SYSTEMATISCH LEREN PROGRAMMEREN

DEEL 2: BOTTOM-UP PROGRAMMERING

HET SYSTEMATISCH ONTWERPEN

VAN SEQUENTIËLE ALGORITMEN

OVER ABSTRACTE, POLYMORFE, RECURSIEVE

EN SAMENGESTELDE DATASTRUCTUREN;

HUN REALISERING IN ELAN

EN DE VERIFICATIE VAN HUN EIGENSCHAPPEN

C. H. A. KOSTER
SCHOOL VOOR INFORMATICA
KATHOLIEKE UNIVERSITEIT NIJMEGEN

Voorwoord

Bottom-Up programmering is als programmeermethode lang niet zo bekend en veelvuldig beschreven als de Top-Down programmering. Toch bestaat er wel degelijk een familie van Bottom-Up technieken die behoren tot het arsenaal van menig ervaren programmeur.

Dit is wellicht het eerste leerboek dat probeert, vanuit een uniform standpunt, het modulair ontwerpen en realiseren van samengestelde, abstracte, recursieve en polymorfe datastructuren en van de algoritmen daarover te verenigen tot een consistente programmeermethode, complementair aan de Top-Down programmeermethode.

Een woord over programmeerstijl en de richtingenstrijd die daarover de laatste jaren woedt.

Een algoritme beschrijft twee duale aspecten van een proces:

- het samenstellen van elementaire effecten tot een bedoeld effect
- het samenstellen van elementaire waarden tot een als resultaat bedoelde waarde

Benadrukt men uitsluitend het tweede aspect, onder uitsluiting van het eerste, dan ontstaat een zuiver *applicatieve* (*functionele*, *logische* of *relationele*) programmeerstijl, waarin door het ontbreken van (zij-)effecten een bewijs van correctheid sterk wordt vereenvoudigd, maar waarin effecten niet elegant zijn in te passen.

Benadrukt men het eerste onder verwaarlozing van het tweede, dan ontstaat de verguisde *imperatieve* programmeerstijl, met een aantal bekende problemen. Voor processen waarvan gebeurtenissen in de tijd de essentie uitmaken, bijvoorbeeld in concurrente en interactieve systemen, blijkt de hiertoe te rekenen *object-georiënteerde* programmmeerstijl echter zeer succesvol, met name ook in industriële toepassingen.

Geen van beide stijlen heeft mijns inziens de wijsheid in pacht, beide hebben belangrijke toepassingsgebieden, en tot dogma verheven zijn ze beide onbevredigend.

In de hier voorgestane *algoritmische* stijl van programmeren worden de applicatieve en imperatieve programmmeerstijl verenigd, waardoor de gelegenheid ontstaat de dualiteit daartussen zoveel mogelijk uit te buiten.

Dit boek poogt de lezer op alle genoemde programmeerstijlen in gelijke mate voor te bereiden, door het gemeenschappelijke daarin te benadrukken: het doelbewust gebruik van decompositie en abstractie om de conceptuele complexiteit te bolwerken, van recursie als zuinig denkmiddel, en van polymorfie ter verhoging van algemeenheid en herbruikbaarheid.

Over de doelgroep

Het boek richt zich tot informaticastudenten aan universiteiten en hogescholen, tot ervaren programmeurs die de grondslagen van hun vak willen begrijpen en tot al diegenen die als hulpmiddel bij hun vakgebied een solide kennis van het systematisch programmeren willen verwerven.

Het vooronderstelt een grondige beheersing van de Top-Down programmering en van de afleiding en implementering van recursieve algoritmen, zoals deze wordt geboden door [KOS84] en andere leerboeken. Beheersing van de programmeertaal ELAN is niet noodzakelijk, mits voldoende beheersing van een andere algoritmische programmeertaal (als PASCAL of MODULA2) aanwezig is, want een ervaren programmeur kan ELAN zeer snel aanleren uit de vele voorbeelden die worden gegeven.

Een essentieel onderdeel van het leren programmeren is het opdoen van vaardigheden. Daarom is ieder hoofdstuk voorzien van leerzame opgaven en behoort bij dit boek een TOOLKIT, een diskette met daarop de ELAN programmmeeromgeving en een groot aantal voorbeeldprogramma's, met behulp waarvan het behandelde aanschouwelijk kan worden gemaakt en waarop in oefeningen kan worden voortgebouwd.

Over dit boek

Dit is het tweede deel in de reeks SYSTEMATISCH LEREN PROGRAMMEREN. Behandelde het eerste deel [KOS84] algoritmen over eenvoudige datastructuren, ontwikkeld door Top-Down programmering met behulp van verfijningen, dit tweede deel behandelt het gebruik en de eigenschappen van recursieve, abstracte en polymorfe algoritmen en datastructuren, hun implementering volgens de Bottom-Up programmeermethode, en de verificatie van belangrijke eigenschappen zoals korrektheid en terminatie.

Een aansluitend derde deel, VAN DATASTRUCTUUR TOT INFORMATIESYSTEEM van de hand van Th. P. van de Weide [TvdW90] rondt deze reeks af met de formele specificatie en complexiteitsanalyse van datastructuren, gegevens-opslag en -terugwinning.

Dit deel is voortgekomen uit de dictaten bij opeenvolgende colleges ALGORITHMEN EN DATASTRUCTUREN, door de auteur gehouden aan de Technische Universiteit Berlijn en de Katholieke Universiteit Nijmegen. Een vorige versie is verschenen als leerboek SYSTEMATISCH PROGRAMMEREN IN ALGOL 68, DEEL2 bij Kluwer [KME79].

Naast de accentverschuivingen die een gevolg zijn van de andere keuze van onderwijstaal vertoont dit deel ook de resultaten van enige jaren langer onderwijs aan studenten: een systematischer en vollediger behandeling van recursieve en abstracte datatypen en een gestroomlijnder behandeling van correctheid.

Over de taal ELAN

ELAN is, ontegenzeggelijk, een ongebruikelijke programmeertaal,niet een wijd-verspreide taal met vele commerciele toepassingen, en in veel details afwijkend van meer gebruikelijke talen.

ELAN is in 1974 ontwikkeld aan de TU Berlijn (en in 1975 geadopteerd door de Westduitse Arbeitskreis Schulsprache) als een alternatief voor BASIC op de middelbare scholen.Het is een hulpmiddel voor het initiële onderwijs in systematisch programmeren, en dat verklaart veel van zijn eigenaardigheden.

De controlstructuren van ELAN zijn uiterst conventioneel gekozen: het zijn de Dijkstra controlstructuren tezamen met een `LEAVE`-constructie. Ook de datastructuren zijn uiterst beperkt gekozen: rijen van vaste lengte (net als in PASCAL) en structuren met benoemde velden (net als in ALGOL68). Een verwijzingsmechanisme (de `REF` van ALGOL68 of de pointersvan PASCAL) ontbreekt in ELAN, wat gezien de oorspronkelijke doelgroep begrijpelijk is.

De nadruk ligt bij ELAN dan ook niet op de constructiemiddelen maar op de abstractiemiddelen: het verfijningsmechanisme, generische procedures en operatoren, abstracte datatypen, polymorfie en het pakket-mechanisme. Deze taalmiddelen vormen een goede basis voor ieder programmeermethodiek die een gericht gebruik maakt van abstractie, zoals de Top-Down en Bottom-Up methoden.

Het meest in het oog lopende kenmerk van ELAN zijn de verfijningen. Als syntactische constructie in de taal zijn deze geboren uit de observatie dat studenten de klassieke wijze van verfijnen (als papier—en—potlood exercitie, voerend tot een programmatekst waaruit alle verfijningen weer zijn verdwenen) als een overbodige verplichte oefening beschouwen. Als ze worden gedwongen hun verfijningswerk in te leveren verfijnen ze nog liever achteraf.

Daarom introduceerden we bij het programmeeronderwijs in Nijmegen een pre-processor die ALGOL68 uitbreidde met verfijningen (zie [KZO78]) waarmee reeds veel betere ervaringen werden opgedaan. Bij die oplossing bleek echter nog te duidelijk dat de verfijningen een toevoeging achteraf zijn, die slecht past in de structuur van ALGOL68. Sinds 1987 gebruiken we daarom voor zowel hoofd- als bijvakstudenten ELAN, waarin de verfijningen goed passen, en de ELAN programmeeromgeving, die het verfijningsproces optimaal ondersteunt.

Een programma wordt daarbij niet opgevat als een lap tekst, waarin naar willekeur kan worden gewijzigd, maar als een verzameling naast elkaar staande, ieder voor zich kleine, onafhankelijke definities van typen, procedures, operatoren en verfijningen, eventueel samengevat in pakketten.

Hierdoor wordt ook het Bottom-Up ontwerp-, constructie- en realiseringsproces op nadrukkelijke wijze ondersteund.

De ervaring is dat op deze wijze studenten, met wijd uiteenlopende achtergrond en leeftijd, op diepgaande wijze de Top-Down en Bottom-Up ontwerpmethoden leren, die ze daarna ook met hele andere programmeertalen en formalismen (waaronder ook query talen en syntactische formalismen) kunnen toepassen.

Nijmegen, november 1990

C.H.A. Koster

Chapter 1

Programmaconstructie

Onder het programmeren van een rekenmachine verstaan we de systematische toepassing van de uitdrukkingsmiddelen van een programmeertaal om een gegeven probleem op te lossen. We zullen in dit hoofdstuk de basis leggen voor een vergelijking van twee programmeermethoden: de Top-Down programmering, die in het eerste deel van dit boek centraal stond, en een nog te beschrijven methode, de Bottom-Up programmering.

De ontwikkeling van een klein programma vindt plaats in een **levenscyclus** waarin ruwweg 4 fasen kunnen worden onderscheiden:

- **specificatie:** vastleggen wat het programma wel en niet moet doen,
- **ontwerp**: probleemanalyse; daaruit afleiden van de opzet van het programma,
- **constructie:** ontwerp uitwerken tot een uitvoerbaar programma, verificatie van belangrijke delen van het programma,
- **consolidatie:** testen, verbeteren en aanpassen tot het programma aan de specificatie voldoet en binnen de vrijheid die de specificatie laat een aangenaam gedrag vertoont.

Werkt de programmeur voor een opdrachtgever, dan komt hierbij nog de communicatie met de opdrachtgever en de aflevering en beschrijving van het product. Werkt er meer dan een persoon aan het programma, of is er sprake van een (deel van een) groot project, dan komen er nog veel meer technische en organisatorische punten naar voren, zodat een kwalitatief volkomen andere situatie ontstaat. Levenscyclus modellen voor de ontwikkeling van grote systemen zijn te vinden in boeken over **programmatuurkunde** ("software engineering"), zoals [FAI85], [PEP84] of [SOM89].

Wij zullen ons hier beperken tot de situatie van de solo-programmeur, die in zijn eentje de oplossing wil vinden voor een niet al te groot probleem dat hij in zijn geheel kan overzien.

In dit hoofdstuk zullen we, als het ware hardop denkend, een programma construeren. We gaan daarbij terwille van het voorbeeld nogal dogmatisch te werk. In het volgende hoofdstuk zullen we aan de hand daarvan de verschillen tussen Top-Down en Bottom-Up programmering behandelen. Daarna geven we een algemeen model aan voor de structuur van programma's.

1.1 Het verschil tussen twee data

We zullen in detail de constructie van een klein programma vervolgen en aan de hand daarvan enige conclusies over het programmeerproces trekken.

De probleemstelling zij als volgt: ontwikkel een programma om het verschil (uitgedrukt in dagen) tussen twee data te berekenen. In het programma zullen we voor de procedures, die immers op allerlei wijzen gebruikt kunnen worden, Engelse namen kiezen, terwijl we voor verfijningen (die alleen een locale rol spelen, binnen een procedure) Nederlandse namen nemen. Of deze conventie nuttig is is een tweede, laten we hem hier maar aanhouden. We ontwikkelen het programma in zorgvuldige stappen.

a) We stellen ons ten doel, een programma te schrijven met de volgende wortel:

```
verschil tussen twee data:
   lees twee data;
   druk hun verschil.
```

b) Hoe representeren we een datum? Een datum bestaat conceptueel uit drie getallen: het jaar, de maand en de dag. Een voor de hand liggende representatie is dan ook:

```
LET DATE = ROW 3 INT ;
{ jaar, maand en dag }
```

De **synoniem declaratie** die we hier toepassen geeft het synoniem `DATE` aan de type-declarer `ROW 3 INT`. Binnen het geldigheidsgebied van deze declaratie kunnen we met gelijke betekenis `DATE` en `ROW 3 INT` schrijven. We hebben hiermee een abstract datatype ingevoerd, waarvoor we nu ook operaties gaan invoeren.
Bovendien introduceren we drie bijbehorende synoniemen voor de indices op de rij

```
LET jaar=1, maand=2, dag=3;
```

c) Wat zijn de elementaire operaties en relaties op data? In de eerste plaats moeten we een datum kunnen lezen en schrijven. We lezen een datum (van het toetsenbord) als drie integers, waarbij we vooraf een "prompt" geven om duidelijk te maken hoe de datum moet worden ingetypt:

```
PROC get (DATE VAR d):
   line; put ("Jaar maand dag:");
   get (d[jaar]); get(d[maand]); get
(d[dag])
ENDPROC get;
```

d) We schrijven een datum (naar het beeldscherm) als drie kleine integers met deelstrepen ertussen, bijvoorbeeld 1989/12/15 :

```
PROC put (DATE d):
    put (text (d[jaar], 4)); put ("/");
    put (text (d[maand], 2)); put ("/");
    put (text (d[dag], 2))
ENDPROC put;
```

e) Bij vele operaties op data moeten we kunnen vaststellen of een datum in een schrikkeljaar ligt:

```
BOOL PROC is in leapyear (DATE d):
    INT CONST j :: d[jaar];
    (j MOD 4 = 0 AND j MOD 100 <> 0)
       OR j MOD 400 = 0
ENDPROC is in leapyear;
```

f) Daarmee kunnen we vaststellen, hoeveel dagen de maand van een datum heeft:

```
INT PROC length of month of (DATE d):
    SELECT d[maand] OF
    CASE 2: IF is in leapyear (d)
            THEN 29 ELSE 28 FI
    CASE 4, 6, 9, 11: 30
    OTHERWISE 31
    ENDSELECT
ENDPROC length of month of;
```

g) We kunnen ook vaststellen, hoeveel dagen het jaar van een datum telt:

```
INT PROC length of year of (DATE d):
    IF is in leapyear (d)
    THEN 366 ELSE 365 FI
ENDPROC length of year of;
```

h) Een ordeningsrelatie tussen data kunnen we als volgt definieren:

```
BOOL PROC before (DATE d1, d2):
    IF  d1[jaar] <> d2[jaar]
    THEN d1[jaar] < d2[jaar]
    ELIF d1[maand] <> d2[maand]
    THEN d1[maand] < d2[maand]
    ELSE d1[dag] < d2[dag]
    FI
ENDPROC before;
```

i) Op dit punt aangeland hebben we al die hulpalgoritmen gedefinieerd, die we van tevoren kunnen voorzien, zodat we terzake kunnen komen.
We willen een procedure `diff` declareren, die het verschil in dagen tussen twee data `d1` en `d2` berekent. Zullen we verlangen dat `d1` gegarandeerd voor `d2` komt, zodat het verschil nooit negatief is, of moeten we ook rekening houden met het geval dat `d1` na `d2` komt? We kiezen de zekerste weg:

```
INT PROC diff (DATE d1, d2):
    IF d2 voor d1
    THEN - diff (d2, d1)
    ELSE het positieve verschil tussen
d1 en d2
    FI.
```

j) De tweede datum valt voor de eerste:

```
d2 voor d1: before (d2, d1).
```

k) Ingeval `d1` en `d2` in hetzelfde jaar liggen is het eenvoudiger om het verschil te berekenen dan wanneer er jaargrenzen tussen liggen. Daarom onderscheiden we twee gevallen:

```
het positieve verschil tussen d1 en d2:
   IF in t zelfde jaar
   THEN verschil binnen jaar
   ELSE verschil buiten jaar
   FI.
```

l) De twee data vallen in hetzelfde jaar:

```
in t zelfde jaar: d1[jaar] = d2[jaar].
```

m) Als `d1` en `d2` binnen dezelfde maand liggen is het eenvoudiger om het verschil te berekenen dan wanneer er maandgrenzen tussen liggen. Weer onderscheiden we daarom twee gevallen:

```
verschil binnen jaar:
   IF in dezelfde maand
   THEN verschil binnen maand
   ELSE verschil buiten maand
   FI.
```

n) De twee data vallen in dezelfde maand:

```
in dezelfde maand: d1[maand] =
d2[maand].
```

o) Dan is het verschil direct te bepalen:

```
verschil binnen maand: d2[dag] -
d1[dag].
```

Even de proef op de som: het verschil tussen twee gelijke data is inderdaad nul; volgt `d2` op `d1` dan is het verschil inderdaad positief.

p) Liggen `d1` en `d2` in verschillende maanden van hetzelfde jaar dan kan het verschil recursief worden uitgedrukt:

```
verschil buiten maand:
   diff (d1, eind van de maand) +
   diff (begin volgende maand, d2) + 1.
```

Vergeet die 1 niet, namelijk het verschil tussen het eind van deze maand en de eerste van de volgende maand, zijnde een dag.

q) Het einde van de maand is een datum in die maand, wiens dag wordt bepaald door de lengte van de maand waarin de datum valt:

```
eind van de maand:
   [d1[jaar], d1[maand], length of month
of (d1)].
```

Deze rij-display denoteert een datum.

r) Het begin van de volgende maand is eenvoudig te bepalen:

```
begin volgende maand:
   [d1[jaar], d1[maand] + 1, 1].
```

Ga na, dat in de context van deze verfijning d1 niet in december ligt.

s) Als d1 en d2 in verschillende jaren liggen, dan is het verschil recursief te berekenen:

```
verschil buiten jaar:
   diff (d1, eind van het jaar) +
   diff (begin volgend jaar, d2) + 1.
```

t) Het einde van een jaar is 31 december van dat jaar:

```
eind van het jaar: [d1[jaar], 12, 31].
```

u) Het begin van een jaar is 1 januari van dat jaar:

```
begin volgend jaar: [d1[jaar] + 1, 1,
1].
```

Daarmee is tevens de procedure `diff` ten einde.

```
ENDPROC diff;
```

v) Na al deze voorbereidingen is het eenvoudig de laatste verfijningen uit te werken:

```
lees twee data:
   DATE VAR nu, straks;
   get (nu); get (straks).
```

w) en het afdrukken van het resultaat

```
druk hun verschil:
   line;
   put (nu); put (straks);
   put (diff (nu, straks)).
```

We ontdekken nu, dat we de hele procedure `length of year of` niet nodig gehad hebben. Dat komt doordat we de operaties op data hebben ontworpen terwijl we nog niet wisten wat we ermee gingen doen.

Is de algoritme `diff` eenmaal correct, dan is het mogelijk hem achteraf een stuk sneller te krijgen, o.a. door de (rechts)recursies te elimineren en door gebruik te maken van de procedure `length of year of`.

1.2 Structuur van het programma

Kijken we eens wat nader naar de stappen a) t/m w). In elke stap wordt een nadere precisering gegeven van de betekenis van een enkele naam in de vorm van een declaratie van een algoritme, object of type.

Iedere stap is de **definitie** van een naam. Zo'n definitie bevat op zijn beurt **toepassingen** van andere namen. Ieder voorkomen van een naam in het programma (buiten commentaren en textdenotaties) is hetzij een definitie, hetzij een toepassing.

We zeggen "A wordt door B direct gebruikt" als de definitie van B een toepassing van A bevat. De **wordt-gebruikt-door relatie** kunnen we grafisch weergeven.

Laat P_1, P_2, ..., P_n de namen zijn van de objecten, typen en algoritmen die in het programma P voorkomen. Onder de **applicatie graph** van P verstaan we een gerichte graph met n knopen, waarvan de i-de knoop gemarkeerd is met P_i, en die juist dan een gerichte pijl $P_i \rightarrow P_j$ bevat, wanneer P_i door P_j direct gebruikt wordt. De pijl betekent dus het ter beschikking staan van P_i voor P_j.

De applicatie-graph geeft de **programma structuur** weer, voor zover dit uitgekouwde begrip iets betekent. De applicatie-graph AG van het zojuist geconstrueerde programma geeft inderdaad belangrijke inzichten in zijn structuur. (Fig. 1.1).

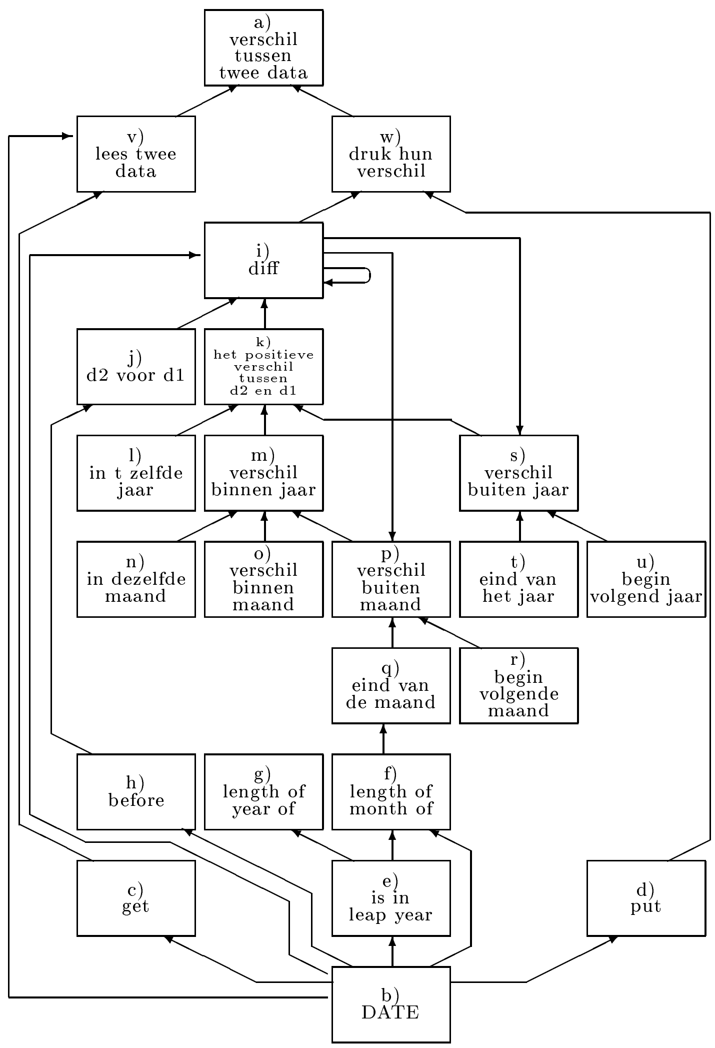


Figure 1.1: Applicatie-graph AG

In het diagram hebben we geprobeerd de knopen zo te arrangeren, dat de pijlen zoveel mogelijk naar boven wijzen, zodat het meest gedetailleerde niveau onder en het minst gedetailleerde (meest abstracte) niveau boven staat. De meeste pijlen wijzen inderdaad (recht of scheef) omhoog, met één uitzondering: de procedure `diff` is recursief, wat voert tot de neergaande pijlen i) → i), i) → p) en i) → s).

Het valt op dat j) tot en met u) allemaal mooi hierarchisch aan i) ondergeordend zijn, en met deze knoop tezamen als het ware conceptueel een blok vormen. Wij kunnen de structuur dus sterk vereenvoudigen als we de knopen i) tot en met u) tot één knoop i) samenvatten. Dit levert de deelgraph AG1, te zien in Fig. 1.2. De structuur van i) tot en met

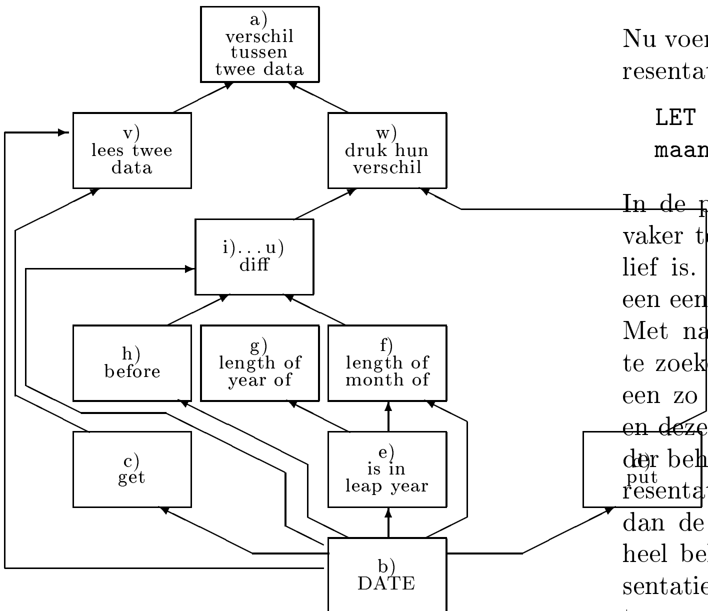


Figure 1.2: Applicatie-subgraph AG1

u) kunnen we nu apart aangeven als de deelgraph AG2, te zien in Fig. 1.3. Door de splitsing van de

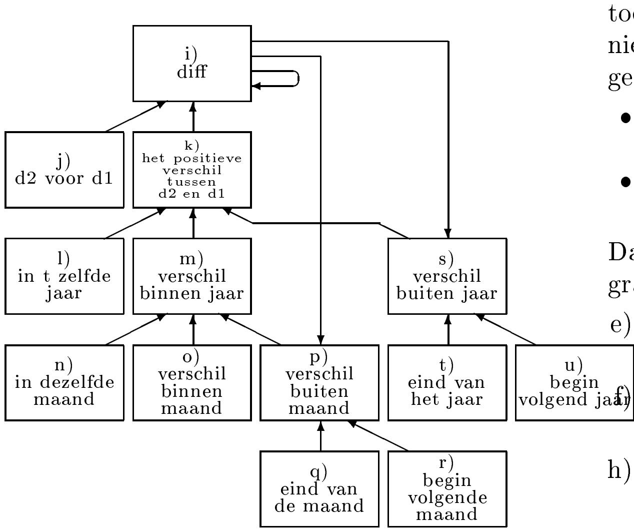


Figure 1.3: Applicatie-subgraph AG2

applicatie-graph AG in de delen AG1 en AG2 is hij veel aanschouwelijker en overzichtelijker geworden.

Als we de volgorde van de knopen in AG1 bekijken dan merken we op dat, met uitzondering van a), de knopen in alfabetische volgorde van onder naar boven staan. In AG2 staan ze juist in alfabetische volgorde van boven naar onder en van links naar rechts. Als we bedenken dat de alfabetische volgorde de tijdsvolgorde van ontwikkeling is, dan wordt duidelijk dat hier twee tegengestelde programmeermethoden zijn toegepast. Meer hierover in het volgende hoofdstuk.

1.3 Veranderen van de representatie

Nu voeren we een experiment uit. We willen de representatie van de data veranderen in

```
LET DATE = INT; { 10000 * jaar + 100 *
maand + dag }
```

In de praktijk van het programmeren moeten we vaker terugkomen op eerdere beslissingen dan ons lief is. Er kunnen echter goede redenen zijn om een eenmaal gekozen representatie later te herzien. Met name willen we de vrijheid behouden, eerst te zoeken naar een evident correcte oplossing met een zo eenvoudig mogelijke gegevensrepresentatie, en deze later naar believen efficiënter te maken onder behoud van de correctheid. Vóór de nieuwe representatie spreekt dat deze 3 maal compacter is dan de vorige, wat ruimte bespaart. Soms is dat heel belangrijk. Maar bovendien heeft deze representatie het voordeel dat de ordening van de integers overeenkomt met de ordening van de data (jj/mm/dd). We raken dus van de ingewikkelde en tijdrovende berekening in `before` af. Wat moeten we nu allemaal in het programma aanpassen?

Die plaatsen, waar een datum gedeclareerd, toegewezen of als parameter doorgegeven wordt zijn niet betroffen. Wel moeten we alle plaatsen wijzigen, waar

- toegegrepen wordt op de bestanddelen van een datum (**acces**), of
- een datum uit stukken wordt opgebouwd (**denotatie**).

Dat komt neer op de volgende plaatsen in ons programma:

e) tweede regel

```
INT CONST j :: d DIV 10000;
```

f) tweede regel

```
SELECT (d DIV 100) MOD 100 OF
```

h) de romp van de procedure kan enorm worden verkort, tot

```
d1 < d2
```

Hier maken we gebruik van de eigenschap van deze representatie dat de ordeningsrelatie op integers tevens als ordeningsrelatie op data dient.

l) `in t zelfde jaar:`
```
d1 DIV 10000 = d2 DIV 10000.
```
n) `in dezelfde maand:`
```
(d1DIV100) MOD 100 = (d2DIV100) MOD
100.
```
o) `verschil binnen maand: d2 - d1.`
Ook hier hebben we weer plezier van deze representatie.

q) `eind van de maand:`
```
100 * (d1 DIV 100) + length of month
of (d1).
```
r) `begin volgende maand:`
```
100 * (d1 DIV 100) + 101.
```
t) `eind van het jaar:`
```
10000 * (d1 DIV 10000) + 1231.
```
u) `begin volgend jaar:`
```
10000 * (d1 DIV 10000) + 10101.
```
v,w) De tekst blijft ongewijzigd, alleen worden de data anders gelezen en geschreven, nl. als geheel getal. Hiermee worden c) en d) overbodig.

We constateren dat we vreselijk veel details in het hele programma op penibele wijze moeten veranderen.

1.3.1 Representatie-afhankelijkheid

Al met al betalen we wel een hoge prijs voor de simpele wens, met een andere representatie te werken. We hebben hier namelijk de fundamentele fout begaan, op vele plaatsen in het programma expliciet van onze kennis over de representatie van de data gebruik te maken. De schade zou veel kleiner zijn uitgevallen, als we van meet af aan die delen van het programma, die van de representatie afhankelijk zijn, op één plaats geconcentreerd hadden, bijvoorbeeld met behulp van volgende procedures:

```
LET DATE = ROW 3 INT ;

INT PROC year of (DATE d):
   d[jaar]
ENDPROC;

INT PROC month of (DATE d):
   d[maand]
ENDPROC;

INT PROC day of (DATE d):
   d[dag]
ENDPROC;

DATE PROC date (INT j, m, d):
   [j, m, d]
ENDPROC;
```

Voor het toegrijpen op delen van data en de samenstelling van nieuwe data hadden we dan in e) tot en met u) uitsluitend deze procedures moeten gebruiken, bijvoorbeeld:

e) tweede regel
```
INT CONST j :: year of (d);
```
f) tweede regel
```
SELECT month of (d) OF
```
g) romp van de procedure
```
IF   year of (d1) <> year of (d2)
THEN year of (d1) < year of (d2)
ELIF month of (d1) <> month of (d2)
THEN month of (d1) < month of (d2)
ELSE day of (d1) < day of (d2)
FI
```
q) Voor het denoteren van een datum gebruiken we niet een display (wat immers vooronderstelt dat een datum als een rij is gerepresenteerd) maar de zojuist ingevoerde procedure `date`:
```
eind van de maand:
  date (year of (d1), month of (d1),
          length of month of (d1)).
```
r) `begin volgende maand:`
```
date (year of (d1), month of (d1) +
1, 1).
```
Zoals u ziet wordt het programma hierdoor ook een stuk begrijpelijker. De bedoelingen van de ontwerper worden abstracter uitgedrukt.

t) `eind van het jaar:`
```
date (year of d1, 12, 31).
```
u) `begin volgend jaar:`
```
date (year of (d1) + 1, 1, 1).
```
De representatie-afhankelijkheid is nu beperkt tot slechts enkele procedures. Het grootste deel van het programma is niet langer afhankelijk van de keuze, gemaakt voor de representatie van een datum.

1.3.2 Nog een andere representatie

Mochten we nu weer op het idee komen, een datum anders voor te stellen, en wel nog ietsje compacter, dan hoeven we slechts die representatie-afhankelijke procedures te wijzigen:

```
LET DATE = INT;
{(year*12 + month - 1)*31 + day - 1}

INT PROC year of  (DATE d):
   d DIV 372
ENDPROC;

INT PROC month of (DATE d):
   (d DIV 31) MOD 12 + 1
ENDPROC;

INT PROC day of (DATE d):
   d MOD 31 + 1
ENDPROC;

DATE PROC date (INT y, m, d):
   (y * 12 + m - 1) * 31 + d - 1
ENDPROC;
```

De hele rest van het programma blijft onaangeroerd en blijft zonder meer correct.

We kunnen de hier gedefinieerde **acces algoritmen**

```
INT PROC year of (DATE d):
INT PROC month of (DATE d):
INT PROC day of (DATE d):
```

en de **denotatie algoritme**

```
DATE PROC date (INT y, m, d):
```

als elementaire algoritmen van een nieuw, abstracter taalniveau opvatten, die geschikt zijn om data te manipuleren zonder enige kennis van hun representatie. Data zijn hiermee elementaire objecten geworden.

1.4 Conclusie

Het gedisciplineerd gebruik van acces- en denotatiealgoritmen (d.w.z. het abstraheren van de representatie) vereenvoudigt zowel het schrijven als het aanpassen van programma's. Hun gebruik betekent dat wij er vrijwillig van af zien, te profiteren van "intieme" kennis over de representatie van de data.

Bij grotere programmasystemen, die flexibel aangepast moeten kunnen worden aan wijzigende behoeften, is deze **aanpasbaarheid** ("adaptability") van levensbelang, vaak zelfs belangrijker dan de efficiëntie.

Het systematisch gebruik van accesalgoritmen in plaats van de onmiddellijke accessen die volgen uit de representatie voert tot een zekere overlast (een procedure-aanroep kost wat tijd, bijvoorbeeld enkele microseconden, en vele kleine aanroepen maken een grote): de prijs voor de aanpasbaarheid.

Het is echter veel belangrijker correcte aanpasbare programma's te kunnen schrijven, die systematisch zijn opgezet en duidelijk documenteren wat ermee bedoeld is, dan met behulp van allerlei domme streken de laatste microseconden uit een armzalige algoritme te persen.

1.5 Opgaven

1. (recursie verwijderen)
   Herschrijf de eerste versie van het programma zo, dat geen gebruik van recursie wordt gemaakt. Verbeter dan de efficiency verder door gebruikmaking van de procedure `length of year of`.
2. (aanpasbaarheid)
   Hoe moeten, in de laatste versie van het programma, de acces- en denotatie-algoritmen luiden voor de representatie *jaartal + nummer van de dag binnen het jaar*?

Chapter 2

Programmeermethodiek

In het vorige hoofdstuk hebben we, aan de hand van een eenvoudig voorbeeld, twee stijlen van programma-ontwikkeling ontmoet, die in zekere zin complementair zijn, maar ook (en meestal) naast elkaar worden gebruikt.

In dit hoofstuk zullen we deze twee methoden nader onderzoeken. Daarbij zullen we af en toe verwijzen naar het voorbeeldprogramma uit het vorige hoofdstuk.

2.1 Top-Down programmering

De methode van programma-ontwikkeling, die wij in het eerste deel vrijwel steeds gebruikt hebben, bestaat erin, door *verfijning* van een aanvankelijke grove formulering te komen tot een gedetailleerde oplossing van het probleem.

Iedere stap bestaat uit de declaratie van een (abstracte) algoritme, type of object in termen van verdere algoritmen, typen en objecten, met elkaar verbonden door middel van controlstructuren en datastructuren.

> Bij de **top-down programmering** is de volgorde waarin de definities gegeven worden zodanig, dat nooit iets gedefinieerd wordt dat niet daarvoor (tenminste éénmaal) is geappliceerd.

In het vorige voorbeeld gebeurden dus de stappen i) tot en met u) Top-Down (TD).

Een consequentie van deze volgorde van werken is, dat het programmeren begint met weinig gedetailleerde definities op een hoog niveau van abstractie en gaandeweg voortschrijdt van abstract naar concreet, tot op het niveau van detail van de concrete programmeertaal. Daarbij zijn er nog vele vrijheidsgraden, zodat verschillende TD stijlen aangehouden kunnen worden:

- eerst in de breedte, één "laag van abstractie" tegelijk afwerken
- eerst in de diepte, een tak tot het eind vervolgen voor je aan een nieuwe tak begint
- allerlei mengsels daarvan.

Het programmeren krijgt hiermee het karakter van een systematische decompositie van een abstracte probleemoplossing.

Overigens is de TD methode veel gemakkelijker toe te passen op algoritmen (door middel van verfijningen en procedures) dan op objecten en typen (door middel van type-declaraties). Wat dat betreft is de situatie niet symmetrisch. Objecten en vooral typen ontstaan gemakkelijker in een Bottom-Up programmeerproces.

2.1.1 Criteria voor TD programmering

Aan welke eigenschappen moeten de definities waaruit een Top-Down programma is samengesteld voldoen?

- Een definitie moet klein en overzichtelijk zijn. Vermijd nesting. Vertel *wat* er moet gebeuren, liever dan *hoe* het moet gebeuren.
- Een definitie moet op een homogeen niveau van abstractie zijn, niet plotseling allerlei details vermelden die op dit niveau niet te vatten zijn.
- Een definitie moet op zijn abstractie-niveau compleet zijn.
- Verbreek de abstractie zo langzaam als mogelijk, van alle voortijdige vastleggingen op details krijg je spijt.
- Verbreek de abstractie zo snel als nodig. Op bepaalde momenten moeten keuzen gedaan worden. Als dat niet langer vermijdbaar is, hak de knoop dan dapper door - maar vermorzel hem niet.

De motivering van de Top-Down programmering is het verlangen zich zo laat mogelijk vast te leggen. Iedere beslissing wordt zo lang mogelijk uitgesteld, in ieder geval zo lang als nodig is om hem met vertrouwen te kunnen treffen.

In 1.1 hebben we de nadelige consequenties gezien van een te vroege beslissing over de representatie van de data:

- De hele algoritme wordt in onnodige mate afhankelijk van de gegevensrepresentatie, met als gevolg grote moeilijkheden bij eventueel noodzakelijke aanpassingen.
- Er bestaat gevaar voor een verkeerde keuze van de representatie, omdat vaak pas uit het gebruik van gegevens de eisen duidelijk worden, waaraan hun representatie moet voldoen.

