzien dan voor een kok. En de fysicus interesseert zich zelfs voor de toestand van de natriumionen in het zout.
- Er is een sterke samenhang tussen algoritmen en objecten: de keuze van de elementaire objecten bepaalt het niveau van detail van de daarop werkende elementaire algoritmen en omgekeerd legt de keuze van de elementaire algoritmen vast op wat voor objecten ze kunnen worden toegepast.

Samen vormen de algoritmen en objecten de bouwstenen van het "algoritmisch universum", dat we als volgt kunnen indelen:

- algoritmen
  - elementaire algoritmen
  - samengestelde algoritmen
- objecten
  - elementaire objecten
  - samengestelde objecten

1.1.3 Breipatroon

In modetijdschriften treft men breipatronen aan zoals het volgende (hier niet volledig weergegeven) voorbeeld.

> Pullover met strepen (leeft. 3-4 jr)
>
> Materiaal: ca. 250 gr. middelzware wol met dralon en mohair in de kl. rose, wit, donkergr. en lichtgr.
>
> Ribbels: 2s.r., 2s.av. afw.
>
> Glad r.: Heeng. t.r.s., terugg. t.av.s.
>
> Stekenber.: 26 s. en 37 t. = 10 x 10 cm.
>
> Strepen: * 4t.rose, 4t.wit, 4t.donkergr., 4t.lichtgr., 2t.rose, 2t.wit, 2t.donkergr., 2t.lichtgr., vanaf * 3 maal herh. en eindig. m. 4 t.rose, 4t.wit, 4t.donkergr.=108t.
>
> Rugpand: Met rose 130s. opz. Strepen patroon volgen. Eerst 4t. ribbels voor het boord en dan glad r. Na 78t. (21 cm) ....

Weer kan een aantal opmerkingen gemaakt worden.

- De algoritme begint weer met het noemen van de objecten die als invoer dienen.
- Deze algoritme maakt niet alleen gebruik van algoritmen die aan iedere breier bekend horen te zijn, die als het ware tot de taal van het breien behoren (**concrete algoritmen**), maar ook zien we dat in de algoritme zelf een aantal nieuwe algoritmen wordt gedefinieerd die blijkbaar niet zo algemeen bekend zijn en speciaal worden ingevoerd om het formuleren van deze algoritme te vereenvoudigen, zoals Ribbels en Glad r. Deze **abstracte algoritmen** worden op hun beurt gedefinieerd in termen van andere algoritmen zoals s.r. (steken recht) en t.av.s (toeren averechte steken). Zo'n abstracte algoritme wordt elders toegepast door het noemen van zijn naam *(opgeroepen)*.
- Sommige algoritmen krijgen bij hun oproep een of meer (**parameters**) mee, een getal of een object waarop zij werken. Het resultaat van het oproepen van een dergelijke algoritme is afhankelijk van de bij die oproep meegegeven nadere informatie. Bijvoorbeeld 4t. wit zal een witte streep in de pullover geven terwijl 2t. donkergr. een wat smallere donkergroene streep geeft.
- In de deelalgoritme strepen is een positie gemarkeerd met * en wat verderop in deze deelalgoritme staat de aanwijzing om het gedeelte vanaf * 3 maal te *herhalen*. In de algoritme wordt dus verwezen naar een ander punt in dezelfde algoritme. De herhaling is een stijlvorm die we in veel handleidingen tegenkomen.
- In de algoritme vinden we een aantal aanwijzingen, die de uitvoerder in staat stellen om tijdens de uitvoering de gemaakte vorderingen te controleren (Stekenber. en (21 cm)). Deze aanwijzingen maken een controle mogelijk op de plausibiliteit van de algoritme en op de uitvoering daarvan, maar zijn zelf geen algoritmen.

Voor iemand die de breikunst niet machtig is, is bovenstaande algoritme bijna niet te begrijpen. Hij is geschreven in een taal die alleen voor de kenners van de concrete algoritmen van het breien te volgen is. Maar zonder het gebruikte *formalisme* en het gekozen abstractieniveau zouden waarschijnlijk de bewegingen die de handen, de breinaalden en de woldraad tijdens het breien moeten maken apart beschreven dienen te worden, wat op zich weer het probleem zou oproepen hoe deze bewegingen te beschrijven.

Bovendien worden in de beschrijving zelf nog enige nieuwe begrippen als abstracte algoritmen ingevoerd. Dat is een van de mooiste eigenschappen van de menselijke taal: dat de spreker of schrijver in een tekst vrijelijk nieuwe begrippen kan invoeren, door hun relatie met andere (concrete) begrippen aan te geven. Programmeertalen bieden iets vergelijkbaars.

Hoe nauwkeurig men ook probeert handelingen algoritmisch te beschrijven, er blijven altijd enige elementaire algoritmen over waarvan de betekenis in de algoritme niet wordt aangegeven. Deze worden opgevat als concrete, tevoren bekende algoritmen. Een algoritme is altijd zo geschreven dat van bepaalde details van het te beschrijven proces wordt afgezien. Zulk een **abstractie** is onvermijdelijk.

De basis van alle abstractie is het *benoemen*: je geeft een algoritme een naam, en gebruikt dan die naam in plaats van de gedetailleerde algoritme (daarmee wordt tevens de hele beschrijving kort gehouden). De zo ingevoerde *abstracte algoritmen* kunnen op hun beurt worden gebruikt als elementaire algoritmen.

1.1.4 Autorijden

Het volgende voorbeeld zou aan een instructieboekje voor autorijden ontleend kunnen zijn.

Vertrek vanuit parkeerstand

1. Wachten tot de directe omgeving van de auto vrij is (spelende kinderen!), handrem aantrekken, versnelling in z'n vrij zetten en indien de wagen koud is de choke uittrekken.
2. De startmotor gedurende korte tijd inschakelen.
3. Herhaal vanaf stap 2 indien de motor niet loopt.
4. Over de linker schouder kijken en wachten tot de weg vrij is.
   Linker voet: koppeling intrappen.
   Rechter hand: van de handrem zetten.
5. Linker voet: koppeling ingetrapt houden.
   Rechter hand: in de eerste versnelling zetten.
6. Linker voet: koppeling voorzichtig op laten komen.
   Rechter voet: gas geven.
   Beide handen: het stuurwiel naar links draaien.
7. Linker voet: koppeling loslaten.
   Rechter voet: gas geven.
   Beide handen: het stuurwiel zo draaien dat de auto in de juiste richting rijdt.

Over de structuur van deze algoritme merken we het volgende op:

- De algoritme bestaat uit een aantal genummerde stappen die strikt na elkaar (**serieel**) uitgevoerd moeten worden.
- De eerste stap bestaat uit 5 onderdelen die **collateraal**, dwz in een willekeurige volgorde, eventueel gelijktijdig, uitgevoerd kunnen worden.
- De laatste drie stappen bevatten onderdelen, die noodzakelijkerwijs gelijktijdig (**parallel**) uitgevoerd moeten worden.

- Stap 3 bestaat uit de **voorwaardelijke uitvoering** van stap 2 en een der onderdelen van stap 1 bevat een voorwaarde.

De collaterale onderdelen van de algoritme kunnen naar believen parallel uitgevoerd worden indien voldoende processoren, in dit geval handen en voeten, voorhanden zijn, maar ze kunnen ook in de een of andere volgorde achter elkaar of door elkaar heen uitgevoerd worden zonder dat daardoor het resultaat beïnvloed wordt. De uitvoering zou hoogstens wat meer of minder tijd kunnen vergen. De parallele onderdelen van de algoritme moeten echter wel gelijktijdig uitgevoerd worden om tot het gewenste resultaat te komen.

We noemen een algoritme **sequentieel** als hij geen wezenlijk gebruik maakt van parallelliteit. Deze algoritme was dus niet-sequentieel. Om een aantal redenen blijken niet sequentiële algoritmen veel moeilijker te ontwerpen en te begrijpen dan sequentiële. Daarom beperken we ons in het vervolg tot sequentiële algoritmen.

1.2 Korte samenvatting

Een algoritme is een *precieze*, *eindige* beschrijving van een of meer *processen*. Een algoritme kan *elementair* zijn, dat wil zeggen slechts met zijn *naam* worden aangeduid, dan wel *samengesteld* zijn uit *oproepen* van andere algoritmen.

Algoritmen werken op *objecten*, die ze meekrijgen als *invoer* of als parameter bij hun oproep. Door de uitvoering van de algoritme kan de toestand van de objecten gewijzigd worden en kunnen nieuwe objecten ontstaan. Objecten kunnen eveneens elementair zijn, dan wel samengesteld zijn uit andere objecten.

De tekst van de algoritme bestaat in hoofdzaak uit

- algoritmen
  - elementaire algoritmen
    - concrete elementaire algoritmen
    - abstracte elementaire algoritmen
  - samengestelde algoritmen
- objecten
  - elementaire objecten
    - concrete elementaire objecten
    - abstracte elementaire objecten
  - samengestelde objecten

Een algoritme is in principe *uitvoerbaar* voor een *processor*, een mens of automaat die het *formalisme* waarin de algoritme is genoteerd kan interpreteren en de betekenis van de elementaire algoritmen kent. Door de algoritme op te schrijven in een programmeertaal bereiken we dat hij door een computer kan worden uitgevoerd.

In het vervolg beperken we ons tot sequentiële algoritmen, d.w.z. algoritmen waarbij strikte gelijktijdigheid (parallelliteit) van uitvoering geen rol speelt.

Chapter 2

Notatie van algoritmen

In het vorige hoofdstuk hebben we min of meer intuïtief een aantal aspecten van algoritmen en objecten gevonden. In dit hoofdstuk voeren wij voor de voornaamste daarvan een notatie in, zoals deze geldt voor de programmeertaal Elan.

Wij zullen niet meteen beginnen met een systematische behandeling van de programmeertaal maar proberen de aandacht veeleer te richten op het stapsgewijs ontwikkelen van algoritmen. Doel van dit hoofdstuk is te wennen aan een aantal essentiële algoritmische ingrediënten van programmeertalen als acties, voorwaarden, besturingsstructuren en verfijningen, onder vermijding (in eerste instantie) van moeilijke concepten als object, type en gegevensstructuur.

2.1 Karel de Robot

Wij zullen in dit hoofdstuk gebruik maken van een didactisch model, ontleend aan het boek Karel the Robot [PAT81]. Dit model, dat duidelijk voortbouwt op ideeën van S. Papert [PAP80], stelt ons in staat in eerste instantie het “object”-begrip van de informatica te vermijden en in plaats daarvan een beroep te doen op de dagelijkse ervaringswereld.

Karel de robot leeft in een eenvoudige algoritmische wereld in het platte vlak. Hij brengt zijn tijd door op de hoeken van straten. Op het hiervolgende plaatje ziet u hem staan, op de hoek van de tweede laan en de achtste straat, met zijn neus naar het oosten.

```
. . . . . . . . . .
. . . . . . . . . .
. > . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

Kareltje kan zijn neus naar elk van de vier windstreken draaien. En hij kan zijn neus achterna lopen, tot de volgende hoek. Bij het begin van de wereld staat hij in de oorsprong (straat 1, laan 1) in de linkeronderhoek van het beeldscherm, met zijn neus naar het noorden.

In Kareltje’s wereld bevinden zich, behalve Kareltje zelf ook piepballen en muurdelen. Een muurdeel kan een straathoek blokkeren. Piepballen kunnen op straathoeken liggen. Kareltje kan op de hoek waar hij staat een piepbal oprapen en in zijn zak steken of juist uit zijn zak halen en neerleggen. Daartoe heeft hij een zak die, net als de zak van Sinterklaas, willekeurig veel piepballen bevat en er nog meer kan bevatten. Op het beeldscherm wordt een muurdeel aangegeven als `X`, en een piepbal als `o`.

```
. . . . . . . . . .
. . X X X X . . . .
. . . . . X o . . .
. . . . . X o . . .
. . . . . X V . . .
. . . . . X . . . .
. . . . . X X X X X
. . . . . . . . . X
. . . . . . . . . X
. . . . . . . . . .
```

Om te vermijden dat Kareltje uit het zichtbare deel van zijn wereld wegloopt is het beeldscherm omringd met een (net buiten het beeld vallende) denkbeeldige muur.

Kareltje kan niet op een straathoek terecht waar een muurdeel staat. Vandaar dat hij soms moet omlopen om te komen waar wij dat willen.

2.2 Kareltje haalt zijn ochtendkrant (1)

Aan de hand van een voorbeeld willen we Kareltje nader preciseren en tegelijk de bedoelde algoritmische notatie invoeren. Kareltje ligt in bed in zijn huis, met zijn gezicht naar het westen. Voor de buitenmuur van zijn huis ligt een krant, in de gedaante van een piepbal.

We willen een algoritme voor Kareltje schrijven om zijn krant op te halen en deze mee naar bed te nemen.

2.2.1 Eerste aanzet

Wij kunnen Kareltje’s taak opsplitsen in drie delen, die hij achter elkaar moet uitvoeren.

```
ga naar voortuin;
pak krant;
ga terug naar bed
```

Ieder van de drie hier optredende zinnetjes beschouwen wij als de naam van een algoritme. We schrijven

```
. . . . . . . . . . . .
. . . . X X X X X X . .
. . . . X . . . . X . .
. . X X X . . . . X . .
. . . . . . . . . X . .
. . X X X . . . . X . .
. . . o X . . . . X . .
. . . . X . . . . X . .
. . . . X . . . . X . .
. . . . X < . . . X . .
. . . . X X X X X X . .
. . . . . . . . . . . .
```

deze namen op, gescheiden door puntkomma's, om aan te geven dat ze sequentieel (en in de gegeven volgorde) moeten worden uitgevoerd. Zo'n reeks deelalgoritmen, gescheiden door puntkomma's, noemen we een **paragraaf**.

Namen van algoritmen kiezen we zodanig dat ze aangeven *wat* de uitvoering van de algoritme bewerkstelligt. We gaan er dus bij de rest van deze beschrijving van uit dat de namen welgekozen verbale formuleringen zijn van wat er gebeuren moet.

Natuurlijk zou Kareltje een heel gek robotje zijn als zo'n algoritme als `ga naar voortuin` een concrete algoritme was, dat wil zeggen een instructie uit zijn grondrepertoire. Veeleer ligt het voor de hand dat we nu nader gaan beschrijven *hoe* de algoritme `ga naar voortuin` uitgevoerd moet worden in termen van eenvoudiger algoritmen. De notatie die we daarbij gebruiken heet een **verfijning**. Een verfijning geeft een naam aan een paragraaf.

```
ga naar voortuin:
  loop langs muur;
  loop door ingang;
  loop naar krant.
```

Deze verfijning definieert de algoritme `ga naar voortuin` als de sequentiële uitvoering van de genoemde algoritmen. Daarmee zijn we nog niet veel wijzer, omdat deze nog veel te weinig algemeen zijn om ze bij Kareltje's concrete algoritmen te laten horen. Toch hebben we al weer wat aan detail gewonnen. We definiëren analoog:

```
ga terug naar bed:
  loop naar ingang terug;
  loop door ingang terug;
  loop langs muur terug.
```

Bij wijze van krant nemen we (natuurlijk) een piepbal.

```
pak krant:
  pak piepbal.
```

2.2.2 Concrete algoritmen van Kareltje

Wat kent Kareltje nu wel aan concrete algoritmen? Wij kiezen in ieder geval de volgende:

- `start kareltje` doet de wereld van Kareltje op uw beeldscherm ontstaan; Kareltje verschijnt in de oorsprong, met zijn neus naar het noorden.
- `rechtsom` doet Kareltje 90 graden naar rechts draaien op zijn platte wereld.
- `linksom` eenzelfde draaiing naar links.
- `stap` doet hem lopen naar de volgende straathoek in de richting van zijn neus. Daarbij moeten wij oppassen dat Kareltje niet poogt door een muur heen te lopen.
- `pak piepbal` laat Kareltje een piepbal van de huidige straathoek oppakken en in zijn zak steken.
- `leg piepbal` laat Kareltje een piepbal uit zijn zak halen en op de huidige straathoek neerleggen. Daarbij moeten we er aan denken dat er op een straathoek maximaal één piepbal kan liggen.
- `stop` maakt op nette wijze een einde aan de wereld van Kareltje.

Deze instructies passen bij een simpel robotje als Kareltje. Later zullen we nog enige andere kiezen.

2.2.3 Vervolg

In termen van deze concrete algoritmen kunnen we nu de resterende verfijningen definiëren.

```
loop langs muur:
  rechtsom;
  stap;
  stap;
  stap;
  stap;
  stap.

loop door ingang:
  linksom;
  stap;
  stap;
  stap;
  stap.

loop naar krant:
  linksom;
  stap;
  stap;
  linksom;
  stap;
  stap.
```

2.2.4 De begrensde herhaling

De bovenstaande algoritmen zijn eerlijk gezegd nogal saai, omdat er zovele herhalingen van de algoritme `stap` in voorkomen. Als we Kareltje wat grotere afstanden willen laten lopen wordt dat nog erger! We hebben blijkbaar behoefte aan een manier om te kunnen aangeven dat een paragraaf een bepaald aantal malen herhaald wordt.

Elan kent daartoe een notatie, de **begrensde herhaling**:

```
UPTO aantal
REP
  handeling
ENDREP
```

wat betekent dat `aantal` maal de `handeling` herhaald wordt.

Wij kunnen nu schrijven (al is 't eerlijk gezegd niet veel korter, het is wel minder saai):

```
loop langs muur:
  rechtsom;
  UPTO 5
  REP
    stap
  ENDREP.
```

We krijgen de tekst van de verfijning nog wat korter (zonder dat dit voor de betekenis enig effect heeft) door een wat minder wijdlopige layout te nemen.

```
loop langs muur:
  rechtsom;
  UPTO 5 REP stap ENDREP.
```

De verfijning `loop naar krant` wordt ook iets eenvoudiger.

```
loop naar krant:
  linksom;
  UPTO 2 REP stap ENDREP;
  linksom;
  UPTO 2 REP stap ENDREP.
```

Wij kunnen nog meer gebruik maken van de regelmaat bij de realisering van deze algoritme.

```
loop naar krant:
  UPTO 2
  REP
    linksom;
    stap;
    stap
  ENDREP.
```

We kunnen zelfs schrijven:

```
loop naar krant:
  UPTO 2
  REP linksom;
    UPTO 2 REP stap ENDREP
  ENDREP.
```

Laten we de deelalgoritmen van een paragraaf zijn **units** noemen (zo heten ze volgens de grammatica van Elan). Zo bestaat deze laatste paragraaf uit drie units, waarvan de eerste een herhaling is; de daarin optredende herhaalde paragraaf bestaat uit twee units, waarvan de tweede een herhaling is. Een herhaling mag op zijn beurt als unit in een herhaling optreden. Men noemt dit een *genest* gebruik van de herhaling (denk maar aan een nest schalen, de een binnen de ander).

We werken nu de overige verfijningen uit.

```
loop naar ingang terug:
  keer om;
  UPTO 2 REP stap ENDREP;
  rechtsom;
  UPTO 2 REP stap ENDREP.

keer om:
  rechtsom;
  rechtsom.
```

Voor hetzelfde geld hadden we natuurlijk ook linksom kunnen draaien.

```
loop door ingang terug:
  rechtsom;
  UPTO 4 REP stap ENDREP.

loop langs muur terug:
  rechtsom;
  UPTO 5 REP stap ENDREP.
```

Ons eerste programma is nu compleet. Met behulp van de bij de Elan-vertolker geleverde Kareltje-omgeving kan het ook worden uitgevoerd, zodat we Kareltje over het scherm kunnen zien stappen.

De structuur van het programma kunnen we inzichtelijk maken door middel van een diagram dat poogt voor elke verfijning aan te geven welke andere verfijningen hij gebruikt, als volgt:

```
programma
| ga naar voortuin
| | loop langs muur
| | loop door ingang
| | loop naar krant
| pak krant
| ga terug naar bed
| | loop naar ingang terug
| | | keer om
| | loop door ingang terug
| | loop langs muur terug
```

2.2.5 Tussenbalans

We hebben tot nog toe twee samenstellingsmechanismen **control**structuur voor algoritmen leren kennen:

- de *paragraaf*, bestaande uit units, gescheiden door puntkomma's, en
- de *begrensde herhaling*, geschreven met de sleutelwoorden `UPTO ... REP ... ENDREP`.

Men noemt zulke samenstellingsmechanismen *controlstructuren*.

Bovendien hebben we gezien hoe abstracte algoritmen met behulp van *verfijningen* kunnen worden gedefinieerd.

2.3 Kareltje haalt zijn ochtendkrant (2)

Wij maken nu de opgavestelling wat losser om daarmee nog wat meer taalconstructies en begrippen te kunnen illustreren. We nemen dezelfde opgavestelling als hiervoor, behalve:

- de krant ligt niet op een vaste plaats, maar ergens in Kareltje's voortuin (op een van de in het volgende plaatje met ? aangegeven plaatsen)
- Kareltje begint niet noodzakelijk met zijn neus naar het westen, zodat we hem eerst met zijn neus naar het noorden moeten draaien. In het plaatje geven we Kareltje, zolang we niet weten welke kant hij opkijkt, aan met een K.

```
. . . . . . . . . . . .
. . . . X X X X X X . .
. . . . X . . . . X . .
. . X X X . . . . X . .
. . . . . . . . . X . .
. . X X X . . . . X . .
. . . . X . . . . X . .
. . . ? X . . . . X . .
. . . ? X . . . . X . .
. . . ? X K . . . X . .
. . . ? X X X X X X . .
. . . . . . . . . . . .
```

Als **wortel** van het programma kiezen we de volgende verfijning, waarbij we nu ook expliciet de wereld van Kareltje laten ontstaan en verdwijnen.

```
programma:
  start kareltje;
  draai naar noorden;
  ga naar voortuin;
  zoek de krant;
  ga terug naar bed;
  stop.
```

2.3.1 De keuze

Bij de uitwerking van `draai naar noorden` moeten we op een of andere manier vaststellen of Kareltje al in de goede richting staat (ligt) en hem zo nodig in de gewenste richting draaien. Afhankelijk van de **voorwaarde**, of Kareltje al of niet naar het noorden staat, moeten we tussen twee verschillende gedragingen kiezen. De **keuze** op grond van een voorwaarde noteren we als

```
IF voorwaarde
THEN het een
ELSE het ander
FI
```

wat betekent: als de uitvoering van `voorwaarde` waar oplevert dan wordt `het een` gedaan, en anders `het ander`. Hiermee schrijven we

```
draai naar noorden:
  IF blik niet noordwaarts
  THEN draai rond
  ELSE doe niks
  FI.
```

```
draai rond:
  IF blik zuidwaarts
  THEN keer om
  ELSE IF blik oostwaarts
       THEN linksom
       ELSE rechtsom
       FI
  FI.
```

Merk op hoe hier twee keuzes in elkaar genest zitten.

Hoe geven we Kareltje opdracht om niets te doen? Het eenvoudigst is het tweede alternatief van de keuze weg te laten.

```
draai naar noorden:
  IF blik niet noordwaarts
  THEN draai rond
  FI.
```

De betekenis hiervan is: als Kareltje's blik niet noordwaarts is dan bestaat de uitvoering van `draai naar noorden` uit `draai naar` en anders is de uitvoering van die algoritme meteen voltooid.

2.3.2 De voorwaardelijke herhaling

We kunnen met behulp van de begrensde herhaling een iets kortere definitie geven voor `draai naar noorden`:

```
draai naar noorden:
  UPTO 3
  REP IF blik niet noordwaarts
      THEN
         rechtsom
      FI
  ENDREP.
```

Dit is nou echt een typische robotoplossing. Op de domst mogelijke wijze wordt driemaal vastgesteld of Kareltje goed staat en anders een rechtse draai gemaakt. Stond Kareltje bij het begin al goed dan is al dat werk voor niets. Bovendien moeten we echt even nadenken voor we zien dat precies drie keer genoeg is. De herhaling met een vast aantal slagen (de *begrensde* herhaling) is niet wat wij hier nodig hebben. We willen naar bevind van zaken handelen, dat wil zeggen herhalen afhankelijk van een voorwaarde (de **voorwaardelijke herhaling**). Wij voeren daartoe een nieuwe notatie in.

```
WHILE voorwaarde
REP handeling
ENDREP
```

wat betekent: zolang als de `voorwaarde` waar is wordt telkens de `handeling` uitgevoerd, tot de `voorwaarde` niet meer waar blijkt.

Nu kunnen we schrijven

```
draai naar noorden:
  WHILE blik niet noordwaarts
  REP
    rechtsom
  ENDREP.
```

2.3.3 Kareltje's concrete voorwaarden

Een *voorwaarde* is een algoritme die als resultaat van zijn uitvoering een waarheidswaarde (ja of nee) levert. We geven Kareltje het volgend repertoire aan concrete voorwaarden

- `noordwaarts` alleen ja als Kareltje met zijn blik naar het noorden staat (en anders nee).
- `oostwaarts` alleen ja als Kareltje met zijn blik naar het oosten staat.
- `zuidwaarts` alleen ja als Kareltje met zijn blik naar het zuiden staat.
- `westwaarts` alleen ja als Kareltje met zijn blik naar het westen staat.
- `muur voor` alleen ja als op de volgende kruising een muurdeel staat zodat Kareltje daar niet heen kan.
- `muur rechts` alleen ja als de kruising rechts van Kareltje door een muurdeel geblokkeerd is.
- `piepbal` alleen ja als op de huidige straathoek een piepbal ligt.

We voeren verder een notatie in voor de **ontkenning** van een voorwaarde:

```
NOT piepbal
```

betekent: alleen ja als op de huidige straathoek geen piepbal ligt, en anders nee. Daarmee kunnen we schrijven

```
blik niet noordwaarts:
   NOT noordwaarts.
```

De betekenis van het bovenstaande is vrij subtiel: er wordt een abstracte algoritme `blik niet noordwaarts` ingevoerd, die een waarheidswaarde levert, namelijk de ontkenning van het resultaat van de voorwaarde `noordwaarts`.

Door invoering van de operator `NOT`, die waarheidswaarden in hun tegendeel verandert, besparen wij ons de invoering van een hele hoop nevenvormen van de voorwaarden.

2.3.4 Vervolg van het voorbeeld

Als Kareltje's neus eenmaal de goede richting uitwijst kunnen we hem naar de voortuin krijgen met dezelfde verfijning `ga naar voortuin` als in het vorige voorbeeld. Kareltje staat dan in de hoek naast de deur, met zijn neus naar het oosten. In Kareltje's voortuin ligt de krant op een van de vier volgende plaatsen.

We draaien Kareltje naar het zuiden en laten hem systematisch zoeken.

```
. . . . . . . . . . . .
. . . . X X X X X X . .
. . . . X . . . . X . .
. . X X X . . . . X . .
. . . . . . . . . X . .
. . X X X . . . . X . .
. . . > X . . . . X . .
. . . ? X . . . . X . .
. . . ? X . . . . X . .
. . . ? X . . . . X . .
. . . ? X X X X X X . .
. . . . . . . . . . . .
```

```
zoek de krant:
  rechtsom;
  stap;
  IF er ligt een krant
  THEN pak krant
  FI;
  stap;
  IF er ligt een krant
  THEN pak krant
  FI;
  stap;
  IF er ligt een krant
  THEN pak krant
  FI;
  stap;
  IF er ligt een krant
  THEN pak krant
  FI;
  vier stappen terug.

er ligt een krant:
  piepbal.

pak krant:
  pak piepbal.

vier stappen terug:
  keer om;
  UPTO 4
  REP stap
  ENDREP.
```

Gezien de regelmaat in deze algoritme is er een kortere versie te bereiken door het gebruik van de begrensde herhaling.

```
zoek de krant:
  rechtsom;
  UPTO 4
  REP
    stap;
    IF er ligt een krant
    THEN pak krant
    FI;
  ENDREP;
  vier stappen terug.
```

Maar dat is weer zo'n typische robotoplossing. We kunnen misschien beter een voorwaardelijke herhaling nemen.

```
zoek de krant:
  rechtsom;
  loop tot de krant;
  pak krant;
  keer om;
  loop terug tot de muur.

loop tot de krant:
  WHILE er is geen krant
  REP
    stap
  ENDREP.

er is geen krant:
  NOT piepbal.
```

Wat gebeurt er als er in heel de voortuin geen krant ligt? Bij de laatste oplossing verdwijnt Kareltje als een meteoor naar het zuiden, en omdat hij geen piepbal tegenkomt blijft hij lopen en lopen tot hij van het scherm valt.

De vorige oplossing, met de beperkte herhaling, is iets robuuster: dan komt Kareltje tenminste onverrichter zake in bed terug.

Bij het ontwerp van algoritmen moet je je dit soort vragen steeds expliciet stellen. Een algoritme is bedoeld om onder bepaalde voorwaarden te functioneren. Is daaraan (bijvoorbeeld door een menselijke vergissing) niet voldaan dan kunnen er onbedoelde dingen gebeuren.

2.3.5 Nog een vorm van voorwaardelijke herhaling

De gegeven formulering van `loop tot de krant` heeft nog iets gekunstelds: Eigenlijk willen we niet dat Kareltje loopt zolang hij niet bij de krant is - hij moet lopen tot hij er wel is. Bij de gegeven formulering kijkt Kareltje telkens of hij al bij de krant is, en doet een stap als hij dat niet is. Men noemt zo een `WHILE`-formulering een **pre-checked loop**. De eerste keer weten we zeker dat de voorwaarde `er is geen krant` niet vervuld is, zodat Kareltje voor niets test. Er bestaat in Elan nog een andere vorm van **voorwaardelijke herhaling**, de **post-checked loop**, die in dit geval toepasselijker zou zijn:

```
loop tot de krant:
  REP stap
  UNTIL bij de krant
  ENDREP.

bij de krant:
  piepbal.
```

Merk op dat we een `NOT`-teken zijn kwijtgeraakt. We definiëren analoog

```
loop terug tot de muur:
  REP stap
  UNTIL muur voor
  ENDREP.
```

De herhaling

```
REP handeling
UNTIL voorwaarde
ENDREP
```

betekent het volgende: eerst wordt de `handeling` uitgevoerd en daarna wordt, zolang de `voorwaarde` onwaar is, telkens de `handeling` uitgevoerd, totdat de `voorwaarde` eindelijk waar is. Deze herhaling is daarom equivalent met de paragraaf

```
handeling;
WHILE NOT voorwaarde
REP handeling
ENDREP
```

Het is best nuttig om beide vormen van voorwaardelijke herhaling ter beschikking te hebben, hoewel (strikt gesproken) een van beide, bijvoorbeeld de `WHILE`-vorm, voldoende zou zijn voor alle gevallen.

2.3.6 Conclusie

Als we de ontbrekende verfijningen erbij schrijven, of ze ontlenen uit het vorige voorbeeld, bereiken wij weer een compleet programma. De onderlinge samenhang van de verfijningen kan als volgt worden weergegeven (een **view**, zoals de Elan-1 programmeeromgeving op verzoek geeft):

```
programma
| ga naar voortuin
| | draai zonodig naar noorden
| | | blik niet noordwaarts
| | | draai naar noorden
| | loop langs muur
| | loop door ingang
| | loop naar krant
| zoek de krant
| | loop tot de krant
| | | bij de krant
| | pak krant
| | keer om
| | loop terug tot muur
| ga terug naar bed
| | loop naar ingang terug
| | loop door ingang terug
| | loop langs muur terug
```

Het schema geeft weer welke verfijningen door een bepaalde verfijning worden opgeroepen, en zo voorts.

2.4 Toverwoorden

De notatie die wij hebben ingevoerd maakt gebruik van bepaalde "toverwoorden" (**delimiters**) zoals `IF` en `REP`, die suggereren (delen van) engelse woorden te zijn. Merk op dat deze woorden hier niet in hun natuurlijke betekenis worden gebruikt, en evenmin als namen van algoritmen of objecten. Delimiters zijn een onderdeel van bepaalde afgesproken stijlfiguren en worden daarom ook, in tegenstelling tot de namen die uit kleine letters en eventueel cijfers bestaan, met grote vette letters geschreven.

Sommige mensen zullen verkorte sleutelwoorden als `REP` voor te cryptisch houden, zij zien ze liever voluit. Elan staat daarom voor een aantal symbolen alternatieve representaties toe:

```
REP       REPEAT
ENDREP    ENDREPEAT
FI        ENDIF
```

Wij zullen in het vervolg stelselmatig de voorkeur geven aan de korte versies van de delimiters.

Overigens moet de manier waarop in computertalen het engels pleegt te worden geradbraakt voor iedere engelsman een bezoeking zijn.

2.5 Samenvatting

We hebben nu kennis gemaakt met de **Top-Down programmering**, een programmeerstijl waarbij we naar behoefte abstracte algoritmen invoeren en deze daarna definiëren tot het concrete niveau is bereikt.

Een abstracte algoritme wordt gedefiniëerd door middel van een *verfijning*.

```
naam : paragraaf .
```

De *naam* van de algoritme wordt geschreven met kleine letters, cijfers en (eventueel) spaties, en begint met een letter.

Een *paragraaf* bestaat uit nul of meer units, onderling gescheiden door puntkomma's

```
unit ; unit ; ... ; unit
```

Zo'n *unit* kan elementair zijn (bijvoorbeeld de naam van een algoritme), of samengesteld zijn (bijvoorbeeld met behulp van een controlstructuur) uit andere units. Naast de paragraaf behoren de *keuze* en de *herhaling* tot de controlstructuren.

De keuze kent twee vormen:

```
IF voorwaarde
THEN ene paragraaf
ELSE andere paragraaf
FI
```

en

```
IF voorwaarde
THEN paragraaf
FI
```

De *voorwaarde* is daarbij een unit die een waarheidswaarde levert, op basis waarvan de keuze gemaakt wordt.

Van de herhaling geven wij drie vormen. De *beperkte herhaling*

```
UPTO uitdrukking
REP paragraaf
ENDREP
```

De *uitdrukking* is daarbij een algoritme die een geheel getal oplevert, het aantal malen dat de paragraaf uitgevoerd moet worden.

De *voorwaardelijke herhaling* heeft twee vormen.

```
WHILE voorwaarde
REP paragraaf
ENDREP
```

en

```
REP paragraaf
UNTIL voorwaarde
ENDREP
```

Bij beide wordt het aantal herhalingen van de paragraaf door de `voorwaarde` bepaald.

Al deze constructies behoren tot de taal Elan. Daarnaast hebben we nog een aantal concrete acties en voorwaarden leren kennen die op Kareltje betrekking hebben en alleen in dit hoofdstuk voorkomen. Zij dienen om te oefenen met Kareltje voor het grotere werk begint. Meteen na dit hoofdstuk moet u ze weer vergeten.

Willen we verder komen in de richting van een echte programmeertaal dan moeten we nu de concrete algoritmen van programmeertaal leren kennen, en vooral ook met objecten leren omgaan. Dat doen we in de volgende hoofdstukken.

2.6 Opgaven

Bij al deze opgaven wordt gebruik gemaakt van de Kareltje-pakketten (`karel` en `omgeving`) die met de Elan-programmeeromgeving verspreid worden. Bij het begin van iedere opgave brengen we door middel van `start kareltje` Kareltje in de oorsprong, met zijn neus naar het noorden. Het beeldscherm is verder leeg.

1. Van Kareltje's concrete algoritmen `linksom` en `rechtsom` is er een overbodig. Definieer deze in termen van de andere.
2. Laat Kareltje een vierkantje omlopen, met zijden 10.
3. Beschouw het beeldscherm als een akker met voren in horizontale richting. Laat Kareltje een akker van 16 voren van elk 20 stappen inzaaien met aardappelen. Tot slot moet Kareltje naar de oorsprong terugkeren.
4. Laat Kareltje midden op het scherm een Volendammer met pofbroek en vlaggetje tekenen (zie plaatje 2.1).
   Probeer daarbij het tekenen van de Volendammer (en zijn vlaggetje) zo verantwoord mogelijk uit te drukken in het tekenen der delen.
5. Idem, teken een huis (naar eigen smaak).
6. (trap) De onderste regel van het beeldscherm wordt opgevat als een vloer, waarop een *trap* staat. Het noorden wordt dus als "boven" geduid. Ontwikkel een algoritme om (onafhankelijk van de plaats, vorm en hoogte daarvan) over een vloer met eventuele trappen te lopen tot een piepbal wordt gevonden. Een voorbeeld-trap wordt gemaakt door `maak trap` (uit de Karel-omgeving).
7. (doolhof) Kareltje bevindt zich in een doolhof, van waaruit hij de buitenlucht kan bereiken door te gaan lopen met zijn rechterwijsvinger slepend langs de wand. Buiten de dool-

```
. . . . . . . . . . . . o o o .
. . . . . o o o o o . . o . o .
. . . . . o . . . o . . o o o .
. . . . . o . . . o . . o . . .
. . . . . o o . o o . o o o . .
. . . . . . o . o . . o . o . .
. o o o o o o . o o o o . o . .
. o . . . . . . . . . . . o . .
. o . o o o . . . o o o o o . .
. o . o . o . . . o . . . . . .
. o o o . o . . . o . . . . . .
. . . . . o . . . o . . . . . .
. . . o o o . . . o o o . . . .
. . . o . . . . . . . o . . . .
. . . o . . o o o . . o . . . .
. . . o . . o . o . . o . . . .
. . . o . . o . o . . o . . . .
. . . o o o o . o o o o . . . .
```

Figure 2.1: Volendammer met pofbroek en vlaggetje

hof, rechts om de hoek, ligt een piepbal zodat hij kan vaststellen of hij buiten is. Ontwikkel een algoritme om uit de doolhof te raken. Een voorbeeld-doolhof wordt gemaakt door `maak doolhof`.

Chapter 3

Gehele getallen

In dit hoofdstuk beginnen we de systematische behandeling van Elan. Achtereenvolgens zullen we de vier concrete elementaire typen behandelen:

`INT`	de gehele getallen
`REAL`	de reële getallen
`BOOL`	de waarheidswaarden
`TEXT`	de teksten.

Deze **standaard typen** van Elan zijn ongeveer dezelfde als in andere programmeertalen en hebben ook op de meeste computers een directe representatie.

De concrete typen van Elan worden gedefinieerd in het **standaard pakket**, een (fictief) stuk programma dat declaraties bevat voor alle concrete objecten, typen en algoritmen. In het tweede deel van deze reeks zullen we ook leren hoe we zelf pakketten kunnen schrijven, een faciliteit voor Bottom-Up-programmering die moderne talen als Elan en ADA bieden.

Bij de presentatie van telkens nieuwe typen in dit hoofdstuk en in de volgende hoofdstukken behandelen we achtereenvolgens

- (naam) hoe heet het type?
- (domein) welke waarden horen tot dit type?
- (denotatie) hoe noteer je een waarde van dit type (in het programma)?
- (operaties) welke operaties zijn specifiek voor dit type? Wat zijn hun eigenschappen?

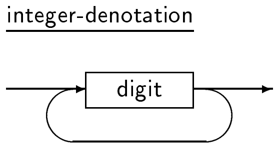
Daarna demonstreren we het gebruik van het type aan een aantal voorbeelden.

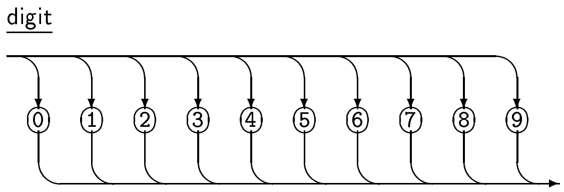
Als eerste behandelen we het type dat de gehele getallen implementeert. De naam van dit type is `INT` en de gebruikelijke naam voor die gehele getallen die in de computer gerepresenteerd kunnen worden is **integers**.

3.1 Denotatie en representatie van integers

In onze algoritme willen we aangeven dat de computer met bepaalde getallen moet rekenen. De manier waarop je in een algoritme een bepaald getal aanduidt noem je de **denotatie** daarvan.

De gebruikelijke denotatie voor integers is de decimale notatie voor (positieve) gehele getallen, d.w.z. een reeks van een of meer decimale cijfers. Wij zullen dit feit weergeven in de vorm van de volgende *syntax-diagrammen*. De hier gegeven diagrammen definiëren





de begrippen integer denotation en digit. Door in het eerste **syntax-diagram** links te beginnen en het volgens de stroomrichting te doorlopen zien we dat een integer denotation bestaat uit een of meer digits, en dat een digit een van de cijfers 0 tot 9 is.

We zullen in de komende hoofdstukken ook veel ingewikkelder begrippen door middel van syntax-diagrammen verduidelijken. Achter in dit boek (Appendix A) bevindt zich een complete beschrijving van de syntax van Elan in een ander formalisme, dat in hoofdstuk 11 wordt uitgelegd.

De waarde die door een integer denotatie wordt voorgesteld volgt uit de gebruikelijke interpretatie als decimaal getal. Merk op dat alleen positieve gehele getallen met behulp van een integer-denotation kunnen worden voorgesteld, het voorteken - of + maakt geen deel uit van de denotatie maar is strikt genomen een operator. Ook kunnen verschillende denotaties eenzelfde waarde aangeven, bijvoorbeeld 0023 en 23.

Uit bovenstaande diagrammen zou men kunnen opmaken, dat een integer-denotation willekeurig lang kan zijn. In werkelijkheid bestaat er steeds een kleinste en een grootste getal dat in de computer nog voorgesteld kan worden. Die hele getallen die door een (bepaalde) computer voorgesteld kunnen worden, heten de *integers* (van die computer). Het gebied tussen kleinste en grootste integer van een computer noemt men het **integer bereik** van die computer. In de volgende tabel zijn voor een aantal computers het kleinste en het grootste gehele getal gegeven, dat door die computer voorgesteld kan worden.

	kleinste	grootste integer
calculator.	-10^8+1	$10^8-1=99999999$
32-bits mach.	-2^{31}	$2^{31}-1=2147483647$
16-bits mach.	-2^{15}	$2^{15}-1=32767$

Uit deze tabel blijkt dat zowel positieve als negatieve gehele getallen kunnen worden voorgesteld. In de meeste computers is er voor de interne representatie van een integer een zeker aantal binaire cijfers (**bits**, van "binary digits") ter beschikking. Let op de asymmetrie van vele integer-bereiken.

Het feit dat de integers intern meestal tweetallig worden voorgesteld (en niet in het tientallig stelsel) heeft weliswaar geen enkele consequentie buiten het feit dat het integer-bereik nogal rare grenzen heeft, maar oefent toch een zeker vervreemdingseffect uit. In ieder geval is het niet nodig iets van het tweetallig stelsel te weten om integers te gebruiken.

Hoewel de integers in de computer bedoeld zijn om gehele getallen voor te stellen blijkt er een aantal addertjes onder het gras verborgen te zitten, die het noodzakelijk maken om bij het programmeren omzichtig te werk te gaan.

3.1.1 Overflow

Het kan voorkomen dat bij een operatie op integers, bijvoorbeeld bij het optellen, het resultaat daarvan niet door de computer voorstelbaar is. In dat geval spreekt men van **overflow**. Sommige computers zijn zo vriendelijk om melding te maken van zo'n gebeurtenis, andere daarentegen leveren zonder morren als resultaat van zo'n optelling een integer, waarvan het verband met de twee opgetelde getallen alleen met kennis van de desbetreffende computer te doorgronden is. Het is daarom geboden, overflow te voorkomen.

Ook al is het resultaat van een berekening representeerbaar, het kan zijn dat een tussenresultaat overflow geeft en het resultaat verpest. Daarom kun je er niet zonder meer van uitgaan dat $(a + b) + c = a + (b + c)$; dit geldt alleen als geen overflow optreedt.

Het is in dit verband interessant om voor jezelf na te gaan welk deel van alle mogelijke optellingen van integers en welk deel van alle mogelijke vermenigvuldigingen van integers overflow geeft.

3.1.2 Machine-afhankelijkheid

Een tweede bron van problemen is het gegeven, dat op verschillende computers verschillende bereiken voor de integers gelden. Deze verschillen kunnen heel groot zijn. Vergelijken we bijvoorbeeld een "numbercruncher" met 48-bits integers met een 16-bits microcomputer dan blijkt dat het grootste voorstelbare gehele getal op de eerste computer ongeveer 10^{10} maal groter is dan op de tweede. Om toch in staat te zijn programma's te schrijven die op verschillende computers uitgevoerd kunnen worden, is een concrete constante `maxint` beschikbaar die als waarde het grootste op de onderhavige computer voorstelbare gehele getal heeft. Het kleinst voorstelbare gehele getal is meestal gelijk aan `- maxint`, bij sommige machines nog één kleiner. Het integer-bereik omvat dus gegarandeerd `- maxint : maxint`. Men kan in zijn algoritme rekening houden met het integer-bereik of de algoritme laten afbreken wanneer het bereik te klein blijkt. Hierdoor is het mogelijk de algoritme zo te formuleren dat deze onafhankelijk is van het integer-bereik van de gebruikte machine.

3.1.3 Terminologie

De volgende begrippen dienen goed onderscheiden te worden:

geheel getal	een bepaald wiskundig begrip
integer	een in de computer voorstelbaar geheel getal
`INT`	de naam van het type van de integers
integer-denotation	de schrijfwijze voor een (positieve) integer.

3.2 Enige belangrijke concepten

We moeten nu, aan de hand van een voorbeeld, een aantal belangrijke begrippen en notaties invoeren die niet alleen op de integers slaan maar de algoritmische omgang met allerlei typen objecten mogelijk maken.

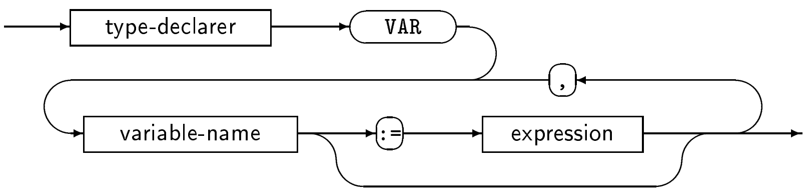
Eerst het voorbeeld. Het verhaal gaat dat Blaise Pascal reeds als jongeling op school een slim wiskundige was. Toen de meester zijn klas eens even rustig wilde houden gaf hij zijn leerlingen de taak de som van de getallen 1 tot en met 100 te berekenen. Pascal ontdekte de somformule voor de rekenkundige reeks en was gauw klaar. Een modern scholier zou uiteraard in plaats van na te denken naar zijn zakcomputer grijpen. Laten wij dat dus ook eens doen. We laten de computer voor ons rekenen, volgens het schema

```
probleem van pascal:
  begin met het getal 1;
  begin met de som nul;
  WHILE getal <= 100
  REP tel getal bij som;
      verhoog getal met een
  ENDREP;
  laat resultaat zien.
```

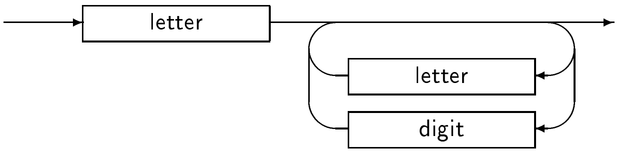
In dit schema treden `getal` en `som` op als een *variabele*: een naam met een vaste betekenis en een veranderlijke waarde. Zo'n naam voer je in met behulp van een *declaratie*; aan een variabele geef je een andere waarde via een *toewijzing*. Het resultaat laat je zien door middel van een *uitvoeralgoritme*. Voor deze fundamentele handelingen kent Elan ieder een bepaalde notatie, bepaalde concrete algoritmen die behoren tot de taal.

3.2.1 Declaratie van een variabele

Een **variabele** is een object, dus de combinatie van een *naam* en een *waarde*, waaraan door een toewijzing een andere waarde kan worden gegeven. Een variabele (of een ander object) mag je in het programma pas gebruiken na hem te hebben gedefiniëerd door middel van een **declaratie**. Zo introduceren we de integer variabele `aantal`, met aanvankelijke waarde 1, door te schrijven variable-declaration



name, variable-name, constant-name




```
INT VAR aantal := 1
```

Het type van `aantal` wordt hier vastgelegd als `INT`, wat wil zeggen dat `aantal` enkel integers als waarden kan aannemen. Het teken `:=` gevolgd door de uitdrukking `1` zorgt hier voor de **initialisatie** van `aantal` met de waarde één. Zo'n initialisatie mag ook weggelaten worden; de variabele krijgt dan aanvankelijk een ongedefinieerde waarde, zoals bijv. in:

```
INT VAR j
```

De declaratie van een aantal variabelen van eenzelfde type mag gecombineerd worden, bijvoorbeeld

```
INT VAR aantal := 1, j
```

door samentrekking van de vorige twee declaraties, of de samentrekking van

```
INT VAR onder; INT VAR boven
```

tot

```
INT VAR onder, boven
```

Het bijbehorende syntax-diagram is fig. 3.2.1.

De naam van een object (in computerjargon "identifier") wordt geschreven met (kleine) letters en cijfers, met een letter aan het begin.

Ter verhoging van de leesbaarheid mogen er in een naam spaties voorkomen, maar zij maken geen deel uit van de naam zelf. Het nogal saaie syntax-diagram voor letter (een opsomming van 26 letter-tekens) blijft u bespaard.

3.2.2 Toewijzing

Door middel van een **toewijzing** kan aan een variabele een andere waarde gegeven worden. Let op: niet de waarde zelf wordt dan gewijzigd maar de variabele krijgt een andere waarde.

Na bovenstaande declaratie bezit `aantal` bijvoorbeeld de waarde één.

Naam	`aantal`
Waarde	1

Direct na de toewijzing `aantal:= 2` bezit `aantal` de waarde twee.

Naam	`aantal`
Waarde	~~1~~ 2

Zo'n toewijzing wordt geschreven als het "wordt-teken" `:=` met links daarvan de naam van een variabele en rechts een uitdrukking.

Voor het begrip *uitdrukking* (expression) geven we later (in 7.2) een syntax-diagram. Het **effect** van de uitvoering van een toewijzing als

```
v:= expr
```

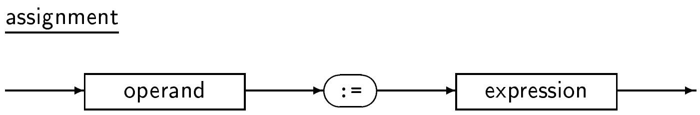
bestaat uit een wijziging van de waarde van `v`, als volgt:

1. het resultaat van de uitdrukking `expr` wordt berekend
2. een copie van de resulterende waarde wordt tot waarde van `v` gemaakt, waarbij een eventuele waarde die `v` te voren bezat verloren gaat.

Het type van `v` moet hetzelfde zijn als dat van het resultaat. Een toewijzing is een actie en levert zelf geen resultaat. Elan kent dus geen kettingtoewijzingen als `x:= y:= 1`, wat moet worden geformuleerd als `y:= 1; x:= y` of zo.

3.2.2.1 Voorbeelden van toewijzingen

Het effect van de toewijzing kan men het beste leren kennen aan de hand van voorbeelden, zoals de volgende. We gaan er daarbij van uit dat `a` en `b` integer variabelen zijn, dus gedeclareerd als



a	b
6	7
13	-1
12	14

a	b
x	y
$x+y$	$x+y-y-y=x-y$
$x+y+x-y=2x$	$2x-(x-y)-(x-y)=2y$

Figure 3.1: Verdubbeling dmv toewijzingen

```
INT VAR a, b
```

Beschouw nu achtereenvolgens de toewijzingen:

- `a:= 13`
  (Nu heeft `a` de waarde dertien)
- `a:= 13; b:= 7`
  (De toewijzing aan `b` verandert de waarde van `a` niet)
- `a:= 13; b:= a`
  (Nu hebben `a` en `b` dezelfde waarde)
- `a:= 13; b:= a-6`
  (Nu is `b` 6 kleiner dan `a`)
- `a:= 13; a:= 14`
  (Het effect van de eerste toewijzing wordt door de tweede ongedaan gemaakt)
- `a:= 13; a:= a`
  (De huidige waarde van `a` wordt nog eens aan `a` toegewezen)
- `a:= 13; a:= a + 1`
  (Eerst wordt de huidige waarde van `a` genomen en 1 daarbij opgeteld, dan wordt het resultaat aan `a` toegewezen)
- ```
  a:= 6; b:= 7;
  a:= a + b; b:= a - b - b;
  a:= a + b; b:= a - b - b
  ```
  (Ga na dat dit vreemde stukje programma de waarden van `a` en `b` verdubbelt mits er geen overflow optreedt).

3.2.3 Uitvoer

Om de waarde van een variabele of (meer in het algemeen) een uitdrukking zichtbaar te maken kun je schrijven

```
put (uitdrukking)
```

Hierbij is `put` de naam van een uitvoeralgoritme, en de uitdrukking tussen haakjes de parameter waarmee hij wordt uitgevoerd. De waarde van de `uitdrukking` wordt berekend (dat levert bijvoorbeeld een integer op). Deze waarde wordt vervolgens in de vorm van een denotatie, zo nodig voorafgegaan door een negatief voorteken, geschreven op het uitvoerorgaan, in dit geval het beeldscherm van de microcomputer of terminal waarmee wij werken. Op dat beeldscherm staat een lopertje ("cursor") dat steeds aangeeft wáár de eerstvolgende uitvoer verschijnen zal.

Eerlijk gezegd is het precieze effect van de uitvoeralgoritme moeilijker te beschrijven dan uit te proberen. Waag je dus maar eens voor de computer en probeer een minimaal programma'tje

```
programma:
  put (1); put (2+3); put (4-10).
```

Als dit programma'tje wordt uitgevoerd zie je dat de cursor bij het schrijven meeloopt. Wordt er te veel op een regel geschreven dan vliegt de cursor rechts uit de regel en gaat hij links op de eerstvolgende regel verder. Zo'n overgang op een nieuwe regel kun je ook zelf bereiken door een oproep van de algoritme `line`. Daarmee kun je een fraaier indeling van het beeld bereiken dan door maar lukraak op het scherm te schrijven. Meer dan één nieuwe regel kan worden gegeven door de oproep van een integer parameter te voorzien, bijvoorbeeld (om vijf regels verder te gaan)

```
line (5)
```

Een oproep van `line` met parameter `1` bewerkstelligt hetzelfde als een oproep zonder parameter.

De oproepen

```
line; put(x)
```

kunnen worden gecombineerd door `put line` op te roepen; de waarde van `x` wordt dan aan het begin van de volgende schermregel geschreven.

3.2.4 Invoer

Het is ook mogelijk tijdens uitvoering van een programma getallen in te voeren met behulp van de invoeralgoritme `get`, met als parameter een variabele. Als `aantal` een integer variabele is, dan heeft de oproep

```
get (aantal)
```

het volgende effect:

- De uitvoering van het programma wordt onderbroken terwijl de computer wacht tot dat op het toetsenbord een regel is ingetypt en afgesloten door op de RETURN- of ENTER-toets te drukken.
```

- Op deze regel moet een integer-denotatie staan, eventueel voorafgegaan door een teken. Daarvan wordt de waarde berekend.
- De waarde wordt aan de variabele `aantal` toegewezen.

Eerlijk gezegd is de zaak nog wat ingewikkelder, aangezien er op zo'n invoerregel meer dan één denotatie kan voorkomen. De eerste oproep van `get` wacht wel tot de regel is ingetikt, maar de volgende oproep kan dan zonder wachten verder met de volgende denotatie, tot de regel geheel is verwerkt waarna een volgende oproep van `get` eerst weer op invoer moet wachten.

Op deze wijze hebben we dus een waarde voor de variabele `aantal` binnengehaald en kunnen daarmee aan de slag.

3.2.5 Vervolg van het voorbeeld

Gewapend met al deze kennis kunnen we nu een compleet programma schrijven. Daarbij bepalen we de som van de getallen *1* t/m *grens*, waarbij `grens` een variabele is waarvan we de waarde inlezen.

```
probleem van pascal:
  vraag de grenswaarde;
  begin met het getal 1;
  begin met de som 0;
  WHILE getal <= grens
  REP tel getal bij som;
      neem het volgende getal
  ENDREP;
  laat het resultaat zien.
```

Nu geven we een verfijning voor ieder van de deelalgoritmen.

```
vraag de grenswaarde:
  INT VAR grens; get (grens).

begin met het getal 1:
  INT VAR getal := 1.

begin met de som 0:
  INT VAR som := 0.

tel getal bij som:
  som:= som + getal.

neem het volgende getal:
  getal:= getal + 1.

laat het resultaat zien:
  line;
  put (som).
```

We hebben ons heel dicht gehouden bij de formulering van de algoritme zoals gegeven aan het begin van dit hoofdstuk, zodat we een vrij groot aantal kleine verfijningen hebben gekregen. In de huidige fase van het leren programmeren kan dat absoluut geen kwaad. Het gevolg is een algoritme die uitstekend te volgen is maar vrij veel tikwerk vraagt. Dat betekent niet dat ieder programma altijd zo gedetailleerd verfijnd moet worden. Verfijningen zijn een hulpmiddel om richting te geven aan het creatieve denken bij het programmeren, en om de tussenstadia daarvan vast te leggen. Naarmate de programmeerervaring groeit kan men zich ook grotere gedachtensprongen veroorloven, maar voorlopig proberen we zo duidelijk mogelijk te verfijnen, waarbij uit de oplossing steeds expliciet moet blijken wat de programmeur gedacht heeft.

3.2.6 Declaratie van een constante

Naast variabelen bestaan er ook constanten. Een **constante** is een object waarvan de waarde niet door toewijzing kan worden gewijzigd.

De behoefte, een bepaalde waarde een naam te geven en hem met die naam aan te duiden in plaats van met een denotatie kan allerlei redenen hebben: bijvoorbeeld kan de waarde op zich weinig interessant zijn, maar ligt zijn belang in de rol die hij speelt. Zo zou, op een bepaalde machine, de ons reeds bekende constante `maxint` een declaratie kunnen hebben als

```
INT CONST maxint = 2147483647
```

Dit is de waarde die `maxint` heeft in de Elan-1 interpreter. Op andere machines heeft hij weer een andere waarde, maar we noemen hem steeds `maxint`.

Een declaratie van een integer constante die bijvoorbeeld in een schaak-programma zijn nut kan hebben is:

```
INT CONST aantal velden = 8 * 8
```

De declaraties van een aantal integer constanten kunnen samengevat worden tot een samengestelde declaratie. Voorbeeld:

```
INT CONST aas = 14, heer = 13,
          vrouw = 12, boer = 11
```

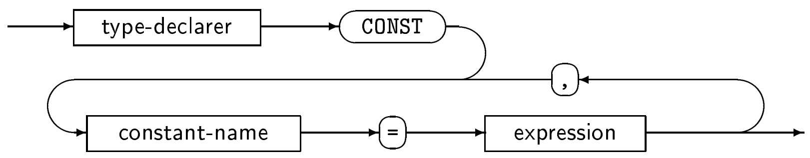
Constanten hebben, als ze eenmaal gedeclareerd zijn, een onveranderlijke waarde. Er kan dus, in tegenstelling tot variabelen, niet aan worden toegewezen. Ook niet per ongeluk. Ook dit kan een goede reden zijn om waar dat mogelijk is een constante in plaats van een variabele te gebruiken. Als vuistregel: willen we aan een object niet toewijzen dan kunnen we er beter een constante van maken.

Het verschil tussen variabelen en constanten is uitsluitend een verschil in **acces**, de mate aan toegang die je ertoe hebt: van een variabele en een constante mag je allebei naar de waarde vragen (**lees acces**) maar alleen aan een variabele mag je een nieuwe waarde toewijzen (**schrijf acces**).

3.3 Integer operaties

In de programmeertaal Elan is een aantal elementaire algoritmen en typen concreet voorgegeven. Deze zijn gedeclareerd in het **standaard pakket**, een doos vol nuttige voorwerpen die u gratis ter beschikking krijgt (zie Appendix B). Ze kunnen zonder nadere declaratie in ieder programma gebruikt worden. Daaronder is ook een aantal concrete algoritmen voor de manipulatie van integers, die we hier behandelen. Vanwege hun samenhang met het type `INT` zullen we ze (de concrete) **integer operaties** noemen.

constant-declaration



Sommige operaties hebben volgens de traditie een naam die niet met kleine letters wordt geschreven (zoals `put`), maar met hoofdletters (zoals `ABS`) of met een of meer bijzondere tekens (zoals `+`). Men noemt zo'n naam een **operator**. De oproep van een operator ziet er ook wat anders uit.

3.3.1 Arithmetische operaties

Als eerste zijn, bijna vanzelfsprekend, de bekende arithmetische operaties gegeven:

operator	betekenis	voorbeeld
`+`	optelling	`1 + 1` (= 2)
`-`	aftrekking	`5 - 9` (= −4)
`*`	vermenigvuldigen	`3 * 4` (= 12)
`DIV`	quotiënt bij delen	`13 DIV 6` (= 2) !!
`MOD`	rest bij delen	`13 MOD 6` (= 1)
`**`	machtsverheffen	`5 ** 2` (= 25)

Naast deze *dyadische* integer operatoren met twee integers als operanden en een integer resultaat, bestaan de `+` en de `-` ook als *monadische* integer operatoren, d.w.z. werkend op één integer operand en met een integer resultaat, met de gebruikelijke betekenis.

operator	betekenis	voorbeeld
`+`	mon. plus	`+8` (= 8)
`-`	mon. minus	`-4` (= −4)

Het resultaat van de modulo-operator `MOD` is voor positieve teller en noemer steeds positief en kleiner dan de noemer. Voor `DIV` en `MOD` gelden de volgende eigenschappen:

```
  x DIV -y   =   - (x DIV y)
  x MOD -y   =   x MOD y
(-x) DIV y   =   - (x DIV y)
(-x) MOD y   =   - (x MOD y)
```

Uiteraard mag daarbij de noemer nooit nul zijn, anders treedt overflow op.

3.3.2 Uitdrukkingen

Met behulp van operatoren, operanden en haakjes kunnen uitdrukkingen ("formules", "expressies") gevormd worden op de gebruikelijke manier, waarbij de gebruikelijke voorrangsregels gelden, zoals die nogal gebrekkig plegen te worden aangeduid met het zinnetje:

```
"Mijnheer Van Dalen At Oliebollen".
```

Machtsverheffen heeft een hogere **prioriteit** dan vermenigvuldigen en delen, die op hun beurt weer een hogere prioriteit hebben dan optellen en aftrekken. Bij gelijke prioriteit worden de operaties van links naar rechts uitgevoerd. De monadische operatoren hebben steeds de hoogste prioriteit.

Bijvoorbeeld:

```
 3 * - 1 (= -3)
-1 **  2 (= +1)
```
let op !!

De laatste formule levert vanwege de hoge prioriteit van de monadische operator `-` de waarde `+1` en niet `-1`, zoals men misschien zou verwachten. Met behulp van haakjes kan, indien gewenst, een andere volgorde van berekening worden aangegeven:

```
- (1 ** 2) (= -1)
```

3.3.3 Vergelijkingsoperatoren

Er bestaat een aantal dyadische operatoren met twee integer argumenten en een waarheidswaarde als resultaat (integer **vergelijkings operatoren**)

operator	betekenis
$<$	kleiner dan
$<=$	hoogstens
$>$	groter dan
$>=$	minstens
$=$	gelijk aan
$<>$	ongelijk aan

Deze kunnen worden gebruikt in een voorwaarde als `x <= 0`.

De prioriteit van deze vergelijkingsoperatoren is kleiner dan die van de arithmetische operatoren, zodat `a + 1 <= x` hetzelfde betekent als `(a + 1) <= x`.

3.3.4 Toewijzende operaties

Een gecombineerde optelling en toewijzing aan een integer variabele als `x:= x+1` komt zo vaak voor dat er een korte schrijfwijze voor is ingevoerd. De toewijzing

```
subtotaal:= subtotaal + term
```

kan ook als

```
subtotaal INCR term
```

geschreven worden.

Dit is niet alleen korter maar bovendien wordt daardoor de intentie, de variabele te incrementeren, explicieter gemaakt. (Het is niet zomaar een toewijzing.) De operator `INCR` verlangt als linker operand een `INT VAR` en als rechter operand een `INT` uitdrukking.

Analoog bestaat er ook een operator `DECR` voor de aftrekking gecombineerd met de toewijzing, bijvoorbeeld:

```
inkomen DECR uitgaven
```

wat hetzelfde betekent als

```
inkomen:= inkomen - uitgaven
```

Vooral als de variabele een lange naam heeft komt deze verkorte schrijfwijze van pas.

3.4 Voorbeeld: maximum van een reeks

Als voorbeeld van het gebruik van integers stellen we ons als opgave om de grootste van een niet lege reeks positieve gehele getallen te vinden.

We gaan er van uit dat van het standaard invoerapparaat (het toetsenbord) een reeks positieve getallen ingegeven wordt. Het aantal getallen dat zal komen is niet tevoren bekend, maar het is er zeker minstens een (de reeks is niet leeg). Ter afsluiting wordt de reeks gevolgd door een negatief getal. Bepaal het maximum, d.w.z. vind de waarde van het grootste getal.

De algoritme die ons probleem oplost berust er op dat na enige voorbereiding de getallen een voor een gelezen worden, terwijl daarbij steeds van de reeds gelezen getallen de tot nog toe grootste als voorlopig maximum wordt onthouden. Op deze manier vinden we het maximum van alle tot dan toe gelezen getallen. Wanneer het laatste getal gelezen is, moet vanzelfsprekend het dan gevonden voorlopig maximum tevens het definitieve maximum zijn.

```
maximum bepalen:
  het maximum is voorlopig nul;
  lees het eerste getal;
  REP
    kijk of dat een nieuw maximum is;
    lees het volgende getal
  UNTIL alle positieve getallen bekeken
  ENDREP;
  schrijf het definitieve maximum.
```

We gebruiken een variabele om het voorlopig maximum in op te bouwen. Voor de herhaling moet deze variabele geïnitialiseerd worden. Twee strategieën bieden zich aan:

- Initialisatie op nul, bij wijze van maximum van nul positieve getallen
- Initialisatie op het eerste element van de rij, omdat het maximum minstens gelijk is aan het eerste element.