Bij een te vroege concretisering van de structuur van een algoritme ontstaan analoge problemen.

Een nadeel van de Top-Down methode is, dat de uitwerkingen van onderweg genomen verkeerde beslissingen vaak pas later herkend worden aan het feit, dat zich bepaalde algoritmen e.d. niet goed verder laten verfijnen. Het is mogelijk "aan de machine voorbij te programmeren".

Ook is de Top-Down programmeermethode er niet op gericht, herbruikbare bouwstenen voor het programma te herkennen. Het gevaar bestaat dat optredende gelijkvormige deelproblemen alle hun eigen individuele verfijningen krijgen, zodat het programma verwordt tot een opgeblazen pudding van verfijningen, waarvan vele vrijwel hetzelfde doen maar onder een andere naam.

Ten slotte wordt er bij de Top-Down programmering op de efficiëntie van het resulterende programma vaak te weinig nadruk gelegd. zodat soms een product ontstaat dat het wel doet, maar kruipt als een slak.

2.1.2 Stijlfouten bij TD programmering

Vooral beginners, maar ook ervaren programmeurs met een sinds jaren vastgeroeste stijl, kost het veel moeite een zinnig gebruik te maken van TD uitdrukkingsmiddelen als verfijningen. De last, voortdurend abstracte algoritmen te verzinnen valt hun zwaar. Het is dan ook bijzonder moeilijk iemand het scheppen van ideeeën uit te leggen en bij te brengen. Het is veel gemakkelijker slechts te mikken op een programma "dat 't doet" en je geen zorgen te maken over de ontstaanswijze. Verfijningen zijn echt bedoeld als hulpmiddel bij het bewust ontwikkelen van een stijl van systematisch programmeren, niet als hindernis.

Het gebruik van verfijningen garandeert op zich geen goed programma, voor ervaren programmeurs met een goed doordachte stijl is zo'n hulpmiddel ook niet meer noodzakelijk. Met deze slagen om de arm kunnen we TD programmering met verfijningen toch beschouwen als een uitstekende toegang tot een systematische stijl van programmeren.

Bij het verwerven van stijl is veel lering te trekken uit slechte voorbeelden. Typische **top-down stijlfouten**, fouten die steeds weer gemaakt worden bij het TD verfijnen zijn:

- De methode wordt tot maniertje. Onderdelen als declaraties en voorbereiding en de verwerking van individuele elementen krijgen steeds platvloerse namen, zoals in het universele schabloon:

```
programma:
  declaraties;
  voorbereiding;
  WHILE niet klaar
  REP verwerk een element
  ENDREP;
  print resultaten.
```

  Het is niet fout, maar volkomen nietszeggend, in tegenstelling tot:

```
sommeer de rij:
  de som van nul elementen is nul;
  nog geen element gehad;
  WHILE nog niet alle elementen gehad
  REP tel element bij som
  ENDREP;
  print som.
```

  Merk op, dat de eerste beide verfijningnamen de vorm hebben van (verbale) uitspraken ("asserties"). Zij kunnen heel wel worden geconcretiseerd tot een of meer geïnitialiseerde declaraties. Geïnitialiseerde declaraties dienen ertoe, een bepaalde toestand te bereiken, die meestal gemakkelijk benoemd kan worden. De preciese manier waarop die toestand wordt bereikt kan dan met vrucht aan een verfijning worden overgelaten.

- De verfijningsnamen worden veel te concreet (en daarom ook veel te lang) gekozen:

```
sommeer de rij:
  declareer gehele variabelen voor
     som en index en zet ze op nul;
  WHILE index niet groter dan bovengrens
      van de te sommeren rij
  REP tel het element met index index op
      bij de som
  ENDREP;
  print de waarde van som.
```

  Zich kort en duidelijk uitdrukken is iets, dat moet worden geleerd.

- Verschillende niveaus van abstractie worden door elkaar gehaspeld, meestal door overhaast doorbreken van de abstractie, bijvoorbeeld in:

```
schuif de hobbel naar achteren:
  WHILE nog niet alle paren gehad
  REP IF twee buren voldoen niet aan
ordening
     THEN INT CONST hulp :: rij [l];
          rij [l] := rij [k];
          rij [k] := hulp
     FI
  ENDREP.
```

  in plaats van:

```
schuif de hobbel naar achteren:
  WHILE nog niet alle paren gehad
  REP IF   dat paar vormt een hobbel
     THEN strijk die glad
     FI
  ENDREP.
dat paar vormt een hobbel:
  rij [l] < rij [l+1].
strijk die glad:
  INT CONST hulp :: rij [l];
  rij [l] := rij [k];
  rij [k] := hulp.
```

- De verfijningen zijn aan elkaar geregen en ieder voor zich niet compleet, bij voorbeeld:

```
programma:
  declareer;
  doe de rest.
```

```
doe de rest:
  doe iets;
  doe wat er over is.
doe wat er over is:
  enzovoort.
```

De eerste verfijning is allesbehalve een complete formulering. Arbitrair is deze in stukken gehakt. De verdere verfijningen hebben nietszeggende namen, maar dat kan ook niet anders omdat ze niets afgeronds voorstellen.

Zo weergegeven ziet het er ridicuul en onwaarschijnlijk uit, maar bij practica leveren studenten steeds weer dit soort vervolgverhalen.

2.2 Bottom-Up programmering

Onder *Bottom-Up* programmering (BU) verstaan we een programmeermethode, waarbij het gewenste stap voor stap gebouwd wordt op het beschikbare, analoog aan de systematische opbouw van mathematische theorie met behulp van axioma's en theorema's, waarbij ieder nieuw theorema herleid wordt tot axioma's en reeds bewezen theorema's.

> Bij de **bottom-up programmering** is de volgorde waarin de definities gegeven worden zodanig gekozen, dat nooit iets toegepast wordt dat niet daarvóór is gedefinieerd.

Als uitgangspunt dienen de concrete algoritmen, typen en objecten, in termen waarvan de meer abstracte worden gedefinieerd. Strikt genomen sluit deze volgorde recursie uit.

Om succesvol BU te programmeren moet men zich van te voren zeer zorgvuldig voor de geest halen, welke algoritmen, typen en objecten voor de oplossing van het probleem noodzakelijk zijn. Men bouwt zich hulpstuk op hulpstuk, tot de oplossing is gevonden.

Daarbij zal de BU-programmering bij juist gebruik niet zomaar de omkering in de tijd zijn van de TD programmering. Het is namelijk menselijkerwijze onmogelijk een TD ontwerp feilloos achterste voren te doorlopen, als een achteruit lopende slaapwandelaar.

Het behoeft geen betoog dat BU programmering lijdt onder drie gevaren: (**bottom-up stijlfouten**):

- Het gevaar, aan het probleem voorbij te programmeren, doordat het samenstelsel van definities geen juiste basis voor de probleemoplossing biedt.
- Het gevaar, door de bomen het bos niet meer te zien. Door het voortdurende gesleutel aan allerlei programmeertechnische details wordt het programma nooit consistent en compleet.
- Het gevaar abstracties te definieren, die later geen toepassing blijken te hebben. Uit hang naar compleetheid en volmaaktheid kan er veel te veel voorwerk worden geleverd.

Door de tendens tot constructie (middels controlstructuren en datastructuren) in plaats van abstractie zijn BU programma's in het algemeen kort en ingewikkeld. BU programmeurs neigen ertoe, de efficiëntiemogelijkheden tot op het bot te benutten.

De BU methode komt vooral tot zijn recht bij de definitie van objecten en typen met hun bijbehorende basisalgoritmen, waar de te overbruggen abstractie-verschillen gering en de bedoelingen algemeen en duidelijk zijn, in combinatie met TD programmering van de meer gespecialiseerde rest van het programma. In ieder geval verdient het geen aanbeveling, hele programma's BU te ontwikkelen.

Later in dit hoofdstuk zullen we een "getemde" versie van BU programmering introduceren, de laagsgewijze programmering.

In volgende hoofdstukken zullen we nog enige taalmechanismen van ELAN introduceren, die vooral bij BU programmering een rol spelen.

2.3 Het programma als abstracte machine

In dit hoofdstuk geven we een model voor programma's dat als basis kan dienen voor een model van het programmeerproces.

2.3.1 Abstractie-niveaus

Bij het TD programmeren blijft nog zeer veel vrijheid in de volgorde van werken. Zo kan, in ons voorbeeld, op n) zowel o) als p) volgen, maar q) en r) kunnen pas na p) komen.

Een mogelijkheid, van deze vrijheidsgraden gebruik te maken is, het programma op te delen in niveaus van gelijke graad aan detail (**abstractie niveaus**). Zo een niveau is geen zichtbaar, concreet onderdeel van het programma, maar is in de eerste plaats een idee in het brein van de programmeur, dat hij als leidraad kan gebruiken bij de ontwikkeling van zijn programma.

Zo een niveau is in het programma terug te vinden als een aantal definities van algoritmen, typen en objecten, die samen dit abstractie-niveau vormen. In een plaatje kan men zich de laagsgewijze programma-ontwikkeling zo voorstellen: Als we van recursie even afzien, worden bij de TD programmering steeds de algoritmen, typen en objecten uit een bepaalde laag gedefinieerd in termen van de algoritmen, typen en objecten van de laag daaronder, enzovoort.

De definities, ingedeeld in lagen, vormen een *hiërarchie* met betrekking tot de relatie "wordt gebruikt door". De onderste laag sluit aan bij de concrete programmeertaal zelf, d.w.z. gebruikt slechts

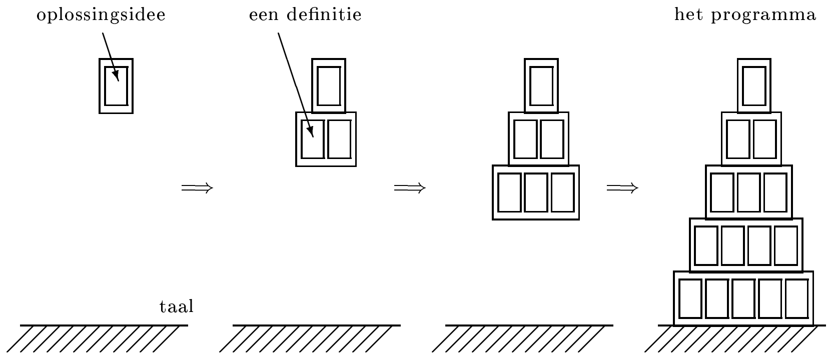


Figure 2.1: Top-Down ontwikkeling

algoritmen, typen en objecten, die daarin voorhanden zijn. Geen enkele laag bevat een applicatie van iets dat pas in een hogere laag gedefinieerd wordt.

Uit de applicatie-graph AG2 kan men de volgende hiërarchie afleiden: Ieder van de lagen wordt

laag 0	i				
laag 1	j	k			
laag 2	l	m	s	t	u
laag 3	n	o	p	q	r

Figure 2.2: Hiërarchie binnen `diff`

gekarakteriseerd door de aard van de objecten waar hij zich mee bezighoudt:

- laag 1 houdt zich bezig met data
- laag 2 houdt zich bezig met jaren
- laag 3 houdt zich bezig met maanden.

In vele gevallen blijkt een indeling in lagen op grond van het type van de behandelde objecten het meest voor de hand liggend en aanschouwelijk. Abstracte datatypen (zie verder) worden dan ook meestal Bottom-Up ontworpen.

2.3.2 Abstracte programma' s, talen en machines

Een programma bestaat conceptueel uit een aantal abstractielagen, waarbij de onderste laag aansluit bij de op de gegeven machine beschikbare programmeertaal. We zullen een programmeertaal *concreet*

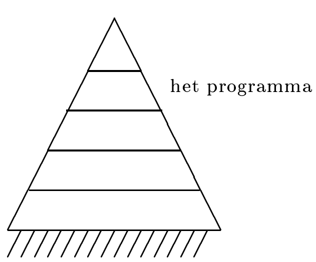


Figure 2.3: Programma als abstracte machine

noemen als deze uitvoerbaar is op onze machine, d.w.z. als er een concrete machine beschikbaar is die programma's in deze taal uitvoert. In slechts weinig gevallen, namelijk voor machine-gerichte talen, zijn er concrete machines, die de programma's direct uitvoeren; voor andere talen moet daarom eerst met behulp van bepaalde technieken (vertaling of vertolking) een abstracte machine gerealiseerd worden. Zo kan men zich de uitvoering van een ELAN programma op bijvoorbeeld een werkstation onder het bedrijfssysteem UNIX als volgt voorstellen: Bij nadere beschouwing kan men echter nog vele

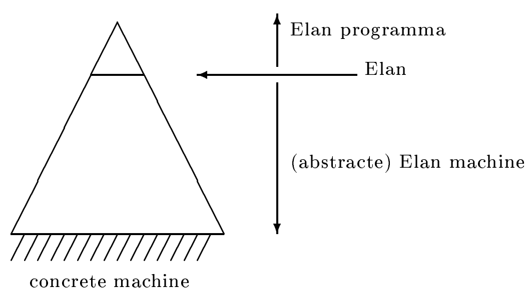


Figure 2.4: Elan programma op computer

verdere niveaus onderscheiden, die een hiërarchie van abstracte machines vormen, bijvoorbeeld

- ELAN programma
- Abstracte ELAN machine
- UNIX
- Kale computer

Bij nog scherper toezien onderscheidt men binnen ieder van deze lagen nog verdere lagen!

De begrippen "abstract" en "concreet" moeten blijkbaar steeds relatief begrepen worden. ELAN is voor de compilerschrijver abstract, voor de compilergebruiker concreet.

Het **snijvlak** ("interface") tussen twee lagen zullen we als een **abstracte taal** opvatten; het wordt doorsneden door de applicatie-pijlen voor een aantal algoritmen, typen en objecten die tezamen als een speciale programmeertaal kunnen worden opgevat, waarin de bovenste laag wordt geschreven.

De concrete machine plus de daaropliggende lagen onder een bepaald snijvlak noemen we een **abstracte machine** en de lagen daarboven een **abstract programma** (plaatje 2.5). De begrippen

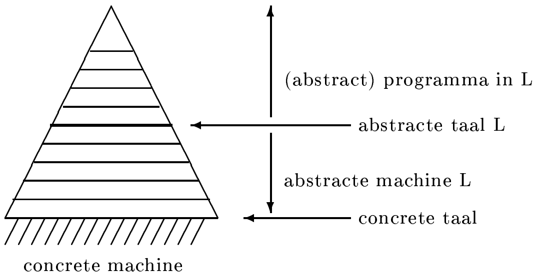


Figure 2.5: Meer-laagse abstracte machine

"machine" en "taal" zijn hier duaal, twee complementaire manieren om dezelfde wereld te bekijken. De taal definieert een abstracte machine en de machine realiseert een taal.

Een heel bijzondere taal vindt men boven de bovenste laag: deze bestaat uit een enkele algoritme, die bedoeld is om een bepaald probleem op te lossen. De programmeur heeft daartoe een gespecialiseerde abstracte machine gebouwd in de vorm van een programma in een uitvoerbare taal. De combinatie van de concrete machine met alle daaropliggende lagen vormt een abstracte machine, die de gestelde opgave oplost.

Systematisch programmeren kan men opvatten als het construeren, met potlood en papier, van een gespecialiseerde abstracte machine, of als u dat liever hebt, het realiseren van een abstracte taal.

In dit beeld correspondeert TD programmering met het van boven naar beneden de pyramide bouwen door toenemende concretisering, BU programmering gaat van onder naar boven door gestaag construeren met toenemende abstractie.

De keuze, welke algoritmen, typen en objecten in een bepaalde laag gedefinieerd worden is, net als de keuze van de niveaus, een kwestie van nadenken, inzicht en ervaring, en wordt slechts gemotiveerd door de ondersteuning die deze keuze biedt aan het denken.

De ervaring wijst uit dat het geen zin heeft, al bij een gering verschil in abstractie van verschillende lagen te spreken: het denken in termen van enkele abstractie-niveaus is al moeilijk genoeg. Binnen een laag kan men dan ook best een "abstractie gradient" toelaten. Ook kan men, binnen een niveau, recursie toelaten, hoewel daarmee strikt genomen de hiërarchie doorbroken is.

De BU-uitbreiding van een abstracte machine in verticale richting is een abstractie-proces, de uitbreiding van een abstracte machine op een en hetzelfde abstractie-niveau noemen we **extensie**.

Extensie is een typisch proces behorend bij de laagsgewijze BU programmering. Met name kan men een bestaande abstracte machine extenderen om hem voor een ruimer probleemgebied te kunnen gebruiken. Zo kunnen we, in ons voorbeeld, naast de procedure `diff` ook definiëren:

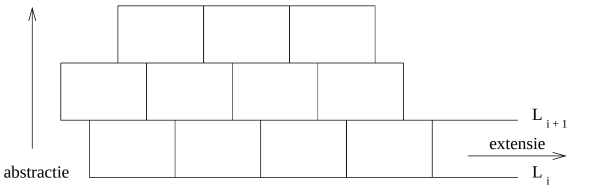


Figure 2.6: Abstractie en extensie

```
DATE PROC delay (DATE d, INT days):
  {de datum die days dagen na d ligt,}
  {d.w.z. delay(d1, diff(d1, d2)) = d2}
```

om op deze wijze een completer pakket algoritmen te hebben ter manipulatie van data.

2.4 Laagsgewijze programmering

In het licht van deze overwegingen schijnt de indeling in niveaus, gegeven in Fig. 1.1, reeds veel te gedetailleerd. Een beter te verdedigen indeling in drie lagen luidt:

- (*representatie*) Definitie van `DATE` en zijn acces- en denotatie-algoritmen. Hier kan men alles samenvatten wat van de representatie afhangt.
- (*functie*) Definitie van `get`, `put`, `is leapyear`, `length of month of`, `before` en `diff`. Samen zijn dit alle elementaire algoritmen voor de manipulatie van data.
- (*applicatie*) Het hoofdprogramma:

```
DATE VAR nu, straks;
get (nu); get (straks);
line; put (nu); put (straks);
put (diff (nu, straks)).
```

Deze indeling maakt duidelijk dat het hoofdprogramma als het ware in een uitbreiding van ELAN met nieuwe elementaire typen en algoritmen geschreven is.

Een dergelijke structuur in lagen kan men zien als een herhaalde opdeling in twee lagen (respectievelijk definitie en applicatie), die leidt tot een methode die wij de **laagsgewijze programmering** zullen noemen:

1. analyseer het probleem en specificeer een probleemgericht taalniveau
2. realiseer dit taalniveau (TD of BU, eventueel in meer lagen)
3. programmeer op dit taalniveau de oplossing van het probleem (TD of BU, eventueel in meer lagen).

Het zwaartepunt ligt hierbij op een zorgvuldige probleemanalyse, waarvoor veel ervaring in het probleemoplossen en een goed overzicht over het probleemgebied noodzakelijk zijn. Een moeilijkheid ligt daarbij in de keuze en de vastlegging van het interface tussen de twee lagen (de **interface specificatie**). Wij zullen voorlopig slechts informele specificaties van interfaces geven, in de vorm van een

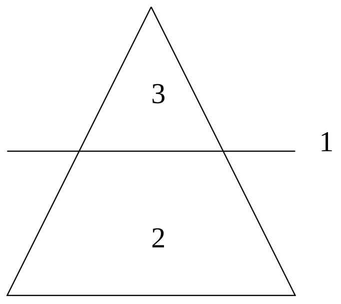


Figure 2.7: Laagsgewijze programmering

lijst namen van algoritmen, typen en objecten met een verbale verklaring, of iets formeler: in de vorm van een of meer NTO's.

Stap 1 leidt tot een specificatie. De stappen 2) en 3) in de laagsgewijze programmering kan men nog in verschillende volgorden uitvoeren:

- eerst 2) dan 3)
  Als men een ervaren BU programmeur is en de te realiseren objecten, typen en algoritmen goed kent, bijvoorbeeld bij de realisering van het domein van de vectoren en vectoroperaties, waarvan de eigenschappen uit de wiskunde voldoende bekend zijn.
- eerst 3) dan 2)
  Als men een ervaren TD programmeur is, die in staat is het interface van te voren te specificeren en dan daarop voort te programmeren in het volste vertrouwen dat hij dit interface achteraf realiseren kan. Bij het programmeren van de top ontstaat dan weer een beter begrip voor de eisen, te stellen aan de realisering, bijvoorbeeld wat betreft datarepresentatie.
- 2) en 3) in de tijd vermengd
  Een te gevaarlijk procédé voor normale stervelingen. Een bewuste scheiding van beide bezigheden voorkomt ongelukken.
- 2) en 3) parallel door verschillende programmeurs
  Dit is slechts bij zeer goede samenwerking mogelijk onder voorwaarde dat de programmeurs het eerst gezamenlijk eens zijn geworden over de precieze specificatie. Optimale verdeling over meer programmeurs van de werkzaamheden aan een groot systeem (modularisering, interfacespecificatie, organisatorische vragen) is een onderwerp uit de programmatuurkunde.
- enkel stap 3), omdat stap 2) al gedaan is
  Als er al een geschikte probleemgerichte taal bestaat, dan moet men ook niet met eigen huisbaksels aankomen, maar deze taal gebruiken. Er bestaat niet zoiets als een ideale taal voor alle toepassingen. Voor bijzondere toepassingsgebieden (zoals bijvoorbeeld de statistische verwerking van tijdrijen en andere metingen, continue en discrete simulaties, symbool-manipulatie en allerlei gespecialiseerde vormen van administratie) zijn er reeds speciale programmeertalen of toepassingspakketten gedefinieerd. Gebruik die dan ook. Het gemak dient de mens, en beschermt hem tegen fouten door misplaatste ijver.

Er zijn hele groepen wetenschapsmensen, die bibliotheken van programma's en algoritmen aanleggen die typisch zijn voor bepaalde gebieden, bijvoorbeeld de algoritmenverzameling van het britse National Physics Laboratory op het gebied van numerieke wiskunde, lineaire algebra en grootschalig rekenen, bekend als de NAG-library.

2.5 Conclusie

Wij hebben twee methoden van systematisch programmeren leren kennen, de TD en BU programmering, ieder met hun eigen voor- en nadelen, en hebben deze gerelateerd aan een model van programma's als abstracte machines. We hebben een synthese daaruit geformuleerd, de laagsgewijze programmering, die ook voor het schrijven van hele grote programma's geschikt is.

De enige programmeerstijl die met zekerheid fout is is het "meteen naar de machine", waarbij het denken vervangen wordt door coderen. Probeer een stijl verwerven waarin TD programmering en BU programmering afwisselend en in combinatie ieder hun rol spelen.

Daarbij doe je er goed aan, het programma globaal gezien als een abstracte machine op te zetten, met weldoordachte interfaces tussen de opeenvolgende lagen.

Om een inzicht in het probleem te krijgen helpt het vaak, eerst TD te beginnen, tot je merkt dat je te veel fouten begint te maken. Dat is een goede reden om nu het een en ander aan primitieve algoritmen, typen en objecten BU te bedenken, waarbij je een dankbaar gebruik maakt van het inzicht dat je TD verworven hebt. Komt je BU niet verder dan helpt het misschien opnieuw TD te beginnen.

Een alternatieve methode, die af en toe toegepast wordt, zou men **zijdelingse programmering** kunnen noemen. Als je van plan bent een grote abstracte machine te bouwen, is het nuttig, eerst een smalspoorversie op alle niveaus te construeren, zodat tenminste een testbare versie verkregen is (**rapid prototyping**). Deze wordt vervolgens met behulp van extensie volledig gemaakt.

Volgens deze methode worden met name zware programma's als bedrijfssystemen of computergesteunde ontwerpsystemen succesvol geïmplementeerd. Het voordeel is dat in een vroeg stadium ten minste een smalspoorversie van het product loopt. Het nadeel is dat je je voortdurend met onvolledige producten moet behelpen, wat op den duur leidt tot frustratie van implementator en gebruiker. Het blijft eeuwig knutselen.

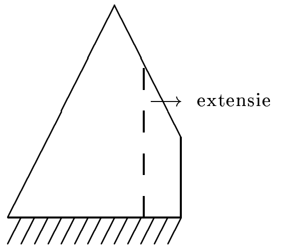


Figure 2.8: Zijdelingse programmering

In de volgende hoofdstukken komen nog veel meer programmeermethoden aan bod: We behandelen abstracte datatypen en modulaire programmering als verdieping van de Bottom-Up programmering, maar ook formelere programmeermethoden zoals de assertionele programmering en de transformationele programmering. In een groot voorbeeld demonstreren we de syntax-gestuurde programmering.

Alleen al de veelvuldigheid aan methoden (en er bestaan er nog veel meer) wijst erop dat geen van deze methoden een panacee vormt. Programmeren blijft een lastige (en daarom leuke) bezigheid, waarbij uiteindelijk steeds kennis en ervaring de doorslag geeft.

Bovendien: wat bij de een "methode" heet, noemt de ander "chaos". Zoals Augustinus het uitdrukte: "regelmatig beschonken is ook regelmaat".

Systematisch te werk gaan moet ook geen doel op zich worden.

2.6 Opgaven

1. Geef een NTO voor het type `DATE` uit dit hoofdstuk.
2. Ontwerp een NTO voor de typen `MATRIX` en `VECTOR`, met de voor de hand liggende operaties uit de lineaire algebra.

Chapter 3

Correctheid (1)

De geschiedenis van het programmeren is die van de programmeerfouten. Telkens weer begint iemand vol goede moed aan een eenvoudige programmeeropgave, waar hij zijn hand niet voor omdraait en telkens weer blijkt het aantal fouten dat hij maakt evenredig toe te nemen met zijn zelfvertrouwen.

In het bedrijfsleven is het al niet beter dan bij een practicum voor studenten: in programmatuur van levensbelang worden na jaren nog fouten gevonden; het verwijderen van fouten nadat een programma is opgeleverd wordt vergoelijkend "maintenance" genoemd (**onderhoud**) — alsof dergelijke fouten het natuurlijk resultaat zijn van onvermijdelijke slijtage!

In dit hoofdstuk zullen we eerst nagaan hoe fouten kunnen worden voorkomen. We beperken ons weer tot de situatie van de solo-programmeur, die van een probleem dat hij begrijpt en in zijn geheel kan overzien een oplossing realiseert in de vorm van een programma. Daarna voeren we een begrippenkader rond correctheid in als voorbereiding op een formelere behandeling van dit onderwerp in het volgende hoofdstuk.

3.1 Soorten fouten

De methode van de stapsgewijze verfijning maakt het mogelijk, systematisch te werk te gaan bij het afleiden van een algoritmische oplossing voor een gegeven probleem. Deze en andere methoden om systematisch programma's te construeren garanderen overigens geenszins dat voor elk probleem een oplossing wordt gevonden noch dat de gevonden oplossing werkelijk het gestelde probleem oplost.

Tijdens de levenscyclus van een programma kunnen op allerlei plaatsen fouten worden gemaakt.
Bij het *ontwerp* van een oplossing:

- begripsfouten (het niet volledig begrijpen van het probleem). Andermans probleem begrijpen is nu eenmaal erg moeilijk!
- denkfouten (het onbedoeld oplossen van een ander probleem)

Bij het *construeren* van het algoritme:

- denkfouten (het verkeerd opsplitsen in deelproblemen, fout ontwerp of foutieve constructie van een deelprobleem)
- onzorgvuldigheden (het na correct denken verkeerd opschrijven van het algoritme of het vergeten van zelfopgelegde regels of conventies)
- lexicale fouten (tik- en spelfouten)
- syntactische fouten (gebrek aan kennis van de notatie van de programmeertaal)
- semantische fouten (gebrek aan kennis van de betekenis van de programmeertaal of van de relevante eigenschappen van de machine)

Bij het *consolideren* van het programma probeer je het van alle fouten en onvolkomenheden te bevrijden, wat kan betekenen dat je terug moet naar een van de vorige fasen.

Bij het *uitvoeren* van programma's kunnen ook nog fouten optreden:

- bedieningsfouten (foute uitvoeringsaanwijzigingen of foute invoergegevens)
- hardware fouten (storingen in de machine). Vooral beginnende programmeurs zijn geneigd alle fouten in eerste instantie aan de machine toe te schrijven - nagenoeg altijd ten onrechte.

Factoren die de kans op fouten verminderen zijn:

- inzicht en ervaring in het ontwerpen van oplossingen
- systematisch te werk gaan bij het construeren van algoritmen
- keuze van een geschikte (probleemgerichte) programmeertaal
- ondersteuning door een geschikte programmeeromgeving
- voldoende kennis van taal en bedrijfssysteem
- buitengewone zorgvuldigheid.

Een nog weinig bevredigende en weinig algemene studie over de invloed van details van de taal op de foutenkans bij het programmeren is te vinden in [GAN75]. Een levendige beschrijving van de kronkelige denkprocessen van de programmeur is te vinden in [WEI71].

3.2 Systematisch testen

De gebruikelijke gang van zaken bij het ontwikkelen van programma's bestaat uit: schrijven van het programma op een min of meer systematische wijze, daarna vinden van fouten door het programma "uit te proberen"; op de een of andere manier verbeteren van die fouten, en nog eens proberen.

Het opsporen van syntactische en lexicale fouten kan vol vertrouwen aan de computer worden overgelaten. Het menselijk brein is niet zozeer ingericht op dat soort problemen, maar de machine des te meer.

Tenslotte komt dan het grote moment dat het programma voor de eerste keer gaat lopen. Doet het wat het doen moet, of doet het iets volkomen onbegrijpelijks? Is het meteen afgelopen of hult het zich in een eindeloos somber stilzwijgen? Wordt de executie meteen afgebroken, met een of andere duistere technische mededeling?

Als je door dit soort spanningen bevangen wordt dan komt dat omdat je in je manier van werken niet verder gekomen bent dan de gebroeders Wright bij het maken van hun eerste vliegtuigen: Je duwt je probeersel over de rand van de rots en je kijkt wat er gebeurt.

In de eerste plaats kun je al vóór het programma gedraaid wordt pogen fouten daarin te voorkomen. Bovendien, al gaat een programma voor nog zoveel testgevallen goed, dat bewijst nog niet dat het correct is. Binnen de informatica bestaat consensus, dat testen niet zo'n beste methode is om de correctheid van programma's aan te tonen: uit het vinden van fouten blijkt weliswaar zonneklaar dat een programma fout is, maar als er geen fouten worden gevonden volgt daaruit nog geenszins dat het correct is. Er bestaan dan ook compilers die, als ze geen syntactische fouten vinden, eerlijk melden: "none of the errors found" .... Testen is op zijn best een plausibiliteits-contrôle.

Als een programma alle tests goed doorloopt, weet je alleen dat bepaalde soorten fouten ontbreken. Maar de smerigste fouten zijn die, die je niet door testen vindt.

3.2.1 Eerst nadenken

Een deel van de semantische fouten kan door systematische inspectie van het programma en eventueel door simulatie met potlood en papier worden opgespoord, waarbij een aantal voor de hand liggende foutenbronnen wordt nagegaan:

- zijn alle variabelen tijdig en juist geïnitialiseerd?
- is het bereik van alle herhalingen in orde? (één te veel of te weinig tellen is een veelgemaakte fout)
- zijn die operaties, die niet voor alle argumentwaarden welgedefinieerd zijn, daartegen afdoende beschermd? (delen door nul, acces buiten een rij, vierkantswortel van een negatief getal)
- wat gebeurt er bij onbedoelde invoergegevens? (is het programma bestand tegen onjuiste invoer, worden zulke fouten verstandig gemeld)

Men doet er verstandig aan dergelijke vragen te beantwoorden voordat een programma voor de eerste keer wordt gedraaid: Nadenken vooraf scheelt vaak heel wat repareren achteraf.

3.2.2 Hij doet het niet

Met al deze slagen om de arm zijn er toch situaties, waarin u een programma zult moeten testen, omdat een mens nu eenmaal niet perfect is.

- U kunt vergissingen bij de realisering gemaakt hebben (bijvoorbeeld gewoon iets anders getypt hebben dan op uw papier stond) en daar telkens straal overheen lezen.
- Al bereidt u alles voor in detail, het is vaak moeilijk te voorzien welke indruk het gedrag van het programma (prompt's, tussenuitvoer) en het schermbeeld (indeling, mate aan uitleg) zullen maken, laat staan hoe gebruikers ermee om zullen gaan. Om deze pragmatische aspecten naar genoegen te krijgen is een zekere mate aan experimenteren noodzakelijk.
- "The proof of the pudding is in the eating." Het verlangen, het met veel zorg en moeite ontworpen programma ook te zien werken is nu eenmaal menselijk. En wat doe je als het programma dan iets verkeerd doet?

Als je dan toch gaat testen kun je het beter goed doen:

- probeer de verschillende gevallen die in het programma onderscheiden worden zoveel mogelijk uit
- probeer randgevallen
- probeer bekende gevallen.

Vaak heb je daarbij hulpmiddelen ter beschikking om de voortgang van de berekening te volgen (**tracing**) of om te zien hoe vaak elk van de delen van het programma is doorlopen (**profiling**); wat nog niet is uitgevoerd is klaarblijkelijk ook niet getest. Misschien is het zelfs onjuist of overbodig. Wat erg vaak wordt doorlopen is een kritisch punt voor de efficiëntie).

Het is een bekende ervaring dat een programmeur zich kan blindstaren op zijn eigen programma, zodat hij de denkfouten daarin niet meer kan ontdekken. In zo'n geval is het uiterst nuttig, aan een ander te "bewijzen" dat het programma in orde is en dus niet het (foutieve) gedrag kán vertonen dat het aan de dag legt. Als die ander zich voldoende achterdochtig opstelt stuit de programmeur vaak plotseling zelf op zijn eigen denkfout. Ook blijkt bij deze methode ineens veel duidelijker hoe goed of slecht het programma gestructureerd is: hoe beter gestructureerd, hoe gemakkelijker uit te leggen. (Het probleem is altijd, iemand te vinden die interesse voor het programma wil veinzen.)

Als door een groep mensen aan een programma wordt gewerkt, kan het zeer dienstig zijn een van de leden van de groep te benoemen tot "advocaat van de duivel". Hij moet proberen aan te tonen dat er manieren zijn waarop het programma fout kan gaan, hij moet gemene tests bedenken en ook steeds eisen dat het programma zelf (en het gebruik ervan) voor hem glashelder blijft.

3.3 Correctheid bewijzen

Als gevolg van onze wetenschappelijke training beschouwen we een met mathematische gestrengheid gevoerd bewijs als de hoogste vorm van zekerheid. Was het geven van een dergelijk bewijs enkele jaren geleden nog een utopie, tegenwoordig ontwikkelt zich de informatica in een richting waarin het voeren van correctheidsbewijzen een normale zaak is.

Van informaticastudenten mag men, anders dan van programmeerartiesten en slecht opgeleide beroepsprogrammeurs, verwachten dat zij zich met deze methoden vertrouwd maken, niet zozeer om uit plichtsbetrachting ieder programma te bewijzen, maar vanwege de weldadige invloed op de juistheid van de programma's die uitgaat van de instelling, programma's zó te schrijven, dat een correctheidsbewijs mogelijk is.

Te allen tijde moet de juistheid van programma's plausibel gemaakt zijn en in moeilijke gevallen moet ook werkelijk een bewijs worden geleverd.

3.3.1 Wat is correctheid?

Met **correctheid** van een programma wordt hier niet de lexicale of syntactische juistheid bedoeld maar de omstandigheid, dat daardoor inderdaad het probleem ter oplossing waarvan het bedoeld is wordt opgelost.

Aan de ene kant moet, opdat de correctheid kan worden vastgesteld, het probleem voldoende nauwkeurig geformuleerd zijn, in de vorm van een **specificatie** van de voorwaarden, waaraan het resultaat van het programma moet voldoen; te bewijzen valt dan, dat het resultaat van uitvoering van het programma aan die specificatie zal voldoen mits aan bepaalde beginvoorwaarden is voldaan.

Aan de andere kant moet het programma een zodanige vorm hebben, dat een dergelijk bewijs kan worden gevoerd. Het meest geschikt daarvoor is een algoritmische formulering die de belangrijkste stappen van de programma-uitvoering op een geschikt abstractieniveau laat zien zonder dat de bewijsvoering in een moeras van details verdwijnt. Voor kleine programma's voldoet een formulering in een algoritmische programmeertaal, voor grote programma's kan het bewijs op verschillende abstractieniveaus, door een hiërarchie van bewijzen van de eigenschappen van de deelalgoritmen en van hun samenhangen worden geformuleerd.

In de literatuur over correctheid is dit laatste voorlopig nog de uitzondering. De meeste inzichten zijn uit armzalig kleine programma's verkregen. Vanzelfsprekend beperken ook wij ons tot kleine programma's, in het vaste vertrouwen, dat de bewijsmethodiek in luttele jaren veel verder ontwikkeld zal zijn - in het bijzonder de geautomatiseerde ondersteuning ervan.

3.3.2 Correctheidsbewijs

Voor het **correctheids bewijs** wordt het programma opgevat als de constructieve realisatie van een functie $F : A \to B$ van invoergegevens naar uitvoergegevens. Intuïtief gesproken: een algoritme die F realiseert dient bij elke invoer hetzelfde resultaat leveren als F zelf en wel in eindig vele stappen, voor die invoer moet het algoritme ook *termineren*.

In het algemeen is de functie F zoals die gerealiseerd is door een algoritme slechts een *partiële functie*, waarbij de functie alleen voor invoergegevens in een bepaald definitiegebied is gedefinieerd; buiten dat gebied is F ongedefinieerd, en een corresponderend programma mag ieder resultaat opleveren en hoeft zelfs niet te termineren.

Het bewijs, dat een algoritme T een gegeven functie F met domein A correct realiseert ("het correctheidsbewijs voor T") bestaat dus uit twee delen:

- (**terminatie**) levert de uitvoering van het algoritme T voorelke invoer in A na een eindig aantal stappen een welgedefinieerde waarde?
- (**partiële correctheid**) geldt voor elke $x \in A$: $T(x) = F(x)$?

Als hulpmiddelen voor zulke bewijzen staan ter beschikking: gevalonderscheiding, volledige inductie, het formalisme van de predicatenrekening, en verder scherp toekijken.

In het volgende hoofdstuk introduceren we een arsenaal aan formele bewijsregels, maar eerst geven we een nogal informeel voorbeeld van een correctheidsbewijs.

3.4 Voorbeeld: maximum bepaling

We bekijken het probleem om het maximum van n metingen te bepalen. Om het ons gemakkelijk te maken gaan we er van uit, dat de meetwaarden zijn opgeborgen in een rij

```
ROW n REAL VAR meting
```

We willen — in de `REAL` variabele `grootste` — het grootste element van deze rij bepalen. Na de berekening moet dus gelden

$\texttt{grootste} = maximum_{i\in[1:n]}(\texttt{meting[i]})$.

Dit probleem is zo eenvoudig, dat we een eerste oplossing ervoor direct kunnen opschrijven.

```
REAL VAR grootste :: min oneindig;
INT VAR i;
FOR i UPTO n
REP IF meting[i] > grootste
    THEN grootste := meting[i]
    FI
ENDREP
```

Met `min oneindig` wordt een waarde bedoeld die in elk geval niet groter is dan het maximum. (Het "echte" min oneindig kan natuurlijk niet worden gerepresenteerd.) In dit geval voldoet elk element van het rij aan de genoemde eis, dus ook het eerste. We hoeven dat element dan bovendien niet meer te testen. Deze overwegingen leveren het programma

```
REAL VAR grootste :: meting[1];
INT VAR i;
FOR i FROM 2 UPTO n
REP IF meting[i] > grootste
   THEN grootste := meting[i]
   FI
ENDREP
```

Nu ontstaat wel een nieuw probleem: wat gebeurt er als $n = 0$? Dan zou `meting[1]` een indexeringsfout geven. In dat geval is echter de waarde van het maximum hoe dan ook ongedefinieerd. Je kunt weliswaar niet zeggen dat de algoritme dan incorrect is, maar het is toch vervelend als het algoritme op een explosieve manier eindigt. We stellen dus vooraf de eis, dat $n > 0$ moet zijn. We moeten dan bewijzen:

> "onder de voorwaarde $n > 0$ geldt na afloop van het algoritme `grootste` $=$ $maximum_{i\in[1:n]}meting[i]$".

We gebruiken de notatie $MAX(1 : t)$ voor $maximum_{i\in[1:t]}meting[i]$ (het partieel maximum) en $P(t)$ voor de bewering: `grootste` $= MAX(1 : t)$. We willen bewijzen dat, onder de aanname $n > 0$, na afloop van het algoritme $P(n)$ geldt.

3.4.1 Asserties

We zullen nu op verschillende plaatsen in het programma uitspraken die daar gelden (**asserties**) invoegen. We noteren deze als commentaren tussen accoladen. Ze behoren niet tot het eigenlijke programma maar vormen een verduidelijkende annotatie daarop. We moeten bewijzen, dat zo'n uitspraak steeds juist is als de uitvoering van het programma is aangekomen op de plaats waar de uitspraak staat (methode der **inductive assertions** [FLO67]).

```
    IF n>0
    THEN {n>0}
         REAL VAR grootste :: meting[1];
{P(1)}
         INT VAR i;
         FOR i FROM 2 UPTO n
         REP IF meting[i] > grootste
             THEN grootste := meting[i]
             FI
         ENDREP {P(n)}
    FI {n>0 -> P(n)}
```

Gemakkelijk is in te zien, dat de correctheid essentieel berust op het feit dat voor toenemende waarden van `i` steeds het maximum van de partiële rij `meting[1:i]` aan `grootste` wordt toegekend. Met andere woorden, dat na iedere slag van de herhaling de uitspraak `P(i)` waar is. Zo'n uitspraak heet een **invariant** van de herhaling. Het probleem van de correctheid van algoritmen is in de practijk vaak "slechts" het vinden van de juiste invarianten voor de herhalingen in die algoritme. We voegen de invariant als assertie toe, en baseren het bewijs dus op:

```
    IF n>0
    THEN {(1) n>0}
         REAL VAR grootste :: meting[1];
{(2) P(1)}
         FOR i FROM 2 UPTO n
         REP {(3) P(i-1)}
             IF meting[i] > grootste
             THEN grootste := meting[i]
             FI {(4) P(i)}
         ENDREP {(5) P(n)}
    FI {(6) n>0 -> P(n)}
```

Om naar de asserties te kunnen verwijzen, hebben we ze genummerd.

3.4.2 Bewijs van partiële correctheid

- (1) geldt in het `THEN` gedeelte. Blijkbaar geldt voor conditional-clauses altijd:
  `IF a THEN {a}` S_1 `ELSE {NOT a}` S_2 `FI`.
- (2) volgt uit het feit dat $MAX(1 : 1)$ =`meting[1]`, zie de definitie van MAX.
- (3) geldt in elk geval de eerste keer dat de herhaling doorlopen wordt.
- (4) geldt in ieder geval bij de eerste doorloop: dan is $i = 2$ en als `meting[2] > meting[1] = grootste`,
  dan is bij (4) `grootste = meting[2]` en anders is (nog steeds) `grootste = meting[1]`. Dus is `grootste`$= maximum(meting_1, meting_2)$.
- Het is duidelijk, dat door het ophogen van `i` met 1 steeds (3) volgt uit (4) bij het opnieuw betreden van de loop.
- Als `meting[i]`$> MAX(1 : i - 1)$, dan zorgt `IF ...FI` ervoor dat $MAX(1 : i)$ gelijk wordt aan `meting[i]` en anders is reeds $MAX(1 : i)$ $=$

$MAX(1:i-1)$. Dus is bij (4) altijd `grootste=` $MAX(1:i)$. Dit is de wezenlijke stap in het bewijs.

- Bij het verlaten van de `FOR`-loop is `i=n AND P(i)`, dus `P(n)`.
- Tenslotte volgt (6) uit (1) en (5).

De terminatie is verzekerd doordat de definitie van de `FOR`-clause in Elan eindigheid van de beperkte herhaling garandeert.

3.5 Asserties in Elan

Een eerste mogelijkheid, asserties op te nemen in een programma, bestaat erin ze als commentaren op te vatten. Niet toevallig zijn de assertie-accoladen in Elan als commentaar-haken toegestaan. Een commentaar zal dan namen (van objecten en operaties) uit het omringende programma bevatten. Zo kan het vorige programma worden ingevoerd als

```
IF n>0
THEN { n>0 }
    REAL VAR max::meting[1];
    { P(1) }
    FOR i FROM 2 UPTO n
    REP { P(i-1) }
        IF meting [i] > max
        THEN max:= meting[i]
        FI { P(i) }
    ENDREP { P(n) }
FI { n>0 -> P(n) }
```

De als commentaar gegeven asserties hebben een belangrijke documentaire waarde: ze leggen vast wat je wilt dat er geldt op de aangegeven punten. Ze hebben echter (zoals alle commentaren) geen functie in het programma, er is met name geen reden onder de zon waarom ze juist zouden zijn (en juist zouden blijven bij programma wijzigingen). Daarom kunnen ze ook in een volkomen vrije stijl worden opgeschreven — ze hoeven alleen maar begrijpelijk te zijn voor de menselijke lezer.

Een mogelijkheid om van een assertie toch een functioneel deel van een programma te maken bestaat daarin, deze op te laten treden als parameter van een oproep van de Elan-procedure

```
PROC assert (BOOL voorwaarde):
  { onderbreek de programma-uitvoering }
  { als de voorwaarde FALSE oplevert }
```

Een oproep van *assert* heeft geen effect als de meegegeven voorwaarde inderdaad geldt, maar als de voorwaarde niet geldt wordt de programmauitvoering beëindigd (en gaat de Elan-vertolker over in trace-toestand, zodat u de momentane variabelen-waarden kunt inspecteren, etc.).

Een voorbeeld: de recursieve berekening van de faculteit kan **robuust** gemaakt worden tegen negatieve argumenten door middel van een oproep van `assert`.

```
INT PROC fac (INT n):
   assert (n>=0);
   IF n = 0 THEN 1 ELSE n * fac (n-1) FI
ENDPROC;
```

Hierbij moet de assertie de vorm hebben van een Boole'se uitdrukking, betekenisvol in de context waarin hij optreedt — dus buiten Elan liggende elementen als quantoren, som-tekens e.d. kunnen we dan niet gebruiken.

Een variant van de procedure `assert` is ook nog bereid bij het eventuele onderbreken een reden op te geven:

```
PROC assert (BOOL voorwaarde, TEXT
bericht):
  { onderbreek programma-uitvoering }
  { als de voorwaarde FALSE oplevert }
  { maar geef dan eerst dit bericht }
```

met als voorbeeld van een oproep

```
REAL PROC f (REAL a,b):
  assert (a>b, "Interval [a,b] is
leeg");
  ...
```

Het is verstandig, in de testfase op alle daarvoor in aanmerking komende punten van uw programma een geschikt geparametiseerde oproep van `assert` op te nemen.

Vervolgens kunt u successievelijk die oproepen, die corresponderen met asserties waarvan u de geldigheid bewezen hebt, weer elimineren. Veelal blijven dan een aantal onbewezen asserties over die aannamen met betrekking tot de invoer e.d. uitdrukken.

Deze laatste kunt u alleen elimineren door expliciet testen; u kunt dan meteen de gelegenheid geven tot correctie van de invoer, wat voert tot gebruikersvriendelijke programma's.

Wat u niet kunt of wilt bewijzen kunt u afzekeren met een geschikte oproep van `assert`.

3.6 Opgaven

1. Haal een aantal programma's van beginners uit de papiermand en onderzoek ze op TD stijlfouten. Doe hetzelfde met je eigen eerste wrochtsels.
2. Ga voor ieder van de vier elementaire typen van Elan na welke operaties *partieel* zijn, d.w.z. voor enigerlei waarde van hun argumenten geen gedefinieerd resultaat opleveren.
3. Als je weet dat de algoritme T correct is ten opzichte van zijn specificatie, die de vorm heeft van een functie F met domein A, wat weet je

dan met betrekking tot het gedrag van T voor invoer die niet tot A behoort:

- T raakt in eindeloze herhaling?
- T wordt afgebroken?
- T levert een onjuist antwoord?
- T levert een willekeurig antwoord?
- T geeft een foutmelding?

Welk gedrag lijkt je het juiste vanuit het standpunt van de programmatuurkunde?

Chapter 4

Correctheid (2)

In dit hoofdstuk behandelen we een calculus van asserties, de bewijsmethode der "inductieve asserties" en een methode voor de constructie van programma's door verfijning van asserties.

Onder een **assertie** verstaan we een logische uitspraak over de waarden van de objecten van een programma waarmee een bepaalde plaats in het programma gemarkeerd is. Met een assertie proberen we uit te drukken wat er bekend is als de uitvoering van het programma die plaats bereikt.

4.1 Een calculus van asserties

De bewijsmethode door asserties is gebaseerd op het opstellen van een reeks uitspraken $\{P_i\}$ afgewisseld met (elementaire of samengestelde) algoritmen $\mathcal{S}_i$, genoteerd

$$\{P_1\}\,\mathcal{S}_1\,\{P_2\}\ \ldots\ \{P_{n-1}\}\,\mathcal{S}_{n-1}\,\{P_n\}$$

Hierin is de reeks $\mathcal{S}_1, \mathcal{S}_2, \ldots \mathcal{S}_{n-1}$ de te bewijzen algoritme. Door uitvoering van achtereenvolgende $\mathcal{S}_i$ bereiken we uiteindelijk dat $\{P_n\}$ geldt. De reeks vormt een bewijs van (partiële) correctheid als voor elke $\mathcal{S}_i$ geldt, dat juist na de uitvoering ervan P_{i+1} waar is mits juist vóór de uitvoering P_i waar is. Dan geldt transitief, dat als vóór de uitvoering van het hele algoritme is voldaan aan de voorwaarde P_1, na de uitvoering P_n geldt. P_1 beschrijft al het relevante dat van tevoren bekend is, P_n dat wat na afloop bekend is (hopelijk de oplossing). Elke tussenliggende P_i dient ertoe, de objecten van de algoritme een stukje dichter bij de oplossing te brengen.

4.1.1 De notatie

We zullen voor een aantal standaardalgoritmen aangeven op welke manier zij predicaten in elkaar doen overgaan. Daarvoor moeten we eerst de interpretatie van $\{P\}\,\mathcal{S}\,\{Q\}$ nog wat aanscherpen.

Definitie: Onder

$$\{P\}\,\mathcal{S}\,\{Q\}$$

verstaan we: als P geldt direct voor uitvoering van S en S termineert dan zal direct na de uitvoering van S de uitspraak Q gelden.

Omdat de voorwaarde P geen terminatie hoeft te garanderen en er geen rekening met bijv. delen door 0 of overflow hoeft te worden gehouden, spreken we in dit geval van **partiële correctheid**.

We voeren ook een bij inductieve asserties aansluitende notatie

$$\{P\}\ \{Q\}$$

in voor de implicatie ($P \Rightarrow Q$). Een belangrijke eigenschap is nu:

$$\begin{array}{rl} \textit{als}: & \{P'\}\ \{P\}\,\mathcal{S}\,\{Q\}\ \{Q'\} \\ \textit{dan ook}: & \{P\}\,\mathcal{S}\,\{Q'\} \\ \textit{en ook}: & \{P'\}\,\mathcal{S}\,\{Q\} \end{array}$$

dat wil zeggen: bij een gegeven P geldt na $\mathcal{S}$ ook elke voorwaarde Q' die zwakker is dan Q. We noemen G een **zwakkere voorwaarde** ("minder beperkend") dan H als $H \Rightarrow G$. Als, bijvoorbeeld,

$$\{x < 3\}\ \mathtt{x := x + 1}\ \{x < 4\}$$

dan is ook

$$\{x < 3 \wedge i = 5\}\ \mathtt{x := x + 1}\ \{x < 4\}$$

en ook

$$\{x < 3\}\ \mathtt{x := x + 1}\ \{x < 8\}$$

We beschouwen nu $\{P\}\,\mathcal{S}\,\{Q\}$ nader. We onderscheiden hierbij de *voorwaartse* en de *achterwaartse* beschouwingswijze.

Laten we de situatie eerst onderzoeken voor vaste P en $\mathcal{S}$. In het algemeen kunnen we nu vele relaties, zeg $Q_1, Q_2, \ldots$ vinden zodanig dat $\{P\}\,\mathcal{S}\,\{Q_i\}$, bijvoorbeeld:

- $\{x > 0\}\ \mathtt{x := x + 1}\ \{x > 1\}\ (Q_1)$
- $\{x > 0\}\ \mathtt{x := x + 1}\ \{x > 0\}\ (Q_2)$
- $\{x > 0\}\ \mathtt{x := x + 1}\ \{x > -1\}\ (Q_3)$

In dit voorbeeld zien we dat Q_1 de sterkste conclusie is die getrokken kan worden, alle andere zijn zwakker:

$$\begin{array}{rcl} Q_1 & \Rightarrow & Q_2 \\ Q_1 & \Rightarrow & Q_3 \end{array}$$

We voeren nu de voorwaartse **uitspraak omvormer** $\mathcal{POST}$ in, die bij gegeven preconditie P en algoritme $\mathcal{S}$ de **sterkste postconditie** oplevert. Met andere woorden:

- $\{P\}\,\mathcal{S}\,\{\mathcal{POST}(P,\mathcal{S})\}$
- als $\{P\}\,\mathcal{S}\,\{Q\}$ dan $\mathcal{POST}(P,\mathcal{S}) \Rightarrow Q$

Op analoge wijze kunnen we de *achterwaartse* uitspraak omvormer $\mathcal{PRE}$ invoeren, die bij gegeven postconditie Q en algoritme $\mathcal{S}$ de zwakste **zwakste preconditie** preconditie (de "weakest precondition") oplevert, onder de aanname dat $\mathcal{S}$ termineert.

- $\{\mathcal{PRE}(\mathcal{S},Q)\}\,\mathcal{S}\,\{Q\}$
- als $\{P\}\,\mathcal{S}\,\{Q\}$ dan $P \Rightarrow \mathcal{PRE}(\mathcal{S},Q)$

OPGAVE: Laat zien dat

1. $P \Rightarrow \mathcal{PRE}(\mathcal{S},\mathcal{POST}(P,\mathcal{S}))$
2. $\mathcal{POST}(\mathcal{PRE}(\mathcal{S},Q),\mathcal{S}) \Rightarrow Q$
3. Wat is de interpretatie van $\mathcal{PRE}(\mathcal{S},true)$?
4. Wat is de interpretatie van $\mathcal{PRE}(\mathcal{S},false)$?
5. Wat is de interpretatie van $\mathcal{POST}(\mathcal{S},true)$?
6. Wat is de interpretatie van $\mathcal{POST}(\mathcal{S},false)$?

We zullen bij de vastlegging van de betekenis van de taalelementen als uitspraak omvormers beide varianten bespreken. Eerst bespreken we de basisoperaties, vervolgens de constructiemiddelen.

4.1.2 Basisopdrachten

Bij de semantische beschrijving van de programmeertaal wordt uitgegaan van syntactisch correcte programma's, dat wil zeggen, dat alle variabelen gedeclareerd zijn, en pas na declaratie worden gebruikt. Aldus mogen we de declaratie

```
X VAR v
```

opvatten als een lege opdracht, en een declaratie met beginwaarde

```
X VAR v :: E
```

als een toewijzing `v := E`.

Ter vereenvoudiging van de argumentatie zullen we een aantal aannamen maken. Tenzij expliciet anders is aangegeven geldt in dit hoofdstuk:

- De namen die in uitspraken voorkomen staan voor namen van objecten (variabelen etc.) van het programma.
- Gelijke namen staan voor eenzelfde object, verschillende namen staan voor verschillende objecten (geen naam is dus een alias voor een andere).
- De typen van de genoemde objecten zijn af te leiden uit de programmacontext, we geven in de uitspraken niet expliciet een typering aan.
- Bij uitspraken over de waarde van een object nemen we steeds aan dat dat object een welgedefinieerde waarde bezit, tenzij uitdrukkelijk wordt aangegeven dat zijn waarde ongedefinieerd is.
- Bij operaties in het domein van de integers en de reals nemen we aan dat geen overflow optreedt, tenzij uitdrukkelijk wordt gevraagd rekening te houden met overflow.

De lege opdracht

Een bijzonder eenvoudige semantiek heeft de lege opdracht. Uiteraard is deze opdracht niet van groot belang om effecten te boeken in een programma, maar hij vult bepaalde gaten. De programmeertaal Elan heeft zelfs geen expliciete notatie voor deze opdracht. We geven de lege opdracht in dit hoofdstuk weer als `SKIP`, net zoals in ALGOL68.

De semantiek luidt nu eenvoudig:

$$\mathcal{PRE}(\texttt{SKIP},Q) = Q$$

$$\mathcal{POST}(P,\texttt{SKIP}) = P$$

We zien hieraan dat we niets winnen met een `SKIP` opdracht, maar ook niets verliezen.

De toewijzing

De meest elementaire bewerking om vooruitgang te boeken in een berekening is gebruik te maken van de toewijzingsopdracht:

```
v := E
```

Als gevolg hiervan krijgt de variabele v als nieuwe waarde de uitkomst van de evaluatie van E. De (eventuele) oude waarde van v gaat hierbij verloren, tenzij deze waarde met behulp van de nieuwe waarde E van v teruggevonden kan worden. Zo kan men het effect van `v := v + 1` teniet doen door dit te laten volgen door `v := v - 1` (tenzij v nog geen waarde had of overflow optreedt).

De toewijzingsopdracht brengt dus niet veel meer tot stand dan een gewijzigde visie op het zover berekende. De semantiek brengt dit tot uitdrukking in de tautologische gedachte:

> *opdat na afloop een relatie geldt*
> *dient deze van tevoren al te*
> *gelden*

die ten grondslag ligt aan de **achterwaartse regel**

$$\mathcal{PRE}(\texttt{x} := \texttt{E},Q) = [\ Q\ ]^x_E$$

Hierin is $[Q]^x_E$ de expressie die ontstaat door in Q alle (vrije) voorkomens van x te vervangen door E.

VOORBEELDEN:

- $\mathcal{PRE}(\texttt{x} := 5, x = 5) =$
  $[\ x = 5\ ]^x_5 = [5 = 5] = \text{true}$
- $\mathcal{PRE}(\texttt{x} := 5, x = 3) =$
  $[\ x = 3\ ]^x_5 = [5 = 3] = \text{false}$
- $\mathcal{PRE}(\texttt{x} := \texttt{x} + 1, x = 1) =$
  $[\ x = 1\ ]^x_{x+1} = [x + 1 = 1] = [x = 0]$

De semantische regel gaat er van uit dat de expressie E tijdens de evaluatie in `v:= E` een welgedefinieerde waarde oplevert. Bij het bepalen van pre- en postconditie moet dus terdege rekening worden gehouden met eventuele partiële operaties.

VOORBEELD:

$$\begin{array}{lll} \mathcal{PRE}(\mathtt{x} := \mathtt{1/y}, abs(x) > 1) & & \\ & = & [\ abs(x) > 1\ ]^{x}_{1/y} \\ & = & [abs\left(\frac{1}{y}\right) > 1] \\ & = & [y \neq 0 \wedge abs(y) < 1] \end{array}$$

We bekijken vervolgens het effect van de toewijzing `v := E` op de variabelen verschillend van v.

VOORBEELD:

$$\mathcal{PRE}(\mathtt{v} := \mathtt{3}, w = 0) = [\ w = 0\ ]^{v}_{3} = [w = 0]$$

In dit voorbeeld gaan we ervan uit dat v en w geen alias zijn van elkaar. (Een variabele is een **alias** voor een andere als een toewijzing aan de ene tevens een toewijzing is aan de andere). In Elan is dit zeker niet het geval voor elementaire namen buiten een procedure. Binnen een procedure moeten we echter oppassen voor het alias probleem, omdat via een `VAR`-parameter een alias voor een globale variabele kan ontstaan.

We gaan er dus van uit dat de toewijzing aan v de waarde van w niet beïnvloedt. Moeilijker is de situatie voor samengestelde namen, zoals in het volgende

VOORBEELD:

$$\mathcal{PRE}(\mathtt{a[i]} := \mathtt{0}, a[j] = 0) = [\ a[j] = 0\ ]^{a[i]}_{0}$$

We moeten in de expressie $a[j] = 0$ alle voorkomens van $a[i]$ vervangen door 0. We dienen dus te onderscheiden tussen de gevallen $i = j$ en $i \neq j$. Dat levert

$$\begin{array}{lll} [\ a[j] = 0\ ]^{a[i]}_{0} & & \\ & = & [(i = j \wedge 0 = 0) \vee (i \neq j \wedge a[j] = 0)] \\ & = & [i = j \vee (i \neq j \wedge a[j] = 0)] \\ & = & [i = j \vee a[j] = 0] \end{array}$$

In geval $i = j$ is $a[i]$ een alias voor $a[j]$ (men noemt zo een waarde-afhankelijk alias een **dynamisch alias**). Nog complexer wordt de situatie in het volgende voorbeeld:

VOORBEELD:

$$\mathcal{PRE}(\mathtt{a[i]} := \mathtt{0}, a = b) = [\ a = b\ ]^{a[i]}_{0}$$

Hier zijn a en b rijen. We zullen de impliciete verwijzing naar $a[i]$ nu expliciet moeten maken om de substitutie uit te kunnen voeren:

$$[\ a = b\ ]^{a[i]}_{0} =$$
$$[\ a[i] = b[i] \wedge \forall_{1 \leq j \leq n, j \neq i} a[j] = b[j]\ ]^{a[i]}_{0} =$$
$$0 = b[i] \wedge \forall_{1 \leq j \leq n, j \neq i} a[j] = b[j]$$

waarbij we aannemen dat n de grootte is van `ROW a`.

Vereenvoudiging is in dit geval mogelijk door een notationele verrijking, bijvoorbeeld een notatie voor de substitutie van rij-elementen:

$$[\ a = b\ ]^{a[i]}_{0} = \Big(\ [\ a\ ]^{i}_{0} = b\ \Big)$$

We geven nu ook de **voorwaartse regel** voor de toewijzing:

$$\mathcal{POST}(P, \mathtt{x} := \mathtt{E}) = \exists_{x'}\ [\ [P]^{x}_{x'} \wedge x = [E]^{x}_{x'}\ ]$$

Hierin is x' een gebonden parameter die bij voorkeur geëlimineerd wordt. Eigenlijk is x' een representant van de *oude* waarde van x.

VOORBEELD:

$$\mathcal{POST}(x+y > 0, \mathtt{x} := \mathtt{x - y}) = \exists_{x'}[x'+y > 0 \wedge x = x'-y]$$

Na eliminatie van x' vinden we voor de postconditie $x + 2y > 0$.

Het is lang niet altijd eenvoudig de parameter te elimineren. In plaats van de lastige voorwaartse regel kunnen we met enige voorzichtigheid in veel gevallen de achterwaartse regel omgekeerd toepassen, n.l. onder de voorwaarde dat de variabele x niet in de preconditie P voorkomt, anders dan als een deel van een voorkomen van E. We kunnen schrijven

$$safe(x, P) \Rightarrow \mathcal{POST}(P, \mathtt{x} := \mathtt{E}) = [\ P\ ]^{E}_{x}$$

waarbij $safe(x, P)$ de genoemde voorwaarde uitdrukt.

4.1.3 Samenstellingsmechanismen

De verfijning

Het verfijningsmechanisme in Elan kan men beschouwen als enkel en alleen een naamgevingsmechanisme voor algoritmen:

$$\mathtt{naam} : \mathcal{S}.$$

de betekenis van een verfijningsnaam kan dus worden teruggebracht tot die van zijn romp. Als regel:

$$\mathcal{PRE}(\mathtt{naam}, Q) = \mathcal{PRE}(\mathcal{S}, Q)$$

$$\mathcal{POST}(P, \mathtt{naam}) = \mathcal{POST}(P, S)$$

Opeenvolging

De eenvoudigste manier om uit onderdelen een nieuw algoritme samen te stellen is door middel van het opeenvolgingsmechanisme. Het beoogde effect is duidelijk, met:

$$\mathcal{S}_1; \mathcal{S}_2$$

wordt bedoeld dat $\mathcal{S}_1$ en $\mathcal{S}_2$ achter elkaar (in die volgorde) worden uitgevoerd. De semantiek laat zich eenvoudig beschrijven:

$$\mathcal{PRE}(\mathcal{S}_1; \mathcal{S}_2, Q) = \mathcal{PRE}(\mathcal{S}_1, \mathcal{PRE}(\mathcal{S}_2, Q))$$

Opgave: Geef zelf de voorwaartse regel voor de opeenvolging.

We demonstreren dit door middel van enkele voorbeelden:

Voorbeeld:

$$\begin{aligned}
&\mathcal{PRE}(\mathtt{x := x+1; x := x-1}, x = 0)\\
&\quad = \mathcal{PRE}(\mathtt{x := x+1}, \mathcal{PRE}(\mathtt{x := x-1}, x = 0))\\
&\quad = \mathcal{PRE}(\mathtt{x := x+1}, x - 1 = 0)\\
&\quad = (x+1) - 1 = 0\\
&\quad = x = 0
\end{aligned}$$

Merk op dat deze laatste voorwaarde impliceert dat x een waarde moet hebben. Dat zullen we steeds verlangen van de objecten die in asserties voorkomen, en daarom meestal niet expliciet aangeven.

Ook zullen we als daar niet expliciet om gevraagd wordt geen rekening houden met overflow; het volgende voorbeelt tooont hoe dat wel zou kunnen.

Voorbeeld:

$$\begin{aligned}
&\mathcal{PRE}(\mathtt{x := x+1; x := x-1}, R)\\
&\quad = \mathcal{PRE}(\mathtt{x := x+1}, \mathcal{PRE}(\mathtt{x := x-1}, R))\\
&\quad = \mathcal{PRE}(\mathtt{x := x+1}, [\ R\ ]^{x}_{x-1})\\
&\quad = \left[\ [\ R\ ]^{x}_{x-1}\ \right]^{x}_{x+1}\\
&\quad = [\ R\ ]^{x}_{(x+1)-1}\\
&\quad = [\ R\ ]^{x}_{x} \wedge (x+1) - 1 \text{ heeft een waarde}\\
&\quad = R \wedge x \text{ heeft een waarde} \wedge x < maxint
\end{aligned}$$

In deze laatste stap hebben we rekening ermee gehouden dat het tussenresultaat $x+1$ niet tot overflow mag leiden.

Opgave: 1. Laat zien dat het volgende programmafragment een verwisseling van de waarden van a en b bewerkstelligt

```
a := a - b;
b := a + b;
a := b - a
```

2. Dezelfde vraag, nu echter dient u *overflow* in rekening te brengen.
3. Laat zien dat de volgende bewering niet geldt

$$\mathcal{PRE}(\mathcal{S}, \mathcal{POST}(P, \mathcal{S})) \Rightarrow P$$

door voor S te kiezen `x := 1`, en voor P te kiezen $x = 3$.

De keuze

Het volgende samenstellingsmechanisme dat we bekijken is de voorwaardelijke uitvoering. We bekijken eerst:

$$\mathtt{IF\ b\ THEN}\ \mathcal{S}_1\ \mathtt{ELSE}\ \mathcal{S}_2\ \mathtt{FI}$$

en bespreken daarna de generalisaties. De intuïtie is heel eenvoudig aan te geven via asserties: indien we kunnen bewijzen dat

1. $\{P \wedge b\}\ \mathcal{S}_1\ \{Q\}$
2. $\{P \wedge \neg b\}\ \mathcal{S}_2\ \{Q\}$

dan kunnen we concluderen

$\{P\}\ \mathtt{IF\ b\ THEN}\ \mathcal{S}_1\ \mathtt{ELSE}\ \mathcal{S}_2\ \mathtt{FI}\ \{Q\}$.

We gaan er hierbij van uit dat de algoritme b geen effect heeft op P (de voorwaarde niet verandert). Dit leidt tot de volgende regel:

$$\begin{aligned}
&\mathcal{PRE}(\mathtt{IF\ b\ THEN}\ \mathcal{S}_1\ \mathtt{ELSE}\ \mathcal{S}_2\ \mathtt{FI}, Q)\\
&\quad = b \Rightarrow \mathcal{PRE}(\mathcal{S}_1, Q)\\
&\quad\quad \wedge \neg b \Rightarrow \mathcal{PRE}(\mathcal{S}_2, Q)
\end{aligned}$$

We geven een voorbeeld:

Voorbeeld:

$$\begin{aligned}
&\mathcal{PRE}(\mathtt{IF\ a} \geq \mathtt{b\ THEN\ m := a\ ELSE\ m := b\ FI}, m = max(a,b))\\
&\quad = a \geq b \Rightarrow \mathcal{PRE}(\mathtt{m := a}, m = max(a,b))\\
&\quad\quad \wedge a < b \Rightarrow \mathcal{PRE}(\mathtt{m := b}, m = max(a,b))\\
&\quad = [a \geq b \Rightarrow a = max(a,b)] \wedge [a < b \Rightarrow b = max(a,b)]\\
&\quad = true
\end{aligned}$$

De varianten van de keuze zijn nu eenvoudig afleidbaar, met behulp van de volgende gelijkheden:

- $\mathtt{IF\ b\ THEN}\ \mathcal{S}\ \mathtt{FI} \equiv \mathtt{IF\ b\ THEN}\ \mathcal{S}\ \mathtt{ELSE\ SKIP\ FI}$
- $\mathtt{IF\ b_1\ THEN}\ \mathcal{S}_1\ \mathtt{ELIF\ b_2\ THEN}\ \mathcal{S}_2\ \mathtt{ELSE}\ \mathcal{S}_3\ \mathtt{FI} \equiv$
  $\mathtt{IF\ b_1\ THEN}\ \mathcal{S}_1\ \mathtt{ELSE\ IF\ b_2\ THEN}\ \mathcal{S}_2\ \mathtt{ELSE}\ \mathcal{S}_3\ \mathtt{FI\ FI}$

4.2 Opgaven (1)

De volgende reeks vingeroefeningen is afkomstig van H. Goemans.

1. (toewijzing) Bewijs dat geldt:
   - $\{x \geq 0\}\ \mathtt{x := x+1}\ \{x > 0\}$
   - $\{a = 13\}\ \mathtt{b := 7}\ \{a = 13\}$
   - $\{a = 13\}\ \mathtt{b := 7}\ \{b = 7\}$
   - $\{true\}\ \mathtt{b := 7}\ \{b = 7\}$
   - $\{false\}\ \mathtt{b := 7}\ \{b = 13\}$
   - $\{x = a, y = b\}$
     $\mathtt{h := x; x := y; y := h}$
     $\{x = b, y = a\}$
   - $\{x = a, y = b\}$
     $\mathtt{x := x+y; y := x-y; x := x-y}$
     $\{x = b, y = a\}$
2. (keuze)
   - $\{true\}$
     $\mathtt{IF\ x > y\ THEN\ m := x\ ELSE\ m := y\ FI}$
     $\{m = max(x,y)\}$
   - $\{x > y\}$
     $\mathtt{IF\ x > 0\ THEN\ x := x-1\ ELSE\ y := y+1\ FI}$
     $\{x \geq y\}$
3. (invarianten)
   - $\{x = n!\}\ \mathtt{n := n+1; x := n*x}\ \{x = n!\}$
   - $\{x = 2^i\}\ \mathtt{x := 2*x; i := i+1}\ \{x = 2^i\}$

- $\{y = a * x + b\}$
  `x := x + 1; b := b - a`
  $\{y = a * x + b\}$

4. Bepaal in de volgende voorbeelden telkens de zwakste preconditie Q.

- $\{Q\}$ `x := x + 10` $\{x > 0\}$
- $\{Q\}$ `x := x + x` $\{x > 0\}$
- $\{Q\}$ `x := x + y` $\{x = 2 * y\}$
- $\{Q\}$ `x := x + y` $\{x = -1 \wedge y = 1\}$
- $\{Q\}$`x := 2 * y; w := -x` $\{x > y > 0 \wedge wx \geq 0\}$
- $\{Q\}$ `x := x + 1` $\{true\}$
- $\{Q\}$ `x := (m + n) DIV 2` $\{m \leq x \leq n\}$
- $\{Q\}$ `x := x DIV y` $\{x = 7\}$
- $\{Q\}$ `z := 0` $\{\exists w(x^2 + y^2 = w^2) \wedge z = 0\}$
- $\{Q\}$
  `IF x = 2 * y THEN x := 3`
  `ELSE y := x DIV 3 FI`
  $\{x = 3y\}$
- $\{Q\}$
  `IF x = y THEN x := 0 ELSE y := 0 FI`
  $\{x = y\}$
- $\{Q\}$ `IF x < 0 THEN x := x + 2 FI` $\{x > 0\}$

5. (sterkste postconditie) Bepaal in de volgende voorbeelden telkens de sterkste postconditie Q.

- $\{false\}$ `x := x + 1` $\{Q\}$
- $\{x^2 + y^2 = z^2\}$ `z := 0` $\{Q\}$
- $\{x = 2y\}$ `x := x + y; y := x + 3 * y` $\{Q\}$
- $\{x = 0\}$
  `IF x = y THEN x := 0 ELSE y := 0 FI`
  $\{Q\}$

4.3 De methode van inductieve asserties

Tot nu toe hebben we uitsluitend het correct bewijzen van “straight line” programma’s behandeld. In deze sectie wordt de calculus uitgebreid tot herhalingen. We geven een informele methode voor het bewijzen van terminatie. Dan keren we de zaak om en beschouwen het afleiden van een programma uit de asserties.

4.3.1 De herhaling

De regels voor de zwakste preconditie en de sterkste**semantiek herhaling** postconditie van een herhalende opdracht zijn tamelijk ingewikkeld. We geven daarom een eenvoudige regel, die het in vele gevallen uitstekend zal doen, voor de simpelste variant van de herhalende opdracht:

$$\texttt{WHILE b REP } \mathcal{S} \texttt{ ENDREP}$$

We gaan daartoe op zoek naar een **invariant**, d.w.z. een *invariante* relatie P, die bij elke doorgang van de lus — op een *vaste* plaats, en wel direct vóór de controle van de voorwaarde b — geldig is. Invariant beduidt hier dat de geldigheid van de relatie P door S niet verstoord wordt. Wanneer de relatie P dus aan het begin van de herhaling geldt, kunnen we concluderen dat deze relatie na afloop van de herhaling nog steeds geldt. Maar er is meer, immers, we weten dat de herhaling getermineerd is, hetgeen betekent dat $\neg b$ geldt. Dit leidt tot de volgende regel:

Stelling: Indien de relatie P de eigenschap heeft dat:

$$\{P \wedge b\}\ \mathcal{S}\ \{P\}$$

dan geldt:

$$\{P\}\ \texttt{WHILE b REP } \mathcal{S} \texttt{ ENDREP}\ \{P \wedge \neg b\}$$

Deze regel staat bekend als de *invariantie-stelling van Hoare*. Het is een kwestie van ervaring om bij een gegeven herhalende opdracht een geschikte invariant te bedenken.

VOORBEELD: We zoeken een invariant P bij de herhaling in het volgende fragment:

$\{x = 1\}$
`i := n;`
`WHILE i <> 0 REP x := x * a; i := i - 1 ENDREP`
$\{x = a^n\}$

Na enig nadenken nemen we voor P: $x = a^{n-i}$. De keuze is redelijk, omdat P na de initialisatie van de herhalende opdracht geldt: $\{x = 1\}$ `i := n` $\{x = a^{n-i}\}$, en omdat P een invariant is van de loop, zoals men eenvoudig na kan gaan. Na afloop van de herhalende opdracht geldt derhalve $P \wedge \neg i \neq 0$, hetgeen vereenvoudigd kan worden tot $x = a^{n-i} \wedge i = 0$, waaruit dan weer volgt $x = a^n$. En hiermee is de correctheid van dit fragment met betrekking tot de asserties aangetoond.

Bovenstaande redenering is echter nog te zwak. Dit zit hem in de zinsnede:

Na afloop van ...