We hebben hier de eerste strategie genomen.

```
het maximum is voorlopig nul:
  INT VAR huidig maximum := 0.
```

Merk op hoe vaak een verfijning als een geïnitialiseerde declaratie kan worden gerealiseerd. Zo'n declaratie schept een welgedefinieerde toestand, waaraan gemakkelijk een naam kan worden gegeven.

```
lees het eerste getal:
  INT VAR getal;
  get(getal).

kijk of dat een nieuw maximum is:
  IF
    getal > huidig maximum
  THEN
    huidig maximum:= getal
  FI.

lees het volgende getal:
  get(getal).

alle positieve getallen bekeken:
  getal < 0.

schrijf het definitieve maximum:
  line;
  put (huidig maximum);
  line.
```

De werking van deze algoritme berust op het feit dat de bewering

> "de waarde van de variabele `voorlopig maximum` is het maximum van alle tot dan toe gelezen getallen"

na iedere slag van de herhaling waar is. Vóór de uitvoering van de herhaling is deze uitspraak waar dankzij onze aanname dat het maximum van een lege reeks positieve getallen 0 is. Wanneer de uitvoering van de herhaling ten einde is zijn alle getallen gelezen en is de uitspraak nog steeds waar. Dus is het voorlopig maximum gelijk aan het definitieve maximum. Deze eigenschap is ook terug te vinden in de naam van de variabele `huidig maximum`, die ook `voorlopig maximum`, of `maximum tot nog toe` had kunnen heten.

De terminatie van de algoritme is verzekerd door het feit dat er in iedere slag van de herhaling een getal gelezen wordt, terwijl er een eindig aantal getallen zal worden ingevoerd, voordat er een negatief getal komt.

Opgave: Hoe zou de oplossing volgens de tweede strategie (neem het eerste getal als initialisatie van `huidig maximum`) er uit zien?

3.5 Voorbeeld: Fibonacci-getallen

Het verhaal gaat, dat in het jaar 1202 de italiaan Leonardo Pisano, ook wel genaamd Leonardo Fibonacci (Filius Bonaccio) het belangrijke economische probleem:

> *"Hoeveel paren konijnen kunnen in een jaar tijd geproduceerd worden, wanneer men uitgaat van één paar konijnen?"*

oploste. Bij deze opgave dient dan te worden aangenomen dat elk paar volwassen konijnen elke maand een paar jongen ter wereld brengt, een van het mannelijk en een van het vrouwelijk geslacht, dat een paar pasgeboren konijnen na een maand tot dezelfde prestatie in staat is als hun ouders en dat konijnen nooit overlijden. Kennelijk gaat het in de wereld der konijnen anders toe dan in de wereld der mensen, zeker in die dagen. (Voor meer gegevens betreffende het leven der konijnen in onze tijd verwijzen we naar [ADM72].)

De oplossing voor het probleem wordt als volgt gevonden. Stel het aantal paren konijnen in de maand n op $F(n)$. (De letter F wordt hier ter nagedachtenis aan Fibonacci gebruikt.) Stel, van deze $F(n)$ konijnen zijn er $V(n)$ volwassen. Wat is nu het aantal paren in maand $n+1$ ofwel $F(n+1)$? In de maand $n+1$ zullen er $F(n)$ volwassen zijn, aangezien het aantal volwassen paren in een bepaalde maand gelijk is aan het totaal aantal paren in de voorgaande maand. Daarnaast hebben de $V(n)$ volwassen paren elk een paar jongen ter wereld gebracht, zodat $F(n+1) = F(n) + V(n) = F(n) + F(n-1)$. Wanneer we aannemen dat onze konijnen uit maand 1 een maand daarvoor nog ongeboren waren omdat alle andere konijnen de maand daarvoor zijn opgegeten, dan laat het probleem zich ook formuleren als: Vind de waarde van $F(12)$ als gegeven is dat:

$$F(0) = 0,$$

$$F(1) = 1 \text{ en}$$

$$F(n+1) = F(n) + F(n-1) \text{ voor } n \geq 1.$$

Na enig rekenwerk vinden we dat de tijdgenoten van Fibonacci, als ze dit proces in januari met één paar onvolwassen konijnen startten, zich met de kerstdagen aan een gros paren konijnen te goed konden doen.

Het probleem dat we ons voor dit voorbeeld stellen is, om met behulp van de zojuist gevonden betrekking voor de Fibonacci-getallen de limiet van de verhouding tussen twee opvolgende getallen te benaderen. Ten einde echter enig inzicht te krijgen in het gedrag van Fibonacci-getallen zullen we eerst afleiden wat de waarde van de limiet is.

Wanneer we de limiet van $F(n)/F(n-1)$ voor n naar oneindig F noemen, dan volgt voor voldoende grote n dat

$$F(n) = F(n)/F + F(n)/(F * F)$$

of

$$F * F - F - 1 = 0$$

waaruit volgt dat voor n naar oneindig

$$F = \frac{F(n)}{F(n-1)} = \frac{1 \pm \sqrt{5}}{2}.$$

We nemen de positieve oplossing (= 1.61803...), aangezien door onze keuze van $F(0)$ en $F(1)$ alle $F(n)$ positief zijn. Ieder Fibonacci-getal is dus ongeveer 1.6 maal groter dan het voorafgaande. Wanneer we voor het gemak de Fibonacci-getallen even als een meetkundige reeks beschouwen met als rede deze limiet, dan volgt hieruit dat het grootste op een machine met `maxint` = 2147483647 nog voorstelbare Fibonacci-getal $F(43)$ zou zijn. Na enig proberen blijkt deze afschatting aan de pessimistische kant te zijn en kan zelfs $F(46)$ nog met deze waarde van `maxint` worden voorgesteld. Om geen last van overflow te krijgen, zullen we met de berekening van de Fibonacci-getallen daarom niet verder gaan dan $F(46)$.

Een algoritme die de eerste 46 Fibonacci-getallen berekent en de verhouding daartussen afdrukt, kan als volgt geformuleerd worden:

```
fibonacci getallen:
  initialiseer;
  WHILE nog niet laatste getal
  REP
    bereken volgend getal;
    druk rangnummer getal en verhouding af
  ENDREP
```

Uit de betrekking voor de Fibonacci-getallen volgt dat voor de berekening van $F(n+1)$ telkens alleen de twee voorgaande getallen $F(n)$ en $F(n-1)$ nodig zijn. In de algoritme zullen we deze variabelen `laatste getal` en `voorlaatste getal` noemen. Tevens hebben we een variabele nodig die het rangnummer van de berekende getallen aangeeft. De initiële waarden van `laatste getal`, `voorlaatste getal` en `rangnummer` volgen uit $F(0) = 0$ en $F(1) = 1$. We krijgen:

```
initialiseer:
  INT VAR voorlaatste := 0,
          laatste := 1,
          rangnummer := 1.
```

We kunnen doorgaan zolang het rangnummer kleiner is dan 46.

```
nog niet laatste getal:
     rangnummer < 46.
```

De deelalgoritme `bereken volgend getal` dient een formulering te zijn van de betrekking $F(n+1) = F(n) + F(n-1)$.

Bij elke stap in de herhaling moet echter het `laatste` het nieuw berekende getal van de vorige stap zijn en het `voorlaatste` moet het `laatste` van de vorige stap zijn. Voor deze verschuiving van waarden hebben we een tijdelijke naam nodig om de waarde van het nieuw berekende getal mee aan te duiden. De waarde van `nieuw getal` is gedurende de verschuiving constant.

```
bereken volgend getal:
    INT CONST nieuw getal = laatste +
voorlaatste;
   voorlaatste:= laatste;
   laatste:= nieuw getal;
   rangnummer INCR 1.
```

Merk op dat de volgorde van de eerste drie regels essentieel is.

Het afdrukken van `rangnummer`, `laatste` en de verhouding gebeurt op een nieuwe regel.

```
druk rangnummer getal en verhouding af:
  line;
  put (rangnummer);
  put(laatste);
  put(laatste / voorlaatste).
```

Bij de formule voor de verhouding tussen `laatste` en `voorlaatste` hebben we een deel-operator gebruikt, die als resultaat een reëel getal oplevert. Daarmee lopen we wat vooruit op de behandeling van de reële getallen.

Bovenstaand programma heeft als nadeel dat het alleen voor computers met `maxint` minstens 2147483647 geschikt is. Beter is het daarom, om in plaats van tot een vast rangnummer te rekenen, te proberen het op onze computer werkelijk grootste berekenbare Fibonacci-getal te vinden. Dit kunnen we bereiken door de berekening van het nieuwe getal uit te voeren zolang er geen overflow kan optreden en juist voordat overflow op zal treden te stoppen. We hoeven daartoe slechts de algoritme `nog niet laatste` te wijzigen. Er treedt duidelijk nog geen overflow op als:

```
laatste + voorlaatste <= maxint
```

Deze test kunnen we echter niet gebruiken, daar dan juist zou gebeuren wat we willen vermijden! We veranderen de test daarom in:

```
nog niet laatste getal:
  laatste <= maxint - voorlaatste.
```

wat hetzelfde resultaat geeft zonder overflow.

De volgende tabel van Fibonacci getallen en hun verhouding is door bovenstaand programma vervaardigd. We zien dat de berekende verhouding na rangnummer 40 niet meer verandert. Hieruit kunnen we echter niet opmaken dat het daar gevonden getal ook exact de limiet is. Een computer berekent zulke getallen nu eenmaal met een beperkte nauwkeurigheid, zoals we in het volgende hoofdstuk zullen zien.

De limiet, afgebroken op 41 decimale cijfers, luidt volgens [KNU72]:

$$1.61803398874989484820458683436563811772 03.$$

De Fibonacci-getallen, zoals de reeks getallen die door bovenstaande betrekking is gedefinieerd genoemd wordt, zijn terug te vinden in allerlei problemen. Zo is bijvoorbeeld de limiet van de verhouding tussen twee opvolgende Fibonacci-getallen een getal dat "Gulden snede" genoemd wordt. In de botanie komen de Fibonacci-getallen voor bij de Phyllotaxis (de rangschikking van bladeren langs een stam) en uit de zoölogie is zojuist een voorbeeld besproken.

3.5.1 Netwerken van typen en operaties

In dit hoofdstuk hebben we een aantal operaties (dat wil zeggen, procedures en operatoren) ingevoerd die op de integers werken. Voor andere typen zullen we nog vele andere operaties invoeren, vaak met dezelfde namen (bijvoorbeeld `+`). Hoe kunnen we een overzicht geven over de operaties behorend bij een bepaald type, anders dan door ze op te sommen?

2	1	1.0000000000000000
3	2	2.0000000000000000
4	3	1.5000000000000000
5	5	1.6666666666666665
6	8	1.5999999999999999
7	13	1.6250000000000000
8	21	1.6153846153846152
9	34	1.6190476190476188
10	55	1.6176470588235292
11	89	1.6181818181818182
12	144	1.6179775280898876
13	233	1.6180555555555554
14	377	1.6180257510729612
15	610	1.6180371352785146
16	987	1.6180327868852458
17	1597	1.6180344478216817
18	2584	1.6180338134001251
19	4181	1.6180340557275541
20	6765	1.6180339631667064
21	10946	1.6180339985218033
22	17711	1.6180339850173577
23	28657	1.6180339901755969
24	46368	1.6180339882053250
25	75025	1.6180339889579018
26	121393	1.6180339886704431
27	196418	1.6180339887802426
28	317811	1.6180339887383028
	...	
37	24157817	1.6180339887498967
38	39088169	1.6180339887498940
39	63245986	1.6180339887498951
40	102334155	1.6180339887498947
41	165580141	1.6180339887498947
42	267914296	1.6180339887498947
43	433494437	1.6180339887498947
44	701408733	1.6180339887498947
45	1134903170	1.6180339887498947
46	1836311903	1.6180339887498947

Een hele sprekende voorstelling is het *Netwerk van Typen en Operaties* (**NTO**) [CRA87]: een type wordt daarbij voorgesteld als een *ovaal*, met daarbinnen de naam van het type, en een operator of procedure als een *rechthoek*, met weer zijn naam daarin geschreven. Deze ovalen en rechthoeken worden met elkaar door *pijlen* verbonden:

- Een pijl van een ovaal *(type)* naar een rechthoek *(operatie)* geeft aan dat die operatie een argument heeft van dat type. Ingeval een operatie meer dan één argument heeft tekenen we voor ieder van zijn argumenten een afzonderlijke pijl naar de bovenkant van de rechthoek, waarbij de argumenten van links naar rechts met de pijlen overeenkomen.
- Een pijl van een rechthoek naar een ovaal betekent dat de betrokken operatie een resultaat van dat type oplevert, hetzij als resultaat van de operatie (zoals bij `+`), hetzij via een variable parameter (zoals bij `get`).

Een NTO laat zien, hoe je door middel van operaties

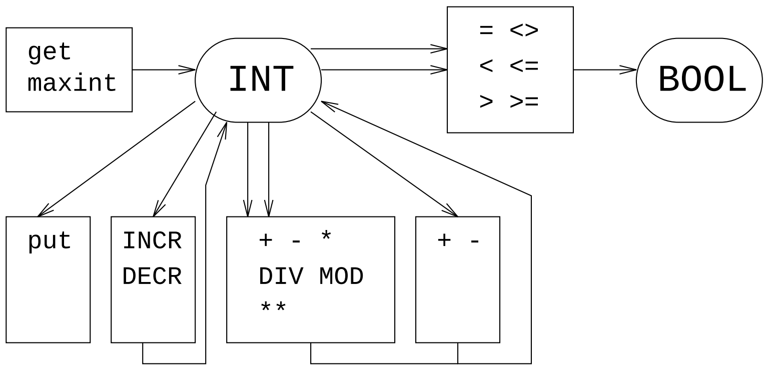


Figure 3.2: NTO voor integers

waarden van de verschillende typen kunt construëren uit andere waarden en wat je daarmee mag doen.

Overigens is een NTO een handige notatie voor operatoren en procedures met weinig parameters, maar als het aantal parameters toeneemt krijg je een wirwar van pijlen. Dat is het nadeel van zon'n plaatje.

In de komende hoofdstukken zullen we voor ieder van de vier concrete elementaire typen een NTO geven. In geen van deze NTO's zullen we expliciet de denotatie, de initialisatie en de toewijzing aangeven, hoewel je kunt argumenteren dat ook die tot de operaties behoren.

De NTO's voor alle verschillende typen vormen tesamen een hypothetische hele grote NTO, waarin ieder van de typen precies ëën keer optreedt. Het heeft weinig zin deze te tekenen, want daarbij zou ieder overzicht verloren gaan.

3.6 Opgaven

1. Verzin een reeks toewijzingen die de inhoud van twee variabelen `a` en `b` verwisselt. Kan het ook zonder hulpvariabele (denk aan overflow)?
2. (intervalsom) Schrijf een programma dat bij een ingelezen getal `s` alle paren integers `i` en `j` vindt met `1 <= i <= j` zodanig dat de som van de getallen in het interval `i` t/m `j` gelijk is aan `s`.
3. (roosterpunten tellen) Bepaal bij een (ingelezen) integer `r` door middel van tellen hoeveel eenheids-roosterpunten binnen de cirkel met straal `r` vallen, d.w.z. hoeveel paren `(x,y)` er bestaan met $x^2 + y^2 <= r^2$.
4. (maximale som van delers) Bepaal dat getal in het interval $[a\ :]$ met $1 \leq a \leq b$ waarvoor de som der delers maximaal is. Let er op dat ook 1 en het getal zelf delers zijn.
5. (Pythagorasgetallen) Bepaal bij een gegeven $k > 0$ alle getallen i en j met $0 < i <= j$ zodanig dat $i^2 + j^2 = k^2$.
6. (nogmaals Pythagorasgetallen) Bepaal alle paren (i, j) met $1 \leq i \leq i_{max}$ en $1 \leq j \leq j_{max}$ zodanig dat $i^2 + j^2$ een kwadraat oplevert.
7. (hagelsteengetallen) De getaltheoreticus Ulam heeft bewezen dat door op een natuurlijk getal herhaaldelijk de volgende regels toe te passen:
   - een even getal wordt door twee gedeeld
   - een oneven getal groter dan 1 wordt met drie vermenigvuldigd en dan met 1 verhoogd

   een reeks natuurlijke getallen wordt verkregen die altijd op 1 eindigt.
   Schrijf een programma om het springerige gedrag te bestuderen van de op deze manier verkregen reeksen.
8. (perfecte getallen) Een getal heet perfect als het gelijk is aan de som van zijn echte delers (1 inbegrepen, maar niet het getal zelf). Schrijf een programma dat alle perfecte getallen onder een bepaalde bovengrens berekent.
9. (bevriende getallen) Laat $s(n)$ staan voor de som van de echte delers van het getal n. Als voor twee getallen $n1$ en $n2$ geldt dat $s(n1) = n2$ en $s(n2) = n1$ dan heten $n1$ en $n2$ bevriend. Schrijf een programma dat alle bevriende getallen onder een bepaalde bovengrens berekent. Denk om de symmetrische paren.
10. (kokosnoten eiland) Vijf zeelieden zijn gestrand op een onbewoond eiland, samen met hun aap. Gelukkig is het eiland rijk aan kokosnoten, waarvan ze er een hele berg verzamelen voordat ze gaan slapen. Een van de zeelieden wordt 'snachts hongerig wakker. Hij geeft 1 kokosnoot aan de aap en neemt van de rest netjes een vijfde deel. Deze deling gaat keurig op, met rest nul. De tweede zeeman wordt wakker, en doet hetzelfde, en zo ook alle anderen — telkens gaat de deling op. Als ze 'smorgens opnieuw wakker worden delen ze de rest keurig in vijven, zonder dat er één kokosnoot overblijft. Deze keer krijgt de aap niets [?].
    Schrijf een programma om het minimum aantal kokosnoten te bepalen dat ze op de vorige dag moeten hebben verzameld.

Chapter 4

Reële getallen

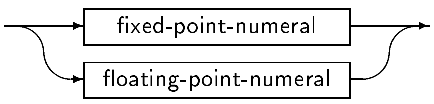
Het volgende elementaire type implementeert de reële getallen. De naam van dit type is `REAL`, en de reële getallen worden in het spraakgebruik als **reals** aangeduid. Reële getallen staan bij allerlei natuurwetenschappelijke en technische berekeningen in het middelpunt; reals zijn daarom in practisch iedere programmeertaal beschikbaar.

4.1 Denotatie en representatie van reals

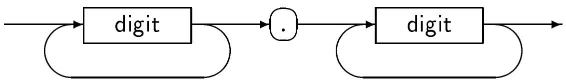
Ook voor reals is de gebruikelijke schrijfwijze een decimale notatie. Ze bestaat uit een reeks decimale cijfers, waarin een decimale punt voorkomt en die gevolgd kan worden door een aanduiding voor een macht van tien.

De syntax-diagrammen voor real-denotaties luiden als volgt:

real-denotation



fixed-point-numeral



De twee verschillende schrijfwijzen heten in het nederlands de "vaste-komma"- en "drijvende-komma"-notatie.

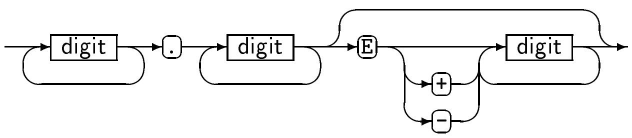
Enkele voorbeelden van real-denotations zijn

```
1.432E-19    1.0      1234.0
0.001234
```

terwijl de volgende notaties er wel op lijken, maar volgens de syntax-diagrammen onjuist zijn (ga dit na!).

```
1.E+2        0001E2   E.2        3.
```

floating-point-numeral



De waarde die door een real-denotation wordt voorgesteld is de gebruikelijke interpretatie als decimaal getal, de exponent geeft een macht van tien als schaalfactor aan. Evenals bij de denotatie van integers kunnen negatieve reals met behulp van de operator `-` worden aangegeven.

Een zelfde real kan met verschillende denotaties worden aangegeven. Zo kan een bepaalde benadering van het getal π bijvoorbeeld worden aangeduid als `3.141592654` of `.3141592654E+001` of `314159265.4E-8`.

4.1.1 Nauwkeurigheid en representatie-fout

De volgende voorwaarden geven een indruk van de stiefmoederlijke wijze waarop de reële getallen door een computer behandeld worden.

- De interne representatie van reals vindt plaats op een wijze die veel weg heeft van een real-denotation.
- Het talstelsel voor de interne representatie van de reals is meestal niet 10 maar 2. Dit is natuurlijk niet essentieel voor de beperkingen die hiermee aan het gebruik van reële getallen in een computer wordt opgelegd, maar het vergroot wel gevoelsmatig de afstand tussen de getallen die de mens hanteert en die welke in de computer gebruikt worden.
- Het aantal binaire cijfers (**bits**) dat een computer voor de interne representatie van een real in het tweetallig stelsel gebruikt is beperkt en verschilt per computer.
- Het aantal binaire cijfers dat voor de interne representatie van de exponent gebruikt wordt is eveneens beperkt.

Deze randvoorwaarden hebben onder andere de volgende gevolgen.

- Slechts een eindig aantal reële getallen kan in een computer exact worden voorgesteld, een eindige deelverzameling van de rationale getallen. De benaming real is dus nogal misleidend.
- Alle andere reële getallen, binnen een bepaald bereik, worden in benadering voorgesteld door een der exact voorstelbare rationale getallen. Hetzelfde geldt voor de resultaten van de reële operaties als `*` en `sqrt`, waar niet eens gegarandeerde de beste benadering wordt geleverd.

- Deze benadering wijkt daardoor met een bepaalde waarde van het bedoelde reële getal af. De afwijking wordt **afrondings fout** genoemd.
- Door de representatie van reële getallen als drijvende-komma-getallen is bij (in absolute waarde) grote reële getallen de representatiefout navenant.
- Op computers die intern het 2-tallig stelsel hanteren zijn decimale breuken als 0.1 niet exact voorstelbaar. Dit heeft tot gevolg dat `10 * 0.1` op zulke computers niet gelijk is aan `1.0`.

4.1.2 Machine-afhankelijkheid

De eigenschappen van de reals verschillen van computer tot computer. Om ze voor een bepaalde computer aan de weet te komen zijn twee real constanten voor U gedeclareerd in de standaard-pakketten.

De constante `smallreal` is de in absolute waarde kleinste positieve real, die bij 1.0 opgeteld een van 1.0 verschillend resultaat levert. Het getal `smallreal` is dus een maat voor de **relatieve nauwkeurigheid** waarmee de computer rekent. De constante `maxreal` heeft als waarde de grootste real die de computer nog kan voorstellen. Het getal `maxreal` vertelt dus iets over het **real bereik** van de computer.

In de volgende tabel zijn voor een aantal computers de waarden van `smallreal` en `maxreal` aangegeven. De in de tabel gebruikte notatie voor deze getallen geeft tevens aan welk talstelsel intern wordt gehanteerd.

	smallreal	maxreal
numbercruncher	2^{-47}	$(2^{-48}-1) * (2^{1022})$
decimale machine	10^{-12}	$(1-10^{-10}) * (10^{49})$
IBM mainframe	16^{-6}	$(1-16^{-6}) * (16^{63})$
32-bits computer	2^{-63}	$(1-2^{-63}) * (2^{127})$
microcomputer	2^{23}	$(1-2^{-23}) * (2^{127})$

4.1.3 Overflow en underflow

Naast **overflow**, die optreedt als een berekend resultaat groter dan `maxreal` blijkt, kan bij de real-arithmetiek ook **underflow** optreden, namelijk als een resultaat te klein is om gerepresenteerd te worden, d.w.z. als de exponent sterker negatief is dan bij de getalrepresentatie past. Dan kan een afronding op nul plaatsvinden. Dat is wat minder tragisch dan overflow.

4.1.4 Terminologie

Samenvattend kunnen we de volgende termen onderscheiden:

reëel getal	een wiskundig begrip.
real	een in de computer voorstelbaar rationaal getal waarmee een reëel getal wordt benaderd.
`REAL`	de naam van het type der reals.
real-denotation	de schrijfwijze voor reals in een programma.

4.2 Operaties op reals

4.2.1 Arithmetische operaties

Reals zijn bij uitstek bedoeld om er berekeningen mee uit te voeren. Alle gebruikelijke arithmetische operatoren voor reële getallen zijn dan ook als standaard-operatoren in de taal Elan gegeven.

operator	betekenis	voorbeeld
`+`	optellen	`1.0 + 3.0` (= 4.0)
`-`	aftrekken	`7.5 - 1.1` (= 6.4)
`*`	vermenigv.	`2.0 * 4.7` (= 9.4)
`/`	delen	`3.8 / 2.0` (= 1.9)
`MOD`	modulo	`3.4 MOD 1.2` (= 1.0)

Deze operatoren zijn gedefinieerd voor real operanden en leveren daarmee een real resultaat. De modulo-operator levert, analoog aan het integer geval, de rest bij deling, en heeft dezelfde vóórteken-eigenschappen.

De real deling kan ook gebruikt worden met twee integer operanden, maar (helaas) niet in mengvormen

operator	betekenis	voorbeeld	resultaat
`/`	integer deling	`3/4`	(= 0.75)

als `3/4.2` (maar zie paragraaf 4.2.4).

De prioriteit van de arithmetische operatoren voor reals is dezelfde als die van de gelijknamige operaties voor integers; de prioriteit is namelijk een eigenschap van de naam van de operator en niet van de er mee bedoelde operatie.

De machtsverheffing heeft als linker operand een real en als rechter een integer (die groter dan nul moet zijn).

operator	betekenis	voorbeeld
`**`	machtsverheffen	`-1.3 ** 2` (= 1.69)

4.2.2 Reële functies

De volgende **functiess** met een real argument en een real resultaat zijn eveneens in Elan gegeven:

De zes goniometrische functies verwachten hun argument in radialen, respectievelijk leveren hun resultaat in radialen. Het zijn geen operatoren maar **procedures**, waarvan bij oproep de **parameters** tussen de haakjes `(` en `)` geplaatst moet worden. Een oproep van een functie kan op zijn beurt als operand in een uitdrukking optreden. Voorbeelden:

```
sin (x) ** 2 + cos (x) ** 2
exp (sin (x) ** 2 + cos (x) ** 2)
```

Deze functies zijn gedefinieerd met de in de wiskunde gebruikelijke beperkingen op hun argument. Zo is de functie `sqrt` ("square root") alleen gedefinieerd voor niet-negatieve argumenten. Probeer dus nooit de wortel uit een negatief getal te trekken.

4.2.3 Vergelijkingsoperatoren

Het vergelijken van reals kan met behulp van de bekende operatoren:

naam	betekenis
`sqrt`	worteltrekken
`exp`	e-tot-de-macht
`ln`	natuurlijke logaritme
`sin`	sinus
`cos`	cosinus
`tan`	tangens
`arcsin`	arcsinus
`arccos`	arccosinus
`arctan`	arctangens
`abs`	absolute waarde

```
<>    <    <=    =    >=    >
```

Wees voorzichtig met het vergelijken van reals: vanwege het optreden van afrondingsfouten hangt de representatie van een resultaat sterk van zijn wordingsgeschiedenis af. Het is bijvoorbeeld vrijwel zeker dat `0.1 * 10` ongelijk is aan `1.0`. In plaats van `a = b` schrijve men liever `abs(a - b) < grens` met een voldoende kleine waarde voor `grens`.

4.2.4 Conversie-operaties

De volgende functies dienen om de **conversie** van een integer naar een real en omgekeerd mogelijk te maken.

naam	betekenis
`round`	afronden naar integer
`trunc`	afkappen naar integer

De functie `round` levert een integer resultaat, de dichtstbij gelegen integer.

```
round (4.3) = 4
round (4.8) = 5
round (-4.3)= -4
round (-4.8)= -5
```

Een getal eindigend op .5 wordt geacht naar boven afgerond te worden, maar als het eindigt op zoiets als .4999999999 kan het heel wel naar beneden afgerond worden.
De functie `trunc` levert de grootste integer waarde die niet groter is dan z'n operand.

```
trunc (4.3) =  4
trunc (4.8) =  4
trunc (-4.3)= -5
trunc (-4.8)= -5
```

Door de inherente onnauwkeurigheid van de reals is het vrijwel zeker dat een uitdrukking als `trunc (10.0 * 0.1)` nul oplevert in plaats van één.

Een belangrijke toepassing van de omzettingsoperaties vinden we bij arithmetische operaties op operanden van verschillend type, zoals `REAL` en `INT`, bijvoorbeeld:

```
INT VAR k :: 0;
REAL CONST dx :: 0.001;
WHILE k <= 100
REP
  k INCR 1;
  REAL CONST x  :: round (k)*dx;
  ...
```

4.2.5 Invoer en uitvoer van reals

Het lezen en schrijven van reals gaat, evenals het lezen en schrijven van integers, met de concrete algoritmen `get` en `put`. De getallen die door de **invoer**-algoritme `get` gelezen kunnen worden hebben dezelfde vorm als de real-denotations en kunnen worden voorafgegaan door een teken.

Het aantal cijfers dat door de **uitvoer**-algoritme `put` wordt afgedrukt hangt af van de nauwkeurigheid waarmee de computer de reals representeert. Dit aantal is in ieder geval voldoende om alle significante cijfers van een real te tonen, waarbij u zich er goed van bewust moet zijn dat er afrondingsfouten zullen zijn opgetreden. Wees dus niet verbaasd als u met veel omhaal een resultaat gelijk aan twee heeft berekend en het wordt afgedrukt als `0.199999994E+1`.

4.2.6 NTO voor de reals

Op de volgende pagina staat een overzicht over het type `REAL` in de vorm van een NTO, waarin de volgende operaties worden onderscheiden:

- de invoer procedure `get`
- de standaard constanten `smallreal`, `maxreal` en `pi`
- de conversie procedures `real`, `round` en `trunc`
- de uitvoer procedure `put`
- de vergelijkingsoperatoren `<`, `<=`, `=`, `>=`, `>` en `<>`
- de monadische operatoren `+` en `-`
- de reële functies `sqrt`, `abs`, `sin`, `cos`, `tan`, `arctan`, `arcsin`, `arccos`, `arctan`, `arcsin`, `ln` en `exp`
- de dyadische operatoren `+`, `-`, `*`, `/`, `MOD` en `**`.

Dit NTO maakt samen met andere NTO's deel uit van een groter geheel dat alle operaties van Elan omvat. Voor een compleet overzicht over die operaties verwijzen wij naar Appendix B.

4.3 Voorbeeld: Vierkantsvergelijking

Hoewel de reals speciaal bedoeld zijn om er berekeningen met reële getallen mee uit te voeren, moet in de praktijk van het rekenen een grote mate van voorzichtigheid in acht worden genomen. De reals vormen slechts een schamele realisering van de reële getallen. We kunnen niet zomaar een formule nemen en deze in een programma verwerken. In vele gevallen zal dan het resultaat, zeker wanneer de tussenresultaten in de buurt van de grenzen van de nauwkeurigheid van de reals komen, slechts in de verte lijken op het verwachte resultaat. We willen dit met een voorbeeld illustreren.

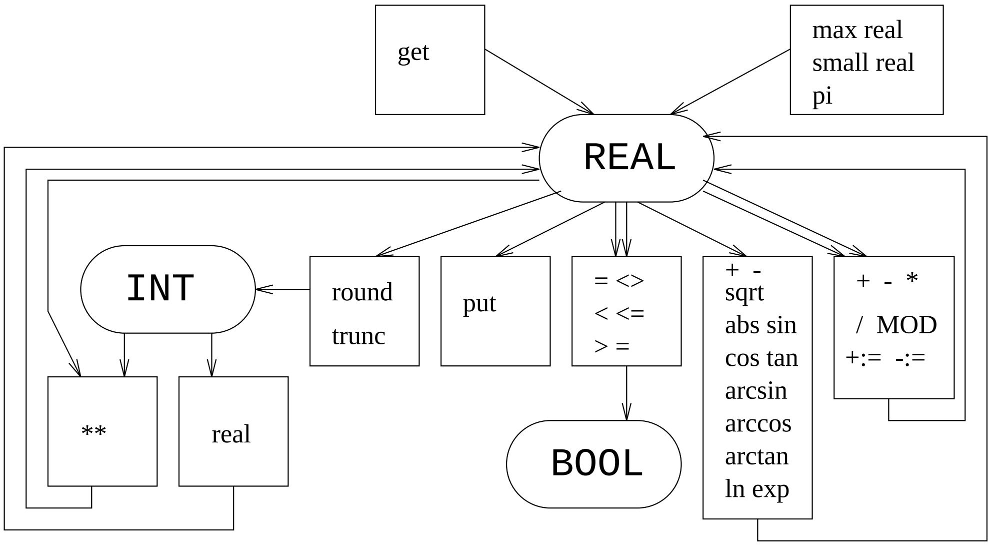


Figure 4.1: NTO voor het type `REAL`

Een bekende formule voor de wortels van een (niet ontaarde) vierkantsvergelijking is de abc-formule. Van de vergelijking $a\ x^2 + b\ x + c = 0$ zijn de wortels, volgens uw middelbareschool-wiskunde, gelijk aan

$$x_1, x2 \ = \ \frac{-b \pm \sqrt{b^2 - 4\ a\ c}}{2\ a}$$

Het schrijven van een algoritme om voor een aantal waarden van `a`, `b` en `c` deze wortels te berekenen is een oefening in het "plat-slaan" (lineair opschrijven) van formules.

```
vierkantswortels:
  lees de coefficienten;
  druk de wortels af.
```

```
lees de coefficienten:
  REAL VAR a, b, c;
  put line ("                 2");
  put line ("De wortels van ax + bx + c");
  put line ("Geef a");
  get (a);
  put line ("Geef b");
  get (b);
  put line ("Geef c");
  get (c).
```

We grijpen hier schaamteloos vooruit op hoofdstuk 6 door teksten uit te geven om een mooi schermbeeld te krijgen: Een oproep als `put ("Geef a")` schrijft de tekst `Geef a` (zonder de aanhalingstekens) vanaf de huidige positie op het beeldscherm.

```
druk de wortels af:
  IF de vierkantsvergelijking is ontaard
  THEN
    bereken snijpunt van de ontaarde vgl
  ELSE
    IF discriminant kleiner dan nul
    THEN
      meld dat er geen wortels zijn
    ELSE
      bereken de wortels;
      druk ze af
    FI
  FI.

de vierkantsvergelijking is ontaard:
  a = 0.0.

bereken snijpunt van de ontaarde vgl:
  IF b = 0.0
  THEN
    IF c = 0.0
    THEN
      meld dat er te veel wortels zijn
    ELSE
      meld dat er geen wortels zijn
    FI
  ELSE
    geef de enige wortel
  FI.

meld dat er te veel wortels zijn:
  put line("Oneindig veel wortels").

meld dat er geen wortels zijn:
   put line ("Er zijn geen wortels").

geef de enige wortel:
  put line ("De enige wortel is: x =");
  put (- c / b).

discriminant kleiner dan nul:
  REAL CONST discriminant :: b**2-4.0*a*c;
  discriminant < 0.0.
```

```
bereken de wortels:
  REAL CONST x1:: (-b-sqrt(discriminant))/
                  (2.0 * a);
  REAL CONST x2:: (-b+sqrt(discriminant))/
                  (2.0 * a).

druk ze af:
  put line ("x1 =");
  put (x1);
  put line ("x2 =");
  put (x2).
```

In tabel 4.1 zijn voor de vierkantsvergelijking in de linkerkolom de wortels volgens deze abc-methode en in de rechterkolom de juiste waarden weergegeven. De berekeningen zijn uigevoerd op een computer met `smallreal` = 1.1920929E-7 en `maxreal` = 1.7014117E+38.

We zien dat voorbij `i = 7` volgens de abc-formule de waarde van de kleinste wortel 0 is in plaats van 1 ! De merkwaardige resultaten liggen niet zozeer aan de algoritme, maar aan het feit dat met reals in plaats van echte reële getallen gerekend wordt. Het blijkt dat de in absolute waarde kleinste wortel een grote relatieve fout kan hebben, vooral wanneer de wortels van de vergelijking in absolute waarde veel verschillen. Dit laat zich gemakkelijk verklaren als we bedenken dat in dat geval `a` en `c` klein zijn ten opzichte van `b`. De wortel uit de discriminant en `b` zullen ongeveer gelijk zijn en het verschil van deze waarden zal, vanwege de eindige precisie, een grote relatieve fout hebben.

Het is daarom beter om voor de wortels van een vierkantsvergelijking de volgende formules te gebruiken:

$$x_1 = -\frac{\frac{2c}{b}}{1 + \sqrt{1 - \frac{4ac}{b^2}}}$$

en

$$x_2 = -\frac{c}{ax_1}$$

Met deze formules vinden we de absoluut kleinste wortel als x_1 en de absoluut grootste wortel als x_2.

Voor meer ingewikkelde berekeningen kunnen de problemen veel groter zijn dan dit voorbeeld illustreert. Het is dan ook niet verwonderlijk dat de kennis van de technieken die nodig zijn om ondanks de beperkte nauwkeurigheid bruikbare resultaten te verkrijgen, een zelfstandig wiskundig vak is: de *numerieke wiskunde*.

4.4 Voorbeeld: Gemiddelde en spreiding

Een bij fabricage-processen veel voorkomend probleem is de bewaking van de kwaliteit van het geproduceerde. Stel bijvoorbeeld dat een zuivelfabriek een krentenpap produceert waaraan de eis gesteld wordt dat iedere fles pap gemiddeld 50 krenten bevat. Het aantal krenten in een fles mag niet te laag zijn wil het product nog de naam krentenpap mogen dragen, terwijl aan de andere kant het aantal krenten per fles pap niet groter mag zijn dan economisch verantwoord is.

Een (toegegeven: ietwat vreemdsoortige) manier om een pap te produceren die aan bovengenoemde voorwaarden voldoet is, om in elke fles pap het aantal krenten te tellen en het productie-proces bij te stellen, zodra belangrijke afwijkingen optreden. Belangrijke afwijkingen treden op als bijvoorbeeld het gemiddelde aantal krenten per fles te veel afwijkt van de doelstelling (50). Ook is het niet goed verkoopbaar om telkens in één fles pap helemaal geen krenten en in de volgende fles twee maal het gemiddelde aantal krenten te doen, hoewel dit het gemiddelde aantal krenten per fles constant zou houden. Een maat voor de constantheid van de kwaliteit van de pap is de spreiding van het aantal krenten per fles rond het gemiddelde. Deze spreiding mag niet te groot zijn.

Het is natuurlijk niet doenlijk om in elke fles pap het aantal krenten te tellen. In de praktijk zullen dan ook van slechts een relatief klein aantal flessen de krenten geteld worden. Het resultaat van zo'n telling heet een steekproef. De statistiek leert ons hoe groot we de steekproeven moeten nemen om (via de volgende formules) een redelijke aanduiding te verkrijgen voor het gemiddelde en de spreiding van het aantal krenten per fles voor alle flessen.

Zij W_i de i-de waarde uit de steekproef, d.w.z. het aantal krenten in de i-de fles, en n de grootte van de steekproef, d.w.z. het aantal flessen waarin krenten geteld zijn. Het gemiddelde wordt dan gegeven door de formule:

$$g = \frac{\sum_{i=1}^{n} W_i}{n}$$

en de spreiding door:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} W_i^2 - n * g^2}{n-1}}$$

Voor een steekproefgrootte van 1 is de spreiding niet bepaald.

Een programma dat het gemiddelde en de spreiding van het aantal krenten per fles met behulp van deze formules berekent zal, na de waarde van de steekproefgrootte, achtereenvolgens de waarden W_i inlezen en de som van de kwadraten berekenen. Als alle waarden ingelezen zijn, kan met behulp van de twee sommen het gemiddelde en de spreiding aan de hand van de formules berekend worden.

Het is niet nodig alle waarden W_i te bewaren, maar het is voldoende om de som van de waarden W_i en van de kwadraten van W_i^2 te berekenen. Daartoe is het voldoende, één W_i tegelijk te lezen.

Bij deze sommaties kan natuurlijk overflow optreden. Dit zal bijna zeker gebeuren wanneer we de sommen als integers willen berekenen. Om deze reden zullen we de sommen als reals berekenen en er maar op hopen dat het bereik van de reals in de te gebruiken computer groot genoeg is.

Zonder verdere verklaring laten we hieronder het programma "verfijnd" volgen:

$(x - 10.0^i) * (x - 1.0) = 0.0$ voor $i = 1, 2, 3, \ldots$			
met abc-formule		**met verbeterde formule**	
x_1	x_2	x_1	x_2
+1.0000000E +0	+1.0000000E +1	+1.0000000E +0	+1.0000000E +1
+1.0000000E +0	+1.0000000E +2	+1.0000000E +0	+1.0000000E +2
+1.0000000E +0	+1.0000000E +3	+1.0000000E +0	+1.0000000E +3
+1.0000000E +0	+1.0000000E +4	+1.0000000E +0	+1.0000000E +4
+1.0000000E +0	+1.0000000E +5	+1.0000000E +0	+1.0000000E +5
+1.0000000E +0	+1.0000000E +6	+1.0000000E +0	+1.0000000E +6
+1.0000000E +0	+1.0000000E +7	+1.0000000E +0	+1.0000000E +7
+0.0000000E +0	+1.0000000E +8	+1.0000000E +0	+1.0000000E +8
+0.0000000E +0	+1.0000000E +9	+1.0000000E +0	+1.0000000E +9
+0.0000000E +0	+1.0000000E+10	+1.0000000E +0	+1.0000000E+10
+0.0000000E +0	+1.0000000E+11	+1.0000000E +0	+1.0000000E+11
+0.0000000E +0	+1.0000000E+12	+1.0000000E +0	+1.0000000E+12
+0.0000000E +0	+1.0000000E+13	+1.0000000E +0	+1.0000000E+13
+0.0000000E +0	+1.0000000E+14	+1.0000000E +0	+1.0000000E+14
+0.0000000E +0	+1.0000000E+15	+1.0000000E +0	+1.0000000E+15
+0.0000000E +0	+1.0000000E+16	+1.0000000E +0	+1.0000000E+16
+0.0000000E +0	+1.0000000E+17	+1.0000000E +0	+1.0000000E+17
+0.0000000E +0	+1.0000000E+18	+1.0000000E +0	+1.0000000E+18
+0.0000000E +0	+1.0000000E+19	+1.0000000E +0	+1.0000000E+19

Table 4.1: De wortels van een vierkantsvergelijking

```
lees aantal metingen;
bereken som en kwadraatsom;
bereken gemiddelde en spreiding;
druk waarden af.

lees aantal metingen:
   put("steekproefgrootte = ");
   INT VAR n;
   get(n);
   line(2).

bereken som en kwadraatsom:
   INT VAR aantal :: 0;
   REAL VAR som :: 0.0,
            kwadraatsom :: 0.0;
   WHILE niet laatste waarde
   REP
     lees volgende waarde;
     pas som en kwadraatsom aan
   ENDREP.

niet laatste waarde:
   aantal < n.

lees volgende waarde:
   REAL VAR waarde;
   get(waarde);
   aantal INCR 1.

pas som en kwadraatsom aan:
   som INCR waarde;
   kwadraatsom INCR waarde ** 2.

bereken gemiddelde en spreiding:
  REAL CONST gemiddelde :: som/round(n);
  REAL CONST spreiding ::

sqrt((kwadraatsom-round(n)*gemiddelde**2)/
            round(n-1)).
```

Let op de wat krampachtige wijze waarop wij de steekproefgrootte `n` in een real om moeten zetten om de real-deling en vermenigvuldiging te kunnen gebruiken.

We kunnen ook schrijven

```
    REAL CONST spreiding ::
       sqrt((kwadraatsom-som*gemiddelde)
          /round(n-1))
```

wat op iets slimmere wijze hetzelfde resultaat geeft.

```
  druk waarden af:
    put line ("Steekproefgrootte =");
    put(n);
    put line ("Gemiddelde        =");
    put(gemiddelde);
    put line ("Spreiding         =");
    put(spreiding).
```

Wij installeren dit programma'tje op de werkvloer van de zuivelfabriek, scholen het noodzakelijke personeel en constateren met trots dat de kwaliteitsbewaking verbeterd is (zodat de klanten nu gemiddeld minder krenten in hun pap vinden). Maar na een paar weken komt de voorman asgrauw ons kantoortje binnengestormd met de woorden: "Hij doet het niet meer ...". Wij volgen hem naar de computer en lezen op het beeldscherm de boodschap

```
  FATAL ERROR AT ADDRESS 000247
              IN MODULE XC00D3E4
  OVERFLOW IN REAL DIVISION
  SYMBOLIC DUMP FOLLOWS
```

met daarachter nog veel meer fraais.

Na enig zoeken vinden we dat, door een type-fout, de operateur als steekproefgrootte één had aangegeven. Het programma is daar niet tegen bestand.

Na enig nadenken wijzigen we één verfijning

```
lees aantal metingen:
  INT VAR n;
  REP put ("steekproefgrootte (>1) =");
      get (n);
      line (2)
  UNTIL n > 1
  ENDREP.
```

zodat het programma wat robuuster is geworden.

4.5 Opgaven

1. (Vierkantswortel) Een van de oudste algoritmen is de volgende manier om van een getal de vierkantswortel te berekenen: Als x_i een benadering is voor de wortel van a dan is een betere benadering x_{i+1} te berekenen als $x_{i+1} = 0.5(x_i + a/x_i)$. Schrijf een programma om van een ingelezen getal de wortel te bepalen. Vergelijk de bereikte nauwkeurigheid met die van de standaard-functie `sqrt`.
2. Ga het gedrag van de real arithmetiek na door de volgende berekeningen uit te voeren:
   - $1 - 1/3 - 1/3 - 1/3$
   - $1 - 1/6 - 1/6 - 1/6 - 1/6 - 1/6 - 1/6$
   - $x - \sqrt{x^2}$ voor enkele waarden van x
   - $x - (\sqrt{x})^2$
   - $x - \tan(\arctan x)$
3. Bereken e en 1/e , gebruik makend van de reeksen

$$e = 1 + 1/1! + 1/2! + 1/3! + \ldots$$

$$1/e = 1 - 1/1! + 1/2! - 1/3! + \ldots$$

   Laat bij iedere stap de benadering van e en 1/e zien, evenals ter controle het product van die twee.
4. (minimaliseren) Zoek een minimum van een concave functie in een variabele door herhaalde halvering van het interval waarin dat minimum moet liggen. Zoek hiermee die waarde in het interval [3.0 : 4.0] waarvoor cos(x) minimaal is.
5. Bereken $\pi/4$ door middel van de volgende reeks:

$$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - \ldots$$

   Druk bij iedere stap de benadering van $\pi/4$ af en vergelijk dit met de computerwaarde van $\pi/4$.
6. Bereken $\ln 2$ met behulp van de reeks

$$ln2 = 1 - 1/2 + 1/3 - 1/4 + 1/5 - \ldots$$

   Neem ter controle steeds de exp van je benadering.
7. Bepaal experimenteel de waarden van `maxreal` en `smallreal` op jouw computer zonder de standaard constanten met die naam te gebruiken. Bepaal eveneens de grootste real die kleiner is dan `maxreal`.

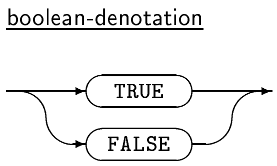
Chapter 5

Waarheidswaarden

We onderscheiden tussen twee **waarheidswaarden**: *waar* en *onwaar*. Ter ere van George Boole (1815-1864) die de "logische algebra" of "schakelalgebra" heeft uitgevonden, worden deze waarden Boole'se waarden genoemd; in computer-engels: **booleans**. Hun type heeft de naam `BOOL`.

Waarheidswaarden hebben in programma's twee aspecten. In de eerste plaats treden ze op als waarden van objecten die door algoritmen gemanipuleerd kunnen worden. In dit opzicht gedragen ze zich op dezelfde manier als integers en reals. Aan de andere kant kunnen waarheidswaarden ook de uitvoering van algoritmen sturen: in de keuze en de herhaling wordt de uitvoering van de algoritme gestuurd door het resultaat van een voorwaarde. In de meeste programmeertalen wordt tussen deze twee aspecten van de waarheidswaarden geen onderscheid gemaakt, maar gevoelsmatig zijn ze wat moeilijk onder een noemer te verenigen (zie 5.2.

Omdat er maar twee waarheidswaarden zijn, bestaat de syntax van de boolean-denotations uit een opsomming van de mogelijkheden.



De ja-waarde wordt gerepresenteerd als `TRUE` en de nee-waarde als `FALSE`. In de standaardpakketten zijn twee Boole'se constanten `true` en `false` gedeclareerd, ten behoeve van diegenen die (zoals de schrijver dezes) niet van opdringerige hoofdletters houden. We mogen `TRUE` en `FALSE` daarom naar believen ook met kleine letters schrijven.

De volgende termen moeten onderscheiden worden:

5.1 Elementaire algoritmen voor booleans

Behalve via een boolean-denotation kan een waarheidswaarde in een programma ook verkregen worden via een algoritme die een boolean waarde levert (voorwaarde, "test"). Zo'n waarde wordt berekend als het resultaat van een uitdrukking, gebruik makend van de hierna genoemde operaties.

5.1.1 Vergelijkingsoperatoren

In Elan zijn voor alle elementaire typen twee operatoren gegeven, die vaststellen of de waarden van hun operanden gelijk of ongelijk zijn.

De bedoelde operatoren zijn:

operator	betekenis
=	gelijk aan
<>	ongelijk aan

die gedefinieerd zijn voor twee `INT`, `REAL`, `BOOL` of `TEXT` argumenten. Het resultaat van zo'n test is steeds een waarheidswaarde.

Voor de booleans zijn dit de enige **vergelijkings operatoren**. Voor die elementaire typen waarvoor een ordeningsrelatie bestaat, zoals `INT` en `REAL` maar ook `TEXT` (zie het volgende hoofdstuk), is bovendien een viertal operatoren gegeven die volgens deze ordeningsrelatie testen. Het zijn de operatoren: die eveneens twee

operator	betekenis
<	kleiner dan
>	groter dan
>=	minstens
<=	hoogstens

operanden van eenzelfde type verlangen en een waarheidswaarde opleveren. Deze operatoren zijn, net als de arithmetische operatoren, niet gedefinieerd voor combinaties van integer en real operanden; het type van beide operanden moet steeds gelijk zijn. De **prioriteit** van de vergelijkingsoperatoren is lager dan de prioriteit van de arithmetische operatoren, zodat een formule als `b ** 2 < 4 * a * c` de vergelijking betekent van `b ** 2` met `4 * a * c`.

De twee verschillende aspecten van waarheidswaarden — zowel elementair object als besturing van de uitvoering van algoritmen — komen het best tot uitdrukking in het gebruik van formules als de bovenstaande. Zo zou deze formule gebruikt kunnen worden in een keuze als:

```
IF
  b ** 2 < 4 * a * c
THEN
  bereken complexe wortels
ELSE
  bereken reele wortels
FI
```

waarheidswaarde	de begrippen waar en onwaar
boolean	de representaties van deze waarheidswaarden in de computer
BOOL	de naam van het type der booleans
boolean denotation	de notatie voor boolean waarden

Het resultaat van de test wordt hier direct gebruikt om de keuze tussen het berekenen van de complexe of reële wortels van een vierkantsvergelijking te bepalen. Het resultaat van een test is echter een waarheidswaarde, zodat de test ook gebruikt kan worden in een toewijzing of een declaratie van een boolean constante.

```
      BOOL CONST discriminant is negatief ::
  b ** 2 < 4 * a * c
```

De uitkomst van de test wordt zo bewaard als een constante en kan dan later gebruikt worden in een keuze als:

```
IF
  discriminant is negatief
THEN
  bereken complexe wortels
ELSE
  bereken reele wortels
FI
```

Vanzelfsprekend is het vooral nuttig deze constante te declareren als de waarde ervan meer dan een keer wordt gebruikt. Merk op dat deze constante uiteraard niet van waarde verandert als we naderhand de waarden van a, b of c veranderen. Zijn waarde wordt niet opnieuw berekend, in tegenstelling tot de waarde van een verfijning als

```
discriminant is negatief:
    b ** 2 < 4 * a * c.
```

die uiteraard bij iedere oproep herberekend wordt.

5.1.2 Logische operatoren

Voor de manipulatie van waarheidswaarden als elementaire objecten zijn een aantal standaard operatoren gegeven, de **logische operatoren**. Het resultaat van deze operatoren voor verschillende waarden van hun operanden volgt uit onderstaande tabel.

De operator AND heeft een hogere prioriteit dan de operatoren OR en XOR, maar een lagere prioriteit dan de vergelijkingsoperatoren. De monadische operator NOT heeft, net als alle andere monadische operatoren, de hoogste prioriteit. Met behulp van deze operatoren en met behulp van de vergelijkingsoperatoren voor booleans = en <> kunnen de zestien operatoren van de propositie-logica op eenvoudige wijze gerealiseerd worden, zelfs op meer dan een manier. Zo zijn zowel

```
p = q  OR  q
```

als

```
NOT p  OR  q
```

formules die de implicatie $p \rightarrow q$ berekenen.

5.1.3 Gecombineerde voorwaarden

Enige voorzichtigheid met het gebruik van vergelijkingsoperatoren voor waarheidswaarden is geboden. De formule

```
p = q = r
```

betekent niet wat u hoopt: omdat bij gelijke prioriteit van de operatoren deze van links naar rechts worden uitgevoerd is de formule equivalent met

```
(p = q) = r
```

Wanneer we alleen als resultaat TRUE willen zien in het geval dat p, q en r alle dezelfde waarde hebben, dan moeten we schrijven

```
p = q  AND  q = r
```

Bij ordenings-tests is een formule van de vorm

```
a <= b <= c
```

syntactisch fout, omdat het resultaat van de eerste (linker) operatie een waarde van het type BOOL is, die niet vergeleken kan worden met bijvoorbeeld een integer. In dit geval zullen we dus

```
a <= b  AND  b <= c
```

moeten schrijven als we het verlangde resultaat willen bereiken.

Let er op dat ook bij de logische operatoren in ieder geval alle operanden berekend worden, zelfs al staat na het berekenen van één operand het resultaat al vast. Zo is:

```
b is minstens tien keer a:
  a <> 0 AND b DIV a >= 10.
```

geen geschikte manier om te voorkomen dat een deling door nul optreedt. In plaats daarvan schrijve men

```
b is minstens tien keer a:
  IF
    a <> 0
  THEN
    b DIV a >= 10
  ELSE
    false
  FI.
```

5.1.4 In- en uitvoer van booleans

Voor de **invoer** en **uitvoer** van waarheidswaarden stelt Elan geen algoritmen ter beschikking. In plaats van deze waarheidswaarden zal men in het algemeen teksten inlezen en uitschrijven, zoals we in het volgend hoofdstuk leren.

operanden		operaties			
p	q	NOT p	p AND q	p OR q	p XOR q
TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
TRUE	FALSE		FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE		FALSE	FALSE	FALSE

5.2 Vertrouwen in de booleans

Het tweeslachtig karakter van de booleans (waarde en besturing) geeft bij vele programmeurs aanleiding tot vreemde omslachtigheden en stijlfouten die kunnen worden verklaard als een gebrek aan vertrouwen in de booleans. Neem bijvoorbeeld aan dat a een Boole'se variabele is. Vele programmeurs schrijven

```
IF a = TRUE THEN ...
```

waar eenvoudigweg

```
IF a THEN ...
```

genoeg zou zijn. Blijkbaar kunnen zij zich niet realiseren dat de uitdrukking a alleen al een boolean oplevert, en proberen zij er via de test een stuurwaarde van te maken. Toegegeven, het staat ook wel een beetje kaal zo.

Een vergelijkbaar wantrouwen tegen de integers zou leiden tot het even absurde

```
put (i + 0)
```

om zeker te zijn dat de waarde van de integer variabele i echt een geheel getal oplevert.

Stijlbloempjes als

```
    BOOL VAR t;
    t := IF a > b THEN TRUE ELSE FALSE FI;
    IF (t = TRUE) = TRUE THEN t := FALSE
FI
```

doen stoethaspelig aan en getuigen van een fundamenteel onbegrip voor het tweeslachtig karakter van de booleans.

Het is instructief, kortere versies voor de volgende teksten te bedenken.

```
IF a > 0 THEN TRUE ELSE FALSE FI

IF a THEN NOT a ELSE TRUE FI

IF a THEN FALSE ELSE a FI
```

5.3 Voorbeeld: Primaliteit bepalen (1)

Beschouw het probleem, van een positief geheel getal vast te stellen of het een priemgetal is, d.w.z. geen delers heeft behalve 1 en zichzelf.

Als wortel van het programma kiezen we

```
priemtest:
  lees een getal;
  bepaal of het getal priem is;
  druk het antwoord af.
```

Het lezen van het getal geeft weinig problemen.

```
lees een getal:
  INT VAR getal;
  get (getal).
```

Of een getal priem is kan men natuurlijk bepalen door in een priementabel te kijken. Om het probleem echter interessant te houden zullen we maar aannemen dat zo'n tabel niet ter beschikking staat.

Men kan ook het getal delen door alle priemgetallen die kleiner zijn dan dat getal en telkens kijken of de rest nul is, maar daarvoor is weer een tabel nodig! We kunnen echter net zo goed delen door alle getallen — ook de niet-priemgetallen — die kleiner zijn dan het getal. Dat kost wel meer werk, maar leidt tot hetzelfde resultaat. Daarvoor hebben we nu eenmaal een computer.

Het antwoord op de vraag naar de primaliteit zullen we in een boolean variabele geen delers gevonden vastleggen, die aanvankelijk TRUE is en FALSE wordt gemaakt als we een deler vinden. De eerste deler die we proberen is 2.

```
bepaal of het getal priem is:
  BOOL VAR geen delers gevonden :: TRUE;
  INT VAR kandidaat :: 2;
  WHILE nog kandidaten te proberen
  REP
    kijk of deling opgaat;
    neem de volgende kandidaat
  ENDREP.

kijk of deling opgaat:
  IF deling gaat op
  THEN geen delers gevonden := FALSE
  FI.
```

Hoe stelt men vast of een getal deelbaar is door een ander getal? Een manier is de volgende. We delen het getal eerst door de deler en vergeten de rest. Vervolgens vermenigvuldigen we het quotiënt weer met de deler en vergelijken het resultaat met het oorspronkelijke getal. Hierbij treedt alleen gelijkheid op als de eerste deling opging.

```
deling gaat op:
  (getal DIV kandidaat) * kandidaat =
getal.
```

Strikt genomen zijn de haakjes hier overbodig, maar ter verduidelijking van dit stukje programma hebben we ze toch maar gezet.

Eenvoudiger is hier het gebruik van de operator `MOD`, die voor positieve operanden de rest na deling levert.

```
deling gaat op:
  getal MOD kandidaat = 0.
```

Als potentiële delers onderzoeken we voorlopig alle getallen kleiner dan het getal zelf, grotere getallen zijn zeker geen delers.

```
nog kandidaten te proberen:
  kandidaat < getal.
```

```
neem de volgende kandidaat:
  kandidaat INCR 1.
```

Voor de laatste verfijning, `druk het antwoord af`, lopen we vooruit op het volgende hoofdstuk en drukken één van twee teksten af.

```
druk het antwoord af:
  IF geen delers gevonden
  THEN put ("priemgetal")
  ELSE put ("geen priemgetal")
  FI.
```

Het programma is af, maar door het beroep op de brute kracht van de computers is het niet erg slim.

Een eerste kritiek op het programma is, dat er veel te veel delers geprobeerd worden: voor een getal `n` altijd `n-2` delers. Een beetje nadenken toont aan dat het voldoende is om slechts die delers te proberen die kleiner zijn dan de wortel uit `n`. Is namelijk `k` een deler groter dan de wortel, dan is ook `n DIV k` een deler, kleiner dan de wortel! Voor grote waarden van `n` maakt het een groot verschil, bijvoorbeeld 100 delers te moeten proberen in plaats van 10000.

```
nog kandidaten te proberen:
  kandidaat * kandidaat <= getal.
```

Een tweede verbetering volgt uit de observatie dat er geen reden meer is de herhaling voort te zetten nadat er een deler gevonden is. In de huidige formulering gaat het proces stug verder tot alle kandidaten geprobeerd zijn. We kunnen deze verbetering bereiken door in de voorwaarde van de herhaling te verlangen dat `geen delers gevonden` waar is.

```
nog kandidaten te proberen:
  kandidaat * kandidaat <= getal AND geen
delers gevonden.
```

We kunnen dan ook de binnenste verfijning wat vereenvoudigen

```
kijk of deling opgaat:
  geen delers gevonden := deling gaat niet
op.
```

De verfijning van `deling gaat niet op` kunnen we realiseren als

```
deling gaat niet op:
  NOT(getal MOD kandidaat = 0).
```

of als

```
deling gaat niet op:
  getal MOD kandidaat <> 0.
```

Merk op dat in de eerste versie van `deling gaat niet op` de haakjes noodzakelijk zijn vanwege de hoge prioriteit van de operator `NOT`. We kiezen de tweede versie.

Gezien de rol die de variabele `geen delers gevonden` speelt bij de snellere terminatie van deze herhaling, noemt men deze vorm een herhaling met een **Boole'se hulpvariabele**.

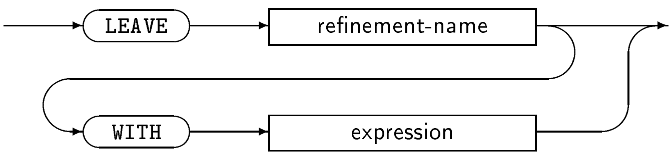
De herhaling kan op twee manieren eindigen: of alle kandidaten zijn geprobeerd en het getal blijkt een priemgetal, of onderweg ging een deling op en de variabele `geen delers gevonden` kreeg de waarde `FALSE`.

5.4 De terminator

Het gebruik van zo'n Boole'se hulpvariabele is een truc om twee vliegen in een klap te slaan. Eigenlijk gaat het hier om een heel fundamenteel probleem. We hebben op een bepaalde plek in de algoritme een bepaalde wetenschap (namelijk dat een deler gevonden is) waar we op die plek (helemaal binnen de algoritme `kijk of deling opgaat`) niet goed raad mee weten. Een andere verfijning `bepaal of het getal priem is` kan op grond van deze wetenschap worden afgebroken.

Elan kent hiervoor een bijzondere constructie, de `LEAVE`-opdracht oftewel **terminator**. Deze constructie

terminator



bewerkt dat de huidige uitvoering wordt afgebroken en in plaats daarvan de genoemde verfijning als het ware in één klap van binnenuit wordt voltooid.

We schrijven daarmee

```
kijk of deling opgaat:
  IF deling gaat op
  THEN geen delers gevonden := FALSE;
       LEAVE bepaal of het getal priem is
  FI.

deling gaat op:
  getal MOD kandidaat = 0.
```

Nu kunnen we nog een verfijning vereenvoudigen:

```
nog kandidaten te proberen:
  kandidaat * kandidaat <= getal.
```

Het resulterende programma is iets efficiënter maar vooral wat doorzichtiger. In talen zonder `LEAVE`-constructie zijn we echter op het gebruik van Boole'se hulpvariabelen aangewezen.

5.5 Voorbeeld: Primaliteit bepalen (2)

Het programma dat we nu verkregen hebben is al een hele verbetering ten opzichte van de vorige versie. Toch is er nog een aantal verbeteringen mogelijk die een flinke besparing in rekentijd kunnen geven. Zo ligt het bijvoorbeeld voor de hand om na de deler 2 alleen nog maar oneven delers te proberen. Dit kunnen we bereiken door, als het getal niet even is, met de deler 3 te beginnen en de volgende kandidaten te berekenen door de deler steeds met 2 te verhogen.

Na de deler 3 hoeven de veelvouden van 3 echter ook niet meer geprobeerd te worden, zodat we, nadat geconstateerd is dat het getal niet deelbaar is door 7, met de deler 11 kunnen beginnen. De volgende kandidaten vinden we door de deler afwisselend met 2 en met 4 te verhogen. We krijgen dan na 2 en 3 als kandidaten voor de delers achtereenvolgens 5, 7, 11, 13, 17, 19, 23, ...

Op deze manier hoeven hoogstens $\lfloor\sqrt{n}/3\rfloor + 2$ delers geprobeerd te worden, in plaats van $n - 2$ delers in de eerste versie. Voor niet te grote waarden van `n`, bijvoorbeeld $n < 10^8$ is dit nog acceptabel.

Het hierna volgende programma gaat volgens bovenstaand idee te werk. Het afwisselend verhogen van de deler met 2 en met 4 wordt verkregen door de deler te verhogen met de waarde van de variabele `increment`, welke na gebruik als nieuwe waarde `6 - increment` krijgt. De variabele `increment` heeft op deze manier achtereenvolgens de waarden 2, 4, 2, 4, 2, ...

We zouden natuurlijk nog verder kunnen gaan en ook veelvouden van grotere priemgetallen als kandidaat-delers buiten beschouwing kunnen laten. De waarden die `increment` achtereenvolgens zou moeten aannemen worden dan moeilijk te berekenen. We laten een dergelijke verbetering graag aan de liefhebbers over.

```
bepaal of het getal priem is:
  BOOL VAR geen delers gevonden :: FALSE;
  IF
    getal MOD 2 = 0 OR getal MOD 3 = 0
  THEN
    het was geen priem
  ELSE
    INT VAR deler :: 5, increment :: 2;
    WHILE nog delers te proberen
    REP
      kijk of deling opgaat;
      neem de volgende kandidaat
    ENDREP
  FI;
  het was een priem.
```

Alle verfijningen zijn als voorheen, behalve:

```
nog delers te proberen:
  deler * deler <= getal.

neem de volgende kandidaat:
  deler INCR increment;
  increment := 6 - increment.
```

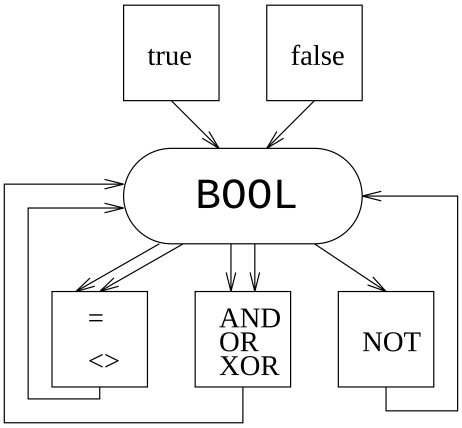


Figure 5.1: NTO voor het type `BOOL`

```
het was geen priem:
  LEAVE bepaal of het getal priem is.

het was een priem:
  geen delers gevonden := TRUE.
```

Het nu verkregen programma is redelijk efficiënt en kan voor niet te grote getallen goede diensten bewijzen.