Wie zegt dat er een *afloop* is? Laat de beginwaarde van n maar eens negatief zijn, dan is eenvoudig in te zien dat de herhaling niet termineert. Het resultaat van de herhaling is dan ongedefinieerd.

Een eerste reactie zal zijn: kies dan voor P: $x = a^{n-i} \wedge i \geq 0$. Uiteraard is ook deze relatie een invariant voor de herhalende opdracht. Maar het initieel waarmaken van de invariant levert nu problemen. We berekenen:

$$\mathcal{PRE}(\texttt{i := n}, x = a^{n-i} \wedge i \geq 0) = (x = 1) \wedge (n \geq 0)$$

. Wat we te kort komen als preconditie is derhalve $n \geq 0$. Maar dat gold eerder ook al.

Samenvattend zien we dat we als sterkste resultaat kunnen bewijzen:

$\{x = 1 \wedge n \geq 0\}$
`i := n;`
`WHILE i <> 0 REP x := x * a; i := i - 1 ENDREP`
$\{x = a^n\}$

Wat dus ontbreekt aan de invariantie-stelling van Hoare is de terminatie-problematiek. We beschouwen dit nader in de volgende paragraaf.

4.3.2 Terminatie

Voor het leveren van een **terminatie bewijs** bestaat geen sluitend stelsel van formele regels. Men kan zelfs bewijzen dat een dergelijk sluitend stelsel niet kan bestaan (omdat het zgn. *halting problem* onbeslisbaar is). Toch heeft de schrijver van de algoritme altijd wel een idee waarom de algoritme niet eindeloos voortgaat – het gaat erom dit te formaliseren tot een bewijs.

In het voorgaande is al aangegeven, dat een `WHILE`-loop moet eindigen: de preconditie moet hier sterk genoeg voor zijn. Alleen als een lus eindigt is zijn effect gedefinieerd. Maar hoe verifiëren we dat de preconditie sterk genoeg is om terminatie te garanderen?

Er zijn minstens twee situaties waarvoor de eindigheid expliciet moet worden aangetoond: iteratie en (in)directe recursie. In beide gevallen zal het bewijs in het algemeen als volgt verlopen: gezocht wordt een geheel-tallige expressie f van een of meer van de objecten (die tijdens de iteratie of recursie veranderen). Van die expressie f moet worden aangetoond dat

- de waarde in elke doorgang (geneste aanroep) kleiner wordt,
- deze niet kleiner kan worden dan een onveranderlijke ondergrens
- de waarde vóór de lus niet kleiner is dan die ondergrens

De geforceerde, in totaliteit echter beperkte groei van de expressie f garandeert dat het aantal herhalingen (geneste aanroepen) niet willekeurig kan groeien, maar beperkt is. En dat betekent terminatie van de algoritme. Grofweg kan men de expressie f zien als een indicatie van het aantal nog uit te voeren herhalingen (geneste aanroepen).

In het bovengegeven voorbeeld kunnen we de expressie $f = i$ beschouwen. In elke doorgang wordt i één kleiner. Maar i kan niet kleiner worden dan 0. De beginwaarde van i is n, zodat aan het begin van de herhaling geldt $i \geq 0$. De gevolgde redenering is naar [FLO67].

4.3.3 Voorbeeld: de Grootste Gemene Deler

De oudste van alle algoritmen voor de bepaling van de grootste gemene deler van twee natuurlijke getallen is die van Euclides. Eeuwenlang heette deze algoritme zelfs De Algorithme. Men vindt deze algoritme in bijna alle teksten over algoritmen en correctheid te vinden. Wij maken daarop geen uitzondering.

De algoritme luidt, in de vorm van een Elan procedure:

```
INT PROC ggd (INT a, b):
    INT VAR m :: a, n :: b;
    WHILE rest bij deling van m door n
is niet nul
    REP m := n; n := rest
    ENDREP;
    n.
rest bij deling van m door n is niet
nul:
    INT CONST rest :: m MOD n; rest <> 0
ENDPROC ggd
```

Eerst een getallenvoorbeeld:

```
5544 = 2548*2+448
       2548 = 448*5+308
              448 = 308*1+140
                    308 = 140*2+28
                          140 = 28*5+0
```

dus

```
    ggd(5544,2548) = 28
```

Het correctheidsbewijs berust op de overweging dat uit de betrekking $x = qy + r$ met $x, q, y, r \geq 0$, volgt dat alle gemeenschappelijke delers van x en y ook precies die van y en r zijn. Dan is ook de *grootste* gemene deler van x en y gelijk aan die van y en r. Dit is gemakkelijk in te zien. Laat eerst d een deler zijn van x èn y. Dan is d ook een deler van r, omdat $r = x - qy$. Omgekeerd, zij d een deler van y en r. Dan is d ook een deler van x, vanwege $x = qy + r$. Hieruit volgt de eigenschap:

$$ggd(m, n) = ggd(n, m \bmod n)$$

Als invariant bij de herhalende opdracht kiezen we:

$$ggd(m, n) = ggd(a, b) \wedge m > 0 \wedge n > 0$$

Deze veelgebruikte voorwaarde noteren we kortheidshalve in het vervolg als $R(m, n)$.

Eerst controleren we of de invariant wel sterk genoeg is. Na (eventuele) terminatie van de herhalende opdracht zouden we volgens de invariantiestelling van Hoare mogen concluderen:

$$R(m, n) \wedge \text{ rest bij deling van m door n is nul}$$

Dit laatste betekent dat m een n-voud is, zodat de grootste gemene deler van m en n uiteraard n is. Op grond van $R(m, n)$ is n dan de grootste gemene deler van a en b.

Vervolgens kijken we of $R(m, n)$ wel een invariant is van de herhalende opdracht. De volgende berekening is eenvoudig:

$$\mathcal{PRE}(\mathtt{m := n; n := rest}, R(m, n)) = R(n, rest)$$

Dat betekent dat we moeten aantonen dat:

$$R(m, n) \wedge rest \neq 0 \Rightarrow R(n, rest)$$

Door $m \bmod n$ voor $rest$ te substitueren leidt dit tot:

$$R(m,n) \wedge m \bmod n \neq 0 \Rightarrow R(m, n \bmod n)$$

Maar dat toonden we reeds eerder aan. De initialisatie en de terminatie tenslotte laten we als een aardigheidje voor de lezer.

4.4 Opgaven (2)

1. $\{x = p\}$
   `WHILE x > 0 REP x := x - 1 ENDREP`
   $\{Q\}$
2. $\{Q\}$
   `WHILE a > 0 REP a := d * e + f ENDREP`
   $\{x > y > 0 \wedge wx \geq 0\}$
3. (vermenigvuldigen door optellen) In het volgende programma-fragment zijn a en b niet-negatieve `INT`-waarden. Verder zijn x, y en z bekend als variabelen van het type `INT`. Bewijs dat het volgende programmafragment:
   ```
   z := 1; x := a; y := b;
   WHILE y > 0
   REP
     IF odd (y) THEN z := z + x FI;
     x := x + x; y := y DIV 2
   ENDREP
   ```
   de eindrelatie $z = a.b$ waar maakt. Doe dit door een geschikte invariant bij de herhalende opdracht te bedenken. Vergeet niet om de terminatie aan te tonen. U behoeft hierbij overigens geen rekening met overflow te houden.
4. (machtsverheffen) Veronderstel a en b niet-negatieve `INT`-waarden. Verder is z een variabele van het type `INT`. Ontwikkel een efficiënt programma dat de eindrelatie $x = a^b$ waarmaakt. Ook in deze opgave hoeft u niet met overflow rekening te houden.
5. Bewijs het volgende programma correct, of geef een tegenvoorbeeld. Neem aan dat alle voorkomende variabelen integer gedeclareerd zijn.
   $\{y \geq 0\}$
   ```
   p:= y;
   q:= 0;
   r:= x;
   WHILE p > 0
   REP
      IF p DIV 2 * 2 = p
      THEN p:= p - 1;
           q:= q + r
      ELSE
           p:= p DIV 2;
           r:= r * 2
      FI
   ENDREP
   ```
   $\{q = x * y\}$
6. Hetzelfde voor
   $\{x > 0\}$
   ```
   IF x DIV 2 * 2 = x
   THEN x:= x DIV 2
   ELSE x:= x * x
   FI
   ```
   $\{xDIV2 * 2 <> x\}$

Chapter 5

Correctheid (3)

In dit hoofdstuk gaan we nader in op de afleiding van herhalingsinvarianten en de omgang met pre- en post-specificaties. Met name behandelen we een inzichtelijke methode om herhalingsinvarianten in diagram-vorm weer te geven (de "hink-stap-sprong diagrammen").

5.1 Predicaten

Een goede gewoonte bij het geven van complexe bewijzen is het invoeren van predicaten, dat wil zeggen benoemde en eventueel van parameters voorziene uitspraken. We geven een (min of meer sprekende) naam aan een uitspraak die in meer dan één assertie zal optreden, en geven hem als parameters de namen van de variabelen waaraan in de algoritme wordt toegewezen. Het predicaat kan dan worden gelezen als een uitspraak over die variabelen.

We gebruikten deze methode al in 3.4. Een goed voorbeeld is te vinden in 4.2.3, waar $R(m,n)$ bij afspraak staat voor

$$R(m,n) =_{def} ggd(m,n) = ggd(a,b) \wedge m > 0 \wedge n > 0$$

waarbij `m` en `n` de variabelen uit de algoritme zijn.

Het bedoelde parametermechanisme berust op substitutie van de actuele parameter (eventueel tussen haakjes gezet) voor ieder voorkomen van de formele parameter; zo zal $R(x, y+z)$ staan voor de uitspraak

$$ggd(x, y+z) = ggd(a,b) \wedge x > 0 \wedge (y+z) > 0$$

Niet alleen maakt de invoering van predicaten de asserties korter, het gebruik van predicaten als abstractiemiddel voorkomt ook bepaalde slordigheidsfouten, omdat gedetailleerde uitdrukkingen niet meermalen hoeven te worden opgeschreven, en toont duidelijker de essentie van het correctheidsbewijs.

Als voorbeeld zou de eigenschap "de eerste `k` elementen van de rij `f` zijn geordend volgens de relatie `R`" kunnen worden geformuleerd met behulp van het predicaat

$$geordend(f,k,R) =_{def} \forall i,j \in 1..k[i < j \Rightarrow f_i \; R \; f_j]$$

Het feit dat predicaten een naam hebben maakt een recursieve definitie mogelijk, zoals

$$geordend(f,k,R) =_{def} k \leq 1 \nabla [geordend(f,k-1,R) \wedge f_{k-1} \; R \; f_k]$$

Het hier gebruikte nabla-teken ∇ staat voor de MacCarthy "or". Als de linker operand van de operator true levert slaagt deze zonder dat de rechter operand geëvalueerd wordt; die hoeft dus in dat geval geen welgedefinieerde waarde te hebben.

De recursieve formulering maakt in sommige situaties de details van het correctheidsbewijs eenvoudiger, met name in conjunctie met een recursieve algoritme.

Predicaten zijn goed te combineren met de Floyd-Hoare calculus. Het is met enige voorzichtigheid mogelijk gespecialiseerde regels te verzinnen, zoals de volgende varianten van de achterwaartse regel:

$$\mathcal{PRE}(\texttt{x := E}, P(x)) = P(E)$$

$$\mathcal{PRE}(\texttt{x := E}, P(f(x))) = P(f(E))$$

mits bij deze laatste regel x niet in P voorkomt.

5.2 Redeneren over iteratie

Het is niet eenvoudig om binnen het formalisme van Floyd en Hoare te redeneren over herhalingen, te meer omdat die meestal iets met rijen te maken hebben. In deze sectie zullen we daarom een aanschouwelijke en toch redelijk formele manier behandelen om herhalingsinvarianten, ook over rijen, te vinden en te beschrijven.

We zullen invarianten over rijen noteren als *diagrammen* waarbij de elementen van de rij worden weergegeven door een horizontale streep, ingedeeld in segmenten, gemarkeerd met de namen van variabelen die als waarde de index van een bepaald element van de rij bevatten, en versierd met uitpraken over deze en andere variabelen. Zulke plaatjes worden in de informatica al tientallen jaren gebruikt.

5.3 Correctheid van Selection Sort

Als voorbeeld stellen we een correctheidsbewijs op voor Selection Sort. Laat gegeven zijn een globale rij `f` bestaande uit `n` elementen van het type `EL`, voor welk type een ordeningsoperatie `<=` bestaat. De rij moet in situ worden gesorteerd.

5.3.1 Het vinden van de invariant

De grondgedachte achter de algoritme is de indeling van de te sorteren rij in een reeds gesorteerd gedeelte, gevolgd door een nog te sorteren deel waarbij ieder van de elementen in het te sorteren gedeelte niet kleiner is dan ieder element uit het reeds gesorteerde. De rij moet op ieder moment een permutatie bevatten van de oorspronkelijke rij. We kunnen dit weergeven in een plaatje dat we $STAP$ zullen noemen:

$STAP$

```
|1                   i-1|i                     n|
|-----------------------|-----------------------|
 |                     | |                      |
 |----- geordend -----| |---- minstens --------|
 |                                              |
 |------------------- perm --------------------|
```

Het plaatje is bedoeld als equivalent met de uitspraak

$$perm \wedge 1 \leq i \leq n \wedge geordend(i-1) \wedge minstens(i,n)$$

waarbij de volgende predicaten zijn gebruikt:

$$\begin{aligned} perm =_{def} &\quad \text{“}f \text{ bevat een permutatie} \\ &\quad \text{van de oorspronkelijke rij } f\text{'”} \\ geordend(a) =_{def} &\quad \forall j,k \in 1..a[j < k \Rightarrow f_j \leq f_k \\ minstens(a,b) =_{def} &\quad \forall j \in 1..a-1, k \in a..b[f_j \leq f_k] \end{aligned}$$

Merk op dat we (bijvoorbeeld bij *perm*) de constanten 1 en n kortheidshalve niet tot parameters hebben gemaakt, evenmin als de (globale) rij f.

Het feit dat het tweede deel ongesorteerd is hebben we niet uitgedrukt, probeer het zelf maar eens: het levert een uitspraak op die altijd waar is en die we dus net zo goed kunnen weglaten.

Aan dit predicaat, dat in het bewijs als herhalingsinvariant zal optreden, geven we de naam $STAP$.

$STAP =_{def}$
$perm \wedge 1 \leq i \leq n \wedge geordend(i-1) \wedge minstens(i,n)$

5.3.2 Het begin

Aan het begin van de algoritme weten (of veronderstellen) we heel weinig:

$HINK$

```
|1                                              n|
|------------------------------------------------|
 |                                               |
 |---------------- perm ----------------------|
```

wat overeenkomt met

$$HINK =_{def} 1 \leq n \wedge perm$$

Uitgaande van de in het plaatje beschreven situatie moet een variable i worden gedeclareerd en zodanig geïnitialiseerd dat de uitspraak $STAP$ waar is. Dat kan eenvoudig door het gesorteerde gebied leeg te kiezen. Dit is het geval als $i = 1$,

$$HINK \wedge i = 1 \Rightarrow STAP$$

De algoritme gaat door totdat het tweede deel van de rij nog slechts uit 1 element bestaat.

$BIJNA$

```
|                                             n|
|1                                       n-1|i|
|-------------------------------------------|-|
 |                                          | |
 |------------ geordend -------------------| |
 |                                            |
 |--------------- perm ----------------------|
                                              |
                                       minstens
```

$BIJNA =_{def}$
$1 \leq i = n \wedge perm \wedge geordend(n-1) \wedge minstens(n,n)$

Maar dan hoort ook het laatste element (dat immers niet kleiner is dan zijn voorgangers) tot het gesorteerde deel, dus

$$BIJNA \Rightarrow geordend(n)$$

wat precies de situatie oplevert die we willen bereiken:

$SPRONG$

```
|1                                              n|
|------------------------------------------------|
 |                                               |
 |-------------- geordend --------------------|
 |                                               |
 |---------------- perm ----------------------|
```

Misschien begrijpt U nu de wat bizarre naamsgeving van de predicaten: we willen van $HINK$ via vele malen $STAP$ naar $SPRONG$. Daarbij zullen we overigens ook de situatie $BIJNA$ tegenkomen.

In navolging van W.A. Vervoort[VERV92] zullen we ze **hink-stap-sprong diagrammen** noemen.

5.3.3 De wortel van de algoritme

We formuleren nu de Selection Sort algoritme en brengen op strategische plaatsen invarianten aan, met $STAP$ als herhalings-invariant. Door middel van commentaren tussen #-tekens verklaren we de overgangen tussen predicaten.

```
selection sort:
  {HINK}
  eerst is het gesorteerde deel leeg;
  {STAP}
  WHILE i < n
  REP {STAP, i < n}
     pak het kleinste uit het
ongesorteerde deel;
     plak dit achteraan het gesorteerde
deel
      {STAP}
  ENDREP
  {STAP, i >= n}
  # STAP impliceert i <= n #
  {STAP, i = n}
  # eigenschap van BIJNA #
  {BIJNA}
  # tja, dat zullen we moeten bewijzen #
  {SPRONG} .
```

In grote lijnen staat hier al het bewijs, alleen in het midden en aan het einde zijn nog onbewezen overgangen.

5.3.4 Het begin

De eerste invariant maken we waar door initialisatie, zoals boven beschreven.

```
eerst is het gesorteerde deel leeg:
  {HINK} .
  INT VAR i :: 1
  {STAP} .
```

5.3.5 Het slot

De overgang aan het einde van de algoritme lijkt simpel, maar is niet te maken door eenvoudigweg toepassen van de calculus. Het bewijs van de implicatie

$$BIJNA \Rightarrow SPRONG$$

is met wat eenvoudige wiskundige kennis te leveren, maar vergt toch wel enig lastig priegelwerk (zie figuur 5.3.5).

Het kost heel wat werk om de zo evidente intuïtie

> als het laatste element van de rij minstens zo groot is als alle voorafgaande en alle voorafgaande zijn opstijgend geordend dan is de hele rij opstijgend geordend

te bewijzen. Het zou handig zijn als een computerondersteund bewijssysteem dit soort mensonwaardig gepriegel voor zijn rekening zou nemen, zodat wij mensen ons kunnen beperken tot het aanleveren van de creativiteit, in de vorm van asserties en invarianten.

5.3.6 De romp van de herhaling

De romp van de algoritme bevat de nog onbewezen overgang

```
   {STAP, i < n}
   pak het kleinste uit het
ongesorteerde deel;
   plak dit achteraan het gesorteerde
deel
   {STAP} .
```

waarbij de eerste verfijning op zijn beurt een ingewikkelde herhaling zal bevatten. We maken opnieuw hink-stap-sprong plaatjes, om op deze herhaling greep te krijgen. Om de predicaten te onderscheiden van de vorige zullen we een Z (van zoeken) achter hun naam plakken.

We zijn op zoek naar het kleinste element uit het ongesorteerde deel (dat daarom niet leeg mag zijn); noem zijn index k. Onderweg hebben we steeds te maken met de kleinste uit een linkerkant daarvan, begrensd door de index j. In een diagram kunnen we de bedoelde invariant als volgt beschrijven:

$STAPZ$

```
|1           |i      k      j              n|
|------------|---------------|---------------|
 |           |       |       |              |
 |           |-- kleinste --|               |
 |                                          |
 |------------------ STAP, i<n --------------|
```

Oftewel $STAPZ =_{def}$

$$STAP \wedge i \leq k \leq j \leq n \wedge i < n \wedge kleinste(k,i,j)$$

met

$$kleinste(k,i,j) =_{def} \forall m \in i..j[f_m \leq f_k]$$

Weer weten we aan het begin van de herhaling heel weinig.

$$HINKZ =_{def} STAP \wedge i < n$$

$HINKZ$

```
|1        |i|                                 n|
|---------|-|----------------------------------|
 |                                            |
 |---------------- STAP, i<n -----------------|
```

Bijwijze van initialisatie kiezen we de linkerkant 1 element lang, want

$$HINKZ \wedge i = k = j \Rightarrow STAPZ$$

Aan het einde van de herhaling dient k de index te zijn van het kleinste element uit het hele rechterdeel, dus

$SPRONGZ$

```
|1        |i      k
n|
|---------|--------------------------------------|
 |         |      |                              |
 |         |---- kleinste ----------------------|
 |                                              |
 |------------------ STAP, i<n ----------------|
```

$$SPRONGZ =_{def} STAP \wedge i < n \wedge kleinste(k,i,n)$$

5.3.7 Zoeken naar het kleinste element

We werken nu het zoeken naar de kleinste uit. De daarbij optredende herhaling markeren we met de asserties verkregen uit de hink-stap-sprong invarianten.

$$BIJNA$$

= (uitvouwen van de definitie van $BIJNA$)

$$perm \wedge 1 \leq i = n \wedge geordend(i-1) \wedge minstens(i,n)$$

= (eliminatie van i, vereenvoudigen)

$$perm \wedge 1 \leq n \wedge geordend(n-1) \wedge minstens(n,n)$$

= (uitvouwen van de definitie van minstens)

$$perm \wedge 1 \leq n \wedge geordend(n-1) \wedge \forall j \in 1..n-1, k \in n..n[f_j \leq f_k]$$

= (uitvouwen van de definitie van geordend)

$$perm \wedge 1 \leq n \wedge \forall j, k \in 1..n-1[j < k \Rightarrow f_j \leq f_k] \wedge \forall j \in 1..n-1, k \in n..n[f_j \leq f_k]$$

= (quantificatie over j buiten haakjes halen)

$$perm \wedge 1 \leq n \wedge \forall j \in 1..n-1\,[\forall k \in 1..n-1[j < k \Rightarrow f_j \leq f_k] \wedge \forall k \in n..n[f_j \leq f_k]]$$

= (j < k in laatste quantificatie, samennemen gebieden)

$$perm \wedge 1 \leq n \wedge \forall j \in 1..n-1\,[\forall k \in 1..n[j < k \Rightarrow f_j \leq f_k]]$$

= (uibreiden domein van j)

$$perm \wedge 1 \leq n \wedge \forall j \in 1..n\,[\forall k \in 1..n[j < k \Rightarrow f_j \leq f_k]]$$

= (quantoren samennemen)

$$perm \wedge 1 \leq n \wedge \forall j, k \in 1..n[j < k \Rightarrow f_j \leq f_k]]$$

= (definitie van geordend)

$$perm \wedge 1 \leq n \wedge geordend(n)$$

= (definitie van $SPRONG$)

$$SPRONG$$

Figure 5.1: Van BIJNA naar SPRONG (bewijs)

```
pak het kleinste uit het ongesorteerde
deel:
   {STAP, i < n}
   {HINKZ}
  INT VAR j :: i, k :: i;
   {HINKZ, j = i, k = i}
   {STAPZ}
  WHILE j < n
  REP
     {STAPZ, j < n}
     j:= j + 1;
     IF f[j] < f[k]
     THEN
          k := j
     FI
     {STAPZ, j ≤ n}
  ENDREP
   {STAPZ, j = n}
   {SPRONGZ} .
```

5.3.8 De binnenste herhalingsromp

Nu bewijzen we de partiële correctheid van de romp van de binnenste herhaling.

$\{STAPZ, j < n\}$
uitvouwen van STAPZ, vereenvoudigen
$\{STAP, 1 \leq i \leq k < j < n, kleinste(k, i, j)\}$
j:= j + 1;
$\{STAP, 1 \leq i \leq k < j \leq n, kleinste(k, i, j-1)\}$
IF f[j] < f[k]
THEN
$\{STAP, 1 \leq i \leq k < j \leq n, kleinste(k, i, j-1), f_j < f_k\}$
$\{STAP, 1 \leq i \leq k < j \leq n, kleinste(j, i, j)\}$
k := j;
$\{STAP, 1 \leq i \leq k = j \leq n, kleinste(k, i, j)\}$
$\{STAPZ\}$
ELSE
$\{STAP, 1 \leq i \leq k < j \leq n, kleinste(k, i, j-1), f_j \geq f_k\}$
$\{STAP, 1 \leq i \leq k < j \leq n, kleinste(k, i, j)\}$
$\{STAPZ\}$
FI
$\{STAPZ\}$

5.3.9 Uitbreiden van het gesorteerde deel

In de algoritme is nog nergens aan i toegewezen, zodat $STAP$ invariant bleef. Nu moeten we echter i gaan ophogen.

plak dit achteraan het gesorteerde deel:
$\{SPRONGZ\}$
uitvouwen, vereenvoudigen
$\{STAP, 1 \leq i \leq k \leq n, i < n, kleinste(k, i, n)\}$
uitvouwen STAP
$\{perm, geordend(i-1), nietkleiner(i, n), 1 \leq i \leq k \leq n, i < n, kleinste(k, i, n)\}$
verwissel (f[i], f[k]);
$\{perm, geordend(i-1), nietkleiner(i, n), 1 \leq i \leq k \leq n, i < n, kleinste(i, i, n)\}$
k elimineren, nadenken
$\{perm, geordend(i), nietkleiner(i, n), 1 \leq i < n\}$
i:= i + 1

$\{perm, geordend(i-1), nietkleiner(i-1, n), 1 < i \leq n\}$
verzwakken
$\{perm, geordend(i-1), nietkleiner(i, n), 1 \leq i \leq n\}$
#definitie van STAP #
$\{STAP\}$.

5.4 Opgaven (1)

1. Bewijs de partiële correctheid van Selection Sort ook eens met de recursieve definitie van *geordend* uit 5.1. Welke definitie is gemakkelijker toe te passen?
2. Maak hink-stap-sprong diagrammen voor Insertion Sort (8.6.2 van Deel I).
3. Maak hink-stap-sprong diagrammen voor Bubble Sort (8.6.3 van Deel I).
4. Maak hink-stap-sprong diagrammen voor Merge Sort (14.1 van Deel I).
5. Bewijs de partiële correctheid van de Halveringsalgorite (8.7 van Deel I) met behulp van hink-stap-sprong diagrammen.
6. Bewijs de partiële correctheid van de splitsingsmethode volgens Hoare (14.2.1 van Deel I) met behulp van hink-stap-sprong diagrammen.
7. Bewijs de partiële correctheid van de splitsingsmethode volgens Dijkstra (14.2.3 van Deel I).

5.5 Procedures

Met procedures kan in principe zo te werk gegaan worden als hiervoor met verfijningen, namelijk door de preconditie van de romp naadloos te laten aansluiten bij de assertie voorafgaand aan de oproep, en de postconditie bij de assertie volgend op de oproep, maar dit leidt tot een weinig flexibele situatie: Het typische van procedures (in tegenstelling tot verfijningen) is juist dat ze op meer plaatsen, en dus in verschillende contexten kunnen worden opgeroepen. Neem als voorbeeld een procedure als `incr`: zijn parameter wordt met 1 opgehoogd, maar welke variabele dat is hangt af van de oproep.

Nu is de precieze inbedding van procedures in de Floyd-Hoare calculus een ingewikkelde technische geschiedenis, vanwege de omgang met parameters en met recursie, een prachtig onderwerp voor een later specialisatiecollege. We zullen echter een aardige benadering daarvan behandelen, waarbij we ons beperken tot partiële correctheid (terminatiebewijzen worden separaat geleverd) en nogal pragmatisch omgaan met parameters. We behandelen daartoe de specificatie van procedures doormiddel van **pre/post specificaties**. Deze betrekkelijk formele wijze van specificeren is goed te combineren met de (in een later college in te voeren) objectgeoriënteerde ontwerpmethode, en heeft zelfs bij de industrie ingang gevonden.

5.5.1 Pre/post specificaties

Onder de **betekenis** ("semantiek") van een procedure verstaan we de combinatie van

1. het **effect** van de uitvoering daarvan (op de waarden van globale variabelen, maar ook op de inhoud van bestanden, beeldscherm en andere physieke variabelen in de omgeving), en
2. het **resultaat** van de uitvoering daarvan (de waarde die wordt berekend),

beide afhangend van de waarden van de bij de oproep van de procedure meegegeven parameters en eventueel van globale objecten.

Om die semantiek te specificeren zetten we aan het begin van de romp van de procedure een invariant neer, de **pre-conditie**, die aangeeft wat de procedure vooronderstelt over zijn parameters en globale objecten (variabelen en constanten). Bij de uitgang van de procedure noteren we de **post-conditie** die aangeeft wat daar zal gelden met betrekking tot parameters, globale objecten en het eventuele resultaat. Als dat resultaat niet is gebonden aan een objectnaam kunnen we het denoteren als een hypothetisch object $result$.

Vaak kan de preconditie leeg zijn ($true$); is de preconditie niet leeg dan hebben we te maken met een partiële operatie. Soms moet in de postconditie worden verwezen naar de waarde die een variabele voor de uitvoering van de romp had; we zullen dan de oorspronkelijke waarde aanduiden met een accent achter de variabele-naam, net als bij de voorwaartse regel voor de toewijzing. Voorbeelden:

- ```
  PROC incr (INT VAR x):
    {PRE : true}
    x:= x + 1
    {POST : x = x' + 1}
  ENDPROC incr;
  ```
  Merk op dat we zoals gebruikelijk niet hebben aangegeven dat `x` een welgedefinieerde waarde moet hebben, en geen rekening hebben gehouden met overflow.

- ```
  REAL PROC sqrt (REAL x):
    {PRE : x ≥ 0}
    bla bla bla
    {POST : result² = x, result ≥ 0}
  ENDPROC sqrt;
  ```

- ```
  INT PROC fac (INT n):
    {PRE : n > 0}
    IF n = 1 THEN 1 ELSE n * fac (n - 1)
  FI
    {POST : result = n!}
  ENDPROC fac;
  ```

5.5.2 Correctheid van de romp

Om de partiële correctheid van de romp van een procedure ten opzichte van zijn specificatie te bewijzen wordt deze, op de gebruikelijke wijze, versierd met asserties, waarbij de overgangen tussen de asserties moeten voldoen aan de Floyd-Hoare calculus. Daarbij is het een goede techniek, voor iedere formele of globale variabele `v` van de procedure de preconditie uit te breiden met een uitspraak $v' = v$, die initiëel zeker geldt en waarin in de loop van de romp verandering kan komen. Zo is de procedure `incr` eenvoudig correct te bewijzen, gebruikmakend (na enige voorbereiding) van de voorwaartse regel:

```
PROC incr (INT VAR x):
  {PRE : true}
  {x = x'}
  {x + 1 = x' + 1}
  x:= x + 1
  {POST : x = x' + 1}
ENDPROC incr;
```

Voorbeeld: faculteitsfunctie

Dat de methode met enig fröbelen ook werkt voor recursieve procedures laten we zien aan de faculteitsfunctie. Om het bewijs te vereenvoudigen transformeren we de functie eerst zodanig dat de waarde van de recursieve oproep benoemd wordt, door introductie van de constante `nf`:

```
INT PROC fac (INT n):
  {PRE : n > 0}
  IF n = 1
  THEN 1
  ELSE
      INT CONST nf :: fac (n - 1);
      n * nf
  FI
  {POST : result = n!}
ENDPROC fac;
```

Merk op dat deze transformatie de semantiek van de procedure onverlet laat (de collaterale verwerking van de operanden van de vermenigvuldiging wordt weliswaar geserialiseerd, maar dat verandert het resultaat niet).

```
INT PROC fac (INT n):
  {PRE : n > 0}
  IF n = 1
  THEN {n = 1} 1
       {n = 1, result = 1}
       # kennis faculteitsfunctie #
       {n = 1, result = 1!}
       # substitutie, verzwakken #
       {result = n!}
  ELSE {n > 1}
       {n - 1 > 0}
       # voldoet aan preconditie fac #
       INT CONST nf :: fac (n - 1);
       {nf = (n - 1)!}
       n * nf
       {nf = (n - 1)!, result = n * nf}
       # kennis fac.functie #
       {result = n!}
  FI
  {POST : result = n!}
ENDPROC fac;
```

De romp van de getransformeerde functie is (partiëel) correct, dus geldt dit ook voor de, daarmee equivalente, oorspronkelijke functie. Terminatie valt makkelijk te bewijzen door inductie naar de recursie-diepte.
```

5.5.3 Correctheid van de oproep

We willen nu in staat zijn de partiële correctheid van een procedure-oproep in een bepaalde context te bewijzen.

$$\{P\}\ \mathtt{p}(\mathcal{A})\ \{Q\}$$

Gemakshalve nemen we aan dat de procedure een actie is (geen resultaat aflevert, anders dan via zijn parameters); functies kunnen worden behandeld als een actie met een extra uitvoer-parameter.

We gebruiken de volgende afkortingen:

- $\mathcal{A}$ de lijst van actuele parameters, meegegeven bij de oproep,
- $\mathcal{F}$ de lijst van daarmee corresponderende formele parameters in de kop van `p`,
- $\mathcal{FV}$ de lijst van de formele variabelen ($\mathcal{F}$ met uitzondering van de formele constanten), en
- $\mathcal{AV}$ de lijst van de daarmee corresponderende actuele variabelen,
- $\mathcal{GV}$ de lijst van globale variabelen die in de romp van `p` voorkomen.

De correctheid van de oproep `p (A)` in de context

$$\{P\}\ \mathtt{p}(\mathcal{A})\ \{Q\}$$

vereist twee dingen:

1. De oproep moet met deze actuele parameters voldoen aan de preconditie van `p`, dwz
$$P \Rightarrow [\ pre_p\ ]_{\mathcal{A}}^{\mathcal{F}}$$
2. De postconditie van `p` voldoet, tezamen met wat er nog geldig is van P, aan Q.
$$\Big[\ [\ P\ ]_{\mathcal{FV}'}^{\mathcal{FV}}\ \Big]_{\mathcal{GV}'}^{\mathcal{GV}} \wedge [\ post_p\ ]_{\mathcal{A}}^{\mathcal{F}} \Rightarrow Q$$

Deze regels zien er moeizaam uit, al zijn ze eigenlijk vanzelfsprekend.

Ze kunnen wat eenvoudiger worden als we alle hier optredende asserties expliciet parametriseren met de parameters waarvoor gesubstitueerd wordt. We herschrijven daartoe de pre- en postconditie van de procedure als volgt:

```
PROC p (𝓕):
  {PRE : pre_p(𝓕)}
  romp van p
  {POST : post_p(𝓕𝓥, 𝓖𝓥)}
ENDPROC p;
```

We splitsen verder de preconditie P in twee delen door alle daarin optredende uitspraken over de variabelen $\mathcal{FV}$ en $\mathcal{GV}$ apart te zetten:

$$P \equiv P1 \wedge P2(FV, GV)$$

waarbij $P1$ geen voorkomen van FV of GV bevat. Dan moeten we verifieren:

1. $P \Rightarrow pre_p(\mathcal{A})$
2. $[P1 \wedge P2(\mathcal{FV}', \mathcal{GV}') \wedge post_p(\mathcal{A}) \Rightarrow\ \{Q\}$

Het volgende voorbeeld zal dit moeten verduidelijken.

5.5.4 Voorbeeld: alweer een sorteerroutine

We bekijken een oproep `sort (f, n)` van een procedure

```
PROC sort (ROW max EL VAR f, INT n):
```

met

$$pre_{sort}(f,n) = \ \{1 \leq n \leq max\}$$

$$post_{sort}(f,n) = ordered(f,n) \wedge perm(f,f')$$

In de context

$\{\forall i \in 1..100\,[meting[i] > 0]\,, sum > 0\}$
`sort(meting, 100)`
$\{Q\}$

willen we Q bepalen.

We stellen vast

- $\mathcal{A} = \{\mathtt{meting,\ 100}\}$
- $\mathcal{F} = \{\mathtt{f,\ n}\}$
- $\mathcal{FV} = \{\mathtt{f}\}$
- $\mathcal{AV} = \{\mathtt{meting}\}$
- $\mathcal{GV} = \emptyset$

Merk op dat max een globale constante is met $max = |f|$. We introduceren een predicaat

$$positive(x) = \forall i \in 1..100[x_i > 0]$$

en schrijven de preconditie van de oproep als

$$\{positive(meting), max = 100, sum > 0\}$$

waarbij $max = 100 \wedge sum > 0$ de rol speelt van $P1$ en $positive$ die van $P2$.

We moeten eerst bewijzen
$max = 100 \wedge sum > 0 \wedge positive(meting)$
$\Rightarrow pre_{sort}(meting, 100)$
Daaraan is voldaan omdat $1 \leq 100 \leq max$.

Dan mogen we concluderen

$$max = 100 \wedge sum > 0 \wedge positive(meting') \wedge perm(meting', meting) \wedge ordered(meting, n)$$

wat we met behulp van een apart te bewijzen lemma

$$positive(meting') \wedge perm(meting', meting) \Rightarrow positive(meting)$$

kunnen herleiden tot

$$max = 100 \wedge sum > 0 \wedge positive(meting) \wedge ordered(meting, n)$$

Een heleboel werk om een voor de hand liggend resultaat te bereiken!

5.5.5 Een vuistregel

Gelukkig hebben we niet altijd dit hele geweld nodig: heeft de procedure geen zijeffecten op globale variabelen, en zijn de eventuele variabele parameters uitsluitend uitvoerparameters (hun waarde vóór de oproep doet dus niet ter zake), dan zijn $\mathcal{FV}$

en $\mathcal{GV}$ leeg, en wordt de resulterende postconditie veel simpeler:

$$P1 \wedge post_p(\mathcal{A}) \equiv \{Q\}$$

Een reden te meer om variabelen zoveel mogelijk te vermijden en te programmeren in een functionele stijl.

Een toepassing van deze vuistregel zagen we al in 5.5.2:

$\{n-1>0\}$
```
#voldoet aan preconditie fac#
INT CONST nf :: fac (n - 1);
```
$\{nf=(n-1)!\}$

wat strikt genomen had moeten luiden

$\{n-1>0\}$
```
# voldoet aan preconditie fac#
INT CONST nf :: fac (n - 1);
```
$\{n-1>0, nf=(n-1)!\}$
```
# verzwakking#
```
$\{nf=(n-1)!\}$

Merk op dat de assertie $\{n-1>0\}$ door de aanroep van `fac` niet wordt ontkracht.

5.6 Opgaven (2)

1. Geef voor de volgende procedure een pre- en postconditie.
```
INT PROC log2 (INT x):
  INT VAR y :: 2;
  WHILE y <= x
  REP y:= y + y
  ENDREP;
  y
ENDPROC log2;
```
Bewijs vervolgens de correctheid van de romp van de procedure.

2. Doe hetzelfde met de procedure
```
INT PROC kwa (INT n):
  INT VAR a :: 0, b:: 0;
  UPTO n
  REP b:= b + 1;
      a:= a + b
  ENDREP;
  b
ENDPROC kwa;
```

3. Schrijf een procedure
```
INT PROC max (ROW n REAL f, INT n):
```
die de index berekent van het grootste element van `f` met index niet groter dan `n`. Geef daarbij pre- en postcondities.

4. Verzin pre- en postcondities voor de procedure
```
PROC verwissel (REAL VAR x, y):
  REAL CONST z :: x;
  x:= y; y:= z
ENDPROC verwissel;
```

5. Bewijs de correctheid van de volgende procedure, gebruikmakend van de procedures uit de vorige twee opgaven.
```
PROC selection sort (ROW n REAL VAR f,
INT n):
```
$\{n \leq |f|\}$
```
   INT VAR i :: n;
   WHILE i > 1
   REP INT CONST im :: max (f, i - 1);
       verwissel (f[i], f[im]);
       i:= i - 1
   ENDREP;
```
$\{ordered(f), perm(f, f')\}$
```
ENDPROC selection sort;
```
waarbij *ordered* en *perm* in de vorige sectie zijn gedefinieerd. Voer zonodig hulp-predicaten in, bewijs eventuele overgangen die niet simpelweg met de Floyd-Hoare calculus bewezen kunnen worden apart.

6. Schrijf een procedure `bubble sort` (zie deel I, hoofdstuk 8) met dezelfde specificaties, bewijs zijn correctheid.

7. Idem voor `insertion sort`.

Chapter 6

Abstracte typen(1)

6.1 Voorbeeld: complexe getallen

Een voorbeeld van een type met wiskundig karakter is dat van de complexe getallen, waarvan we een realisering met de naam `COMPL` willen geven.

Bij de representatie van het domein kunnen we kiezen tussen een cartesische representatie (Fig. 6.1) als een paar reële getallen, voorstellend respectievelijk het reële en het imaginaire deel van een complex getal

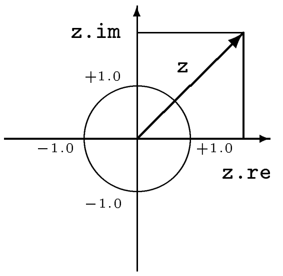


Figure 6.1: Cartesische representatie

```
LET COMPL = STRUCT (REAL re, im);
```

en een representatie in poolcoördinaten (Fig. 6.2), met als realisering

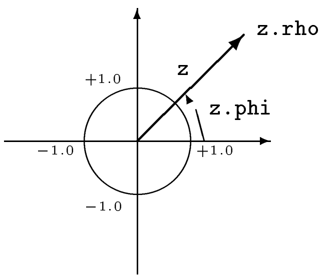


Figure 6.2: Poolcoördinaten

```
LET COMPL = STRUCT (REAL rho, phi);
```

We prefereren (voorlopig) de eerste representatie, omdat daarin de verschillende operaties wat gemakkelijker zijn uit te voeren. We zullen het complexe getal (x, y) denoteren als `x I y`, met behulp van een dyadische *denotatie operator* `I`:

```
COMPL OP I (REAL x, y):
  [x, y]
ENDOP;
```

Voor het selecteren van het reële en imaginaire deel van een complexe waarde voeren we ieder een monadische *acces operator* in:

```
REAL OP IM (COMPL z):
  z.im
ENDOP;

REAL OP RE (COMPL z):
  z.re
ENDOP;
```

Voor de omrekening naar poolcoördinaten voeren we twee operaties in die de absolute waarde $\mid z \mid$ en de hoek ϕ berekenen.

```
REAL OP ABS (COMPL z):
  sqrt ( RE z * RE z + IM z * IM z )
ENDOP ABS;

INT OP PHI (COMPL z):
  round (arctan (IM z / RE z) / pi *
180.0)
ENDOP PHI;
```

Merk op dat we hier eventueel maatregelen moeten nemen om delen door nul te voorkomen.

We definiëren vooraf enige veelgevraagde constanten:

```
   COMPL CONST complex 1 :: 1.0 I 0.0,
               complex 0 :: 0.0 I 0.0,
               complex i :: 0.0 I 1.0;
```

In termen van bovengenoemde operatoren voeren we nu één voor één de complexe operatoren in, te beginnen met de monadische.

```
COMPL OP - (COMPL u):
  - RE u I - IM u
ENDOP -;

COMPL OP + (COMPL u):
  u
ENDOP +;

COMPL OP CONJ (COMPL u):
  RE u I - IM u
ENDOP CONJ;
```

Merk opdat we hier geen enkel gebruik meer maken van de representatie van het domein! Ook bij de volgende declaraties zullen we daar strikt voor waken.

```
COMPL OP + (COMPL u, v):
  (RE u + RE v) I (IM u + IM v)
ENDOP +;

COMPL OP - (COMPL u, v):
  (RE u - RE v) I (IM u - IM v)
ENDOP -;

COMPL OP * (COMPL u, v):
  (RE u * RE v - IM u * IM v) I
  (RE u * IM v + IM u * RE v)
ENDOP *;
```

De deling nemen we over uit een leerboek over de complexe getallen (bijvoorbeeld [CHU60]):

```
COMPL OP / (COMPL u, v):
  REAL CONST d :: RE (v * CONJ v);
  COMPL CONST n :: u * CONJ v;
  RE n / d I IM n / d
ENDOP /;
```

Ook voeren we een speciale monadische operator in voor de reciproke waarde $\frac{1}{v}$, verkregen uit de vorige door substitutie van 1.0 voor `u` en vereenvoudiging:

```
COMPL OP RECIP (COMPL v):
  COMPL CONST n :: CONJ v;
  REAL CONST d :: RE (v * n);
  RE n / d I IM n / d
ENDOP RECIP;
```

Een invoer-algoritme voor complexe getallen zullen we niet introduceren, wel een eenvoudige uitvoer-algoritme.

```
PROC put (COMPL  z):
  put (" (");
  put (RE z);
  put (IM z);
  put (")")
ENDPROC put;
```

Het domein van de complexe getallen is niet volledig geordend, het kent slechts twee vergelijkings-operatoren `=` en `<>`, die bovendien van weinig nut zijn omdat ze op de grillen van de REAL-arithmetiek zijn gebaseerd.

```
BOOL OP = (COMPL  u, v):
  IF RE u = RE v
  THEN IM u = IM v
  ELSE FALSE
  FI
ENDOP =;

BOOL OP <> (COMPL  u, v):
  IF RE u = RE v
  THEN FALSE
  ELSE IM u <> IM v
  FI
ENDOP <>;
```

We willen ook een operatie invoeren voor de conversie van `REAL` naar `COMPL`, die aan een reeël getal een maginaire component nul toevoegt. Naar analogie met andere conversieoperaties als `int`, `real` en `text` kiezen we hiervoor een monadische procedure met de naam `compl` (in kleine letters).

```
COMPL PROC compl (REAL r):
  r I 0.0
ENDPROC compl;
```

6.1.1 Een andere representatie

Mochten we nu alsnog tot een representatie in poolcoördinaten willen overgaan dan hoeven we alleen denotatie en acces aan te passen:

```
LET COMPL = STRUCT (REAL rho, phi);

COMPL OP I (REAL a,b):
  [sqrt (a*a + b*b), arctan (a/b)]
ENDOP;

REAL OP IM (COMPL u):
  u.rho * cos (u.phi)
ENDOP;

REAL OP RE (COMPL u):
  u.rho * sin (u.phi)
ENDOP;
```

Alle andere operaties blijven correct werken aangezien we de **representatie onafhankelijkheid** zo groot mogelijk hebben gehouden. Wel zullen met deze representatie de meeste operaties wat trager worden. De operatoren `ABS` en `PHI` kunnen nu juist eenvoudiger:

```
REAL OP ABS (COMPL  u):
  u.rho
ENDOP;

REAL OP PHI (COMPL  u):
  u.phi
ENDOP;
```

maar dat gaat wel ten koste van de representatie-onafhankelijkheid.

6.2 Signatuur van een type

Onder de **signatuur** van een type verstaan we de verzameling van alle operaties die een argument (parameter dan wel operand) of een resultaat hebben van dat type. Onder een **operatie** verstaan we naast een algoritme (gedefinieerd als procedure, operator of eventueel verfijning) ook een constante, variabele of denotatie — we abstraheren op die manier van de concrete taalmiddelen van Elan.

We hebben nu een aantal operaties ingevoerd op complexe getallen, die komen te staan naast de vele (vaak gelijknamige) al eerder ingevoerde operaties op andere typen. Hoe kunnen we het overzicht behouden over de operaties behorend bij een type, anders dan door een opsomming?

Een sprekende representatie van de signatuur van een of meer typen is een **netwerk van typen en operaties** (**NTO** [CRA87]). We geven daarbij een type weer als een cirkel of ovaal, met daarin de naam van het type geschreven, en een operatie als een rechthoekje, met daarin de naam daarvan. Deze rondjes en rechthoekjes verbinden we met pijlen:

- Een pijl van een rondje (type) naar een rechthoekje (operatie) geeft aan dat die operatie een argument van dat type heeft. Heeft een operatie meerdere argumenten, dan trekken we naar de betrokken rechthoek ook meer pijlen, waarbij de argumenten van links naar rechts worden aangeduid.
- Een pijl van een rechthoekje (operatie) naar een rondje (type) betekent dat die operatie een resultaat van dat type oplevert.

Het NTO voor het zojuist gedefinieerde type `COMPL` bestaat uit een aantal delen:

- de denotatie operator `I`
- als inversen daarvan de accesoperatoren `RE` en `IM` alsmede de monadische operatoren `ABS` en `PHI`
- de constanten `complex 1`, `complex 0`, `complex i`
- de monadische `-`, `+`, `CONJ` en `RECIP`
- de dyadische `+`, `-`, `*` en `/`
- de vergelijkingsoperatoren `=` en `<>`
- de conversieprocedure `compl`.

Al deze delen vinden we terug in onderstaand schema (Fig. 6.3). Dit NTO vormt zelf weer een deel van een groter geheel met de NTO's van `REAL` en `BOOL`, die we hier echter niet hebben opgenomen.

6.3 Kwadratuur van de eenheidscirkel

We gaan het zojuist gedefinieerde abstracte datatype `COMPL` gebruiken in een voorbeeld waar het als elementair type optreedt (we kennen enkel zijn naam, zijn representatie blijft voor ons verborgen).

Het voorbeeld illustreert, behalve enige complexe operaties, vooral het effect van de onnauwkeurigheid in de `REAL`-arithmetiek waar het op opbouwt.

De complexe getallen waarvoor de absolute waarde gelijk is aan 1.0 liggen alle op de *complexe eenheidscirkel*, een cirkel rond de oorsprong (0.0,0.0) met straal 1.0:

$$\mid z \mid = 1.0 \leftrightarrow \sqrt{Re\,z.Re\,z + Im\,z.Im\,z} = 1.0$$

Als de waarde van de complexe variabele `z` op die eenheidscirkel ligt, dan ligt ook de waarde van zijn kwadraat `z*z` op de eenheidscirkel — afgezien van eventuele afrondfouten van de onderliggende `REAL`-arithmetiek. Als `z` echter behept is met een afrondingsfout dan geldt $\mid z \mid > 1.0$ of $\mid z \mid < 1.0$. Neem aan $z = 1 + \varepsilon$, dan is $z^2 = (1+\varepsilon)(1+\varepsilon) \approx 1 + 2\varepsilon$ voor kleine ε. Als we een getal `z` op de eenheidscirkel herhaaldelijk kwadrateren dan verwachten we dat daarbij de afrondfout iedere keer ongeveer verdubbelt, tenzij we net een complex getal bij de kop hebben waarvan alle machten exact representeerbaar zijn, zoals geldt voor de getallen 1, -1, i en -i. Al kwadraterend moet het getal zich dus in het algemeen steeds verder van de eenheidscirkel af bewegen, naar binnen of naar buiten. We zullen dit fenomeen eens experimenteel gaan onderzoeken.

```
program:
  vraag startpunt;
  jaag punt rond de cirkel.
```

```
vraag startpunt:
  REAL VAR t;
  REP
    line;
    put ("Getal tussen -1.0 en 1.0: ");
    get (t)
  UNTIL - 1.0 <= t AND t <= 1.0
  ENDREP.
```

```
jaag punt rond de cirkel:
  COMPL VAR z :: t I sqrt (1.0 - t * t);
  INT VAR i;
  FOR i FROM 1
  REP
    line;
    put (i);
    put (z);
    z := z * z;
    put (ABS z)
  UNTIL ABS z > 2.0 OR ABS z < 0.5
  ENDREP.
```

Het programma vraagt een reëel getal t en vormt daaruit het complexe getal $(t, \sqrt{1-t^2})$ dat op de eenheidscirkel hoort te liggen. Voor $t = 0.23$ krijgen we de volgende resultaten:

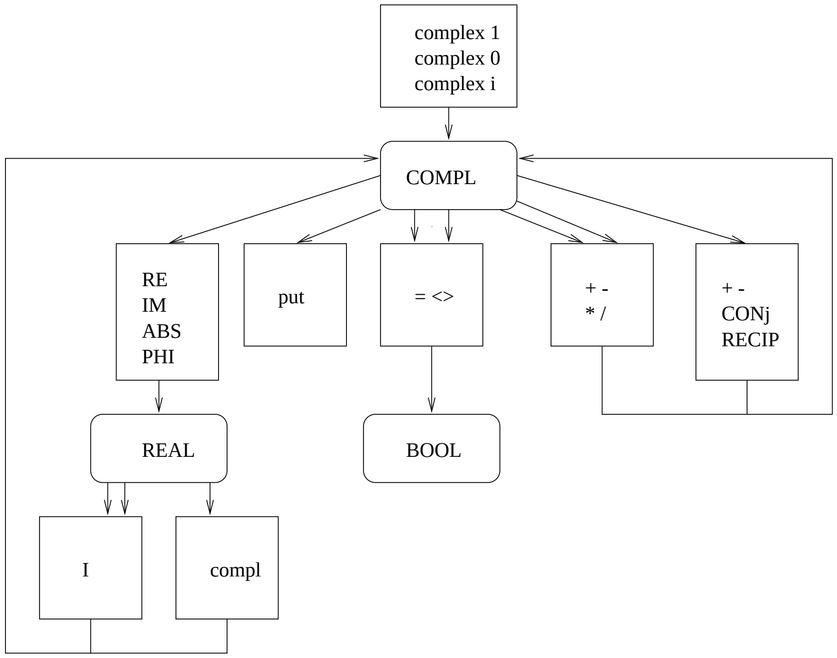


Figure 6.3: Signatuur van COMPL

```
 1 ( 2.300000e-01  9.731906e-01) 1.000000e+00
 2 (-8.942000e-01  4.476677e-01) 1.000000e+00
 3 ( 5.991873e-01 -8.006089e-01) 1.000000e+00
 4 (-2.819492e-01 -9.594293e-01) 1.000000e+00
 5 (-8.410093e-01  5.410207e-01) 1.000000e+00
 6 ( 4.145932e-01 -9.100068e-01) 1.000000e+00
 7 (-6.562249e-01 -7.545654e-01) 1.000000e+00
 8 (-1.387378e-01  9.903291e-01) 1.000000e+00
 9 (-9.615036e-01 -2.747922e-01) 1.000000e+00
10 ( 8.489785e-01  5.284275e-01) 1.000000e+00
11 ( 4.415289e-01  8.972470e-01) 1.000000e+00
12 (-6.101045e-01  7.923209e-01) 1.000000e+00
13 (-2.555449e-01 -9.667972e-01) 1.000000e+00
14 (-8.693936e-01  4.941202e-01) 1.000000e+00
15 ( 5.116904e-01 -8.591699e-01) 1.000000e+00
16 (-4.763458e-01 -8.792580e-01) 1.000000e+00
17 (-5.461893e-01  8.376618e-01) 1.000000e+00
18 (-4.033545e-01 -9.150438e-01) 1.000000e+00
19 (-6.746104e-01  7.381740e-01) 1.000000e+00
20 (-8.980173e-02 -9.959597e-01) 1.000000e+00
21 (-9.838713e-01  1.788778e-01) 1.000000e+00
22 ( 9.360055e-01 -3.519855e-01) 1.000000e+00
23 ( 7.522125e-01 -6.589206e-01) 1.000000e+00
24 ( 1.316472e-01 -9.912966e-01) 1.000000e+00
25 (-9.653380e-01 -2.610029e-01) 1.000000e+00
26 ( 8.637550e-01  5.039120e-01) 1.000000e+00
27 ( 4.921455e-01  8.705130e-01) 1.000000e+00
28 (-5.155857e-01  8.568380e-01) 1.000000e+00
29 (-4.683428e-01 -8.835468e-01) 1.000000e+00
30 (-5.613100e-01  8.276057e-01) 1.000000e+00
31 (-3.698622e-01 -9.290867e-01) 1.000000e+00
32 (-7.264040e-01  6.872681e-01) 1.000000e+00
33 ( 5.532534e-02 -9.984686e-01) 1.000001e+00
34 (-9.938786e-01 -1.104812e-01) 1.000002e+00
35 ( 9.755887e-01  2.196099e-01) 1.000003e+00
36 ( 9.035447e-01  4.284978e-01) 1.000007e+00
37 ( 6.327827e-01  7.743339e-01) 1.000014e+00
38 (-1.991789e-01  9.799702e-01) 1.000028e+00
39 (-9.206694e-01 -3.903788e-01) 1.000055e+00
40 ( 6.952364e-01  7.188197e-01) 1.000111e+00
41 (-3.334803e-02  9.994992e-01) 1.000222e+00
42 (-9.978866e-01 -6.666265e-02) 1.000443e+00
43 ( 9.913338e-01  1.330435e-01) 1.000887e+00
44 ( 9.650422e-01  2.637811e-01) 1.001775e+00
45 ( 8.617260e-01  5.091199e-01) 1.003552e+00
47 (-5.362621e-01  8.482574e-01) 1.014286e+00
46 ( 4.833686e-01  8.774436e-01) 1.007118e+00
48 (-4.319636e-01 -9.097765e-01) 1.028776e+00
49 (-6.411008e-01  7.859806e-01) 1.058380e+00
50 (-2.067553e-01 -1.007786e+00) 1.120168e+00
51 (-9.728842e-01  4.167300e-01) 1.254775e+00
52 ( 7.728398e-01 -8.108600e-01) 1.574461e+00
53 (-6.021267e-02 -1.253330e+00) 2.478927e+00
```

Inderdaad zien we in de rechterkolom de afrondfout steeds verdubbelen, dus exponentieel toenemen. Lange tijd blijven de getallen aardig op de eenheidscirkel (waar ze wel wild heen en weer vliegen) tot ze vrij plotseling divergeren. Voor deze startwaarde van t explodeert de cirkel, voor andere startwaarden (zoals t=0.44) implodeert hij. De reden waarom in de eerste 15 termen nog geen spoor van divergentie te zien is, terwijl deze daarna opkomt, is vrij platvloers: de gebruikte afdruk-procedure geeft heel wat minder cijfers weer dan waarin gerekend wordt. De afrondfout zit er dus meteen al in, alleen wordt hij pas later zichtbaar.

Als we, in plaats van steeds te kwadrateren, een vermenigvuldiging met een of ander complexe constante `c` hadden genomen `t:= t * c` dan was de afrondfout veel langzamer toegenomen (lineair in plaats van exponentieel) en hadden we misschien wel eeuwen kunnen rekenen voor de afrondfout merkbaar was.

In dit voorbeeld gebruikten we slecht enkele van de complexe operaties. We zullen een tweede voorbeeld geven, waarin bijna het hele pakket wordt gebruikt.

6.4 Impedantie van een serie/parallel netwerk

Als diepgaander voorbeeld van een toepassing van complexe getallen nemen we een onderwerp uit de electrotechniek: de bepaling van de impedantie van serie/parallel netwerken bestaande uit weerstanden, condensatoren en spoelen. We recapituleren eerst enige begrippen uit de electrotechniek (zie bijvoorbeeld [VRI48] als u dat kunt vinden — en vergeet

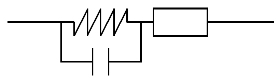


Figure 6.4: Een serie/parallel netwerk

dan niet aandachtig de inleiding te lezen. Of zie een leerboek als [GRPH]).

6.4.1 Enige begrippen uit de electrotechniek

Aansluiting van een gelijkspanning v op een weerstand r veroorzaakt een stroom i, met $v = i * r$ oftewel $i = v/r$ (*Wet van Ohm*), waarin

v = spanning, gemeten in Volts,

r = weerstand, in Ohms,

i = stroom, in Ampères.

Een wisselstroom is een stroom die voortdurend van richting en sterkte verandert, volgens een periodiek patroon. Op de tekening (fig. 6.4.1) ziet U enkele

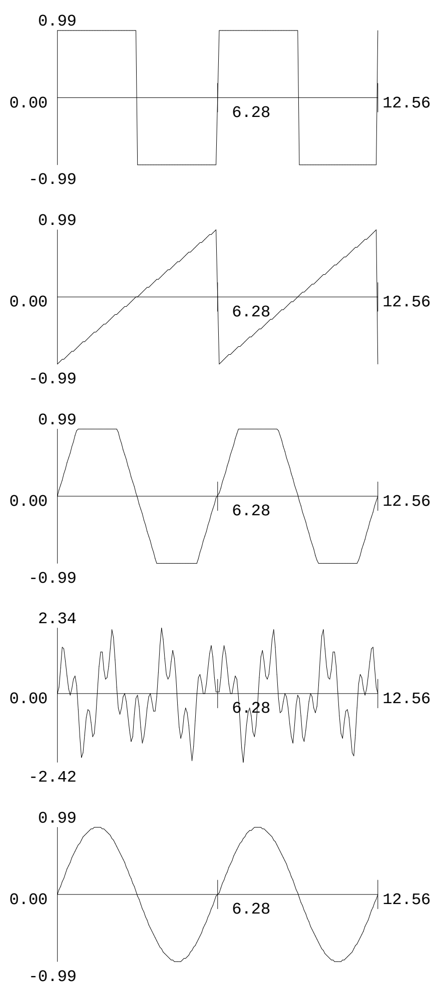


Figure 6.5: Verschillende vormen van wisselstroom

periodieke patronen. De laatste vorm noemen we een *sinusoide*, zijn verloop laat zich beschrijven als $I\cos(\omega t + \phi)$, met I=amplitude, $\omega = 2\pi f$, ϕ=fase.

De frequentie van een wisselstroom is het aantal perioden per seconde $f = 1/T$, waar T = tijdsduur van een periode.

Ieder periodiek patroon is te verkrijgen door de superpositie van een aantal sinusoiden van verschillende frequentie (gehele veelvouden van de grondfrequentie $1/T$), amplitude en fase. Daarom zullen we ons (eenvoudigheidshalve) in het vervolg beperken tot de beschouwing van één enkele sinusoidale wisselstroom.

Loopt een wisselstroom $i = I\cos\omega t + \phi$ door een weerstand r dan staat er over de weerstand een wisselspanning $v = ri = rI\cos(\omega t + \phi)$.

Staat, omgekeerd, over een weerstand r een wisselspanning v dan loopt er door de weerstand een stroom i, als boven. De wet van Ohm gaat dus zonder meer op voor wisselstroom en spanning over een (Ohmse) weerstand.

Nu bekijken we het effect van een condensator. Zetten we een wisselspanning $v = V\cos(\omega t + \phi)$ op een condensator met *capaciteit* c (gemeten in Farad) dan leidt dit tot een wisselstroom i door de condensator, waarvan de grootte afhangt van de frequentie, maar die bovendien een faseverschuiving van 90^o heeft met de wisselspanning.

De condensator gedraagt zich als een frequentieafhankelijke weerstand (*impedantie* of reactantie) Z, waarvan de absolute waarde gelijk is aan $Z = 1/\omega c$. De genoemde faseverschuiving komt tot stand doordat de stroom als het ware de condensator moet voltanken voordat de spanning daarover is opgebouwd. De spanning ijlt daardoor 90^o achter de stroom aan.

Als $i = I\cos(\omega t+\phi)$ dan is $v = ZI\cos(\omega t+\phi-90^o)$.

Op dezelfde wijze gedraagt een spoel met zelfinductie L (gemeten in Henry) zich als een frequentieafhankelijke weerstand met impedantie $Z = \omega L$, waarbij deze keer de spanning op de stroom voorijlt (als je er een spanning opzet duurt het even tot de stroom het magneetveld heeft opgebouwd).

Ach hoe ingewikkeld is dit alles, zult u denken. Het wordt allemaal echter veel eenvoudiger als we overgaan op een model met complexe getallen.

6.4.2 Complexe impedantie

We modelleren nu de wisselspanning over een impedantie z, en de wisselstroom die daar met een van z afhankelijke fase en grootte van afhangt, als twee grootheden, een reële spanning v en een complexe stroom $\bar{\imath}$, verbonden via de wet van Ohm:

$$v = V\cos(\omega t + \phi) = \bar{z}\bar{\imath}$$

De impedantie heeft hier een complexe waarde. Is de impedantie een zuivere weerstand, dan volgt de stroom de spanning op de voet. Heeft $\bar{z}$ echter een imaginaire component ongelijk aan nul, dan ijlt navenant de stroom $\bar{\imath}$ voor dan wel na.

Laten we nu de impedantie onderzoeken van netwerken van de volgende structuur: Een netwerk is

1. een weerstand R (in Ohms), of
2. een condensator met capaciteit C (in Farad), of
3. een spoel met zelfinductie L (in Henry), of
4. een netwerk in serie geschakeld met een ander netwerk, of
5. een netwerk parallel aan een ander netwerk.

Figure 6.6: Serie- en parallel-schakeling

Aan ieder netwerk kennen we een complex getal toe, zijn impedantie, als volgt berekend:

1. De impedantie van een weerstand R is $(r, 0)$.
2. De impedantie van een condensator C is $(0, -1/\omega c)$.
3. De impedantie van een spoel L is $(0, \omega l)$
4. Voor twee impedanties $\bar{Z}_1$ en $\bar{Z}_2$ in serie geldt dat ook de totale impedantie $\bar{Z} = \bar{Z}_1 + \bar{Z}_2$.
5. Voor twee impedanties $\bar{z}_1$ en $\bar{z}_2$ in parallel geldt dat de reciproke van de totale impedantie $1/\bar{z} = 1/\bar{z}_1 + 1/\bar{z}_2$ oftewel $\bar{z} = \frac{1}{\frac{1}{\bar{z}_1} + \frac{1}{\bar{z}_2}}$

Bij de voorgegeven frequentie `freq` kunnen we nu de impedantie van een netwerk (en daarmee de fase en grootte van de stroom ten opzichte van de spanning) eenvoudig bepalen door complexe rekenkunde. We voeren daartoe een type `IMP` (voor impedantie) in:

```
LET IMP = COMPL;
```

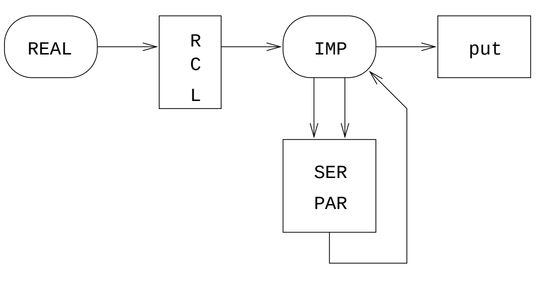


Figure 6.7: NTO voor netwerken van impedanties

Voor de drie grondcomponenten van een netwerk hebben we ieder een denotatie operator.

```
IMP OP R (REALr):
  r I 0.0
ENDOP R;

IMP OP C (REALc):
  0.0 I - 1.0 / omega / c
ENDOP C;

IMP OP L (REALl):
  0.0 I omega * l
ENDOP L;
```

Voor de serie- en parallel schakeling hebben we ieder een dyadische operator.

```
IMP OP SER (IMP  z1, z2):
  z1 + z2
ENDOP SER;

IMP OP PAR (IMP  z1, z2):
  RECIP (RECIP z1 + RECIP z2)
ENDOP PAR;
```

De impedantie van een eenvoudig netwerk, bestaand uit een condensator van 200 pF(pico Farad) in serie met met een spoel van 125 mH(milli Henry) kunnen we nu afdrukken (voor de gegeven frequentie `freq`) door

```
  IMP CONST z :: C 200.0e-12 SER L
125.0e-6;
  put (z)
```

Als `freq=1.0e6` (1Mhz) wordt hiervoor het complexe getal `(0.000000e+00 6.023195e+04)` afgedrukt. Dat getal alleen zegt ons niet veel, het lijkt verstandiger om het verloop van de impedantie als functie van de frequentie nader te gaan bekijken.

We schrijven daartoe een programma dat dient als een soort van impedantiemeter, en waarmee we de impedantie van de boven gegeven schakeling voor verschillende frequenties gaan onderzoeken.

```
maak impedantiegrafiek:
  vraag frequentiebereik;
  LET aantal stappen = 19;
  REAL CONST df :: (maxfreq-minfreq) /
                   real(aantal stappen);
  INT VAR t;
  FOR t FROM 0 UPTO aantal stappen
  REP REAL CONST f :: minfreq +
real(t)*df;
      bereken impedantie z bij f;
      plot (f, z)
  ENDREP.

vraag frequentiebereik:
  REAL VAR minfreq, maxfreq;
  put ("Van (Khz)?");
  get (minfreq);
  minfreq := minfreq * 1000.0;
  put ("Tot (Khz)?");
  get (maxfreq);
  maxfreq := maxfreq * 1000.0.

bereken impedantie z bij f:
  REAL CONST omega :: 2.0 * pi * f;
  IMP CONST z :: netwerk.

netwerk:
  C 200.0e-12 SER L 125.0e-6.
```

We verzinnen een nogal primitieve manier om grafieken op het beeldscherm weer te geven. Langs de linkerkant van het scherm geven we telkens een frequentiewaarde, met rechts daarvan een punt, voorafgegaan door een aantal spaties evenredig met $^2\log \mid z \mid^2$. We nemen de logaritme om een groot dynamisch bereik te kunnen afdekken, en het kwadraat om de variabiliteit van z te benadrukken. Op die manier krijgen we dus een grafiek te zien. De breedte die voor de grafiek ter beschikking staat is 60 posities, we bestrijken dus een bereik 2^{30} (een punt rechts staat voor een bedrag dat 2^{30} groter is dan een punt links), een bereik waarom menig oscilloscoop-bezitter ons zal benijden.

```
PROC plot (REAL f, IMP i):
  COMPL CONST z :: i;
  INT CONST x ::int(2.0*log2(ABSz))+18;
  put (f);
  put (min (60, x) * " ");
  put (".");
  line
ENDPROC plot;
```

De constante 18 in de plot-procedure is experimenteel gekozen. Voor andere netwerken zal hij anders gekozen moeten worden, zodanig dat de grafiek niet rechts of links uit het scherm loopt.

In het frequentiegebied van 500 tot 1500 KHz verloopt de impedantie van de bovengenoemde LC parallelschakeling als volgt.

```
5.000000e+05                                .
5.526316e+05                               .
6.052632e+05                              .
6.578947e+05                             .
7.105263e+05                            .
7.631579e+05                           .
8.157895e+05                         .
8.684211e+05                       .
9.210526e+05                    .
9.736842e+05              .
1.026316e+06           .
1.078947e+06                   .
1.131579e+06                      .
1.184211e+06                        .
1.236842e+06                         .
1.289474e+06                          .
1.342105e+06                           .
1.394737e+06                            .
1.447368e+06                            .
1.500000e+06                             .
```

De waarde van de impedantie vertoont een minimum bij ongeveer 1 Mhz.

6.4.3 Resonantie

Wat we waarnemen is een *resonantie verschijnsel*: bij een bepaalde frequentie zijn de fase verschuivingen, veroorzaakt door de spoel en de condensator, even groot en precies tegengesteld. De samengestelde faseverschuiving is dan nul. Door beschouwing van telkens kleinere frequentiegebieden rond de resonantiefrequentie stellen we vast dat deze ongeveer 1006.58KHz bedraagt.

Het geval van serieschakeling van een spoel L en een condensator C is eenvoudig analytisch te onderzoeken: de imaginaire component van $(0, \frac{-1}{\omega c}) + (0, \omega L)$ is nul als $1/\omega C = \omega L$ oftewel $\omega^2 = 1/CL$. Invullen van de gekozen waarden voor C en L geeft:

$$\omega^2 = \frac{1}{CL} = \frac{1}{200 * 10^{-12} * 125 * 10^{-6}} = 4.10^{13}$$

$$\omega = 2\pi f = 2 * \sqrt{10} * 10^6$$

$$f = \frac{\omega}{2\pi} = \frac{\sqrt{10}}{\pi} * 10^6 = 1.006584\ldots MHz$$

Onder de resonantiefrequentie overheerst het effect van de condensator, daarboven het effect van de spoel. Bij de resonantiefrequentie is de impedantie van deze kring nul. Bij de resonantiefrequentie verwachten we dan ook een omkering van de fase. We wijzigen de plot-procedure zo, dat hij naast de frequentie ook de fase-hoek `PHI z` afdrukt (in 4 posities).