Voor nog grotere waarden van `n` moeten heel andere methoden te hulp worden geroepen. Vaak zal gebruik gemaakt kunnen worden van een stelling van Fermat: Voor zekere x geldt

$$x^{n-1} \; MOD \; n \;=\; 1 \text{ als } n \text{ priem is.}$$

Kunnen we nu een getal x vinden met

$$x^{n-1} \; MOD \; n \;\neq\; 1$$

dan volgt daar uit dat n geen priemgetal is. Voor veel niet-priemgetallen blijkt dat $x = 3$ al voldoet. Voor voldoende grote waarden van n is echter, ondanks de “brute kracht” van de computer, ook deze methode niet toereikend om in een practische eindige tijd na te gaan of n een priemgetal is.

5.6 NTO voor de booleans

Het hierna volgende NTO vat de verschillende operaties van het type der booleans samen:

- de constanten `true` en `false`
- de dyadische operatoren `AND`, `OR` en `XOR`
- de monadische operator `NOT`
- de vergelijkingsoperatoren `=` en `<>`.

In- en uitvoer-procedures ontbreken.

5.7 Opgaven

Het valt niet mee, met de tot nog toe behandelde taalmiddelen opgaven voor de booleans te verzinnen. Ze hebben nu eenmaal zo'n nederige, dienende taak bij de realisering van voorwaarden. Ook de voorbeelden uit dit hoofdstuk gaan maar nauwelijks over de booleans zelf. Maar vooruit, hier is er een, zij het dat we daarbij weer gebruik maken van het afdrukken van teksten, vooruitlopend op het volgende hoofdstuk:

1. (waarheidstabel) Schrijf een programma dat een waarheidstabel afdrukt van de operatoren AND, OR en XOR. Gebruik daarbij geen van deze operatoren.

5.8 Toevoeging: commentaren

Een **commentaar** is een mededeling in het programma die geen effect heeft voor de uitvoering daarvan. Een commentaar is niet voor de computer bedoeld maar voor de mens die het programma leest.

Commentaren mogen in Elan vrijelijk tussen de units van een paragraaf geschreven worden. Een commentaar begint met een comment-open-symbol en eindigt met een comment-close-symbol. Tussen deze symbolen mogen alle tekens voorkomen die niet zo'n symbool vormen.

De mogelijke representaties van deze symbolen zijn:

comment-open-symbol	{	(*	#
comment-close-symbol	}	*)	#

Aangezien één verloren commentaarsluithaak de rest van het programma binnenste buiten keert (programma wordt commentaar en omgekeerd) is het in Elan niet toegestaan een commentaar over meer dan een regel uit te strekken; wel kunnen meer commentaarregels op elkaar volgen.

Commentaren dienen voor zulke doeleinden als:

- aangeven van naam, versie en auteur van een programma, bijvoorbeeld

```
# GWB versie 2.3 dd 23/10/93, MJN #
```

- geven van korte karakterisering, beperkingen en voorwaarden, bijvoorbeeld:

```
{Oplossing van het vergelijkingssysteem}
{  Ax = b}
{en berekening van de determinant}
{ter controle op de nauwkeurigheid,}
{volgens de methode van Gauss-Jordan.}
{De algoritme is onbruikbaar voor n>20.}
```

- Aangeven van een bewering die op die plaats in de algoritme geldt, ten behoeve van de menselijke lezer en eventueel een bewijs.

```
het maximum van een rij positieve
integers:
  INT VAR max :: 0;
  {max= maximum van een lege rij}
  FOR i FROM 1 UPTO n
  REP
    IF max < rij[i] THEN max := rij[i] FI
    {max= maximum van eerste i elementen}
  ENDREP
  {max= maximum van alle n elementen}.
```

- Absoluut noodzakelijke verklaringen.

```
(*In het belang van de efficiency wordt
  hier niet op overflow getest.)
```

Commentaren dienen er niet toe domme opmerkingen voor het nageslacht vast te leggen, zoals

```
    x := 0;   { x wordt op nul gezet }
```

of

```
    (* nou begint het pas goed! *)
```

Het is altijd beter door het gebruik van verfijningen voor de algoritmen en door het kiezen van zinvolle namen voor de objecten de abstracties functioneel in het programma aan te geven, dan (meestal achteraf) verklaringen te geven, die het programma alsnog begrijpelijk moeten maken.

Een verfijning kan natuurlijk best een misleidende naam hebben, maar dat is dan meestal redelijk snel te zien uit het programma. Een commentaar is echter geen functioneel onderdeel van het programma en geen haan kraait er dus naar als bij wijzigingen in het programma de commentaren niet aangepast worden.

In dit boek wordt dan ook de nadruk op abstractie gelegd en blijkt een uitgebreid gebruik van commentaren overbodig. In een meer industriële omgeving wordt echter grote waarde gehecht aan uitgebreide commentariëring, omdat de beperkingen van de gebruikte programmeertalen het vasthouden van abstracties nauwelijks toestaan, zodat het programma alleen geen voldoende documentatie biedt.

Tot slot nog een wijsheid:

> *"Het achteraf invoegen van commentaren kan een slecht gestructureerd programma niet meer redden en een chaotische ontstaanswijze niet versluieren."*

Een systematisch ontworpen, zorgvuldig verfijnd en goed geverbaliseerd programma heeft geen scheepsladingen commentaren nodig om voor een ander begrijpelijk te zijn.

Chapter 6

Teksten

De algoritmische manupulatie van *teksten* is de sleutel tot een hele wereld van niet-numerieke toepassingen:

- de weergave van de resultaten van berekeningen in appetijtelijke vorm
- vertalers en vertolkers voor programmeertalen
- linguïstisch onderzoek, of ander onderzoek met een linguïstische component
- tekstverwerking bij kranten, bij uitgevers en op kantoren (de fransen spreken van "Burotique")
- allerlei vormen van bedrijfsadministratie.

De laatstgenoemde toepassingen zijn het meest prozaïsch, maar economisch gezien ook het belangrijkst. Met de komst van sprekende en spraakherkennende in- en uitvoerapparatuur zullen de niet-numerieke toepassingen beslist nog in omvang toenemen.

Voor de manipulatie van teksten zijn speciale programmeertalen ontworpen (zoals SNOBOL en allerlei macro-processoren) die een beknopte formulering van ingewikkelde tekstmanipulaties toelaten. De subset Elan-0 levert op dit punt simpele maar toereikende taalmiddelen (zie 6.2). De standaardbibliotheek van Elan kent daarnaast nog een hele verzameling knip- en plakwerktuigen voor de omgang met teksten (zie 6.7 en Appendix B).

6.1 Denotatie van teksten

Onder een **tekst** verstaan we een reeks van nul of meer **tekens**, gekozen uit een bepaalde verzameling van in de computer representeerbare tekens, het **alfabet**. Het type van zo'n tekst is `TEXT`.

Welke tekens in de computer representeerbaar zijn hangt af van de gebruikte implementering, maar dit alfabet omvat in ieder geval de tekens waarmee Elan-programma's geschreven worden (kleine en grote letters, cijfers, leestekens, operator-tekens, spaties etc.). Een tekst wordt gedenoteerd door hem tussen aanhalingstekens *(quotes)* te zetten, bijvoorbeeld:

```
"dit is een tekst"
```

De waarde van de denotatie is de rij van tekens die je krijgt door de buitenste aanhalingstekens weg te laten. Om binnen een tekst ook een aanhalingsteken te kunnen representeren is afgesproken dat daarin optredende aanhalingstekens worden verdubbeld (een typische informaticatruc die vaker van pas komt).

Voorbeeld:

```
"""Stilte!"" sprak Gandalf, ""Hoort
Thorin's rede""."
```

Met name wordt de tekst, bestaande uit een enkel aanhalingsteken, als volgt gedenoteerd:

```
""""
```

De **lege tekst** (een rij van nul tekens) wordt gedenoteerd als `""`. Op Uw computer wordt de spatie weergegeven door de spatiebalk resp. een lege positie op het scherm. Om verwarring te voorkomen zullen we op enkele plaatsen in dit hoofdstuk de spatie binnen een tekst-denotatie met het teken ␣ aangeven, zodat het wat makkelijker wordt te tellen hoeveel spaties er bedoeld zijn in

```
"␣␣␣␣␣"
```

Hoewel in principe teksten van willekeurige lengte gedenoteerd kunnen worden kan een bepaalde implementatie een bovengrens aan de lengte van representeerbare teksten stellen. Zo begrenst de Elan-compiler in het EUMEL-systeem de lengte tot 255 (op te vragen als de `INT`-constante `max textlength`). Grotere teksten moeten daar als rijen van regels worden behandeld.

6.2 Tekst operaties

Voor het rekenen met teksten hebben we in Elan een aantal standaard operaties. Helaas is slechts een klein deel daarvan ook binnen Elan-0 ter beschikking — dit is een van de plaatsen waar op de taal besnoeid moest worden. In plaats daarvan bevat Elan-0 een simpeler stel operaties. Wij behandelen hier eerst de gemeenschappelijke operaties en daarna die van Elan-0. Voor de overige operaties wordt u verwezen naar hoofdstuk 6.7.

6.2.1 De algemene tekstoperaties

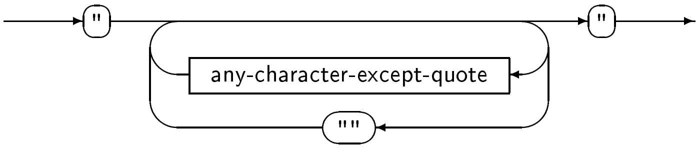
De operator `+` rijgt twee teksten aan elkaar (**concatenatie**):

```
"abc" + "def" = "abcdef"
```

De operator `LENGTH` levert het aantal tekens in de tekst, een integer groter dan of gelijk aan nul.

```
LENGTH "abc" = 3
```

text-denotation




```
LENGTH """" = 1
```

De operator `*` met een niet-negatieve integer `n` als linker operand en een tekst als rechter levert de `n`-malige concatenatie daarvan:

```
3 * "abc" = "abcabcabc"
```

De tekens van een tekst `x` kan men zich voorstellen als genummerd vanaf 1. Om het `k`-de teken uit de tekst te verkrijgen beschikken we over de operator `SUB` (linkeroperand een tekst `x`, rechteroperand een integer `k`, resultaat een tekst bestaande uit één teken, namelijk het `k`-de teken van `x`). Voorbeeld:

```
("abc" SUB 2) = "b"
```

Is het tweede argument van `SUB` kleiner dan één of groter dan het aantal tekens in het eerste argument dan is het resultaat een lege tekst, bijvoorbeeld in:

```
"abc" SUB 37 = ""
```

De dyadische operator `CAT` (met links een tekstvariabele en rechts een tekst) combineert een concatenatie met een toewijzing.

```
t CAT "abc"
```

betekent hetzelfde als

```
t := t + "abc"
```

De operator `SUB` heeft een heel lage prioriteit, zodat uitdrukkingen met `SUB` erin in vele contexten tussen haakjes geschreven moeten worden. Laten we de haakjes weg dan blijkt bijvoorbeeld dat

```
"abc" SUB 2 = "b"
```

betekent:

```
"abc" SUB (2 = "b")
```

wat de Elan-implementering met een wat cryptische foutmelding afwijst.

Het verdient daarom aanbeveling, in uitdrukkingen waar de operator `SUB` voorkomt liever te veel dan te weinig haakjes te zetten om verkeerde duidingen te voorkomen, bijvoorbeeld

```
"abc" SUB (i + 1)
```

6.2.2 Vergelijken van teksten

De vergelijkingsoperatoren voor teksten zijn dezelfde als die voor getallen:

```
=     <=          >     >=     <>
```

De bedoelde ordeningsrelatie is de alfabetisch lexicografische, d.w.z.

```
"" < "a"
"a" < "b"
"a" < "ab"
"aa" < "b"
"aa" < "ab"
```

enzovoorts.

Onthoud dat deze ordening als eigenschap heeft dat als een tekst `x` overeenkomt met het beginsegment van een tekst `y` (dus `y` bestaat uit `x` eventueel gevolgd door nog wat) in ieder geval geldt `x <= y`.

Ieder teken heeft een **code**, een klein geheel getal (meestal < 128 of < 256) als interne representatie. De individuele tekens van een tekst worden vergeleken op basis van hun codes. Aangezien verschillende implementeringen hun eigen code mogen gebruiken is buiten de volgorde van de letters en de volgorde van de cijfers niet veel vastgelegd (komt de spatie voor of na de kleine "a"? Komt het dollarteken "$" voor of na de apestaart "@"? Op dit soort vragen is geen algemeen antwoord te geven, dat hangt van de gebruikte code af). Het meest verbreid zijn de codes

- EBCDIC (Extended Binary-Coded Decimal Interchange Code) op de grotere IBM machines, en
- ASCII (American Standard Code for Information Interchange) op bijna alle andere.

maar de situatie wordt gecompliceerd doordat er naast ASCII een aantal nationale varianten in omloop is (volgens de ISO-standaard toegelaten) met zulke tekens als ä, ö en ç erin. In de praktijk leidt dit overigens nauwelijks tot problemen want gekke dingen laat je wel uit je hoofd.

6.2.3 Deelteksten in Elan-0

De deeltaal Elan-0 kent een tweetal eigen algoritmen voor het werken met deelteksten (de volledige taal Elan kent er een hele verzameling, zoals aangegeven in 6.7).

De monadische operator `HEAD` levert een tekst bestaande uit het eerste teken van zijn operand, bijvoorbeeld:

```
HEAD "abc" = "a"
```

De `HEAD` van een lege tekst is bij afspraak leeg, dus

```
HEAD "" = ""
```

en de `HEAD` van een tekst met lengte 1 is diezelfde tekst. De monadische operator `TAIL` levert als resultaat een tekst gelijk aan zijn operand na verwijdering van het eerste teken, dus

```
TAIL "abc" = "bc"
```

De `TAIL` van een lege tekst is weer leeg, evenals de `TAIL` van een tekst van lengte 1. In alle gevallen geldt

```
HEAD s + TAIL s = s
```

en

```
HEAD s = (s SUB 1)
```

6.3 Invoer en uitvoer van teksten

Een tekst kan op het beeldscherm worden geschreven met de algoritme `put`. Raakt daarbij de uitvoerregel vol, dan wordt vanzelf op een nieuwe regel overgegaan. Hierdoor kan het met de layout nooit helemaal uit de hand lopen, maar het verdient toch de voorkeur zelf met `line` tijdig op een nieuwe regel over te gaan. Voorbeeld:

```
line; put ("Som = "); put (som); line
```

De oproep van `line` gevolgd door `put` kan weer worden samengevat tot een oproep van `put line`:

```
put line ("Som = "); put (som); line
```

De oproep `line (n)` met $n \geq 0$ geeft n keer een nieuwe regel, oftewel $n - 1$ blanco regels. Overgang op een nieuwe pagina op papier, dan wel een schoon beeldscherm, wordt bereikt door `page` op te roepen.

De algoritme `get`, opgeroepen met een `TEXT`-variabele als parameter, leest invoer tot aan het het einde van de regel. Is de regel leeg dan wordt de volgende niet lege regel genomen. De resulterende tekst wordt aan zijn parameter toegewezen. Voorbeeld:

```
TEXT VAR regel; get (regel)
```

Na afloop zal de waarde van `regel` een niet lege tekst zijn.

6.4 Voorbeeld: ruitjespapier maken

We willen papier bedrukken met nogal grote ruitjes, volgens het patroon:

```
+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |     |     
|
|     |     |     |     |     |     |     
|
+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |     |     
|
|     |     |     |     |     |     |     
|
+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |     |     
|
|     |     |     |     |     |     |     
|
+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |     |     
|
|     |     |     |     |     |     |     
|
+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |     |     
|
|     |     |     |     |     |     |     
|
+-----+-----+-----+-----+-----+-----+-----+
```

We programmeren:

```
teken ruitjespapier:
  INT VAR rijnummer :: 1;
  REP
    teken rijtje hokjes;
    rijnummer INCR 1
  UNTIL
    rijnummer = 6
  ENDREP;
  teken horizontale streep.

teken rijtje hokjes:
  teken horizontale streep;
  verticaal aanhalen;
  verticaal aanhalen.

teken horizontale streep:
  put (7*(kruis + 5*horizontaal) + kruis);
  line.

verticaal aanhalen:
  put (8*(verticaal + 5*blank) +
verticaal);
  line.

horizontaal:   "-".
verticaal:     "|".
kruis:         "+".
blank:         "␣".
```

Merk op, dat we in plaats van de verfijningen `horizontaal`, `verticaal`, `kruis` en `blank` voor hetzelfde geld constanten hadden kunnen declareren in plaats van (de hier gekozen) verfijningen. Verfijningen die een waarde leveren zijn wel vaker een alternatief voor constanten.

Merk ook op dat de **prioriteit** van de operatoren `+` en `*` voor teksten dezelfde is als die voor bijvoorbeeld integers. De prioriteit is verbonden met de operator en niet afhankelijk van het type van de operanden.

6.5 Voorbeeld: conversie van integers

We construeren een programma dat twee positieve (decimale) gehele getallen inleest en het eerste getal uitdrukt in het talstelsel met het tweede getal als grondtal.

Wij zijn gewend aan het tientallig stelsel, met 10 als grondtal: getallen worden decimaal voorgesteld als een gewogen som van tienmachten met de cijfers van het getal als gewicht. Zo geldt tientallig $4711 = 4 * 10^3 + 7 * 10^2 + 1 * 10^1 + 1 * 10^0$ Ieder van de cijfers is een positief geheel getal kleiner dan het grondtal (dus $0 \leq cijfer < grondtal$).

Dit systeem kunnen we generaliseren voor andere grondtalwaarden dan 10; zo krijgen we voor grondtal 2 tientallig $23 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$ = tweetallig 10111 We kunnen ook grotere grondtallen dan 10 nemen, waarbij we de dan optredende cijfers tien, elf ... zullen aangeven met A, B ... .

De wortel van dit programma kan luiden:

```
grondtal conversie:
  vraag om het grondtal;
  WHILE
    er is nog een getal te converteren
  REP
    bepaal de voorstelling van het getal
        bij dit grondtal;
    druk de voorstelling af
  ENDREP.

vraag om het grondtal:
  put line ("Het grondtal, alstublieft:");
  INT VAR grondtal;
  get (grondtal).

er is nog een getal te converteren:
  put line ("Een getal, alstublieft:");
  INT VAR getal;
  get (getal);
  getal > 0.
```

De voorstelling van het getal bij dit grondtal bepalen we door, van achter naar voren, cijfers in het grondtaltallig stelsel van het getal af te splitsen.

```
bepaal de voorstelling van het getal
    bij dit grondtal:
  TEXT VAR voorstelling :: "";
  REP
    splits rechts een cijfer af;
    voeg het aan de voorstelling toe
  UNTIL
    geen cijfers meer over
  ENDREP.

splits rechts een cijfer af:
  INT CONST cijfer :: getal MOD grondtal;
  getal := getal DIV grondtal.
```

Wij plakken een tekst, corresponderend met het afgesplitste cijfer, voor aan de voorstelling.

```
voeg het aan de voorstelling toe:
  voorstelling :=
     representatie van het cijfer
        + voorstelling.
```

We zullen voor de cijfers tot vijftien een representatie invoeren, hogere cijfers worden als een vraagteken afgedrukt.

```
representatie van het cijfer:
  IF
    cijfer > 15
  THEN
    "?"
  ELSE
    "0123456789ABCDEF" SUB (cijfer + 1)
  FI.
```

We moeten bij de waarde van het `cijfer` één optellen, omdat de nummering van de tekens in een tekst niet bij nul maar bij één begint.

```
geen cijfers meer over:
  getal = 0.

druk de voorstelling af:
  line(2);
  put ("Voorstelling is ");
  put (voorstelling);
  line(2).
```

Het voorbeeld is daarom zo leerzaam omdat het de relatie tussen teksten (die cijfers mogen bevatten) en getallen (die uit cijfers bestaan) inzichtelijk maakt. Het wordt de lezer aanbevolen voor enkele getallen en grondtallen het proces stapsgewijs te volgen.

6.6 Voorbeeld: rondschuiven van teksten

Tot slot nog een klein voorbeeldje dat het gebruik van de operatoren `HEAD` en `TAIL` verduidelijken moet. Onder het "circulair naar links schuiven" van een rij verstaan we een cyclische verwisseling als in

```
A B C D E
B C D E A
C D E A B
...
```

waarbij ieder element een plaats naar links verschuift behalve het eerste, dat helemaal rechts belandt.

We willen een programma construeren dat de tekens van een tekst telkens circulair een plaats naar links schuift totdat de originele tekst weer hersteld is.

```
rondschuiven:
  declareer teksten;
  REP
    verschuif s circulair naar links;
    toon s
  UNTIL
    s = origineel
  ENDREP.

declareer teksten:
  TEXT VAR s;
  get (s);
  TEXT CONST origineel :: s.

verschuif s circulair plaats naar links:
  s := TAIL s + HEAD s.

toon s:
  put line (s).
```

6.7 Deelteksten in Elan

Het volle Elan kent voor het knippen en plakken van **deelteksten** naast `+`, `SUB`, `CAT` en `LENGTH` nog een hele verzameling bijzondere algoritmen. Deze maken het werken met teksten een stuk eenvoudiger dan in Elan-0, hoewel ook moet worden vastgesteld dat geen van deze algoritmen strikt noodzakelijk is: met de operatoren `+` en `SUB` kan met enige moeite reeds ieder effect worden bereikt. Bij vele toepassingen is het echter wel gemakkelijk wat machtiger algoritmen te gebruiken, zoals beschreven in de volgende secties.

6.7.1 Standaard constanten

In de standaard paketten is een aantal `TEXT`-constanten voorgedefinieerd:

```
TEXT CONST niltext :: "", blank :: "␣",
           quote :: """"
```

Deze kunnen dus zonder verdere declaratie gebruikt worden.

6.7.2 Deelteksten: de algoritme `text`

Uit een tekst `t` kan een tekst met een lengte `len` worden afgeleid (door afkappen als `len < LENGTH t` en door rechts aanvullen met spaties als `len > LENGTH t`) met de algoritme

```
text (t, len)
```

In de hier volgende voorbeelden nemen we aan dat `t = "abc"`.

```
text (t, 2) = "ab"
text (t, 5) = "abc␣␣"
```

Hierbij kunnen we ook het gedeelte links van een bepaalde positie `p` weglaten met behulp van een oproep van de vorm

```
text (t, len, p)
```

Betekenis: een tekst van lengte `len` gevormd uit de tekst `t`, te beginnen bij het element met index `p`. Voorbeelden:

```
text (t, 2, 2) = "bc"
text (t, 5, 2) = "bc␣␣␣"
```

Er geldt algemeen:

```
text (t, 1, p) = (t SUB p)
text (t, len, 1) = text (t, len)
```

Een ingewikkelder voorbeeld (cyclische verwisseling):

```
text (t, LENGTHt -1, 2) + (tSUB1) = "bca"
```

Strikt genomen is hier sprake van twee algoritmen, gedefinieerd in de standaardomgeving, beide met de naam `text`, maar verschillend in het aantal en/of type van hun parameters. Later zullen we nog meer algoritmen met diezelfde naam tegenkomen. Ze hebben gemeenschappelijk dat ze uit "iets" een tekst brouwen — zoals de naam poogt te suggereren.

Men noemt zo'n familie van algoritmen met dezelfde naam een **generische algoritmen**. Ook `get`, `+` zijn daarvan voorbeelden.

6.7.3 Deelteksten: de algoritme `subtext`

De generische algoritme `subtext` is een handige schaar. Zijn eenvoudigste vorm

```
subtext (t, van)
```

levert de tekst die men uit `t` krijgt door pas bij het `van`-de teken te beginnen:

```
subtext (t, 2) = "bc"
```

Er geldt algemeen

```
subtext (t, i) = text (t, LENGTHt -i +1,
i)
```

De oproep

```
subtext (t, van, tot)
```

levert de subtext uit `t` te beginnen met positie `van` en eindigend met positie `tot`, bijvoorbeeld:

```
subtext (t, 1, 2) = "ab"
```

Is `van > tot` dan is het resultaat de lege tekst:

```
subtext (t, 2, 1) = ""
```

Er geldt algemeen

```
subtext (t, p, p) = (t SUB p)
```

Als voorbeeld formuleren we weer de cyclische verwisseling:

```
subtext (t, 2) + (t SUB 1) = "bca"
```

Ook `subtext` is een generische algoritme, want hij heeft meer dan een definitie met dezelfde naam maar verschillende parametrisering.

6.7.4 Zoeken in een tekst: de algoritme `pos`

Het is mogelijk in een tekst `t` te zoeken naar een tekst `x` met de oproep

```
pos (t, x)
```

Deze oproep levert de index van het meest linkse voorkomen van `x` in `t` als `x` daarin voorkomt, en anders nul. Voorbeeld (zij `t = "a"`:

```
pos (t, "b")  = 0
pos (t, "ab") = 1
pos (t, "ba") = 2
```

Het generische broertje

```
pos (t, x, van)
```

levert de index van het eerste voorkomen van `x` in `t` vanaf de positie `van` (en nul als `x` niet vanaf positie `van` voorkomt). Zo kan men het tweede voorkomen van `x` in `t` vinden met de oproep

```
pos (t, x, pos (t, x) + 1)
```

6.7.5 Vervangen van een deeltekst: de algoritme `change`

Van een tekst kan men een deeltekst vervangen door een andere tekst door een oproep van

```
change (tekstvar, deeltekst, andere tekst)
```

waarbij het meest linkse voorkomen van `deeltekst` in de `tekstvariabele` wordt vervangen door de `andere tekst`. Komt de `deeltekst` in de `tekstvariabele` niet voor dan gebeurt er niets. Omdat de lengten van `deeltekst` en `andere tekst` kunnen verschillen is het mogelijk dat hierbij de lengte van de waarde van de `tekstvariabele` gewijzigd wordt.

Zo kan men een tekst te lijf gaan met

```
WHILE
  pos (mijn tekst, "jan") > 0
REP
  change (mijn tekst, "jan", "johannes")
ENDREP
```

6.7.6 Lezen en schrijven: de algoritmen `get` en `put`

Een tekst kan weer worden geschreven met de **uitvoer**-algoritme `put`. Raakt daarbij de uitvoerregel vol, dan wordt vanzelf op een nieuwe regel overgegaan, net als in Elan-0.

Voor het lezen van een tekst staan in de volle taal verschillende algoritmen ter beschikking. Zij `x` een `TEXT`-variabele. De eenvoudigste **invoer**-algoritme

```
get (x)
```

heeft het volgende effect:

- spaties en overgangen op nieuwe regel in de invoer worden overgeslagen tot een niet-spatie is gevonden
- vanaf de huidige positie tot aan de eerstvolgende spatie of regeleinde worden tekens gelezen en tot een tekst samengevoegd
- deze tekst wordt aan de variable `x` toegewezen.

Het verschil met de `get` van Elan-0 is dus dat ook spaties als scheiders tussen teksten worden beschouwd.

Merk op dat de zo gelezen tekst geen spaties kan bevatten en dat regelgrenzen van de invoer vrijwel onzichtbaar zijn. Deze vorm van invoer is blijkbaar vooral geschikt voor min of meer linguïstische toepassingen.

Een generische variant hierop met een geheel getal als tweede parameter

```
get (x, length)
```

kent aan `x` een tekst van hoogstens `length` tekens toe, gelezen vanaf de huidige positie zonder spaties aan het begin over te slaan. Er wordt niet over het regeleinde heengelezen. Een lege regel wordt dus overgeslagen. Wil men in Elan hetzelfde bereiken als met `get` in Elan-0 dan moet men deze laatste versie gebruiken met `length` gelijk gekozen aan de regellengte van de invoer. Kiest men `length` gelijk aan 1 dan wordt er één teken gelezen.

6.7.7 Conversie-algoritmen: de algoritmen `text` en `int`

Ook voor het type der teksten bestaat er een aantal **conversie**-algoritmen die waarden van andere typen omzetten in teksten en omgekeerd.

Als eerste is te noemen de procedure `text` (let op de kleine letters) die een `REAL` of `INT` waarde omzet in een tekst die sprekend lijkt op een denotatie daarvoor.

Een voorbeeld:

```
TEXT VAR t :: text (123)
```

De waarde van het getal 123 wordt omgezet in de tekst bestaande uit de tekens `"1"`, `"2"` en `"3"`, in die volgorde, dus `"123"`. Met zijn generische broertje kun je ook opgeven hoe lang de opgeleverde tekst moet zijn; zo zal in

```
TEXT VAR t :: text (123, 7)
```

de tekstvariabele `t` de waarde ”␣␣␣␣123” krijgen.

In de omgekeerde (richting `TEXT` naar `INT`) gebruik je de procedure `int` met als parameter een tekst die lijkt op een integer-denotatie, eventueel voorafgegaan door een voorteken `"+"` of `"-"`. Zo drukt

```
put (7 + int (t))
```

nu het getal 130 af.

Met de procedure `real` kunnen we teksten die op real-denotaties lijken omzetten naar `REAL` waarden:

```
REAL VAR x :: real(t + ".0")
```

wijst aan `x` de waarde 123.0 toe.

De procedure `text` kan ook met een real parameter worden aangeroepen en levert dan een tekst die sprekend lijkt op een denotatie daarvan in “drijvende komma vorm”. Zo levert

```
text (0.1234)
```

de waarde `"1.2340000E-1"` of zo.

De conversie kan (vooral uitgaande van een tekst) vrij makkelijk mislukken; per slot van rekening hoeft er daartoe maar een verkeerd teken in de tekst te staan, zoals bijvoorbeeld in

```
int ("123a")
```

Om na te gaan of de zojuist uitgevoerde conversie succesvol is verlopen, bestaat er een boole'se standaard-procedure `last conversion ok`, die gebruikt kan worden om een eigen wijze van foutbehandeling te realiseren

```
IF NOT last conversion ok
THEN put ("conversie mislukt");
     plof
FI
```

6.8 Voorbeeld: een zakrekenmachine

In dit voorbeeld zullen we pogen een zakrekenmachine na te doen van het soort dat elk warenhuis voor afbraakprijzen aanbiedt. Gemakshalve beperken we ons tot de gehele getallen en hun vier grond-operaties.

De rekenmachine voert met ons een soort dialoog. Als hij gereed is om te rekenen geeft hij het dollarteken (bij wijze van "prompt"). De invoer bestaat telkens uit een uitdrukking gevolgd door een gelijkheidsteken (=). Als antwoord drukt de machine de waarde van de uitdrukking af gevolgd door weer een dollarteken om aan te geven dat invoer verlangd wordt. Willen we met rekenen ophouden dan geven we de tekst `halt` in. (Aan de verkoop van zo'n onding zullen we niet rijk worden, maar de lezer mag uiteraard proberen een wat realistischer calculator te realiseren.)

De dialoog tussen rekenmachine en gebruiker kan er bijvoorbeeld zo uit zien:

```
$4*3=
           12
$5+8=
           13
$halt
```

We programmeren deze opgave in standaard Elan. Eerst realiseren we de dialoog.

```
dialoog:
  vraag eerste regel;
  WHILE verdere berekening gewenst
  REP
    druk waarde formule;
    vraag volgende regel
  ENDREP.

verdere berekening gewenst:
  regel <> "halt".

vraag eerste regel:
  TEXT VAR regel;
  vraag volgende regel.

vraag volgende regel:
  line (2);
  put ("$");
  get (regel, max linelength).
```

De verdere uitwerking van dit programma baseren we op enkele syntax-diagrammen, die het verschil in prioriteit tussen de operaties uitdrukken.

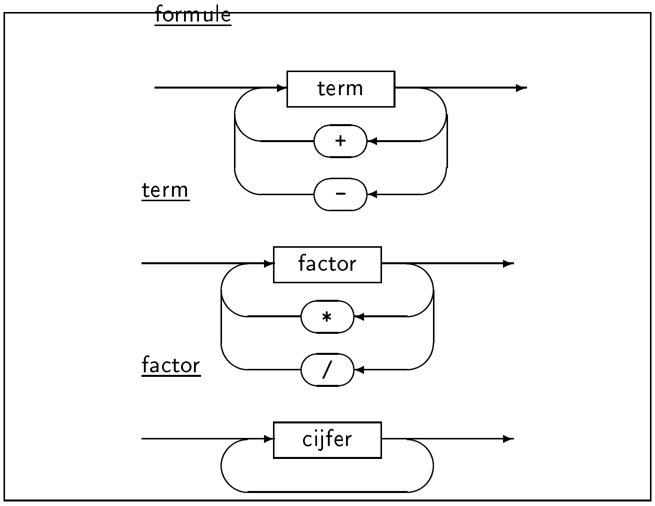


Figure 6.1: Syntaxdiagrammen voor de zakrekenmachine

Voor ieder van deze drie syntactische begrippen introduceren we een verfijning die poogt dit begrip te herkennen, en als waarde het resultaat van de corresponderende uitdrukking levert.

```
formule:
 INT VAR resultaat formule :: term;
 WHILE operator = "+" OR operator = "-"
 REP
   IF operator = "+"
   THEN resultaat formule INCR term
   ELSE resultaat formule DECR term
   FI
 ENDREP;
 resultaat formule.


term:
 INT VAR resultaat term :: factor;
 WHILE operator = "*" OR operator = "/"
 REP
   IF operator = "*"
   THEN resultaat term :=
          resultaat term * factor
   ELSE resultaat term :=
          resultaat term DIV factor
   FI
 ENDREP;
 resultaat term.
```

```
factor:
 INT VAR getal; get (getal);
 IF NOT last conversion ok THEN fout FI;
 TEXT VAR operator;
 REP
    get (operator, 1)
 UNTIL
    pos ("+-*/=", operator) > 0
 ENDREP;
 getal.
```

Wij maken hier expliciet gebruik van het (uit de diagrammen af te leiden) feit dat een getal steeds door een operator wordt gevolgd. Eventuele spaties worden door `get` overlezen.

We voegen toe

```
druk waarde formule:
  INT CONST resultaat :: formule;
  IF operator = "="
  THEN put (resultaat)
  ELSE fout
  FI.

fout:
  put line ("foutieve uitdrukking");
  LEAVE druk waarde formule.
```

Het voorbeeld is, in al zijn kortheid, niet eenvoudig. Het gebruik van syntax-diagrammen als leidraad bij het programmeren (of nog beter: syntaxregels zoals behandeld worden in hoofdstuk 11) is een zeer machtige techniek, waarbij echter wel enige grammaticale kennis komt kijken.

De gegeven oplossing laat zich niet gemakkelijk overzetten naar Elan-0 vanwege het ontbreken van `pos`, alsmede het verschil in werking van `get` met een tekstvariabele als parameter. We moeten de invoer dus regelsgewijs inlezen en tekengewijs verwerken. De realisering van deze opgave in Elan-0 zou daarom heel anders verlopen.

6.9 Samenvatting

Bij het type `TEXT` behoren zoveel operaties, dat het ondoenlijk is ze overzichtelijk in één NTO weer te geven. Daarom beschrijven we eerst het netwerk voor de eenvoudige operaties, en daarna dat van de andere operaties. Tesamen geven ze een goed overzicht. Voor meer details verwijzen we weer naar Appendix B.

6.10 Opgaven

1. (palindromen) Een “palindroom” is een tekst die van links naar rechts gelezen dezelfde is als van rechts naar links gelezen. Om interessante palindromen te krijgen worden naast de letters ook spaties en leestekens toegelaten maar die spelen bij de vergelijking geen rol. Een palindroom uit de vorige eeuw luidt

   `"a man, a plan, a canal: panama"`

   Schrijf een algoritme die een regel tekst inleest en bericht of de daarin voorkomende letters een palindroom vormen.
2. (geheimschrift) In een eenvoudige vorm van geheimschrift worden alle letters en de spatie een bepaald aantal plaatsen rondgeschoven. Schrijf een programma'tje dat een tekst inleest en met circulaire verschuiving 0, 1, ... 25 afdrukt. Gebruik dat programma om het volgende (gedeeltelijk verminkte) bericht te ontcijferen:

   `"swvqvouqliapmmnbmbmtawzmu"`
3. (arabisch naar romeins) Schrijf een programma dat een (ingelezen) geheel getal in romeinse cijfers uitschrijft.
4. (romeins naar arabisch) Schrijf een programma dat een getal in romeinse cijfers inleest en in het tientallig stelsel uitschrijft.
5. (verkleinwoorden) Schrijf een programma dat van een ingelezen zelfstandig naamwoord volgens de regels van “van Dale” een verkleinwoord vormt.
6. (latijnse grammatica) Schrijf een programma dat latijnse zelfstandige naamwoorden kan verbuigen.
7. (Calculator in Elan-0) Realiseer de rekenmachine uit het laatste voorbeeld nu in Elan-0.
8. (plaatskaartenautomaat) Programmeer een eenvoudige loketrobot, die in een eindeloze herhaling
   (a) vertelt wat de beschikbare kaarten kosten
   (b) de gelegenheid geeft een keus te maken
   (c) om geld vraagt tot er genoeg betaald is
   (d) het eventuele overschot terugbetaalt, en
   (e) een plaatskaart afdrukt.

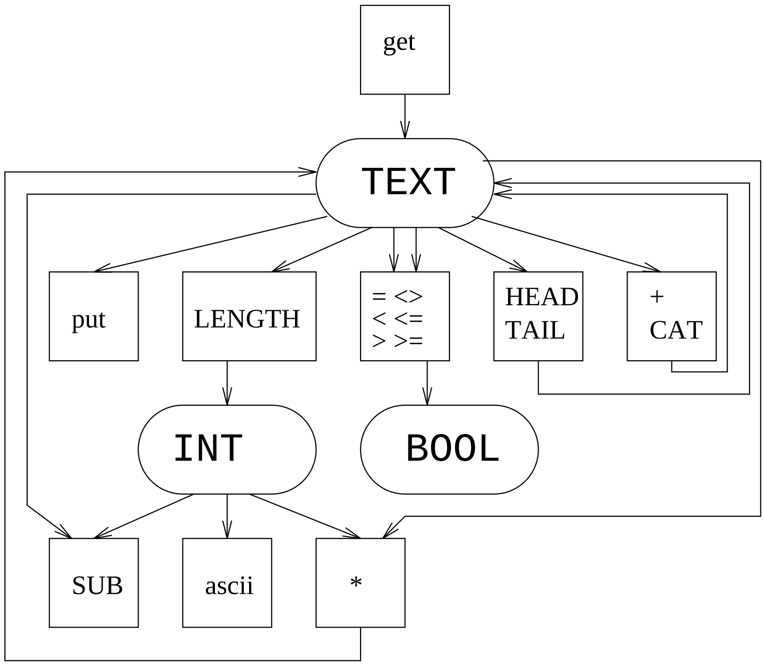


Figure 6.2: Tekstoperaties op Elan-0-Niveau

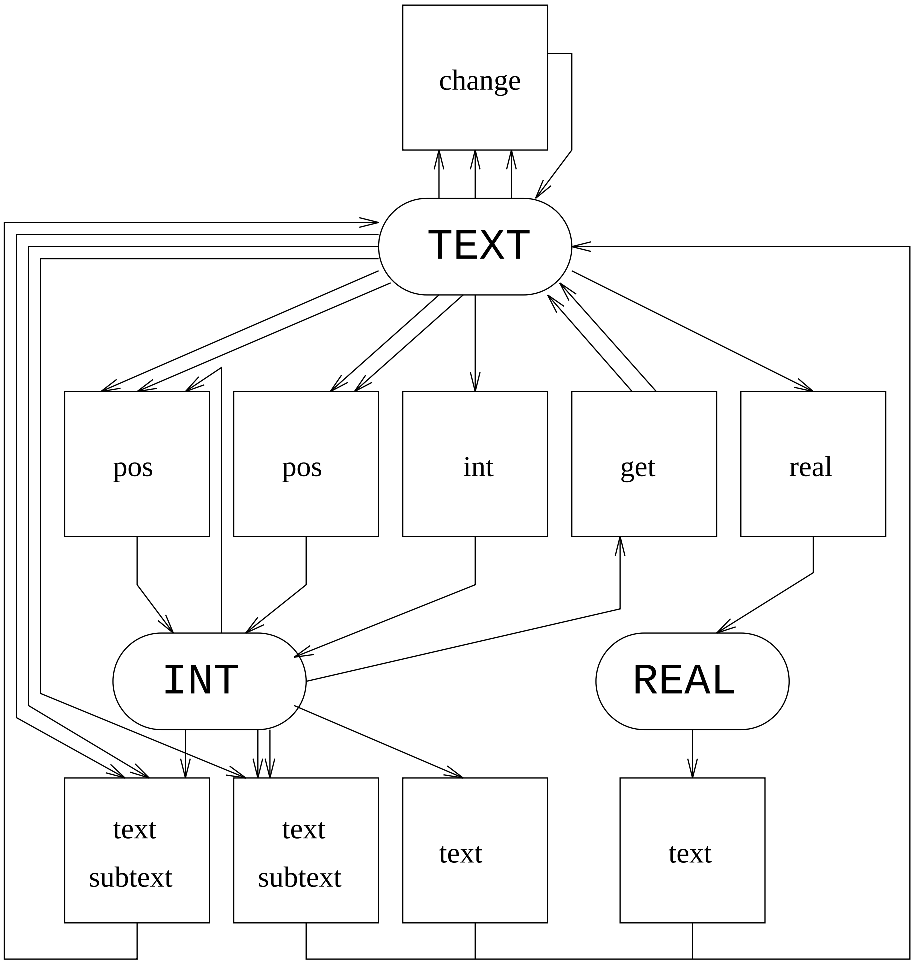


Figure 6.3: Tekstoperaties op Elan-1-Niveau

Chapter 7

Controlstructuren

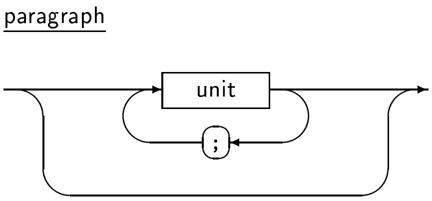
De Controlstructuren behoren tot de stijlmiddelen van de programmeertaal en bepalen voor een groot deel het gezicht van die taal. Zij dienen voor de constructie van samengestelde algoritmen, net zoals gegevensstructuren middelen ter constructie van samengestelde objecten zijn. Syntactisch gezien geven controlstructuren aan hoe een aantal algoritmen kan worden samengesteld tot één algoritme die weer zelf als bouwsteen kan optreden. Zij zijn dus de cement van de programmeertaal.

Ook bepalen de controlstructuren hoe de betekenis (de **semantiek**) van de samengestelde algoritme kan worden uitgedrukt in termen van de betekenis van zijn deelalgoritmen, d.w.z. hoe zijn **effect**, **resultaat** en **type** worden uitgedrukt in termen van de effecten, resultaten en typen van deze deelalgoritmen.

Zij sturen ("to control") de uitvoering van het programma en ontlenen aan dit laatste hun naam. De controlstructuren van Elan hebben we grotendeels al behandeld. In dit hoofdstuk behandelen we hun syntax en semantiek wat preciezer.

7.1 Paragrafen en units

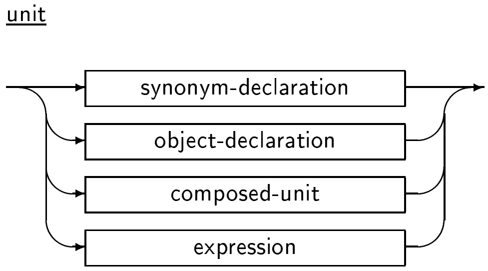
De meest fundamentele controlstructuur van algoritmische talen is de **opeenvolging**, in Elan gerealiseerd door de paragraaf.



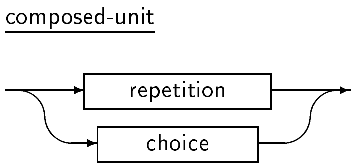
Een paragraaf bestaat uit een aantal **units**, van elkaar gescheiden door puntkomma's. Merk op dat een paragraaf ook leeg kan zijn.

Voor het engelse woord "unit" (evenals voor het hybride woord "controlstructuur") is het de auteur niet gelukt een algemeen aanvaardbare nederlandse vertaling te verzinnen. Allerlei alternatieve benamingen ("eenheid", "zin", "frase", "stap", "joenit"; "besturingsstructuur", "stuurstructuur") klinken nog verschrikkelijker.

Een unit kan elementair zijn, of op zijn beurt weer samengesteld zijn: namelijk een controlstructuur, een **uitdrukking** ("expression") of een object declaratie. In werkelijkheid (Appendix A) is de syntax nog wat



ingewikkelder, maar deze diagrammen geven voorlopig voldoende houvast. Merk op dat elke *unit* in een para-



graaf zelf weer een *composed-unit* kan zijn.

7.1.1 Effect en resultaat van een paragraaf

De **betekenis** van een paragraaf (zijn effect, resultaat en type) kan worden afgeleid uit de betekenis van zijn delen.

De uitvoering van een paragraaf bestaat uit de uitvoering in de gegeven volgorde van zijn achtereenvolgende units. Daarom bestaat het **effect** van die uitvoering ook uit het serieel samengestelde effect van zijn deelalgoritmen.

Het **resultaat** van een paragraaf die op een uitdrukking eindigt is het resultaat van de uitvoering van die uitdrukking; de overige paragrafen zijn acties en leveren geen waarde. Een paragraaf die op een actie eindigt levert geen resultaat.

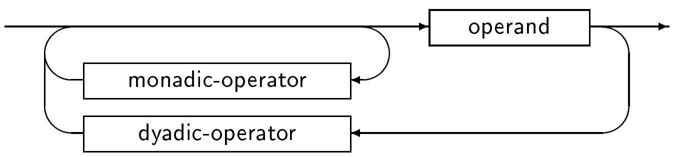
Het type van een paragraaf wordt bepaald door zijn laatste unit. Een actie heeft het hypothetische type `VOID`.

7.2 Uitdrukkingen

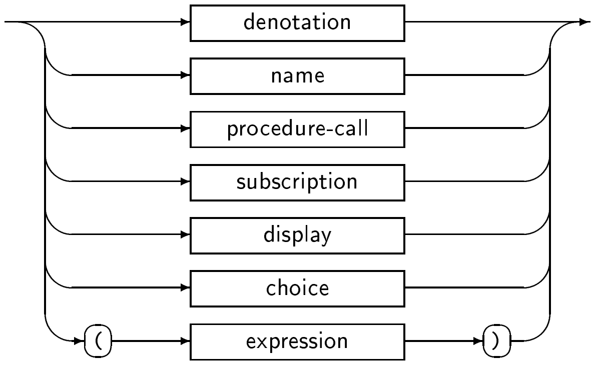
Een **uitdrukking** bestaat uit **operanden** en **operatoren**, met eventueel haakjes om ze te groeperen. De belangrijkste van de mogelijke operanden zijn reeds behandeld, de overige volgen nog.

7.2.1 Prioriteit van operatoren

Merk op dat het hierboven gegeven diagram voor expression meerduidig is voor uitdrukkingen met meer expression



operand



dan één dyadische operator. Om aan uitdrukkingen hun (eenduidige) betekenis te geven moeten we daarnaast rekening houden met de prioriteiten van de operatoren. De **operator prioriteiten** zijn af te leiden uit de volgende tabel.

1. De toewijzing en de operatoren gecombineerd met toewijzing
   `:=` `INCR` `DECR` `CAT`
2. Alle abstracte dyadische operatoren evenals `SUB`
3. `OR` `XOR`
4. `AND`
5. `=` `<>`
6. `<=` `<` `>=` `>`
7. `+` `-`
8. `*` `/` `DIV` `MOD`
9. `**`
10. Alle monadische operatoren, zoals
    `+` `-` `LENGTH` `ABS` `SIGN`

De operatoren met de hoogste prioriteit binden het sterkst. Bij gelijke prioriteit wordt van links naar rechts gebonden. Zo betekent

```
2.0 + x / 2.0 / a + 3.0 - c
```

hetzelfde als

```
(((2.0 + ((x / 2.0) / a)) + 3.0) - c)
```

De operanden van de zwakst bindende operator worden het eerst uitgerekend, daarna wordt de operator op hun waarden toegepast. Deze operanden kunnen op hun beurt weer expressies zijn, waarvoor hetzelfde geldt.

7.2.2 Zijeffecten

De beide operanden van een dyadische operator worden collateraal uitgerekend (d.w.z. over de volgorde van hun berekening valt niets te zeggen).

In een programma'tje als

```
    INT VAR x;
    x := 0;
    put (a + b + x).

a:
     x := x + 4;
     1.

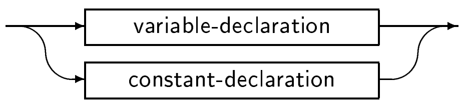
b:
     x := 2 * x;
     2.
```

mag je er niet van uitgaan dat `a,` `b` en `x` in die volgorde worden berekend: iedere andere volgorde is ook goed! Je weet dus niet of hier 11 dan wel 7 (eerst `b`, dan `a`, dan `x`) dan wel 3 (eerst `x` dan `a` dan `b`) uitkomt. Zo'n programma als dit, waarin de uitvoering van de ene operand de waarde van de andere operand beïnvloedt (een **zijeffect**) kan dan ook beter achterwege blijven. Onthoud dat de tekstuele volgorde van de operanden en van de operaties beslist niet de volgorde van uitvoering hoeft te zijn.

7.3 Object-declaraties

Een **declaratie** is een algoritme die een naam aan een waarde (ver)bindt. Het simpelste voorbeeld daarvan zijn de **object declaraties**, die voorkomen in twee soorten:

object-declaration



Deze verschillen overigens onderling alleen in het acces-attribuut en in het feit dat een variabele niet geïnitialiseerd hoeft te worden, terwijl dat voor een constante verplicht is.

7.3.1 Effect van een object-declaratie

De uitvoering van een object-declaratie heeft als **effect** dat eerst de eventuele initialisatie-unit wordt uitgevoerd en vervolgens een object wordt ingevoerd met het gegeven type, acces-attribuut, naam en de berekende waarde. Ontbreekt de initialisatie dan wordt (vooreerst) een ongedefinieerde waarde genomen. Een declaratie levert geen **resultaat** en is dus een actie.

7.3.2 Geldigheidsgebied van declaraties

Een declaratie is geldig binnen een bepaald **geldigheidsgebied**. Dit betekent dat binnen dat deel van het programma een bepaald object met die naam wordt aangeduid. Buiten het geldigheidsgebied van de declaratie is dat object niet bekend en kan de naam voor een ander doel gebruikt worden.

Bij een programma zoals wij tot nog toe schreven, dat bestaat uit een paragraaf en een paar verfijningen, is iedere declaratie in het hele programma geldig. In Elan komen echter ook beperkter geldigheidsgebieden voor (zie 10).

Denk er wel om dat een object niet gebruikt mag worden voordat zijn declaratie is uitgevoerd. Wel mag de naam van het object al eerder in het programma genoemd worden, bijvoorbeeld in een verfijning die tekstueel voorafgaat aan de verfijning waarin de declaratie staat. Dat kan geen kwaad, zo lang het object maar reeds is gedeclareerd op het moment dat het voor het eerst gebruikt wordt.

Binnen het geldigheidsgebied van een object declaratie mag geen tweede declaratie voorkomen voor dezelfde naam. Voor ieder object moet in het programma precies één declaratie voorkomen. Wel mag deze ene declaratie binnen een herhaling voorkomen. Zo is in

```
   WHILE i <= LENGTH boodschap
   REP TEXT CONST teken :: boodschap SUB
i;
       put (3 * teken);
       put (".");
       i INCR 1
   ENDREP
```

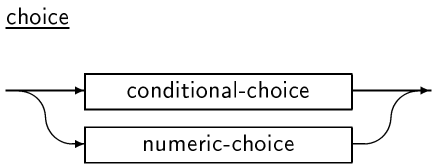
het object `teken` een constante die in iedere slag van de herhaling een volgend teken van de `boodschap` als waarde heeft. Toch is het een constante, want eraan toewijzen is niet mogelijk. Het is een constante met telkens een andere waarde (toegegeven, een vreemd soort constantheid; denk maar aan iemand die telkens weer verliefd is op een ander, en iedere keer zeker weet nu het ware geluk te hebben gevonden).

7.4 De keuze

Met de choice wordt een **keuze** gemaakt tussen een aantal paragrafen in afhankelijkheid van een waarheidswaarde of van een geheel getal.

7.4.1 Voorwaardelijke keuze

De keuze op grond van een voorwaarde is reeds behandeld.



Door dit syntax-diagram zijn veel mogelijke wegen. Met name kan het `ELSE`-deel ontbreken en kunnen er willekeurig veel `ELIF`-delen voorkomen. Een alternatieve schrijfwijze voor `FI` is `ENDIF` of zelfs `END IF`.

Het **effect** van een voorwaardelijke keuze is de seriële samenstelling van het eventuele (zij-)effect van de voorwaarde en het effect van de paragraaf die op grond van de voorwaarde uitgevoerd wordt.

Alle paragrafen waartussen wordt gekozen moeten hetzelfde type hebben. Dat is ook het type van het eventuele resultaat van de keuze. Dat type kan `VOID` zijn (zie boven), wat betekent dat geen resultaat wordt geleverd (de keuze is een actie). Alleen in dat geval mag het `ELSE`-deel weggelaten worden.

Als de keuze gemaakt worden tussen paragrafen die een waarde leveren, dan levert de gekozen paragraaf tevens het **resultaat** van de keuze. In dit geval is het `ELSE`-deel verplicht.

De `ELIF`-constructie dient om, zonder een waterval van `FI`-haakjes, een aantal keuzemogelijkheden te onderscheiden, bijvoorbeeld:

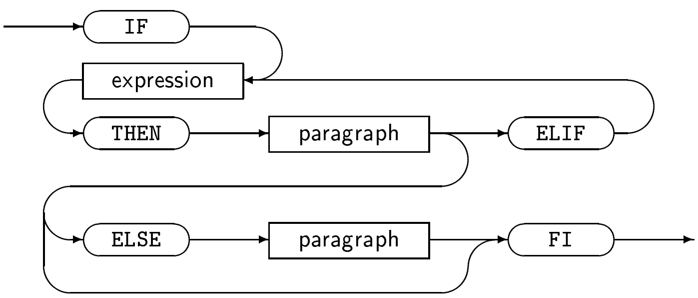
```
   IF   x > 0.0 AND y > 0.0
   THEN put ("punt ligt in eerste
kwadrant")
   ELIF x < 0.0 AND y > 0.0
   THEN put ("punt ligt in tweede
kwadrant")
   ELIF x < 0.0 AND y < 0.0
   THEN put ("punt ligt in derde
kwadrant")
   ELIF x > 0.0 and y < 0.0
   THEN put ("punt ligt in vierde
kwadrant")
   ELSE put ("punt ligt op een as")
   FI
```

Komt in het `THEN`- of `ELSE`-deel een declaratie voor dan is het geldigheidsgebied daarvan, zoals boven gezegd, het hele programma. Toch kan het aldus gedeclareerde object niet gebruikt worden om een resultaat uit de keuze naar buiten te brengen, omdat er ook een weg door het programma bestaat waarop deze declaratie niet wordt uitgevoerd.

```
   IF ene geval
   THEN INT VAR resultaat;
        bereken resultaat op de ene wijze
   ELSE bereken resultaat op de andere
wijze
   FI;
   put (resultaat)
```

is onjuist, omdat het mogelijk is dat een weg door het programma genomen wordt waarbij de declaratie van `resultaat` achterwege blijft.

conditional-choice



Het helpt ook niet om zowel in het THEN-deel als in het ELSE-deel een declaratie van `resultaat` op te nemen:

```
    IF ene geval
    THEN INT VAR resultaat;
         bereken resultaat op de ene wijze
    ELSE INT VAR resultaat;
         bereken resultaat op de andere
 wijze
    FI;
    put (resultaat)
```

Hier bevat het programma twee object-declaraties met dezelfde naam, wat verboden is. De declaratie moet hier reeds buiten de keuze plaatsvinden.

```
    INT VAR resultaat;
    IF ene geval
    THEN bereken resultaat op de ene wijze
    ELSE bereken resultaat op de andere
 wijze
    FI;
    put (resultaat)
```

7.4.2 De gevalonderscheiding

Elan kent nog een andere vorm van keuze, de SELECT-opdracht, die dient om op grond van een geheel getal een bepaalde paragraaf uit te kiezen. In andere talen wordt deze **gevalonderscheiding** een “case-clause”, “switch” of “computed goto” genoemd. Zijn syntax luidt:

Dat is een hele mond vol, daarom geven we eerst een voorbeeld.

```
 weekdag:
    SELECT dag OF
      CASE 1: "maandag"
      CASE 2: "dinsdag"
      CASE 3: "woensdag"
      CASE 4: "donderdag"
      CASE 5: "vrijdag"
      CASE 6: "zaterdag"
      CASE 7: "zondag"
      OTHERWISE "helemaal geen dag"
    ENDSELECT.
```

We nemen hierbij aan dat de dagen van de week op de voor de hand liggende wijze zijn gecodeerd als de gehele getallen 1 t/m 7 en bereiken een afbeelding van zo’n getal op een tekst. Deze oplossing is wat korter en wellicht sneller dan via een cascade van ELIFs.

Dit is nog een eenvoudig voorbeeld: de paragrafen in de CASE-delen zijn erg simpel uitgevallen. Merk op dat de gevallen niet op volgorde hoeven te zijn geordend, en dat twee of meer gevallen die dezelfde paragraaf bevatten kunnen worden samengevat, bijvoorbeeld:

```
 het cijfer is even:
    SELECT cijfer OF
      CASE 0, 2, 4, 6, 8: true
      OTHERWISE false
    ENDSELECT.
```

(hoewel dit ook eenvoudiger geprogrammeerd zou kunnen worden met MOD).

De SELECT-opdracht komt vooral te pas bij het versleutelen van waarden als gehele getallen en het ontsleutelen daarvan tot een keuze tussen algoritmen. Een leeg OTHERWISE-deel is weer facultatief als de SELECT geen waarde oplevert (analoog aan ELSE).

7.5 De herhaling

Voor de herhaling heeft Elan een aantal notaties, die echter alle varianten zijn van eenzelfde grondvorm. De herhaling is een controlstructuur die geen waarde oplevert.

7.5.1 De grondvorm

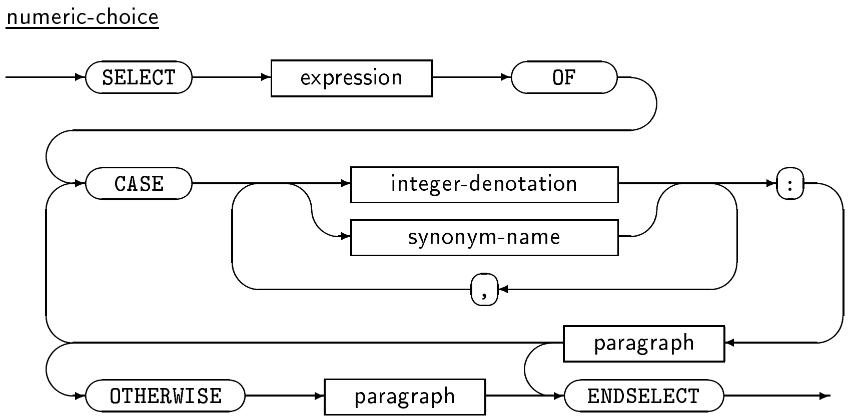
De simpelste vormen van de herhaling zijn de **pre-checked loop**

```
    WHILE er is nog iets te doen
    REP doe dat dan ENDREP
```

en de **post-checked loop**

```
    REP wat er te doen is
    UNTIL klaar
    ENDREP
```

Laten we het WHILE- en UNTIL-deel beide weg, dan ontstaat een in principe **eindeloze herhaling**, zoals de volgende:




```
    REP
      lees een programmatekst;
      vertaal die en controleer hem op
 fouten;
      IF de tekst is foutloos
      THEN
        voer het programma uit;
        druk de resultaten af
      ELSE
        druk de gevonden fouten af
      FI
    ENDREP
```

Alleen een ramp (of het uitschakelen van de computer) kan deze herhaling beëindigen. Vele niet-sequentiële algoritmen, die in dit boek niet behandeld worden, beschrijven zich eindeloos herhalende processen.

7.5.2 De telvariabele

Bij veel herhalingen treedt een variabele op, waarvan de waarde in stapjes veranderd wordt tot een bepaalde situatie bereikt wordt. In verreweg de meeste gevallen is dit een integer variabele, de **telvariabele**, waarvan de waarde met 1 verhoogd wordt, zolang een bepaalde conditie geldt, of tot een bepaalde grens bereikt is. Bijvoorbeeld:

```
INT VAR i :: 1;
WHILE i <= grens
REP
  te herhalen handeling;
  i := i+1
ENDREP
```

Dit voorbeeld vertoont de volgende karakteristieke elementen:

- de *telvariabele* `i`
- de *initialisering* `:: 1`
- de *test* `i <= grens`
- de *te herhalen handeling* die van `i` afhangt
- het *incrementeren* van de telvariabele `i := i+1`

Er is een aantal vrijheidsgraden in de volgorde van deze elementen. Zo kan het incrementeren en testen verwisseld worden, wat zowel invloed heeft op de initialisering als op de test. Als verzekerd is dat `grens > 0` kan men het voorbeeld ook schrijven als

```
INT VAR i :: 1;
REP
  te herhalen handeling;
  i := i+1
UNTIL i > grens
ENDREP
```

en ook als

```
INT VAR i :: 0;
WHILE i < grens
REP
  i := i+1;
  te herhalen handeling
ENDREP
```

Bij het bewijzen van de correctheid en de terminatie van een algoritme speelt een telvariabele vaak een centrale rol. Na voltooiing van de tekst van een programma doet men er goed aan om, alvorens het programma de eerste keer uit te voeren, even na te gaan of alle telvariabelen met de bedoelde waarden zullen beginnen en eindigen.

7.5.3 De begrensde herhaling

Het gebruik van de pre-checked loop brengt het gevaar met zich mee, dat men vergeet om de telvariabele zinvol te initialiseren. Voor het geval dat een `INT`-variabele met stappen van één een bepaald gebied moet doorlopen, voor het normale geval dus, kan de programmeur zijn intentie duidelijk maken in de vorm

```
   FOR index FROM beginwaarde UPTO
grensswaarde
   REP te herhalen handeling ENDREP
```

Voor deze **begrensde herhaling** ("for-loop") geldt:

- `index` moet een variabele zijn, die reeds is gedeclareerd. Zijn waarde vóór de herhaling doet er niet toe. Na afloop van de herhaling heeft hij een ongedefinieerde waarde.
- `beginwaarde` en `grenswaarde` dienen integer expressies te zijn. Ze worden eenmaal, bij het begin van de herhaling, collateraal berekend.
- De telvariabele `index` doorloopt een zeker *herhalingsgebied* met stappen van één, te beginnen met de `beginwaarde`, tot de `grenswaarde` gepasseerd wordt en voor ieder van de (nul

of meer) waarden in dit gebied wordt de `te herhalen handeling` een maal uitgevoerd.
- Bij een variant met `DOWNTO` in plaats van `UPTO` wordt afwaarts geteld.

Voorbeelden (wat wordt er afgedrukt?):

```
FOR t FROM 1 UPTO 10 REP put (t) ENDREP
```

Er wordt 1 2 3 4 5 6 7 8 9 10 afgedrukt.

```
FOR t FROM -10 UPTO 0 REP put (t)
ENDREP
```

Er wordt -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 afgedrukt.

```
FOR t FROM 10 DOWNTO 1 REP put (t)
ENDREP
```

Er wordt 10 9 8 7 6 5 4 3 2 1 afgedrukt.

```
FOR t FROM 10 DOWNTO 1 REP put (11-t)
ENDREP
```

Er wordt 1 2 3 4 5 6 7 8 9 10 afgedrukt.

```
FOR t FROM 10 UPTO 10 REP put (t)
ENDREP
```

Drukt 10 af.

```
FOR t FROM 1 UPTO -1 REP put (t) ENDREP
```

Er wordt niets afgedrukt: bij positieve stapgrootte ligt 1 al voorbij -1.

7.5.4 Verkorte vormen

Bij deze vorm van de herhaling bestaan enige afkortingen:

- `FOR index` mag worden weggelaten mits de telvariabele in de `te herhalen handeling` niet voorkomt.
- `FROM 1` mag worden weggelaten
- `UPTO maxint` mag worden weggelaten.

Voorbeeld:

```
UPTO 3 REP line ENDREP
```

heeft hetzelfde effect als `line; line; line` of `line (3)`.

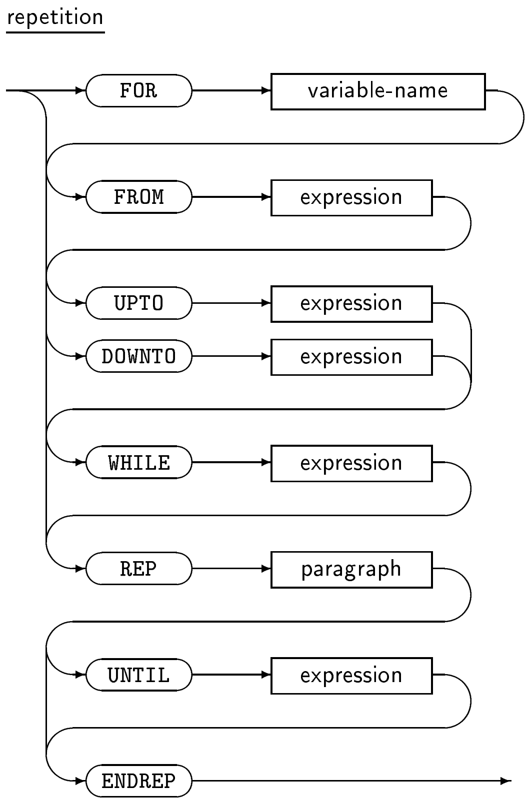
7.5.5 De begrensde herhaling met voorwaarde

Combineert men de voorwaardelijke herhaling met de begrensde herhaling, dan krijgt men de algemene vorm van de herhaling.