```
PROC plot (REAL f, IMP i):
  COMPL CONST z :: i;
  INT CONST x :: int(2.0*log2(ABSz))+10;
  line;
  put (f);
  put (text (PHI z, 4));
  put (min (60, x) * " ");
  put (".")
ENDPROC plot;
```

Als we nu vol verwachting ons experiment herhalen krijgen we een onaangename verrassing: real-overflow bij de deling in de operator `PHI`. En dat is ook geen wonder: we delen door de reële component van `z`, die gelijk is aan nul!

Enig nadenken leert ons dat ons model van het netwerk natuurlijk fysiek niet erg realistisch is: In werkelijkheid heeft de spoel ook een interne weerstand, van bijvoorbeeld 1 Ohm, die we ons kunnen voorstellen als in serie geschakeld met die (ideale, weerstandloze) spoel van 125 μH. Ook heeft onze condensator een zekere lekweerstand, die we parallel geschakeld kunnen denken aan de (ideale, lekvrije) condensator van 200 pF. Die laatste weerstand zullen we echter weglaten.

We passen de beschrijving van het circuit aan tot

```
netwerk2:
  R 50.0 SER L 125.0e-6 SER C 200.0e-12.
```

en krijgen dan de volgende grafiek:

```
5.000000e+05  -88                              .
5.526316e+05  -87                             .
6.052632e+05  -87                            .
6.578947e+05  -86                           .
7.105263e+05  -85                          .
7.631579e+05  -84                         .
8.157895e+05  -82                       .
8.684211e+05  -78                     .
9.210526e+05  -70                  .
9.736842e+05  -46              .
1.026316e+06   32             .
1.078947e+06   66                 .
1.131579e+06   75                    .
1.184211e+06   79                      .
1.236842e+06   81                       .
1.289474e+06   83                        .
1.342105e+06   84                         .
1.394737e+06   85                          .
1.447368e+06   85                          .
1.500000e+06   86                           .
```

We zien dat de kromme iets vlakker verloopt dan voor een weerstandsloos netwerk, maar dat de resonantiefrequentie dezelfde is gebleven. Ook is te zien dat in de buurt van de resonantiefrequentie de faseverschuiving afneemt, terwijl deze bij de resonantiefrequentie van teken wisselt.

Nu kunnen we de impedantie en de resonantiefrequentie van ingewikkelder netwerken gaan onderzoeken, bijvoorbeeld de serieschakeling van twee parallel-netwerken met licht verschillende resonantiefrequenties, zoals

```
netwerk3:
  C500.0e-12 PAR (R1.0 SER L1.2e-3) SER
  (C490.0e-12 PAR (R1.0 SER L1.2e-3)).
```

Chapter 7

Abstracte typen (2)

Bij veel van de vorige voorbeelden verkeerden wij in de verleiding om van onze detailkennis van de representatie gebruik te maken, en daarmee representatie-onafhankelijkheid te verliezen. De manier om dit gevaar volledig uit te bannen is *sterke abstractie* te introduceren: type en representatie moeten zorgvuldig uit elkaar gehouden worden. Naast de synoniem-declaratie voor zwakke abstractie zullen we daarom mechanismen voor sterke abstractie introduceren, namelijk de type-declaraties en de pakketten met expliciete interfaces.

Tot slot van dit hoofdstuk behandelen we nog een aantal abstracte typen. De voorbeelden zijn alle vrij technisch of vrij wiskundig. Dat komt voornamelijk omdat de wiskunde (en de toepassing daarvan in de natuurkunde) nu eenmaal de bron bij uitstek is van weldoordachte algebra's die op elegante wijze als abstracte typen gemodelleerd kunnen worden.

We behandelen pakketten voor de vectoren in de $\Re^3$ en de rationale getallen.

7.1 Abstractiemiddelen

Het woord **abstract** wordt in de informatica om de haverklap gebruikt, het is een sleutelbegrip in vrijwel iedere programmeermethode. Maar wat betekent het eigenlijk? Volgens Van Dale [vDALE] betekent abstraheren:

> "ontdoen van het bepalende of toevallige, van het concrete; als begrip afleiden..."

Volgens het meest gezaghebbende amerikaanse woordenboek [WEBST] zijn de belangrijkste betekenissen:

> "to take away from, to consider apart from any particular instance"

In al deze omschrijvingen keren twee elementen weer:

- abstractie is een menselijke wilshandeling — voor computers is geen abstractie nodig, abstractie hoort bij ons denkproces
- abstractie is een vorm van beperking — je laat dingen weg. Het is de kunst, dóór de bomen heen het bos te zien.

Als definitie van abstractie in de context van het systematisch programmeren beschouw ik

> Abstractie is het bewust afzien van details.

De belangrijkste manieren om bij het programmeren van details af te zien zijn:

benoemen	aan een entiteit een naam geven om die naam in plaats daarvan te gebruiken.
verbergen	de realisering en eigenschappen zodanig verbergen dat er niet op onbedoelde wijze gebruik van kan worden gemaakt.
generaliseren	de toepasbaarheid van operaties verruimen, analogieën trekken.

We zullen zien hoe deze ideeën in de programmeertaal ELAN zijn weer te vinden.

7.1.1 Zwakke en sterke abstractie

We zullen spreken van **zwakke abstractie** ingeval een abstracte entiteit precies alle eigenschappen van zijn realisering heeft, zoals geldt voor de abstracte typen gerealiseerd via een synoniem-declaratie. Alle operaties van het domein zijn dan tevens operaties van het type, en twee typen met verschillende naam maar overigens gelijke realisering zijn in alle opzichten equivalent.

Soms hebben we behoefte aan **sterke abstractie**, een vorm van abstractie waarbij de abstracte entiteit niet alle eigenschappen van zijn realisering heeft (maar eigenschappen meer of minder):

- We willen abstraheren van de realisering van het domein, ten einde daarvoor op eenvoudige wijze verschillende representaties te kunnen kiezen.
- We willen kunnen bereiken dat voor het abstracte type andere operaties gelden dan volgen uit de realisering van zijn domein.

Voor algoritmen is de verfijning een zwak abstractiemiddel, omdat de verfijningsnaam precies hetzelfde effect en dezelfde waarde heeft als de romp van zijn realisering; de procedure en de operator zijn sterke abstractiemiddelen, vanwege het feit dat de daarin optredende declaraties naar buiten onzichtbaar zijn (minder eigenschappen!) en vanwege de mogelijkheid van recursie.

7.1.2 Type-declaraties

De synoniem-declaratie voert een zwakke abstractie in. Willen we sterke abstractie bereiken, dan moeten we typen invoeren middels een**type declaratie**.

type-declaration:
  type—token, type-association—list.

type-association:
  type-name, equal—token, type-declarer.

bijvoorbeeld:

```
    TYPE DATE = INT;
```

Het domein van `DATE` is hetzelfde als `INT` — alleen is `DATE` een ander type dan `INT`. De `INT` is als het ware zo stevig ingepakt dat ie alleen nog maar als `DATE` te gebruiken is. Dat betekent met name dat geen van de operaties van `INT` toepasbaar is op een `DATE` object: We kunnen bijvoorbeeld geen `DATE`'s bij elkaar optellen zonder daarvoor eerst een speciale operator in te voeren. En dat is maar goed ook, want anders kregen we misschien wel de grootste ongelukken.

Laten we een wat minder kritisch probleem bekijken: we willen een operator `NEXT` invoeren die ons de datum volgend op een gegeven datum oplevert:

```
DATE OP NEXT (DATE d):
    # de datum volgend op d #
```

Laten we daarbij aannemen dat we als representatie hebben gekozen

```
TYPE DATE = INT;
  #372 * jaar + dagnummer binnen jaar#
```

We moeten in de romp van `NEXT` de datum `d` als het ware heel voorzichtig uitpellen, met kennis van de representatie de volgende datum als een integer bepalen en die dan weer inpakken tot een `DATE`.

7.1.3 De concretizer

Dat uitpellen gaat met een nieuwe constructie, de **concretizer**, die de abstractie van een type doelbewust doorbreekt door een waarde van dat type op te vatten als een waarde van zijn representatie — in dit geval door een `DATE` te beschouwen als een `INT` — zonder daarbij overigens aan die waarde iets te veranderen. De concretizer ziet er syntactisch uit als een soort monadische operator.

concretizer:
  concr—token, compact-primary.

Hij heeft echter een bijzondere eigenschap (die monadische operatoren niet hebben): de concretizer erft het acces-recht van de primary. Als de primary een variabele is kan dus zelfs zijn waarde (door een toewijzing) veranderd worden.

7.1.4 De abstractor

Het omgekeerde, de abstractie van realisering tot abstract type, wordt bereikt door de **abstractor**.

abstractor:
  type-name, colon—token, compact-primary.

Als het type van de primary van de abstractor overeenkomt met de realisering van het daarin voor de dubbele punt genoemde abstracte type dan is zijn waarde de waarde van zijn primary, maar met het genoemde abstracte type. En daarmee is de abstractie weer hersteld. Ook de abstractor erft het accesrecht van zijn primary.

Zowel abstractor als concretizer mogen uitsluitend worden gebruikt binnen het pakket waarin het bijbehorende abstracte type wordt gedeclareerd. Daarbuiten bestaat er dus geen rechtstreekse manier om aan de representatie van het abstracte type te komen.

7.1.5 Voorbeelden daarvan

Terug naar het data-voorbeeld: De operator `NEXT` vergt een concretizer, en moet dus in één pakket met het type `DATE` gedeclareerd worden, bijvoorbeeld als

```
DATE OP NEXT (DATE d):
  # de datum volgend op d #
  INT CONST rep:: (CONCR d) + 1;
  IF rep MOD 372 <= length of year of(d)
  THEN DATE : rep
  ELSE DATE : (rep DIV 372 * 372 + 373)
  FI
ENDOP NEXT;
```

We nemen het voorbeeld van de complexe getallen nog eens ter hand, maar declareren nu

```
TYPE COMPLEX = STRUCT (REAL re, im)
```

Weer beschrijven we hetzelfde domein, alleen is een object van type `COMPLEX` nu niet automatisch tevens van het type `STRUCT (REAL re, im)` en omgekeerd (zoals gold voor het type `COMPL` als synoniem gedeclareerd).

Willen we in een bepaalde context gebruik maken van de representatie van `COMPLEX` dan moeten we expliciet de abstractie verbreken met een concretisatie, bijvoorbeeld in de acces algoritme

```
REAL OP IM (COMPLEX u):
    (CONCR u).im
ENDOP;
```

Hebben we een waarde van het type `STRUCT (REAL re, im)` dan is dat nog geen `COMPLEX`, zolang we niet expliciet aangeven dat deze abstractiestap genomen moet worden.

```
COMPLEX OP I (REAL a, b):
    COMPLEX: [a, b]
ENDOP;
```

7.1.6 Verkorte selectie

In de definities van de acces operatoren van een abstract type zal telkens een `CONCR` verschijnen, bij iedere selectie een.

De situatie dat we, binnen het pakket waarin een abstract type wordt gedefinieerd, daaruit moeten selecteren, komt vaker voor. In die omstandigheid mag de concretizer ook worden weggelaten, omdat de selectie en de subscriptie zo nodig zelf een `CONCR` operatie uitvoeren. We kunnen ons daarmee de expliciete concretizer besparen op een aantal plaatsen in het pakket, waar er geen twijfel over bestaat dat het abstracte type moet worden gekraakt.

Overigens laat de volgende poging, een procedure `put` voor `RAADSEL`'s te definiëren, zien dat in andere situaties de `CONCR` beslist noodzakelijk is om abstract type en realisatie uit elkaar te houden.

```
TYPE RAADSEL = INT;
PROC put (RAADSEL u):
   put (u)
   #recursieve oproep of put van INT?#
ENDPROC put;
```

Het antwoord op de raadselvraag is simpel — dit is een eindeloze recursie. Willen we de `put` van `INT` gebruiken dan moeten we echt schrijven:

```
PROC put (RAADSEL u):
   put (CONCR u)
ENDPROC put;
```

7.1.7 Paketten

Een **pakket** is een ingekapseld programma, dat een aantal **entiteiten** (d.w.z. typen, objecten en algoritmen) **exporteertexport** (d.w.z. ter beschikking stelt van andere pakketten en van het hoofdprogramma. De bijbehorende syntax luidt:

elan-program:
  packet—sequence—option, main-packet.

Een programma bestaat dus uit een reeks van pakketten, met als laatste het hoofd-pakket. Binnen een pakket kunnen alle entiteiten gebruikt worden die door een van de voorgaande pakketen geëxporteerd zijn.

Het geheel van regels die een programmeertaal kent met betrekking tot het **geldigheidsgebied** van declaraties noemt men ook wel bloemrijk de **zichtbaarheids regels** van die taal.

Volgens de zichtbaarheidsregels van ELAN is een globale declaratie uit een pakket zichtbaar in dat hele pakket (met uitzondering van daarin geneste geldigheidsgebieden van gelijknamige locale objecten). Door een pakket geëxporteerde entiteiten zijn bovendien zichtbaar in alle daarachter volgende paketten voor zover daarin niet een daarmee conflicterende declaratie voorkomt.

packet:
  packet-head, packet-body, packet-tail.

De kop van een pakket bevat zijn naam en zijn *interface* ("grensvlak").

packet-head:
  packet—token, packet-name, packet-interface, colon—token.

In het **interface** wordt opgesomd welke entiteiten door dit pakket geëxporteerd worden.

packet-interface:
  export-interface.

ELAN kent, in tegenstelling tot vele implementeringstalen, geen expliciet import-interface. De taal is nu eenmaal bedoeld voor het initiële onderwijs in de systematische programmering, niet voor de grootschalige beoefening daarvan.

export-interface:
  defines—token, export-name—list.

In het export-interface mogen uitsluitend namen genoemd worden van typen, operaties en constanten die binnen dat pakket gedeclareerd zijn (niet dus van variabelen en verfijningen, deze komen niet voor export in aanmerking).

Het geldigheidsgebied van deze declaraties wordt daardoor uitgebreid tot het **uitgebreid geldigheidsgebied** d.w.z. alle na dit pakket volgende pakketten, voor zover deze niet een daarmee concurrerende declaratie bevatten (zo zullen bijvoorbeeld de procedures `max` uit de standaardpakketten onzichtbaar zijn in een pakket waarin een constante `max` wordt gedeclareerd).

packet-body:
  packet-paragraph, refinement-train—option;

De hier gegeven syntax laat zien dat de romp van een pakket bestaat uit een bottom-up gedeelte (inclusief de wortel) en een top-down gedeelte. De syntax is licht vereenvoudigd (volgens de taaldefinitie mogen deze delen nog herhaald worden, zie Appendix A van het eerste deel).

packet-paragraph:
  packet-unit;
  packet-unit, semicolon—token, packet-paragraph.

packet-unit:
  basic-declaration;
  closed-declaration;
  expression.

Een pakket-paragraaf onderscheidt zich doordat daarin ook de zgn. **gesloten declaraties**, de typische bottom-up declaraties zijn toegestaan.

closed-declaration:
  procedure-declaration;
  operator-declaration;
  type-declaration.

packet-tail:
  end-packet—token, packet-name—option, semicolon—token.

De staart van een pakket is enkel een benoemde sluithaak.

main-packet:
  packet-body.

Het hoofdpakket (datgene wat we tot nog toe als programma kennen) bestaat enkel uit een romp. Het idee is dat je eerst de romp van een pakket ontwikkelt en daar later pakkethaken om heen slaat.

Het pakket-interface bevat enkel een opsomming van namen, die tezamen met de declaraties in het pakket als het ware de signatuur van het pakket vormen.

7.2 Voorbeeld: vectoren in $\Re^3$

De hiernavolgende invoering van vectoren in de driedimensionale ruimte en operaties daarop is afkomstig uit [MEU75]. Het is een klassiek voorbeeld van Bottom-Up programmering.

7.2.1 Representatie en operaties

We kiezen de meest voor de hand liggende representatie, als drie coördinaten:

```
TYPE VEC = STRUCT (REAL x coord, y
coord, z coord);
```

We introduceren een denotatie-procedure om uit waarden voor de coördinaten een vector te construeren. Dit zal de enige plaats zijn waar we een abstractor gebruiken.

```
VEC PROC vec (REAL x, y, z):
  VEC : [x, y, z]
ENDPROC;
```

Vervolgens introduceren we drie acces-algoritmen om op de coördinaten van een vector te kunnen toegrijpen.

```
REAL OP X (VEC n):
   n . x coord
ENDOP X;

REAL OP Y (VEC n):
   n . y coord
ENDOP Y ;

REAL OP Z (VEC n):
   n . z coord
ENDOP Z ;
```

Merk op dat we in de drie selecties de concretizers achterwege hebben gelaten. Van nu af aan zullen we ook geen selecties meer gebruiken (omdat die bekendheid vooronderstellen met de realisering van het type) maar uitsluitend de acces-operatoren.

Nu volgen (voor de volledigheid) de twee monadische grondoperaties.

```
VEC OP + (VEC u):
   u
ENDOP + ;

VEC OP - (VEC u):
  vec (- Xu, - Yu, -Zu)
ENDOP - ;
```

Dan volgen enige dyadische grondoperaties.

```
VEC OP + (VEC u, v):
  vec (Xu + Xv, Yu + Yv, Zu + Zv)
ENDOP + ;

VEC OP - (VEC u, v):
  vec (Xu - Xv, Yu - Yv, Zu - Zv)
ENDOP - ;

VEC OP * (REAL r, VEC u):
  # scalar maal vector #
  vec (r * Xu, r * Yu, r * Zu)
ENDOP * ;

VEC OP * (VEC u, REAL r):
  # vector maal scalar #
  r * u
ENDOP * ;

VEC OP / (VEC u, REAL r):
   # vector gedeeld door scalar #
   1/r * u
ENDOP / ;
```

Nu kunnen we de beide vectorproducten definiëren.

```
REAL OP * (VEC u, v):
  # inproduct #
  Xu * Xv + Yu * Yv + Zu * Zv
ENDOP * ;

VEC OP **  (VEC u, v):
  # uitproduct #
  vec (Yu * Zv - Zu * Yv,
       Zu * Xv - Xu * Zv,
       Xu * Yv - Yu * Xv)
ENDOP ** ;
```

We definiëren operatoren om de norm van een vector en de eenheidsvector in een gegeven richting te berekenen.

```
REAL OP NORM (VEC u):
  sqrt (u * u)
ENDOP NORM ;
```

```
VEC OP E (VEC u):
  # eenheidsvector in richting u #
  u / NORM u
ENDOP E ;
```

We definiëren operaties die als nultest gebruikt kunnen worden.

```
LET eps = 1.0e-10; # voldoende klein #

BOOL OP EPS (REAL r):
  ABS r < eps
ENDOP EPS ;

BOOL OP EPS (VEC u):
  EPS NORM u
ENDOP EPS ;
```

Nu komen nog drie uiterst avontuurlijke operaties, om te bepalen of twee vectoren parallel dan wel loodrecht staan, en of drie vectoren in één vlak liggen. De laatste moet een procedure zijn, omdat we geen operatoren met méér dan twee operanden kunnen definiëren.

```
BOOL OP PARL (VEC u, v):
  # u parallel v? #
  EPS (E u **  E v)
ENDOP PARL ;

BOOL OP PERP (VEC u, v):
  # u loodrecht op v? #
  EPS (E u * E v)
ENDOP PERP ;

BOOL PROC in one plane (VEC u, v, w):
  # liggen u, v en w in een vlak?#
  u PERP E v **  E w
ENDPROC in one plane ;
```

Eigenlijk hadden we eerst een signatuur moeten bedenken om daarna aan het programmeren te gaan, maar voor deze keer zullen we de signatuur uit het programma afleiden (fig. 7.1).

Uit de signatuur laat zich op zijn beurt het volgende interface afleiden:

```
PACKET vectors in r3 DEFINES
     VEC,
     X, Y, Z, NORM, E,
     +, -, *,** , /, EPS,
     PARL, PERP, in one plane:
```

Achter de declaraties voegen we nog toe

```
ENDPACKET vectors in r3;
```

en het pakket is gereed.

Voor Bottom-Up programmeurs is bovenstaand proza ideale ontspanningslectuur. De complete versie van deze declaraties plus een toepassingsvoorbeeld vindt men in hoofdstuk 8.4.1 van het genoemde boek [MEU75].

7.3 Voorbeeld: rationale getallen

Ons volgende voorbeeld betreft een implementering van de rationale getallen, dat wil zeggen van die reële getallen die als het quotiënt van twee integers kunnen worden voorgesteld. Deze representatie kent niet de onnauwkeurigheid van de reals (maar heeft weer andere beperkingen door het beperkte bereik van de integers). Het rekenen met rationale getallen is voor een mens zo moeilijk dat we het maar weinig proberen. Voor een computer is het echter geen bijzonder lastige representatie.

7.3.1 De rationale rekenkunde

We definiëren een pakket voor het werken met rationale getallen, met het volgende grensvlak:

```
PACKET rationale rekenkunde
DEFINES RAT,
  T , N , fract,
  zero, one,
  <, =, +, -, *, /, RECIP,
  sign, real, get, put:
```

We kiezen de meest voor de hand liggende representatie:

```
TYPE RAT = STRUCT (INT teller, noemer);
```

Hierbij moet de noemer ongelijk zijn aan nul. We zullen er zorg voor dragen dat een `RAT` altijd genormaliseerd is, dat wil zeggen dat teller en noemer zo klein mogelijk zijn en de noemer positief. Het eerste bereiken we door te zorgen dat ze geen factor (anders dan 1) gemeenschappelijk hebben. De kans dat twee willekeurige gehele getallen een factor anders dan 1 gemeenschappelijk hebben is verrassend hoog: ongeveer 58.5% (zie bijvoorbeeld 4.5.3, Theorem E in [KNU81]). Het is dus alleszins de moeite waard, gemeenschappelijke factoren uit teller en noemer te verwijderen.

We definiëren daartoe een hulp-algoritme `gcd` ("greatest common divisor", de grootste gemene deler) die we op strategische plaatsen zullen oproepen. We exporteren deze niet naar buiten het pakket.

```
INT PROC gcd (INT teller, noemer):
  IF noemer <= 0
  THEN
    teller
  ELIF noemer > teller
  THEN
    gcd (noemer, teller)
  ELSE
    gcd (noemer, teller MOD noemer)
  FI
ENDPROC gcd ;
```

Vervolgens definieëren we een denotatie-algoritme `fract` die een genormaliseerde `RAT` oplevert.

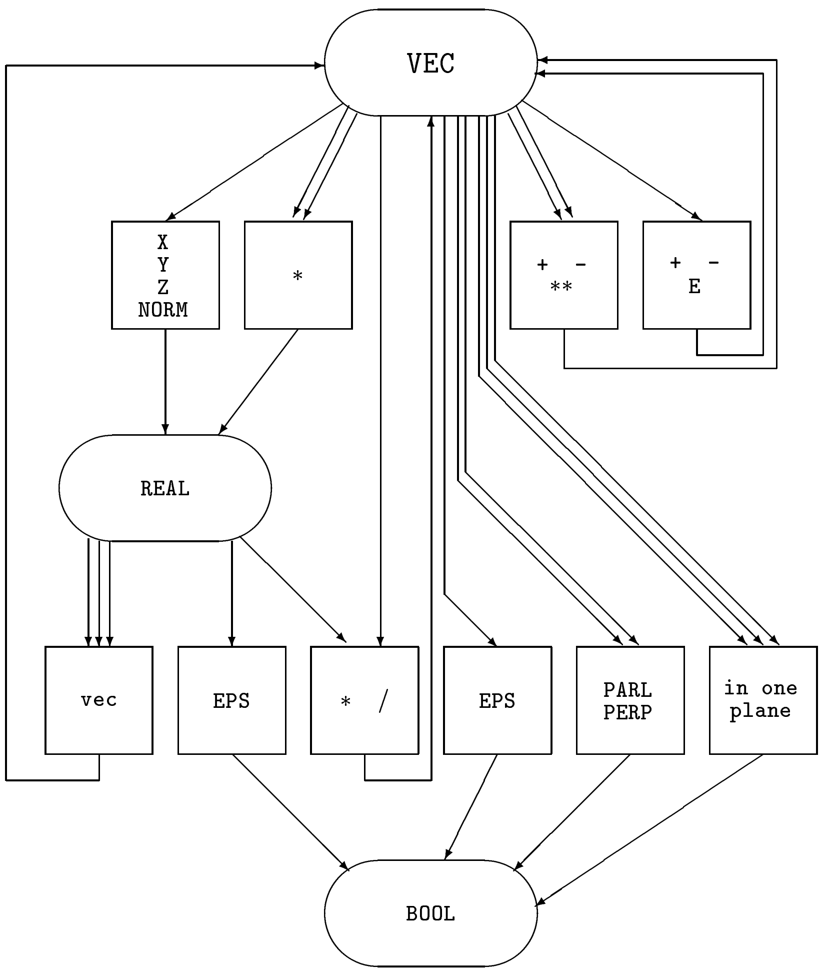


Figure 7.1: Signatuur van het type VECTOR

```
RAT PROC fract (INT teller, noemer):
  INT CONST s:: sign(teller) *
              sign(noemer);
  INT CONST t:: abs(teller),
            n:: abs(noemer);
  assert (noemer <> 0);
  INT CONST c:: gcd (t, n);
  RAT : [ s * t DIV c, n DIV c ]
ENDPROC fract ;
```

We definiëren twee constanten `one` en `zero`.

```
RAT PROC one :
  RAT : [ 1, 1 ]
ENDPROC one ;

RAT PROC zero :
  RAT : [ 0, 1 ]
ENDPROC zero ;
```

en twee access-algoritmen

```
INT OP N (RAT a):
  a . noemer
ENDOP N ;

INT OP T (RAT a):
  a . teller
ENDOP T ;
```

Nu komen de arithmetische operatoren op `RAT`'s.

```
RAT OP + (RAT a, b):
  fract(T a * N b + T b * N a,
        N a * N b)
ENDOP + ;

RAT OP + (INT a, RAT b):
  b + RAT : [ a, 1 ]
ENDOP + ;

RAT OP - (RAT a):
  RAT : [ - T a, N a ]
ENDOP - ;

RAT OP - (RAT a, b):
  a + (- b)
ENDOP - ;

RAT OP RECIP (RAT a):
  RAT : [ N a, T a ]
ENDOP RECIP ;

RAT OP / (RAT a, b):
  a * RECIP b
ENDOP / ;

RAT OP / (INT a, RAT b):
  RECIP b * RAT : [ a, 1 ]
ENDOP / ;
```

Zoals u ziet streven we eerder naar helderheid dan naar de grootste efficiëntie — sommige van deze operatoren zouden wel wat slimmer gerealiseerd kunnen worden.

```
RAT OP * (RAT a, b):
  RAT CONST c:: fract(T a, N b),
            d:: fract(T b, N a);
  fract (T c * T d, N c * N d)
ENDOP * ;
```

Deze moeizame realisering van de vermenigvuldigingsoperator is gekozen om overflow te vermijden door eventuele gemeenschappelijke factoren zo snel mogelijk weg te delen.

Zij $a = \frac{a_t}{a_n}$ en $b = \frac{b_t}{b_n}$. Laat de notatie $x||y$ staan voor: $gcd(x, y) = 1$. We weten dat a en b genormaliseerd zijn, dus $a_t||a_n$ en $b_t||b_n$. Als we nu zonder meer zouden berekenen $a * b = \frac{a_t * b_t}{a_n * b_n}$ dan zouden (door eventuele gemeenschappelijke factoren van a_t en b_n dan wel b_t en a_n) de teller en de noemer van de rechterkant wel eens onnodig groot kunnen uitvallen. Weliswaar wordt dat door de oproep van `fract` in de laatste regel weer opgelost, maar ondertussen zou overflow kunnen zijn opgetreden. Om overflow te vermijden halen we dus eerst de gemeenschappelijke factoren weg.

```
RAT OP * (INT a, RAT b):
  b * RAT : [ a, 1 ]
ENDOP * ;
```

Als volgende introduceren we twee van de zes vergelijkings-operatoren (de overige zijn niet zo noodzakelijk, bijv. `a >= b` kan worden geschreven als `NOT (a < b)`).

```
BOOL OP = (RAT a, b):
  T a = T b AND N a = N b
ENDOP = ;

BOOL OP < (RAT a, b):
  T a * N b < N a * T b
ENDOP < ;
```

We geven drie afdruk-procedures voor `RAT`'s: een die teller en noemer laat zien, een die de waarde (als reëel getal van 36 cijfers) afdrukt en een die deze beide oproept.

```
PROC put rat (RAT a):
  put (T a);
  put ("/");
  put (N a)
ENDPROC put rat ;
```

```
PROC put val (RAT a):
  INT VAR t:: abs (T a),
          n:: abs (N a);
  TEXT CONST h:: text (t DIV n);
  put (sign of a + h + ".");
  UPTO 36 - length (h)
  REP
   one digit
  ENDREP .
one digit :
  t := 10 * (t MOD n);
  put ("0123456789" SUB (t DIV n + 1)).
sign of a :
  IF a < zero
  THEN
    "-"
  ELSE
    " "
  FI.
ENDPROC put val ;

PROC put (RAT a):
  put rat (a); put val (a)
ENDOP put;
```

We geven een nogal rudimentair uitgevallen invoerprocedure:

```
PROC get (RAT VAR a):
  INT VAR d, n ;
  get (d);
  get (n);
  a := fract (d, n)
ENDPROC get ;
```

Nog wat klein vuil:

```
INT PROC sign (RAT n):
  sign (T n) * sign (N n)
ENDPROC sign ;

REAL PROC real (RAT a):
  real (T a) / real (N a)
ENDPROC real ;

ENDPACKET rationale rekenkunde;
```

Slimme verbeteringen

Vanwege het eindige integer-bereik is het gevaar groot dat er tijdens het rekenen met rationale getallen overflow optreedt. Dit gevaar hebben we trachten te bezweren door onderweg de berekende rationale getallen steeds te normaliseren (door gebruik van `fract`). Het is de moeite waard om na te gaan of nog meer overflow vermeden kan worden.

Bekijken we nogmaals de rationale optelling:

```
RAT OP + (RAT a, b):
  fract(T a * N b + T b * N a,
        N a * N b)
ENDOP + ;
```

Ingeval de noemers van `a` en `b` een factor gemeen hebben, dan zit die factor ook in teller en noemer van de berekende breuk. Bij normalisatie zal deze factor verdwijnen, maar we kunnen hem beter eerst zelf verwijderen, zodat teller en noemer met minder kans op overflow berekend kunnen worden. Als $a = \frac{a_t}{a_n}$ en $b = \frac{b_t}{b_n}$, als verder $d = gcd(a_n, b_n)$ dan berekenen we dus

$$\frac{a_t}{a_n} + \frac{b_t}{b_n} =$$

$$\frac{b_n * a_t + a_n * b_t}{a_n * b_n} =$$

$$\frac{b_n \div d * a_t + a_n \div d * b_t}{a_n \div d * b_n}$$

Als we nog op soortgelijke wijze eventuele gemeenschappelijke factoren van de teller en b_n uitdelen krijgen we:

```
RAT OP + (RAT a, b):
  INT CONST at:: T a,
            an:: N a,
            bt:: T b,
            bn:: N b;
  INT CONST d:: gcd (an, bn);
  IF d = 1
  THEN fract (at * bn + an * bt, an *
bn)
  ELSE
   INT CONST t:: bnDIVd * at + anDIVd *
bt;
   INT CONST e:: gcd (t, d);
   fract (tDIVe, anDIVd * (bnDIVe))
  FI
ENDOP +;
```

Ook kunnen we ons afvragen of niet te vaak een (tijdrovende) normalisatie plaatsvindt. Dit blijkt het geval te zijn bij de rationale vermenigvuldiging: Volgens de eerder gegeven argumentatie hebben we de gemeenschappelijke factoren uit het resultaat al verwijderd, zodat we de laatste regel kunnen vereenvoudigen:

```
RAT OP * (RAT a, b):
  RAT CONST c:: fract (T a, N b),
            d:: fract (T b, N a);
  RAT : [ T c * T  d, N c * N d]
ENDOP * ;
```

Het effect van de normalisatie op geschikt gekozen plaatsen is heel groot, zoals U kunt nagaan door in de volgende voorbeelden de normalisatie eens buiten gevecht te stellen.

7.3.2 Toepassingen van de rationale getallen

De volgende mini-programmaatjes maken op een aantal verschillende wijzen gebruik van het pakket `rationale rekenkunde`. Ze zitten allemaal in de

ELAN-toolkit en zijn zonodig gemakkelijk over te typen. Daarom stellen we voor dat U er zelf wat mee experimenteert en geven we de resultaten hier niet weer.

Voorbeeld: Som van reciproke getallen

We sommeren getallen van de vorm $1/i$. Proefondervindelijk blijkt (bij een 32-bits integer-arithmetiek) de vierentwintigste term overflow te geven, daarom houden we juist daarvoor op. Het effect van de automatische normalisatie van de rationale getallen is goed te zien.

```
sum of reciprocal numbers:
  RAT VAR sum:: zero;
  INT VAR i;
  FOR i UPTO 23
  REP
   line;
   put (i);
   sum := sum + fract (1, i);
   put (sum)
  ENDREP.
```

Voorbeeld: Som van reciproke faculteiten

We sommeren getallen van de vorm $1/i!$, wat ons een rationale benadering voor `e` oplevert.

```
sum of reciprocal factorials:
  RAT VAR sum:: one;
  RAT VAR fact:: one;
  INT VAR i;
  FOR i UPTO 12
  REP
   line;
   put (i);
   fact := fact * fract (1, i);
   sum := sum + fact;
   put (sum)
  ENDREP;
  line (2);
  put (e).
```

Na de twaalfde slag treedt al overflow op, terwijl er nog weinig nauwkeurigheid is bereikt.

Voorbeeld: Rationale vierkantswortel

We zullen met de methode van Newton de vierkantswortel van een rationaal getal bepalen. Eerst wordt de wortel ter controle met de gebruikelijke `REAL`-arithmetiek berekend en afgedrukt. We gaan door tot overflow optreedt.

```
square root:
  INT VAR i;
  RAT VAR arg:: zero;
  put ("Argument: ");
  get (arg);
  line;
  put (sqrt (real (arg)));
  line (2);
  tab;
  put (arg);
  RAT VAR x:: zero;
  RAT CONST a1:: arg - one;
  RAT CONST two:: fract (2, 1);
  FOR i FROM 1
  REP
   x := a1 / (two + x);
   line;
   put (i);
   put (one + x)
  ENDREP;
```

Het aantal slagen van de herhaling is afhankelijk van de ingevoerde waarde voor `arg`. Voor `arg` $= 4/5$ hebben we na 12 slagen inderdaad een betere precisie dan de real-berekening.

Voorbeeld: Gulden snede

We berekenen de gulden snede $(\frac{1+\sqrt{5}}{2})$ met een rationale en een reële benadering.

```
golden ratio:
  INT VAR i;
  RAT VAR x:: zero;
  FOR i FROM 1 UPTO 40
  REP
   x := RECIP (one + x);
   line;
   put (i);
   put (one + x)
  ENDREP;
  line (2);
  put ((1.0 + sqrt (5.0)) / 2.0).
```

Inderdaad blijkt de rationale benadering veel preciezer. Het aantal slagen (40) is weer proefondervindelijk gekozen.

7.4 Conclusie

We hebben nu alle taalmiddelen van ELAN voor het implementeren van abstracte datatypen behandeld, en een begrippenkader ingevoerd (NTO als signatuur, zwakke abstractie, sterke abstractie, encapsulatie en zichtbaarheidsregels) dat exemplarisch is voor datgene wat moderne programmeertalen in deze aanbieden.

Wat we niet behandeld hebben zijn de formele specificatie van de semantiek van ADT's, de verificatie van de correctheid van implementeringen en de bepaling van hun complexiteitseigenschappen. Deze thema's komen in het volgende deel [TvdW90] aan bod.

7.5 Opgaven

1. (matrix-vector rekening) Ontwerp en implementeer een pakket voor het rekenen met matrices en vectoren, dat twee abstracte typen `VEC` en `MAT` voor resp. de vectoren en de (vierkante) matrices van een gegeven orde n realiseert, met de bijbehorende operaties:
   - lezen en schrijven van matrices en vectoren
   - denotatie van de eenheidsmatrix en eenheidsvectoren
   - accesoperatoren `SUB` voor de elementen van een vector resp. matrix
   - accesoperatoren `THROW` en `THCOL` voor de n^e rij resp. kolom van een matrix ("nth row")
   - vector optelling, aftrekking, inproduct, uitproduct, scalar maal vector
   - matrix optelling, aftrekking, vermenigvuldiging, transpositie, inversie
   - matrix maal vector
   - oplossen van een lineair vergelijkingsstelsel

   We voeren geen operaties in voor het wijzigen van een element van een rij of vector variabele. Stel eerst een NTO op. Toepassingen zijn te vinden in ieder elementair leerboek over lineaire algebra. Ook zonder toepassing is dit echter een verhelderende oefening.
2. (multi-lengte rekening) Ontwerp een abstract type `LINT` (de lange integers), gerealiseerd als een rij van bijvoorbeeld tien `INT`'s, de cijfers van een representatie met als radix $\sqrt{maxint}$, met tenminste de operaties `+`, `-`, `*`, `MOD` en denotatie-, lees- en schrijf-operaties.
   In navolging van de snelle deling in **??** kan de modulo-operator voor positieve argumenten als volgt luiden:

```
LINT OP MOD (LINT a, b):
  IF a < b
  THEN a
  ELSE LINT VAR v:: b;
       WHILE a >= v + v
       REP   v:= v + v
       ENDREP;
       (a - v) MOD b
  FI
ENDOP MOD;
```

3. (priemgetallen) Onderzoek met behulp van de multilengte rekening de Mersenne priemgetallen (priemgetallen van de vorm $2^k-1, k \in \mathcal{N}$). Maak hierbij gebruik van de Lucas-Lehmer test: voor $k > 2$ geldt:

   $2^k - 1$ is priem dan en slechts dan als $e_{k-2} \bmod (2^k - 1) = 0$, waarbij e_k gedefinieerd is d.m.v. $e_0 = 4$ en $e_{k+1} = e_k^2 - 2$.
4. (nogmaals priemgetallen) Bepaal zoveel mogelijk Mersenne getallen die <u>waarschijnlijk</u> priem zijn, met behulp van de test van Rabin:

   Kies een random getal $r \in 1 : n-1$ en bepaal het grootste getal k zodanig dat 2^k een deler is van $n-1$.

   Als er een $i \in 1 : k$ bestaat waarvoor geldt

   $$r^{\frac{n-1}{2^i}} \bmod n = n - 1$$

   of als

   $$r^{\frac{n-1}{2^k}} \bmod n = 1$$

   dan is n waarschijnlijk priem, en anders zeker niet.
5. (multi-lengte rationale rekening) Pas het pakket `rationale rekenkunde` zo aan, dat het bouwt op het type `LINT` in plaats van `INT`. Onderzoek hiermee nogmaals de toepassingsvoorbeelden in dit hoofdstuk.

Chapter 8

Modulariteit

In dit hoofdstuk behandelen we de **modulaire programmering**, met als illustratie het ontwerp en de realisering van een modulaire digitale klok.

8.1 Modulaire programmering

Het woord *modulair* is al vrijwel even afgetrapt als de woorden “structuur” en “systeem”: van de zitmeubelen tot het onderwijsplan, alles is modulair.

Iets is modulair als het is samengesteld uit modulen, afgeronde gedeelten van (min of meer) gelijke grootte, met een welgedefinieerd grensvlak (“interface”) naar andere modulen. Maar wanneer is een gedeelte afgerond? In hoeveel en welke delen moet ik een geheel delen? Dat alles hangt er van af wat je er mee wilt. Een modulaire indeling is juist als ie ergens goed voor is.

Onder modulaire programmering verstaan we iedere programmeermethode waarbij een doelbewust gebruik gemaakt wordt van de indeling van een programma in een aantal (afgeronde) modulen. Daarbij kunnen verschillende doeleinden (en dus verschillende **modulariserings criteria**) worden nagestreefd:

- indeling van het programma in homogene, overzichtelijke delen
- tot een afzonderlijk geheel bundelen van declaraties die tezamen een bepaalde abstractie vormen
- verdeling van de ontwikkeling in zoveel mogelijk onafhankelijke taken
- indeling van het programma in delen met zo klein mogelijke grensvlakken
- tot een apart module maken van een gedeelte waar-
voor verschillende (meer en minder efficiënte of gecompliceerde) realiseringen verwacht worden
- indelen van het programma in kleinere, apart te vertalen delen, om tegemoet te komen aan beperkingen van een taal-implementering
- meermaals herbruikbare delen tot een apart module maken

en nog veel meer.

Al deze criteria zijn rijkelijk onprecies: ze geven slechts een gering houvast, zij leiden tot min of meer verschillende indelingen en iedere de facto getroffen modulaire indeling zal verschillende van deze aspecten pogen te combineren. In ieder geval gaat het om een indeling van zowel het programmeerproces als van zijn resultaat (het programma of systeem van programma’s met bijbehorende documentatie etc.), die men zoveel mogelijk poogt met elkaar in overeenstemming te brengen.

8.1.1 Modulen en abstracte datatypen

Abstracte datatypen zijn te zien als een bijzondere vorm van modulen: ga maar na in hoeverre ze aan elk van de bovengenoemde criteria beantwoorden. Een ADT heeft een naam, die gebruikt kan worden om willekeurig veel objecten van dat type te declareren. De waarde van zo’n object noem je dan een **instantie** van dat type.

Een **module** zullen we daarom opvatten als een abstract datatype waar maar één exemplaar (instantie) van bestaat, zoals bijvoorbeeld het bevolkingsregister van een bepaalde gemeente, de naamlijst van een ELAN-vertolker of een fysiek apparaat dat is aangesloten op de computer en dat toegesproken kan worden via een stel concrete algoritmen, zoals bijvoorbeeld het beeldscherm: omdat er maar één instantie van bestaat hoeven we deze niet expliciet als argument aan een operatie van dat type mee te geven. Dat ene exemplaar hoeft dan ook niet expliciet een type en naam te hebben, terwijl zijn waarde uitsluitend indirect, via bepaalde operaties, gemanipuleerd kan worden. Een module heeft daarmee een locaal geheugen, dat moet worden afgeschermd tegen onbedoelde accessen. Daarom zullen we een module inkapselen tot een pakket, van waaruit slechts enkele operaties naar buiten zichtbaar zijn.

Ook een abstract datatype realiseren we als een pakket, dat een type met de daarbijbehorende operaties naar buiten zichtbaar maakt.

8.1.2 Relaties tussen modulen

Er bestaat geen eenvoudig recept om te komen tot een optimaal modulair ontwerp. Wel kunnen we enige ideeën behandelen die daarbij als leidraad dienen, en mechanismen invoeren ter ondersteuning daarvan.

Alle benaderingen van modulariteit zijn gebaseerd op het onderscheid tussen een definitie-omgeving, die entiteiten exporteert, en een applicatie- omgeving, die ze importeert, als volgt:

- De auteur van de definitie-omgeving is wijs, almachtig en goedaardig. Hem (of haar) kan ieder geheim worden toevertrouwd, alle intieme kennis omtrent representaties etc. die nodig is om te komen tot een consistente en efficiënte implementering van de bedoelde abstracte entiteiten.
- De auteur van de applicatie-omgeving is hetzij volkomen incapabel, hetzij uiterst slim maar geheel boosaardig. Als er ook maar enigerlei manier bestaat waarop hij (of, wederom, zij) van de abstractie misbruik kan maken, de geheime inconsistenties daarvan kan opsporen, dan zal dat gebeuren. Deze persoon moet dus tegen zichzelf beschermd worden, en anderen tegen hem.

Deze defensieve houding verklaart de noodzaak tot **inkapseling** van de definitie-omgeving en de bescherming daarvan door middel van interfaces. Vertrouwen is goed, maar contrôle is nu eenmaal beter!

Het consequente onderscheid tussen definitie en applicatie wordt met name interessant als je zelf van beiden de auteur bent — een situatie waarin het veel te gemakkelijk is de zaak te bedriegen, door in de ene context informatie te gebruiken die aan de andere is voorbehouden. Op die manier kan een programma veel meer representatie- of zelfs machine-afhankelijk worden dan nodig, en daardoor moeilijk aan te passen aan andere representaties.

Het inkapselen van een definitie-omgeving tot een pakket met een expliciet export-interface geeft bescherming tegen het onbedoeld gebruik in de applicatie-omgeving van operaties uit de definitie-omgeving.

8.1.3 Interface en initialisatie

In het eenvoudigste geval worden in de applicatie-omgeving een of meer operaties gebruikt, afkomstig uit de definitie-omgeving, maar niet omgekeerd. Daartoe moeten deze operaties worden geëxporteerd (zichtbaar gemaakt) van de een naar de ander. De relatie tussen definitie- en applicatie-omgeving is dus een van **zichtbaarheid**, en de pijl tussen modulen is dus een generalisatie van de zichtbaarheids-pijl uit de applicatie-graph (zie 1.2). In ELAN wordt de definitie-omgeving daartoe tot een **pakket** gemaakt.

Om een voorbeeld te geven, het pakket `turtle graphics` exporteert een aantal algoritmen naar iedere omgeving die ze wil gebruiken, zoals aangegeven in het volgende interface.

```
PACKET turtle graphics
  DEFINES enter turtle graphics,
         leave turtle graphics,
         turtle window, move, draw,
         turn, turn right, turn left:
```

Gezamelijk hebben deze algoritmen een bepaald effect op de toestand van het beeldscherm, dat kan worden opgevat als de enige instantie van een hypothetisch datatype `SCREEN` (als we meer geld hadden hadden we best meer dan een instantie daarvan kunnen hebben), dat zodanig is geïmplementeerd dat een deel van zijn toestand voortdurend op het fysieke beeldscherm wordt weergegeven. Telkens als die toestand door uitvoering van een van de geëxporteerde algoritmen wordt veranderd verandert het schermbeeld navenant. Initieel, voordat de geëxporteerde algoritmen hun werk verrichten, moet de een of andere **module initialisatie** worden verricht, in dit geval de initialisatie van een paar variabelen verborgen in het pakket of zelfs in de machine waar het beeldscherm aan vast zit. Om dit te bereiken exporteert het pakket `turtle graphics` een initialisatie-algoritme `enter turtle graphics`, die moet worden uitgevoerd voordat een van de andere geëxporteerde algoritmen wordt uitgevoerd. Er bestaat hier ook een **module finalisatie** in de vorm van de algoritme `leave turtle graphics` die als laatste graphische algoritme moet worden opgeroepen, zodat het scherm van het graphisch gebruik kan bijkomen.

Uiteraard is het exporteren van noodzakelijke initialisatie- en finalisatie algoritmen een penibele zaak, omdat hun tijdige aanroep veel te gemakkelijk vergeten wordt. Zo kan het gebeuren dat het schermbeeld dat we juist met zoveel moeite hebben opgebouwd ontijdig wordt gewist.

De pakketten van ELAN kennen een veiliger mechanisme voor de initialisatie: als het pakket een niet-lege wortel bevat wordt deze wortel (in de syntax: de packet-paragraph met de bijbehorende verfijningen) uitgevoerd voorafgaand aan dat van de volgende pakketten, zodat de daarin gedeclareerde variabelen etc. geïnitialiseerd kunnen worden. Op deze wijze heeft een pakket als het ware zijn eigen locale privé geheugen.

ELAN
kent geen corresponderend finalisatie-mechanisme, er zijn trouwens maar weinig talen (zoals de implementeringstaal C++) die dit wel kennen. Voor een onderwijstaal als ELAN werd dit niet nodig geacht.

De EUMEL-implementering van ELAN kent het volledige pakket-mechanisme, en bleek daardoor machtig en flexibel genoeg om het hele EUMEL-bedrijfssysteem (dat zeer rijk is aan ondersteunende software) in de vorm van pakketten te implementeren.

8.2 Modulen in soorten

De boven beschreven dichotomie tussen definite en applicatie kan op een aantal manieren worden gebruikt, verfijnd of herhaald toegepast, leidend tot een klassificatie van soorten modulen.

8.2.1 Definitie modulen

De eenvoudigste vorm van module bestaat uitsluitend uit de encapsulatie van een aantal bij elkaar behorende declaraties, zonder initialisatie of finalisatie. Zo'n **definitie module** of **bibliotheek** dient uitsluitend de **herbruikbaarheid**. Het hoeft dus niet meer te zijn dan een bijeengeraapt allegaartje, maar bij Bottom-Up programmering kan hergebruik van bijvoorbeeld de C-bibliotheek of voorgedefinieerde widgets van cruciaal belang zijn. Vele programmeertalen ontlenen hun belang meer aan de uitgebreidheid van de standaard bibliotheek dan aan de schoonheid en expressiviteit van de taal.

8.2.2 Abstracte datatypen als modules

Een tweede, wat specifieker uitgangspunt voor modularisatie is het **abstracte (data)type** dat, zoals eerder aangegeven, kan worden opgevat als een module waar willekeurig veel instanties van kunnen bestaan. Een ADT kan worden geïmplementeerd als een pakket dat een (of meer) typenamen exporteert. Daardoor is kan gemakkelijk en veilig worden geabstraheerd van de gekozen representatie voor de instanties van het type (**representatie onafhankelijkheid**, **aanpasbaarheid**). De standaard pakketten voor de concrete typen zijn daarvan een voorbeeld.

Als het abstracte type in het definiërend module met `TYPE` gedeclareerd is wordt zijn representatie niet geëxporteerd — buiten dat module kan geen enkel gebruik gemaakt worden van eventuele kennis daaromtrent (**sterke abstractie**).

```
definierend module                applicatie module
PACKET ... DEFINES                het abstracte type
  ABC, veilige operaties:         ABC is zichtbaar,
 TYPE ABC =  ------------------>  tezamen met zijn
       realisering;               veilige operaties.
 onveilige operaties              Zijn realisering is
 veilige operaties ------------>  onzichtbaar.
 verborgen variabelen
 eventuele initialisatie
ENDPACKET
```

Het is ook mogelijk, een type te exporteren met **zwakke abstractie**, als een synoniem voor zijn realisering. In dat geval zijn ook alle operaties die uit zijn realisering voortvloeien op dit type van toepassing.

```
definierend module                applicatie module
PACKET ... DEFINES                het abstracte type
  ABC, veilige operaties:         ABC is zichtbaar,
 LET ABC =  ------------------->  tezamen met zijn
       realisering; ----------->  veilige operaties.
 onveilige operaties              Zijn realisering is
 veilige operaties ------------>  ook zichtbaar,
 verborgen variabelen             vanwege de LET.
 eventuele initialisatie
ENDPACKET
```

Vaak is het applicerend module zelf een ADT dat *bouwt op* een ander ADT, zoals de definitie van `TEXT` wel zal bouwen op die van `INT`. Bovendien kan een applicatie op meer dan een ADT bouwen, zoals het geval is in de laatste opgave bij het vorige hoofdstuk (zie fig. 8.1).

8.2.3 Alternatieve implementeringen

Een volgend modulariseringscriterium ligt daarin, een stuk programma tot module of ADT te maken als daarvoor meer dan een verschillende implementeringen zijn te voorzien. De beste raad in zo'n geval is:

> Kies eerst voor de simpelste implementering
> die met het interface in overeenstemming is;
> Pas later (als de applicatie correct is voltooid)
> en pas als dat niet te vermijden is
> moet je je wagen aan een gecompliceerdere implementering
> zonder daarbij het interface te wijzigen.

Het encapsuleren van een ADT is een toepassing van dit criterium.

8.2.4 Functionele modularisering

Het ligt voor de hand, te pogen een programma in te delen in een aantal *functionele modulen*, met elkaar communicerende modulen die ieder een bepaalde welomschreven functie of dienst vervullen.

```
invoer ------> verwerking ------> uitvoer
```

Op het eerste gezicht lijkt dit idee evident, maar: wat is communicatie? Wat betekenen de gestippelde opijlen in bovenstaand diagram? Invoer en uitvoer van bestanden, of gemeenschappelijke toegang tot globale variabelen, of wederzijdse (procedure-)oproepen?

8.2.5 Tweedimensionale modularisering

In plaats van het vage begrip "communicatie" kunnen we ook uitgaan van het eerder ingevoerde be-

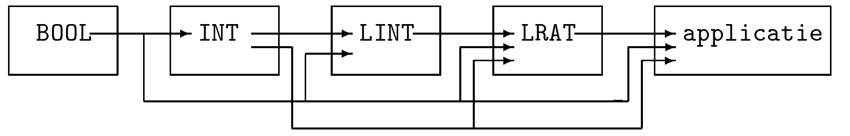
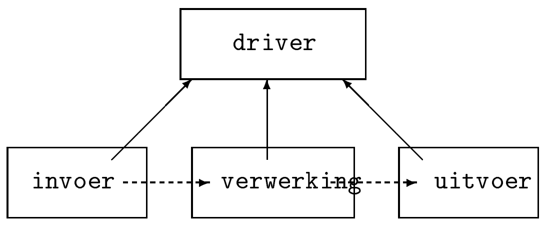


Figure 8.1: Op elkaar bouwende paketten

grip **zichtbaarheid**: De funtionele modulen exporteren ieder een aantal entiteiten die door een ander module (een **driver** of stuurprogramma) aan elkaar worden gekoppeld. De bedoelde communicatie ontstaat dan door het bereikte samenspel.



In deze opvatting is de indeling in modulen een partitionering van de applicatie-graph: Een module is een deel van de applicatie-graph en zijn exports de daarvan uitgaande zichtbaarheidspijlen.

8.3 Voorbeeld: digitaal horloge

Vijftien jaar geleden had ieder horloge nog een wijzerplaat waar wijzers over draaiden. Alleen de oudsten onder ons herinneren zich nog de komst van de eerste digitale horloges, exotische luxe voorwerpen, die er uit zagen zoals op plaatje 8.2 te zien is.

Figure 8.2: Digitaal horloge

- de klok geeft de tijd weer in een *tableau* van 4 cijfers.
- tussen het tweede en het derde cijfer wordt soms, bijwijze van scheider, een knipperende dubbele punt getoond.
- de klok heeft, in zijn simpelste versie, twee *drukknoppen*: met de ene knop kun je de weergegeven tijd wijzigen, met de andere de toestand van het horloge wijzigen.

Bij het horloge hoort een handleiding, geschreven in een bijzondere vorm van pidgin-engels dan wel door een bewoner van Taiwan met behulp van een vlaams dialecten woordenboek letterlijk uit het pidgin-engels vertaald. Aangezien deze handleiding volkomen onbegrijpelijk is kun je het beste door experimenten vaststellen hoe je de knoppen moet gebruiken:

- Als je herhaaldelijk op de ene drukknop (D in fig. 8.3) drukt, vertoont het tableau om de beurt de uren en minuten (met de dubbele punt ertussen), de dag en de maand (zonder dubbele punt ertussen) of de seconden.

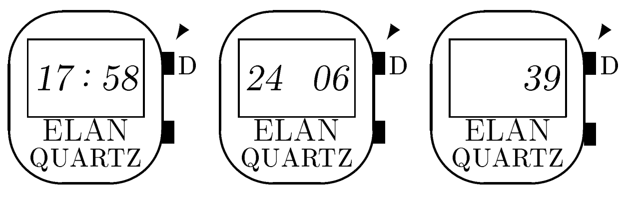


Figure 8.3: Uren & minuten, dag & maand, seconden

- Als je op de andere drukknop drukt (M in fig. 8.4) komt het horloge in de wijzig toestand. Het tableau vertoont nu alleen de maand en met de knop D kun je het maandnummer laten rondtellen. Als het nummer van de maand je bevalt druk je weer op M, waarna je op dezelfde wijze het dagnummer kunt wijzigen. Door verder op M te drukken kun je hetzelfde doen met de uren en de minuten. Tot slot komt het horloge terug in de weergave-toestand, waarbij het secondental automatisch op nul wordt gezet.

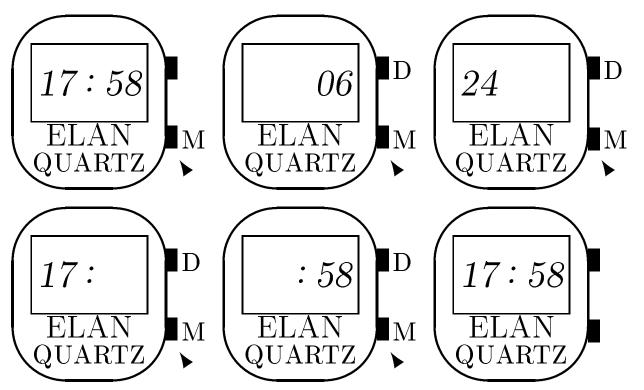


Figure 8.4: Via modify modes terug naar display mode

In deze sectie maken we een modulair ontwerp voor zo'n horloge en realiseren dit als een stelsel van pakketten.

Bij de realisering is gekozen voor engelse verfijningsen objectnamen. Om jargon te vermijden (als "displeejen" en zo) kiezen we in de begeleidende nederlandse tekst de volgende terminologie:

the display	het tableau; de weergave
to display, show	tonen, vertonen
the control	de besturing
to control	besturen
the display mode	de presentatiewijze
the colon	de dubbele punt

8.3.1 Modulaire structuur

In ieder geval moet een digitaal horloge een *uurwerk* hebben, met als taken:

- het bijhouden van de tijd, en
- het verwerken van de commando's, via drukknoppen gegeven.

Beide taken vereisen de kennis van toestandsvariabelen van het uurwerk, die overigens tegen nieuwsgierige blikken moeten worden afgeschermd. We verpakken het uurwerk in een module `clock`.

Het vertonen van de tijd op het *tableau* daarentegen, is een aparte taak, die zijn gegevens van het uurwerk krijgt. Daarbij zijn allerlei alternatieve manieren van weergave denkbaar, wat een goede reden is hiervan een apart module `display` te maken. We knopen deze modulen aan elkaar via een *besturingsmodule* die als "driver" dient.

Van het tableau zullen we eerst alleen de allersimpelste realisering geven, waarbij de tijd in cijfers op het beeldscherm van de computer verschijnt. In een later stadium, als alles goed werkt, zullen we aandacht besteden aan ingewikkelder (en meer realistische) realiseringen:

- tableau van grote cijfers, samengesteld uit losse tekens, weergegeven op het beeldscherm of op een regeldrukker
- tableau bestaand uit *7-segment displays* zoals die in de handel zijn, teneinde een zo realistisch mogelijk horloge te fabrieken.

8.3.2 De besturing

We vervolgen ons onderwerp met een Top-Down exploratie van de besturing, die ons ideeën moet opleveren over de export interfaces van de andere modulen.

Het hart van de besturing is een eindeloze herhaling, waarin telkens een seconde wegtikt.

```
program:
  initialize clock;
  init buttons;
  init display mode;
  init display;
  REP
    tick;
    may change display mode;
    display time;
    may change time
  ENDREP.
```

Over de initialisatie van één en ander zullen we het later hebben, bij elk van de modulen apart. De besturing heeft geen andere initialisatie nodig dan die van de twee modulen `clock` en `display`.

Het vertonen en bijhouden van de tijd behoort bij de taken van het uurwerk, gerealiseerd door het module `clock`.

8.3.3 Het uurwerk

Binnen het module `clock` wordt de tijd bijgehouden in een aantal variabelen die niet naar buiten zichtbaar zijn, voor ieder component van de tijd één.

Het hart van het uurwerk is heel simpel: telkens als een seconde voorbij is moet het secondental verhoogd worden, als een minuut voorbij is moet het minutental verhoogd worden, etc.

```
tick:
  IF one second is over
  THEN
    increment seconds;
    IF one minute is over
    THEN
      increment minutes;
      IF one hour is over
      THEN
        increment hours;
        IF one day is over
        THEN
          increment day;
          IF one month is over
          THEN increment month
          FI
        FI
      FI
    FI
  FI.
```

Wat een prachtige waterval van sluithaakjes!

Het voorbijgaan van de tijd zouden we kunnen modelleren door een of andere herhaling uit te voeren die precies de verlangde tijd kost; in plaats daarvan gebruiken we de ingebouwde klok van de computer door een oproep van de standaard procedure `sleep(1)`. Helaas komt de uitvoeringstijd van ons programma daarmee nog bovenop de ene seconde vertraging, zodat onze klok steeds verder achterloopt, maar daar trekken we ons niet veel van aan: het gaat toch maar om een **simulatie**.

```
one second is over:
  sleep (1);
  true.

one minute is over:
  seconds = 0.

one hour is over:
  minutes = 0.

one day is over:
  hours = 0.

one month is over:
  day = 1.
```

Seconden en minuten liggen in het gebied van 0 tot 59, uren van 0 tot 23. Maanden lopen van 1 tot 12 en dagen van 1 tot een bovengrens die van de maand afhangt. Al deze getallen moeten worden verhoogd op de manier die "klokrekenen" genoemd wordt.

Om niet overal op bovengrenzen te hoeven testen maken we daartoe twee hulpprocedures, `increment 0` voor het tellen tussen 0 en `limit` exclusief en `increment 1` voor het tellen tussen 1 en `limit` inclusief.

```
PROC increment 1 (INT VAR n, INT limit):
  n := n MOD limit + 1
ENDPROC increment 1;

PROC increment 0 (INT VAR n, INT limit):
  n := (n + 1) MOD limit
ENDPROC increment 0;
```

Met behulp hiervan kunnen we verfijnen:

```
increment seconds:
  increment 0 (seconds, 60).

increment minutes:
  increment 0 (minutes, 60).

increment hours:
  increment 0 (hours, 24).

increment day:
  increment 1 (day, last day of month).

increment month:
  increment 1 (month, 12).
```

Bij het berekenen van de laatste dag van de maand houden we (net als alle goedkope horloges) geen rekening met de schrikkeljaren. Om de vier jaar moet de gebruiker het horloge maar gelijkzetten.

```
last day of month:
  SELECT month OF
    CASE 2: 28
    CASE 4, 6, 9, 11: 30
    OTHERWISE 31
  ENDSELECT.
```

Bijwijze van initialisatie kiezen we startwaarden voor de interne variabelen.

```
initialize clock:
  INT VAR month :: 6, day :: 24,
          hours :: 17,
          minutes :: 58, seconds :: 0.
```

Veranderen van de presentatiewijze

Iedere seconde geven we de gebruiker de gelegenheid, de aangegeven tijd te wijzigen of de presentatiewijze van het tableau (de *displaymode*).

Ons horloge kent drie verschillende wijzen om de tijd weer te geven:

- uren en minuten
- dagen en maanden
- alleen de seconden

Deze verschillende presentatiewijzen kunnen we cyclisch doorlopen door op knop D te drukken.

```
may change display mode:
  IF button d is in effect
  THEN increment display mode
  FI.
```

We houden de presentatiewijze bij in een variabele.

```
init display mode:
  LET hours and minutes = 1,
      day and month = 2,
      seconds only = 3,
      nmb of display modes = 3;
  INT VAR display mode :: hours and
minutes.

increment display mode:
  increment 1 (display mode,
      nmb of display modes).
```

8.3.4 Tonen van de tijd

Bij het realiseren van de manier waarop de tijd wordt weergegeven moeten we een interface kiezen voor het tableau. Dat bestaat uit drie delen, een 2-cijferig getal, een enkel teken als separator en nog een tweecijferig getal.

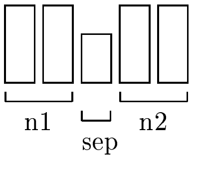


Figure 8.5: Onderdelen van het tableau

We willen in staat zijn alle drie deze delen te gebruiken, via een procedure

```
PROC show (INT n1, TEXT sep, INT n2):
  start display;
  put 2 digits (n1);
  put separator (sep);
  put 2 digits (n2)
ENDPROC show;
```

of één van de twee getallen kunnen weglaten (voor de verschillende bijzondere weergavewijzen). Meestal zal de dubbele punt als separator optreden, willen we hem weglaten dan geven we een spatie als separator mee. We hebben dus nog twee gespecialiseerde vertoningsprocedures nodig

```
PROC show (TEXT sep, INT n2):
  start display;
  put 2 spaces;
  put separator (sep);
  put 2 digits (n2)
ENDPROC show;

PROC show (INT n1, TEXT sep):
  start display;
  put 2 digits (n1);
  put separator (sep);
  put 2 spaces
ENDPROC show;
```

We hadden een ander interface kunnen kiezen, bijvoorbeeld met een negatief getal als parameter ingeval die parameter moet worden weggelaten, maar dit soort coderingstrucs maakt het programma er niet helderder op.

We hadden, al Top-Down programmerend, ook het tableau via verfijningen kunnen realiseren, communicerend via globale variabelen, maar zoals aangegeven in hoofdstuk 2 van dit boek dreigt dan ons programma te verworden tot een grote slappe pudding van verfijningen.

Het hier aangegeven, betrekkelijk eenvoudige interface, bestaande uit enkele procedures met parameters, is voldoende algemeen en stelt ons in staat, de realisering daarvan in een apart module onder te brengen.

8.3.5 Wijzigen van de tijd

Als de gebruiker op de *M* knop drukt kan hij de aangegeven tijd wijzigen, één component tegelijk.

```
may change time:
  IF button m is in effect
  THEN
    adjust month;
    adjust day;
    adjust hours;
    adjust minutes;
    reset seconds
  FI.
```

Bij het wijzigen van de componenten krijgen we ze om de beurt te zien, waarbij de component wordt verhoogd telkens als we op *D* drukken. Door weer op *M* te drukken geven we aan dat we klaar zijn met een component. Tussen het drukken van de knoppen kunnen we rustig de tijd nemen - het programma laat geduldig de betrokken component zien.

```
adjust month:
  show month;
  REP
    IF button d is in effect
    THEN
      increment month;
      show month
    FI
  UNTIL button m is in effect
  ENDREP.
```

Bij het vertonen van de maand willen we de dubbele punt en de overige componenten achterwege laten.

```
show month:
  show ("", month).
```

Voor de dagen gaan we analoog te werk.

```
adjust day:
  show day;
  REP
    IF button d is in effect
    THEN
       increment day;
       show day
    FI
  UNTIL button m is in effect
  ENDREP.

show day:
  show (day, "").
```

De overige verfijningen `adjust...` spreken voor zich, alleen het vertonen van de andere componenten verdient nog nadere uitwerking.

```
show hours:
  show (hours, ":").

show minutes:
  show (":", minutes).
```

Tot slot wordt het secondental op nul gezet.

```
reset seconds:
  seconds := 0.
```

8.3.6 Tussenbalans

We hebben een module `clock` ontworpen dat gebruik maakt van twee andere modulen, `display` en `button`, via interfaces bestaande uit enkele procedures.

Binnen de module `clock` zijn duidelijk sporen te vinden van een verdere indeling in drie functionele delen.

8.3.7 De drukknoppen

We realiseren de drukknoppen in een apart module `buttons`, omdat we ons daarvan weer verschillende realiseringen kunnen voorstellen, gebruik makend van verschillende hardware voorzieningen (**devices**).

In de simpelste versie worden de drukknopen vervangen (gesimuleerd) door middel van toetsen op ons toetsenbord. Maar als we gewoon via `get` van het toetsenbord zouden lezen zou het programma telkens gaan staan wachten tot de inputbuffer is afgesloten met een RETURN-toets. Men spreekt van "synchrone communicatie". Onze gesimuleerde klok wordt daardoor een farce!

Goed nieuws! In de standaard bibliotheek bevindt zich ook een `TEXT PROC incharety` die bij iedere oproep zonder te wachten de momentane inhoud van de inputbuffer oplevert en deze schoonmaakt ("cleart"; "kliert"?). Als er sinds de vorige leesactie geen teken is ingevoerd levert deze procedure de lege tekst. Met deze asynchrone leesactie kunnen we de knoppen zeer eenvoudig simuleren.

De bizarre naam van de procedure kan worden gezien als een contractie van `inchar or empty` en inderdaad bestaat er ook een procedure `inchar`, een synchrone versie van `incharety` die eerst wacht tot er een toets is aangeraakt.

```
PACKET buttons
  DEFINES button d is in effect,
          button m is in effect:


BOOL PROC button m is in effect:
  inspect buttons;
  IF button m
  THEN
   reset (button m);
   true
  ELSE false
  FI
ENDPROC button m is in effect;
```

Bij het inspecteren van de knoppen moeten we telkens naar allebei kijken en beide onthouden, omdat (bij de gekozen realisering) het kijken naar een knop deze afzet. We kunnen dus niet eerst naar de ene knop en dan weer naar de andere kijken.

```
PROC inspect buttons:
  TEXT CONST buttons :: incharety;
  IF pos (buttons, "m") > 0
  THEN set (button m)
  FI;
  IF pos (buttons, "d") > 0
  THEN set (button d)
  FI
ENDPROC inspect buttons;
```

Gezien de frequentie (1 x per seconde) waarmee de knoppen geïnspecteerd worden zal het geen zin hebben, snel achter elkaar op de knoppen te drukken.

We voltooien dit pakket

```
BOOL PROC button d is in effect:
  inspect buttons;
  IF button d
  THEN
   reset (button d);
   true
  ELSE false
  FI
ENDPROC button d is in effect;

PROC set (BOOL VAR b):
  b := true
ENDPROC set;

PROC reset (BOOL VAR b):
  b := false
ENDPROC reset;

init buttons:
  BOOL VAR button m :: false, button d
:: false.
ENDPACKET buttons;
```

8.3.8 Het module `display`

Voor het module `display` zullen we een aantal alternatieve implementeringen bekijken, met alle hetzelfde interface, de generieke procedure `show`.

```
PACKET display DEFINES show:
```

Wat moeten de procedures `show` doen? Ze vertonen (van links naar rechts) 2 getallen van 2 cijfers elk, met daartussen één scheidingsteken. Van de twee getallen kan er telkens één door 2 spaties worden vervangen.

Deze beschouwing suggereert een soort ontleding van de actie `show` in simpeler operaties.

We schrijven op het tableau van links naar rechts, zoals bij uitvoer gebruikelijk, en hebben daartoe twee acties nodig:

```
PROC start display:
PROC put char (TEXT c):
```

Om links te beginnen op een schoon tableau respectievelijk een teken op het tableau te schrijven. Onder gebruik van deze procedures kunnen we schrijven:

```
PROC show (INT n1, TEXT sep, INT n2):
  start display;
  put 2 digits (n1);
  put separator (sep);
  put 2 digits (n2)
ENDPROC show;

PROC show (INT n1, TEXT sep):
  start display;
  put 2 digits (n1);
  put separator (sep);
  put 2 blanks
ENDPROC show;
```

```
PROC put 2 blanks:
  put blank;
  put blank
ENDPROC put 2 blanks;

PROC put blank:
  put char (blank)
ENDPROC put blank;

PROC show (TEXT sep, INT n2):
  start display;
  put 2 blanks;
  put separator (sep);
  put 2 digits (n2)
ENDPROC show;

PROC put 2 digits (INT n):
  assert (0 <= n AND n < 100);
  IF n >= 10
  THEN
    put char (n DIV 10);
    put char (n MOD 10)
  ELSE
    put blank;
    put char (n)
  FI
ENDPROC put 2 digits;

PROC put separator (TEXT sep):
  IF sep = ":"
  THEN put char (colon)
  ELSE put char (blank)
  FI
ENDPROC put separator;
```

Voor onjuiste waarden van de parameter `n` wordt de uitvoering afgebroken.

Daarmee is het pakket compleet, op de initialisatie na.

```
init display:
  start display.
```

Het simpelste tableau

Bij de simpelste implementering van het tableau gebruiken we simpelweg de bovenste regel van het beeldscherm, die we tevoren schoonmaken.

```
PACKET small character display
  DEFINES start display, put char:

PROC start display:
  page
ENDPROC start display;

PROC put char (INT c):
  put ("0123456789 :" SUB c + 1)
ENDPROC put char;

ENDPACKET small character display;
```

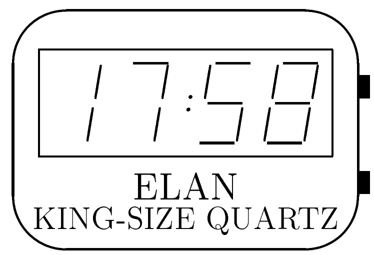
Het resultaat is wat magertjes, maar we hebben tenminste een werkende klok.

Fraaiere tableaus

Op dit punt aangeland is het heel eenvoudig, bij ditzelfde interface een fraaier tableau te programmeren: We kunnen de tijd in grote letters en cijfers op het beeldscherm weergeven door deze uit X-en en spaties samen te stellen, of door graphics te gebruiken. We kunnen ze ook afdrukken op een printer, of zelfs een “echte” display aansturen, zoals die in een digitale klok gebruikt wordt.

```
  11   7777777          5555555    88888
 1 1        7    :::    5         8
8
1  1       7     :::    55555     8
8
   1      7                  5     88888
   1     7      :::          5   8
8
   1    7       :::          5   8
8
   1   7                555555     88888
```

17:58



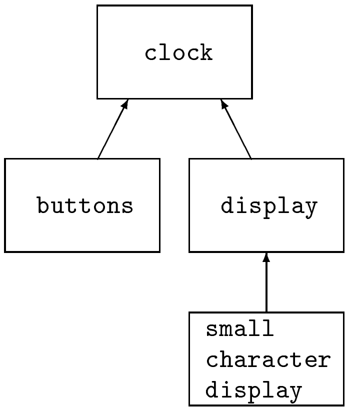
Deze verfraaiingen van het tableau laten we echter aan de lezer over — het voorbeeld is nu wel voldoende uitgediept.

8.4 Conclusie

In een vrij groot voorbeeld hebben we een aantal vormen van modularisatie aan het werk gezien:

- Het module `clock` is te zien als een ADT met maar één instantie. Datzelfde geldt ook voor de andere modulen.
- Het tableau is geïmplementeerd als twee op elkaar bouwende modulen, waarbij het module `display` die algoritmen bevat die van de weergave-machinerie onafhankelijk zijn.
- Van het module `small character display` zijn vele alternatieve implementeringen te bedenken.

De modulaire structuur van het programma is als volgt:

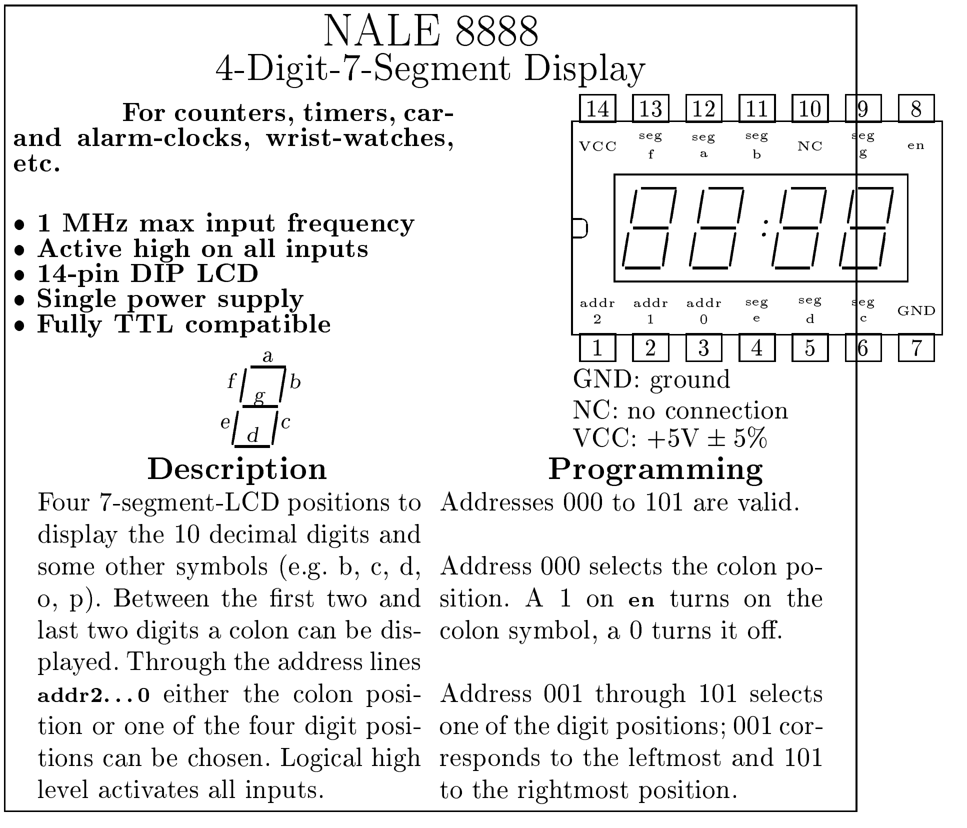


Modularisatie is een techniek die met name van belang is bij het ontwerp van grote systemen, veel groter dan ons voorbeeld. Welbeschouwd was het ontwerp van de klok een vrij simpele Top-Down aangelegenheid. Door de indeling in modulen is echter ook in dit simpele geval een aantal voordelen behaald boven een “monolithische” aanpak:

- het systeem bestaat uit niet te grote, coherente en overzichtelijke stukken die gemakkelijk correct te krijgen, te begrijpen en te wijzigen zijn.
- eenvoudige aanpasbaarheid aan andere weergavemechanismen.
- eenvoudige aanpasbaarheid van de geleverde functies (toevoegen van meer knoppen en diensten).

8.5 Opgaven

1. (regeldrukker) Vervang de module `small character display` bij gelijkblijvend interface door een andere, zodanig dat het tableau in grote tekens op het beeldscherm wordt weergegeven.
2. (NALE 8888) Doe hetzelfde, maar nu zodanig dat de 7-segments-display NALE 8888, waarvan u de gegevens vindt in fig. 8.6, kan worden aangestuurd. Simuleer dit display in een afzonderlijk module, bouwend op de schildpadgrafiek.
3. (oscilloscoop) Ontwerp een modulair programma, bouwend op de schildpad- grafiek, om de impedantie van een serie/parallel netwerk in een fraaie grafiek uit te zetten, met geannoteerde assen.


NALE 8888
4-Digit-7-Segment Display

For counters, timers, car- and alarm-clocks, wrist-watches, etc.

- **1 MHz max input frequency**
- **Active high on all inputs**
- **14-pin DIP LCD**
- **Single power supply**
- **Fully TTL compatible**

GND: ground
NC: no connection
VCC: +5V ± 5%

Description

Four 7-segment-LCD positions to display the 10 decimal digits and some other symbols (e.g. b, c, d, o, p). Between the first two and last two digits a colon can be displayed. Through the address lines **`addr2...0`** either the colon position or one of the four digit positions can be chosen. Logical high level activates all inputs.

Programming

Addresses 000 to 101 are valid.

Address 000 selects the colon position. A 1 on **`en`** turns on the colon symbol, a 0 turns it off.

Address 001 through 101 selects one of the digit positions; 001 corresponds to the leftmost and 101 to the rightmost position.

Figure 8.6: NALE 8888

Chapter 9

Lijsten

Rijen zijn uiterst beperkte structuren: het aantal en type van hun elementen ligt vast. In dit hoofdstuk voeren we datastructuren in waarvan het aantal elementen vrij is. Ze hebben echter met de rij gemeen dat alle elementen van eenzelfde type zijn en dat de elementen een geordende eindige verzameling vormen: we kunnen spreken van het *eerste*, *tweede*, *laatste*, *n-de*, *volgende* en *vorige* element.

Ten opzichte van de rijen zullen deze lineaire lijststructuren twee belangrijke verschillen vertonen:

1. het aantal van hun elementen kan dynamisch worden uitgebreid of verminderd
2. op ieder moment is slechts acces mogelijk op een beperkt aantal van de elementen, bijvoorbeeld op het eerste en eventueel het laatste element.

We zullen een aantal klassieke lineaire lijststructuren invoeren en hun implementatie beschrijven.

9.1 Een uitleenbibliotheek

Eerst een inleidend voorbeeld om enige problemen en begrippen van lijststructuren te verduidelijken. We willen een eenvoudige administratie ontwerpen voor een *bibliotheek* die een aantal *boeken* bevat. Een boek heeft een *schrijver* en een *titel*. Van ieder boek is een bepaald *aantal* exemplaren aanwezig. De boeken van eenzelfde schrijver verschillen in titel. Verschillende schrijvers kunnen echter eventueel dezelfde titel gebruiken (ook dan gaat het om verschillende boeken).

We willen een zodanige representatie van de bibliotheek bedenken dat allerlei voor een uitleenbibliotheek typische operaties daarop mogelijk zijn:

- toevoegen van een nieuwe schrijver (met vooreerst nul boeken)
- toevoegen van een boek bij een schrijver (vooreerst in nul exemplaren)
- opsommen van alle boeken van een schrijver
- boeken inkopen, uitlenen, retourneren, afschrijven etc.

Om deze administratie te realiseren kunnen we natuurlijk alle boeken, eventueel zelfs ongeordend, in een lange rij onderbrengen. Het nadeel daarvan is dat we, als we een boek van een bepaalde schrijver zoeken, de boeken van vele andere schrijvers moeten overslaan. Daarom organiseren we de bibliotheek zó, dat we een bepaald aantal schrijvers bijhouden, en voor ieder van die schrijvers de bijbehorende boeken (het *oeuvre*).

We laten ten hoogste 100 verschillende schrijvers toe, een aantal dat we bij gebleken behoefte kunnen verhogen. We beginnen met het schrijvers-gedeelte

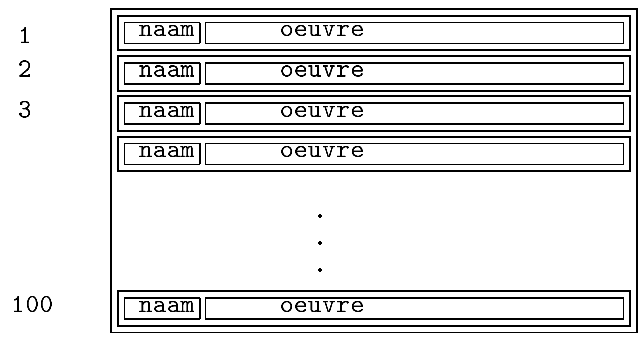


Figure 9.1: Opzet van de administratie

van de administratie.