Onwillekeurig komt hier de gedachte boven aan een zwitsers universeel zakmes met grote en kleine bladen, een schroevendraaier, flessenopener en kurkentrekker en zelfs een heel klein hamertje en nijptangetje. Omdat men die delen kan weglaten die men niet nodig heeft, is dit toch een redelijk bruikbaar herhalingsmechanisme.

7.5.6 Over de telvariabele

Bij een begrensde herhaling ligt het in de bedoeling dat de telvariabele precies het herhalingsgebied doorloopt. Wanneer de `te herhalen handeling` zelf weer



de waarde van `i` wijzigt, kan het programma heel moeilijk te begrijpen zijn. Het is beter, trucjes te vermijden en de telvariabele zonder zijeffecten zijn loop te laten volgen. Wij geven daarom ook geen antwoord op de vraag, wat het volgende programma doet. Wint het tel-mechanisme of de toewijzing de slag?

```
INT VAR i;
FOR i FROM 1 UPTO n
REP put (i);
    i := 2*i
ENDREP
```

Laat men de begrensde herhaling *niet* zijn loop gaan, dan heeft na afloop de telvariabele de laatst aangenomen waarde.

7.5.7 Voorbeeld: Trailing blanks

Wij geven een voorbeeld van het samenspel tussen een vast herhalingsgebied en een conditie in een `WHILE`-deel. Van een ingelezen tekst willen we de aan het einde voorkomende spaties negeren. Daartoe dienen we de positie van de eerste "trailing blank" te bepalen, d.w.z. van de eerste spatie in de regel die slechts gevolgd wordt door spaties.

```
eerste trailing blank:
  INT VAR plaats;
  FOR plaats FROM LENGTH text DOWNTO 1
  WHILE nog steeds een spatie
  REP
  ENDREP;
  plaats + 1.
```

```
  nog steeds een spatie:

     (tekst SUB plaats) = blank.
```

Als er geen trailing blanks voorkomen levert dit de waarde `LENGTH text + 1` en als de hele tekst bestaat uit spaties de waarde `1`.

7.6 Conclusie

Dit was een wat taai hoofdstuk, vol syntaxdiagrammen en muggenzifterij, met weinig sprekende voorbeelden. Maar het was nodig om een aantal constructies die we al enige tijd losjes gebruiken wat precieser te behandelen, als voorbereiding op de invoering van de samengestelde objecten.

7.7 Opgaven

- (Morse-code) Beschouw het probleem, een tekst om te zetten naar morse-code. In het Morse-alfabet wordt ieder teken voorgesteld door een reeks korte en lange signalen met korte en lange pauzes daartussen. Een kort signaal heet een *punt*, een lang signaal een *streep*. Het Morse-alfabet ziet er als volgt uit:

A	. –	K	– . –	U	. . –	1	. – – – –
B	– . . .	L	. – . .	V	. . . –	2	. . – – –
C	– . – .	M	– –	W	. – –	3	. . . – –
D	– . .	N	– .	X	– . . –	4	. . . . –
E	.	O	– – –	Y	– . – –	5	. . . . .
F	.. – .	P	. – – .	Z	– – . .	6	– . . . .
G	– – .	Q	– – . –	Ä	. – . –	7	– – . . .
H	. . . .	R	. – .	CH	– – – –	8	– – – . .
I	. .	S	. . .	Ö	– – – .	9	– – – – .
J	. – – –	T	–	Ü	. . – –	0	– – – – –

Punt	. – . – . –
Vergissing	. . . .
SOS	. . . – – – . . .
Begin bericht	– . – . –
Einde bericht	. – . – .

  Tussen twee letters komt een spatie, tussen twee woorden wordt een drietal spaties gegeven.
  Schrijf nu een programma om een bericht bestaande uit één regel om te zetten in Morse. Gebruik daarbij een gevalonderscheiding.
- (Morse ontsleutelen) Schrijf een programma dat een Morse-bericht ontsleutelt. Probeer het uit samen met het vorige programma.

Chapter 8

Rijen en sorteren

In de vorige hoofdstukken hebben we de concrete typen van Elan met hun operaties behandeld. In dit hoofdstuk gaan we verder in op de *rij* als mechanisme voor het samenstellen van typen (en daarmee voor het samenstellen van objecten). Als toepassing behandelen we enkele klassieke algoritmen voor zoeken en sorteren.

8.1 samengestelde objecten

Alvorens nader in te gaan op de rijen zetten we een aantal belangrijke begrippen en termen nog eens op een rijtje. Een **samengesteld object** is een object met een samengestelde waarde.

8.1.1 Waarden, objecten en typen

De wereld om ons heen bevat ideeen en fysieke objecten, feiten, dingen en hersenspinsels. Bij het schrijven van een programma verzinnen wij daarvoor representaties binnen de computer, zodat we daarmee kunnen laten rekenen. Onder een **waarde** verstaan we enerzijds een element van een abstracte verzameling waarvoor een representatie binnen de computer bestaat, en anderzijds ook die representatie zelf.

Als we spreken van "de waarde drie" dan kunnen we daarmee zowel een bepaald wiskundig concept bedoelen, de opvolger van twee, als een representatie daarvan in de computer. In wijder verband kunnen we het hebben over "de waarde van de dollar" of zelfs "de waarden van de westerse beschaving". Het woord waarde heeft nu eenmaal een meervoudige betekenis. Over dit soort meerduidigheden in de betekenis van woorden maken we ons meestal niet druk, omdat de context wel aangeeft welke betekenis bedoeld is.

We kunnen waarden niet rechtstreeks manipuleren, maar schrijven in plaats daarvan algoritmen om het gewenste resultaat te bereiken. In een algoritme kunnen we een bepaalde waarde aanduiden (denoteren) door middel van een **denotatie**, een schrijfwijze voor die waarde in de programmeertaal. We kunnen ook een **naam** geven aan die waarde, door middel van een declaratie of een toewijzing. De naam is ons handvat op die waarde. De combinatie van een naam en een waarde, daaraan in de loop van de uitvoering van de algoritme gebonden, noemen we een **object**.

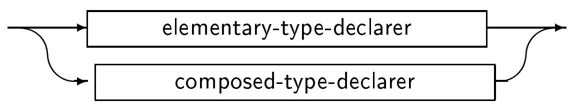
De binding van een waarde aan een naam is slechts tijdelijk. In de loop der tijd kunnen achtereenvolgens verschillende waarden aan een naam verbonden worden, bijvoorbeeld door het herhalen van een geïnitialiseerde declaratie. Sommige objecten (de *variabelen*) staan toe dat hun waarde door een toewijzing vervangen wordt door een andere waarde. Andere objecten (de *constanten*) laten zo'n toewijzing niet toe.

De waarden kunnen we indelen in verschillende klassen, de **typen**. Een type heeft een naam, een **domein** (de verzameling van waarden van dat type) en een verzameling operaties die op waarden van dat type van toepassing zijn. Een type kan elementair zijn, of samengesteld (uit benoemde typen) door middel van een **datastructuur** ("gegevensstructuur", "data structure").

8.1.2 Type-declarers

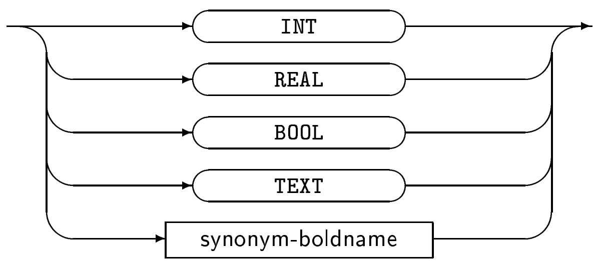
Bij de declaratie van een variabele of constante wordt zijn type (het type dat zijn waarde moet bezitten) aangeduid door een *type-declarer*.

type-declarer



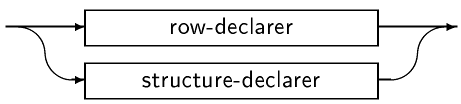
De vier concrete elementaire typen kennen we al.

elementary-type-declarer



De mogelijkheid, een abstract type te declareren via een *synonym-boldname* behandelen we later in dit hoofdstuk.

composed-type-declarer



De *rijen* en de *structuren* zijn de datastructuren van Elan. In dit hoofdstuk behandelen we daarvan de rijen. De structuren worden beschreven in het tweede deel van dit boek.

8.2 Rijen

Onder een (eendimensionale) **rij** verstaan we een lineaire rangschikking van een beperkt aantal objecten van eenzelfde type, de **elementen** van de rij. De nummering van de elementen loopt ononderbroken vanaf de **ondergrens** één tot en met de **bovengrens** van de rij, gelijk aan het aantal elementen. De positie van een element in de rij heet zijn **index**.

Het type van een rij wordt aangeduid door een row-declarer, waarin we het aantal en het type van de elementen terugvinden.

Het syntax-diagram voor de row-declarer staat op de volgende pagina.

Het row-symbol heeft als representatie `ROW`. Achter het row-symbol volgt het aantal van de elementen. Over de mogelijkheid een synonym-name als aantal aan te geven zullen we later spreken in de context van synonym-declarations. Voorbeelden:

`ROW 10 REAL`	een rij van 10 reals, genummerd van 1 tot 10
`ROW 200 ROW 60 TEXT`	een boek bestaand uit 200 pagina's van 60 regels elk.

Evenals de reeds bekende declarers `INT`, `REAL` etc. worden de row-declarers gebruikt om typen aan te duiden in object-declaraties. Bijvoorbeeld:

```
ROW 10 REAL VAR oude meetwaarden,
                nieuwe meetwaarden
```

We kunnen een rij als één enkel object beschouwen en hem dus in een klap aan een geschikte variabele toewijzen

```
oude meetwaarden:= nieuwe meetwaarden
```

De waarde van `oude meetwaarden` wordt een **kopie** van de waarde van `nieuwe meetwaarden`. Onmiddelijk na deze toewijzing hebben de elementen van beide rijen dus dezelfde waarden, maar verder hebben de rijen niets gemeen: toewijzing aan de een heeft geen effect op de ander. Dit zou anders liggen als bij rij-toewijzingen geen kopie werd genomen! Vanwege het kopiëren is een rij-toewijzing natuurlijk een wat langduriger operatie.

Behalve de toewijzing bestaan er geen concrete operaties op een rij als geheel.

8.2.1 De elementen van een rij

Men kan de elementen van een rij afzonderlijk manipuleren, d.w.z. de waarde van een element nemen (**lees acces**) of het element een andere waarde geven (**schrijf acces**). Dit laatste is alleen mogelijk voor de elementen van een rij-variabele.

In de context van de declaratie van de row-variabele

```
ROW 10 REAL VAR meetwaarde
```

wordt i-de meetwaarde aangeduid met `meetwaarde[i]`. Zijn waarde kan bijvoorbeeld worden afgedrukt door

```
put (meetwaarden[i])
```

en worden gewijzigd door de toewijzing

```
meetwaarde[i]:= 0.0
```

Zo'n aanduiding heet een **subscriptie**.

De waarde van de integer uitdrukking tussen de vierkante haken heet de *index*. De index behoort een waarde te hebben tussen de ondergrens 1 en de bovengrens van de rij; zo niet dan is de subscriptie ongedefinieerd en wordt de uitvoering van het programma afgebroken.

Toewijzing van een waarde aan een van de elementen van een rij verandert de waarde van de andere elementen niet.

Het stukje programma

```
FOR i UPTO 10
REP
  oude meetwaarden[i]:=
      nieuwe meetwaarden[i]
ENDREP
```

heeft eenzelfde effect als de bovengenoemde toewijzing

```
oude meetwaarden:= nieuwe meetwaarden
```

8.2.2 Overerving van accesrecht

De aanduiding `meetwaarde[i]` gedraagt zich in alle opzichten als een real variabele: de subscriptie erft het acces-attribuut `VAR` van de rij waaruit gesubscribeerd wordt. Laat `EL` staan voor het type van de elementen van de rij. Er geldt:

> Subscriptie van een `EL`-rij-variabele levert een `EL`-variabele.
> Subscriptie van een `EL`-rij-constante levert een `EL`-constante.

Van een samengestelde constante zijn dus de componenten constanten. Zij kunnen dus niet door een toewijzing gewijzigd worden.

De andere constructen uit Elan waarvoor een overerving van **accesrecht** geldt (en die zich dus als een variabele kunnen gedragen) zijn de keuze en de verfijning, bijvoorbeeld:

```
pas de kleinste aan:
  de kleinste:= de grootste.

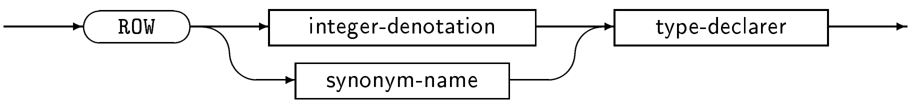
de kleinste:
  IF a > b THEN b ELSE a FI.

de grootste:
  IF a > b THEN a ELSE b FI.
```

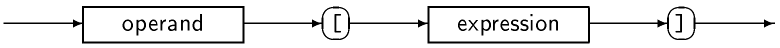
In het volgende deel van dit boek zullen we een dergelijke overerving van accesrecht tegenkomen bij de selectie uit structuren.

8.2.3 Denotatie van eendimensionale rijen

Anders dan voor de elementaire typen als `INT` en `REAL` kent Elan strikt genomen geen denotatie voor rijconstanten. In plaats daarvan bestaat er een algemene controlstructuur, die uit waarden voor de elementen een samengesteld object construeert, de **display**. Deze row-declarer

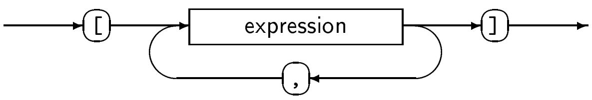


subscription



vervult mede de rol van denotatie voor rijen. De syntax voor de display luidt:

display



Voor een row-display geldt het volgende:

- De expressies in een display, gebruikt om een rij te denoteren, dienen allemaal van eenzelfde type te zijn, het type van de elementen van de verkregen rij.
- Het aantal daarvan bepaalt de bovengrens van de verkregen rij.
- Zij worden collateraal berekend, en hun resultaten tot een rij samengesteld.

Voorbeelden:

`[0, 1, 3 DIV 2, 7]` is een `ROW 4 INT`-display
`[0.0, 1.0, 3/2, 7.0]` is een `ROW 4 REAL`-display
`[ik", "jij", "hij"]` is een `ROW 3 TEXT`-display.

Met behulp van een constanten-declaratie kunnen we aan de waarde van een display een naam koppelen.

```
ROW 10 INT CONST eerste 10 priemen::
          [2,3,5,7,11,13,17,19,23,29]
```

8.3 Voorbeeld: woorden turven

Als voorbeeld van de toepassing van rijen beschouwen we het probleem bij een invoer, bestaande uit woorden, te bepalen hoe vaak ieder woord voorkomt. We houden daartoe een lijst van woorden met hun frequenties bij.

```
woorden turven:
  begin met een lege lijst;
  WHILE er volgt nog een woord
  REP
    turf dat woord
  ENDREP;
  druk lijst af.
```

We reserveren plaats voor hoogstens 100 woorden.

```
begin met een lege lijst:
  ROW 100 TEXT VAR woordenlijst;
  ROW 100 INT VAR frequenties;
  INT VAR eerste vrije:: 1.
```

Het einde van de invoer geven we aan met een woord bestaand uit negen letters z.

```
er volgt nog een woord:
  TEXT VAR woord; get(woord);
  woord <> 9 * "z".
```

Gelijke woorden worden slechts eenmaal in de lijst opgenomen.

```
turf dat woord:
  IF het woord staat al in de lijst
  THEN verhoog zijn telling
  ELSE voeg het toe
  FI.
```

De woordenlijst is niet geordend, zodat we hem telkens in zijn geheel zullen moeten doorzoeken.

```
het woord staat al in de lijst:
  INT VAR plaats;
  FOR plaats FROM 1 UPTO eerste vrije - 1
  REP
    IF woordenlijst [plaats] = woord
    THEN
      LEAVE het woord staat al in de lijst
         WITH true
    FI
  ENDREP;
  false.
```

Het turven zelf is eenvoudig.

```
verhoog zijn telling:
  frequenties [plaats] INCR 1.
```

```
voeg het toe:
  IF eerste vrije > 100
  THEN
    geen plaats meer
  ELSE
    woordenlijst [eerste vrije]:= woord;
    frequenties [eerste vrije]:= 1;
    eerste vrije INCR 1
  FI.
```

Wat doen we als de woordenlijst vol is, dat wil zeggen: honderd verschillende woorden met hun tellingen zijn al opgenomen en het woord waar we nu mee aan komen zetten is daar niet bij? Een mogelijkheid is, er op dat moment de brui aan te geven

```
geen plaats meer:
  put line ("Woordenlijst is vol.");
  druk lijst af;
  LEAVE woorden turven.
```

Een alternatieve mogelijkheid is, het turven van de voorkomens van deze honderd woorden eerst af te maken. Dan wordt deze laatste verfijning nog eenvoudiger.

```
geen plaats meer: .
```

We maken ons geen zorgen over de volgorde waarin we de woorden afdrukken:

```
druk lijst af:
  put line ("Frequentie:        woord:");
  INT VAR w;
  FOR w FROM 1 UPTO eerste vrije - 1
  REP
    put line (frequenties[w]);
    put ("  " + woordenlijst[w])
  ENDREP.
```

We drukken eerst de frequentie (een geheel getal) en dan het woord, om een hele banale reden: dan krijgen we twee nette kolommen.

8.4 Geneste rijen

Een rij is zelf weer een object met een bepaald type en kan daarom weer optreden als element in een andere rij; zo krijg je een geneste rij. Zo is bijvoorbeeld een samengestelde type-declarer in een declaratie als

$$\underbrace{\texttt{ROW 200 }\underbrace{\texttt{ROW 60 TEXT}}_{\text{declarer}}}_{\text{declarer}}\texttt{ VAR book}$$

geheel volgens de regels en we kunnen op deze manier verder gaan met declaraties als

```
   ROW 10000 ROW 200 ROW 60 TEXT VAR
bibliotheek;
   ROW 200 ROW 60 TEXT VAR boek;
   ROW 60 TEXT VAR bladzijde;
```

Het dupliceren van een `boek` in de `bibliotheek` zou nu kunnen gebeuren met

```
bibliotheek[4711]:= bibliotheek[9999]
```

Iets wat hier duidelijk vlugger is gezegd (geschreven) dan gedaan.

Een samengestelde rij kent een samengestelde subscriptie,

```
bibliotheek[12][14][8]
```

is een aanduiding van de 8-ste regel op de 14-de pagina van het 12-de boek uit de `bibliotheek`.

Een display voor een rij met daarin geneste rijen bestaat natuurlijk uit geneste displays. De buitenste haakjes van zo'n geneste display horen bij de linker row-aanduiding en de binnenste haakjes bij de rechter row-aanduiding, terwijl de diepte van de nesting gelijk moet zijn aan de diepte van de nesting van de rijen.

Voorbeelden:

```
 ROW 15 ROW 2 TEXT CONST vertaling::
   [
    ["aap",    "monkey"],
    ["noot",   "nut"],
    ["Mies",   "Mary"],
    ["Wim",    "Bill"],
    ["zus",    "sister"],
    ["Jet",    "Harriet"],
    ["Teun",   "Tony"],
    ["vuur",   "fire"],
    ["Gijs",   "Gilbert"],
    ["lam",    "lamb"],
    ["Kees",   "Cornelius"],
    ["bok",    "he-goat"],
    ["weide",  "meadow"],
    ["does",   "poodle"],
    ["schapen","sheep"]
   ]

ROW 3 ROW 3 INT CONST magisch vierkant::
  [[8, 3, 4], [1, 5, 9], [6, 7, 2]]
```

8.5 Over de grenzen van rijen

De ondergrens van een rij is altijd één. De bovengrens moet een gehele waarde zijn die niet mag afhangen van de uitvoering van het programma, dus een denotatie, niet het resultaat van een berekening. De reden voor deze weinig flexibele afspraak ligt daarin dat het aantal elementen mede bepalend is voor het type: `ROW 3 INT` en `ROW 4 INT` zijn verschillende typen, en toewijzing van een waarde van het ene type aan een variabele van het andere type is uitgesloten.

De bovengrens komt meestal op meer dan één plaats in de algoritme voor (in de rij-declaratie, als limiet voor een herhaling e.d.). Wil ik de bovengrens veranderen dan moet ik vaak op vele plaatsen in het programma een wijziging aanbrengen. Dat is vervelend, en daarom is er een speciaal mechanisme in Elan ingevoerd, de *synonym-declaration*.

> De **synomiem declaratie** geeft een naam aan een denotatie of aan een delcarer, zodat die naam in plaats daarvan gebruikt kan worden.

Voor deze synoniem-declaratie staat het syntax-diagram op de volgende pagina.

Een voorbeeld van de eerste vorm is:

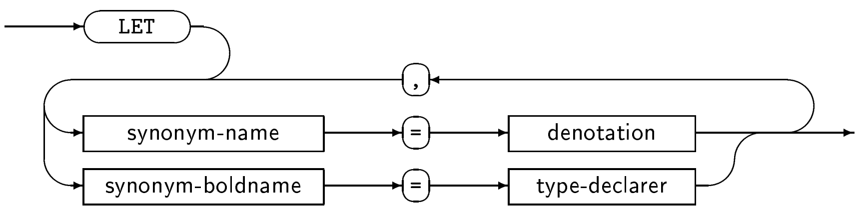
```
LET aantal = 200
```

Deze declaratie voert een **synoniem** `aantal` in voor de denotatie 200 dat overal mag worden gebruikt waar die denotatie mag staan, bijvoorbeeld:

```
ROW aantal REAL VAR meting;
FOR i UPTO aantal
REP
  meting[i]:= 0.0
ENDREP
```

Met behulp hiervan kan van het precieze aantal metingen geabstraheerd worden. De synonym-declaratie is dus een abstractie-middel.

synonym-declaration



Overigens valt niet te ontkennen dat er in Elan een verwarrend aantal manieren bestaat om aan een waarde een naam te verbinden. De verschillen tussen de volgende constructies zijn nogal subtiel.

- `INT VAR aantal:: 200`
  Er wordt een `INT`-variabele aangemaakt met de naam `aantal` die (voorlopig) de waarde `200` krijgt. Door een toewijzing kan hij een andere waarde krijgen.
- `INT CONST aantal:: 200`
  Er wordt een `INT`-constante aangemaakt met de naam `aantal` en met de waarde `200`; zijn waarde kan niet door toewijzing gewijzigd worden.
- `aantal : 200.`
  Iedere keer dat hij wordt opgeroepen levert de verfijning `aantal` de waarde van de denotatie `200`.
- `LET aantal = 200`
  De naam `aantal` wordt synoniem voor de denotatie `200`.

Alleen de laatstgenoemde kan in plaats van een denotatie optreden als bovengrens van een rij.

8.6 Declaratie van abstracte typen

Het is vaak nuttig om in staat te zijn een type als elementair te beschouwen (door het een naam te geven) ook al zijn zijn waarden samengesteld. Zo kun je verkiezen een rij van 3 reals te beschouwen als een *vector* in de $\Re^3$.

De eenvoudigste manier om zo'n **abstract type** in te voeren is het gebruik van de zojuist ingevoerde synoniem declaratie, die ook een naam (in hoofdletters) kan geven aan een declarer. Voorbeeld:

```
LET VEC = ROW 3 REAL
```

Het type met de naam `VEC` wordt gerealiseerd als een rij van 3 reals. Er is geen enkel verschil in betekenis tussen

```
ROW 3 REAL VAR my vec
VEC VAR my vec
```

maar het laatste is korter en abstracter: je hebt het niet over zomaar een rij van 3 reals maar over een die bedoeld is als vector.

Onder het **domein** van een type verstaan we de verzameling daartoe behorende waarden.

Na deze declaratie liggen naam en domein van het type vast, maar de enige beschikbare operaties zijn die van een rij van 3 reals: toewijzing, denotatie en subscriptie. De karakteristieke operaties van vectoren als optelling en vermenigvuldiging moet je zelf declareren, in de vorm van procedures (zie volgende hoofdstuk) of operatoren (helaas niet meer in dit deel).

Een *matrix* in de $\Re^3$ kun je representeren als een rij van drie rijen van elk 3 reals, dus als een rij van 3 vectoren:

```
LET MAT = ROW 3 VEC
```

Elan kent alleen **eendimensionale rijen**, andere talen gericht op numerieke toepassingen kennen ook **meerdimensionale rijen** (zo heeft een matrix de dimensie twee).

8.7 Sorteren

We behandelen nu een aantal eenvoudige algoritmen voor het probleem, een rij te **sorteren** (ordenen, rangschikken).

Gegeven is een rij-variabele van n elementen $x[1], x[2], \ldots, x[n]$ en een ordeningsrelatie $\leq$ tussen de elementen. Gevraagd is deze elementen zo te rangschikken dat voor iedere i en j in $[1 : n]$ geldt: $i < j \rightarrow x[i] \leq x[j]$. Wij willen dus de elementen van de rij zo herschikken dat ze aan de ordeningsrelatie voldoen, zonder dat er elementen bijkomen of verloren gaan. In termen van de programmeertaal kunnen we dit opvatten als het her-schikken van de elementen van de rij-variabele `x`. Men noemt dit **in-situ sorteren** oftewel ter-plekke sorteren, in tegenstelling tot de mogelijkheid, de oorspronkelijke rij ongemoeid te laten en de gesorteerde rij in een andere rij-variabele op te bouwen.

We nemen bij ieder van de algoritmen de volgende declaraties aan:

```
LET n = 1000;
ROW n EL VAR x
```

Hierin is `EL` het type van de elementen (vul maar naar believen `INT` of `TEXT` o.i.d. in), `n` is het aantal elementen oftewel de index van het laatste element. Verder nemen we aan dat alle elementen van de rij `x` een waarde gekregen hebben.

8.7.1 Voorbeeld: Selection Sort

De grondgedachte van deze algoritme is als volgt: we splitsen de rij op in een gesorteerd deel (aanvankelijk leeg) en een ongesorteerd deel. In het ongesorteerde deel zoeken wij telkens het kleinste element en plakken dat achteraan het gesorteerde gedeelte, dat daardoor één groter wordt. In een plaatje

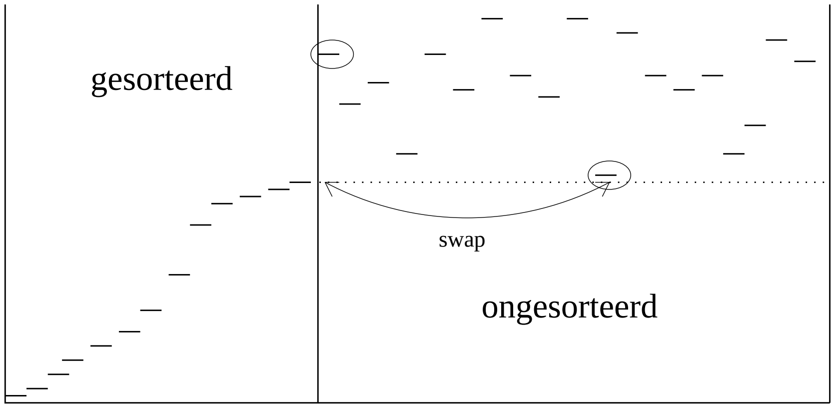


Figure 8.1: Selection Sort

```
selection sort:
  eerst is het gesorteerde deel leeg;
  WHILE niet alle elementen op hun plaats
  REP
     pak het kleinste uit het
ongesorteerde deel;
     plak dit achteraan het gesorteerde
deel
  ENDREP.

eerst is het gesorteerde deel leeg:
  INT VAR i:: 0.
```

Merk op dat hier, zoals in zo vele voorbeelden, een geïnitialiseerde declaratie een duidelijk abstracte betekenis heeft, die makkelijk is te verbaliseren.

```
niet alle elementen op hun plaats:
  i < n-1.
```

We sorteren het n-de element niet mee want als de eerste $n-1$ elementen op hun plaats staan staat het n-de vanzelf ook goed.

```
pak het kleinste uit het ongesorteerde
deel:
  begin bij de eerste;
  WHILE niet allemaal bekeken
  REP
    kijk of je een kleinere hebt
  ENDREP.

begin bij de eerste:
  INT VAR j:: i+1;
  EL VAR kleinste:: x[j];
  INT VAR index kleinste:: j.

niet allemaal bekeken:
  j INCR 1; j <= n.

kijk of je een kleinere hebt:
  IF x[j] < kleinste
  THEN kleinste:= x[j];
       index kleinste:= j
  FI.

plak dit achteraan het gesorteerde deel:
  EL CONST hulp:: x[i+1];
  x[i+1]:= kleinste;
  x[index kleinste]:= hulp;
  i INCR 1.
```

Hoe hangt de uitvoeringstijd van deze algoritme af van n, het aantal elementen? Er moet in totaal $n-1$ keer een kleinste gezocht worden, in een steeds kleiner interval, waarbij de tijd om een interval te doorzoeken practisch recht evenredig is met de lengte van dat interval. De tijd is van de orde $\sum_{i=1}^{n-1}(n-i)$ dus voor grotere n evenredig met n^2.

8.7.2 Voorbeeld: Insertion Sort

Weer is het eerste deel van de rij gesorteerd, het tweede ongesorteerd. Nu pakken we echter steeds het eerstvolgende van de ongesorteerde elementen en voegen dat op de juiste plaats bij de gesorteerde in, in een plaatje:

```
insertion sort:
  INT VAR j;
  FOR j FROM 2 UPTO n
  REP
    voeg jde element op juiste plaats in
  ENDREP.

voeg jde element op juiste plaats in:
  bepaal de plaats waar het hoort;
  zet het op die plaats.
```

We bepalen de plaats door lineair zoeken. In 8.8 beschrijven we een intelligentere zoekmethode.

```
bepaal de plaats waar het hoort:
  INT VAR plaats:: 1;
  WHILE x[plaats] < x[j]
  REP
    plaats INCR 1
  ENDREP.
```

Merk op dat deze onbegrensde herhaling altijd eindigt: in het ergste geval worden de indices 1 tot en met j allemaal doorlopen, gemiddeld echter de helft van dit gebied.

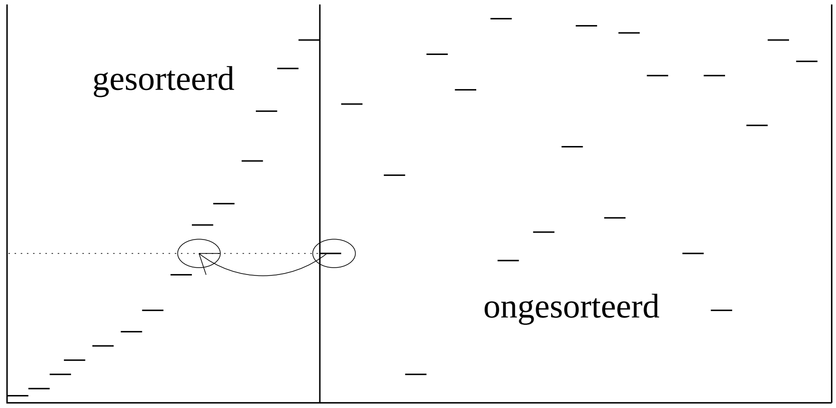


Figure 8.2: Insertion Sort

```
zet het op die plaats:
  pak het jde element;
  verschuif de andere;
  drop het jde element.

pak het jde element:
  EL CONST element:: x[j].
```

Het verschuiven van de andere elementen naar rechts is nodig om ruimte te maken voor het element; daarbij wordt het gesorteerde deel één langer en overschrijft het oorspronkelijke j-de element. Merk op dat de verschuiving van boven naar beneden *moet* gebeuren (wat gebeurt er als we de andere kant op werken?).

```
verschuif de andere:
  INT VAR k;
  FOR k FROM j DOWNTO plaats + 1
  REP
    x[k]:= x[k-1]
  ENDREP.

drop het jde element:
  x[plaats]:= element.
```

8.7.3 Voorbeeld: Bubble Sort

Iedere permutatie kan door verwisseling van telkens 2 opeenvolgende elementen bereikt worden. Als we dus maar vaak genoeg een tweetal uit volgorde staande elementen omwisselen moet de rij gesorteerd raken. Dit voert tot de gedachte de rij herhaaldelijk te doorlopen, waarbij elk opeenvolgend tweetal geordend wordt. Doe dit zo vaak tot niets meer te doen is.

```
bubble sort:
  REP zwaai door de rij
  UNTIL niets uit volgorde
  ENDREP.

zwaai door de rij:
  INT VAR i;
  BOOL VAR iets verwisseld:: FALSE;
  FOR i FROM 1 UPTO n-1
  REP
    verwissel eventueel paar
  ENDREP.

verwissel eventueel paar:
  IF x[i] > x[i+1]
  THEN EL CONST hulp:: x[i];
       x[i]:= x[i+1];
       x[i+1]:= hulp;
       iets verwisseld:= TRUE
  FI.

niets uit volgorde:
  NOT iets verwisseld.
```

Is de rij "bijna" gesorteerd dan zal de Bubble Sort aan een klein aantal zwaaien genoeg hebben. Alleen in dit speciale geval is hij merkbaar efficiënter dan de twee eerdere methoden.

Na de eerste zwaai zal, zoals eenvoudig is na te gaan, het grootste element naar boven zijn versleept. Dus kan de volgende zwaai een element korter worden. De volgende zwaai brengt het een-na-grootste element naar de een-na-bovenste plaats, enzovoorts. Hiermee rekening houdend kan de algoritme als volgt verbeterd worden:

```
bubble sort:
  INT VAR k;
  FOR k REP zwaai door de rij
  UNTIL niets uit volgorde
  ENDREP.

zwaai door de rij:
  INT VAR i;
  BOOL VAR iets verwisseld:: FALSE;
  FOR i FROM 1 UPTO n - k
  REP
    verwissel eventueel paar
  ENDREP.
```

Opgave: Na een zwaai zit gegarandeerd de "grootste" boven. Daarom kan de volgende zwaai 1 korter zijn. Pas daartoe de algoritme aan.

8.7.4 Moet je nou echt zo sorteren?

Hoe hangt de uitvoeringstijd van de gegeven algoritmen af van n, het aantal elementen (men zegt: wat is zijn **complexiteit**)? In de eerste plaats hangt deze natuurlijk af van de snelheid van de computer — een microcomputer is zo'n 10 keer trager dan een "echte" computer. Bij een vergelijkbare opbouw ("architectuur") van de verschillende soorten computers kunnen we er echter van uitgaan dat ze ongeveer dezelfde handelingen moeten verrichten. In het algemeen kunnen we de uitvoeringstijd uitdrukken als het aantal handelingen maal een factor die de machinesnelheid aangeeft. De uitvoeringstijd hangt niet alleen van het aantal maar ook in zekere mate van de waarden van de elementen af. We kunnen daarom onderscheiden

- (**worst case**) In het ergste geval, hoeveel handelingen zijn er dan te verrichten als functie van n?
- (**best case**) Idem, in het beste geval.
- (**average case**) In het gemiddelde geval. Maar wat betekent hier eigenlijk "gemiddeld"?

Voor grote n zinken alle handelingen in het niet ten opzichte van de vaakst herhaalde.

De uitvoeringstijd hangt uiteraard direct af van de snelheid van de machine die de handelingen verricht. Bovendien is de uitvoeringstijd ook afhankelijk van de aard van de te verrichten handelingen en van details van de programmering. Deze laatste twee aspecten bepalen slechts de vermenigvuldigingsfactor die de uitvoeringstijd en het aantal verrichte handelingen verbindt.

In het ergste geval en in het gemiddelde geval hangt voor de gegeven algoritmen de uitvoeringstijd kwadratisch af van n, d.w.z. de uitvoeringstijd $= O(n^2)$, zoals voor de algoritme Selection Sort beredeneerd is.

Tweemaal zoveel elementen sorteren kost dus viermaal zoveel tijd!

In het ergste geval en in het gemiddelde geval hangt voor de gegeven algoritmen de uitvoeringstijd kwadratisch af van n, d.w.z. de uitvoeringstijd *is van de orde* $O(n^2)$, zoals voor de algoritme Selection Sort beredeneerd is. In het beste geval kent Bubble Sort maar een enkele zwaai door de rij en is daarmee van de orde $O(n)$ (*lineair*). Hetzelfde geldt voor Insertion Sort, waar in het beste geval geen (tijdrovende) verschuivingen hoeven plaats te vinden (ga dat na).

	beste geval	gemiddeld geval	slechtste geval
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$

Figure 8.3: Complexiteit van drie sorteeralgoritmen

Als rekenvoorbeeld beschouwen we het sorteren van een lijst namen met een $O(n^2)$-algoritme op een of andere vrij trage computer. Vul de volgende tabel aan.

sportclub	300	10 sec.
grote club	3000	
ANWB	300000	

De betere sorteeralgoritmen, zoals Quicksort die we later behandelen, zijn $O(nlogn)$, d.w.z. voor voldoende grote n willekeurig veel sneller. Desondanks neemt de sorteertijd ook daar nog meer dan lineair toe met het aantal elementen. De inwoneradministratie van de USA wordt zeker niet dagelijks opnieuw gesorteerd, maar een bestand van die grootte wordt met behulp van bijzondere datastructuren op achtergrondgeheugen gesorteerd bijgehouden.

8.8 Voorbeeld: Halveringsalgoritme

We beschrijven tot slot van dit hoofdstuk de **halveringsalgoritme** (**binair zoeken**, **binary search**), een klassieke algoritme voor het vinden van de index van een gegeven element el in een gesorteerde rij x. Bij de opstijgend gesorteerde rij

$$x[1], x[2], ...x[n], n > 1$$

zoeken we een index k zo, dat

$$x[k] \geq el \wedge x[k-1] < el$$

d.w.z. als er een element gelijk aan el in de rij voorkomt dan is k de kleinste index van zo'n element, en als zo'n element niet voorkomt dan is k de index die el zou hebben als dit element in de rij werd toegevoegd.

Beschouw daartoe een willekeurige index $t \in [1:n]$. Er geldt:

$$x[t] > el \vee x[t] = el \vee x[t] < el.$$

Het beschouwen van het element met index t stelt ons in staat iets over de index k te zeggen:

$$x[t] \leq el \rightarrow k \in [1:t]$$

$$x[t] > el \rightarrow k \in [t+1:n].$$

Als we de index t in het midden van de rij kiezen kunnen we dus in één klap het te doorzoeken gebied halveren. Dit doen we zolang tot we nog maar een element over hebben.

```
halveringsalgoritme:
   begin met het hele interval;
   REP
     kies een t;
     IF x [t] >= el
     THEN ga verder in linkerhelft
     ELSE ga verder in rechterhelft
     FI
   UNTIL nog maar 1 element over
   ENDREP.
```

```
begin met het hele interval:
  INT VAR onder:: 1, boven :: n.
```

Merk op dat *onder* $<$ *boven* omdat $n > 1$. We kiezen t zoveel mogelijk in het midden van het interval [*onder* : *boven*].

```
kies een t:
  INT CONST t:: (onder + boven) DIV 2.
```

Voor de terminatie van deze algoritme moet gegarandeerd zijn dat de helften echt kleiner zijn dan het geheel. Door de speciale betekenis van `DIV` (deling met veronachtzaming van de rest) geldt $onder < boven \rightarrow (onder \leq t < boven)$.

```
ga verder in linkerhelft:
  boven:= t.
```

Hierbij wordt het interval echt kleiner omdat $t < boven$.

```
ga verder in rechterhelft:
  onder:= t + 1.
nog maar 1 element over:
  onder = boven.
```

Ook hier wordt het interval echt kleiner, omdat $t \geq onder$, dus $t + 1 > onder$. Merk op dat een gesorteerde rij van 2^k elementen op deze wijze in k stappen wordt doorzocht, en in het algemeen n elementen in ongeveer $log(n)$ stappen. Men zegt: de algoritme is van de orde $log(n)$.

Een rij van 1 miljoen elementen wordt in zo'n 20 stappen doorzocht — heel wat sneller dan lineair zoeken. We kunnen de algoritme Insertion Sort hiermee wat sneller krijgen, maar niet al te veel omdat het op de juiste plaats zetten van het element kwadratisch blijft en de meeste tijd neemt.

8.9 Opgaven

1. (Shaker Sort) Een variant van Bubble Sort wordt verkregen door afwisselend van links naar rechts en van rechts naar links te zwaaien, waarbij telkens een grootste rechts en een kleinste links belandt. Het ongesorteerde gebied wordt dus links en rechts ingesloten tussen twee gesorteerde delen. Formuleer deze Shaker Sort, waarbij U de zwaaien telkens korter laat worden.
2. (Symmetrische halveringsalgoritme) Bij de in dit hoofdstuk gegeven formulering van de halveringsalgoritme hebben we eigenlijk maar twee van de drie mogelijkheden benut (omdat we het geval dat t toevallig de index van het juiste element is niet bijzonder onderscheiden). Als gevolg hiervan is de algoritme niet geheel symmetrisch in zijn behandeling van linker- en rechterhelft. Schrijf een variant van de halveringsalgoritme die de drie mogelijkheden wel onderscheidt en symmetrischer is. Gebruik daarbij de `LEAVE`-opdracht.
3. (Vergelijking sorteermethoden) Doe experimenteel en door redeneren een vergelijkend warenonderzoek tussen de genoemde (en eventueel andere) sorteermethoden. Onderscheid het beste, gemiddelde en ergste geval. Verklaar de optredende verschillen.
4. (Kalender) Schrijf een programma dat voor een in te lezen jaar na 1900 en voor 2100 een kalender afdrukt. Maak daarbij gebruik van de kennis dat 1 januari 1901 een dinsdag was. De kalender dient een goed verzorgde indruk te maken, dat wil zeggen, dat de layout van de output erg belangrijk is. Druk bovenaan het betrokken jaartal af. Daaronder moeten 4 blokken komen van elk 3 maanden naast elkaar. In zo'n blok moeten op de eerste regel de namen van de 3 maanden staan en daaronder in verticale kolommen de weken met boven elke week nog een weeknummer. Vooraan komt natuurlijk een kolom met de afgekorte namen van de dagen te staan.
   Advies: Bouw elk kwartaal eerst helemaal in het geheugen op als een rij (bijvoorbeeld van teksten: de regels) en druk die dan af.
5. Schrijf een programma dat, van een ingelezen tekst, een frequentietabel maakt van de letters, en deze tabel vervolgens als een staafdiagram met verticale staven op het scherm zet. Kies de verticale schaal zodanig dat de langste staaf juist nog op het scherm past.
6. (Tekstschaaf) Schrijf een programma dat een tekst (bestaande uit woorden met leestekens en layout, verdeeld over ettelijke regels en afgesloten met een dollarteken) inleest en in een minimaal aantal uitgelijnde regels weer uitgeeft. Een regel is uitgelijnd (zoals de meeste regels in dit boek) als het eerste woord op de meest linkse positie van de regel begint, het laatste woord op de meest rechtse positie van de regel eindigt, de woorden door een of meer spaties van elkaar te scheiden zijn, en de spaties zo evenwichtig mogelijk over de ruimten tussen de woorden zijn verdeeld. De laatste regel hoeft niet naar rechts te worden uitgelijnd. Ook een regel bestaande uit een enkel lang woord kan niet naar rechts worden uitgelijnd.

Chapter 9

Bestanden

Computers danken hun snelle ontwikkeling niet zozeer aan het feit dat zij kunnen rekenen alswel aan het feit dat zij een economische oplossing geven voor belangrijke administratie-problemen. De administratieve automatisering brengt het grote geld in, de zuivere rekentoepassingen, zelfs de militaire, zouden de grote investeringen in de laatste 30 jaar niet hebben kunnen rechtvaardigen. De opslag en verwerking van grote hoeveelheden gegevens staan daarbij centraal.

Denk eens aan de administratie van reserveonderdelen bij een groot productiebedrijf. Er moet over duizenden soorten reserveonderdelen ieder een bepaalde hoeveelheid informatie worden opgeslagen: naam van het artikel, nummer, aantal, prijs per stuk. Al deze gegevens moeten niet alleen tijdens de uitvoering van één programma, maar over langere perioden bij uitvoering van vele programma's bewaard blijven. De invoer en verwerking vinden in het algemeen door verschillende programma's op verschillende tijden plaats, waarbij via meerdere terminals allerlei veranderingen, eventueel ook gelijktijdig (niet alleen na elkaar) in dezelfde verzameling gegevens moeten worden aangebracht. Zo'n stelsel van samenwerkende programma's, ingebed in een bedrijf of organisatie waarvan het (een deel van) de informatievoorziening verzorgt, noemt men een **informatie systeem**.

Dit vereist een heel andere vorm van gegevensopslag dan wij tot nog toe hebben gezien.

9.1 Enige begrippen

We zullen hier niet de fysieke opslagmethoden en de beheers- en programmeermethoden behandelen, die voor de opslag en verwerking van grote gegevensverzamelingen zijn ontwikkeld. We geven slechts enkele essentiële begrippen en wat eenvoudige taalmiddelen waarmee een eenvoudig informatiesysteem kan worden gemaakt. Een **bestand** (**file**) is een opgeslagen verzameling gegevens, voorzien van een **bestand naam** ("file-naam"). Het begrip bestand stelt ons in staat te abstraheren van de fysieke eigenschappen van bepaalde opslagmedia (vroeger ponskaarten of ponsband, magnetische band of magnetische trommels; tegenwoordig magnetische of optische schijven).

De gegevens worden opgeslagen in de vorm van *regels* ("records") tekst. Die regels denken we op een bepaalde manier onderverdeeld in **velden** die ieder een getal of tekst bevatten (voor een reserveonderdeel bijvoorbeeld: de getallen *nummer*, *aantal* en *prijs per stuk* gevolgd door een tekst van maximaal 40 tekens, de *naam*). Verschillende programma's kunnen de regels van eenzelfde bestand natuurlijk op verschillende wijze in velden onderverdelen, maar dat moet met grote omzichtigheid gebeuren. Zo kan een regel die 3 getallen bevat ook in zijn geheel als een tekst beschouwd worden, maar dat maakt het moeilijk nog aan de 3 getallen te komen.

Middels de inleesprocedures worden de velden als waarden ingelezen. Ook kunnen waarden door middel van uitvoerprocedures naar een bestand worden weggeschreven. Bestanden kunnen worden bewaard of gewist, zij kunnen een andere naam krijgen en hun inhoud kan telkenmale weer gewijzigd worden.

9.2 Eenvoudige bestandsoperaties

Elan kent een concreet bestandstype `FILE` met een volledig stel operaties dat representatief is voor de omgang met bestanden in programmeertalen. Wij voeren eerst een simpel stel operaties in, voldoende voor een kennismaking met bestanden en voor het begrijpen van de wezenlijke aspecten daarvan. Een beschrijving van het volledige standaardpakket bestandsorganisatie vindt U in sectie B.6.1 achteraan in dit boek.

9.2.1 Openen en sluiten van bestanden

De computer kent op ieder moment een aantal bestanden. Ieder bestand heeft een tekst als naam. Om een bestand vanuit een programma te kunnen aanspreken moeten we het eerst *openen*, en daarbij opgeven wat voor bestand we nodig hebben (nieuw of oud, om te lezen of te schrijven) en welke naam het heeft. Tot slot moeten we het bestand weer vrijgeven voor andere programma's door het te *sluiten*.

Er kan in dit simpele systeem maar één bestand tegelijk geopend zijn. Deze beperking zorgt ervoor dat de algoritmen die daarvan gebruik maken (voorbeelden in dit hoofdstuk, dan wel door u te schrijven) zich meer op de interne gegevensstructuren (rijen) richten dan op de gegevensstructuren op bestanden. Zulke bestandsstructuren zijn zeker wel interessant, maar worden pas later [P3dictaat] behandeld.

Wij hebben een aantal algoritmen ter beschikking, waarvan we de naam en de parameters zullen geven, door middel van een procedure-kop. Hoewel de procedures pas in het volgend hoofdstuk worden behandeld hopen we dat de gebruikte notatie voor zich spreekt.

a) `PROC new file (TEXT name):`
Een leeg bestand met de naam `name` wordt geopend voor schrijven, wat wil zeggen dat we er wel op mogen schrijven maar niet eruit mogen lezen. Het schrijven zal beginnen bij de eerste regel. Als er al een bestand met die naam bestaat dan wordt het oude eerst gewist (pas op!).

b) `PROC old file (TEXT name):`
Als er al een bestand met de naam `name` bestaat dan wordt dit geopend voor lezen, te beginnen bij de eerste regel. Bestaat zo'n bestand niet dan wordt de uitvoering van het programma afgebroken.

c) `PROC close file:`
Alle bestanden die geopend waren worden gesloten. (Ja, `close file` is een bizarre naam). Hierna kunnen dezelfde of andere bestanden worden geopend met de vorige procedures. Het gesloten bestand blijft bewaard tot het wordt gewist.

d) `PROC erase file (TEXT name):`
Als er een bestand met de naam `name` bestaat dan wordt dit gewist, anders gebeurt er niets. Hiermee kunnen we het aantal bestanden dat de computer moet opslaan binnen de perken houden.

Heb je een bestand geopend en vergeet je het te sluiten dan wordt het bij het beëindigen van het programma automatisch gesloten.

9.2.2 Schrijven naar een bestand

Er kan alleen worden geschreven naar een voor schrijven geopend bestand.

De hierna volgende algoritmen schrijven op de huidige regel van het bestand vanaf de huidige positie.

e) `PROC write (INT x):`
Van de waarde van `x` wordt een denotatie geschreven, eventueel voorafgegaan door een voorteken.

f) `PROC write (TEXT x):`
De tekens van de waarde van `x` worden achter elkaar geschreven. Er worden dus, net als bij de procedure `put`, geen quotes aan weerszijden afgedrukt, wat wel een fraai schriftbeeld oplevert, maar zorgt voor enige ingewikkeldheid bij het teruglezen.

g) `PROC writeline:`
Wordt een regel bij schrijven te vol dan wordt er vanzelf op een nieuwe regel overgegaan. Met een oproep van `writeline` kan eenzelfde overgang op een nieuwe regel van het uitvoerbestand geforceerd worden.

9.2.3 Lezen van een bestand

Er kan alleen worden gelezen van een voor lezen geopend bestand.

Bij het lezen van een bestand is het mogelijk iets dat als een getal was weggeschreven als een tekst terug te lezen en andersom, er moet dus met enige omzichtigheid te werk worden gegaan. Bovendien mag er niet voorbij het einde van het bestand worden gelezen. Gebeurt dit wel dan wordt de uitvoering van het programma afgebroken.

h) `PROC read (INT VAR x):`
Er wordt een integer-denotatie met eventueel voorteken en voorafgaande layout (spaties, overgang op nieuwe regel) gelezen. De waarde daarvan wordt aan `x` toegekend.

i) `PROC read (TEXT VAR x):`
De rest van de regel wordt gelezen en als tekst aan `x` toegewezen. Daarna wordt overgegaan op de volgende regel van het bestand. Bij het lezen gebeurt een overgang op een nieuwe bestandsregel altijd automatisch, zodat daar geen aparte procedure voor bestaat.

j) `BOOL PROC file ended:`
Geeft antwoord op de vraag of het einde van het invoerbestand bereikt is. Wie daar tijdig op test heeft geen andere conventies nodig om dit einde aan te geven.

Merk op dat in het algemeen een regel nul of meer getallen gevolgd door een eventuele tekst kan bevatten. Keren we de volgorde om (tekst gevolgd door getal) dan wordt dit getal bij het lezen tot de tekst gerekend en dus niet apart verwerkt, wat tot onvoorziene gedragingen van het programma kan voeren.

9.3 Editeren

Naast het invoeren van gegevens komt het wijzigen daarvan in allerlei informatiesystemen zo vaak van pas dat we daarvoor een speciale operatie zullen invoeren die uitstekend met de bestandsoperaties kan samenspelen. Het is een wat ingewikkeld geval:

k) `PROC edit (TEXT VAR t, INT p):`
De eerste parameter van `edit` is een tekstvariabele; de waarde daarvan wordt op het beeldscherm afgedrukt en vervolgens wordt de gelegenheid geboden de tekst op het scherm te wijzigen. De aldus gewijzigde tekst wordt tot slot weer aan de met `t` corresponderende actuele parameter toegewezen.
De constante `p` (met `1 <= p <= LENGTH t`) geeft een drempelpositie aan. Links daarvan kan de cursor niet bewogen worden. Kiest men `p = 1` dan komt dus de hele tekst voor wijziging in aanmerking.

Met behulp van `edit` is het kinderspel om eenvoudige informatiesystemen zoals het volgende voorbeeld te realiseren.

9.4 Voorbeeld: Adreslijst bijhouden

Ik heb een zakboek waarin ik de namen, adressen en telefoonnummers van een groot aantal personen bijhoud.

Zo nu en dan voeg ik een persoon toe, of wijzig het adres of telefoonnummer bij een bepaalde naam. In de loop van de tijd maken deze wijzigingen zo'n knoeiboel van mijn zakboek dat ik een hele avond opoffer om het oude zakboek in een nieuw over te schrijven, opnieuw keurig op alfabet en zonder kriebelige tussenvoegingen.

Wij zullen een programma schrijven om een zakboek netjes bij te houden. We nemen daarbij aan dat het zakboek op een bestand staat dat zo nu en dan – door het hier volgende programma – wordt aangepast.

Dit wordt tevens het grootste voorbeeld van Top-Down programmering dat in dit boek voorkomt.

9.4.1 Gebruikersinterface

Eerst ontwerpen we het gedrag van het programma ten opzichte van de gebruiker en de daartoe door de gebruiker te geven commando's. Het is van het grootste belang dat dit **gebruikers interface** goed ontworpen is zodat de gebruiker er gemakkelijk mee kan omgaan, zich een goede voorstelling van het werken met het systeem kan maken en niet voor plotselinge verrassingen komt te staan.

9.4.1.1 Fasen

We onderscheiden 3 fasen bij de uitvoering van het programma. Aan het begin van het programma wordt het huidige zakboekje van zijn bestand ingelezen. Gedurende de tweede fase van de uitvoering van het programma staat een copie van het hele zakboek in geheugen, waarop allerlei wijzigingen kunnen worden aangebracht. Tot slot wordt het resultaat van deze wijzigingen als een nieuw bestand weggeschreven. Heeft dit nieuwe bestand dezelfde naam als het oude dan wordt eerst het oude bestand verwijderd.

De wijzigingen worden dus niet direct op het bestand aangebracht. Deze gang van zaken brengt mee dat het oude zakboek in ongewijzigde vorm ter beschikking blijft en biedt zelfs de mogelijkheid tijdens uitvoering van het programma "spijt" te krijgen en de programmauitvoering af te breken, zodat het nieuwe bestand niet wordt weggeschreven.

9.4.1.2 Gegevens per persoon

Wij gebruiken een zeer eenvoudige bestandsstructuur: 3 regels per persoon, bevattend naam, eerste deel adres en tweede deel adres. Meer detail leggen we niet vast. Het staat bijvoorbeeld vrij het telefoonnummer in de tweede regel op te nemen, maar we treffen daartoe geen bijzondere maatregelen.

9.4.1.3 Commando's

De gebruiker stuurt het programma door het geven van commando's. De commando's zijn een letter lang, gevolgd door het indrukken van de RETURN-toets en het intypen van de eventuele informatievelden. Is het bestand eenmaal ingelezen dan kunnen we de volgende commando's geven:

- `r` een persoon toevoegen. Verlangt een naam, adr1 en adr2, ieder op een eigen regel. Is een persoon met die naam al bekend dan worden zijn gegevens overschreven, anders worden de nieuwe persoonsgegevens op de juiste plaats in de alfabetische lijst opgenomen.
- `d` een persoon verwijderen. Verlangt een naam. Is een persoon met die naam in de lijst al bekend dan wordt hij verwijderd, anders gebeurt er niets.
- `s` een persoon laten zien. Verlangt een naam. Is een persoon met die naam aanwezig dan worden zijn gegevens vertoond, anders gebeurt er niets.
- `e` wijzigen van gegevens. Verlangt een bekende naam, zoekt daarbij adr1 en adr2 en geeft de gelegenheid deze te wijzigen. Is de opgegeven naam onbekend dan gebeurt er niets.
- `l` lijst maken. Geeft op het beeldscherm een alfabetische adreslijst.
- `q` stoppen. Einde van de dialoog, overgang naar de derde fase, waarin de resulterende alfabetische adreslijst na verwijdering van het invoerbestand onder dezelfde naam wordt weggeschreven.

9.4.2 Het programma

De drie fasen van uitvoering zijn in de wortel van het programma terug te vinden.

```
program:
  definieer commandos;
  lees oude bestand;
  toon alle namen;
  REP
    vraag commando;
    voer commando uit
  UNTIL commando = ophouden
  ENDREP;
  schrijf nieuw bestand.
```

Het programma heeft de structuur van een eenvoudige vertolker (zie **??**).

```
definieer commandos:
  TEXT CONST lezen ::"r";
  TEXT CONST uitvegen ::"d";
  TEXT CONST tonen ::"s";
  TEXT CONST wijzigen ::"e";
  TEXT CONST lijsten ::"l";
  TEXT CONST ophouden ::"q" .
```

De verschillende commando's worden hier bij elkaar als tekstconstanten ingevoerd, waardoor het kiezen van andere namen daarvoor heel eenvoudig is.

9.4.2.1 Inlezen van het oude bestand

We lezen het oude bestand met behulp van de in dit hoofdstuk ingevoerde bestandsoperaties en slaan de regels daarvan op

```
lees oude bestand:
  LET max = 100;
  INT VAR aantal :: 0;
  ROW max TEXT VAR naamrij;
  ROW max TEXT VAR adr1rij;
  ROW max TEXT VAR adr2rij;
  TEXT CONST bestand ::"zakboek";
  old file (bestand);
  REP
    lees persoon;
    voeg toe;
  UNTIL file ended
  ENDREP;
  close file .
```

Gemakshalve geven we het bestand de vaste naam "zakboek" (we hadden ter verhoging van de flexibiliteit de bestandsnaam ook kunnen inlezen). Door wijziging van de waarde van `max` kunnen we ook grotere zakboeken toestaan.

```
lees persoon:
  lees naam;
  lees adres1;
  lees adres2 .
```

De gegevens van een persoon bestaan uit 3 regels die we afzonderlijk lezen.

```
lees naam:
  TEXT VAR naam;
  read (naam) .

lees adres1:
  TEXT VAR adres1;
  read (adres1) .

lees adres2:
  TEXT VAR adres2;
  read (adres2) .
```

9.4.2.2 Toevoegen van een persoon

We houden de lijst van namen gesorteerd in het geheugen. Daarom kunnen we een nieuwe persoon niet eenvoudig aan het einde van de lijst toevoegen maar moeten hem ergens in de lijst invoegen (vgl. 8.7.2, Insertion Sort). Eerst zoeken we zijn index, d.w.z. de plaats in de naamtabel waar een persoon met deze naam reeds staat dan wel had moeten staan volgens de alfabetisch-lexicografische ordening.

```
voeg toe:
  zoek index;
  IF reeds bekend
  THEN overschrijf
  ELSE
    schuif omhoog vanaf index;
    overschrijf
  FI .
```

```
zoek index:
  INT VAR index :: 1;
  WHILE index <= aantal
  REP
    IF naamrij [ index ] >= naam
    THEN LEAVE zoek index
    FI;
    index INCR 1
  ENDREP .
```

We gebruiken hier de meest eenvoudige vorm van lineair zoeken. De lezer mag ter verhoging van de efficiency een vorm van binair zoeken verkiezen (zie 8.8).

```
reeds bekend:
  IF index > aantal
  THEN false
  ELSE naamrij [ index ] = naam
  FI.
```

Deze voorwaarde is nogal "tricky" omdat rekening gehouden moet worden met toevoeging achteraan de lijst.

```
overschrijf:
  naamrij [ index ] := naam;
  adr1rij [ index ] := adres1;
  adr2rij [ index ] := adres2 .

schuif omhoog vanaf index:
  INT VAR i :: aantal;
  IF i = max
  THEN zakboek vol
  ELSE
    FOR i FROM aantal DOWNTO index
    REP schuif een omhoog
    ENDREP
  FI;
  aantal INCR 1 .

schuif een omhoog:
  naamrij [ i + 1 ] := naamrij [ i ];
  adr1rij [ i + 1 ] := adr1rij [ i ];
  adr2rij [ i + 1 ] := adr2rij [ i ] .

zakboek vol:
  put line ("Zakboek vol..." );
  LEAVE program.
```

9.4.2.3 Verwerking van een commando

```
vraag commando:
  TEXT VAR commando;
  REP
    line (2);
    put ("Commando = " );
    TEXT VAR regel;
    get (regel);
    commando := regel SUB 1
  UNTIL commando >="a" AND commando <="z"
  ENDREP.
```

```
voer commando uit:
  IF commando = lezen
  THEN
    vraag persoon;
    voeg toe
  ELIF commando = uitvegen
  THEN
    vraag bekende naam;
    verwijder
  ELIF commando = tonen
  THEN
    vraag bekende naam;
    toon persoon
  ELIF commando = wijzigen
  THEN
    vraag bekende naam;
    wijzig adres1;
    wijzig adres2
  ELIF commando = lijsten
  THEN toon alle namen
  ELIF commando = ophouden
  THEN
  ELSE
    put line ("Toegestane commando's:");
    put (" " + lezen + " " + uitvegen
       + " " + tonen);
    put (" " + wijzigen + " " + lijsten
       + " " + ophouden)
  FI .
```

9.4.2.4 Vragen om gegevens

Wij zullen namen als “vraag ...” hanteren voor de algoritmen die interactief (d.w.z. niet van een bestand maar via toetsenbord en beeldscherm direct van de gebruiker van het programma) gegevens lezen.

```
vraag persoon:
  vraag naam;
  vraag adres1;
  vraag adres2 .

vraag naam:
  put line ("Naam:" );
  get (naam) .

vraag adres1:
  put line ("Adres1:" );
  get (adres1) .

vraag adres2:
  put line ("Adres2:" );
  get (adres2) .

vraag bekende naam:
  vraag naam;
  zoek index;
  IF NOT reeds bekend
  THEN
    LEAVE voer commando uit
  FI .
```

9.4.2.5 Wijzigen van gegevens

Voor het wijzigen van gegevens gebruiken we de procedure `edit`.

```
wijzig adres1:
  put line ("Adres1:");
  edit (adr1rij[index], 1) .

wijzig adres2:
  put line ("Adres2:");
  edit (adr2rij[index], 1) .
```

Op dezelfde wijze kunnen we het ook mogelijk maken dat de naam gewijzigd wordt, alleen moeten we dan wel de oude persoon uitwissen en de nieuwe persoon toevoegen.

9.4.2.6 Verwijderen van personen

```
verwijder:
  zoek index;
  IF reeds bekend
  THEN schuif omlaag tot index
  FI .

schuif omlaag tot index:
  INT VAR j;
  FOR j FROM index UPTO aantal - 1
  REP schuif een omlaag
  ENDREP;
  aantal DECR 1 .

schuif een omlaag:
  naamrij [ j ] := naamrij [ j + 1 ];
  adr1rij [ j ] := adr1rij [ j + 1 ];
  adr2rij [ j ] := adr2rij [ j + 1 ] .
```

9.4.2.7 Tonen van gegevens

```
toon alle namen:
  FOR index FROM 1 UPTO aantal
  REP
    toon naam
  ENDREP .

toon naam:
  line;
  tab;
  put (naamrij [ index ] ) .

toon persoon:
  line;
  tab;
  put (naamrij [ index ] );
  line;
  tab;
  put (adr1rij [ index ] );
  line;
  tab;
  put (adr2rij [ index ] ) .

tab:
  put ("        " ) .
```

9.4.2.8 Schrijven van het nieuw bestand

```
schrijf nieuw bestand :
  new file (bestand);
  schrijf alle personen;
  close file .

schrijf alle personen:
  FOR index FROM 1 UPTO aantal
  REP
    schrijf persoon
  ENDREP .

schrijf persoon:
  write ( naamrij [ index ] );
  writeline;
  write ( adr1rij [ index ] );
  writeline;
  write ( adr2rij [ index ] );
  writeline .
```

9.4.3 Het eerste zakboek

We hebben nu een prachtig programma ontworpen, typen het in en bedenken dan plotseling dat het één merkwaardige eigenschap heeft: het veronderstelt de aanwezigheid van een bestand dat we willen wijzigen. Dit bestand moet al bestaan voor uitvoering van het programma, en mag niet leeg zijn. Hoe komen we daaraan?

De eenvoudigste weg is nog een programma te maken, dat een bestand met minstens een persoon schrijft. Hebben we dat bestand eenmaal, dan is dat opstartprogramma niet meer nodig. We willen een bestand hebben met de volgende personen erin:

We kiezen er één van uit, en schrijven het volgende programma:

```
start database:
  new file ("zakboek");
  write ("Tiel, Frans van");
  writeline;
  write ("Begoniaweg 13");
  writeline;
  write ("6525 ED  Nijmegen");
  closefile.
```

Een leeg bestand was natuurlijk makkelijker geweest om aan te maken (alleen de eerste en de laatste unit zijn dan nodig) maar het programma zoals boven gegeven verlangt nu eenmaal minstens één persoon.

9.4.4 Gebruik van het programma

We starten het programma en zien na enige tijd de volgende **prompt** (engels voor "aansporing") verschijnen:

```
Commando:
```

Onmiddellijk hierachter typen we een `r` en sluiten de regel af met de RETURN- of ENTER-toets.

Er verschijnt een nieuwe prompt

```
Naam:
```

Waarop wij intypen

```
Leeuwen, Hans van
```

Dan volgt de prompt voor het eerste deel van het adres. We voeren ook de beide adresvelden in en krijgen dan weer de prompt

```
Commando:
```

Nieuwsgierig naar het resultaat van ons werk geven we een `l` en krijgen te zien

```
Leeuwen, Hans van
Tiel, Frans van
```

We voeren met een aantal `r`-commando's de overige personen in en geven nog eens `l`, waarop we een alfabetisch overzicht over alle personen krijgen.

```
Andelst, Harry van
Gendt, Heino van
Leeuwen, Hans van
Ophemert, Marco van
Tiel, Frans van
```

Nu komt Hans van Leeuwen binnen, en vraagt wat we over hem hebben opgeslagen. We geven een `s` gevolgd door

```
Leeuwen, Hans van
```

(Het is wel erg lang, misschien hadden we beter een conventie kunnen verzinnen voor het zoeken naar afgekorte namen als `Lee*`) en zien zijn gegevens verschijnen. We geven een `q`, waarop het nieuwe zakboek wordt weggeschreven en het programma eindigt.

Het gebruik van de andere commando's zullen we niet beschrijven, u probeert ze zelf maar.

Het gegeven programma is vrij lang maar hopelijk volkomen doorzichtig. Het is het grootste Top-Down programma dat in dit boek voorkomt. Het kan overigens model staan voor allerlei soorten informatiesystemen met eenzelfde structuur.

9.5 Opgaven

1. (Bestandsbeheer) Schrijf een programma om je bestanden te beheren, dat in ieder geval het volgende kan:
   - (wissen) vraag een naam, wis het bestand met die naam (als het bestaat)
   - (copiëren) vraag twee bestandsnamen, `van` en `naar`. Lees de regels van het bestand `van` in het geheugen en schrijf ze daarna naar het bestand `naar`.
   - (printen) Vraag een bestandsnaam en copieer het bestand met die naam (als het bestaat) naar het bestand `lp`.
2. (Ledenadministratie) Ben je lid van een club? Probeer dan te bedenken hoe deze een overzicht over zijn leden kan houden. Per lid is o.a. van belang: adres, lidmaatschapsnummer, laatst betaalde jaar van contributie, andere gegevens die van de soort club afhangen. Zorg dat in ieder geval de volgende rapportages gemaakt kunnen worden:
   - complete ledenlijst op alfabetische volgorde
   - idem op volgorde van nummer, met adressen, geschikt om verknipt te worden tot adreslabels

Tiel, Frans van	Begoniaweg 13	6525 ED Nijmegen
Leeuwen, Hans van	Uithof 67	6531 AT Nijmegen
Ophemert, Marco van	Doeckenborgh 43-21	6530 AV Nijmegen
Kesteren, Heino van	Brink 2	6720 DC Bemmel
Andelst, Harry van	Dorpsstraat 43	6421 HL Wychen

Table 9.1: Klein adresboekje

- lijst van wanbetalers.

3. (Schooladministratie) Maak een administratie voor een school, tenminste omvattend
   - van iedere leerling de vakken met resultaten
   - van iedere klas de leerlingen.

   Welke wijzigingen moet dit bestand kunnen ondergaan? Welke vragen moeten er aan kunnen worden gesteld?
4. (Girosysteem) Ontwerp en realiseer een simpel girosysteem dat naam, adres, rekeningnummer en saldo van de klanten bijhoudt, overboekingen verzorgt, dagafschriften stuurt en periodieke renteberekeningen maakt.
5. (Plaatsreserveren) Ontwerp en realiseer een simpel plaatsreserveringssysteem voor vliegtuigreizigers, dat per vliegtuig de klanten en de vrije plaatsen bijhoudt en boekingen verricht voor (groepen) klanten. Dit systeem kan willekeurig gecompliceerd worden gemaakt door het toevoegen van allerlei aspecten van de realiteit (roken/niet roken, wachtlijsten, alternatieve routes, klokverschillen ...).

æ

Chapter 10

Procedures

Als middel om abstracte algoritmen te definiëren hebben we tot nog toe de verfijningen gebruikt. Dit mechanisme is van groot belang bij het aanleren van een systematische programmeerwijze, maar wordt door verreweg de meeste programmeertalen niet in die vorm aangeboden. In plaats daarvan kennen deze talen als abstractiemiddel voor algoritmen de *procedure* ("subroutine", "subprogram", "function"). ELAN kent zowel verfijningen als procedures, ieder met een bijbehorende programmeerstijl.

De **procedure** is net als de verfijning een middel om aan een stuk programmatekst een naam te geven, die dan in de plaats van dat stuk tekst gebruikt kan worden. Het doel van dit mechanisme is in eerste instantie vereenvoudiging van het programmeren door verkorting van de programmatekst en vermindering van de kans, bij het herhalen van stukken tekst fouten te introduceren. Daardoor kunnen beide mechanismen gebruikt worden als abstractiemiddel voor algoritmen bij de Top-Down programmering.

Procedures onderscheiden zich van verfijningen in de eerste plaats door de mogelijkheid ze te voorzien van **parameters**, die bij hun oproep worden meegegeven. Geparametriseerde bouwstenen zijn in zekere mate aan te passen aan de behoefte, en daardoor veel flexibeler te gebruiken dan ongeparametriseerde. Daarom lenen procedures zich voor een andere programmeerstijl (**Bottom-Up programmering**) waarvoor verfijningen beduidend minder geschikt zijn.

Hoewel we in dit boek nog niet op de Bottom-Up programmeerstijl ingaan zijn de volgende hoofdstukken daarop wel een voorbereiding.

10.1 Verfijningen: een terugblik

Mensen hebben nu eenmaal grote moeite met het overzien van ingewikkelde opgaven. Om aan deze menselijke zwakte tegemoet te komen proberen we niet een ingewikkelde taak in een klap op te lossen maar stellen ons er in eerste instantie mee tevreden, deze taak in kleinere deeltaken uit elkaar te pluizen, waarvan we het gevoel hebben dat ze gemakkelijker zijn op te lossen. Vervolgens herhalen we dit **decompositie**proces op ieder van de deeltaken totdat we aangeland zijn bij taken die klein genoeg zijn om ze in een keer op te lossen.

Op dezelfde wijze gaan we te werk bij de **top-down programmering**: we willen een abstracte algoritme construeren die een gegeven probleem oplost. Daartoe bedenken we een aantal abstracte algoritmen die deelproblemen oplossen, die we met besturingsstructuren en andere taalmiddelen samenstellen tot een realisering van de bedoelde algoritme.

Daarna staan we voor de taak, ieder van die abstracte algoritmen te realiseren in termen van (andere) abstracte algoritmen en, in laatste instantie, van de concrete algoritmen van de te gebruiken programmeertaal.

Voor het definiëren van de abstracte algoritmen die ontstaan als tussenstadium van dit proces gebruiken we de **verfijningen**.

10.1.1 Voorbeeld: beschrijving van een arbeidsproces

We willen een algoritme construeren die beschrijft hoe men een lek in de voorband van een fiets kan plakken. We nemen aan dat we beschikken over een eenvoudige maar hooggespecialiseerde processor, die fietsen en hun onderdelen manipuleren kan. Het voorbeeld is in zoverre realistisch dat het als programma voor een soort van robot best voorstelbaar is.

Op het hoogste abstractieniveau beschouwen we de fiets als bestaande uit enkele grote onderdelen, waaronder de wielen. Dat zijn de objecten waar we over kunnen praten. Op dit niveau kunnen we aangeven wat we bedoelen met `plak voorband van fiets`.

```
plak voorband van fiets:
  draai de fiets om;
  verwijder het voorwiel;
  plak de voorband;
  bevestig het voorwiel;
  draai de fiets om.
```

Dit is een complete zij het niet bijzonder gedetailleerde algoritme. Hij is ook vast niet onjuist want er staat nog niets verkeerds in. Het "enige" dat we nog moeten doen is het realiseren van de abstracte algoritmen die hier gebruikt werden.

We noemen zo'n formulering op het hoogste niveau van abstractie de **wortel** van de algoritme. Belangrijk is dat de wortel de wezenlijke trekken van de oplossing aangeeft, anders hebben we nog niets bereikt. Daarbij nemen we echter de vrijheid, naar believen bepaalde nuttige abstracte algoritmen te postuleren ook als ze niet tot het concrete repertoire van enigerlei processor behoren. Daarna gaan we deze door middel van verfijningen concretiseren.

Nemen we bijvoorbeeld de abstracte algoritme met de suggestieve naam `verwijder het voorwiel`. De naam drukt op min of meer adequate wijze uit *wat* er gedaan moet worden. In de verfijning moeten we nu aangeven *hoe* dat gedaan wordt. Deze realisering drukken we uit in termen van een lager abstractieniveau, waar we het voorwiel op zijn beurt als samengesteld uit een naaf met moeren, spaken, velg, binnen- en buitenband opvatten.

```
verwijder het voorwiel:
  verwijder de linker voorwielmoer;
  verwijder de rechter voorwielmoer;
  neem het voorwiel uit de vork.

bevestig het voorwiel:
  steek het voorwiel in de vork;
  bevestig de rechter voorwielmoer;
  bevestig de linker voorwielmoer.
```

Ook hier hebben wij weer van vele zaken geabstraheerd. We hebben bijvoorbeeld helemaal niet vermeld, waar de moeren bewaard worden. De beslissing daarover is bestanddeel van de verfijning van (onder andere) `verwijder de linker voorwielmoer`.

Een detailbeslissing die we bij de aangegeven realisering hebben getroffen betreft de vraag, in welke volgorde de moeren los- en vastgeschroefd worden. We hebben vastgelegd dat eerst de linker en pas daarna de rechter voorwielmoer verwijderd wordt. De volgorde had net zo goed andersom kunnen zijn omdat het resultaat daarvan niet afhankelijk is. Wel worden door deze beslissing weer andere details bepaald. Zo ligt het voor de hand hieruit op te maken dat de processor zich tussen het verwijderen van de moeren en het weer bevestigen daarvan aan de rechterzijde van de fiets bevindt.

De vastlegging "eerst links, dan rechts" is een voorbeeld van een **overspecificatie**. Om de algemeenheid zo groot mogelijk te houden hadden we een collaterale demontage moeten aangeven, zodat de een of andere octopus beide moeren zelfs gelijktijdig had kunnen verwijderen. Maar wij beperken ons tot sequentiële algoritmen en hebben geen notatie ter beschikking om collaterale uitvoering te beschrijven. Overspecificaties als deze treden bij het programmeren voortdurend op: telkens zien we ons gedwongen een keuze te maken die hele klassen van net zo goede of zelfs betere realiseringen overboord gooit.

We hadden de beslissing nog een verfijningsstap kunnen uitstellen door invoering van een abstracte algoritme `verwijder beide moeren` maar dat was ook slechts uitstel geweest.

Van deze fenomenen moet men zich bij het programmeren voortdurend bewust zijn en men moet proberen concrete keuzes voor een bepaalde strategie zo laat mogelijk te treffen, zodat men geen spijt krijgt van overijlde beslissingen. Van de andere kant moet men zo gauw dat niet meer te vermijden is de knoop welbewust doorhakken. Het verdient hierbij aanbeveling eerst de abstracte algoritmen die op eenzelfde abstractieniveau horen, te realiseren alvorens met het volgende abstractieniveau te beginnen.

We verfijnen verder:

```
plak de voorband:
  verwijder de buitenband;
  haal de binnenband eruit;
  plak de binnenband;
  stop de binnenband erin;
  breng de buitenband aan.
```

Bij het plakken van de binnenband moeten we nog gedetailleerder te werk gaan, zodat nu ook het gat tot ons abstractieniveau behoort. Het kan namelijk zijn dat er meer dan een gat is, of dat we ons vergist hebben en er helemaal geen gat te vinden is.

```
plak de binnenband:
  WHILE er zit nog een gat
  REP plak dat gat dicht
  ENDREP.
```

En zo moeten we verder verfijnen tot we (hopelijk) het niveau van direct uitvoerbare concrete algoritmen bereiken.

De Top-Down methode die wij hier en in vele eerdere voorbeelden hebben toegepast staat ook bekend als de hiërarchische decompositie met behulp van verfijningen.

Door het systematisch gebruik van verfijningen met welgekozen namen houden we een groot deel van de kleine gedachten, ideeën en inzichten vast die bij het ontwerpproces een rol spelen. Bij een latere wijziging van het programma hoeft de bedoeling van de maker niet moeizaam uit zijn daden gereconstrueerd te worden (zie o.a. [MEE77]).

Verfijningen zijn een ondersteuning bij het programmeren, maar doen geen wonderen. Zij dienen ertoe om de voorbijgaande gedachten tijdens het programmeerproces te kunnen vastleggen. Zij geven een zeker ritme aan het programma. Zij beperken de afstand in abstractie die in één keer overbrugd moet worden. Zij beperken de horizon die de programmeur in een keer moet overzien. Zij stellen hem in staat het "wat" van het "hoe" te onderscheiden. Zij stellen een ander in staat, later de gedachtenkronkels van de auteur van een programma te kunnen volgen. Nogmaals — zij dienen als hulpmiddel voor het vastleggen van gedachten. Maar de juiste gedachten ingeven kunnen zij niet.

10.2 Procedures als bouwstenen

Bij de Top-Down programmering verfijnen we in principe tot op het niveau van de concrete uitdrukkingsmiddelen van de programmeertaal. Vooral bij het ontwikkelen van grotere programma's is dit vaak niet bepaald het meest conveniënte niveau om op te mikken. Je kunt je vaak heel wat werk besparen door tevoren een aantal welgekozen algoritmen en objecten te definiëren als laagste niveau voor dit bepaalde programma.

Voor dit doel zullen we een andere vorm van abstracte algoritme invoeren, de **procedure**, die we gebruiken als we de behoefte voelen aan een herbruikbare algoritme.

Als we een abstracte algoritme meer dan eenmaal in het programma willen gebruiken, dan beschouwen we hem niet langer als een tussenstadium in het denkproces maar als een aanvulling op de concrete algoritmen van de taal, als een nieuwe algemene elementaire bouwsteen. Zo hoort het omdraaien van de fiets, de omgang met linker- en rechter moeren van voor- en achterwiel, maar ook het dichtplakken van één gat, eigenlijk tot de basisvaardigheden van iemand die zijn eigen fiets repareert.

Bij wat grotere programmeeropgaven is het sterk de moeite waard eerst zorgvuldig een pakket van elementaire algoritmen, objecten en typen te ontwerpen, in termen waarvan de oplossing van het gestelde probleem eenvoudiger is dan bij een poging in één klap de hele afstand tussen gegeven probleem en gegeven taalmiddelen te overbruggen. Het consequent doorvoeren van dit idee leidt tot een andere programmeerstijl, de **Bottom-Up programmering** ("Methode van de stapsgewijze synthese") (zie bijv. [KLE81]), waarvoor ELAN een aantal bijzondere constructies kent (naast de procedure-declaratie het pakketmechanisme met interfaces, type-declaratie en operator-declaratie). In deel twee gaan we daar dieper op in. Natuurlijk is er een spectrum van mogelijkheden tussen Top-Down en Bottom-Up programmeerstijl, waarin de procedures een belangrijke rol spelen.

Abstracte algoritmen die het waard zijn als meermalen gebruikte elementaire bouwstenen in een programma te dienen, realiseren we als procedures.

Abstracte algoritmen die slechts een eenmalige, vluchtige, voorbijgaande rol spelen bij het ontwerpproces van de algoritme realiseren we als verfijningen.

Overigens hebben we in de voorafgaande hoofdstukken al stilzwijgend kennis gemaakt met de aanroep van procedures, omdat vele concrete algoritmen van ELAN procedures zijn. Zo is `put` een standaard-procedure, evenals `sin` en `cos`. In dit hoofdstuk zullen we aangeven hoe procedures gedeclareerd worden, hoe ze gebruikt worden en in welke soorten ze onderscheiden kunnen worden.

10.3 Procedure-declaraties

Procedures dienen, evenals andere objecten (variabelen en constanten), te worden gedeclareerd voordat ze kunnen worden gebruikt.

Zo'n declaratie ziet er als volgt uit:

Een voorbeeld van een *procedure-declaration* is:

```
REAL PROC gemiddelde (REAL a, b):
  (a + b) / 2.0
ENDPROC gemiddelde
```

Deze declaratie voert een functie van twee argumenten in, met de naam `gemiddelde`. Ieder van zijn delen (kop, romp en staart) staat in dit voorbeeld op een eigen regel. De eerste regel is dus de kop.

De uitvoering van een procedure-declaratie heeft als effect dat een **routine** (een waarde, die je je kunt voorstellen als een interne representatie voor de procedure) wordt verbonden met de naam van de procedure. Ook een procedure is dus een object, met een naam en een waarde. Zoals bij alle declaraties heeft de naam van de procedure geen waarde voordat de declaratie is uitgevoerd. Ook het geldigheidsgebied van een procedure-declaratie is hetzelfde als bij andere declaraties.

10.3.1 De kop van een procedure

In de kop van een procedure wordt een koppeling aangegeven tussen de omgeving waarin de procedure wordt uitgevoerd en de romp van de procedure. Deze koppeling wordt, zoals later beschreven, tot stand gebracht door de waarde van een object of uitdrukking uit de omgeving van de oproep (de **actuele parameter**) te binden aan een naam, die alleen binnen de procedure bekend (een **formele parameter**). Zo is bijvoorbeeld in de aanroep `sin (0.5 * x)` de uitdrukking `0.5 * x` de actuele parameter van `sin`. Hoe de daarmee corresponderende formele parameter heet kunnen we in de declaratie van `sin` opzoeken.

De syntax van de kop van de procedure vindt u in fig. 10.3.1. Het type van het eventuele resultaat van de procedure blijkt uit de type-declarer die aan `PROC` voorafgaat.

Een procedure kan als resultaat een waarde van een bepaald type afleveren, maar er zijn ook procedures die enkel een effect hebben. In het laatste geval blijft deze type-declarer achterwege. Het resultaat van de procedure in het voorbeeld is van het type `REAL`.

Het formal-parameter-pack geeft voor ieder van de formele parameters een **specificatie** van naam, type en accesrecht. Het pak formele parameters wordt weggelaten als de procedure geen parameters heeft. Het facultatieve accesrecht `CONST` wordt in de kop van een procedure meestal weggelaten.

In het voorbeeld definieert het pak formele parameters `(REAL a, b)` twee reële constanten `a` en `b` ten behoeve van de romp van `gemiddelde`.

Voor iedere formele parameter wordt bij de oproep van de algoritme een actuele parameter meegegeven, d.w.z. een object of uitdrukking waarvan de waarde aan de formele parameter wordt gebonden. De formal-parameter-list geeft het aantal, de typen en de volgorde van de verlangde parameters aan en geeft ze tevens een naam voor gebruik binnen de romp van de procedure.

10.3.2 Voorbeelden van headings

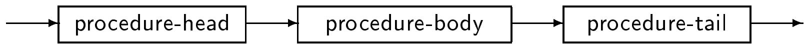
Met korte, pakkende namen kan de leesbaarheid van de kop verhoogd worden. Een juist gekozen kop geeft aan *wat* de procedure doet, zonder te zeggen *hoe* deze dat doet.

Om het verhaal niet al te droog te maken, zullen we eerst een aantal suggestieve voorbeelden van procedure-headings geven.

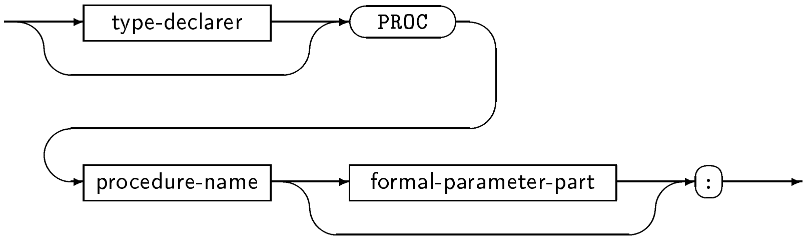
```
REAL PROC sin (REAL x):

REAL PROC max (REAL a, b):
```

procedure-declaration



procedure-head




```
BOOL PROC even (INT n):

TEXT PROC multipliceer (INT n, TEXT s):

     PROC put(INT x):

     PROC get(INT VAR x):
```

Vooruitlopend op de mogelijkheid zelf typen te declareren geven we ook wat voorbeelden van headings met formele parameters van sprekende abstracte typen.

```
REAL PROC nulpunt (FUNCTIE f,
                   INTERVAL i):

     PROC inverteer (MATRIX VAR a):

BOOL PROC zijn gelijk (ELEMENT a, b):

BOOL PROC bevat (VERZAMELING m,
                 ELEMENT x):

     PROC overboeken
              (REKENING VAR debiteur,
                   GELD bedrag):

     PROC voeg toe (VERZAMELING VAR m,
                    ELEMENT x):

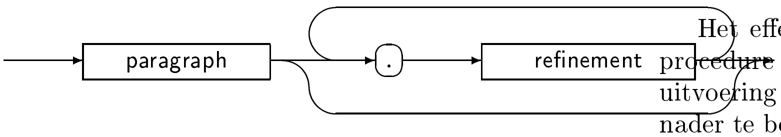
     PROC signaleer (TEXT mededeling):

     PROC stop :
```

10.3.3 De romp van de procedure

De romp van de procedure bestaat uit een paragraaf, eventueel gevolgd door zijn verfijningen.

procedure-body



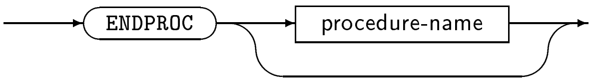
In dit voorbeeld is de romp van de procedure wel heel eenvoudig: de paragraaf bestaat uit een enkele unit. De romp mag ook een of meer verfijningen bevatten, zoals in het (nogal gekunstelde) voorbeeld

```
REAL PROC gemiddelde (REAL a, b):
  som der waarden / som der gewichten.
  som der waarden:
    a + b.
  som der gewichten:
    2.0.
ENDPROC gemiddelde;
```

10.3.4 De staart van de procedure

De naam van de procedure mag in zijn staart herhaald worden, een redundantie die het makkelijk maakt sommige structuurfouten (zoals verloren ENDPROC 's) adequaat te signaleren.

procedure-tail



Tussen END en PROC mag eventueel een spatie staan.

In de voorbeelden zullen wij achter de staart van de procedures steeds een puntkomma zetten, maar die hoort strikt genomen niet meer bij de procedure-declaratie: hij dient als scheider met de volgende declaratie.

10.3.5 De procedure-aanroep

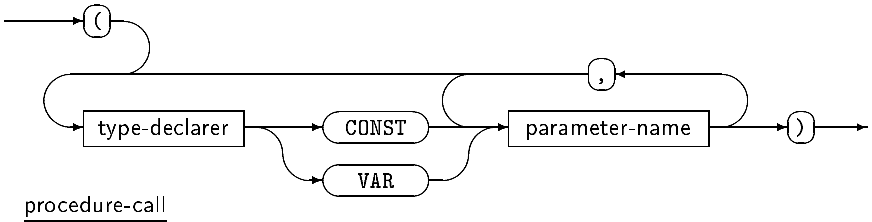
Wanneer een procedure eenmaal gedeclareerd is kan hij op verschillende plaatsen in een programma worden aangeroepen. Door een **procedure oproep** wordt eerst een koppeling van de actuele parameters aan de formele parameters van de oproep tot stand gebracht en daarna de romp van de procedure uitgevoerd. De syntax van de procedure-call luidt:

actual-parameter

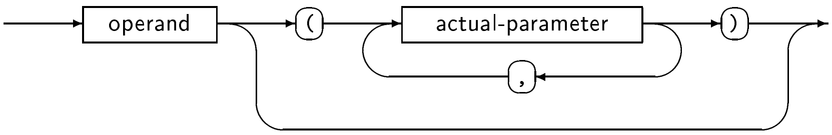
Het effect en het resultaat van de oproep van een procedure zijn resp. het effect en het resultaat van de uitvoering van zijn romp, rekening houdend met de hier nader te bespreken koppeling met de omgeving waarin de procedure aangeroepen wordt (het **parameter mechanisme**).

Zo levert de aanroep sin(0.2) als resultaat de real op die binnen de procedure sin berekend wordt.

formal-parameter-part



procedure-call



De waarden van de actuele parameters worden bij uitvoering van de procedure op de rij af gebonden aan de corresponderende formele parameters. Vervolgens wordt de romp uitgevoerd. De typen van de formele parameters moeten gelijk zijn aan die van de daarmee corresponderende actuele parameters. Heeft de formele parameter het accesrecht `VAR` dan moet ook de actuele parameter een `VAR` zijn, zodat er aan kan worden toegewezen.

We bespreken nu het parameter-mechanisme nader.

10.3.5.1 `CONST`*-parameters*

Een formele parameter met `CONST` acces-attribuut (d.w.z. een **formele constante**) krijgt bij aanroep van de procedure als **initialisatie** de waarde van de corresponderende actuele parameter.

Beschouw de aanroep `gemiddelde (1.3, 1.7)` van de bovengenoemde procedure. Deze levert dezelfde waarde als de paragraaf

```
REAL CONST a :: 1.3, b :: 1.7;
(a + b) / 2.0
```

bestaande uit: voor ieder van de formele parameters een declaratie met de waarde van de corresponderende actuele parameter als initialisatie, gevolgd door de romp van de procedure. De aanroep

```
put (gemiddelde (1.3, 1.7) )
```

schrijft dus het getal anderhalf.

10.3.5.2 `VAR`*-parameters*

Bij een **formele variabele** moet de corresponderende actuele parameter eveneens een variabele zijn. Bij ingaan van de procedure wordt de waarde van de actuele parameter als initialisatie voor de formele gebruikt; bij verlaten van de procedure wordt de danmalige waarde van de formele parameter toegewezen aan de actuele. Beschouw bijvoorbeeld:

```
PROC hoog op (INT VAR x, INT y):
  x := x + y
ENDPROC hoog op;

  INT VAR aantal :: 0;
  hoog op (aantal, 3)
```

De oproep heeft het effect van

```
INT VAR x :: aantal;
INT CONST y :: 3;
  x := x + y
aantal:= x
```

Het effect van de oproep is dus dat de variabele `aantal` met `3` wordt opgehoogd.

Het hier beschreven parameter-mechanisme (dat ELAN o.a. gemeenschappelijk heeft met ADA, maar niet met bijvoorbeeld C, wordt het **value/result m**echanisme genoemd.

10.3.6 Geldigheidsgebied van locale declaraties

Een procedure-declaratie maakt de naam van de procedure zichtbaar in het hele **geldigheidsgebied** van de declaratie, in ons geval het hele programma. De formele parameters van de procedure en alle objecten gedeclareerd binnen de romp van de procedure (**locale objecten**) hebben als geldigheidsgebied de procedure-declaratie zelf, en zijn daarbuiten niet zichtbaar. Dit betekent dat twee procedures locale objecten mogen hebben met dezelfde naam, zonder dat verwarring kan optreden. Zo zal in

```
PROC hoog op (INT VAR x, INT y):
  x := x + y
ENDPROC hoog op;
PROC laag af (INT VAR x, INT y):
  x := x - y
ENDPROC laag af;
```

ieder van de procedures zijn eigen parameter `x` hebben, die (afgezien van het feit dat ze toevallig dezelfde naam hebben) niets met elkaar te maken hebben.

Het feit dat een procedure locale objecten kan hebben is een van de wezenlijke verschillen met verfijningen. Declaraties gedaan binnen een verfijning zijn immers geldig in het hele programma!

De syntax van ELAN staat overigens niet toe dat een procedure zelf weer een locale procedure heeft, alle procedures staan dus naast elkaar in een globale omgeving.

10.3.7 Communicatie met de omgeving

Behalve via parameters kunnen procedures met hun omgeving en met andere procedures communiceren via invoer en uitvoer (traag en onhandig) of via **globale variabelen**.

10.3.7.1 Globale variabelen

Een object heet *globaal* ten opzichte van een procedure als het is gedeclareerd in de omgeving buiten de procedure. Via een globale variabele die zowel vanuit de omgeving die de oproep bevat als vanuit de procedure zelf zichtbaar is kan informatie worden meegegeven. Verschillende procedures kunnen, net als verfijningen, via variabelen uit hun omgeving waarden uitwisselen. Ook kan de omgeving via zo'n globale variabele het gedrag van de procedure beïnvloeden, zoals het volgende voorbeeld illustreert.

```
    INT CONST is sinus :: 1,
              is cosinus :: 2;
    INT VAR type :: is sinus;

  REAL PROC sinus of cosinus (REAL x):
    IF type = is sinus
    THEN sin (x)
    ELIF type = is cosinus
    THEN cos (x)
    ELSE
       put ("onbekend type functie");
       line;
       0.0
    FI
  ENDPROC sinus of cosinus;
```

De kameleon `sinus of cosinus` gedraagt zich als de sinus of de cosinus, al naar gelang de waarde van de globale variabele `type`.

Bij de communicatie via parameters kunnen we bij de aanroep expliciet zien welke variabelen door de procedure veranderd kunnen worden. We kunnen de procedure ook telkens met andere actuele parameters aanroepen.

Bij de communicatie via globale variabelen zijn de variabelen die door de procedure kunnen worden veranderd impliciet vastgelegd. Aan de aanroep van de procedure kunnen we niet zien welke globale variabelen veranderd kunnen worden.

Beide communicatie-technieken hebben hun toepassingen — het hangt van allerlei omstandigheden af, welke de voorkeur verdient. Wanneer een procedure in verschillende omgevingen opgeroepen moet kunnen worden en wanneer het resultaat van de procedure van die omgeving moet afhangen dan ligt het voor de hand om parameters te gebruiken.

10.3.7.2 Wanneer liever globaal?

Bij een stelsel van samenwerkende procedures die tesamen een bepaalde abstractie vormen en daartoe een zekere mate aan *gemeenschappelijk geheugen* nodig hebben is het wenselijk de communicatie tussen de procedures via globale variabelen te laten lopen. De communicatie gebeurt dan als het ware achter de schermen, omdat ze voor de omgeving waarin de procedure wordt aangeroepen niet relevant is; daar wordt van geabstraheerd.

Als je bemerkt of verwacht dat alle oproepen van een bepaalde procedure voorzien zijn van eenzelfde actuele parameter ligt het voor de hand deze **stereotype parameter** globaal te houden in plaats van hem telkens expliciet door te geven.

10.3.7.3 Naamconflicten

Wat gebeurt er als een formele parameter of locale variabele van een procedure dezelfde naam heeft als een actuele parameter? Volgens de regels in 10.3.6 kunnen de actuele parameter dan niet in de romp van de procedure zichtbaar zijn; men spreekt van een **naam conflict**. In zo'n geval worden de formele en locale namen als het ware zo aangepast dat het conflict verdwijnt, zodat de actuele parameter toch als initialisatie in de procedure kan optreden.

Als voorbeeld, de oproep `raadsel (t)` in de context

```
    INT VAR t :: 0;

  PROC raadsel (INT VAR t):
    t INCR 1;
  ENDPROC raadsel;
```

wordt uitgevoerd als

```
    INT VAR t' :: t;
      t INCR 1;
    t:= t'
```

waarbij `t'` staat voor een geschikt gekozen unieke naam.

10.4 Classificatie van procedures

Procedures kunnen we op verschillende wijzen classificeren. Een eerste onderscheid is te maken naar gelang de procedure al of niet een resultaat heeft, d.w.z. of zijn uitvoering al of niet een waarde aflevert.

Wanneer de uitvoering van een procedure (al of niet via een formele parameter) de waarde van een globale variabele verandert dan zegt men dat die procedure een *effect* heeft. Overigens hoeft het effect niet merkbaar te zijn binnen het programma dat de procedure gebruikt, maar kan het bijvoorbeeld ook bestaan uit het afdrukken van een tekst op het beeldscherm van een terminal (een soort **externe variabele**). Strikt genomen heeft een procedure altijd een effect, omdat de aanroep van een procedure tenminste tot gevolg heeft, dat enige computertijd verbruikt wordt. We zullen hier echter afzien van het aspect van de tijd.

De observatie dat procedures al of niet een effect kunnen hebben en al of niet een waarde kunnen opleveren leidt tot een indeling van de procedures in vier soorten, in afhankelijkheid van hun gedrag ten opzichte van hun omgeving. Een procedure kan:

- een effect hebben en een waarde opleveren (**functie met zij-effect**)

- een effect hebben en geen waarde opleveren (**actie**)
- geen effect hebben en een waarde opleveren (**zuivere functie**)
- geen effect hebben en geen waarde opleveren (**dummy**)

De eerste soort is een mengvorm van actie en functie. Deze procedures zijn het lastigst in het gebruik, omdat hun tweeslachtig gedrag een bron kan zijn van onoverzichtelijkheid en fouten tijdens het programmeren. De vierde soort is de onschuldigste en heeft als belangrijkste parameterloze representant:

```
PROC doe niets:
ENDPROC doe niets
```

Een nuttig gebruik van procedures van deze laatste soort kan gemaakt worden bij het testen van programma's. Door voor een actie die nog niet gerealiseerd is een "dummy" procedure te nemen met een juiste kop en een lege romp kunnen andere delen van een programma vroegtijdig getest worden.

10.4.1 Zuivere functies

Een voorbeeld van een functie is:

```
REAL PROC max (REAL a, b):
  IF a > b THEN a ELSE b FI
ENDPROC max;
```

die het maximum van de parameters `a` en `b` oplevert. Andere bekende voorbeelden zijn de standaard goniometrische functies zoals `sin` en `cos`. Met behulp van `max` kunnen we definiëren

```
REAL PROC max (REAL a, b, c, d):
  max (max (a, b), max (c, d))
ENDPROC max;
```

die het maximum van 4 reals berekent.

Een zuivere functie werkt als een afbeelding van zijn parameters op een resultaat, die alleen van de waarden van die parameters afhankelijk is.

10.4.2 Acties

Een actie heeft een effect op zijn omgeving en levert geen resultaat. Bekende voorbeelden van acties zijn de procedures `get` en `put` om te lezen respectievelijk te schrijven. Vele voorbeelden zijn al gegeven, zoals `verhoog` en `verlaag`. Nog een voorbeeld (een nuttige procedure):

```
PROC verwissel (INT VAR x, y):
  INT CONST hulp :: x;
  x := y;
  y := hulp
ENDPROC verwissel;
```

Strikt genomen zijn functies niet noodzakelijk en kunnen we met acties precies hetzelfde bereiken (zoals in "statement-talen" als COBOL). Zo zouden we in plaats van de standaardfunctie `sin` een actie `bereken sinus` kunnen definiëren:

```
PROC bereken sinus (REAL x, REAL VAR
sinus):
  sinus := ingewikkelde uitdrukking
ENDPROC bereken sinus;
```

Om met de procedure `bereken sinus` bijvoorbeeld de waarde van de formule `1 - sin(x) ** 2` te bepalen, zouden we dan moeten schrijven:

```
REAL VAR s, resultaat;
bereken sinus (x, s);
resultaat := 1 - s ** 2
```

De notatie als functie is hier heel wat genietbaarder.

Men kan ook helemaal op de functie-toer gaan (zoals in zuivere "expressie-talen" als LISP), maar in het algemeen is het prettig de keuze te hebben tussen beide uitdrukkingsvormen en wijst zich hun gebruik vanzelf. Wanneer voorop staat dat de procedure een afbeelding van de (eventuele) parameters naar een waarde berekent, zal de functievorm gekozen worden. Wanneer geen resultaat of juist meer dan een waarde moet worden opgeleverd, zal de procedure als een actie worden gedeclareerd.

Een voorbeeld van het laatste is:

```
PROC deel met rest (INT deeltal, deler,
INT VAR quot, rest):
   quot := deeltal DIV deler;
   rest := deeltal - quot * deler
ENDPROC deel met rest;
```

10.4.3 Functies met zij-effect

Heeft een functie tevens een effect dan spreekt men van een **zij effect**. Deze naam geeft al aan dat de purist voor zij-effecten zijn neus ophaalt. Een functie hoort een waarde te leveren, niet tevens de globale omgeving te veranderen!

Dat neemt niet weg dat voorbeelden van zinvol gebruik van zij-effecten heel wel te vinden zijn. Zij hebben gemeenschappelijk dat de globale omgeving als geheugen van de functie dient en impliciet diens gedrag beïnvloedt. Voorbeelden:

```
INT VAR n :: 0;

INT PROC volgende klant:
  n INCR 1; n
ENDPROC volgende klant;
```

Ga zelf na welke waarde de procedure `volgende klant` bij de eerste aanroep, tweede aanroep, etc. oplevert.

```
BOOL VAR a;
BOOL PROC flipflop:
  a := NOT a; a
ENDPROC flipflop;

PROC set:
  a := TRUE
ENDPROC set;

PROC reset:
  a := FALSE
ENDPROC reset;
```

Voor electronici is dit een drietal procedures met een vertrouwd gedrag. Let op de wijze, waarop `flipflop` zijn waarde levert.
Nog zo'n functie (weer zonder parameters) is:

```
INT PROC next nat:
  INT VAR numb;
  REP get(numb)
  UNTIL numb >= 0
  ENDREP;
  numb
ENDPROC next nat;
```

die als waarde het eerstvolgende niet negatieve getal van de invoer oplevert en een zij-effect heeft op de invoer.

Een veilig gebruik van procedures met globale variabelen als geheugen vergt een faciliteit voor de **afscherming** van die variabelen ten opzichte van de omgeving waarin die procedures worden aangeroepen. In ELAN is hiervoor het **pakket mechanisme** aanwezig, dat wij in het tweede deel zullen behandelen.

Voor de programmering van grote programmasystemen, waar vele mensen aan werken, is een dergelijk expliciet afschermings-mechanisme onontbeerlijk.

10.4.4 Genericiteit

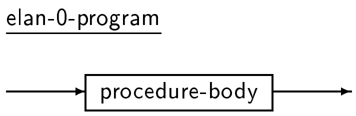
ELAN kent in tegenstelling tot oudere talen **generische procedures**: procedures met eenzelfde naam mogen naast elkaar voorkomen mits zij verschillen in aantal of type van de formele parameters. Van een procedureoproep dient het aantal, de volgorde en het type van de actuele parameters overeen te stemmen met het aantal, de volgorde en het type van de formele parameters van één van de procedures, gedeclareerd met die naam. Die procedure wordt dan door deze procedure-oproep geïdentificeerd.

In het tweede deel zullen we ingaan op het gebruik van **genericiteit** als abstractiemechanisme.

10.5 Structuur van programma's

Op dit punt gekomen kunnen we de structuur van ELAN-programma's bespreken.

Wij geven nog niet de volledige syntax (zie daarvoor Appendix A) maar beschrijven twee subsets van ELAN die we respectievelijk ELAN-0 en ELAN-1 zullen dopen.



De programmeermiddelen die we in de eerste 8 hoofdstukken hebben ingevoerd blijken precies die te zijn die in de romp van een procedure gebruikt kunnen worden.

Een conclusie is dat we ieder van de tot nog toe gegeven complete ELAN-0-programma's kunnen opnemen tussen de regels

```
PROC programma:
```

en

```
ENDPROC programma;
```

en daarmee bombarderen tot een procedure, een actie zonder parameters.

Door het toelaten van closed-declarations voor procedures (en operatoren en typen) raken we in de tweede, ruimere subset ELAN-1, d.w.z. ELAN zonder pakketten.

Duidelijk te zien is dat een ELAN-1-programma bestaat uit een **Bottom-Up gedeelte**, waarin enige abstracte algoritmen en typen worden gedefinieerd, gevolgd door een **Top-Down gedeelte** waarin ze worden gebruikt.

In de rest van dit boek hebben we genoeg aan de subset ELAN-1.

10.6 Nogmaals: procedures en verfijningen

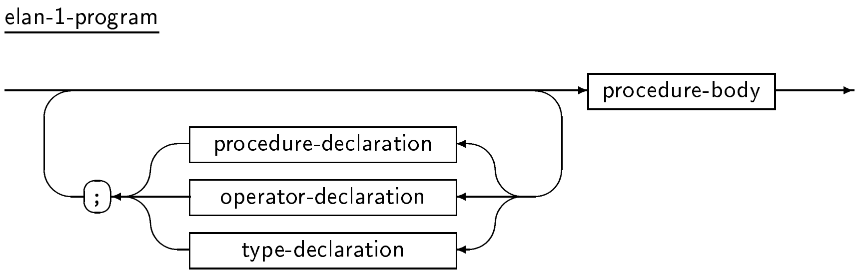
Verfijningen en procedures hebben veel gemeenschappelijks. Waarom dan toch twee mechanismen onderscheiden?

De verschillen tussen procedures en verfijningen zijn subtiel maar hebben vrij grote consequenties.

- Een procedure heeft als omgeving (dus als geldigheidsgebied van zijn declaratie) het programma. Datzelfde geldt voor verfijningen van het programma maar niet voor verfijningen van procedures. Die laatste hebben als omgeving de procedure.
- Communicatie tussen verfijningen gaat uitsluitend impliciet, via effecten op gemeenschappelijk globale objecten. Communicatie tussen procedures gaat bij voorkeur expliciet, via parameters.
- Een object gedeclareerd binnen een procedure is locaal in die procedure; een object gedeclareerd in een verfijning is in het hele programma zichtbaar. Daarmee kan dus ook een declaratie verfijnd worden! Vele voorbeelden tonen aan dat met name een geïnitialiseerde declaratie (die zowel een variabele bekend maakt als leidt tot een welgedefinieerde toestand van het programma) uitstekend geverbaliseerd kan worden.
- Een procedure krijgt zijn betekenis pas door zijn declaratie; daarvóór mag hij niet worden opgeroepen. Een verfijning is zonder uitvoering van een declaratie overal en steeds in zijn geldigheidsgebied oproepbaar.

In andere programmeertalen komen verfijningen niet als aparte constructies voor. Daar kan men (parameterloze) procedures als verfijningen gebruiken. Daarbij ontstaat echter een aantal problemen:

- Sommige programmeertalen (zoals standard BASIC) hebben zodanige beperkingen op de naamsgeving van procedures dat deze niet meer als abstractiemiddel gebruikt kunnen worden.



- Het feit dat procedures pas na uitvoering van hun declaratie mogen worden opgeroepen betekent dat de declaraties tekstueel vooraf moeten gaan aan hun eerste toepassing. Bij het Top-Down programmeerproces gaat het juist omgekeerd: op een bepaald moment ontstaat de behoefte aan een bepaalde abstracte algoritme die pas zo lang mogelijk daarna gerealiseerd wordt. De ontstaans-volgorde en de tekstuele volgorde kunnen dus bij gebruik van procedures niet dezelfde zijn.
- Het schrijven van een procedure-declaratie met een gedetailleerde kop voegt zoveel "overhead" toe aan het ontwerpproces dat de programmeur er van afgehouden wordt vele korte abstracte algoritmen te schrijven. Impliciete communicatie geeft veel minder overhead en past goed in de Top-Down denkwijze.
- De in vele programmeertalen (zoals PASCAL) bijkomende verplichting alle variabelen vóór het algoritmische gedeelte van een procedure of programma te declareren staat een natuurlijke Top-Down stijl sterk in de weg. ELAN is daarentegen zeer liberaal wat de plaatsing van declaraties betreft.
- De blokstructuur die door de mogelijkheid van nesting van procedure-declaraties ontstaat stelt beginners voor grote moeilijkheden. De zeer eenvoudige geldigheidsregels van ELAN zijn veel eenvoudiger te volgen.

Het is ongetwijfeld waar dat procedures ook als verfijningen gebruikt kunnen worden, maar de genoemde problemen rechtvaardigen ten volle de invoering van verfijningen als apart syntactisch mechanisme. Ook vanuit het oogpunt van systematiek van de taalconcepten is het voordelig de hulpmiddelen voor Top-Down programmering (verfijningen, controlstructuren) en Bottom-Up programmering (pakketten, procedures, operatoren, typen) te kunnen onderscheiden.

Verfijningen zijn vooral een hulpmiddel om systematisch programmeren te leren. Beheerst men eenmaal het vak dan zal men ook zonder het gebruik van dit hulpmiddel systematisch te werk gaan.

Verfijningen zijn voor de programmeur wat de navel is voor de Zen-boeddhist.

10.7 Opgaven

1. (invoer/uitvoer van Booleans) Schrijf procedures `put` en `get` voor de invoer en uitvoer van Booleans. Ondervind daarbij hoe moeilijk het is een conventie voor de representatie van waarheidswaarden te vinden die voor iedereen acceptabel is.
2. (romeinse getallen) Schrijf een procedure

```
INT PROC waarde van (TEXT romeins getal)
```

   die een romeins getal omzet in een integer. Gebruik daarbij een hulpprocedure voor de waarden van romeinse cijfers.
3. (naar Romeinse getallen) Schrijf een procedure

```
TEXT PROC romeins van (INT getal)
```

   die een natuurlijk getal in het romeins weergeeft. Test hem met de vorige.
4. (standaardbibliotheek uit hoofdstuk 6) Realiseer de procedures `text`, `subtext`, `pos` en `LENGTH` met behulp van de operaties `+`, `=`, `HEAD` en `TAIL`.

Chapter 11

Een taalkundige toepassing

In dit hoofdstuk behandelen we enige begrippen rond talen en grammatica's en beschouwen we een taalkundige toepassing van de informatica: het genereren en analyseren van zinnen aan de hand van een grammatica. In het voorbijgaan zullen we nog een onderwerp van algemeen belang aan de orde stellen, nl. de toevalsgetallen.

De meeste toepassingen van de informatica in de menswetenschappen, en met name die in de taalkunde, kunnen als volgt worden ingedeeld:

- Opslag van teksten. Invoer, opslag en wijziging van teksten ("word-processing"), preparatie voor druklegging, automatisch zetten. Het zal u bijvoorbeeld niet ontgaan zijn dat de tekst van dit boek met behulp van de computer (en heel veel mensenwerk) drukrijp is gemaakt. Door het steeds perfecter en professioneler worden van de uitvoerapparatuur van de computer wordt de computer steeds meer tot een super-schrijfmachine.

- Rekenen aan teksten. De computer met zijn grote geduld en betrouwbaarheid is een bij uitstek geschikt apparaat om tellingen aan teksten te verrichten (taalstatistiek), gegevens op te zoeken en gegevens te sorteren. De computer vervangt hier steeds meer de schoenendoos met fiches waarin de ouderwetse taalwetenschapper zijn aantekeningen bewaarde.

- Toepassingen van syntactische methoden op teksten, zoals: spelling checkers, grammatica-checkers en stijl-checkers; methoden voor het zoeken in grote tekstbestanden naar teksten die op een bepaald onderwerp betrekking hebben ("Information filtering"; "Information retrieval"); tekst-samenvatters ("abstracting") en automatische vertalers; maar ook allerlei onderzoek naar de eigenschappen van talen en van het gebruik dat daarvan gemaakt wordt. De taalkundige en de informaticus hebben een gemeenschappelijk interesse in de beschrijving van talen en de automatische verwerking van zinnen.

Op deze laatste soort toepassingen willen we wat dieper ingaan.

11.1 Over de beschrijving van talen

We vatten een **taal** op als een verzameling van zinnen. Zo bedoelen we met "de nederlandse taal"(nogal tautologisch) de verzameling van die zinnen die tot de nederlandse taal behoren. Onder de taal Elan verstaan we de verzameling van alle Elan programma's.

Hoe kun je vaststellen dat een zin tot een bepaalde taal behoort? Je zou natuurlijk bij je eigen taalkennis te raden kunnen gaan. Of je zou de zin ter beoordeling aan een of meer "native speakers" van de taal kunnen voorleggen.

Een natuurlijke taal is een levend organisme, onderdeel van een heel sociaal, cultureel en economisch systeem. De grenzen van de taal zijn vaag en tijdsgebonden. Uitdrukkingen, zinsconstructies en woorden kunnen gaandeweg gemeengoed worden, andere raken in onbruik. Er bestaan dialecten en taalvarianten. De spelling en de uitspraak wijzigen zich in de loop der tijd.

Bij een kunsttaal als onze programmeertaal, door mensen bedacht en beschreven, liggen de grenzen veel scherper vast. Maar daar hebben wij het probleem van de overdracht van de taal aan anderen: we worden niet van jongs af aan erin ondergedompeld, en de taal kent geen echte "sprekers".

Daarom zou je willen beschikken over een betrouwbare taalbeschrijving. Maar hoe beschrijf je een taal?

Louter een opsomming van alle zinnen te geven is onmogelijk, omdat het er veel te veel zijn. Er is zelfs geen (harde) grens aan het aantal zinnen: ik kan iedere Nederlandse zin twee woorden langer maken door er omheen te zetten:

`Hij zei:` " ... ".

Ook is zo'n opsomming niet voldoende, omdat er grotere samenhangen bestaan dan zinnen, en omdat de zinnen pas zin hebben in relatie tot een bepaalde context. Een taalbeschrijving, zij het van een natuurlijke of van een kunsttaal, onderscheidt drie gezichtspunten: syntax, semantiek en pragmatiek.

De **syntax** legt vast welke rijen van symbolen zinnen van de taal vormen en welke structuur ze bezitten. De **semantiek** legt bij syntactisch juiste zinnen vast wat ze betekenen. De **pragmatiek** houdt zich bezig met de verhouding tussen de taal en de mens die hem gebruikt, kortom: de eigenlijke kern van de zaak.

Met semantische en pragmatische aspecten zullen we ons hier niet bezig houden. We behandelen een syntactisch beschrijvingsmiddel uit de mathematische linguïstiek, de **contextvrije grammatica** (CF-grammatica, **CFG**). Dit mechanisme is sterk verwant aan de syntax-diagrammen die we eerder in dit leerboek hanteerden, maar leidt tot een wat concieser notatie met een hogere graad aan abstractie.

We noteren deze grammatica's ietwat anders dan de taalkundigen, in de notatie genoemd naar A. Van Wijngaarden [ALG68], de vader van de Nederlandse informatica. Er bestaan nog vele andere wijzen om een contextvrije grammatica te noteren, maar de verschillende notatiesystemen zijn eenvoudig tot elkaar herleidbaar.

11.2 Een syntactische notatie

Een contextvrije grammatica is een formele notatie die sterk verwant is aan het begrip "redekundige ontleding" voor natuurlijke talen. Taalkundige begrippen worden gedefinieerd in termen van andere begrippen en uiteindelijk in termen van bepaalde woorden of klassen van woorden uit het vocabulaire van de taal.

Als voorbeeld een fragment uit een nederlandse grammatica.

- een zin bestaat uit een onderwerp, gevolgd door een gezegde, gevolgd door een lijdend voorwerp.
- een onderwerp is hetzij een persoonlijk voornaamwoord, hetzij een lidwoord gevolgd door een zelfstandig naamwoord.
- een lidwoord is een van de symbolen "de", "het" of "een".

In de bedoelde notatie zou je dit stuk grammatica als volgt kunnen uitdrukken:

```
zin:
  onderwerp, gezegde, lijdend voorwerp.

onderwerp:
  persoonlijk voornaamwoord;
  lidwoord, zelfstandig naamwoord.

lidwoord:
  de-symbol;
  het-symbol;
  een-symbol.
```

De leestekens in deze grammatica kan men als volgt uitspreken:

:	als "is een"
,	als "gevolgd door een"
;	als "of een"
.	als een korte pauze.

Het is geen toeval dat deze notatie met de Elan-notatie voor verfijningen zeer sterk overeenkomt. Verfijningen hebben een zelfde beschrijvend karakter als een grammatica en we zouden de verfijningen wel de "speciale grammatica" van een programma kunnen noemen.

11.2.1 Contextvrije grammatica

In een contextvrije grammatica onderscheiden we tussen *begrippen* en *symbolen*. Ieder begrip en ieder symbool heeft een naam, geschreven als een *woord* bestaande uit kleine letters en eventueel cijfers en eventueel, ter verhoging van de leesbaarheid, verbindingsstreepjes ("-") of spaties.

Woorden zoals de-symbol, het-symbol etc. zijn bij (namen voor) de symbolen van de taal, de zgn. *terminale* symbolen of **terminals**. Voor de terminals is geen produktieregel gegeven maar een of meer *representaties*. Neem als voorbeeld het het-symbol: het kan worden uitgesproken, met de hand geschreven of door een printer afgedrukt, terwijl toch steeds hetzelfde symbool bedoeld is, net zoals 2 altijd zijn tweeheid behoudt. Om ze gemakkelijk te kunnen herkennen zullen we de namen van terminale symbolen veelal laten eindigen op -symbol.

Woorden als zin, onderwerp en lidwoord, dus de namen van begrippen, worden *niet-terminale* symbolen of **nonterminals** genoemd. De grammatica bevat voor ieder nonterminal een (productie)**regel** die precies definieert voor welke tekenrijen het bedoelde begrip kan staan.

Een regel bestaat uit een *linkerkant*, een dubbele punt, een *rechterkant* en een punt. De linkerkant bestaat uit de naam van het te definiëren nonterminal. De rechterkant bestaat uit een of meer *alternatieven*, gescheiden door puntkomma's. Een **alternatief** uit de definitie van een nonterminal noemt men ook een **directe productie** van dit nonterminal.

Een alternatief bestaat uit een *lijst* van woorden, d.w.z. uit nul of meer terminals of nonterminals, die door komma's van elkaar gescheiden worden. Een alternatief kan dus ook leeg zijn.

Uitgaande van het begrip **directe productie** definiëren we nu de begrippen productie, terminale productie en zin.

Een **productie** van een nonterminal x is hetzij een directe productie van x, hetzij een lijst van woorden, verkregen door in een productie van x een daarin optredende nonterminal y te vervangen door een directe productie van y.

Enkele producties van zin:

zin
⇒ onderwerp, gezegde, lijdend voorwerp
⇒ persoonlijk voornaamwoord, gezegde, lijdend voorwerp
⇒ lidwoord, zelfstandig naamwoord, gezegde, lijdend voorwerp
⇒ "de", zelfstandig naamwoord, gezegde, lijdend voorwerp
⇒ ...

Onder een **terminale productie** van een nonterminal verstaan we een productie daarvan die geen nonterminals meer bevat en dus uitsluitend bestaat uit een lijst van terminals. De representatie van een terminale productie bestaat uit de opeenvolging van representaties van zijn symbolen.

Een **zin** is een representatie van van een terminale productie van een speciale start-nonterminal, bijv. zin.

Een contextvrije syntax van Elan is te vinden achterin dit leerboek (Appendix A). Aan de hand daarvan is het mogelijk vast te stellen of een bepaalde symboolrij tot de constructies van Elan behoort. Zo kun je antwoord krijgen op vragen als:

- is `mar1lyn` een identifier?
- kan een verfijning eindigen met een `LEAVE`-opdracht (de terminator)?
- kan een verfijning eindigen met een object-declaration?

zonder voorbeelden uit te proberen op de computer.

11.3 Wat doe je met een grammatica?

Stel dat we nu een grammatica van een groot deel van het Nederlands hebben. Hoe kunnen we dan vaststellen of deze grammatica klopt?

Een manier zou zijn, aan de hand van de grammatica een grote verzameling zinnen te **analyseren** en te kijken of een voldoende groot deel van deze zinnen volgens die grammatica geanalyseerd kan worden. De taalkundige richting die zo te werk gaat heet de *corpuslinguïstiek*, naar de (grote) verzamelingen van teksten *(corpora)* waarmee gewerkt wordt.

Een andere techniek bestaat uit het **genereren** van zinnen: het voortbrengen van voorbeeldzinnen die door mensen op grammaticaliteit bekeken kunnen worden. Een grammatica die geschikt is voor het genereren van zinnen noemt men een **generatieve grammatica**. Nu heeft het weinig zin eenvoudigweg in enigerlei volgorde alle zinnen van de taal te genereren. Dat zou bijzonder vervelend en weinig leerzaam zijn. Interessanter is het, "willekeurige" voorbeelden van nederlandse zinnen te produceren.

Wij zullen aangeven hoe je uitgaand van een generatieve grammatica een **generator** kunt programmeren. Stel, een van de grammaticaregels luidt:

```
stellende zin:
  onderwerp, gezegde, lijdend voorwerp.
```

Om een voorbeeld van een stellende zin te genereren moet ik een voorbeeld genereren van een onderwerp, en dan een (voorbeeld van een) gezegde, en dan een lijdend voorwerp. Dit ruikt uiteraard naar het oproepen van algoritmen. Voor iedere nonterminal introduceren we een procedure die daar voorbeelden van kan genereren, te beginnen met:

```
PROC stellende zin:
  onderwerp;
  gezegde;
  lijdend voorwerp
ENDPROC;
```

Het zal u zeker niet ontgaan hoezeer de voorgaande notatie in de programmeertaal lijkt op de syntaxregel. Alleen hebben (gevaarlijk genoeg) de leestekens ; en , in de twee systemen een verschillende betekenis.

We kunnen verder gaan volgens de syntaxregel

```
onderwerp:
  persoonlijk voornaamwoord;
  zelfstandig naamwoord groep.
```

Nu moeten we hetzij een voorbeeld van een persoonlijk voornaamwoord, hetzij een zelfstandige naamwoordsgroep genereren, bijvoorbeeld met gelijke waarschijnlijkheid:

```
PROC onderwerp:
  IF fifty fifty
  THEN persoonlijk voornaamwoord
  ELSE zelfstandig naamwoord groep
  FI
ENDPROC;
```

Hier moet `fifty fifty` een Boole'se uitdrukking zijn die met 50% waarschijnlijkheid `TRUE` oplevert en met 50% waarschijnlijkheid `FALSE`. Hoe bereiken we dit?

11.4 Zijsprong: over het toeval

Een dobbelsteen met 6 zijden is een hulpmiddel bij het verkrijgen van toevallige getallen. Van een dobbelsteen moeten we vereisen:

- het is, ook bij kennis van de voorgeschiedenis, *onvoorspelbaar* wat de volgende uitkomst zal zijn
- de zes mogelijke uitkomsten zijn *homogeen verdeeld*, d.w.z. hebben alle een gelijke waarschijnlijkheid.

Ieder model dat deze twee eigenschappen bezit kan de plaats innemen van een dobbelsteen. Wij kunnen bijvoorbeeld proberen zulke toevalsgetallen ("random numbers") af te leiden uit een natuurlijk proces, zoals een ruisgenerator, of uit de "toevallige" inhoud van een deel van het geheugen van de computer, maar het eerste verlangt extra apparatuur en het tweede werkt bijzonder weinig bevredigend.

We kunnen ook proberen algoritmisch een reeks te bepalen van reële toevalsgetallen in het bereik 0.0 tot 1.0 met

R_0 = startwaarde
R_1 = verkregen uit R_0
R_2 = verkregen uit R_1
...

zodanig dat de reeks R_i onvoorspelbaar en homogeen verdeeld is. We rekenen daarbij liever met een reeks niet-negatieve integers I_i:

$$R_i = I_i / max\ int$$

Hierbij moet, om R_i in het bereik 0.0 tot 1.0 te houden, gelden: $I_i > 0$, en vanzelfsprekend $I_i \leq max\ int$.

Een schema met verbluffend goede eigenschappen blijkt

$$I_{i+1} = [a * I_i + b] \bmod c$$

met a, b, c geschikt gekozen integers. Om de berekening te vereenvoudigen kiezen we $c = max\ int$, dus op de meeste machines een tweemacht min 1. Merk op dat de reeks in ieder geval periodiek is: er zijn maar eindig veel integers, dus na een zeker aantal malen moet de reeks een element herhalen. Als we een bepaald getal I_i ten tweede male bereiken volgt daarop hetzelfde getal I_{i+1}. We moeten proberen de periode zo lang mogelijk te houden. Het maximum is uiteraard `max int`. Ook moet de reeks onvoorspelbaar zijn. Daartoe kieze men voor a en b priemgetallen, zodanig dat a, b en c geen delers gemeenschappelijk hebben.

De reeks blijkt bij een zorgvuldige keuze van a en b homogeen verdeeld, en ook sterkere statistische tests wijzen niet op storende regelmatigheden. Bovendien is de reeks reproduceerbaar, in tegenstelling tot het echte toeval. Dit stelt ons in staat een "toevalsproces" te herhalen. Zo'n reproduceerbare toevalsreeks heet **pseudo-random**.

11.4.1 Toevalsgetallen in Elan

Wij hoeven niet zelf te zoeken naar geschikte waarden voor a, b en c op onze computer. Ergens in de standaardpakketten van Elan is een integer variabele `last random` gedeclareerd en verder een algoritme met een reëel resultaat, in de trant van

```
REAL PROC random:
  last random := a * last random + b;
  abs (last random / max int)
ENDPROC random;
```

Als toepassing hiervan heeft de uitdrukking

```
random < p
```

een kans p om `TRUE` op te leveren.

De variabele `last random` kan worden geïnitialiseerd met behulp van een oproep `initialise random (expr)` waarbij `expr` een reële uitdrukking moet zijn, waarvan de waarde tussen 0.0 en 1.0 inligt.

We kunnen een dobbelsteen simuleren als

```
maak een dobbelsteen:
  INT VAR dobbelsteen;
  dobbelsteen := trunc(random * 6.0) + 1.
```

Dit levert een fraaie dobbelsteen: hij onvoorspelbaar, gelijk verdeeld en bovendien ook reproduceerbaar. Is de reproduceerbaarheid een bezwaar? Gebruik dan een fysisch proces als bijvoorbeeld de klok om de voorspelbaarheid te verstoren, dan is het resultaat echt **random**.

Teneinde het gemakkelijker te maken om integer toevalsgetallen te produceren kan men `random` ook met twee integer parameters oproepen. Hij levert dan een integer die in ligt tussen deze beide grenswaarden. Zo geeft `random (1,6)` het equivalent van een dobbelsteen.

De algoritme `fifty fifty` kan nu luiden:

```
fifty fifty:
  random < 0.5.
```

11.5 Een engelse grammatica

Wij geven nu een complete contextvrije grammatica voor een heel klein deel van het engels. We willen deze grammatica gebruiken voor het genereren van voorbeeldzinnen, daarom zullen we alle keuzes tussen alternatieven voorzien van een waarschijnlijkheid van elk alternatief. Zo'n met waarschijnlijkheden versierde grammatica heet een **probabilistische grammatica**.

Aan de hand daarvan willen we willekeurige zinnen laten genereren. Alleen de syntax van de geproduceerde zinnen hebben we in de hand, de semantiek ontglipt ons. Om nu toch niet al te onzinnige zinnen te produceren zullen wij ons in het vocabulaire beperken tot woorden uit een bepaald semantisch gebied. Als inspiratie diende daarbij het amerikaanse vers "Mary had a little lamb".

```
sentence:
  subject, predicate.
```

Hier is nog geen keuze te maken.

```
subject:
  [.7] substantive group;
       pers pronoun 1.
```

De notatie met vierkante haken dient ertoe om aan te geven dat hier de waarschijnlijkheid van het eerste alternatief gelijk is aan 0.7, en die van het tweede alternatief $1.0 - 0.7 = 0.3$. Wij programmeren deze regel als volgt:

```
PROC subject:
  IF random <= 0.70
  THEN substantive group
  ELSE pers pronoun 1
  FI
ENDPROC subject;
```

De grammatica gaat verder:

```
substantive group:
  article, noun phrase.

article:
  [.45] "a";
        "the".
```

Voor een terminaal symbool schrijven we een representatie daarvan naar het beeldscherm, gevolgd door een spatie.

```
PROC article:
  IF random <= 0.45
  THEN put ("a ")
  ELSE put ("the ")
  FI
ENDPROC article;
```

```
noun phrase:
  [.25] adjective, noun phrase;
        nounpart.
```

Merk op dat het begrip noun phrase in termen van zichzelf gedefinieerd is, een **recursieve definitie**. De bedoeling van deze regel is aan te geven dat er geen à priori bovengrens bestaat op het aantal adjectieven voorafgaand aan een nounpart. De kans op heel veel adjectieven is echter heel klein. We realiseren dit effect door een onbegrensde herhaling met telkens 25% kans op voortzetting.

```
PROC noun phrase:
  WHILE random <= 0.25
  REP adjective
  ENDREP;
  nounpart
ENDPROC noun phrase;
```

We hebben de keus tussen een aantal adjectieven

```
adjective:
  [.30] "little";
  [.60] "meek";
  [.90] "big";
        "bad".
```

Merk op dat we de kansen cumulatief aangeven, met het laatste alternatief als "restpost". De kans op de vier alternatieven van adjective is dus resp. 0.3, 0.3, 0.3 en 0.1.

We hebben hier de adjectiva expliciet in de grammatica opgenomen. Voor een natuurlijke taal is hun aantal zo groot dat het meer voor de hand ligt ze in een aparte woordenlijst (**lexicon**) op te nemen. In feite gaat het om een open categorie van symbolen, die nooit afsluitend beschreven kan worden.

Om een voorbeeld van een adjective te genereren trekken we een random getal en kiezen in afhankelijkheid van de waarde daarvan een alternatief, oftewel in Elan:

```
PROC adjective:
  REAL CONST r :: random;
  IF r <= 0.30
  THEN put ("little ")
  ELIF r <= 0.60
  THEN put ("meek ")
  ELIF r <= 0.90
  THEN put ("big ")
  ELSE put ("bad ")
  FI
ENDPROC adjective;
```

De rest van de grammatica spreekt hopelijk voor zich.

```
nounpart:
  [.25] noun, rel clause;
        noun.
noun:
  [.20] "boy";
  [.40] "girl";
  [.60] "lamb";
  [.80] "bear";
        "tree".
```

Ook deze open categorie kan beter via een lexicon worden beschreven. Vanwege de eenvoud beperken we ons hier tot 5 substantiva.

Aangezien de realisering van de volgende regels voor de hand ligt geven we verder geen programmeervoorbeelden maar beperken ons tot de syntax.

```
rel clause:
  "that", predicate.
predicate:
  adverbial option, verb, object.
adverbial option:
  [.20] adverbial; .
adverbial:
  [.33] "always";
  [.67] "often";
        "never".
verb:
  [.25] "sees";
  [.50] "likes";
  [.75] "eats";
        "dreams about".
object:
  [.80] substantive group;
        pers pronoun 4.
```

De persoonlijke voornaamwoorden (waartoe we hier ook de persoonsnamen rekenen) komen voor in de 1^e en de 4^e naamval.

```
pers pronoun 1:
  [.25] "he";
  [.50] "she";
  [.75] "Mary";
        "Jim".
pers pronoun 4:
  [.25] "him";
  [.50] "her";
  [.75] "Mary";
        "Jim".
```

11.5.1 Resultaten

We laten het aan de lezer over, aan de hand van deze regels procedures te schrijven die tesamen een programma vormen dat willekeurige zinnen uitbraakt van een hele kleine deeltaal van het engels.

Een aantal zinnen zoals ze uit dit programma rollen luidt

> the bear that sees the bear likes a bad bear
> the bear likes him
> the boy dreams about the bear
> a girl never sees the girl
> Jim sees Mary
> a boy that sees the big tree always likes
> a boy that always eats a tree
> Mary eats a little bear

Duidelijk is dat een zin betekenisvoller lijkt naarmate hij korter is. Maar ja, iedere betekenis berust dan ook op toeval. Hoe langer de zin, hoe meer hij de kans krijgt zichzelf tegen te spreken.

11.6 Voorbeeld: syntaxanalyse

Met het begrip **syntax analyse** worden twee nauw samenhangende activiteiten bedoeld: het herkennen en het ontleden aan de hand van een grammatica.

Een **herkenner** ("recognizer") voor een CF grammatica G is een algoritme die voor een gegeven reeks symbolen s vaststelt of s een van de zinnen van de taal van G is. Het resultaat is een waarheidswaarde

$$recognize(G,s) = \begin{cases} s \in L(G) \rightarrow true \\ s \notin L(G) \rightarrow false \end{cases}$$

Een **ontleder** ("parser") is een herkenner die, gegeven een zin $s \in L(G)$, de structuur van s bepaalt (in de vorm van een ontleedboom).

Onder een **ontleedboom** verstaan we een boom, waarvan de knopen zijn gemarkeerd met nonterminals en de bladen met terminals, zoals in Fig. 11.1.

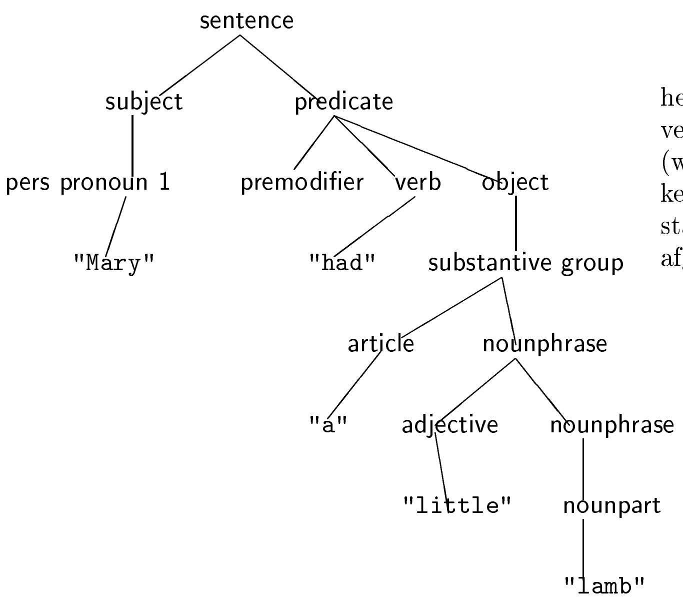


Figure 11.1: Een ontleedboom

Deze twee activiteiten kunnen gemakkelijk worden gecombineerd: een ontleder moet per slot van rekening ook niet-zinnen aan de kaak stellen en een herkenner kan gemakkelijk tot een ontleder worden uitgebreid. Het beste bewijs dat een reeks symbolen een zin vormt is trouwens een ontleedboom!

Wij zullen een aantal aspecten van syntaxanalyse nader bekijken, en een herkenner construeren voor het fragment van de engelse grammatica dat we zojuist hebben beschreven.

11.6.1 Recursive descent

We willen een herkenner construeren voor zinnen, dat wil zeggen, een Boole'se procedure `is sentence` die true levert dan en slechts dan als de invoer een terminale productie is van de nonterminal sentence uit bovenstaande grammatica. Een reeks terminale symbolen is een terminale productie van sentence als hij bestaat uit een terminale productie van subject gevolgd door een van predicate.

We gaan in principe net zo te werk als bij het construeren van een generator: we schrijven een Boole'se procedure als herkenner voor elk van de niet-terminale symbolen, in termen van herkenners voor de andere niet-terminale en terminale symbolen. Heel eenvoudig, maar er liggen serieuze voetangels en klemmen op onze weg.

We bewaren de reeks symbolen als een tekst in de variable `input` en gebruiken een index `inptr` om te onthouden hoe ver het herkenningsproces is voortgeschreden. Het herkennen van een terminaal symbool komt neer op het overslaan van de kop van de invoer mits die kop overeenkomt met het symbool dat we pogen te herkennen.

```
TEXT VAR input :: de reeks symbolen;
INT VAR inptr :: 1; {de positie van het 1e
symbool}
```

We introduceren een Boole'se procedure voor het herkennen van terminale symbolen. We volgen de conventie dat achter ieder symbool een spatie moet staan (waaraan het einde van het symbool kan worden herkend). Ook na het laatste symbool moet dus een spatie staan. Het te herkennen symbool mag verder ook voorafgaan worden door spaties.

```
BOOL PROC is (TEXT s):
  sla eventuele spaties over;
  IF volgende woord gelijk aan s
  THEN
    sla het over;
    true
  ELSE false
  FI.
  sla eventuele spaties over:
    WHILE (input SUB inptr) = " "
    REP inptr INCR 1
    ENDREP.
  volgende woord gelijk aan s:
    subtext (input + " ", inptr,
             inptr + LENGTH s ) = s + " ".
  sla het over:
    inptr INCR LENGTH s.
ENDPROC is;
```

Hoe moeten we een herkenner maken voor

sentence: subject, predicate.

uit de herkenners voor zijn delen? Een eerste gedachte

```
BOOL PROC is sentence:
  is subject AND is predicate
ENDPROC is sentence;
```

blijkt fout: de tweede herkenner wordt niet alleen opgeroepen ingeval de eerste slaagt, maar ook als ie faalt. Een dyadische operator zoals AND berekent nu eenmaal allebei zijn operanden. En wat erger is, ze worden collateraal berekend, zodat we zelfs niet zeker zijn van de volgorde waarin de beide herkenners worden opgeroepen! Wij moeten veel voorzichtiger zijn.

```
BOOL PROC is sentence:
  IF is subject
  THEN is predicate
  ELSE false
  FI
ENDPROC is sentence;
```

De volgende syntaxregel

subject:
  substantive group;
  pers pronoun 1.

heeft twee alternatieven, die we op de rij af proberen. Als een ervan slaagt hebben we een subject gevonden.

```
BOOL PROC is subject:
  IF is substantive group
  THEN true
  ELSE is pers pronoun 1
  FI
ENDPROC is subject;
```

substantive group:
  article, noun phrase.

De volgende regel leidt tot een keuze tussen terminale symbolen

article:
  "a";
  "the".

```
BOOL PROC is article:
  IF is ("a")
  THEN true
  ELSE is ("the")
  FI
ENDPROC is article;
```

De overige regels kunnen zonder verdere complicaties in herkenners worden omgezet, behalve die voor adverbial option. Omdat het om een optioneel deel gaat kan de corresponderende herkenner niet falen. De lege tekst staat er immers altijd. Daarom realiseren we hem liever als een actie dan als een Boole'se procedure die toch altijd TRUE levert.

predicate:
  adverbial option, verb, object.

adverbial option:
  modifier; .

Merk op dat hier het tweede alternatief leeg is.

```
BOOL PROC is predicate:
  adverbial option;
  IF is verb
  THEN is object
  ELSE false
  FI
ENDPROC is predicate;
```

```
PROC adverbial option:
  IF is modifier
  THEN
  ELSE
  FI
ENDPROC adverbial option;
```

Om aan te geven dat de herkenner niet kan falen beginnen we zijn naam ditmaal niet met `is...` zoals bij andere herkenners.

We laten het aan de lezer over, de overige syntaxregels in herkennende procedures om te zetten.

Tot slot voegen we een stuur-programma'tje toe. Het poogt een invoerregel te herkennen als een zin, en blijft de invoer net zolang ter correctie aanbieden totdat er inderdaad een zin is gevonden.

```
simple parser:
  start with empty input;
  REP ask further input
  UNTIL is sentence
  ENDREP;
  congratulations.

  start with empty input:
    TEXT VAR input :: "";
    INT VAR inptr :: 1;
    { index of next input character }
    put ("Input, please ...").

  ask further input:
    line;
    edit (input, inptr);
    inptr := 1.

  congratulations:
    line;
    put (inptr * "-");
    line;
    put ("Success!");
    line.
```

Als er een zin herkend is stopt het programma, waarbij het herkende deel van de zin wordt onderstreept. Dit is bedoeld als oplossing voor het probleem dat de herkenner, van links naar rechts werkend, tevreden is met de eerste de beste zin en er niet om maalt als er achter die zin nog iets staat. Uiteraard zouden we gemakkelijk genoeg de extra voorwaarde kunnen opleggen dat een zin de hele invoer moet overdekken, maar dit probleem geeft al een hint dat er problemen zijn met het samenstellen van herkenners.

11.6.2 Enige complicaties

Deze bepaalde herkenner zal correct werken voor de grammatica waaruit hij was afgeleid, maar dat betekent beslist niet dat we op deze wijze voor iedere contextvrije grammatica een herkenner kunnen construeren.

Om te beginnen kan een grammatica een regel bevatten waarvan twee alternatieven met eenzelfde (terminaal of niet-terminaal) symbool beginnen, zoals

nounpart:
  noun, rel clause;
  noun.

Dit leidt tot een behoorlijke verwarring bij de herkenner, die best eens een heel andere taal zou kunnen herkennen dan generatief bedoeld is (denk aan een noun gevolgd door een tweede noun in plaats van een relclause). De oplossing is deze regel te links-factoriseren door het gemeenschappelijke noun "buiten haakjes te halen":

nounpart: noun, rel clause option.

rel clause option: rel clause; .

Bovendien is er het probleem van de **links recursie**: Een nonterminal heet (direct) links-recursief als een van zijn alternatieven weer met een exemplaar van die nonterminal kan beginnen. Naast directe links-recursie is ook indirecte links-recursie mogelijk (via een aantal andere nonterminals).

De links-recursieve regel

noungroup:
  noun;
  noungroup, postmodifier.

drukt uit dat een noun door willekeurig veel postmodifiers kan worden gevolgd. Schrijven we dit zo om tot een herkenner dan onstaat een eindeloos proces: Als de invoer niet met noun begint dan zal de procedure is noungroup zichzelf oproepen, en zo voort tot aan het einde van de wereld of tot het geheugen van de computer vol zit.

Deze en andere problemen maken dat een gegeven CF grammatica slechts onder hele beperkende voorwaarden via recursive descent herkend kan worden. De boven gegeven grammatica houdt zich (uiteraard!) aan de bedoelde, zgn. LL(1)-voorwaarden. Er bestaat een uitgebreide literatuur over syntaxanalyse waar we de geïnteresseerde lezer naar verwijzen [ASU86, WGO84].

De recursive descent techniek is veel te beperkt voor serieuze taalkundige toepassingen, omdat hij bijvoorbeeld geen raad weet met **ambiguïteit**. Een zin is **ambigu** (meerduidig) als hij op meer dan één manier door de grammatica geproduceerd kan worden, zoals de beroemde zinnen

They are flying planes
Time flies like an arrow.

Er bestaan wel degelijk ontleedmethoden die tegen ambiguïteit bestand zijn - maar dit is niet de plaats om daarop in te gaan.

Het zal de lezer ook zijn opgevallen dat de gegeven grammatica zorgvuldig alle problemen vermijdt van overeenkomst in persoon en getal tussen voornaamwoorden, substantieven en werkwoorden: alleen de 3e persoon enkelvoud komt voor. Het is heel wel mogelijk de grammatica uit te breiden zo dat ie met meervouden en andere persoonsvormen kan omgaan, maar dan groeit hij geweldig in omvang. Contextvrije grammatica's zijn, zoals de naam aangeeft, niet het meest geschikte formalisme om contextuele samenhang uit te drukken. In de taalkunde, waar CF grammatica's onder allerlei gedaanten optreden, worden ze dan ook meestal aangevuld met een vorm van parameter-mechanisme om met context om te gaan.

Ook in de informatica zijn een aantal uitbreidingen van de contextvrije grammatica's uitgevonden, zoals van Wijngaarden's twee-niveau Grammatica's [ALG68], Attribuut Grammatica's [KNU68] en Affix Grammatica's [KOS71, KOS91]. Maar wederom, het zou te ver voeren om dit hoogst interessante onderwerp hier verder uit te diepen.

11.7 De analyse voorbij

Het vertalen van een zin van de ene taal naar de andere kan worden gezien als het probleem, de zin eerst te analyseren volgens de syntax (en semantiek) van de ene taal en daarna een equivalente zin in de andere taal te genereren. Dit proces lijdt onder alle complicaties die we net noemden, plus nog een aantal meer: hoe kun je de betekenis van een zin zodanig afleiden en representeren dat er een equivalente zin kan worden geproduceerd?

Het probleem van de "machinale vertaling" behoort tot de oudste toepassingen van computers, reeds in de vijftiger jaren toen die dingen vaak als "electronisch brein" werden gezien. Na enige aanvankelijke successen en tegenslagen bewees Bar-Hillel tegen het eind van de vijftiger jaren de ontoereikendheid van de beschikbare theorie en methoden [BAR60]. Er volgde een wat nuchterder periode, waarin de taalkundigen zich en masse bekeerden tot de pragmatiek - zij vonden syntax te beperkt en semantiek te ingewikkeld. In de zestiger en zeventiger jaren werden er binnen de informatica machtiger analyse- en vertaalmethoden bedacht, en de computers werden zoveel sneller dat er een nieuwe belangstelling is ontstaan voor de automatische verwerking van natuurlijke talen en voor de machinale vertaling.

Allerlei toepassingen vragen om taalkundige interfaces. Met de komst van de sprekende chip verkrijgen de analyse en synthese van menselijke spraak en (geschreven) taal een overweldigend belang. Dit hoofdstuk is bedoeld als een bescheiden introductie tot de taalkundige toepassingen van computers. Tegelijkertijd biedt het een prachtige gelegenheid, procedures aan het werk te zien in een wat groter voorbeeld.

11.8 Opgaven

1. (roulette) Meier en de Boer spelen roulette. Beide spelen uitsluitend op rood respectievelijk zwart. Hun doel is, hun beginkapitaal te verdubbelen. Beide beginnen met eenzelfde startkapitaal en basisinzet 1. Meier reageert op verlies door zijn inzet te verdubbelen. Bij winst gaat hij terug naar de basisinzet. De Boer denkt nog slimmer te zijn. Hij verhoogt bij verlies zijn inzet met 1 en gaat bij winst terug naar de basisinzet. De kans op rood is

even groot als die op zwart, maar niet helemaal 50 % omdat met kans 1/37 de bal in het vakje 0 terecht komt, dat noch rood noch zwart is. Schrijf een programma dat een willekeurige avond op het casino simuleert. Probeer het met als beginkapitalen bij de spelers 10 en 100. De bank heeft willekeurig veel geld.

2. (fruitautomaat) Simuleer een fruitautomaat. Ter verhoging van het realisme kan enig veldwerk (in kroegen en taveernen waar die dingen hangen) onvermijdelijk blijken.
3. (beschrijven) Beschrijf, via een contextvrije grammatica, de structuur van
   - een trein (uit wagons, locomotief, kolenwagen, remwagen)
   - het nederlandse leger, voor zover je die informatie kunt achterhalen (legerkorpsen, divisies, bataljons, compagnieën, pelotons)
   - een contextvrije grammatica.
4. (poëzie) Schrijf een generatieve grammatica voor het voortbrengen van vrije poëzie, vol sterk beeldende zelfstandige naamwoorden, bijvoeglijke naamwoorden en werkwoorden (dieren, zee, kleuren, mensen, triestheid, geluk).
5. (brief) Schrijf een generatieve grammatica die, in de uiterlijke vorm van een keurige brief, een reeks van beledigingen voortbrengt. Wijd daartoe uit over de afkomst van betrokkene, zijn gewoonten, geestelijke en lichamelijke gezondheid en toekomst.

Chapter 12

Recursieve algoritmen

Vaak bevat een voor de hand liggende formulering van een algoritme op zijn beurt weer een oproep van de algoritme: het probleem wordt herleid tot een eenvoudige versie van hetzelfde probleem. In de wiskunde is het een gebruikelijke gang van zaken bijvoorbeeld bepaalde reeksen te beschrijven door middel van recurrente betrekkingen zoals

$$t_n = t_{n-1} * xDIVn.$$

Wanneer een algoritme zichzelf oproept spreekt men van **recursie**. We noemen een algoritme *direct recursief* als hij een oproep van zichzelf bevat, en *indirect recursief* als hij een algoritme oproept die hem weer (direct of indirect) oproept.

We zullen het gebruik van recursie als een programmeertechniek invoeren en illustreren aan de hand van enige voorbeelden.

12.1 Recursie

Om een recursieve oplossing van een probleem te geven zoeken we naar

- een (of meer) *eenvoudigste geval*(len), waarin een niet-recursieve oplossing voor de hand ligt
- een (of meer) *recursieve geval*(len), waarin een ingewikkelde versie van het probleem wordt herleid tot een eenvoudiger versie van hetzelfde probleem.

Een verwantschap met de methode van mathematische inductie ligt voor de hand.

12.1.1 Voorbeeld: print getal

We willen een procedure `print getal` schrijven, zo dat `print getal(x)` de integer `x` in een minimaal aantal posities afdrukt. We veronderstellen de procedure `print cijfer`, die een enkel cijfer afdrukt, als gegeven.

Een positief getal onder de 10 kunnen we direct met `print cijfer` afdrukken; dat is het *eenvoudigste geval*. Van een getal van `n` cijfers, met `n>1`, kunnen we een cijfer afhalen, zodat we een getal van `n-1` cijfers overhouden; een *eenvoudiger geval* van dezelfde algoritme.

We kunnen proberen dit cijfer aan de *voorkant* van het getal te verwijderen, maar moeten dan met machten van 10 gaan stoethaspelen om `n` te bepalen, met allerlei gevaar voor overflow; hebben we dan `n` eenmaal bepaald, dan verkrijgen we de achtereenvolgende cijfers door delen door `10**(n-1)`, `10**(n-2)`, enzovoorts — al met al weinig aantrekkelijk.

We kunnen het cijfer ook aan de *achterkant* van het getal afhalen, door eenvoudig het getal modulo 10 te nemen. Daar zitten we dan, met in onze hand een cijfer dat we pas mogen afdrukken als alle voorafgaande cijfers zijn afgedrukt. Hoe drukken we dan die voorafgaande? Wel, op dezelfde manier: het gaat om een cijfer minder dan we eerst hadden en dus om een eenvoudiger geval van hetzelfde probleem.

Een kleine complicatie is het feit dat een integer negatief kan zijn. In dat geval drukken we gewoon een minteken af, gevolgd door het tegengestelde van het getal, dat dus positief is.

```
PROC print getal (INT getal):
  IF getal is negatief
  THEN
    druk minteken af;
    druk tegengestelde getal af
  ELIF getal bestaat uit 1 cijfer
  THEN
    druk het cijfer af
  ELSE
    druk de voorafgaande cijfers af;
    druk het achterste cijfer af
  FI.

  getal is negatief:
    getal < 0.

  druk minteken af:
    put("-").

  druk tegengestelde getal af:
    print getal ( -getal).

  getal bestaat uit 1 cijfer:
    getal < 10.

  druk het cijfer af:
    print cijfer(getal).

  druk de voorafgaande cijfers af:
    print getal(getal DIV 10).

  druk het achterste cijfer af:
    print cijfer(getal MOD 10).

ENDPROC print getal;
```

Tot slot nog een mogelijke definitie voor `print cijfer`:

```
PROC print cijfer (INT waarde):
  IF waarde < 0 OR waarde > 9
  THEN put ("?")
  ELSE put ("0123456789" SUB (waarde + 1))
  FI
ENDPROC print cijfer;
```

We hebben nu een procedure voor het zuinig afdrukken van een integer. De gevonden oplossing is wellicht niet al te efficiënt, omdat de test `getal is negatief` nodeloos vaak wordt uitgevoerd, maar wel is de werking duidelijk. Merk op dat in de hele procedure geen variabele en geen toewijzing voorkomt. Dat draagt niet weinig bij tot de doorzichtige structuur.

Om een inzicht te krijgen in de werking van deze procedure, zullen we de aanroep `print getal(473)` in detail bekijken.

```
print getal(473)
  |
  | INT CONST getal:: 473
  | print getal(473 DIV 10)
  .   |
  .   | INT CONST getal:: 47
  .   | print getal(47 DIV 10)
  .   .   |
  .   .   | INT CONST getal:: 4
  .   .   | print cijfer(4)          4
  .   .   |
  .   | print cijfer(47 MOD 10)      7
  .   |
  | print cijfer(473 MOD 10)         3
```

We zien dat op het hoogtepunt van de activiteit drie exemplaren van `print getal` actief zijn. Zo'n "versie in uitvoering" van een algoritme, met zijn eigen locale gegevens, noemt men een **incarnatie** van die algoritme.

Een niet-recursieve algoritme, gebaseerd op herhaling in plaats van recursie, noemt men een **iteratieve algoritme**. Probeert men van deze algoritme een iteratieve versie te geven — een schets daarvan hebben we al gegeven — dan blijkt deze beduidend bewerkelijker in zijn formulering en als gevolg daarvan zal hij in eerste instantie wel vol met fouten zitten.

De voorwaarde `getalbestaatuit1cijfer` speelt een centrale rol bij de terminatie van de algoritme: als aan deze voorwaarde is voldaan wordt geen verdere recursieve oproep gedaan. Wij noemen dit de **afbreekvoorwaarde** van de algoritme.

Het vinden van een geschikte afbreekvoorwaarde (het "eenvoudigste geval") is veelal de sleutel tot het vinden van een recursieve oplossing.

Achteraan dit hoofdstuk vindt u een aantal kleine opgaven waarvoor een recursieve oplossing voor de hand ligt, en waarvan U gevraagd wordt de afbreekvoorwaarde te zoeken.

12.2 Meervoudige recursie

We noemen een algoritme *enkelvoudig recursief* als iedere incarnatie in de recursie-stap hoogstens één verdere incarnatie van de algoritme voortbrengt. Dat geldt dus met name voor een direct recursieve procedure die slechts een oproep van zichzelf bevat, buiten een herhaling. Deze zijn betrekkelijk eenvoudig iteratief te maken (zie hoofdstuk 16). Het vorige voorbeeld was enkelvoudig recursief.

Figure 12.1: De torens van Hanoi

We bekijken nu een aantal algoritmen die meer dan een recursieve oproep bevatten. Met name het eerste voorbeeld (torens van Hanoi) is opgenomen wegens zijn hoge didactische waarde, niet om zijn practisch belang. In latere hoofdstukken te geven voorbeelden van *meervoudig recursieve* algoritmen, zoals Quicksort (zie 14.2) tonen aan dat met behulp van recursie korte, lucide en efficiënte oplossingen te geven zijn voor practische problemen die heel moeilijk te vinden zijn als iteratieve algoritme.

12.2.1 Voorbeeld: De torens van Hanoi

Een van de oudste geschiedenissen uit de informatica is de legende van de torens van Hanoi, die hier wordt weergegeven zoals de auteur hem hoorde van Leo Geurts, vele jaren geleden.

Volgens deze legende stonden er, heel lang geleden, voor een tempel in Hanoi drie zuilen; de eerste was gemaakt van koper, de tweede van zilver en de derde van goud. Op de koperen zuil bevonden zich honderd porphyren schijven. Van onder naar boven namen de schijven steeds in doorsnede af.

Een oude monnik had zich tot taak gesteld, de toren schijven van de koperen zuil naar de gouden zuil te dragen, schijf voor schijf, door telkens de bovenste schijf van een zuil weg te nemen en aan een andere zuil te rijgen, ervoor zorgdragend dat nooit een grotere schijf op een kleinere terecht kwam. Als de monnik met zijn werk klaar was zou, volgens de legende, het einde van de wereld daar zijn.

De monnik zag al gauw in, dat hij bij zijn opgave ook de zilveren zuil moest gebruiken. Dus zette hij zich aan zijn bureau om een plan op te stellen.

Hij mediteerde en dacht na, dacht na en mediteerde, en kreeg opeens een ingeving: hij kon zijn probleem in drie stappen oplossen:

1. Transporteer de toren, bestaande uit de bovenste 99 schijven van de koperen naar de zilveren zuil.

2. Transporteer de laatste, grootste, schijf van de koperen naar de gouden zuil.
3. Transporteer tot slot de toren van 99 schijven van de zilveren naar de gouden zuil.

Bij het bekijken van dit schema bemerkte de monnik, dat Stap 1 en 3 wel buitengewoon moeizaam zouden zijn; en omdat hij niet slechts een oude, maar ook een wijze monnik was, besloot hij deze stappen door zijn oudste leerling te laten uitvoeren. Als deze met de eerste stap klaar was, zou onze monnik zelf de grote schijf van de koperen naar de gouden zuil dragen; en dan zou hij nogmaals een beroep doen op de diensten van zijn oudste leerling.

Om zijn oudste leerling, die zelf al op leeftijd was, niet te veel werk te bezorgen, wilde hij hem dit plan meedelen, zodat ook deze eenvoudig werk had.

De algoritme, die de monnik de volgende dag aan de tempeldeur aansloeg, vertalen we hier uit het Oud-Vietnamees:

> Aard ende wijze, een toren van n schijven te verplaatsen van de ene naar de andere zuil onder gebruik van de derde zuil:
> Zo de toren uit meer dan een schijf bestaat, verzoek dan uw oudste leerling een toren van $n-1$ schijven te verplaatsen van de ene naar de derde zuil onder gebruik van de andere zuil.
> Draag dan zelf een schijf van de ene naar de andere zuil.
> Zo de toren uit meer dan een schijf bestond, verzoek dan uw oudste leerling, een toren van $n-1$ schijven te verplaatsen van de derde naar de andere zuil onder gebruik van de ene zuil.

Nadat de monnik dit document had vastgespijkerd, rustte hij even uit; na zich aldus te hebben verkwikt, vroeg hij zich af wat hij nu moest doen: o ja, hij moest een toren van 100 schijven verplaatsen van de koperen naar de gouden zuil onder gebruik van de zilveren zuil. En omdat hij na de zware denkarbeid van de vorige dag ietwat verstrooid was, herinnerde hij zich niet precies meer hoe dat gedaan moest worden. Maar toen hij voor de tempeldeur een hoop mensen zich zag verdringen om iets te lezen, wist hij wat hem te doen stond. Vastbesloten drong ook hij door tot aan de tempeldeur en begon te lezen.

En aldus riep hij zijn oudste leerling tot zich en verzocht hem, een toren van 99 schijven te verplaatsen van de koperen naar de zilveren zuil onder gebruik van de gouden zuil en zich daarna weer bij hem te melden.

Vragen:

- Wat is het eerste dat deze monnik nu doet?
- Hoeveel monniken zijn er aan het werk gezet voor de eerste schijf wordt verplaatst?
- Hoe kan men het werk van de k-de monnik karakteriseren?
- Wat doet onze oude monnik wanneer zijn oudste leerling zich eindelijk meldt?

Wij formuleren nu de algoritme van de monnik in Elan. De zuilen geven we aan met hun namen in de vorm van teksten, de schijven met hun nummers (de grootste schijf het hoogste nummer).

```
PROC transporteer toren
       (INT n, TEXT ene, andere, derde):
  IF   n>1
  THEN transporteer toren
       (n-1, ene, derde, andere)
  FI;
  verplaats schijf (n, ene, andere);
  IF   n>1
  THEN transporteer toren
       (n-1, derde, andere, ene)
  FI
ENDPROC transporteer toren;
```

Het verplaatsen van een schijf geven we aan door simpelweg af te drukken, welke schijf van welke zuil naar welke zuil gedragen wordt.

```
PROC verplaats schijf
       (INT n, TEXT ene, andere):
  put line ("Schijf"); put (n);
  put (" van de "); put (ene);
  put (" naar de "); put (andere)
ENDPROC verplaats schijf;
```

Hiermee luidt de oplossing van het gestelde probleem:

```
transporteer toren (100,
     "koperen", "gouden", "zilveren")
```

De procedure `transporteer toren` bevat twee direct recursieve oproepen en is daarmee meervoudig recursief. De voorwaarde `n>1` is hier de ontkenning van de afbreekvoorwaarde. Een variant waarin de afbreekvoorwaarde explicieter is maar die minder getrouw het oudvietnamese origineel volgt luidt:

```
PROC transporteer toren
       (INT n, TEXT ene, andere, derde):
  IF n>0
  THEN
    transporteer toren
       (n-1, ene, derde, andere);
    verplaats schijf (n, ene, andere);
    transporteer toren
       (n-1, derde, andere, ene)
  FI
ENDPROC transporteer toren;
```

De recursie eindigt nu op het niveau `n=0`, waar niets meer te doen valt.

12.2.1.1 Complexiteit van de algoritme

Hoe vaak moet er een schijf verplaatst worden om een toren van hoogte n te verplaatsen? Noem dit getal S_n.

$$\begin{aligned} S_1 &= 1 \\ S_2 &= 1 + 2 * S_1 = 3 \\ &\dots \\ S_n &= 1 + 2 * S_{n-1} \text{ voor } n > 1. \end{aligned}$$

Testen we de hypothese:

$$
\begin{aligned}
S_i &= 2^i - 1? \\
1 + 2 * S_{i-1} &= 2^i - 1? \\
1 + 2 * (2^{i-1} - 1) &= 2^i - 1? \\
1 + (2^i - 2) &= 2^i - 1? \\
2^i - 1 &= 2^i - 1 \\
qed &
\end{aligned}
$$

De hoeveelheid werk neemt dus exponentieel toe met n, het aantal schijven.

Vraag: Als alle monniken zeer vlijtig werken, zodat iedere seconde een schijf getransporteerd wordt, hoe lang duurt het dan tot het einde van de wereld?

12.2.1.2 Voorbeeld van de uitvoer

Tot slot een voorbeeld van de uitvoer van deze algoritme:

```
(n=4)
Schijf 1 van de koperen naar de zilveren
Schijf 2 van de koperen naar de gouden
Schijf 1 van de zilveren naar de gouden
Schijf 3 van de koperen naar de zilveren
Schijf 1 van de gouden naar de koperen
Schijf 2 van de gouden naar de zilveren
Schijf 1 van de koperen naar de zilveren
Schijf 4 van de koperen naar de gouden
Schijf 1 van de zilveren naar de gouden
Schijf 2 van de zilveren naar de koperen
Schijf 1 van de gouden naar de koperen
Schijf 3 van de zilveren naar de gouden
Schijf 1 van de koperen naar de zilveren
Schijf 2 van de koperen naar de gouden
Schijf 1 van de zilveren naar de gouden
```

In de schijfnummers is een regelmaat te ontdekken, die een aardig verband heeft met tellen in het tweetallig stelsel.

12.3 Voorbeeld: De muis in de doolhof

In het voorbeeld van de torens van Hanoi hebben we ons voorgesteld dat iedere monnik een andere, namelijk zijn oudste leerling, een opdracht kan geven, die dan weer op dezelfde wijze te werk gaat.

Dit is uitstekend voor het begrip van recursie en helemaal niet zo ver weg van de machinale uitvoering van de algoritme: iedere incarnatie van de procedure `transporteer toren` kan verdere incarnaties van dezelfde procedure in het leven roepen. Als, in ons model, vier monniken tegelijk bezig zijn met het vervullen van hun opdracht en op een na allemaal op een ander staan te wachten, dan bestaan er ook vier incarnaties van de procedure, waarvan alleen de jongste actief is.

Een verschil is, dat het aantal monniken met leerlingen, en ook de mate van hun geduld en plichtsbesef, in de werkelijkheid sterk begrensd is. Het getal der mogelijke incarnaties van een procedure is daarentegen vrijwel onbeperkt: de processor kan willekeurig veel nieuwe incarnaties aanmaken en oude vergeten, zolang er niet méér dan een bepaald aantal gelijktijdig bestaan (afhankelijk van de gebruikte machine, de beschikbare geheugenruimte en details van de gebruikte implementering van de programmeertaal, in de orde van enkele honderden tot vele duizenden incarnaties).

Figure 12.2: Muis in doolhof

Desondanks is het zeer aanschouwelijk, het (recursieve) aanroepen van een procedure te vergelijken met het geven van een opdracht door een persoon aan een ander. Wij geven nu een tweede voorbeeld, waarin een antropomorf beeld tot het begrip bijdraagt.

Een muis wil in een doolhof een kortste weg vinden van een gegeven punt `A`, naar een ander punt `kaas`. (Let er op dat er meer kortste wegen van dezelfde lengte kunnen zijn. Welke we daaruit kiezen is niet van belang.)

Een *doolhof* stellen we ons voor als een rechthoek van $n * m$ velden. Ieder veld, behalve dat waarop de muis zit of de kaas ligt, is óf met een muur opgevuld óf vrij. Denken we ons de velden aangeduid door hun positie (x, y) met $1 <= x <= n$ en $1 <= y <= m$.

Onder een *weg* van lengte k verstaan we een rij van k velden $F_i = (x_i, y_i)$, zodanig dat voor $1 < i <= k$ geldt dat F_i met F_{i-1} precies één zijde gemeen heeft:

$$
\begin{aligned}
1 < i <= k \quad \Rightarrow \quad & x_{i-1} = x_i \wedge y_{i-1} = y_i + -1 \\
\vee \quad & x_{i-1} = x_i + -1 \wedge y_{i-1} = y_i
\end{aligned}
$$

Een *lusvrije* weg is een weg zodanig dat

$$i \neq j \quad \Rightarrow \quad x_i \neq x_j \vee y_i \neq y_j$$

Een *succesvolle* weg van lengte k is een weg zodanig dat

$$
\begin{aligned}
(x_1, y_1) &= A \\
(x_i, y_i) &= vrij \ (2 <= i <= k - 1) \\
(x_k, y_k) &= kaas
\end{aligned}
$$

12.3.1 De lengte van een kortste weg

Gevraagd wordt, de lengte van een kortste succesvolle weg te bepalen. Ga na dat zo'n weg altijd lusvrij is.

Men kan proberen, eerst alle wegen te berekenen, dan daaruit de succesvolle te bekijken en tenslotte

Figure 12.3: Muis en omgeving

daarvan de kortste te bepalen, maar dat is in de eerste plaats nog niet eens zo eenvoudig iteratief te formuleren en kost bovendien veel werk van de processor.

Idee voor een oplossing: de lengte van een kortste weg vanuit een bepaald veld is één meer dan het minimum van de lengten van de kortste wegen vanuit zijn buurvelden.

Een muis, beland op een zeker veld, stuurt voor elk van de vrije buurvelden een andere muis er op uit, met de opdracht, de lengte van een kortste weg vanaf dat buurveld te bepalen. Zelf blijft de muis op het veld (x, y) zitten totdat die andere hun opdracht voltooid hebben. Geen andere muis kan daardoor over dit veld. Dit garandeert dat elke weg, op deze wijze onderzocht, lusvrij is.

Dan berekent de muis het minimum van de gevonden waarden, telt daar 1 bij op (voor het veld waarop zij zelf zit), bericht deze waarde aan de muis die haar gezonden heeft, en ruimt het veld. Heel eenvoudig, maar het vergt wel veel toegewijde muizen (zij het hoogstens n * m).

Als afbreekvoorwaarde maken we gebruik van het feit, dat als de kaas op een bepaald veld ligt, de weg van dat veld naar de kaas lengte nul heeft. In de vorm van een algoritme:

```
IF   het veld bevat kaas
THEN 0
ELSE stuur vanaf ieder vrij buurveld een
     nieuwe muis met de taak de lengte
     van een kortste weg naar de kaas
     te berekenen en neem het minimum
     van de zo gevonden lengten;
     1 + het minimum
FI
```

De status van een veld (vrij, muur, muis of kaas) zullen we coderen als een integer. Wij declareren daartoe:

```
LET  vrij = 1, muur = 2,
     muis = 3, kaas = 4
```

We zullen de doolhof voorstellen door een rij van rijen:

```
ROW n ROW m INT VAR doolhof;
```

Het gedrag van een muis realiseren we als een procedure die de lengte van een kortste weg vanaf (x, y) naar het doel berekent:

```
INT PROC kortste weglengte vanaf(INT x,y):
  IF   reeds bij de kaas
  THEN 0
  ELSE
    grijp het veld;
    neem het minimum van de 4 richtingen;
    geef het veld vrij;
    1 + minimum
  FI.
```

We kunnen de elementaire doolhofoperaties verfijnen:

```
reeds bij de kaas:
  doolhof [x] [y] = kaas.

grijp het veld:
  doolhof [x] [y] := muis.

geef het veld vrij:
  doolhof [x] [y] := vrij.
```

Merk op, dat de gegeven procedure niet robuust is tegen een aanroep met een veld dat muis of muur bevat; we zullen daarop moeten testen alvorens de procedure aan te roepen.

Als een muis in een doodlopende weg raakt zullen we ook een weglengte moeten berekenen. We kiezen in dit geval een weglengte “oneindig”, dat wil zeggen: groter dan iedere zinvolle weglengte. Daarvoor nemen we niet `max int` (waarom niet?), maar bijvoorbeeld:

```
INT CONST oneindig : n * m;
```

Als er geen weg is van de muis naar de kaas, zal de lengte van de gevonden weg tenminste `oneindig` zijn. Daarmee kunnen we verfijnen:

```
neem het minimum van de 4 richtingen:
  INT VAR minimum :: oneindig;
  een muis naar het noorden;
  een muis naar het zuiden;
  een muis naar het westen;
  een muis naar het oosten.
```

(Terzijde: iedere gelijkenis met het gedicht ”The Rock” van T.S. Eliot berust op puur toeval.)

De volgorde van de laatste vier units speelt eigenlijk geen rol, maar onze sequentiële programmeertaal dwingt ons ertoe een volgorde vast te leggen. (Merk op dat de vier onderzoekstochten niet gelijktijdig kunnen worden uitgevoerd maar alleen *na elkaar*, omdat anders muizen van verschillende expedities elkaar in de war zouden brengen.)

We definiëren twee hulpprocedures die van een te algemeen nut zijn om ze als verfijningen uit te drukken.

```
PROC kortste  (INT VAR min, INT term):
  IF min > term THEN min := term FI
ENDPROC kortste;
```

```
BOOL PROC kan gaan naar (INT x, y):
  IF   binnen doolhof
  THEN doolhof [x] [y] = vrij OR
       doolhof [x] [y] = kaas
  ELSE FALSE
  FI.
binnen doolhof:
  1 <= x AND x <= n AND 1 <= y AND y <= n.
ENDPROC kan gaan naar;
```

Let op de noodzaak, pas dan te testen of een veld vrij is als tevoren gegarandeerd is dat het binnen de doolhof ligt, dus niet:

```
binnen doolhof AND
  (doolhof [x] [y] = vrij OR
   doolhof [x] [y] = kaas)
```

aangezien de indicering met `[x] [y]` buiten de doolhof betekenisloos is en voert tot het afbreken van de programma-uitvoering.

We programmeren heel voorzichtige muizen, bijvoorbeeld:

```
een muis naar het noorden:
  IF   kan gaan naar (x, y+1)
  THEN kortste (minimum,
              kortste weglengte vanaf (x, y+1))
  FI.
```

en de andere richtingen analoog.

We sluiten de procedure af met

```
ENDPROC kortste weglengte vanaf;
```

Het resulterende programma is wat saai, maar niet slecht gestructureerd. Toch heeft het nog een tekortkoming.

12.3.2 Kortste weg — alternatieve benadering

Onder welke voorwaarden werkt de oproep `kortste weglengte vanaf (x,y)`? Aan het volgende moet voldaan zijn:

- de procedure mag uitsluitend aangeroepen worden met $[x][y] \in [1:n][1:m]$, anders ontstaat een indiceringsfout;
- het veld [x] [y] moet kaas bevatten of vrij zijn. We mogen bijvoorbeeld niet op een muur beginnen.

Op deze voorwaarden moet binnen de procedure bij iedere recursieve aanroep worden getest! Hetzelfde geldt ook voor de allereerste aanroep, die buiten de procedure staat, in totaal dus op vijf plaatsen. Zulke tests vragen er natuurlijk om te worden vergeten.

Het is veel verstandiger, direct bij het binnengaan in de procedure te testen en deze daardoor robuust te maken tegen misbruik. Daardoor worden ook de tests vóór het uitzenden van de vier nieuwe muizen overbodig: de muizen zijn slim genoeg, zelf te testen.

Met deze wijzigingen (en nog wat stroomlijning om de twee hulpprocedures overbodig te maken) krijgen we:

```
INT PROC kortste weglengte via (INT x, y):
  IF   NOT binnen doolhof
  THEN oneindig
  ELIF doolhof [x] [y] = vrij
  THEN
    grijp het veld;
    neem het minimum van de 4 richtingen;
    geef het veld vrij;
    1 + minimum,
  ELIF doolhof [x] [y] = kaas
  THEN 0
  ELSE oneindig
  FI.
binnen doolhof:
  1 <= x AND x <= n AND 1 <= y AND y <= n.
grijp het veld:
  doolhof [x] [y] := muis.
neem het minimum van de 4 richtingen:
  INT VAR kw ::
    min (kortste weglengte via (x+1, y),
    min (kortste weglengte via (x-1, y),
    min (kortste weglengte via (x, y+1),
      kortste weglengte via (x, y-1)))).
geef het veld vrij:
  doolhof [x] [y] := vrij.
```

We sluiten de procedure weer af met:

```
ENDPROC kortste weglengte via;
```

Let erop dat de beschreven procedures weliswaar tijdelijk veranderingen in de doolhof aanbrengen door middel van `grijp het veld`, maar dat bij terugkeer telkens de doolhof weer in de originele staat wordt achtergelaten. Zij ruimen dus hun eigen puin op.

De werkwijzen `kortste weglengte vanaf` en `kortste weglengte via` zijn verschillend in de zin, dat de eerste alleen werkt als aan stringente voorwaarden voldaan is, terwijl de tweede robuust is. Een algemene vuistregel is: liever testen bij de ingang van de procedure dan bij ieder van zijn aanroepen.

Natuurlijk hadden we meteen een robuuste versie moeten programmeren. De tweede versie is korter, duidelijker en geeft minder kans op denkfouten bij het programmeren. Toch komen (vooral!) beginners steevast op oplossingen als de eerste. Het is voor ons mensen blijkbaar erg moeilijk voor onze problemen direct de simpelste oplossing te vinden.

12.4 Tussenbalans

Een goed begrip van recursie leidt in vele opzichten tot een dieper inzicht in de programmering.

- Vele problemen hebben van huis uit een constructieve recursieve definitie die als eerste grove formulering voor de algoritme kan dienen.
- Recursie is de meest economische vorm van verfijning: een recursieve algoritme is tevens zijn eigen verfijning.

- Recursie voert tot een economische vorm van denken. Een groot probleem (bijvoorbeeld de analyse van een doolhof) wordt gereduceerd tot een kleiner locaal probleem (de analyse van een vakje in de doolhof) plus het oorspronkelijke probleem in een eenvoudiger vorm. Is het locale probleem correct opgelost en klopt het inductie-schema dan is een correcte oplossing voor het globale probleem gevonden.
- Recursie leidt tot een economische vorm van bewijzen van correctheid en terminatie, want alleen de correctheid van de oplossing van het locale probleem alsmede de inductiestap hoeft te worden bewezen.
- Recursie voert tot een eenvoudige vorm van administratie van tussenresultaten die in locale variabelen kunnen worden bijgehouden en gemanipuleerd, zonder interferentie met globale grootheden.
- Recursie voert tot korte pregnante programma-formuleringen, die overzichtelijker zijn dan equivalente iteratieve oplossingen en daarom minder makkelijk leiden tot fouten.

Natuurlijk betaalt men een prijs voor deze voordelen: in de meeste implementaties wordt een recursief programma wat langzamer uitgevoerd dan een equivalent iteratief programma. Maar dat scheelt maar heel weinig. Bovendien kennen sommige ouderwetse programmeertalen geen recursie.

In hoofdstuk 16 wordt ingegaan op technieken om uit een correcte recursieve algoritme een equivalente iteratieve algoritme af te leiden onder behoud van de correctheid, met in het algemeen een verbetering van de efficiëntie. Enige belangrijke algoritmen zijn zo ontstaan. Door een laag-bij-de-grondse iteratief-denker zouden zij vast nooit zijn gevonden.

12.5 Opgaven

Bedenk afbreekvoorwaarden bij de volgende schetsmatige formuleringen en programmeer ze als recursieve procedures.

1. Zij gegeven `ROW n INT CONST r`. Het maximum van de n elementen uit de rij is het maximum van de waarde van het n-de element en het maximum van de daaraan voorafgaande.
2. Gegeven is `ROW n INT CONST d`. De waarde van een decimaal getal $d_n d_{n-1}...d_1 d_0 (met 0 <= d_i <= 9)$ is de waarde van het laatste cijfer d_0 plus tien keer de waarde van het decimale getal $d_n d_{n-1}...d_1$
3. Gegeven is weer `ROW n INT CONST d`. De waarde van een decimale breuk $0.d_0 d_1...d_n$ is een tiende maal de som van de waarden van het voorste cijfer d_0 en de decimale breuk $0.d_1...d_n$.
4. Gegeven is de `TEXT CONST a`. De rij karakters $a_1 a_2...a_{n-1} a_n$ is symmetrisch, wanneer
   - a_1 gelijk is aan a_n en tevens
   - de rij $a_2...a_{n-1}$ symmetrisch is.
5. a tot de macht n is a maal (a tot de macht $n-1$).
6. Gegeven is een `ROW n BOOL CONST rij`. De pariteit van de `rij` is per definitie `TRUE` als de rij een even aantal `TRUE` elementen bevat en `FALSE` als de `rij` een oneven aantal `TRUE` elementen bevat. Schrijf een procedure

```
BOOL PROC pariteit (INT k):
```

die recursief de pariteit van de eerste k elementen van de `rij` berekent.

Tot slot volgen nog een paar meervoudig recursieve opgaven.

- 7) (opdelen van een lijn) Op hoeveel wijzen kan ik een lijnstuk van lengte n opdelen in stukken van lengte 1 of 2? (Hint: Er bestaan 2 soorten opdelingen, die waarbij ik met lengte 1 begin en die waarbij ik met lengte 2 begin.)
- 8) (muis in doolhof, herzien) De test of we binnen de doolhof blijven, kunnen we ons besparen door om de rand van de doolhof een muur te bouwen.

```
ROW n ROW m STATUS VAR doolhof;
initialiseer rand als muur
```

Werk dit verder uit.

- 9) (alle wegen) Schrijf een procedure met kop

```
PROC print alle wegen vanaf (INT x, y):
```

die alle lusvrije wegen, beginnend op (x, y) en eindigend bij de kaas afdrukt.
Hint: op het moment dat de kaas bereikt wordt ligt er een lusvrije ketting muizen in de doolhof van het beginpunt tot aan de kaas.

Chapter 13

Computer grafiek

In dit hoofdstuk introduceren we eerst enkele begrippen uit de computer grafiek en daarna een verzameling Elan-algoritmen voor computer grafiek, met voorbeelden van hun toepassing. Tot slot beschrijven we een aantal recursieve algoritmen voor het tekenen van plaatjes.

computer grafiek ("computer graphics") behelst de verwerking en presentatie van visuele informatie met de computer. De technologie voor de verwerking en presentatie van gegevens in grafische vorm ontwikkelt zich snel. Bij de vormgeving van gebruikersvriendelijke mens-machine interfaces speelt computer grafiek een steeds belangrijker rol.

Het tekenen van plaatjes op het computerscherm is een aardige oefening in het systematisch programmeren. Met name bestaan er een aantal fraaie families van recursieve tekeningen, die aardige voorbeelden opleveren van het ontwerp van recursieve algoritmen. Voor een diepgaander studie van de computer grafiek verwijzen we naar leerboeken als [SAS77] [FDF90]. Grafische softwarepakketten kunnen goede voorbeelden leveren voor de modulaire en gelaagde struktuur van grote programma's.

13.1 De fysieke laag

Het bestaande assortiment aan apparatuur voor grafische invoer en uitvoer is heel groot. We noemen slechts een aantal grafische uitvoerapparaten: plotters om tekeningen op papier te maken met een mechanisch gestuurde pen; kathode buis (CRT) schermen volgens het televisie principe: plaatjes bestaan uit horizontale lijnen die op hun beurt bestaan uit zwart/wit of gekleurde punten — deze methode heet **raster grafiek**; CRT schermen bestuurd door aparte processoren die de plaatjes zelfstandig met zeer hoge snelheid aanpassen, of hun armzalige oudere broers waarin een enkele processor zowel het produceren van plaatjes als het rekenwerk moet doen, met als gevolg dat het plaatje zo nu en dan wegvalt. Er is een heel arsenaal aan apparaten te koop — het enige dat nog toekomstmuziek lijkt is de 3-dimensionale holografische kleurenplotter.

We beperken ons tot het grafisch beeldscherm waarmee de meeste microcomputers zijn uitgerust. Het **beeldscherm** is een rechthoek bestaande uit punten, **pixels** (= picture elements) genaamd. Iedere pixel kan een kleur aannemen uit een vast palet (zwart of wit; een of andere grijstint; of wellicht een van 100.000 japanse kleuren). Wij beperken ons nog verder tot monochrome beeldschermen, die slechts wit en zwart onderscheiden.

Het aantal pixels in een horizontale (resp. verticale) lijn wordt de horizontale (resp. verticale) **resolutie**, het "oplossend vermogen" genoemd. Voor microcomputers is 720x348 een gebruikelijke resolutie, maar grote variaties worden gevonden. Met de genoemde resolutie kun je de struktuur van regels bestaande uit pixels van nabij nog juist onderscheiden, maar van enige afstand zijn de puntjes niet te zien.

Op zo'n beeldscherm is het heel eenvoudig horizontale of verticale regels te tekenen. Maar om scheeflopende lijnen (zie Fig. 13.1) te tekenen heb je een of ander slim lijnteken algoritme nodig, bijv. dat van Bresenham, zie [BRE65].

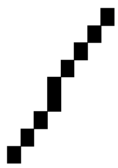


Figure 13.1: Scheve lijn getrokken met de Bresenham algoritme

De situatie wordt vaak gecompliceerd door het feit dat vele CRT schermen een speciale hardware ondersteuning (een "character generator") bezitten voor het vertonen van teksten. Daarom kan tekst output niet worden afgewisseld met graphics output — eerst moet het scherm worden omgeschakeld naar een andere toestand. Het lijkt net als of je twee schermen hebt: een tekst scherm en een grafisch scherm, waartussen je heen en weer moet schakelen.

Een verdere complicatie is dat de hoogte van een pixel meestal verschilt van zijn breedte. Een vierkant, samengesteld uit een gelijk aantal pixels in beide richtingen, ziet er op het beeldscherm dan ook meestal niet vierkant maar rechthoekig uit. De verhouding tussen de verticale en horizontale lengte eenheid noemt men het **aspect**. Om vertekening van het beeld te vermijden moet met het aspect rekening worden gehouden.

13.2 Integer grafiek

De onderste laag van de grafische operaties van Elan wordt de **integer grafiek** genoemd, omdat het scherm wordt geadresseerd in integer coördinaten. Deze laag vormt een uniform interface naar de grafische hardware, die zelf van machine tot machine verschilt, en garandeert daardoor een consistent gedrag van programma's.

De integer grafiek is een (machine afhankelijk) pakket dat deel uitmaakt van de standaard bibliotheek van iedere Elan Programmeer Omgeving. We beschrijven hier wat dit pakket de gebruiker te bieden heeft zonder te vertellen hoe het geïmplementeerd is.

Voordat je een van de volgende operaties mag gebruiken moeten eerst bepaalde verborgen variabelen geïnitialiseerd worden en het scherm worden voorbereid, door een oproep van

```
PROC enter graphics mode:
  {wis het scherm en schakel naar graphics
mode}
ENDPROC enter graphics mode;
```

Tot slot moet de graphics mode verlaten worden door een aanroep van

```
PROC enter text mode:
  {wis het scherm en schakel naar text
mode}
ENDPROC enter text mode;
```

Het tekenen gebeurt steeds op een bepaalde positie binnen het gebied

$$[1 : \textit{graphics x limit},\ 1 : \textit{graphics y limit}]$$

op het grafische scherm.

```
LET graphics x limit = 720,
    graphics y limit = 348,
    aspect = 1.35;
```

Deze waarden, typisch voor de Hercules kaart van de IBM PC, zijn machine afhankelijk; het aspect 1.35 betekent dat de pixels hoger zijn dan ze breed zijn.

De huidige schermpositie kan worden gestuurd door het effect van de verschillende grafische acties. Bij het ingaan van de grafische toestand is de beginwaarde van de huidige schermpositie gelijk aan $(1, 1)$, wat overeenkomt met de linker bovenhoek.

De huidige positie kan worden veranderd door een oproep van

```
PROC move (INT x, y):
  {ga naar positie (x, y)}
ENDPROC move;
```

Als `x` of `y` of beiden buiten de machine afhankelijke grenzen van het scherm vallen, klinkt er een piep en neemt de betreffende coördinaat de waarde van de boven aangegeven grens aan.

Het is natuurlijk mogelijk iets op het scherm te tekenen door telkens een andere plaats op het scherm te kiezen en daar de procedure

```
PROC plot pixel:
  {plot een pixel in de huidige positie}
ENDPROC plot pixel;
```

aan te roepen. Het effect van zo'n oproep hangt af van de kleur van de "penseel". Wij kunnen een kleur uitkiezen door de procedure

```
PROC color (INT c):
  {gebruik voortaan kleur nummer 'c'}
ENDPROC color;
```

aan te roepen. De nummering van de kleuren is sterk machine afhankelijk. Als je niet uit een manual kunt opmaken wat de kleurnummers betekenen op jouw machine kun je er wellicht beter vanaf blijven. Na de eerste oproep van `enter graphics mode` is de penseel wit (of lichtgrijs of lichtgroen) en de achtergrondkleur van het scherm zwart.

Met behulp van deze procedures kun je eigenlijk al alles tekenen op het scherm, maar hun gebruik is niet bepaald gemakkelijk vanwege de vele saaie details. Daarom bestaat er ook een procedure, voor het tekenen van lijnen, die alle noodzakelijke details voor je regelt. Deze is zelf gerealiseerd via `plot pixel` en maakt gebruik van de Bresenham algoritme.

```
PROC draw (INT x, y):
  {trek een zo recht mogelijke lijn}
  {vanaf de huidige positie naar (x, y)}
  {en plot onderweg alle pixels}
ENDPROC draw;
```

De gewone procedures `get`, `put` en `line` voor invoer en uitvoer werken ook in graphics mode, maar het gebruikte type font verschilt vermoedelijk van dat in text mode: In de grafische toestand worden alle tekens pixel voor pixel apart getekend, inplaats van te worden voortgebracht door een "character generator".

Zoals je je wel kunt voorstellen, bestaat ieder karakter uit pixels gearrangeerd in de vorm van een rechthoek (zie Fig. 12.2). Zowel horizontaal als verticaal is vanwege de leesbaarheid een tussenruimte nodig tussen twee aangrenzende letters. De ruimte waarin een teken wordt weergegeven, tezamen met de tussenruimten, is constant maar machine afhankelijk; de hoogte daarvan wordt gegeven door de constante `line height`, de breedte door `character width`. (Op de IBM PC, bijvoorbeeld, is `line height` gelijk aan 11 en `character width` gelijk aan 8.) Het referentiepunt van een teken is zijn *linker bovenhoek*. Als zijeffect van de uitvoering

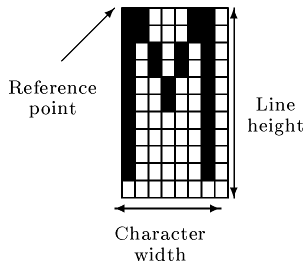


Figure 13.2: Letter M in karakter matrix

van `draw` en de in/uitvoer procedures wordt telkens de huidige positie aangepast. Deze wordt bijgehouden in een verborgen variabele (of wellicht een paar verborgen variabelen) in het pakket dat het grafische interface realiseert.

De procedure `line` bewerkstelligt in grafische mode de overgang naar het begin van de volgende regel op het grafische scherm. De positie gaat daarbij `line height` pixels omlaag en naar de uiterste linkerkant van de regel. Uitgaand van de positie *(x, y)*, is het effect van deze procedure dus hetzelfde als dat van `move (1, y + line height)`.

In het vervolg introduceren we een aantal voorbeeld programma's die plaatjes tekenen op het grafische scherm. de programma's onafhankelijk te houden van de machine gebruiken we geen absolute coördinaten maar enkel relatieve coördinaten in termen van `graphics x limit` en `graphics y limit`.

13.2.1 Nuttige hulpprocedures

Een oproep van `enter text mode`, of de alternatieve vorm `leave graphics mode` die is gedefinieerd als

```
PROC leave graphics mode:
  enter text mode
ENDPROC leave graphics mode;
```

maakt onmiddellijk het grafische scherm schoon. Het is handig om een procedure `wait for confirmation` in te voeren die wacht tot de gebruiker de spatiebalk aanraakt en hem daarmee tijd geeft om zijn zo juist geproduceerde plaatje te bekijken.

Deze procedure krijgt twee parameters, de coördinaten van de positie op het scherm waar de mededeling `Hit space!` dient te verschijnen.

```
PROC wait for confirmation (INT x, y):
  move (x, y);
  put ("Hit space!");
  TEXT t :: inchar
ENDPROC wait for confirmation;
```

Merk op dat ieder karakter wordt geaccepteerd, niet alleen de spatie.

Voor de volgende voorbeelden introduceren we twee handige procedures voor de invoer van getallen:

```
INT PROC ask int (TEXT message):
  INT VAR x;
  put (message);
  get (x);
  x
ENDPROC ask int;

REAL PROC ask real (TEXT message):
  REAL VAR x;
  put (message);
  get (x);
  x
ENDPROC ask real;
```

De trigonometrische functies in de standaard bibliotheek verwachten als parameter een real, namelijk de hoek in radialen. Bij de integer grafiek is het natuurlijker om deze hoek in graden op te geven, als een integer. Daarom declareren wij:

```
REAL PROC sin (INT a):
  sin (pi * real (a) / 180.0)
ENDPROC sin;

REAL PROC cos (INT a):
  cos (pi * real (a) / 180.0)
ENDPROC cos;
```

13.2.2 Voorbeeld: Radar

Het eerste programma tekent een aantal lijnen van eenzelfde lengte, ontspringend uit een punt ongeveer in het midden van het scherm. In het ronde patroon dat op deze manier ontstaat is het effect van de toegepaste lijnteken algoritme duidelijk te herkennen. Dit geeft hele specifieke inferentiepatronen (Fig. 12.3).

```
radar:
  enter graphics mode;
  print heading;
  determine parameters;
  draw radar;
  wait for confirmation
     (2 * graphics x limit DIV 3, 1);
  leave graphics mode.

print heading:
  put ("Radar");
  line;
  put ("=====").

determine parameters:
  REAL radius ::
    real (min (graphics x limit,
               graphics y limit)) / 2.1;
  INT centre x :: graphics x limit DIV 2,
      centre y :: graphics y limit DIV 2.

draw radar:
  INT VAR i;
  FOR i FROM 0 UPTO 359
  REP
    move (centre x, centre y);
    draw (centre x +
        round (aspect * sin (i) * radius),
          centre y +
        round (cos (i) * radius))
  ENDREP.
```

13.2.3 Voorbeeld: Bewegend radar

Het volgende programma tekent dezelfde soort van lijnen, maar deze keer ontspringen zij uit een bewegend centrum.

```
moving radar:
  enter graphics mode;
  print heading;
  determine parameters;
  draw moving radar;
  wait for confirmation
     (2 * graphics x limit DIV 3, 1);
  leave graphics mode.
```

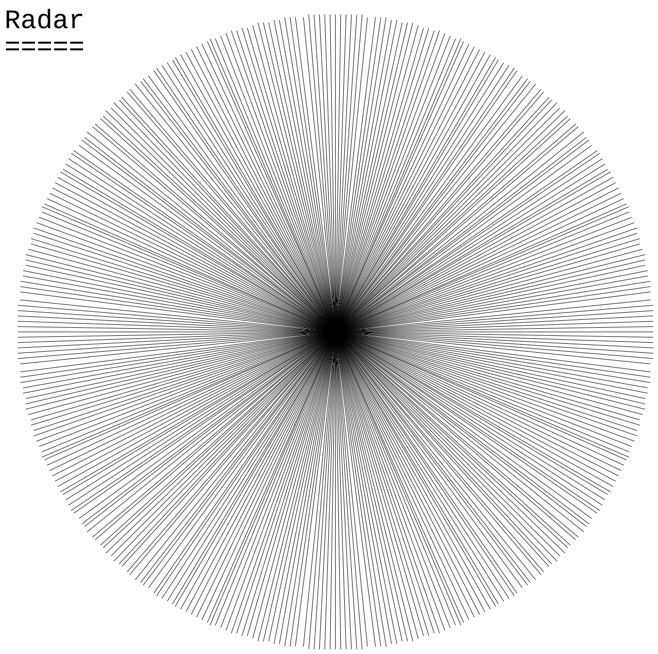


Figure 13.3: Radar

```
print heading:
  put ("Moving radar");
  line;
  put ("============");
  line.

determine parameters:
  REAL VAR radius:: ask real("Radius? ");
  radius := min (radius,
     real (min (graphics x limit,
               graphics y limit)) / 2.1);
  put ("Radius used: ");
  put (text (int (radius), 3));
  REAL step x :: min (1.0,
        real (graphics x limit) / 360.0);
  REAL VAR centre x ::
         max (0.0, radius - 260.0);
  INT centre y ::
         graphics y limit - int (radius).

draw moving radar:
  INT VAR i;
  FOR i FROM 0 UPTO 360
  REP
    move (round (centre x), centre y);
    draw (round (centre x +
       aspect * sin (i) * radius),
      centre y + round (cos (i)*radius));
    centre x INCR step x
  ENDREP.
```

Om het bestuderen van verschillende interferentiepatronen mogelijk te maken vraagt het programma de gebruiker om de straal die moet worden gebruikt. Het programma beperkt deze waarde zo dat de tekening altijd binnen het grafische scherm blijft. De experimenteel gevonden waarde 2.1 wordt gebruikt om genoeg plaats te laten voor de mededelingen.

Net zoals in het vorige voorbeeld willen we een volledige draai van 360 graden maken, wat betekent dat we ook 360 horizontale stapjes moeten nemen, dwars over het scherm. Op bepaalde beeldschermen (en ook bij de laser drukker die dit boek produceerde) is `graphics x limit` kleiner dan 360. Daarom introduceren we de variabele `step x` met een waarde 1.0 of kleiner, naar gelang nodig.

De coördinaten van het middelpunt staan in `centre x` en `centre y`. De eerste is een *reële variabele* aangezien het centrum horizontaal moet bewegen in stappen die wellicht kleiner uitvallen dan 1. Om te beginnen krijgt `centre x` de waarde 0.0 of (voor grote straal) een waarde die de tekening binnen het grafische scherm houdt. De nogal willekeurig gekozen waarde 260.0 vloeit voort uit het feit dat (voor grote straal) het meest linkse punt in ongeveer stap 260 wordt bereikt. De beginwaarde van `centre y` is zo gekozen dat genoeg plaats overblijft voor mededelingen bovenaan het scherm, zelfs bij een grote straal.

13.2.4 Voorbeeld: Mondriaan

Als een laatste voorbeeld laten we zien hoe met behulp van integer grafiek plaatjes kunnen worden voortgebracht die doen denken aan de schilderijen van Mondriaan. Dit programma is niet bedoeld als een serieuze bijdrage aan de computer kunst, hoewel het wel iets toont van de manier waarop sommige kunstenaars de computer als werkgereedschap gebruiken. In de computer kunst wordt een uitgebreid gebruik gemaakt van random getallen, met behulp waarvan de kunstenaar het randgebied tussen chaos en orde kan bestuderen.

Na vergelijking met het ontstane plaatje moet dit programma eenvoudig te begrijpen zijn.

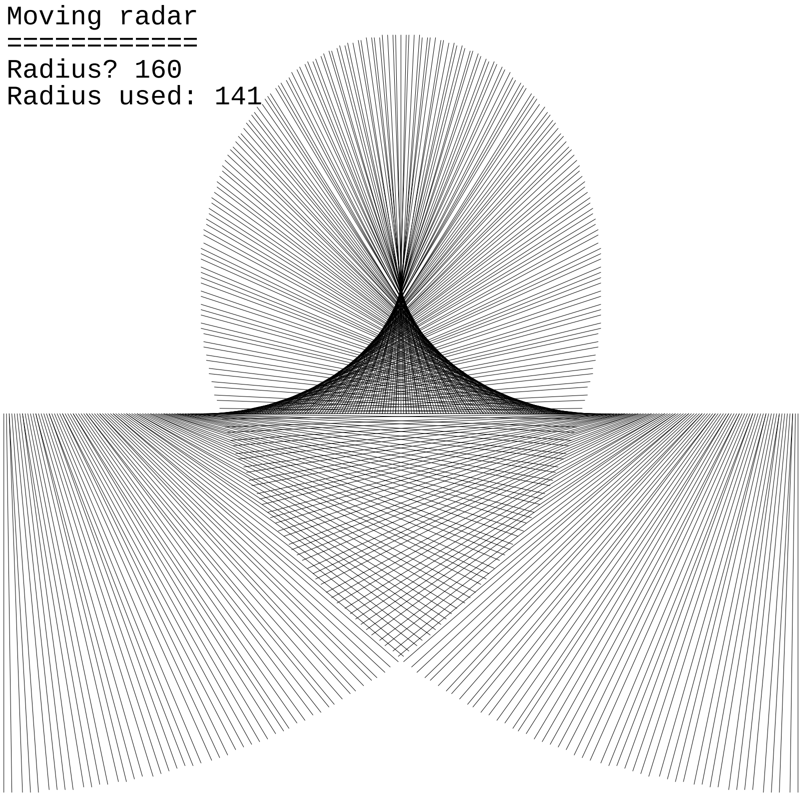


Figure 13.4: Moving radar

```
maak een beroemd schilderij:
  zet het doek gereed;
  bekladder het vlijtig;
  het schilderij is af.
```

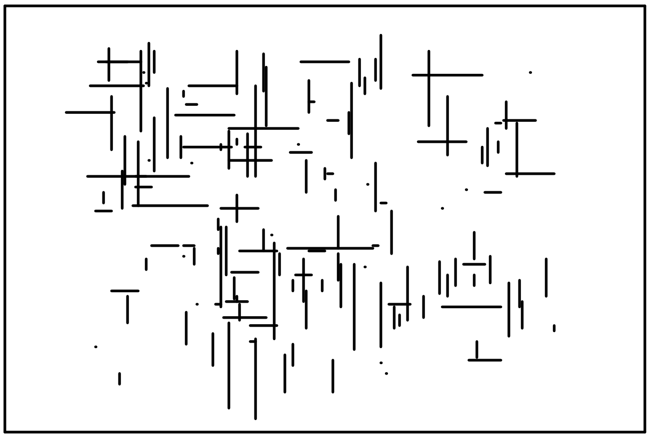


Figure 13.5: Een Mondriaan?

```
zet het doek gereed:
  enter graphics mode;
  INT xmin :: graphics x limit DIV 10,
      ymin :: graphics y limit DIV 10,
      xmax :: graphics x limit - xmin,
      ymax :: graphics y limit - ymin;
  move (xmin, ymin);
  draw (xmin, ymax);
  draw (xmax, ymax);
  draw (xmax, ymin);
  draw (xmin, ymin).
```

We laten (door de keuze van de minima en maxima) 10% van de horizontale en de verticale ruimte op het "canvas" ongebruikt.

Nu gaan we in horizontale of vertikale richting streepjes zetten, waarvan de lengte een bepaalde (willekeurige) verdeling heeft, maximaal 10% van de canvasbreedte, met een zekere voorkeur voor lange streepjes.

```
bekladder het vlijtig:
  INT VAR k, l, x, y;
  FOR k FROM 1 UPTO 150
  REP
    x := random (2 * xmin, xmax - xmin);
    y := random (2 * ymin, ymax - ymin);
    l := int (real (ymin) *
               (1.0 - sqrt (random)));
    IF random > 0.5
    THEN
       move (x - l, y);
       draw (x + l, y)
    ELSE
       move (x, y - l);
       draw (x, y + l)
    FI
  ENDREP.

het schilderij is af:
  wait for confirmation (5, 5);
  leave graphics mode.
```

13.3 De schildpad grafiek

De standaard bibliotheek bevat een pakket dat een laag bouwt boven op de integer grafiek waarmee het grafisch programmeren sterk vereenvoudigd wordt. Dit pakket is geïnspireerd door Karel de Robot en de **schildpad grafiek** van de onderwijstaal LOGO.

Om te beginnen vervangen we de integer coördinaten door reële coördinaten. Het is veel gemakkelijker een eenmaal berekende reële positie af te ronden naar een integer dan de positie voortdurend als integer bij te houden terwijl er (vanwege scheve lijnrichtingen) allerlei afrondingsfouten kunnen optreden.

Vervolgens gebruiken we relatieve coördinaten in plaats van absolute. Onze tekeningen bestaan nu eenmaal meestal uit opeenvolgende getrokken lijnen, zichtbaar of onzichtbaar, met elkaar verbonden. Naast de huidige positie ("grafische cursor") introduceren we nu ook een *huidige hoek*. Een lijnstuk begint altijd in de huidige positie en strekt zich uit in de richting aangegeven door de huidige hoek; alleen de lengte moet worden aangegeven (zie Fig. 13.6).

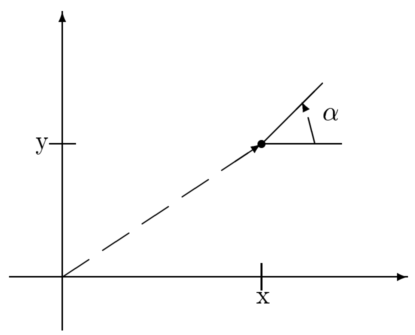


Figure 13.6: Huidige positie (x, y) en huidige hoek α

13.3.1 Het schildpad grafiek interface

We voeren twee lijntrek algoritmen in: `move` voor onzichtbare lijnen en `draw` voor zichtbare.

```
PROC move (REAL l):
   {teken een onzichtbare lijn van lengte
l}
   {in de richting van de huidige hoek}
   {te beginnen in de huidige positie}
ENDPROC move;

PROC draw (REAL l):
   {teken een zichtbare lijn van lengte l}
   {in de richting van de huidige hoek}
   {te beginnen in de huidige positie}
ENDPROC draw;
```

Na afloop is het eindpunt van de lijn tot huidige positie geworden.

Om de lijnen te kunnen sturen moeten we in staat zijn de waarde van de huidige hoek te veranderen.

```
PROC turn (INT angle):
   {tel 'angle' graden bij de huidige
hoek}
ENDPROC turn;
```

Volgens de gebruikelijke wiskundige conventie is de positieve hoek tegen de klok in. Voor het speciale geval van een bocht van 90 graden linksaf of rechtsaf introduceren we twee snellere procedures:

```
PROC turn left:
   {tel 90 graden bij de huidige hoek}
ENDPROC turn left;

PROC turn right:
   {trek 90 graden af van de huidige hoek}
ENDPROC turn right;
```

Naast de relatieve bewegingen (lijkend op die van Karel de Robot die achter zijn neus aangaat) introduceren wij nog twee lijn teken algoritmen in absolute coördinaten:

```
PROC move (REAL x, y):
   {ga naar positie (x, y)}
ENDPROC move;

PROC draw (REAL x, y):
   {teken een lijn van de huidige positie
naar (x, y)}
ENDPROC draw;
```

De plaats en grootte van het te gebruiken deel van het scherm, de schaal in horizontale en verticale richting, en daarmee ook de eenheid van afstand, kunnen we vrijelijk kiezen. Daartoe dient de procedure `turtle window`, die in verborgen variabelen van het pakket vastlegt hoe het real-gebied $[0.0 : x\ rng, 0.0 : y\ rng]$ moet worden afgebeeld op het integer grafiek gebied $[x\ min : x\ max, y\ min : y\ max]$.

Voor het omschakelen van en naar schildpad grafiek zijn er twee procedures: `enter turtle graphics` en `leave turtle graphics`. Na uitvoering van `enter turtle graphics` wordt het real-gebied [0.0 :

100.0, 0.0 : 100.0] afgebeeld op het gehele beeldscherm. Voor de meeste toepassingen zal dat toereikend zijn.

13.3.2 Voorbeeld: Tekenen van een rozet

We geven eerst een niet al te ambitieus voorbeeld van het gebruik van de schildpad grafiek: we tekenen een rozet.

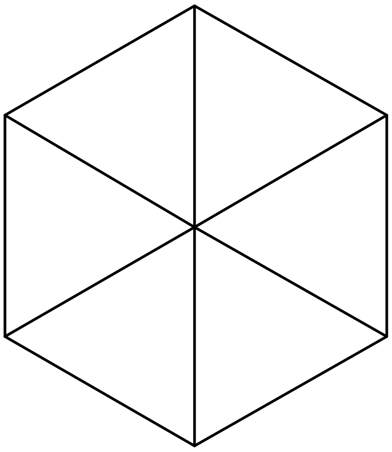


Figure 13.7: Rozet

Het rozet bestaat uit zes gelijkzijdige driehoeken, waarbij iedere driehoek ten opzichte van de vorige over 60 graden verdraaid is. De tekening begint in het midden van het scherm en de kanten van de driehoeken zijn gelijk aan eenderde van de schermlengte.

```
teken een rozet:
   enter turtle graphics;
   ga naar het midden;
   UPTO 6
   REP teken gelijkzijdige haak;
       turn (60)
   ENDREP;
   wait for confirmation (10, 10);
   leave turtle graphics.

ga naar het midden:
   REAL d:: turtle y limit / 3.0;
   move (turtle x limit / 2.0,
         turtle y limit / 2.0).

teken gelijkzijdige haak:
   draw (d); turn (120);
   draw (d); turn (120);
   move (d); turn (120).
```

In de rest van dit hoofdstuk vindt U een aantal ingewikkelder toepassingen van de schildpad grafiek.

13.4 Recursief tekenen

De volgende voorbeelden van teken algoritmen zijn door hun meervoudige recursie niet bepaald eenvoudig, maar zij tonen wel de kracht en elegantie van recursie als programmeermiddel aan.

13.4.1 Voorbeeld: Peano-krommen

De getallentheoreticus Peano heeft zich niet alleen beziggehouden met de axiomatische behandeling van de natuurlijke getallen, maar ook met het continuüm, d.w.z. met de reële getallen. Hij beschouwde o.a. continue afbeeldingen van het lijnstuk [0:1] op het vierkant [0:1, 0:1].

Daartoe definieerde hij families van krommen, opgebouwd uit rechte lijnstukken, met de eigenschap dat het n-de lid van de familie een continue kromme vormt, die aan alle punten van het vierkant op een afstand van ten hoogste 2^{-n} voorbijgaat. Een triviaal voorbeeld van zo een kromme is:

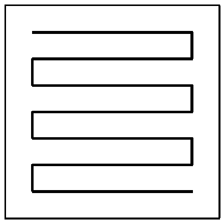


Figure 13.8: Eenvoudige Peano-kromme

Er bestaan echter veel fraaiere families van krommen. In deze sectie zullen we een algoritme van Aad van Wijngaarden presenteren, die zo'n familie (die overigens voor het eerst beschreven is door Hilbert) berekent en uittekent.

We definiëren nu een familie van Peano-approximanten op de volgende wijze.

Het nulde lid moet aan alle punten uit het eenheidsvierkant op een afstand van ten hoogste 2^{-0} voorbijgaan. Hieraan voldoet bijvoorbeeld een punt in het midden van het vierkant:

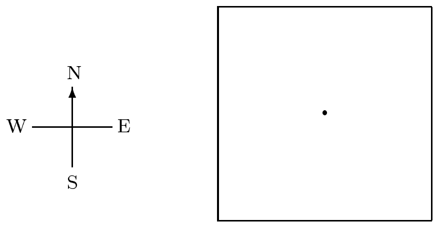


Figure 13.9: Peano-kromme van orde 0

Als volgende approximant (het eerste lid van de familie) delen we het vierkant in vier vierkanten met ribbe 1/2. In deze kleine vierkanten gebruiken we de vorige approximant en verbinden de kleine tekeningen door middel van drie lijnstukken van lengte 1/2, de eerste in oostelijke richting, de tweede naar het noorden en de derde naar het westen:

We noemen dit een approximant met oriëntering noord, omdat de netto beweging naar het noorden is.

We leggen de oriënteringen als volgt vast (voor $n = 1$):

Onze algoritme is gebaseerd op het idee, de n-de approximant te construeren uit vier verkleinde $(n-1)$ste approximanten, aan elkaar verbonden door middel van

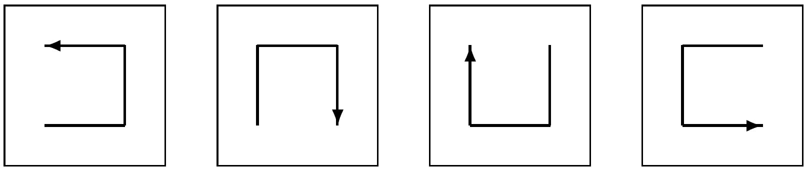


Figure 13.11: De vier orientaties

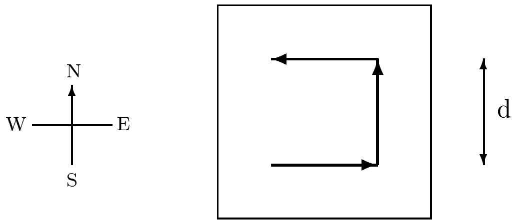


Figure 13.10: Peano-kromme van orde 1

drie lijnstukken van de lengte 2^{-n}, dit alles met slim gekozen oriënteringen. Tekenen we als voorbeeld een approximant van orde 2 met oriëntering noord:

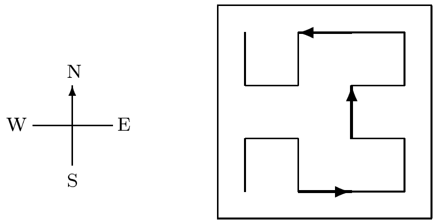


Figure 13.12: Peano-kromme van orde 2

Een Peano-approximant van orde n met oriëntering noord begint in het punt $(2^{-(n+1)}, 2^{-(n+1)})$ en eindigt, na een zwaai door het eenheidsvierkant, in het punt $(2^{-(n+1)}, 1-2^{-(n+1)})$. De bovenvermelde approximant van de orde $n = 2$ bestaat uit:

```
approximant oost (1);
lijnstuk oost;
approximant noord (1);
lijnstuk noord;
approximant noord (1);
lijnstuk west;
approximant west (1)
```

Het tekenen van een approximant van orde `n` kunnen we nu gieten in de vorm van vier onderling recursieve procedures:

```
PROC approximant noord (INT n):
     IF n <> 0
     THEN approximant oost  (n-1);
          lijnstuk oost;
          approximant noord (n-1);
          lijnstuk noord;
          approximant noord (n-1);
          lijnstuk west;
          approximant west  (n-1)
     FI
ENDPROC approximant noord;
```

Ingeval `n = 0` moet een punt getekend worden; daartoe hoeft de pen niet over het papier te worden bewogen.

```
PROC approximant oost (INT n):
     IF n <> 0
     THEN approximant noord (n-1);
          lijnstuk noord;
          approximant oost  (n-1);
          lijnstuk oost;
          approximant oost  (n-1);
          lijnstuk zuid;
          approximant zuid  (n-1)
     FI
ENDPROC approximant oost;

PROC approximant zuid (INT n):
     IF n <> 0
     THEN approximant west  (n-1);
          lijnstuk west;
          approximant zuid  (n-1);
          lijnstuk zuid;
          approximant zuid  (n-1);
          lijnstuk oost;
          approximant oost  (n-1)
     FI
ENDPROC approximant zuid;

PROC approximant west (INT n):
     IF n <> 0
     THEN approximant zuid  (n-1);
          lijnstuk zuid;
          approximant west  (n-1);
          lijnstuk west;
          approximant west  (n-1);
          lijnstuk noord;
          approximant noord (n-1)
     FI
ENDPROC approximant west;
```

Alle optredende lijnstukjes hebben eenzelfde lengte, $d = 2^{-n}$. De huidige positie (in [0:1, 0:1]) houden we voor ons zelf bij in een tweetal variabelen `REAL VAR x,y`. Om de lijnstukken van lengte `d` te tekenen introduceren we vier procedures:

```
PROC lijnstuk noord:
   y := y+d; draw (x,y)
ENDPROC lijnstuk noord;

PROC lijnstuk zuid:
   y := y-d; draw (x,y)
ENDPROC lijnstuk zuid;

PROC lijnstuk oost:
   x := x+d; draw (x,y)
ENDPROC lijnstuk oost;

PROC lijnstuk west:
   x := x-d; draw (x,y)
ENDPROC lijnstuk west;
```

Om het geheel in beweging te zetten moeten we beginnen in het punt $(d/2,\ d/2)$. Het hoofdprogramma kan luiden:

```
teken peanokromme:
  enter turtle graphics;
  vraag de orde;
  ga naar linker onderhoek;
  approximant noord (n);
  wait for confirmation (1, 20);
  leave turtle graphics.

vraag de orde:
  put ("Peano kromme van orde?");
  INT VAR n;
  get (n);
  REAL d :: turtle x limit / 2.0 ** n.

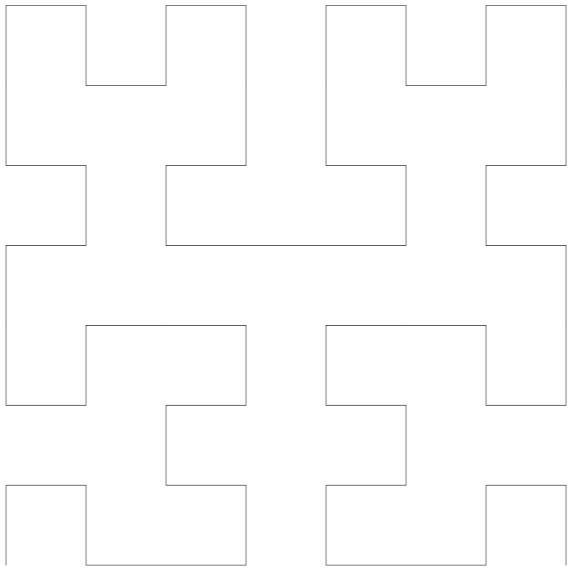
ga naar linker onderhoek:
  REAL VAR x :: d / 2.0, y :: d / 2.0;
  move (x, y).
```

Tot slot tonen we twee op deze wijze voortgebrachte plaatjes in fig. 13.13.

Vragen:

1. Hoe hangt de complexiteit, uitgedrukt in het totale aantal procedure-aanroepen of plot-aanroepen, van n af?
2. Opvallend aan de gegeven algoritme is dat de recursie eindigt op een niveau waar niets meer te doen is; dit is een typisch stijlkenmerk van Aad van Wijngaarden. Gebruikelijker is het, de recursie te beëindigen op het niveau waar de handeling voldoende triviaal is om hem in een slag te doen. Wat moet er aan de procedures gewijzigd worden om de recursie bij $n = 1$ te doen eindigen? Welk effect heeft deze wijziging op het aantal procedure-aanroepen?

Peano curve of order? 3



Peano curve of order? 6

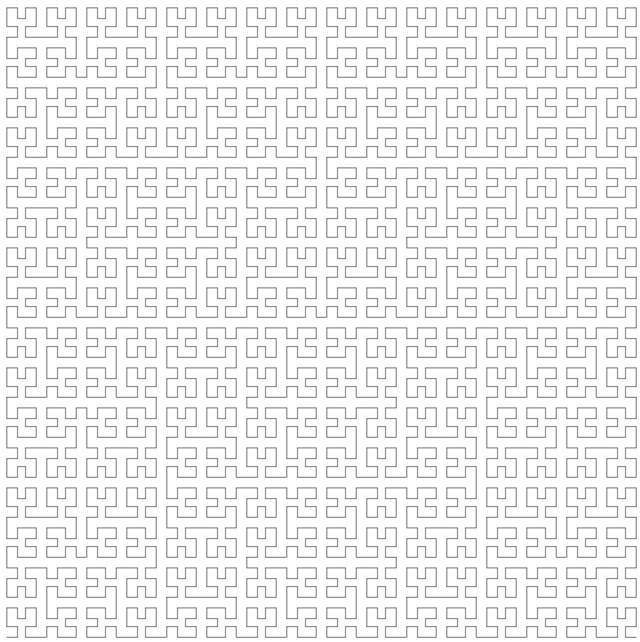


Figure 13.13: Peano-krommen van orde 3 en orde 6

13.5 Opgaven

1. (beverige lijn) Schrijf een algoritme analoog aan draw dat een beverig lijntje tekent, dat ongeveer van begin- naar eindpunt verloopt.

2. (sneeuwvlokken) Teken willekeurige sneeuwvlokken, met zes- en twaalftallige symmetrie. Maak gebruik van spiegeling en rotatie.
3. (kubus) Maak met behulp van schildpadgrafiek een perspectief-tekening van een kubus, waarbij de nabijheid wordt benadrukt door meerdere, iets divergente lijnen te tekenen.
4. (nogmaals Peano) Formuleer, onder gebruikmaking van de schildpadgrafiek, een algoritme voor het tekenen van Peano-krommen dat gebruik maakt van twee onderling recursieve procedures, voor het tekenen van resp. een linksdraaiende en een rechtsdraaiende Peanokromme.
5. (Pythagoras-bomen) Een Pythagoras-boom van orde n is een recursieve tekening zoals weergegeven in Fig. 13.14 voor de ordes 0,1 en 2. Voor $n > 0$ bevat zo'n boom op zijn beurt twee wat kleinere Pythagoras-bomen van orde $n - 1$, gemonteerd op de rechte zijden van een denkbeeldige rechthoekige driehoek met de schuine zijde als basis. Voor $n = 0$ wordt alleen de basis getekend. Construeer een algoritme voor het tekenen van Pythagorasboom, met als (geheeltallige) parameters: de dikte van de stam, de hoek α van zijn linker onderboom en de diepte van recursie, die de vorm van de boom bepalen.

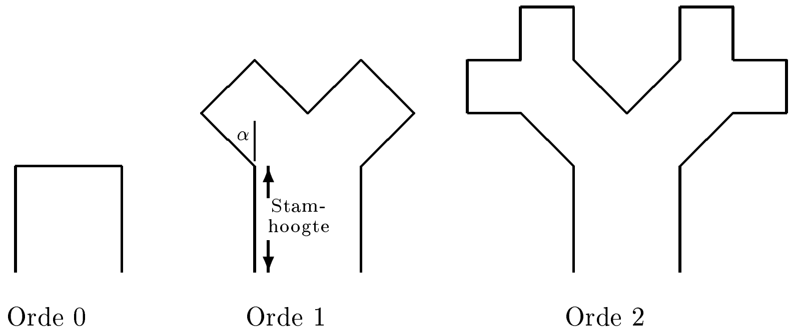


Figure 13.14: Eenvoudigste Pythagoras-bomen ($\alpha = 45^o$)

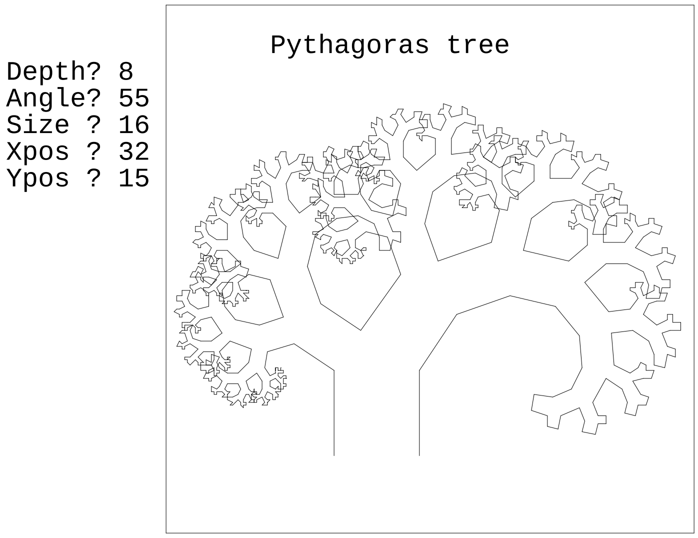


Figure 13.15: Pythagorasboom van orde 8

Chapter 14

Recursief sorteren

In dit hoofdstuk behandelen we, als voorbeeld van recursie, enige efficiënte sorteeralgoritmen. We sluiten aan bij het behandelde in hoofdstuk 8.

Sorteren kunnen we met allerlei verschillende algoritmen, die verschillen wat betreft hun complexiteit in tijd (het aantal uit te voeren elementaire algoritmen) en in ruimte (aantal elementaire variabelen dat als geheugen nodig is). Van alle rekentijd die ter wereld dagelijks wordt opgebruikt, wordt een groot gedeelte besteed aan het sorteren van allerlei bestanden volgens verschillende criteria. Het heeft dus grote economische betekenis te zoeken naar sorteeralgoritmen die zuinig omspringen met tijd en ruimte.

Zuinig met ruimte: de noodzakelijke geheugenruimte hangt tenminste lineair af van n, het aantal te sorteren elementen. We zullen alleen die sorteeralgoritmen bekijken die naast de elementen zelf slechts een klein aantal elementaire variabelen extra nodig hebben, liefst onafhankelijk van n. We beperken ons daarom tot **in situ sorteren**, waarbij het permutatieproces in de variabele f zelf gebeurt.

Zuinig met tijd: als we rechttoe, rechtaan een in situ sorteeralgoritme programmeren kunnen we komen tot formuleringen als gegeven in 8.7.2. De tijdscomplexiteit van deze in situ sorteeralgoritmen, en van allerlei voor de hand liggende varianten, is kwadratisch in het aantal te sorteren elementen. Dat betekent met name dat bij verdubbeling van het aantal elementen, de hoeveelheid werk vier maal zo groot wordt. De algoritme kan best heel dragelijk werken voor het sorteren van de gegevens van duizend employees; maar om de gegevens van 14 miljoen nederlanders te sorteren moet ongeveer 200 miljoen maal zoveel werk verzet worden. Het is wel duidelijk dat kwadratische methoden voor het sorteren van echt grote bestanden ongeschikt zijn.

14.1 Merge Sort

Een uitweg is, niet alle elementen tegelijk te sorteren, maar telkens de helft van de elementen apart te sorteren en de gesorteerde delen daarna aan elkaar te knopen. We pogen dus te komen tot een "verdeel en heers"-techniek, een halveringsmethode.

We kunnen in dit geval als volgt inzien dat de methode minder tijd vergt: als ik de te sorteren elementen in twee even grote groepen splits, kan ik elk van die groepen volgens een kwadratisch algoritme sorteren in een vierde van de tijd die ik voor de hele rij nodig zou hebben. Daarna moet ik de twee gesorteerde groepen "vermengen" door telkens de kleinste uit beide groepen naar beneden te halen. Als dit mengen nu maar snel genoeg gaat winnen we in totaal tijd.

We willen de globale rij `f` in situ sorteren. We declareren daartoe

```
PROC merge sort (INT lwb, upb):
  IF lwb < upb
  THEN
    INT CONST midden :: (lwb + upb) DIV 2;
    merge sort (lwb, midden);
    merge sort (midden + 1, upb);
    in situ mengproces
  FI.

  in situ mengproces:
    INT VAR beneden :: lwb;
    INT VAR boven :: midden + 1;
    REP
      IF de kleinste zit boven
      THEN neem kleinste apart;
           schuif benedengroep 1 op;
           zet kleinste beneden
      ELSE passeer benedenelement
      FI
    UNTIL benedengroep leeg
          OR bovengroep leeg
    ENDREP.

  de kleinste zit boven:
    f[boven] < f[beneden].

  neem kleinste apart:
    ELEMENT CONST kleinste :: f[boven].

  schuif benedengroep 1 op:
    FOR i FROM boven-1 DOWNTO beneden
    REP
      f[i+1] := f[i]
    ENDREP;
    beneden INCR 1; boven INCR 1.

  zet kleinste beneden:
    f[beneden-1] := kleinste.

  passeer benedenelement:
    beneden INCR 1.

  benedengroep leeg:
    boven = beneden.

  bovengroep leeg:
    boven > upb.
```

```
ENDPROC merge sort;
```

Het sorteren van de twee helften gaat nu wel sneller, maar het `in situ mergproces` bederft alles weer: daarbij kunnen grote deelrijen meermaal worden verschoven, waardoor de algoritme toch een kwadratisch worst-case gedrag heeft. Het is makkelijk in te zien dat bij gebruik van een tweede rij als hulprij het mengen wel in lineaire tijd kan gebeuren. Maar dan vergt de algoritme teveel hulpgeheugen om "in situ" genoemd te worden. Varianten van de Merge Sort algoritme, die niet in situ werken maar gebruik maken van grote achtergrondgeheugens, worden inderdaad veel toegepast. Wij zullen ze niet verder behandelen.

Vraag: hoe hangt de tijdscomplexiteit van deze versie van Merge Sort af van het aantal te sorteren elementen?

14.2 Quicksort

In Merge Sort worden twee helften apart gesorteerd en daarna vermengd. Kunnen we de zaak ook omdraaien: eerst de rij splitsen in twee delen en die twee delen daarna apart sorteren?

C.A.R. Hoare heeft in 1961 op basis van dit idee een recursieve sorteeralgoritme voorgesteld, die niet anders dan fraai te noemen is.

Wij gaan uit van een ongesorteerde rij van n elementen, waarvan de elementen alle mogelijke waarden kunnen hebben.

```
|                                      |
|//////////////////////////////////////|
|1////////////////////////////////////n|
----------------------------------------
|//////////////////////////////////////|
|//////////////////////////////////////|
|                                      |
```

Nu is de grondgedachte, de rij door verplaatsen van de elementen in twee delen $[1 : m]$ en $[m+1 : n]$ op te splitsen, zodanig dat alle elementen uit het eerste deel volgens de gegeven ordeningsrelatie vóór alle elementen uit het tweede deel komen, d.w.z.:

(1) $k \in [1 : m], j \in [m+1 : n] \rightarrow$
$f_k <= fj$

In een plaatje:

```
|              |                       |
|              |///////////////////////|
|1            m|//////////////////////n|
---------------|------------------------
|//////////////|m+1                    |
|//////////////|                       |
|              |                       |
```

Hiertoe moeten in het algemeen ettelijke elementen verwisseld worden, een hoeveelheid werk die lineair is in n.

Is de tabel eenmaal zo opgedeeld dan kan men de beide delen ieder apart in situ sorteren (met behulp van dezelfde algoritme Quicksort), zodat geldt:

(2) $k < j \in [1 : m] \rightarrow f_k <= fj$
(3) $k < j \in [m+1 : n] \rightarrow f_k <= fj$.

Uit (1), (2) en (3) volgt dan onmiddellijk:

(4) $k < j \in [1 : n] \rightarrow f_k <= fj$

d.w.z. zonder verder mengen is de hele rij in situ gesorteerd.

Het eigenlijke werk zit hem in het opsplitsen. Laten we dit proces nader bekijken.

14.2.1 Splitsen van de rij

Wat is de ideale splitsing van de rij? Dat is die, waarbij de rij in twee precies even lange delen verdeeld wordt. Dan hebben we het meeste profijt van de splitsing. We moeten dus een element x proberen te vinden, zodanig dat er evenveel elementen van de rij voor als na x komen. Zo een element heet de **mediaan**, omdat het dat element is dat bij sortering van de rij in het midden komt. Met deze vaststelling zijn we nog niet veel opgeschoten:

- elementen kunnen gelijk zijn, zodat de mediaan niet eenduidig gedefinieerd is,
- bij een even aantal elementen bestaat er geen middelste element, het midden ligt tussen twee elementen in,
- de voor de hand liggende algoritme om de mediaan te vinden, is: de tabel sorteren en dan het middelste element nemen.

We zullen tevreden moeten zijn met een schatting van de mediaan; is deze schatting redelijk, dan zullen we het verschil in werk niet merken; het gedrag bij een slechte schatting zullen we nog onderzoeken.

Als we verder niets over de elementen van de rij weten kunnen we als schatting van van de mediaan een willekeurig element van de rij nemen. Kiezen we bijvoorbeeld het middelste element als geschatte mediaan, dan zitten we goed als de rij toevallig al (bijna) gesorteerd is.

Wat is de slechtst mogelijke splitsing? Dat is die, waarbij een van beide helften leeg is en de andere helft weer gelijk aan de originele rij. Met deze splitsing schieten we niets op, want het sorteren van de grootste helft bestaat dan uit het sorteren van de originele rij dus als we niet oppassen raakt de algoritme in een niet-afbrekende herhaling. We zullen er daarom op toezien dat ieder van de helften echt kleiner is dan de oorspronkelijke rij.

Nu is het idee van de splitsing als volgt. Laat de rij niet leeg zijn. We kiezen een element in het midden als de geschatte mediaan. We laten een wijzer `p` van onderen naar boven door de rij lopen en een wijzer `q` van boven naar onderen. Telkens als we bij `p` een element vinden groter dan de geschatte mediaan en bij `q` een element kleiner dan de geschatte mediaan verwisselen we deze beide elementen. Met andere woorden: als we aan beide kanten een element vinden dat "verkeerd" staat verwisselen we die twee.

Preciezer: zij gegeven een globale rij van elementen

```
ROW n ELEMENT VAR f
```

waarbij `ELEMENT` een of ander type is, welk doet niet ter zake. Zij $1 <= i < j <= n$. We kiezen m ongeveer in het midden van $[i : j]$. Zij $x = f_m$ (de geschatte mediaan). We willen $f_i, ..., f_j$ zo in situ ordenen en een p en q zo kiezen uit $[i-1 : j+1]$ dat voor alle k in $[i : j]$ geldt:

$$k < p \to f_k \le f_m \text{ en}$$
$$k = p \to f_k > f_m \text{ en}$$
$$k > q \to f_k \ge f_m \text{ en}$$
$$k = q \to f_k < f_m \text{ en}$$
$$q \le p.$$

Bij de omordening krijgt de variabele `p` dus als waarde de index van het eerste element groter dan f_m, en `q` wijst op dezelfde wijze naar het laatste element kleiner dan f_m; is het linkerdeel leeg dan krijgt `q` de waarde `i-1`; is het rechterdeel leeg dan krijgt `p` de waarde `j+1`. In een plaatje:

```
      ≤ m      p
i-------------------|--------------j
               q      ≥ m
i---------------|------------------j
                 = m
i---------------|---|--------------j
               q     p
```

We kunnen de rij nu in 3 gedeelten splitsen, waarbij het middelste deel tenminste uit de geschatte mediaan bestaat, dus niet leeg is. Als het mediaan-element niet vanzelf tussen `p` en `q` zit, moeten we het daartoe naar het midden halen. Het middendeel kan ook andere elementen dan de mediaan bevatten, die daaraan gelijk zijn, maar hoeft niet alle elementen gelijk aan de mediaan te bevatten (ga dat na). Het middendeel hoeven we niet meer te sorteren. Het linker- en rechterdeel kunnen leeg zijn. Omdat echter beide delen kleiner zijn dan de oorspronkelijke rij is een niet-afbrekende herhaling uitgesloten.

```
PROC splits (INT i, j, INT VAR p, q):
  initialiseer;
  WHILE p en q geschikte kandidaten
  REP
    verwissel p en q element
  ENDREP;
  haal zonodig de mediaan naar t midden.

  initialiseer:
    p := i; q := j;
    INT CONST m :: (i+j) DIV 2;
    ELEMENT CONST med :: f[m].

  p en q geschikte kandidaten:
    schuif p zover mogelijk omhoog;
    schuif q zover mogelijk omlaag;
    p < q.

  schuif p zover mogelijk omhoog:
    WHILE kan p nog verhogen
    REP
      p INCR 1
    ENDREP.

  kan p nog verhogen:
    IF p > j THEN FALSE ELSE f[p] <= med
FI.

  schuif q zover mogelijk omlaag:
    WHILE kan q nog verlagen
    REP
      q DECR 1
    ENDREP.

  kan q nog verlagen:
    IF q < i THEN FALSE ELSE f[q] >= med
FI.

  verwissel p en q element:
    ELEMENT CONST wissel :: f[p];
    f[p] := f[q]; f[q] := wissel;
    p INCR 1; q DECR 1.

  haal zonodig de mediaan naar t midden:
    IF m > p
    THEN verwissel m en p element
    ELIF m < q
    THEN verwissel m en q element
    FI.

  verwissel m en p element:
    f[m] := f[p];
    f[p] := med;
    p INCR 1.

  verwissel m en q element:
    f[m] := f[q];
    f[q] := med;
    q DECR 1.

ENDPROC splits;
```

Het splitsproces is lineair in `n`, want `p` en `q` wandelen samen over hoogstens `2n` elementen. Het middelste deel is gegarandeerd niet leeg, het kan zelfs vele elementen bevatten, bijvoorbeeld bij het sorteren van een rij bestaande uit allemaal gelijke elementen.

Merk op dat de aangegeven volgorde in de romp van de procedure `splits` een overspecificatie is: in werkelijkheid zijn we er niet in geïnteresseerd dat *eerst* `p` omhoog en *dan pas* `q` omlaag verplaatst wordt. Eigenlijk moesten deze twee acties collateraal of zelfs (samenwerkend) parallel verricht worden.

14.2.2 Het sorteren met behulp van splits

Een eenmaal gesplitste rij sorteren we uiteraard weer met dezelfde sorteerprocedure, dus recursief:

```
PROC quicksort (INT lwb, upb):
  IF de rij bevat meer dan 1 element
  THEN
    INT VAR p, q;
    splits (lwb, upb, p, q);
    quicksort (lwb, q);
    quicksort (p, upb)
  FI.
  de rij bevat meer dan 1 element:
    lwb < upb.
ENDPROC quicksort;
```

Is in deze procedure sprake van een overspecificatie van de volgorde? Waarom termineert deze procedure gegarandeerd?

De splitsing zit tegen als bij het splitsen juist het hoogste of laagste element in het midden staat, want dan wordt de meest ongunstige splitsing verkregen. Ga dat na. Welke complexiteit heeft deze algoritme in het slechtste geval? En in het beste geval?

De "gemiddelde" complexiteit hangt sterk af van de verdeling van de waarden van de elementen. In bijvoorbeeld [KNU73] is na te lezen, dat de tijdscomplexiteit van de algoritme, onder aanname dat deze elementen willekeurig uit een homogene verdeling getrokken zijn, O(n log n) bedraagt, net als in het beste geval, maar met een andere vermenigvuldigingsfactor. Voor voldoende grote n is dit willekeurig veel beter dan O(n2).

Tot slot een voorbeeld van een sorteerproces met Quicksort. De nog te sorteren elementen worden onderstreept, en de geschatte mediaan bij de splitsing tussen verticale streepjes aangegeven.

```
 3   1   4   2   6  |9|  5   7   5   8   0
-------------------------------------------
 3   1   4   2  |6|  0   5   7   5   8   9
---------------------------------------
 3   1   4  |2|  5   0   5   6  |7|  8   9
----------------------------    -------
|0|  1   2   4  |5|  3   5   6   7   8   9
-------     ----------------
 0   1   2  |4|  3   5   5   6   7   8   9
            -------
 0   1   2   3   4   5   5   6   7   8   9

 0   1   2   3   4   5   5   6   7   8   9
```

14.2.3 Splitsen van de rij, alternatieve methode

In [DIJ76] wordt een probleem gesteld en opgelost dat bekend staat als het "Dutch National Flag problem":

> Beschouw een rij van elementen die ieder een van de kleuren rood, wit of blauw hebben. Orden ze in situ tot ze een nederlandse vlag vormen.

Deze oplossing kunnen wij ook gebruiken voor het splitsen van de rij in Quicksort. Dit leidt tot een elegante algoritme, die evenwel iets meer verplaatsingen vereist dan de zojuist gepresenteerde, oorspronkelijke algoritme van Hoare.

We nemen weer een willekeurig element `med` als geschatte mediaan en denken ons de elementen f_k van de rij alsvolgt gekleurd:

$$f_k < med(\text{rood})$$

$$f_k = med(\text{wit})$$

$$f_k > med(\text{blauw}).$$

Nu willen we de oorspronkelijk ongesorteerde rij zodanig permuteren dat we uiteindelijk drie gescheiden gebieden krijgen, van links naar rechts: alle rode elementen (elementen kleiner dan de mediaan), alle witte elementen (elementen gelijk aan de mediaan) en alle blauwe elementen (elementen groter dan de mediaan).

Zolang we hiermee nog bezig zijn, hebben we vier gebieden: een met rode elementen, een met witte, een met blauwe en een met nog te onderzoeken elementen. De gekleurde gebieden houden we in de bedoelde volgorde (dat ligt nogal voor de hand), maar waar plaatsen we het *onderzoekgebied*? Deze keus is geheel vrij: bij het begin is de hele rij onderzoekgebied, aan het eind is dat gebied leeg, dus doet de plaats er niet toe. Toch is de keuze belangrijk. We geven niet de overwegingen, maar doen een bepaalde keus. Ga later na, dat andere keuzes in elk geval niet tot betere algoritmen leiden.

We kiezen het onderzoekgebied tussen rood en wit en markeren de gebieden met de indices r, w en b, als volgt:

```
-------|----------------|-----|--------
i       r                w     b       j
```

dus:

$$i <= k < r \rightarrow f_k < med \text{ (rood)}$$

$$r <= k <= w \rightarrow \text{kleur onbekend (onderzoekgebied)}$$

$$w < k <= b \rightarrow f_k = med \text{ (wit)}$$

$$b < k <= j \rightarrow f_k > med \text{ (blauw).}$$

We bekijken nu steeds f_w (ga later na, dat het bekijken van f_r minder handig is). Is f_w wit, dan schuiven we w eenvoudig 1 naar links. Is f_w rood, dan verwisselen we hem met f_r en schuiven r een naar rechts. Is f_w blauw, dan verwisselen we hem met f_b en schuiven b een naar links.

De correctheid van de algoritme is bijzonder eenvoudig in te zien. De rode, witte en blauwe gebieden blijven rood, wit en blauw. Bij elke stap wordt het onderzoekgebied noodzakelijk 1 element korter, dus termineert de algoritme zeker. Als we als `med` een element uit de rij hebben gekozen zal het witte gebied minstens 1 lang zijn, waardoor de rode en blauwe gebieden na afloop zeker korter zijn dan de hele rij, en dus ook de Quicksort algoritme zeker termineert.

Initieel zullen de rode en blauwe gebieden leeg zijn. Kiezen we f_j als med dan zal het witte gebied initieel 1 lang zijn. (Ga na, dat we het witte gebied ook initieel leeg kunnen kiezen.)

Hiermee wordt de procedure `splits`:

```
PROC splits (INT i, j, INT VAR p, q):
  initialiseer;
  WHILE onderzoekgebied niet leeg
  REP
    inspecteer element op w;
    IF element is wit
    THEN schuif w 1 naar links
    ELIF element is rood
    THEN
      verwissel met element op r;
      schuif r 1 naar rechts
    ELSE
      verwissel met element op b;
      schuif w en b 1 naar links
    FI
  ENDREP;
  q := r - 1;
  p := b + 1.

  initialiseer:
      ELEMENT CONST med :: f[j];
      INT VAR r :: i, w :: j-1, b :: j.

  onderzoekgebied niet leeg:
      r <= w.

  inspecteer element op w:
      ELEMENT CONST deze :: f[w].

  element is wit:
    deze = med.

  schuif w 1 naar links:
    w DECR 1.

  element is rood:
    deze < med.

  verwissel met element op r:
    f[w] := f[r];
    f[r] := deze.

  schuif r 1 naar rechts:
    r INCR 1.

  verwissel met element op b:
    f[w] := f[b];
    f[b] := deze.

  schuif w en b 1 naar links:
    w DECR 1;
    b DECR 1.

ENDPROC splits;
```

Als f_r rood is, wordt dit element eerst naar f_w gesleept (als ook `deze` rood is); in de volgende iteratie wordt het dan weer teruggebracht naar het rode gebied. Dat is een zwak punt van de algoritme (een offer aan de eenvoud). We kunnen de algoritme zo wijzigen, dat steeds eerst de index r naar rechts schuift zolang daar rode elementen staan. De algoritme wordt daardoor minder doorzichtig, maar komt meer in de buurt van de splitsing volgens Hoare. Ga deze modificatie na.

14.3 Opgaven

1. probeer een iteratieve oplossing te vinden voor het sorteren in tijd $O(nlogn)$. Hint: probeer merge sort met een hulprij, of zoek bij Knuth.
2. ontwerp een splitsingsalgoritme waarbij niet een element van de rij maar het gemiddelde van 3 elementen als mediaan wordt gekozen [EMB70].

Chapter 15

Backtrack-programmeren

De muis in de doolhof leverde een goed voorbeeld van een algoritme waarvan de werking berust op het systematisch eerst zetten en later eventueel weer ongedaan maken van stappen in de richting van de oplossing. De muis keert als het ware op zijn schreden terug. Er zijn belangrijke klassen van problemen die volgens een dergelijke *backtrack-strategie* kunnen worden opgelost.

15.1 Het aftel-probleem

Het probleem van de muis in de doolhof kan als volgt geformuleerd worden: vind alle manieren om op de vrije vakjes een lusvrije ketting van muizen te leggen van de uitgangspositie tot de kaas. Voor ieder vakje van de doolhof dat niet met `muur` of `kaas` gevuld is moeten we als het ware beslissen of we het met een muis beleggen dan wel vrij laten. We kunnen daartoe de hoogstens $n*m$ vrije vakjes van de doolhof opvatten als variabelen die ieder een van de waarden `vrij` en `muis` dienen aan te nemen.

Wij noemen de verzameling van waarden die de i-de variabele (het i-de vrije vakje) aan kan nemen de **selectieruimte** voor die variabele. Daarmee wordt dit probleem opgevat als een voorbeeld van het **aftel probleem**:

> Vind alle reeksen $(x_1, x_2, \ldots, x_n)$ met x_i uit een eindige verzameling $X_i, 1 \leq i \leq n$, die voldoen aan een predicaat $P(x_1, x_2, \ldots, x_n)$ [KNUTH 1975].

Laat k het aantal lege vakjes zijn in de doolhof ($1 \leq k \leq n*m$). De verlangde eigenschap $P(x_1, x_2, \ldots, x_k)$ is nu in ons voorbeeld de eigenschap, dat in de vakjes $1, 2, \ldots, k$ een lusvrije ketting van muizen ligt die zich van de uitgangspositie uitstrekt tot aan de kaas, terwijl de resterende vakjes vrij zijn.

Alternatief kunnen we ook als i-de variabele de positie van de i-de muis beschouwen, $X_i = (s, t)$ met $1 \leq s \leq n, 1 \leq t \leq m$ wat leidt tot een ander aftelprobleem met (bij juiste formulering) hetzelfde resultaat, d.w.z. dezelfde verzameling van lusvrije wegen. De tweede formulering is misschien wat meer voor de hand liggend en in ieder geval meer in de geest van het reeds gegeven programma. Wij vervolgen echter als voorbeeld de eerste formulering om een wat frissere kijk op het probleem te krijgen.

Wij zullen nu enige algemene methoden voor het oplossen van aftelproblemen onder de loupe nemen en daarbij een natuurlijke toepassing van recursie ontdekken.

15.1.1 De brute force methode

Het aftelprobleem is een eindig combinatorisch probleem en dus in principe recht door zee programmeerbaar: we hoeven "slechts" alle combinaties van variabelen-waarden systematisch te doorlopen.

```
WHILE niet alle combinaties geprobeerd
REP
  genereer volgende combinatie;
  test of deze aan P voldoet
ENDREP
```

Het is niet meteen evident hoe telkens de volgende combinatie gegenereerd kan worden, maar er zijn verschillende schema's denkbaar om systematisch alle combinaties te doorlopen.

In ons geval zijn er volgens de eerste formulering maximaal 2^k, volgens de tweede zelfs maximaal k^k combinaties (in het algemeen $l_1 * l_2 * \ldots * l_k$, waarin l_i het aantal elementen van selectieruimte X_i is). Dat is wel wat te veel van het goede. Klaarblijkelijk heeft het hier geen zin, systematisch de totale selectieruimte $X_1 * X_2 * \ldots * X_k$ te doorlopen en moeten we het wat slimmer aanpakken.

15.1.2 De selectieboom

Zonder verlies van algemeenheid kunnen we de verzameling van mogelijke deelrijen uit de selectieruimte weergeven als een boom van vertakkende mogelijkheden, de **selectie boom** (zie fig. 15.1.2). Voor de eerste variabele x_1 zijn er l_1 verschillende mogelijkheden. Voor ieder daarvan zijn er voor de tweede variabele x_2 weer l_2 mogelijkheden, enzovoort.

In ons geval zijn er k variabelen en is de selectieboom dus k diep. Schematisch kunnen we de boom ook aangeven als:

Volgens de **brute force** methode kunnen we de gehele selectieboom systematisch doorlopen, bijvoorbeeld (in bastaard Elan):

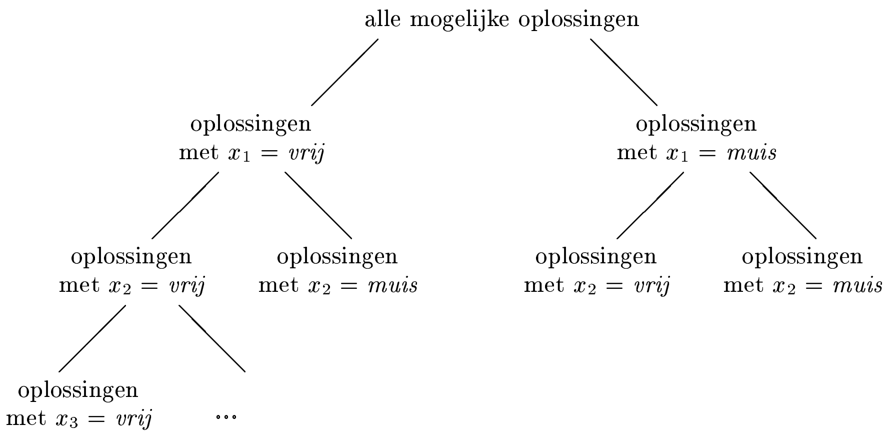


Figure 15.1: De selectieboom

Figure 15.2: Selectieboom met niveaus 1 ... k

Figure 15.3: Selectieboom, deels verworpen

```
FOR v1 ∈ (vrij, muis)
REP FOR v2 ∈ (vrij, muis)
  REP
    ...
      FOR vk ∈ (vrij, muis)
      REP IF P (v1, v2, ..., vk)
          THEN meld een oplossing
          FI
      ENDREP
    ...
  ENDREP
ENDREP
```

maar zoals wij reeds vaststelden heeft dit geen zin: de selectieboom is veel te groot.

Bij nadere beschouwing bedenken we echter dat we bij het doorlopen van de boom vaak vroegtijdig (d.w.z.: voor we k diep zijn) kunnen beslissen dat een bepaalde tak geen oplossing kan opleveren, bijvoorbeeld omdat er een lus in de weg zit of omdat de muizen een doodlopend gangetje inslaan. In dat geval kunnen we deze tak in zijn geheel verwerpen. Op die manier hoeven we onder omstandigheden slechts een fractie van de boom te doorlopen:

Het is nu de kunst de mogelijke combinaties te genereren in een zodanige volgorde, dat onproductieve takken van de selectieboom zo vroeg mogelijk gevonden en weggelaten worden.

15.1.3 Heuristieken

Bij het bepalen van de volgorde waarin de selectieboom doorlopen wordt hebben we verschillende vrijheidsgraden:

- De *keuze van de volgorde* der variabelen. Het verdient aanbeveling de variabelen in een zodanige volgorde te doorlopen dat onproductieve takken zo vroeg mogelijk gevonden worden en zo groot mogelijke stukken van de selectieruimte verworpen kunnen worden. Bij de muis in de doolhof moeten we de vakjes bijvoorbeeld niet in willekeurige volgorde of in een vaste volgorde als “van links naar rechts, van boven naar beneden” proberen, maar steeds aan het einde van het pad van een muis verderbreien, zodat eventuele lussen en doodlopende paden vroeg gevonden worden.
- *Inperking van de selectieruimte* per variabele. Vaak volgt uit de waarden van de tot op dit punt vastgelegde variabelen een beperking op de selectieruimte van de volgende.
- Het gebruik van min of meer sterke *verwerpingscriteria* voor delen van de boom. In ons voorbeeld: lussen in de weg, doodlopende wegen. Het zou bijvoorbeeld aanbeveling verdienen van de doolhof eerst alle doodlopende wegen met een muur af te sluiten.

- De exploitatie van *symmetrie-overwegingen* om meer oplossingen in een keer te verkrijgen. Hier niet zo evident, maar in sommige voorbeelden eenvoudig en vaak noodzakelijk om de complexiteit van de algoritme in de greep te houden.

Op welke wijze we deze en andere vrijheidsgraden bij de oplossing van een gegeven probleem kunnen benutten hangt af van de eigenschappen van ons probleem, en van onze eigen fantasie: we moeten er iets op vinden.

Ter nagedachtenis van Archimedes, de auteur van het gezegde "Eureka" en eerste bekende toepasser van heuristische methoden, gebruikt men het woord **heuristiek** voor een strategie, methode, vuistregel of truc, die ertoe dient om de complexiteit van een zoekproces te verminderen, zonder daarbij op de juistheid van de gevonden oplossingen te beknibbelen [SLAGLE 1971]. (In sommige wetenschapsgebieden bezigt men dit woord ook wel voor een methode om de complexiteit van een zoekproces te verminderen die "voldoende goede" oplossingen geeft; in een dergelijk geval zouden wij spreken van een **benaderende heuristiek** in plaats van de hier bedoelde **exacte heuristiek**.)

15.1.4 De backtrack-methode

Het gaat erom, alle reeksen $(x_1, x_2, \ldots, x_n)$ met $x_i \in X_i, 1 \leq i \leq n$ aan te geven die voldoen aan een eigenschap $P_n(x_1, x_2, \ldots, x_n)$.

Daartoe proberen we tussenliggende eigenschappen $P_k(x_1, x_2, \ldots, x_k)$ te vinden, zodanig dat voor $0 \leq k < n$ geldt $P_{k+1}(x_1, x_2, \ldots, x_k, x_{k+1})$ impliceert $P_k(x_1, x_2, \ldots, x_k)$.

Met andere woorden, als P_k *niet* geldt voor $(x_1, x_2, \ldots, x_k)$ dan kan P_{k+1} voor geen enkele keuze van x_{k+1} gelden. Inductief verder redenerend: de reeks $(x_1, x_2, \ldots, x_k)$ kan dus niet tot een reeks $(x_1, x_2, \ldots, x_k, \ldots, x_n)$ voortgezet worden die aan P_n voldoet. We nemen aan dat bij iedere P_k een algoritme hoort om vast te stellen of een rij daaraan voldoet.

In abstracto kunnen we dan zoeken volgens de volgende algoritme:

```
ROW n EL VAR x; INT VAR k :: 0;
PROC genereer alle voortzettingen:
   {P_k(x_1,...,x_k) ∧ 0 ≤k≤n}
   IF k = n
   THEN
     {P_n(x_1,...,x_n)}
     meld succes
   ELSE
     {P_k(x_1,...,x_k) ∧ 0 ≤k<n}
     EL VAR y;
     FOR y ∈ X_{k+1}
     REP
       IF P_{k+1}(x_1,...,x_k,y)
       THEN
         x[k+1] := y;
         k := k+1;
         {P_k(x_1,...,x_k) ∧ 0 <k≤n}
         genereer alle voortzettingen;
         k := k-1
       FI
     ENDREP
   FI
 ENDPROC genereer alle voortzettingen;
```

De beweringen tussen accoladen zijn **invarianten**, beweringen waarvan we weten dat ze op dat punt in het programma gelden. Zij dienen hier als verklaring maar vormen geen onderdeel van het programma en kunnen dus uitsluitend als commentaar worden opgevat.

Merk op dat steeds na afhandeling van een tak de algoritme naar een vorig punt in de selectieboom terugkeert; daaraan ontleent hij via het engels zijn naam **backtrack algorithme**. Een mogelijke nederlandse naam is **terugkrabbel algoritme**: de algoritme krabbelt telkens na een recursieve aanroep terug van een reeds (door `x[k+1]:=y; k:=k+1`) gemaakte keuze (door middel van `k:=k-1`). In het geval van de doolhof bestond het maken van een keuze uit het bezetten van een veld door een muis, wat (na een recursieve aanroep) expliciet ongedaan werd gemaakt.

De backtrack-algoritme doorloopt de selectieboom in **depth-first volgorde**: van links naar rechts en steeds zodanig in de diepte gaand dat iedere tak volledig afgelopen wordt voor zijn rechterbuurman wordt doorlopen, waarbij als niet-productief herkende takken overgeslagen worden:

Figure 15.4: Depth-first volgorde

15.1.5 Over de keuze van de P_k's

De predicaten P_k vormen de belichaming van de eerder genoemde heuristieken. Er is kennis van het probleem en enige inventiviteit voor nodig om ze te kiezen. Bovendien moeten ze nog eenvoudig als algoritme te formuleren en uit te voeren zijn. Hoe kiezen we ze?

Kiezen we P_k `TRUE` voor $k < n$ dan krijgen we weer de Brute Force methode; blijkbaar zijn de P_k's wat te licht. In het ideale geval zou moeten gelden

> $P_k(x_1, \ldots, x_k)$ dan en slechts dan als er $x_{k+1}, \ldots, x_n$ bestaan zodat $P_n(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$

d.w.z. P_k stelt precies vast of er nog een voortzetting bestaat. Maar zo'n algoritme is lastig te vinden en waarschijnlijk bijna net zo tijdrovend in de uitvoering als de oorspronkelijke probleemstelling.

In het algemeen zullen we een compromis moeten sluiten tussen restrictieve maar tijdrovende en zwakke maar gemakkelijk te berekenen P_k's. Kiezen we ze te zwak, dan vinden we wel de juiste oplossingen maar doorlopen we te veel van de selectieboom. Kiezen we ze te sterk, dan leveren ze geen efficiëntie-voordeel meer op.

15.2 Voorbeeld: 8 koninginnen op een schaakbord

Een klassiek voorbeeld van een backtrack-algoritme is de oplossing van het volgende probleem. Verdeel 8 koninginnen over een schaakbord op zodanige wijze dat geen enkele koningin een andere kan slaan (horizontaal, verticaal, rechts-diagonaal of links-diagonaal).

- *Keuze van de variabelen.*
  Als variabelen kiezen we de posities van de koninginnen, dus 8 coördinaten $\in [1:8, 1:8]$.
- *Inperking van de selectieruimte.*
  In iedere rij kan maar 1 koningin staan, want als er twee staan kunnen ze elkaar slaan. In iedere rij moet ook precies 1 koningin staan, want anders brengen we er nooit 8 op een schaakbord onder.
- *Volgorde van de variabelen.*
  We nummeren de variabelen (arbitrair) van 1 tot 8 naar de rij waarin ze voorkomen.

```
PROC plaats koninginnen vanaf
          (INT deze rij):
  IF deze rij > 8
  THEN meld succes
  ELSE
    INT VAR i;
    FOR i UPTO 8
    REP
      IF plaats i is onbedreigd
      THEN
        pak plaats i;
        plaats koninginnen vanaf
             (deze rij + 1);
        geef plaats i vrij
      FI
    ENDREP
  FI.
```

Wij houden de plaats van de i-de koningin bij in een globale rij

```
ROW 8 INT VAR kolom koningin;
```

We verfijnen

```
pak plaats i:
  kolom koningin [deze rij] := i.

geef plaats i vrij:  .
```

Om de plaats vrij te geven hoeven wij niet expliciet iets te doen — de terugkeer uit de procedure is voldoende.

```
  plaats i is onbedreigd:
    INT VAR j;
    FOR j UPTO deze rij -1
    REP
      IF jde koningin bedreigt i
      THEN LEAVE plaats i is onbedreigd
             WITH false
      FI
    ENDREP;
    true.

  jde koningin bedreigt i:
    zelfde kolom
    OR rechtsboven bezet
    OR linksboven bezet.

  zelfde kolom:
    kolom koningin [j] - i = 0.

  rechtsboven bezet:
    kolom koningin [j] - i = deze rij - j.

  linksboven bezet:
    kolom koningin [j] - i = j - deze rij.

  meld succes:
    put ("Oplossing:");
    line;
    print t schaakbord uit.
ENDPROC plaats koninginnen vanaf;

PROC print t schaakbord uit:
  INT VAR k;
  FOR k UPTO 8
  REP print regel uit;
      line
  ENDREP.

  print regel uit:
    INT VAR l;
    FOR l UPTO 8
    REP print positie
    ENDREP.

  print positie:
    IF l = kolom koningin [k]
    THEN put (" Q ")
    ELSE put (" . ")
    FI.

ENDPROC print t schaakbord uit;
```

Zonder de rij `kolom koningin` te moeten initialiseren kunnen we als hoofdprogramma volstaan met:

```
ROW 8 INT VAR kolom koningin;
plaats koninginnen vanaf (1)
```

Dit programma levert zonder al te veel werk onzerzijds alle 92 oplossingen voor het 8 koninginnen probleem. Door een slim gebruik van symmetrie-overwegingen te maken zou de hoeveelheid rekenwerk nog te reduceren zijn, want slechts 12 hiervan zijn wezenlijk verschillend.

15.3 Het optimalisatie-probleem

Een generalisatie van het aftelprobleem is het discrete **optimalisatie probleem** [GOB65].

> Vind die rij $(x_1, x_2, \ldots, x_n)$ met x_i uit een eindige verzameling X_i, $1 \leq i \leq n$, waarvoor een bepaalde **criterium functie** $F_n(x_1, x_2, \ldots, x_n)$ een minimale (resp. maximale) waarde aanneemt. Bepaal die waarde van F_n.

In geval er meer dan één oplossing is kan één daarvan uitgekozen worden, dan wel een opsomming van alle oplossingen verlangd worden, afhankelijk van de probleemstelling.

Dit probleem is klaarblijkelijk op te lossen door aftellen waarbij steeds de beste van de gevonden rijen $(x_1, \ldots, x_n)$ onthouden wordt. Ook kan een aftelprobleem omgekeerd gezien worden als een discreet optimalisatie-probleem met een tweewaardige functie (bruikbaar/onbruikbaar) waarbij gevraagd wordt alle bruikbare oplossingen te geven. Merk op dat we ons beperken tot eindige discrete selectieruimten X_i. Is X_i bijvoorbeeld een segment uit de reële getallen en F_n daarop partieel differentieerbaar dan zijn heel andere methoden van toepassing. Er zijn legio voorbeelden van discrete optimalisatie-problemen te vinden, onder andere in de Operations Research en de Cryptografie.

Beschouw nu zo'n discreet maximalisatieprobleem. We leiden uit de probleemstelling een backtrack-algoritme af door de constructie van partiële criteriumfuncties

$$F_k(x_1, x_2, \ldots, x_k),\ 0 \leq k < n$$

die voldoen aan

$$F_k(x_1, \ldots, x_k) \leq F_{k+1}(x_1, \ldots, x_k, x_{k+1}),\ 0 \leq k < n$$

De partiële criteriumfunctie F_k geeft dus een ondergrens voor de minimale waarde die de criteriumfunctie F_n kan aannemen voor enigerlei argument beginnend met $(x_1, \ldots, x_k)$.

Anders gezegd: F_k is een pessimistische schatter voor F_n; hij kan wel te laag schatten, maar nooit te hoog. Als dus volgens F_k een tak verworpen kan worden dan is dat in geen geval ten onrechte.

15.3.1 De branch-and-bound methode

In zekere zin is in deze functies F_k meer informatie onder te brengen dan in de tweewaardige functies die we bij het aftelprobleem construeerden. Daarvan kunnen we een systematisch gebruik maken: zodra we een volledige kandidaatrij $(x_1, \ldots, x_n)$ geconstrueerd hebben, kennen we een **grenswaarde** ("bound") voor de minimale waarde van de oplossing, die we kunnen gebruiken om die partiële kandidaatrijen $(x_1, \ldots, x_k)$ uit te sluiten die nooit meer tot een beter resultaat kunnen voeren. Naarmate we betere oplossingen vinden kunnen we zo steeds grotere delen van de selectieboom uitsluiten waarvoor geldt $F_k \geq$ *grenswaarde*.

Symbolisch kunnen we dit aangeven in een selectieboom die zodanig vertekend is, dat de lengte van een pad vanaf de wortel gelijk is aan de waarde, die de criteriumfunctie F_n daarvoor aanneemt.

Figure 15.5: Vertekende selectieboom

Men noemt deze variant van de backtrack-algoritme de **branch-and-bound algoritme** — een naam waarin zowel het "vertakken" van de selectieboom, als de "begrenzing" aan de hand van gevonden oplossingen doorklinkt.

15.4 Voorbeeld: kortste weg

Tussen een aantal steden, doorlopend genummerd van 1 tot n, bevindt zich een stelsel van wegen, zodanig dat er tussen ieder paar steden i en j hetzij geen directe verbinding is, hetzij een *directe weg* van een bepaalde lengte dw_{ij} (een reëel getal).

Voorbeeld:

Figure 15.6: Wegenplan met afstanden

Gezocht wordt een algoritme `kortste weglengte (i, j)` die de lengte berekent van een kortste weg van i naar j. Er kunnen uiteraard meer kortste wegen zijn, maar deze hebben dan wel alle dezelfde lengte.

Hoe geven we aan dat er geen directe weg is van i naar j? We zullen aannemen dat dit gecodeerd is door middel van een hele hoge waarde waarde voor dw_{ij}, bijvoorbeeld `oneindig`, zodat als er een kortste weg is, daarin niet dit stuk directe weg voorkomt (vinden we zo'n grote waarde voor de kortste weg dan is er dus helemaal geen weg).

De matrix `dw` is symmetrisch (tenzij we eenrichtingswegen invoeren).

```
  LET n = ... ;
  ROW n ROW n REAL VAR dw;
  REAL CONST oneindig :: max real / round
(n);
  leer wegenstelsel;
  bepaal de kortste weg van i naar n.
```

Als variabelen kiezen we de beginpunten van aaneensluitende directe wegstukken. In termen van de definitie van het optimalisatie-probleem: vind die rij van directe wegen (beginnend in i, onderling aansluitend, lusvrij en eindigend in j) waarvoor de som minimaal is, d.w.z. minimaliseer

$$F_n(x_1, \ldots, x_n) = dw(x_1, x_2) + dw(x_2, x_3) + \ldots + dw(x_{n-1}, x_n)$$

met $x_1 = i$ en $x_n = j$, $n \geq 1$.

Als partiële criteriumfuncties F_k kiezen we uiteraard de lengte van het pad van i tot k. Deze keuze is zinvol omdat voor elke weg van i naar j die met het segment $i, \ldots, k$ begint geldt:

$$F_n(x_1, \ldots, x_k, \ldots, x_n) \geq F_k(x_1, \ldots, x_k)$$

De backtrack-algoritme is in essentie

```
voortzettingen vanaf p:
  INT VAR q;
  FOR q UPTO n
  REP
    IF er is een directe weg van p naar q
    THEN verhoog partiele lengte met
            die directe weg;
         onderzoek voortzetting via q;
         verlaag partiele lengte met
            die directe weg
    FI
  ENDREP.
```

Zodra een weg is gevonden is de lengte daarvan een grenswaarde voor de partiële criteriumfuncties. Is een partiële weg langer dan de grenswaarde, dan hoeven we geen enkele voortzetting na te gaan, ze zijn allemaal langer dan de grenswaarde. Vinden we een nieuwe weg korter dan de grenswaarde, dan stellen we de grenswaarde bij.

```
onderzoek voortzetting via q:
  { via q naar j }
  IF partiele lengte < grenswaarde
  THEN
    IF q=j
    THEN grenswaarde := gevonden lengte
    ELSE op het pad [q] := TRUE;
         voortzettingen vanaf q;
         op het pad [q] := FALSE
    FI
  FI.
```

Om terminatie te garanderen zullen we er op toezien, dat de partiële weg lusvrij is. We moeten daartoe bijhouden welke steden op de weg liggen en elke nieuwe stad daartegen testen. Daarvoor is voldoende voor elke stad een BOOL variabele bij te houden. Merk overigens op dat we daaruit niet direct de weg zelf kunnen reconstrueren.

De algoritme wordt in gang gezet door het beginpunt te onderzoeken (immers i kan gelijk zijn aan j). Aanvankelijk is dus de partiële weglengte nul en de grenswaarde oneindig.

Hiermee kunnen we de algoritme concretiseren:

```
bepaal de kortste weg van i naar n:
  REAL VAR grenswaarde :: oneindig;
  ROW n BOOL VAR op het pad;
  geen stad ligt op het pad;
  zoek kortste weg (i, j, 0.0).

PROC zoek kortste weg (INT i, j, REAL
weglengte):
  IF verder gaan heeft geen zin
  THEN
  ELIF doel bereikt
  THEN verbeter grenswaarde
  ELSE onderzoek alle voortzettingen
  FI.
```

De parameter weglengte bevat de geaccumuleerde lengte van de weg tot i aan toe. Hij speelt de rol van F_k.

```
  verder gaan heeft geen zin:
    weglengte >= grenswaarde.

  doel bereikt:
    i = j.

  verbeter grenswaarde:
    grenswaarde := weglengte.
```

Merk op dat dit altijd een verbetering geeft.

```
  onderzoek alle voortzettingen:
    INT VAR k;
    FOR k UPTO n
    REP
      onderzoek voortzetting via k
    ENDREP.

  onderzoek voortzetting via k:
    IF NOT op het pad [k] AND
       er is een directe weg van i naar k
    THEN op het pad [k] := TRUE;
         zoek kortste weg (k, j,
           weglengte + dw [i][k]);
         op het pad [k] := FALSE
    FI.

  er is een directe weg van i naar k:
    dw [i][k] < oneindig.

ENDPROC zoek kortste weg;
```

15.4.1 Resultaten

Uit deze optimalisatie-algoritme is heel eenvoudig een aftel-algoritme te maken door één verfijning te wijzigen

```
  verder gaan heeft geen zin:
    FALSE.
```

Dit stelt ons in staat het effect van de branch-and-bound heuristiek aan een voorbeeld na te gaan. Als maat voor de hoeveelheden te verzetten werk nemen we het aantal malen dat de procedure `zoek kortste weg` wordt opgeroepen. Dit aantal hangt af van het aantal steden, het aantal (eindige) directe wegen in de matrix `dw` en van hun verdeling en verhoudingen.

Wij genereren daartoe telkens een "willekeurige" matrix met een bepaalde *vullingsgraad*, d.w.z., waarvan een gegeven percentage van de wegen niet oneindig lang is. Daarbij zullen we de directe weg van 1 naar `n` uitsluiten.

```
vul de matrix:
  INT VAR i;
  FOR i UPTO n
  REP FOR j FROM i+1 UPTO n
    REP
      REAL CONST afstand ::
        willekeurig getal tussen 0 en 100;
      dw [i][j] := afstand;
      dw [j][i] := afstand
    ENDREP;
    dw [i][i] := 0.0
  ENDREP;
  dw [1][n] := oneindig;
  dw [n][1] := oneindig.
```

We maken gebruik van de procedure `random` die bij iedere aanroep een pseudo-random real tussen 0.0 en 1.0 oplevert.

```
willekeurig getal tussen 0 en 100:
  IF random < vullingsgraad
  THEN random * 100.0
  ELSE oneindig
  FI.
```

Vervolgens bepalen we het aantal aanroepen in `zoek kortste weg (1, n)` met behulp van een geïnstrumenteerde (d.w.z. van tellertjes voorziene) versie van de beide algoritmen.

Aangezien backtrack-algoritmen een zeer sterk fluctuerend gedrag vertonen middelen we de resultaten over tien keren en ronden ze af. Desondanks laten de gemeten aantallen nog enige wilde sprongen zien, ze geven echt niet meer dan een idee van het gedrag van beide algoritmen.

Hieronder volgen de resultaten van zo'n simulatie.

Deze getallen vallen zeer sterk in het voordeel van de branch-and-bound methode uit. Wel wordt duidelijk dat bij grote aantallen steden vermoedelijk ook deze methode prohibitief veel rekenwerk kan meebrengen.

Daarbij moet wel worden bedacht dat de "willekeurige" keuze van de `dw [i][j]` niet bepaald realistisch was: bij grote stedenaantallen zal iedere stad verbonden zijn met een vrij klein aantal naburige steden, terwijl directe verbindingen tussen wat verder verwijderde steden nauwelijks voorkomen. Met als enig hulpmiddel de vullingsgraad is dit niet bevredigend te modelleren. Het is dus voorshands niet uit te sluiten dat een aangepaste versie van de branch-and-bound methode voor dit probleem ook in realistische toepassingen bruikbaar is. Overigens bestaan er voor dit simpele "kortste afstand" probleem veel efficiënter algoritmen.

vullingsgraad	aantal steden	aftel-algorithme	Branch-and-bound
.1	5	7	6
	8	12	10
	11	27	20
	14	52	37
	17	127	68
.3	5	11	12
	8	102	40
	11	1867	99
	14	4728	305
	17	> 10.000	1090
.5	5	32	18
	8	356	77
	11	> 10.000	280
	14		471
	17		1661

Table 15.1: Aantal oproepen van `zoek kortste weg`

15.5 Opgaven

1. (permutaties) Druk alle permutaties af van de getallen $1 : n$.
2. Uit een plaat van n bij n moet een aantal vierkanten met onderling verschillende geheeltallige ribben tussen 1 en $n-1$ gesneden worden, zodanig dat een zo groot mogelijk deel van de plaat wordt benut.
3. (partities) Onder een partitie van een getal $n > 0$ verstaan we een rijtje $x_1, x_2, \ldots, x_m$ met $x_i > 0$ en $x_1 + x_2 + \ldots + x_m = n$. Schrijf een recursief programma dat alle verschillende partities van een ingelezen getal bepaalt.
4. (rondgang van een Paard) Schrijf een recursief programma dat één rondgang van een paard uitgaand van een gegeven beginpunt over een schaakbord van 5*5 vindt, zodanig dat ieder veld precies eenmaal bezocht wordt.
5. (gepast betalen) Een bedrag aan geld kan op verschillende wijzen worden samengesteld uit een aantal munten. Schrijf een recursief programma dat, gegeven een bedrag en een muntstelsel, vaststelt op hoeveel verschillende manieren dit bedrag in dat muntstelsel gevormd kan worden. Probeer dit met een aantal bedragen < 1000 en de volgende muntstelsels:

(H)	10, 20, 50, 100, 200, 500,
(NL)	1, 5, 10, 25, 100, 250,
(BRD)	1, 2, 5, 10, 20, 50, 100, 200, 500,
(SU)	1, 2, 3, 5, 10, 15, 20, 50, 100,

6. (paardesprongen) Op een (verder leeg) schaakbord staat een Paard. Schrijf een recursief algoritme dat, gegeven de positie

van het Paard, alle velden van het schaakbord markeert met met minimum aantal sprongen waarmee het Paard op dat veld kan komen.

7. (kamelen) In een tunnel ontmoeten twee karavanen elkaar, ieder bestaand uit n kamelen. Tussen beide karavanen (die bestaan uit aaneensluitende kamelenlijven) is nog een kameellengte vrij. De beginsituatie voor $n = 4$ kan bijvoorbeeld worden weergegeven als
   pppp-qqqq.
   Een kameel die staat voor een lege plek kan een kameellengte doorlopen. Een kameel die door een ander kameel wordt gescheiden van een lege plek kan over die andere heenspringen. Geef een kortste reeks van kameelbewegingen die de rechtsgaande en de linksgaande karavaan van plaats verwisselt.
8. (spiraal) Schrijf een programma dat de nauwste rechtsdraaidende spiraal afdrukt van 100 vakjes, beginnende in de oorsprong $(0, 0)$ en naar het noorden. Daarbij moet ieder vakje van de spiraal een zijde met zijn directe voorganger en met zijn directe opvolger gemeen hebben, terwijl een vakje met een ander vakje dat geen directe opvolger of voorganger is geen zijde of hoekpunt gemeen mag hebben. Houd rekening met het feit dat er hindernissen op het bord kunnen staan, die ook niet geraakt mogen worden.

Backtrack-opgaven zijn overigens legio te vinden in de sfeer van spelletjes en puzzels of als optimalisatieproblemen in de Operations Research.

Chapter 16

Van recursie naar iteratie

Iedere enkelvoudig iteratieve algoritme, van de vorm

```
    a;
    WHILE b REP c ENDREP;
    d
```

kan klaarblijkelijk worden herschreven in de equivalente recursieve vorm

```
  PROC bc:
    IF b THEN c; bc FI
  ENDPROC bc;
    a; bc; d
```

In dit hoofdstuk zullen we aantonen dat omgekeerd iedere recursieve algoritme omgezet kan worden in een equivalente iteratieve, al is dit niet steeds eenvoudig. We behandelen daartoe **recursie eliminatie**, een voorbeeld van een correctheid-bewarende **programma transformatie**.

In een moderne programmeertaal met een goede implementering van recursie heeft deze transformatie op het eerste gezicht niet veel zin. Deze transformatie is echter onderdeel van een verzameling van transformaties waarmee het mogelijk is, een eenmaal correcte algoritme door kleine herschrijvingen met behoud van correctheid om te zetten in een efficiënter exemplaar (**transformationele programmering**). De belangrijkste reden echter waarom wij dit onderwerp behandelen is het feit dat daardoor een verhelderend licht wordt geworpen op vrijheidsgraden en keuzen bij het programmeren.

Omwille van de eenvoud zullen we ons beperken tot de behandeling van procedures met een enkele parameter die geen resultaat leveren (acties), van de vorm

```
  PROC p (INFO x):
    IF   c(x)
    THEN n(x); p(f(x)); m(x)
    ELSE d(x)
    FI
  ENDPROC p;
```

waarbij `c(x)`, `d(x)`, `n(x)`, `f(x)` en `m(x)` staan voor stukken programma waarin `x` mag voorkomen maar die geen locale declaraties bevatten waarvan een toepassing in een van de andere stukken voorkomt. De parameter x is dus de enige locale grootheid. Voor het abstracte type `INFO` kan naar believen ieder concreet type worden ingevuld.

16.1 Het verwijderen van rechts-recursie

Beschouw het volgende schoolvoorbeeld van een rechts-recursieve procedure

```
  PROC p (INFO x):
    IF   c(x)
    THEN n(x); p(f(x))
    ELSE d(x)
    FI
  ENDPROC p;
```

Hierbij zijn `c(x)`, `d(x)`, `n(x)` en `f(x)` zoals gebruikelijk stukken programma waarin `x` optreedt en is `p(f(x))` een recursieve aanroep van `p`.

We kunnen de rechts-recursie elimineren door invoering van een hulpvariabele `x1` en een `WHILE` vorm

```
  PROC p (INFO x):
    INFO VAR x1 :: x;
    WHILE c(x1)
    REP n(x1);
      x1 := f(x1)
    ENDREP;
    d(x1)
  ENDPROC p;
```

De equivalentie van beide procedures valt bijvoorbeeld te bewijzen door inductie.

Als voorbeeld nemen we een procedure om de som van de eerste `n` elementen van een gegeven rij van reals te bepalen:

```
    REAL VAR totaal :: 0.0;
  PROC sommeer (INT n):
    IF   n >= 1
    THEN totaal INCR rij [n];
         sommeer (n-1)
    FI
  ENDPROC sommeer;
```

Na transformatie en vereenvoudiging krijgen we

```
    REAL VAR totaal :: 0.0;
  PROC sommeer (INT n):
    INT VAR n1 :: n;
    WHILE n1 >= 1
    REP totaal INCR rij [n];
        n1 := n1 - 1
    ENDREP
  ENDPROC sommeer;
```

16.2 Voorbeeld: toepassing op Quicksort

Toepassing van deze transformatie op

```
PROC quicksort (INT lwb, upb):
 IF upb-lwb > 0
 THEN
 INT VAR p, q;
   splits (lwb,upb,p,q);
   quicksort (lwb,q);
   quicksort (p,upb)
 FI
ENDPROC quicksort;
```

kan alleen de laatste recursieve aanroep elimineren. Ook wordt de transformatie bemoeilijkt door het feit dat de locale variabelen `p` en `q` in de aanroepen van `quicksort` voorkomen. De declaratie van `p` en `q` kan echter zonder problemen buiten de herhaling plaatsvinden. Enig puzzelen voert tot

```
PROC quicksort (INT lwb, upb):
  INT VAR lwb1 :: lwb;
  INT VAR p, q;
  WHILE upb-lwb1 > 0
  REP splits (lwb1,upb,p,q);
      quicksort (lwb1,q);
      lwb1 := p
  ENDREP
ENDPROC quicksort;
```

De toewijzing `lwb1 := p` is hier cruciaal. Deze correspondeert met `x1 := f(x1)` uit het transformatieschema. De op deze wijze verkregen procedure is iets sneller dan het origineel en gebruikt wat minder hulpgeheugen. Hij is niet meer rechts-recursief maar nog wel middel-recursief.

Een verdere verbetering is te bereiken door gebruik te maken van het feit dat de intervallen `[lwb:q]` en `[p:upb]` waarschijnlijk niet even lang zijn. Als we telkens alleen voor het kortere interval in recursie gaan beperkt dat de recursiediepte aanzienlijk. Dit idee leidt tot

```
PROC quicksort (INT i, j):
  INT VAR lwb :: i, upb :: j, p,q;
  WHILE upb-lwb > 0
  REP splits (lwb,upb,p,q);
    IF q-lwb < upb-p
    THEN
      quicksort (lwb,q);
      lwb := p
    ELSE
      quicksort (p,upb);
      upb := q
    FI
  ENDREP
ENDPROC quicksort;
```

Merk op dat hier een creatief gebruik gemaakt wordt van de (reeds eerder gesignaleerde) overspecificatie van de volgorde van sorteren van de beide helften.

Rechtsrecursie is zo eenvoudig te elimineren dat sommige vertalers (bijvoorbeeld voor LISP en ook die voor de taal CDL3 waarmee Elan geïmplementeerd is) alle rechts-recursies automatisch verwijderen.

16.3 Het verwijderen van middel-recursie

Als de recursie niet meer "rechts" plaatsvindt wordt het moeilijker. Beschouwen we het volgende schema:

```
PROC p (INFO x):
  IF c(x)
  THEN n(x); p(f(x)); m(x)
  ELSE d(x)
  FI
ENDPROC p
```

Het probleem is dat `m(x)` even vaak moet gebeuren als `n(x)` en voor dezelfde reeks waarden van `x`, maar in omgekeerde volgorde.

Schematisch kunnen we ons de uitvoering van `p` in afhankelijkheid van het aantal keren dat `c(x)` `TRUE` oplevert als volgt voorstellen:

0 keer: $\overline{\mathtt{c(x)}}$`; d(x)`
(met het streepje boven de `c(x)` geven we de eerste aanroep van `c` aan die `FALSE` levert).
1 keer: `c(x); n(x);`
   $\overline{\mathtt{c(f(x))}}$`; d(f(x));`
`m(x)`
2 keer: `c(x); n(x);`
   `c(f(x)); n(f(x));`
     $\overline{\mathtt{c(f(f(x)))}}$`;d(f(f(x)));`
   `m(f(x));`
`m(x)`
enzovoort.

We moeten `p` zien te herschrijven tot een iteratieve procedure die bij uitvoering dezelfde reeks van aanroepen van `c`, `n`, `p`, `f` en `m` doorloopt en met dezelfde argumenten als bij invoering van de originele procedure `p`. Een mogelijke oplossing maakt gebruik van een stapel van `INFO's`.

16.3.1 De stapel

Onder een **stapel** oftewel **stack** verstaan we een gegevensstructuur voor het opslaan van elementen die weer teruggehaald moeten worden in de omgekeerde volgorde van opbergen ("last-in-first-out", "LIFO-stack", "LIFO-list"). Een fysiek model van een dergelijke stapel vind je in de meeste cafetaria's: een stapel borden of dienbladen waar uitsluitend bovenaan een exemplaar aan kan worden toegevoegd of weggenomen (tenzij men gevaarlijke toeren gaat uithalen). Een stapel bevat op ieder moment een aantal elementen. Is dit aantal nul, dan is de stapel leeg.

Typische operaties op een stapel, met hun engelse benamingen, zijn:

`push` bovenaan toevoegen van een element. Hierdoor neemt het aantal elementen met één toe.

pop wegnemen van het laatst toegevoegde element. Het aantal elementen neemt met één af.

Daarnaast kan (via een voorwaarde) worden vastgesteld of de stapel leeg is. Een stapel kan een a priori begrenzing hebben op het aantal elementen, bijvoorbeeld omdat er een vaste hoeveelheid ruimte voor is gereserveerd. In dat geval kan ook worden vastgesteld of de stapel vol is zodat er geen element meer toegevoegd kan worden. (Overigens is iedere stapel vanzelfsprekend begrensd, omdat de cafetaria resp. de computer niet oneindig groot zijn. Er zijn echter stapels denkbaar waar de bovengrens niet a priori te bepalen is.)

Een voor de hand liggende realisering van een stapel van elementen (van het abstracte type `EL`) gebruikt een rij van elementen. Zo'n rij heeft een vast aantal elementen, waardoor de bovengrens ook vast ligt. Met deze representatie kunnen we de stapel als volgt realiseren:

```
maak een lege stapel:
  LET max = ...;
  ROW max EL VAR stapel;
  INT VAR top :: 0.
```

De integer variable `top` geeft de index aan van het laatste gevulde element, wat tevens het aantal elementen in de stapel aangeeft. Deze variabele mag geen waarde groter dan `max` aannemen.

Het toevoegen en verwijderen zullen we stapelen resp. ontstapelen noemen.

```
stapel x:
  IF stapel is vol
  THEN explosie
  ELSE top INCR 1;
       stapel[top] := x
  FI.

stapel is vol:
  top >= max.

ontstapel x:
  IF stapel is leeg
  THEN implosie
  ELSE x := stapel[top];
       top DECR 1
  FI.

stapel is leeg:
  top < 1.
```

In geval van `implosie` of `explosie` zal er vermoedelijk niet veel anders opzitten dan het probleem te melden en de uitvoering van het programma vroegtijdig te beëindigen.

Met de taalmiddelen die we later krijgen kunnen we een stapel wat eleganter realiseren (bijvoorbeeld als een apart pakket dat een abstract type `STACK` definieert). Ook kan een zodanige representatie worden gekozen (als "lineaire lijst") dat de noodzaak een a priori begrenzing aan de stack te geven komt te vervallen. In deel twee komen we op de stapel terug.

Met behulp van een stapel kunnen we de middelrecursieve procedure herschrijven tot:

```
PROC p (INFO x):
  maak een lege stapel;
  INFO VAR x1 :: x;
  WHILE c(x1)
  REP
    n(x1);
    stapel x1;
    x1 := f(x1)
  ENDREP;
  d(x1);
  WHILE NOT stapel is leeg
  REP
    ontstapel x1;
    m(x1)
  ENDREP
ENDPROC p;
```

De stapel bevat hierbij de argumenten voor de later uit te voeren aanroepen van `m`.

Dit stuk programma lijkt wel wat op de code, die een vertaler van de recursieve versie maakt. Weer zitten we het werk te doen dat we aan een goede vertaler zouden moeten kunnen overlaten.

16.3.2 Voorbeeld: afdrukken van een natuurlijk getal

Bij wijze van voorbeeld beschouwen we een recursieve procedure `print getal` voor het ruimtesparend afdrukken van natuurlijke getallen, onder de aanname dat we als hulpmiddel uitsluitend een procedure `print cijfer` hebben die één cijfer kan afdrukken.

De recursieve oplossing uit hoofdstuk 12 luidt ongeveer:

```
PROC print getal (INT n):
  IF n >= 10
  THEN
    print getal (n DIV 10);
    print cijfer (n MOD 10)
  ELSE
    print cijfer (n)
  FI
ENDPROC print getal;
```

Herschrijven onder gebruik van een stapel levert:

```
PROC print getal (INT n):
  INT VAR n1 :: n;
  maak een lege stapel;
  WHILE n1 >= 10
  REP stapel n1;
      n1 := n1 DIV 10
  ENDREP;
  print cijfer (n1 MOD 10);
  WHILE NOT stapel is leeg
  REP ontstapel n1;
      print cijfer (n1 MOD 10)
  ENDREP.
```

```
maak een lege stapel:
  ROW 12 INT VAR stapel;
    INT VAR top :: 0.
```

We hebben hier aangenomen dat het integer bereik zodanig is dat het getal hoogstens 12 cijfers telt. We hoeven daarom niet expliciet erop toe te zien dat de `top` niet te hoog komt.

```
stapel n1:
  top INCR 1; stapel[top] := n1.

ontstapel n1:
  n1 := stapel[top]; top DECR 1.

ENDPROC print getal;
```

16.3.3 Voorbeeld: faculteitsfunctie

Laten we nu proberen dezelfde methode los te laten op een “klassieker”: de faculteitsfunctie.

```
INT PROC faculteit (INT n):
  IF n = 1
  THEN 1
  ELSE n * faculteit (n-1)
  FI
ENDPROC faculteit;
```

Op het eerste gezicht is deze procedure rechts-recursief, maar er schuilt een addertje onder het gras: het schema, gegeven in 16.1 is niet van toepassing, want het is geen actie maar een functie. Met behulp van een globale variabele, waarin het resultaat wordt opgebouwd, kunnen we er een actie van maken.

```
  INT VAR resultaat;

PROC bereken faculteit (INT n):
  IF n = 1
  THEN bereken faculteit (n-1);
       resultaat:= n * resultaat
  ELSE resultaat:= 1
  FI
ENDPROC bereken faculteit;
```

Maar weer is het schema voor rechts-recursie eliminatie niet van toepassing: de procedure is middel-recursief. We passen daarom het schema uit 16.3.1 toe, en voeren een stapel in van 44 elementen.

```
PROC bereken faculteit (INT n):
  ROW 44 INT VAR stack; INT VAR top::1;
  INT VAR n1 :: n;
  WHILE n1 <> 1
  REP # n(x) ontbreekt (is leeg) #
      stack[top]:= n1; top INCR 1;
      n1:= n1 - 1
  ENDREP;
  resultaat:= 1;
  WHILE top > 0
  REP n1:= stack[top]; top DECR 1;
      resultaat:= n1 * resultaat
  ENDREP
ENDPROC;
```

Inspectie van deze procedure leert ons dat geldt $stack_i = n - i + 1$; invullen daarvan en weglaten van de eerste (overbodige) herhaling leidt tot

```
PROC bereken faculteit (INT n):
  ROW 44 INT VAR stack; INT VAR top::1;
  INT VAR n1 :: n;
  resultaat:= 1;
  WHILE top > 0
  REP n1:= n - top + 1; top DECR 1;
      resultaat:= n1 * resultaat
  ENDREP
ENDPROC;
```

wat na enige verdere vereenvoudiging, en terugtransformatie naar een functie, de voor de hand liggende oplossing oplevert.

16.3.4 Gebruik van een inverse functie

Een eenvoudiger herschrijving zonder gebruik van een stapel is mogelijk mits er voldaan is aan de volgende twee voorwaarden:

- naast de functie `f(y)` hebben we ook zijn inverse `g(y)`, met de eigenschap: `g(f(y)) = y` voor elke `y` in de reeks `x`, `f(x)`, `f(f(x)), ...`
- alle elementen uit de reeks `x`, `f(x)`, `f(f(x))`, `...` zijn verschillend.

Als aan deze twee voorwaarden voldaan is kunnen we de gegeven **inverse functie** gebruiken om de recursieve procedure te transformeren tot:

```
PROC p (INFO x):
  INFO VAR x1 :: x;
  WHILE c(x1)
  REP n(x1);
      x1 := f(x1)
  ENDREP;
  d(x1);
  WHILE x1 <> x
  REP x1 := g(x1);
      m(x1)
  ENDREP
ENDPROC
```

Deze transformatie verdient de voorkeur als de berekening van de inverse van `f` “goedkoper” is dan het gebruik van de stapel.

We kunnen de schrijfwijze met een stapel zien als een algemene wijze om de inverse te onthouden in plaats van hem te berekenen.

In het geval van de procedure `print getal` is aan de eerste voorwaarde niet voldaan, aangezien bij `x1 := x1 DIV 10` informatie verloren gaat die niet te herstellen is. Er is natuurlijk wel degelijk een iteratieve versie aan te geven, gebaseerd op deling door geschikte machten van tien, maar deze is niet op evidente wijze te krijgen door herschrijving van de procedure in de aangegeven vorm. Dit komt omdat in zekere zin niet voldoende informatie als parameter werd meegegeven.

16.3.5 Embedding

Vaak is het eenvoudiger een probleem op te lossen door het te beschouwen als een speciaal geval van een meer algemeen probleem. Datkunnen we bijvoorbeeld doen door een hulp-procedure met extra parameters te definiëren. Het parametergebied van de oorspronkelijke procedure wordt als het ware ingebed in een groter parametergebied (**embedding**). Deze parameters moeten tesamen alle informatie beschrijven die in de recursie een rol speelt. We passen dat idee toe op ons vorige voorbeeld.

In plaats van `n` herhaaldelijk door 10 te delen voeren we een extra parameter in, de tienmacht waardoor we willen delen, en laten `n` onveranderd. We declareren daartoe een recursieve hulp-procedure met dezelfde naam maar één parameter meer.

```
PROC print getal (INT n):
  print getal (n, 1)
ENDPROC print getal;

PROC print getal (INT n, tienmacht):
  IF n DIV tienmacht >= 10
  THEN
    print getal (n, 10 * tienmacht);
    print cijfer (n DIV tienmacht MOD10)
  ELSE
    print cijfer (n DIV tienmacht)
  FI
ENDPROC print getal;
```

In vergelijking met het boven gegeven schema is de parameter `INFO x` gesplitst in twee grootheden `INT n, tienmacht`. De (symbolische) toewijzing `x1 := f(x1)` moet dan ook gerealiseerd worden als twee toewijzingen, voor ieder van de componenten een. Ook blijkt `n(x)` leeg te zijn, zodat alleen de toewijzing aan `x1` in de eerste herhaling belandt.

Verwijderen van de middel-recursie uit de hulpprocedure leidt tot:

```
PROC print getal (INT n, tienmacht):
  INT VAR n1 :: n, tienmacht1 ::
tienmacht;
  WHILE n1 DIV tienmacht1 >= 10
  REP n1 := n1;
      tienmacht1 := 10 * tienmacht1
  ENDREP;
  print cijfer (n1 DIVtienmacht1 MOD10);
  WHILE n1 <> n OR tienmacht1 <> tienmacht
  REP
    n1 := n1;
    tienmacht1 := tienmacht1 DIV10;
    print cijfer (n1 DIVtienmacht1 MOD10)
  ENDREP
ENDPROC print getal;
```

De resulterende procedure is iteratief en wordt slechts een keer opgeroepen. We substitueren zijn romp voor de aanroep in de eerste `print getal`, verwijderen de overbodige variabele `n1` en brengen wat kleine vereenvoudigingen aan, met als resultaat:

```
PROC print getal (INT n):
  INT VAR tienmacht :: 1;
  WHILE n DIV tienmacht >= 10
  REP tienmacht := tienmacht * 10
  ENDREP;
  print cijfer (n DIV tienmacht MOD10);
  WHILE tienmacht <> 1
  REP
    tienmacht := tienmacht DIV10;
    print cijfer (n DIVtienmacht MOD10)
  ENDREP
ENDPROC print getal;
```

Een verdere vereenvoudiging is mogelijk, maar dit is al een heel respectabel iteratief programma.

16.3.6 Andere methoden ter verwijdering van recursie

Tot slot willen wij nog een tweetal andere gezichtspunten noemen van waaruit de transformatie recursief naar iteratief met behulp van een stapel kan worden uitgevoerd.

Een eerste methode gebruikt een **incarnatie stapel**: associeer met iedere incarnatie van de procedure een element in een stapel, dat zowel de waarden van de parameters als die van eventuele locale variabelen bevat. Een aanroep van de procedure wordt omgezet in een herhaling van de romp van de procedure, nadat te voren het bij de huidige incarnatie behorend element is gestapeld. Na iedere herhaling van de romp wordt een element ontstapeld.

Nog een andere bekijkwijze is mogelijk: de stapel te zien als een verzameling *nog te verrichten taken*, de **klusjespot**. Zo zal bij een iteratieve versie van Quicksort de stapel een aantal intervallen bevatten die nog te sorteren zijn. Wij zullen deze ideeën niet nader uitwerken omdat het resultaat sprekend lijkt op de reeds genoemde wijze een stapel te gebruiken, alleen de motivatie is anders.

16.4 Conclusie

De bedoeling van de aangegeven transformaties is niet een volledig recept te geven, maar te pogen de essentie van dergelijke transformaties aan te geven. In ieder concreet geval zal enig handen- en voetenwerk nodig zijn voor zo'n transformatie lukt. Enige verdergaande transformatieregels zijn te vinden in [BIRD 1977].

Eliminatie van recursie uit een algoritme heeft maar een zeer beperkt effect op de snelheid. Het is echter vaak mogelijk tegelijkertijd het (impliciete) gebruik van een stapel voor recursie te elimineren. De correctheid van de resulterende iteratieve algoritme is meestal moeilijk in te zien, maar kan worden gegarandeerd door uit te gaan van een correcte recursieve versie en deze onder behoud van correctheid te transformeren. Een aantal belangrijke algoritmen heeft zo'n ontstaansgeschiedenis.

Er bestaat ookmeer een stroming in de programmeermethodiek die het programmeren opvat als een herhaald transformatieproces (van idee via formele specificatie via correcte maar weinig efficiënte realisering tot correcte en bovendien efficiënte realisering). Een souverein exposé van deze opvatting is te vinden in [BAUER 1976].

16.5 Opgaven

1. Elimineer
   ook de middel-recursie uit `quicksort` met gebruikmaking van een stapel. Het eenvoudigst gaat dit door tijdelijk een hulp-procedure

```
PROC quicksort1 (INT VAR p, q):
```

   te declareren, die expliciet geparametriseerd is met de grensindices `p` en `q`, en die globaal acces heeft op de rij `f`.
2. Elimineer de middel-recursie uit het programma voor de 8 koninginnen. Een aparte stack blijkt niet nodig omdat de rij `kolom koningin` die rol kan spelen. Vergelijk het resultaat met dat uit [WIRTH 1971].
3. Elimineer zo veel mogelijk de recursie uit een procedure die de Ackermann-functie realiseert
   $ack(m, n) =$
   $m = 0 \to n + 1$
   $m \neq 0, n = 0 \to ack(m - 1, 1)$
   $m \neq 0, n \neq 0 \to ack(m - 1, ack(m, n - 1))$
   Deze opgave vereist heel wat inventiviteit en doorzettingsvermogen. Nadere informatie over deze functie vindt U onder andere in [SUNDBLAD 1971].

Appendix A

Grammatica van Elan

De hiernavolgende Contextvrije Grammatica van Elan is een vrije parafrase van de officiële syntactische beschrijving [HOMMEL et al. 1977] waarbij de terminologie op enkele plaatsen afwijkend is gekozen.

We geven een Contextvrije Grammatica, in de notatie van hoofdstuk 9. Deze kan worden beschouwd als een samenvatting van de syntax-diagrammen die her en der door de tekst verspreid staan, maar bevat ook die constructies van Elan die nog niet behandeld zijn. Het is dus een grammatica voor de volledige taal.

Er is een bijzondere reden hier niet simpelweg dezelfde plaatjes nog eens bij elkaar af te drukken: het mechanisme van de syntax-diagrammen heeft wel het voordeel van de onmiddellijke inzichtelijkheid voor de beginner maar bergt het gevaar in zich dat grote hoeveelheden informatie in één plaatje worden ondergebracht waarbij de abstractie (in de vorm van nuttige "tussenbegrippen") verloren gaat. Het gebruik van een Contextvrije Grammatica nodigt veel meer uit tot een zorgvuldig "verfijnen" van de begrippen dan het gebruik van plaatjes, terwijl daarbij de overzichtelijkheid behouden kan blijven. Bovendien bestaan er vele mensen (onder wie de auteur) die meer houvast hebben aan een hiërarchisch stelsel van zorgvuldig geverbaliseerde definities dan aan visuele hulpmiddelen.

A.1 Afkortingsregels

Om deze syntax kort en bondig te houden is gebruik gemaakt van een aantal afkortingsregels. Zo zal op verschillende plaatsen een lijst van identifiers voorkomen, met een syntaxregel als

identifier—list:
  identifier;
  identifier, comma—token, identifier—list.

Maar ook zullen bijvoorbeeld lijsten van units voorkomen, met een gelijkvormige regel. Daarom spreken we bij voorbaat af dat bij een begrip N—list (waarin N staat voor enigerlei woord) de volgende regel van toepassing is:

a1) N—list: N; N, comma—token, N—list.

Hieruit volgt dat we voor unit-list niet meer apart hoeven te schrijven

unit—list:
  unit;
  unit, comma—token, unit—list.

We introduceren dus het meta-begrip N—list. De hieruit voortkomende abstractie vergroot tevens enigszins de leesbaarheid van de grammatica.

Verdere afkortingen zijn:

a2) N—option: N; .

Een optioneel iets mag worden weggelaten.

a3) N—sequence: N; N, N—sequence.

Een reeks van dingen bestaat uit een of meer van die dingen achter elkaar.

a4) N—pack: open—token, N—list, close—token.

Een pak is een lijst, ingesloten tussen haakjes.

a5) N—token: comment—option, N-symbol.

Deze regel geeft aan dat vóór ieder symbool een commentaar is toegestaan (maar niet aangeraden — teveel commentaar maakt een programma ook onleesbaar). Achteraan dit hoofdstuk worden representaties van de symbolen aangegeven.

A.2 Programmas en pakketten

elan-program:
  packet—sequence—option, main-packet.

De constructie van pakketten wordt in dit boek nog niet behandeld. In de deeltaal Elan-0 zijn pakketten niet geïmplementeerd.

packet:
  packet-head, packet-body, packet-tail.

packet-head:
  packet—token, packet-name, packet-interface, colon—token.

packet-interface:
  export-interface.

export-interface:
  defines—token, export-name—list.

packet-body:
  packet-paragraph, refinement-train—option;
  packet-paragraph, refinement-train—option,
    period—token, packet-body—option.

packet-paragraph:
  packet-unit;
  packet-unit, semicolon—token, packet-paragraph.

packet-unit:
  basic-declaration;
  closed-declaration;
  expression.

Alleen in een pakket-paragraaf zijn gesloten declaraties toegestaan.

packet-tail:
end-packet—token, packet-name—option, semicolon—token.

main-packet:
packet-body.

A.3 Procedures en operatoren

procedure-declaration:
procedure-head, procedure-body, procedure-tail.

procedure-head:
result—option, proc—token, procedure-name, formal-parameter-specification—pack—option, colon—token.

result:
type-declarer.

formal-parameter-specification:
formal-declarer, formal-parameter-name—list.

procedure-body:
paragraph, refinement-train—option.

paragraph:
unit;
unit, semicolon—token, paragraph;
.

Merk op dat een paragraaf ook leeg kan zijn.

unit:
basic-declaration;
expression.

procedure-tail:
end-proc—token, procedure-name—option.

operator-declaration:
operator-head, operator-body, operator-tail.

operator-head:
result—option, op—token, operator-name,
formal-parameter-specification—pack, colon—token.

operator-body:
procedure-body.

operator-tail:
endop—token, operator-name—option.

A.4 Refinements

refinement-train:
period—token, refinement, refinement-train—option.

refinement:
refinement-head, refinement-body.

refinement-head:
refinement-name, colon—token.

refinement-body:
paragraph.

A.5 Declaraties

basic-declaration:
object-declaration;
synonym-declaration.

object-declaration:
object-declarer, object-association—list.

object-association:
object-name, object-initialization—option.

Een variabele mag een initialisatie hebben, maar een constante moet er een hebben.

object-initialization:
initial—token, expression.

synonym-declaration:
let—token, synonym-association—list.

synonym-association:
synonym-value-association;
synonym-type-association.

synonym-value-association:
synonym-value-name, equal—token, denoter.

synonym-type-association:
synonym-type-name, equal—token, type-declarer.

closed-declaration:
procedure-declaration;
operator-declaration;
type-declaration.

Deze gesloten declaraties zijn in Elan-0 niet geïmplementeerd.

type-declaration:
type—token, type-association—list.

type-association:
type-name, equal—token, type-declarer.

A.6 Declarers

object-declarer:
type-declarer, access-declarer—option.

access-declarer:
const—token;
var—token.

Ontbreekt de access-declarer dan wordt `CONST` aangenomen.

type-declarer:
elementary-type-declarer;
composed-type-declarer.

elementary-type-declarer:
concrete-type-declarer;
abstract-type-declarer.

concrete-type-declarer:
int—token;
real—token;
bool—token;
text—token.

abstract-type-declarer:
type-name.

composed-type-declarer:
row-declarer;
struct-declarer.

row-declarer:
row—token, cardinality, type-declarer.

cardinality:
denoter;
synonym-value-name.

struct-declarer:
struct—token, field-specification—pack.

field-specification:
type-declarer, field-name—list.

formal-declarer:
object-declarer;
procedure-declarer.

procedure-declarer:
result—option, proc—token, parameter-declarer—pack—option.

parameter-declarer:
formal-declarer.

A.7 Uitdrukkingen

expression:
priority-i-formula.

priority-i-formula:
priority-ii-formula, rest-priority-i-formula.

rest-priority-i-formula:
priority-i-operator, priority-i-formula;
.

priority-ii-formula:
priority-iii-formula, rest-priority-ii-formula.

rest-priority-ii-formula:
priority-ii-operator, priority-ii-formula;
.

priority-iii-formula:
priority-iii-i-formula, rest-priority-iii-formula.

rest-priority-iii-formula:
priority-iii-operator, priority-iii-formula;
.

priority-iii-i-formula:
priority-iii-ii-formula, rest-priority-iii-i-formula.

rest-priority-iii-i-formula:
priority-iii-i-operator, priority-iii-i-formula;
.

priority-iii-ii-formula:
priority-iii-iii-formula, rest-priority-iii-ii-formula.

rest-priority-iii-ii-formula:
priority-iii-ii-operator, priority-iii-ii-formula;
.

priority-iii-iii-formula:
priority-iii-iii-i-formula, rest-priority-iii-iii-formula.

rest-priority-iii-iii-formula:
priority-iii-iii-operator, priority-iii-iii-formula;
.

priority-iii-iii-i-formula:
priority-iii-iii-ii-formula, rest-priority-iii-iii-i-formula.

rest-priority-iii-iii-i-formula:
priority-iii-iii-i-operator, priority-iii-iii-i-formula;
.

priority-iii-iii-ii-formula:
priority-iii-iii-iii-formula, rest-priority-iii-iii-ii-formula.

rest-priority-iii-iii-ii-formula:
priority-iii-iii-ii-operator, priority-iii-iii-ii-formula;
.

priority-iii-iii-iii-formula:
monadic-operator, priority-iii-iii-iii-formula;
primary.

A.8 Primaries

primary:
compact-primary;
open-primary.

compact-primary:
simple-primary;
closed-primary.

simple-primary:
denoter;
entity-name.

closed-primary:
choice;
repetition;
display;
open—token, expression, close—token.

choice:
conditional-choice;
numerical-choice.

conditional-choice:
if—token, condition, then-part, else-part—option, end-if—token.

condition:
expression.

Een voorwaarde moet een waarheidswaarde leveren.

then-part:
then—token, paragraph.

else-part:
else—token, paragraph;
elif—token, condition, then-part, else-part—option.

numerical-choice:
select—token, expression, of—token, case-part—sequence,
otherwise-part—option, end-select—token.

De expressie moet een integer leveren.

case-part:
case—token, case-label—list, colon—token, paragraph.

case-label:
denoter;
synonym-value-name.

otherwise-part:
otherwise—token, paragraph.

repetition:
for-part—option, while-part—option,
repeat—token, repetition-body,
until-part—option, end-repeat—token.

for-part:
for—token, variable-name, from-part—option, direction-part—option.

from-part:
from—token, expression.

direction-part:
upto—token, expression;
downto—token, expression.

while-part:
while—token, condition.

repetition-body:
paragraph.

until-part:
until—token, condition.

display:
sub—token, expression—list, bus—token.

open-primary:
call;
subscription;
selection;
abstractor;
concretizer;
terminator.

call:
primary, open—token, actual-parameter—list, close—token.

De primary moet een routine leveren. De actuele parameters moeten in aantal en type daarbij passen.

actual-parameter:
expression;
procedure-declarer, procedure-name.

subscription:
primary, sub—token, expression, bus—token.

De primary moet een rij leveren. De expressie moet een integer leveren.

selection:
primary, period—token, field-name.

De primary moet een structuur leveren, van één van welks velden de field-name de naam moet zijn. Structuren worden in het volgende boek behandeld.

abstractor:
type-name, colon—token, compact-primary.

De abstractor dient voor het denoteren van abstracte waarden. Hij wordt pas in het volgende boek behandeld.

concretizer:
concr—token, compact-primary.

terminator:
leave—token, algorithm-name, premature-result—option.

De beëindiging van een verfijning `x` mag enkel voorkomen in die verfijning `x` zelf, of in één van de verfijningen die door `x` direct of indirect worden aangeroepen.

premature-result:
with—token, compact-primary.

A.9 Namen

export-name:
constant-name;
procedure-name;
operator-name;
type-name.

formal-parameter-name:
variable-name;
constant-name;
procedure-name.

entity-name:
object-name;
procedure-name;
refinement-name;
synonym-value-name.

algorithm-name:
procedure-name;
operator-name;
refinement-name.

object-name:
constant-name;
variable-name.

constant-name:
identifier.

variable-name:
identifier.

procedure-name:
identifier.

refinement-name:
identifier.

synonym-value-name:
identifier.

field-name:
identifier.

packet-name:
identifier.

type-name:
bold-identifier.

synonym-type-name:
bold-identifier.

bold-operator-name:
bold-identifier.

operator-name:
bold-identifier;
special-identifier.

identifier:
letter, letgit—sequence—option.

letgit:
letter;
digit.

bold-identifier:
bold-letter, bold-letter—sequence—option.

special-identifier:
equal—token;
not-equal—token;
less—token;
less-equal—token;
greater—token;
greater-equal—token;
plus—token;
minus—token;
asterix—token;
divide—token;
int-div—token;
obelix—token;
becomes—token;
initial—token.

A.10 Operatoren

priority-i-operator:
becomes—token.

priority-ii-operator:
bold-operator-name.

priority-iii-operator:
or—token;
xor—token.

priority-iii-i-operator:
and—token.

priority-iii-ii-operator:
equal—token;
not-equal—token;
less—token;
less-equal—token;
greater—token;
greater-equal—token.

priority-iii-iii-operator:
plus—token;
minus—token.

priority-iii-iii-i-operator:
asterix—token;
divide—token;
int-div—token;
modulo—token.

priority-iii-iii-ii-operator:
obelix—token.

Dat gedoe met Asterix en Obelix is een grapje uit de taaldefinitie van Elan, zo'n sterretje heet namelijk een asterisk. Het lompe teken voor machtsverheffing kreeg daarom ook een eigentijdse benaming.

monadic-operator:
not—token;
plus—token;
minus—token;
bold-operator-name.

A.11 Denoters

denoter:
comment—sequence—option, denotation.

denotation:
int-denotation;
real-denotation;
bool-denotation;
text-denotation.

int-denotation:
digit—sequence.

real-denotation:
digit—sequence, period-symbol, digit—sequence, exponent-part—option.

exponent-part:
exponent-symbol, sign—option, digit—sequence.

sign:
plus-symbol;
minus-symbol.

bool-denotation:
true-symbol;
false-symbol.

text-denotation:
quote-symbol, text-item—sequence—option, quote-symbol.

text-item:
quote-image;
character-image;
any-character-except-quote-symbol.

quote-image:
quote-symbol, quote-symbol.

character-image:
quote-symbol, digit—sequence, quote-symbol.

Het teken-beeld dient er toe een teken met een bepaalde code (aangeduid door de cijferreeks) te denoteren.

sectionCommentaren

Tussen de symbolen mogen commentaren staan, volgens de afkortingsregel **a5**.

comment:
comment-open-symbol, comment-item—sequence—option,
comment-close-symbol.

comment-item:
(* any symbol other than a comment-open-symbol or
comment-close-symbol *)

A.12 Representaties

De representaties van de genoemde symbolen zijn als volgt:

type-symbol	TYPE
int-symbol	INT
real-symbol	REAL
bool-symbol	BOOL
text-symbol	TEXT
row-symbol	ROW
struct-symbol	STRUCT
const-symbol	CONST
var-symbol	VAR

packet-symbol	PACKET	
endpacket-symbol	ENDPACKET	END PACKET
defines-symbol	DEFINES	
colon-symbol	:	
comma-symbol	,	
semicolon-symbol	;	
period-symbol	.	
proc-symbol	PROC	
end-proc-symbol	ENDPROC	END PROC
op-symbol	OP	
end-op-symbol	ENDOP	END OP
open-symbol	(	
close-symbol	)	
sub-symbol	[	
bus-symbol	]	
initial-symbol	::	
equal-symbol	=	
let-symbol	LET	

if-symbol	IF		
end-if-symbol	FI	ENDIF	END IF
then-symbol	THEN		
else-symbol	ELSE		
elif-symbol	ELIF		
select-symbol	SELECT		
of-symbol	OF		
end-select-symbol	ENDSELECT	END SELECT	
case-symbol	CASE		
otherwise-symbol	OTHERWISE		

repeat-symbol	REP	REPEAT	
end-repeat-symbol	ENDREP	ENDREPEAT	
	END REP	END REPEAT	PER
for-symbol	FOR		
from-symbol	FROM		
upto-symbol	UPTO		
downto-symbol	DOWNTO		
while-symbol	WHILE		
until-symbol	UNTIL		

concr-symbol	CONCR
leave-symbol	LEAVE
with-symbol	WITH
becomes-symbol	:=
or-symbol	OR
xor-symbol	XOR
and-symbol	AND

comment-open-symbol	{	(*
comment-close-symbol	}	*)

not-equal-symbol	<>	
less-symbol	<	
less-equal-symbol	<=	
greater-symbol	>	
greater-equal-symbol	>=	
plus-symbol	+	
minus-symbol	-	
divide-symbol	/	
int-div-symbol	DIV	%
modulo-symbol	MOD	
asterix-symbol	*	
obelix-symbol	**	
not-symbol	NOT	
quote-symbol	"	

Appendix B

Standaard paketten

Een aantal concrete algoritmen, objecten en typen is in Elan ter beschikking bij de gratie van de *standaardpaketten*, die geacht worden vóór het eigenlijke programma te zijn gevoegd. Dit voorvoegen gaat automatisch en ongemerkt, de gebruiker merkt alleen dat hij dingen die daarin gedeclareerd zijn zonder verdere declaratie kan gebruiken. Met name betreft dit de vier concrete typen en hun operaties.

Het volgende overzicht toont de belangrijkste standaard-algoritmen, objecten en typen van Elan, in de vorm van declaraties of delen daarvan (procedure-heading's). Hieruit kan men (ook zonder het hoofdstuk over procedures verwerkt te hebben) afleiden welke typen en aantallen parameters voor de verschillende operaties zijn toegestaan, enz. Voor de betekenis van de niet in dit boek behandelde objecten wordt naar de Elan taalbeschrijving verwezen.

B.1 Prioriteiten van operatoren

De hiernavolgende lijst vat de prioriteiten van alle operatoren van de Elan standaard-paketten samen, waarbij prioriteitstrap 10 de hoogste en 1 de laagste is.

1. De toewijzing en de operatoren gecombineerd met toewijzing

   `:=` `INCR` `DECR` `CAT`
2. Alle abstracte dyadische operatoren (door de gebruiker gedeclareerd; de taalmiddelen daartoe leren we later) evenals `SUB`
3. `OR` `XOR`
4. `AND`
5. `=` `<>`
6. `<=` `<` `>=` `>`
7. `+` `-`
8. `*` `/` `DIV` `MOD`
9. `**`
10. Alle monadische operatoren, zoals

    `+` `-` `LENGTH` `ABS` `SIGN`

B.2 Integer

```
TYPE INT

PROC get (INT VAR x):
PROC put (INT x):
PROC put line (INT x):

BOOL OP = (INT x, y):
BOOL OP <> (INT x, y):
BOOL OP < (INT x, y):
BOOL OP <= (INT x, y):
BOOL OP > (INT x, y):
BOOL OP >= (INT x, y):

INT OP + (INT x, y):
INT OP - (INT (x, y):
INT OP * (INT x, y):
INT OP DIV (INT x, y):
INT OP MOD (INT x, y):
INT OP ** (INT x, y):

INT OP + (INT x):
INT OP - (INT x):

OP INCR (INT VAR x, INT y):
OP DECR (INT VAR x, INT y):

INT OP SIGN (INT x):
INT OP ABS (INT x):
INT PROC sign (INT x):
INT PROC abs (INT x):
BOOL PROC even (INT x):
BOOL PROC odd  (INT x):
INT PROC max (INT a, b):
INT PROC min (INT a, b):

INT PROC trunc (REAL x):
INT PROC round (REAL x):
INT PROC int (TEXT x):

INT CONST maxint
```

B.3 Real

```
TYPE REAL

PROC get (REAL VAR x):
PROC put (REAL x):
PROC put line (REAL x):

BOOL OP = (REAL x, y):
BOOL OP <> (REAL x, y):
BOOL OP < (REAL x, y):
BOOL OP <= (REAL x, y):
BOOL OP > (REAL x, y):
BOOL OP >= (REAL x, y):
```

```
REAL OP + (REAL x, y):
REAL OP - (REAL x, y):
REAL OP * (REAL x, y):
REAL OP / (REAL x, y):
REAL OP MOD (REAL x, y):
REAL OP ** (REAL x, INT y):
REAL OP ** (REAL x, y):

REAL OP + (REAL x):
REAL OP - (REAL x):

OP INCR (REAL VAR x, REAL y):
OP DECR (REAL VAR x, REAL y):

INT OP SIGN (REAL x):
REAL OP ABS (REAL x):
INT PROC sign (REAL x):
REAL PROC abs (REAL x):
REAL PROC max (REAL a, b):
REAL PROC min (REAL a, b):

REAL PROC real (INT x):
REAL PROC real (TEXT x):

REAL CONST smallreal, maxreal

REAL CONST pi
REAL PROC sqrt (REAL x):
REAL PROC sin (REAL x):
REAL PROC cos (REAL x):
REAL PROC tan (REAL x):
REAL PROC arctan (REAL x):
REAL PROC arccos (REAL x):
REAL PROC arcsin (REAL x):

REAL CONST e
REAL PROC exp (REAL x):
REAL PROC ln (REAL x):
REAL PROC log10 (REAL x):
REAL PROC log2  (REAL x):
```

B.4 Text

```
TYPE TEXT

PROC get (TEXT VAR t):
PROC get (TEXT VAR t, INT maxlen):
PROC get (TEXT VAR t, TEXT delimiter):
PROC put (TEXT t):
PROC put line (TEXT x):

BOOL OP = (TEXT x, y):
BOOL OP <> (TEXT x, y):
BOOL OP < (TEXT x, y):
BOOL OP <= (TEXT x, y):
BOOL OP > (TEXT x, y):
BOOL OP >= (TEXT x, y):

INT OP LENGTH (TEXT t):
INT PROC length (TEXT t):

TEXT OP + (TEXT x, y):
     OP CAT (TEXT VAR x, TEXT y):
TEXT OP * (INT i, TEXT t):
TEXT PROC compress (TEXT t):
TEXT PROC text (TEXT t, INT length):
TEXT PROC text (TEXT t, INT length, from):
TEXT PROC subtext (TEXT t, INT from):
TEXT PROC subtext (TEXT t, INT from, to):
TEXT OP SUB (TEXT t, INT p):
     PROC replace (TEXT VAR t, INT from,
                   TEXT new):
     PROC change (TEXT VAR t, TEXT old,
                   new):
     PROC change all (TEXT VAR t, TEXT
old,
                   new):
INT PROC pos (TEXT t1, t2):
INT PROC pos (TEXT t1, t2, INT from):

TEXT OP HEAD (TEXT t):
TEXT OP TAIL (TEXT t):
TEXT PROC ascii (INT code):

TEXT PROC text (INT i):
TEXT PROC text (REAL r):

TEXT CONST niltext :: ‘‘"
TEXT CONST blank :: ‘‘ ‘‘
TEXT CONST quote :: ‘‘"""
```

B.5 Boolean

```
TYPE BOOL
BOOL CONST false, true

BOOL OP NOT (BOOL b):
BOOL OP AND (BOOL b1, b2):
BOOL OP OR (BOOL b1, b2):
BOOL OP XOR (BOOL b1, b2):
BOOL OP = (BOOL b1, b2):
BOOL OP <> (BOOL b1, b2):
```

B.6 Bestandsverwerking

Elan kent een vrij uitgebreide verzameling bestandsoperaties, waarvan slechts een deel (met enige toevoegingen) ook in Elan-0 ter beschikking staat.

B.6.1 Standaard bestandsverwerking

Zowel in Elan-1 als in het EUMEL systeem is het volgende stel bestandsoperaties beschikbaar.

```
TYPE FILE, DIRFILE

FILE PROC sequential file
(TRANSPUTDIRECTION d,
                           TEXT ident):
FILE PROC direct file (TRANSPUTDIRECTION
d,
                       TEXT ident):
TRANSPUTDIRECTION CONST input, output,
update
```

```
PROC close (FILE f):
PROC close (DIRFILE f):

PROC erase (FILE f):
PROC erase (DIRFILE f):

PROC putline (FILE f, TEXT t):
PROC getline (FILE f, TEXT VAR t):

INT PROC maxlinelength (FILE f):
INT PROC maxlinelength (DIRFILE f):
INT PROC maxpagelength (FILE f):

PROC line (FILE f):
PROC line (FILE f, INT i):
PROC page (FILE f):
PROC reset (FILE f):

PROC put (FILE f, INT i):
PROC put (FILE f, REAL r):
PROC put (FILE f, TEXT t):

PROC get (FILE f, INT VAR i):
PROC get (FILE f, REAL VAR r):
PROC get (FILE f, TEXT VAR t):
PROC get (FILE f, TEXT VAR t, TEXT
          delimiter):
PROC get (FILE f, TEXT VAR t, INT maxlen):

PROC putline (DIRFILE f, TEXT key, t):
PROC getline (DIRFILE f, TEXT key,
          TEXT VAR t):

BOOL PROC opened (FILE f):
BOOL PROC opened (DIRFILE f):

BOOL PROC new (FILE f):
BOOL PROC new (DIRFILE f):
BOOL PROC eof (FILE f):
```

B.6.2 Bestandsverwerking in Elan-0

De bestandsoperaties van Elan-0 zijn ook in Elan-1 beschikbaar.

```
PROC new file (TEXT name):
PROC old file (TEXT name):
PROC close file :
PROC erase file (TEXT name):
PROC write (INT x):
PROC write (TEXT x):
PROC writeline :
PROC read (INT VAR x):
PROC read (TEXT VAR x):
BOOL PROC file ended:
```

B.7 Schermbesturing

```
PROC line:
PROC line (INT i):
PROC page:
PROC cursor (INT x,y):
PROC get cursor (INT VAR x, y):

INT CONST max line length

PROC edit (TEXT VAR t, INT p):
PROC sleep (INT seconds):
PROC sleep (REAL seconds):
```

B.8 Voortbrengen van random getallen

```
PROC initialize random (INT a):
PROC initialize random (REAL a):
INT PROC random (INT a, b):
REAL PROC random:

INT PROC choose128:
```

Bibliography

Verwijzingen bij de inleiding

[BAU81] F.L.Bauer, H. Wössner, *Algoritmische Sprache und Programmentwicklung*, Springer Verlag, Berlin 1981.

[GRI81] D.Gries, *The Science of Programming*, Springer Verlag, New York, 1981.

[KOS91] C.H.A. Koster, *Systematisch leren programmeren, deel 2: Bottom-Up programmering*, ISBN 90-6233-382-6, Academic Service, Schoonhoven, 1991.

Verwijzingen bij hoofdstuk 1: *Algoritmen*

[EXN70] Eva Exner: *Braten und Schmoren in Römertopf.* Eduard Bay, D 5412 Ransbach, 1970.

[SAM76] J.E. Sammet, *Roster of Programming Languages for 1974-1975*, Communications of the ACM. Vol 19 No 12, Dec 1976. De babylonische spraakverwarring op het gebied van programmeertalen.

[DAH72] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, Structured Programming. Academic Press, 1972. Aan de hand van argumentatie en goede voorbeelden wordt in dit boek de techniek van het gestructureerd programmeren uiteengezet.

[HOM79] G. Hommel et al., Elan Sprachbeschreibung. Akademische Verlagsgesellschaft, Wiesbaden 1979. De officiële beschrijving van Elan.

Verwijzingen bij hoofdstuk 2: *Notatie van algoritmen*

[PAT81] Richard E. Pattis, Karel the Robot: a gentle Introduction to the Art of Programming. John Wiley and Sons, 1981.

[PAP80] S. Papert, Mindstorms. Basic Books, Inc., Harper Colophon Books, 1980. Beschrijft de onderwijsfilosofie die ten grondslag ligt aan de programmeertaal LOGO en ook, in een veel bescheidener vorm, aan ons gebruik van Kareltje als toegang tot het algoritmisch denken.

Verwijzingen bij hoofdstuk 3: *Gehele getallen*

[ADM72] Richard Adams, Watership Down. Penguin Books Ltd. Middlesex England (1972).

[KNU72] D.E. Knuth, The Art of Computer Programming, Vol 1, Fundamental Algorithms. Addison-Wesley Publishing Company, 1972.

[CRA87] D. Craemer: *Abstrakte Maschinen und modulares Programmieren in ELAN.* GMD ITF Bericht, Bonn 1987.

Verwijzingen bij hoofdstuk 4: *Reele getallen*

[PIJ82] W. Pijls, Programmeeropgaven 1 en 2. Educaboek 1982.

[STO72] J. Stoer, Einführung in die numerische Mathematik I. Heidelberger Taschenbücher, 1972.

[RAL65] A. Ralston, A first Course in Numerical Analysis. Mc-Graw - Hill, 1965.

[VDG78] J. van der Graft, Introduction to Numerical Computations. Academic Press, 1978

[DAB74] G. Dahlquist and A. Björk, Numerical Methods, Prentice Hall, 1974

Verwijzingen bij hoofdstuk 5: *Waarheidswaarden*

[KNU69] Donald E. Knuth, The Art of Computer Programming, Vol 2, Seminumerical Algorithms. Addison-Wesley Publishing Company, 1969.

Verwijzingen bij hoofdstuk 6: *Teksten*

Verwijzingen bij hoofdstuk 7: *Controlstructuren*

Verwijzingen bij hoofdstuk 8: *Rijen*

[KNU69] D.E. Knuth, The Art of Computer Programming, vol. 3: Sorting and Searching. Addison-Wesley 1969.

[MEH81] K. Mehlhorn, Data Structures and Algorithms 1: Sorting and Searching, Springer EATCS Monographs on Theoretical Computer Science.

Verwijzingen bij hoofdstuk 9: *Bestanden*

[P3dictaat] Th.P. Van der Weide, *Van Datastructuur tot Informatiesysteem*, School voor Informatica, K.U. Nijmegen, 1994. Dictaat bij het P3 college, gat nader in op Informatiesystemen.

[HOM83] G.Hommel, S.Jähnichen, C.H.A.Koster, methodisches programmieren. De Gruyter Lehrbuch,1983. Beschrijft in hoofdstuk 11 de bestandsoperaties van Elan en geeft een uitgebreid voorbeeld van de toepassing daarvan.

Verwijzingen bij hoofdstuk 10: *Procedures*

[MEE77] L.G.L.T. Meertens Program text and program structure. `& in : Constructing quality software`, P.G. Hibbard/S.A. Schuman (eds.), IFIP WG2.1/WG2.4 conference, Novosibirsk, May 77 North Holland Publishers, 1978, pp. 271-283.

[KLE81] K. Kleine, S. Jähnichen, W. Koch, G. Hommel, Program construction with abstract notions in Elan. `& in : Proceedings 3d World Conference on Computers` in Education, Lausanne, North Holland Publ. Cy. 1981

Verwijzingen bij hoofdstuk 11: *Linguïstische toepassing*

[ALG68] A. van Wijngaarden, Revised Report on the Algorithmic Language ALGOL68

[HBC78] H. Brandt Corstius, Computer-taalkunde. Coutinho Muiderberg 1978.

[HBC83] Hugo Battus, Rekenen op taal¡ Querido, 1983.

[BAK84] J.J.A. van Bakel, Automatic Semantic Interpretation a computer model of understanding natural language, FORIS, Dordrecht, 1984.

[BAR60] Y. Bar-Hillel: *The present status of automatic translation of languages*, **in:** Advances in Computers I, 1960.

[KNU68] D.E. Knuth, *Semantics of context-free languages*, Mathematical systems theory 2, 1968.

[KOS71] C.H.A. Koster, *Affix Grammars*, **in:** J.E.L. Peck (Ed.), *ALGOL 68 implementation*, North Holland Publ. Cy., Amsterdam, 1971.

[KOS91] C.H.A. Koster, *Affix Grammars for Natural Languages*, **in:** H.Alblas, B. Melichar (Eds.), *Attribute Grammars, Applications and Systems*, Springer Lecture Notes in Computer Science 545, 1991.

[ASU86] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[WGO84] W.M. Waite, G. Goos, *Compiler Construction*, Springer-Verlag, 1984.

Verwijzingen bij hoofdstuk 12: *Recursie*

[BUR75] W.H. Burge, Recursive Programming Techniques. Addison-Wesley Publishing Company, 1975.

[DEL78] P. van Delft, J. Botermans, Spelen met Puzzels. De Bezige Bij, 1978.

[WIR76] N. Wirth, Algorithms + Data = Program. Prentice-Hall, 1976.

Verwijzingen bij hoofdstuk 13: *Computer grafiek*

[BRE65] J.E. Bresenham, *An Algorithm for Computer Control of a Digital Plotter*, IBM Systems Journal 4(1), 1965.

[SAS77] R. Salmon, M. Slater, *Computer Graphics: Systems and Concepts*, Addison-Wesley, 1977.

[FDF90] J.D.Foley, A.van Dam, S.K.Feiner, *Computer Graphics, Principles and Practice*, Addison-Wesley, 1990.

[PEA90] G. Peano, Sur une courbe, qui remplit toute une aire plaine. Math. Annln. Volume 36, pp 157-160, 1890.

Verwijzingen bij hoofdstuk 14: *Recursief sorteren*

[HOA62] C.A.R. Hoare, Quicksort. Computer Journal. Volume 5, Number 1, pp 10-15, 1962.

[EMB70] H.M. van Embden, Increasing the Efficiency of Quick Sort. Communications of the ACM. Volume 13, pp 563-567, 693-694, 1970.

[KNU73] D.E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley Publishing Company, 1973.

Verwijzingen bij hoofdstuk 15: *Backtrack algoritmen*

[GOB65] S.W. Golomb, L.D. Baumert, Backtrack Programming. Journal of the ACM. Volume 12, Number 4, pp 516-524, October 1965.

[KNU75] D.E. Knuth, Estimating the Efficiency of Backtrack Programs. Mathematics of Computation. Volume 29, pp 121-136, 1975.

[LAW66] E.L. Lawler, D.E. Wood, Branch-and-Bound Methods: A Survey. Operations Research. Volume 14, pp 699-719, 1966.

[SLA71] J.R. Slagle, Artificial Intelligence, The Heuristic Programming Approach. McGraw Hill Series in Systems Science, 1971.

[GAR66] M.Gardner, New mathematical diversions from Scientific American, Allen and Unwin, London 1969.

[FLE65] J.G. Fletcher, A Program to Solve the Pentomino Problem by the Recursive Use of Macros. Communications of the ACM. Volume 8, Number 10, October 1965.

Verwijzingen bij hoofdstuk 16: *Recursie naar iteratie*

[AHO74] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company, 1974.

[DIJ76] E.W. Dijkstra, A Discipline of Programming. Prentice-Hall, 1976.

[HOR78] E. Horowitz, S. Sahni, Fundamentals of Computer Algorithms. Pitman Publishers, 1978.

[BIR77] R.S. Bird, Notes on Recursion Elimination. Communications of the ACM. Volume 20, Number 6, June 1977.

[BAU76] F.L. Bauer, Programming as an Evolutionary Process. Proceedings of the 2nd International Conference on Software Engineering. IEEE, 1976.

[WIR71] N. Wirth, Program Development by Stepwise Refinement. Communications of the ACM. Volume 14, Number 4, April 1971.

[SND71] Y. Sundblad, The Ackermann function, a theoretical, computational and formula manipulative study. BIT vol. 11 (1971) pp. 107-119.

Index