```
  LET max schrijvers = 100;
  LET AUTEUR = STRUCT (TEXT naam,
BOEKENLIJST oeuvre);
  ROW max schrijvers AUTEUR VAR
bibliotheek;
  INT VAR aantal schrijvers :: 0; { <=
100 }
```

Nu rijst de vraag: hoe realiseer je de boekenlijsten van de verschillende schrijvers?

9.1.1 Representatie als rij

Als het om een vast en gelijk blijvend aantal boeken per schrijver gaat, bijvoorbeeld 25, dan kunnen we kiezen

```
  LET aantal boeken = 25;
  LET BOEKENLIJST = ROW aantal boeken
BOEK;
```

Dit is echter een weinig flexibele opzet: het ligt niet voor de hand dat alle schrijvers precies evenveel boeken produceren. We zullen het aantal boeken per schrijver willen kunnen uitbreiden en (bij definitief verlies, vernieling of verbranding) verminderen. We kiezen daarom liever een opzet waarbij het maximum aantal boeken per schrijver vast gekozen is, maar het aantal feitelijk aanwezige boeken per schrijver mag variëren.

We vervolgen het programma:

```
  LET max boeken = 25 { per schrijver };
  LET BOEKENLIJST = STRUCT (INT aantal,
                            ROW max
boeken BOEK lijst);
  LET BOEK = STRUCT (TEXT titel, INT
voorraad);
```

We zullen met deze representatie niet een heel programma schrijven, de lezer is zelf wel in staat te verzinnen welke operaties moeten worden ingevoerd.

Bij de hier gekozen representatie kunnen we hoogstens 100 schrijvers bolwerken, die ieder ten hoogste 25 boeken op hun naam hebben. Weliswaar zijn deze begrenzingen ieder voor zich vrij eenvoudig te wijzigen, maar (ook al omdat we allerlei tellingen moeten bijhouden) flexibiliteit en helderheid laten te wensen over.

9.1.2 Representatie met verwijzingen

Een andere oplossing krijgen we als we voor iedere auteur in plaats van zijn boeken een *verwijzing* daarnaar opnemen.

Als verwijzing naar een boekenlijst gebruiken we zijn index in de rij van boeken, een geheel getal dus, dat aangeeft vanaf welke plaats in de kast zijn boeken staan opgesteld.

```
  LET max schrijvers = 100;
  LET AUTEUR = STRUCT(TEXT naam,
BOEKENLIJST oeuvre);
  ROW max schrijvers AUTEUR VAR
bibliotheek;

  LET max boeken = 2000; { of zo }
  LET BOEKENLIJST = INT; {tussen 1 en
max boeken}
  LET BOEK = STRUCT (TEXT auteur, titel,
INT voorraad);
  ROW max boeken BOEK VAR de kast;
```

Bij deze representatie heeft het toevoegen of verwijderen van een boek erg veel voeten in de aarde, al is er geen individuele bovengrens op het aantal boeken per schrijver; we zullen hem daarom niet verder uitwerken.

Een bijzonderheid van deze representatie is dat we nu de situatie kunnen uitdrukken dat een schrijver een *pseudoniem* heeft: twee verschillende namen voor dezelfde persoon, zodat bij beide namen dezelfde boekenlijst hoort: We geven dan beide namen dezelfde verwijzing.

Het begrip verwijzing (zie verder 11) is een sleutelbegrip dat ten grondslag ligt aan het vervolg van dit hoofdstuk. Daarbij zullen we echter niet verder kijken naar representaties met expliciete verwijzingen (die nogal wat kunst en vliegwerk verlangen) maar een nieuwe ELAN constructie invoeren die gebaseerd is op verborgen verwijzingen (d.w.z. verwijzingen die niet als zodanig herkenbaar en toegankelijk zijn, maar beschermd en onzichtbaar optreden).

9.2 Recursieve datatypen

Een **lineaire lijst** L van elementen kunnen we recursief definiëren als

- een *lege* lineaire lijst, of
- een element (het *eerste element* van L), en een lijst van de *overige elementen* van L.

Wij kunnen deze recursieve definitie rechtstreeks modelleren in ELAN, en wel als volgt:

```
TYPE LIST = STRUCT (EL head, LIST tail);
```

Hoewel een recursieve typedeclaratie in standaard ELAN niet is toegestaan zullen wij gebruik maken van een uitbreiding van ELAN met **recursieve typen**, speciaal bedoeld voor het onderwijs in datastructuren.

Als `l` een lijst is van het zojuist gedefinieerde type `LIST` dan is `l.head` van het type `EL` en `l.tail` van het type `LIST`. Het tweede deel van de recursieve typedefinitie is dus goed gemodelleerd, hoe zit het met het eerste? Hoe kunnen we een **lege lijst** representeren?

9.2.1 NIL en de lege lijst

In het domein van ieder abstract type is een bijzondere waarde aanwezig, die *nil* genoemd wordt. Waar een abstracte waarde verlangd wordt kunnen we de waarde *nil* denoteren met behulp van `NIL`, een constructie die zelf geen eigen type heeft maar onder dwang van een abstractor ieder abstract type wil aannemen. Om na te gaan of een gegeven abstracte waarde die van `NIL` is kunt u de concrete operator `ISNIL` op zijn realisering gebruiken, met een boolean resultaat.

Wat zich achter deze merkwaardige constructen verbergt is dat alle abstracte typen door tussenkomst van een verwijzing gerealiseerd worden, waarbij de nil een bijzondere, herkenbare verwijzing is. U kunt overigens op geen enkele wijze ontdekken dat er sprake is van een verwijzing en op welke wijze deze gerealiseerd is, hij blijft geheel onder de tafel. Met name de toewijzing gedraagt zich alsof er geen sprake is van een verwijzing. Toewijzing van een waarde (ook van een abstract type) betekent altijd toewijzing van een <u>kopie</u> daarvan, het origineel blijft onaangeroerd.

Wel verklaart dit waarom een recursief datatype een eindige representatie heeft: de recursie kan altijd met een nil-waarde worden afgebroken.

Een conclusie hieruit is dat NIL een uitstekende waarde levert ter representatie van de lege lineaire lijst.

9.3 Lineaire lijsten

We willen een type LIST invoeren, gerealiseerd als recursief datatype.

```
TYPE LIST = STRUCT (EL head, LIST tail);
```

Traditioneel heeft een lijst een signatuur die wel wat weg heeft van die van TEXT. Om te beginnen kiezen we een denotatie voor de lege lijst, en een test om te bepalen of een lijst leeg is.

```
LIST CONST empty list :: LIST : NIL;

BOOL PROC is empty (LIST l):
  ISNIL CONCR l
ENDPROC is empty;
```

Let op het gebruik van de abstractor en de concretizer. Dit zijn overigens de enige plaatsen in de realisering van dit type waar we NIL en ISNIL moeten gebruiken. Zoals u ziet geldt `is empty (empty list)`. We definiëren de beide accesoperatoren.

```
EL OP HEAD (LIST l):
  assert (NOT is empty(l));
  l.head
ENDOP;

LIST OP TAIL (LIST l):
  assert (NOT is empty (l));
  l.tail
ENDOP;
```

Voor de denotatie van lijsten voeren we twee operaties in. De eerste operator construeert een lijst bestaande uit één element (een **singleton lijst**).

```
LIST PROC list (EL a):
  LIST: [a, empty list]
ENDPROC list;
```

De tweede operator (de **links-concatenatie**) voegt een element voor aan een lijst.

```
LIST OP + (EL a, LIST b):
  LIST: [a, b]
ENDOP;
```

Tesamen met de denotatie voor de lege lijst noemt men deze operaties ook de **bouw operatoren** van het lijst-type aangezien iedere lijstwaarde daarmee kan worden opgebouwd.

9.3.1 Lijsten tekenen

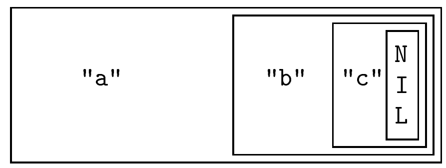
Lijsten kunnen op een aantal verschillende wijzen graphisch worden weergegeven. Beschouw bijvoorbeeld de lijst gebouwd door

```
"a" + ("b" + ("c" + empty list))
```

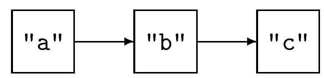
of door

```
"a" + ("b" + list ("c"))
```

Strict genomen zou je deze lijst het beste kunnen weergeven als

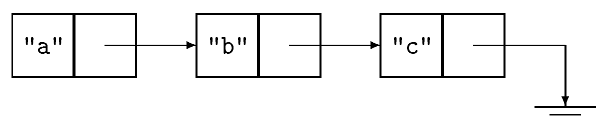


maar aangezien het tekenen van dit soort geneste plaatjes bij een wat groter aantal elementen niet meevalt zullen we liever een wat conventioneler graphische representatie gebruiken, namelijk



De gebruikte pijl geeft hier de *opvolger-relatie* weer.

Daarnaast is er nog een derde representatie in omloop



waarbij de pijl iets prijsgeeft omtrent de interne realisering van het lijst-type, namelijk het feit dat de elementen van een lijst intern via *verwijzingen* ("pointers") aan elkaar verbonden worden. Meer hierover in hoofdstuk 11.

9.3.2 Verdere lijst-operaties

De vergelijkingsoperatoren < en = voor lijsten zijn gebaseerd op die van de elementen.

```
BOOL OP = (LIST l, r):
  IF is empty (l)
  THEN is empty (r)
  ELIF is empty (r)
  THEN false
  ELIF HEAD l = HEAD r
  THEN TAIL l = TAIL r
  ELSE false
  FI
ENDOP =;
```

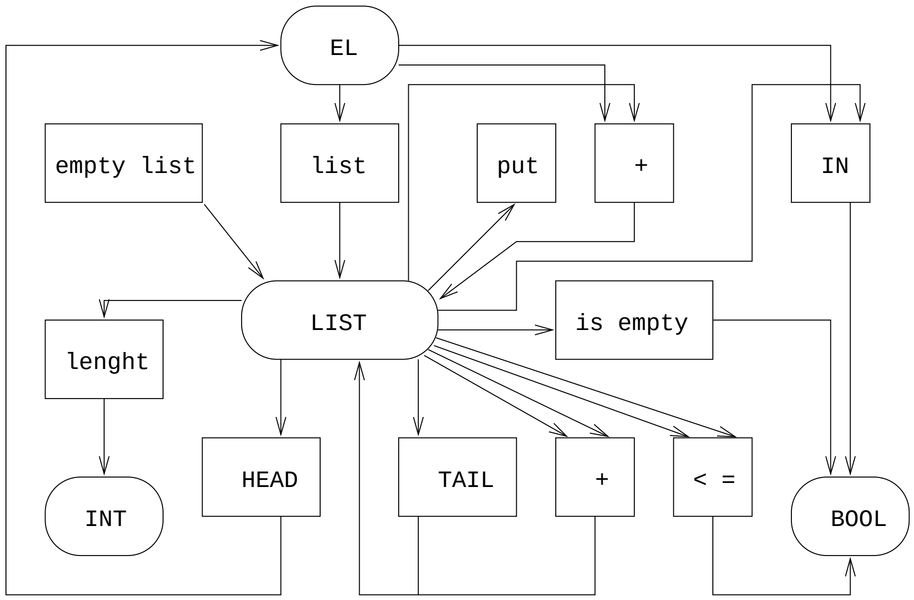


Figure 9.2: Signatuur van lineaire lijst

```
BOOL OP < (LIST l, r):
  IF is empty (l)
  THEN NOT is empty (r)
  ELIF is empty (r)
  THEN false
  ELIF HEAD l < HEAD r
  THEN true
  ELIF HEAD l > HEAD r
  THEN false
  ELSE TAIL l < TAIL r
  FI
ENDOP <;
```

Hiermee hebben we de grond-vorm van een lineaire lijst geïmplementeerd. Merk op hoeveel gelijkenis dit type vertoont met dat van de teksten: een tekst is blijkbaar een lineaire lijst van tekens.

Uit de aan het begin van dit hoofdstuk gegeven definitie van lineaire lijst kunnen we nu gemakkelijk een recursieve procedure afleiden om de lengte van een lijst te berekenen.

```
INT PROC length (LIST l):
  IF is empty (l)
  THEN 0
  ELSE 1 + length (TAIL l)
  FI
ENDPROC length;
```

Deze recursieve oplossing is te verkiezen boven een iteratieve zoals

```
INT PROC length (LIST l):
  LIST VAR l1 :: l;
  INT VAR result :: 0;
  WHILE NOT  is empty (l1)
  REP result INCR 1;
      l1 := TAIL l1
  ENDREP;
  result
ENDPROC;
```

De iteratieve versie maakt eerst een kopie `l1` van de lijst `l` en breekt deze dan af. Er worden twee hulpvariabelen gebruikt, de formulering is langer (en daarmee gevoeliger voor vergissingen) dan de recursieve en het is sterk de vraag of hij efficiënter is (dat hangt van details van de ELAN-implementering af). In ieder geval maakt het niet veel verschil.

De recursieve oplossing is daarentegen veel eenvoudiger van structuur, omdat hij de recursieve structuur van het datatype volgt, en heeft een grotere overtuigingskracht. Bij het werken met recursieve datastructuren geven we steeds de voorkeur aan recursieve algoritmen boven iteratieve, met (buiten de kop van de algoritme) zo min mogelijk variabelen. Dit wordt nog in de hand gewerkt door het ontbreken in ELAN van "pointers". Hiermee lopen we vooruit op een programmeerstijl waarin geen variabelen en toewijzingen optreden, een *applicatieve* ("logische" of "functionele") stijl.

Beschouw nu het probleem twee lijsten aaneen te voegen tot een nieuwe lijst, waarvan de elementen gelijk zijn aan de elementen van de eerste lijst, gevolgd door de elementen van de tweede.

Wat preciezer, onder gebruikmaking van een hopelijk voor zich sprekende notatie voor lijsten: zij

`lijst1` $= < e_1, e_2, \ldots, e_n >$, $n > 0$
`lijst2` $= < f_1, f_2, \ldots, f_m >$, $m > 0$

dan is

$$\texttt{lijst1 + lijst2} = \\ < e_1, e_2, \ldots, e_n, f_1, f_2, \ldots, f_m >$$

en

$$\texttt{length (lijst1 + lijst2)} = n + m.$$

We realiseren de lijst-concatenatie recursief, gebruik makend van de eerder ingevoerde links-concatenatie.

```
LIST OP + (LIST l1, l2):
  IF is empty (l1)
  THEN l2
  ELSE HEAD l1 + (TAIL l1 + l2)
  FI
ENDOP;
```

Hierbij maken we gebruik van de eerder gedefinieerde operator `+` tussen een `EL` en een `LIST`. Een ietwat efficiëntere oplossing (die echter een groter beroep doet op kennis over de representatie van lijsten) is

```
LIST OP + (LIST l1, l2):
  IF is empty (l1)
  THEN l2
  ELSE LIST: [HEAD l1, TAIL l1 + l2]
  FI
ENDOP;
```

Merk op dat iets vóór een lijst plakken veel eenvoudiger is dan iets achter een lijst te plakken, de gekozen representatie (en ook de gegeven definitie van lineaire lijst) heeft een voorkeur voor manipulatie van de kop van een lijst. Dit komt naar voren bij de realisering van een tweetal procedures om een element voor resp. achter aan een lijst-variabele te hangen.

```
PROC prepend (LIST VAR l, EL a):
  l:= LIST: [a, l]
ENDPROC prepend;

PROC postpend (LIST VAR l, EL a):
  IF is empty (l)
  THEN l:= list (a)
  ELSE postpend (l.tail, a)
  FI
ENDPROC postpend;
```

Het effect van de laatste bestaat uit het (recursief) aflopen van de lijst tot aan zijn laatste element, waarvan de staart wordt overschreven met de singleton-lijst `a`. Merk op hoe essentieel hier het gebruik van de selectie is, de acces operator `TAIL` had ons enkel een lijst-constante geleverd (een kopie van de staart) terwijl de selectie ons een variabel acces op de staart verleent. Vanwege het beroep dat hij doet op de representatie van lijsten kan de procedure op deze manier enkel gedeclareerd worden binnen het pakket waar ook het type `LIST` wordt gedeclareerd — buiten dat pakket zijn immers de namen van de velden van het type onbekend.

Tot slot definiëren we een algoritme om vast te stellen of een bepaald element in een lijst aanwezig is.

```
BOOL OP IN (EL el, LIST l):
  IF isempty (l)
  THEN FALSE
  ELIF el = HEAD l
  THEN TRUE
  ELSE el IN TAIL l
  FI
ENDOP IN;
```

Merk op dat de vergelijkingsoperator `=` voor het type `EL` gedefinieerd moet zijn.

9.3.3 Filter-operaties

De volgende operaties maken gebruik van procedures als parameters, en kunnen als lijst-filters worden opgevat. Dit voorbeeld dient tevens als illustratie voor de mogelijkheid, procedures als parameters door te geven. De notatie daarvoor in ELAN is wat zwaarmoedig.

Een algoritme om alle elementen die aan een bepaalde predicaat `p` voldoen uit een lijst bijeen te sprokkelen:

```
LIST PROC filter(BOOL PROC(EL)p, LIST
l):
  IF is empty (l)
  THEN empty list
  ELIF p (HEAD l)
  THEN HEAD l +
       filter (BOOL PROC (EL) p, TAIL l)
  ELSE filter (BOOL PROC (EL) p, TAIL l)
  FI
ENDPROC;
```

Een procedure die een als parameter meegegeven procedure toepast op alle elementen van een lijst luidt als volgt:

```
PROC apply to all (PROC(EL)p, LIST l):
  IF NOT is empty (l)
  THEN p (HEAD l);
       apply to all (PROC(EL)p, TAIL l)
  FI
ENDPROC apply to all;
```

Een toepassing van de vorige procedure is een `put` voor lijsten, gebaseerd op de `put` van het type `EL`:

```
PROC put (LIST l):
  apply to all (PROC (EL) put, l)
ENDPROC put;
```

Tot slot van deze paragraaf een programmaatje dat een lijst maakt van 10 willekeurige gehele getallen en daar de oneven uit wegfiltert. Het maakt gebruik van de standaardprocedure `even`.

```
program:
  LIST VAR z :: empty list;
  UPTO 10
  REP z := random (1, 99) + z
  ENDREP;
  line; put (z); line;
  put (filter (BOOL PROC (INT) even,
z)).
```

9.3.4 Andere representaties voor lijsten

Overigens is de hier gekozen representatie van een lineaire lijst wel een hele natuurlijke, maar zeker niet de enige mogelijke. Andere zijn:

- representatie als een gedeeltelijk gevulde rij van elementen, zoals in het inleidende voorbeeld — dan kent de lineaire lijst echter een harde bovengrens op zijn lengte ("begrensde lineaire lijst"). Bij deze representatie blijven alle (verborgen) verwijzingen achterwege, waardoor het geheugenbeslag van de lijst kleiner kan zijn (men spreekt van een "impliciete" representatie of **impliciete datastructuur** omdat er geen expliciete verwijzingen optreden — uit het beginpunt van de rij en het indexnummer van een element is zijn plaats in het geheugen meteen af te leiden). Bovendien is niet alleen het kop-element in constante tijd toegankelijk (de rij is willekeurig toegankelijk, heeft **random access**). Meer hierover in hoofdstuk 11.
- representatie als een aantal paren `<EL, REF>`, via expliciete verwijzingen `REF` met elkaar verbonden (bijvoorbeeld via indices in een rij van paren). Sommige programmeertalen zoals ALGOL68 en C kennen hiervoor een in de taal ingebouwd verwijzings-mechanisme.
  Als we zelf die verwijzingen gaan construeren en zelf ons geheugen gaan beheren zijn we op een gevaarlijke weg, vanwege de mogelijkheid met verwijzingen te manipuleren (een alias te vormen, e.d.) bijzonder machtig, maar grote voorzichtigheid vereisend In de onderwijstaal ELAN is daarom geen expliciet verwijzingsmechanisme aanwezig — we prefereren het gebruik van recursieve datatypen. Zie hierover verder hoofdstuk 11.

9.4 Veel voorkomende lijststructuren

We behandelen een aantal klassieke lijststructuren met hun bijbehorende operaties (zie ook [KNU69, DEF77, AHU83]).

9.4.1 De stapel

De **stapel** (**stack**, "last-in-first-out list", "LIFO-list") ontleent zijn naam aan het beeld van de stapel borden in een cafetaria of de stapel kratten in een warenhuis: alleen bovenop kun je er een toevoegen of weghalen, je laat het wel uit je hoofd om er in het midden een toe te voegen of weg te halen. Als gevolg daarvan kun je steeds alleen het laatst toegevoegde exemplaar van een stapel afhalen. De stapel wordt met name gebruikt waar een omkering in de tijd nodig is, zoals in het laatste hoofdstuk van Deel I van dit boek bij een transformatie van recursie naar iteratie.

Fig. 9.4.1 toont de signatuur van een stapel (`STACK`) met `EL`-elementen. De voornaamste stapel-operaties zijn `push` en `pop`. Onmiddellijk na `push (s,x)` geldt `pop(s) = x`, d.w.z. het laatste gepushte element wordt het eerst gepopt (vergeef dit bizar jargon). De functie `top` dient ter inspectie van het laatst gepushte element, zonder pop-effect. Een **begrensde stapel** heeft bovendien een bovengrens aan het aantal elementen alsmede een Boole'se functie `is full` die vertelt of de stapel vol is (d.w.z. het maximum aantal elementen bevat).

Het type `STACK` is uitstekend op `LIST` op te bouwen, waarbij alle gemeenschappelijke operaties via concretizers en abstractors op de overeenkomstige operaties van de realisering worden afgebeeld.

```
TYPE STACK = LIST;

STACK CONST empty stack :: STACK: empty
list;

BOOL PROC is empty (STACK s):
  is empty (CONCR s)
ENDPROC is empty;
```

Deze constructie, waarbij de romp van een algoritme voor een abstract type slechts bestaat uit de oproep van een verwante algoritme voor de realisering van dat type, is typisch voor de Bottom-Up programmering. Denk bijvoorbeeld ook aan acces-algoritmen, die veelal als romp niet meer zullen hebben dan een selectie of subscriptie. Een goede implementering houdt met dit fenomeen rekening door zulke eenvoudige oproepen "inline" (d.w.z. zonder de overhead van het procedure-mechanisme) te vertalen.

```
PROC push (STACK VAR s, EL el):
  CONCR s:= el + CONCR s
ENDPROC;
```

Merk op dat de hier gebruikte operaties die van `LIST` zijn.

```
EL PROC top (STACK s):
  HEAD CONCR s
ENDPROC;

EL PROC pop (STACK VAR s):
  assert (NOT is empty (s));
  EL CONST result :: top (s);
  CONCR s:= TAIL CONCR s;
  result
ENDPROC;
```

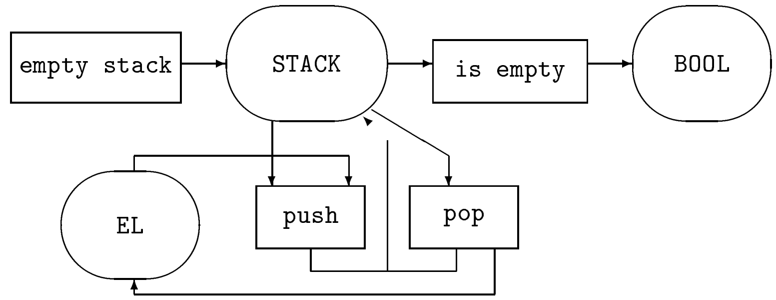


Figure 9.3: Signatuur van STACK

Uiteraard is een begrensde stapel ook heel eenvoudig als een impliciete datastructuur, n.l. een rij, te representeren (zie bijvoorbeeld het al genoemde 14e hoofdstuk van Deel I), wat wellicht ook de meest gebruikelijke representatie is.

De hier beschreven stapel is een *zuivere stapel*: Alleen op het top-element is onmiddelijk acces mogelijk, de andere elementen kunnen slechts via `pop`'s worden bereikt. Als bovendien ook de andere elementen bereikbaar zijn (bijvoorbeeld via hun index) spreek je van een **onzuivere stapel**.

Soms heeft een stapel geen `pop`-operatie, tijdens zijn leven wordt er alleen maar toegevoegd, niet afgehaald. Een voorbeeld is de naamlijst van een compiler of interpreter, een onzuivere stapel die enkel toeneemt totdat hij in zijn geheel wordt weggesmeten. Men spreekt dan van een **quasi stack**.

9.4.2 Queues

Een **queue** (wat een woord!) of **wachtrij** ("first-in-first-out list", "FIFO-list", "FIFO-queue") is een lijst waarbij elementen uitsluitend aan de voorkant kunnen worden toegevoegd en aan de achterkant kunnen worden afgehaald (of omgekeerd, afhankelijk van je perspectief). In plaats van het type `QUEUE` op het type `LIST` op te bouwen zullen we het deze keer ab initio implementeren, bijwijze van oefening. We declareren:

```
TYPE QUEUE = STRUCT (EL head, QUEUE
tail);

QUEUE CONST empty queue :: QUEUE: NIL;

BOOL PROC is empty (QUEUE q):
  ISNIL CONCR q
ENDPROC is empty;
```

De signatuur van het type `QUEUE` ziet er vrijwel hetzelfde uit als die van `STACK`, maar de operaties hebben een andere semantiek.

```
PROC insert (QUEUE VAR q, EL el):
  q := QUEUE: [el, q]
ENDPROC insert;
```

```
PROC remove (QUEUE VAR q, EL VAR el):
  assert (NOT is empty (q));
  IF is empty (q.tail)
  THEN
    el := q.head;
    q := empty queue
  ELSE remove (q.tail, el)
  FI
ENDPROC remove;
```

Het is belangrijk de werking van deze laatste procedure te begrijpen. De procedure is recursief en de eerste parameter is een variabele. Neem aan dat hij wordt opgeroepen met een `QUEUE`-variabele `s`, die een queue van zeg 3 elementen bevat. De eerste incarnatie roept een volgende met als zijn eerste parameter `q.tail`, waar `q` een alias is voor `s`, dus `s.tail` welke uitdrukking binnen de tweede incarnatie op zijn beurt `q` heet; deze roept een derde incarnatie met `q.tail`, wat dus een alias is voor `s.tail.tail` oftewel de staart van het tweede element van `s`, wat op zijn beurt binnen die derde incarnatie weer `q` heet. Daar wordt vastgesteld dat de staart van `q` leeg is, dus wordt toegewezen `q:= empty queue`. De staart van het tweede element van `s` is nu overschreven met `empty queue`, dus `s` heeft zijn derde element verloren.

De sleutel tot het begrip ligt daarin dat formele variabelen als alias worden doorgegeven — zoals beschreven in Hoofdstuk 12 van Deel I.

We zien dat de verschillende incarnaties van de recursieve procedure `remove` ieder op een volgend element van `s` vat hebben, en voldoende diep in de recursie kunnen we dus ook het achterste element van de lijst bereiken en (omdat het als variabele is meegegeven) door een toewijzing veranderen.

9.4.3 Deques

Een **deque** ("double-ended queue"; "deck"; "tweezijdige wachtlijst") is een lineaire lijst waarbij zowel aan de kop als aan de staart elementen kunnen worden toegevoegd en weggehaald, met de 4 operaties

```
PROC insert front (DECK VAR d, EL el):
  {el wordt vooraan d gevoegd}

PROC insert back (DECK VAR d, EL el):
  {el wordt achteraan d gevoegd }
```

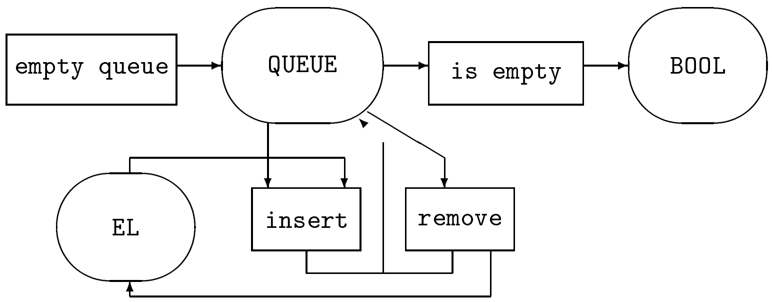


Figure 9.4: Signatuur van `QUEUE`

```
PROC remove front (DECK VAR d, EL VAR
el):
  {het voorste element van d wordt}
  {verwijderd en aan el toegewezen}

PROC remove back (DECK VAR d, EL VAR
el):
  {het achterste element van d wordt}
  {verwijderd en aan el toegewezen}
```

Hiermee kan bijvoorbeeld een wachtrij met voordringen voor VIP's (of voor dringende zaken) worden gerealiseerd, terwijl natuurlijk zowel de queue als de stapel als een specialisering van de deque (weglaten van enige operaties) kunnen worden gezien.

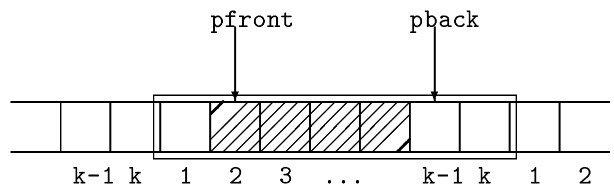
In plaats van een nieuw type `DECK` op het type `LIST` op te bouwen zullen we een module implementeren dat één begrensde deque realiseert.

```
PACKET deque module DEFINES
  is empty, is full,
  insert front, insert back,
  remove front, remove back:
```

Aangezien er toch maar één deque bestaat zullen we die niet expliciet als parameter meegeven. De deque is dus een module in de zin van 8.1.1.

We representeren deze ene deque als een impliciete datastructuur, in de vorm van een rij, verborgen in een pakket, waarvan het aantal elementen een grens k stelt aan het aantal elementen van de deque. Dit laatste aantal varieert, tussen nul en `k`, waardoor dus telkens maar een deel van de elementen van de rij in gebruik is. In de variabele `pfront` bewaren we de index van het voorste element van de deque. In de variable `pback` houden we de index net voorbij het achterste element. In plaats daarvan hadden we ook de index van het achterste element kunnen nemen, maar dit leidt tot een iets ingewikkelder implementering.

Aangezien elementen aan beide zijden worden toegevoegd respectievelijk verwijderd dreigen we buiten het indexbereik van de rij terecht te komen. We kunnen dit probleem oplossen door iedere keer dat vooraan een element wordt toegevoegd of verwijderd de rij een plaats naar achteren of naar voren te schuiven, maar dat is een tijdrovende zaak. Beter is het, de rij op te vatten als een **circulaire lijst**:



We denken ons daartoe de rij aan beide zijden cyclisch voortgezet, en laten zowel `pfront` als `pback` cyclisch door het indexbereik `1..k` van de rij schuiven. Zowel in volle als in lege toestand van de deque geldt `pfront = pback`. We introduceren daarom in de initialisatie van het module een Boole'se variabele om die twee toestanden van elkaar te onderscheiden.

```
  LET k = 25;
  ROW k EL VAR buffer;
  INT VAR pfront :: 1, pback :: 1;
  BOOL VAR in empty state :: true.

BOOL PROC is full:
  pfront = pback AND NOT in empty state
ENDPROC is full;

BOOL PROC is empty:
  in empty state
ENDPROC is empty;
```

We introduceren twee (niet-geëxporteerde) operatoren voor het cyclisch verhogen en verlagen van indices.

```
OP UPMOD (INT VAR p):
  p INCR 1;
  IF p > k
  THEN p := 1
  FI
ENDOP UPMOD;

OP DOWNMOD (INT VAR p):
  p DECR 1;
  IF p = 0
  THEN p := k
  FI
ENDOP DOWNMOD;
```

Nu volgen de 4 hoofdoperaties.

```
PROC insert front (EL x):
  assert (NOT is full);
  pfront DOWNMOD k;
  buffer [pfront] := x;
  in empty state := false
ENDPROC insert front;

PROC insert back (EL x):
  assert (NOT is full);
  buffer [pback] := x;
  pback UPMOD k;
  in empty state := false
ENDPROC insert back;

PROC remove front (EL VAR x):
  assert (NOT is empty);
  x := buffer [pfront];
  pfront UPMOD k;
  in empty state := pfront = pback
ENDPROC remove front;

PROC remove back (EL VAR x):
  assert (NOT is empty);
  pback DOWNMOD k;
  x := buffer [pback];
  in empty state := pfront = pback
ENDPROC remove back;
```

Het afdrukken van de deque is net iets lastiger dan je verwacht, vanwege het feit dat de lege en de volle toestand moeten worden onderscheiden.

```
PROC print deque:
  line;
  IF is empty
  THEN put ("Empty!")
  ELSE
   INT VAR j :: pfront;
   REP
     put (buffer [j]);
     j UPMOD k
   UNTIL j = pback
   ENDREP
  FI
ENDPROC print deque;

ENDPACKET deque module;
```

9.4.4 Geordende lijsten

We noemen een lijst v = $< v_1, v_2, \ldots, v_n >$ geordend volgens een relatie R als

$$\forall i, j \in 1..n\, [v_i R v_j]$$

dat wil zeggen als de volgorde van de elementen van v in overeenstemming is met de relatie R. Zo zouden we een lijst kunnen hebben van getallen, op grootte geordend volgens de relatie <=, of een aantal paren $< \mathit{tijdstip}_i, \mathit{meetwaarde}_i >$ geordend naar toenemend tijdstip, etc.

Bij het invoegen van elementen in een geordende lijst willen we graag dat de ordeningsrelatie bewaard blijft — iets waarmee de implementering van de operaties rekening moet houden. Het type der **geordende lijsten** (**zoeklijst**, "ordered list", "search list") heeft dezelfde operaties en grotendeels dezelfde realisering als de lineaire lijst.

```
TYPE OLIST = STRUCT (EL head, OLIST
tail);

OLIST CONST empty olist :: OLIST : NIL;

BOOL PROC is empty olist (OLIST s):
  ISNIL CONCR s
ENDPROC;

EL OP HEAD (OLIST s):
  s.head
ENDOP HEAD;

OLIST OP TAIL (OLIST s):
  s.tail
ENDOP TAIL;
```

We voeren twee bouw-operaties in, waarvan de tweede een eenvoudige links-concatenatie is die de ordeningseigenschap van de lijst niet garandeert.

```
OLIST PROC olist (EL el):
  olist (el, empty olist)
ENDPROC olist;

OLIST PROC olist (EL el, OLIST l):
  OLIST: [el, l]
ENDPROC olist;
```

De cruciale operatie is de toevoeging, die wel de ordening moet behouden.

```
OLIST OP + (EL el, OLIST s):
  IF is empty olist (s)
  THEN olist (el)
  ELIF el <= HEAD s
  THEN olist (el, s)
  ELSE olist (HEAD s, el + TAIL s)
  FI
ENDOP +;
```

Hierop baseren we de concatenatie van twee OLIST's met behoud van ordening:

```
OLIST OP + (OLIST s1, s2):
  IF is empty olist (s2)
  THEN s1
  ELSE (HEAD s2 + s1) + TAIL s2
  FI
ENDOP +;
```

Merk op dat in de laatste ELSE-regel twee verschillende optellingen zijn gebruikt — de eerste is de toevoeging van een element aan een OLIST en de tweede de concatenatie van twee OLIST's (recursief gebruikt). De recursie termineert omdat de tweede parameter telkens kleiner wordt.

Het zoeken naar een bepaald element in een geordende lijst is efficiënter dan het zoeken in een lineaire ongeordende lijst: vanwege de ordening kunnen we ophouden met zoeken zodra we een element ontmoeten dat niet meer `<=` het gezochte element is.

```
BOOL OP IN (EL el, OLIST l):
  IF isempty (l)
  THEN FALSE
  ELIF el = HEAD l
  THEN TRUE
  ELIF el < HEAD l
  THEN el IN TAIL l
  ELSE FALSE
  FI
ENDOP IN;
```

Maar voor dit doel is de impliciete representatie van een begrensde geordende lijst (als een rij) nog heel wat efficiënter: daarop kunnen we zoeken met de halveringsmethode, die veel sneller is dan een lineaire zoekmethode.

9.5 Conclusie

Lineaire lijststructuren worden gekenmerkt door het feit dat ieder element daarvan (hoogstens) één opvolger heeft.

We hebben een aantal klassieke lineaire lijststructuren leren kennen, en drie verschillende manieren om ze te implementeren:

1. als een recursief datatype
2. als een impliciet datatype (met een begrensd aantal elementen)
3. gebaseerd op een voorgedefinieerd type (met name op de lineaire lijsten)

In een later hoofdstuk behandelen we nog de realisering van verzamelingen als lineaire lijsten. Maar eerst wordt het tijd voor de generalisatie naar nietlineaire lijststructuren.

9.6 Opgaven

1. (Merge Sort) Ontwerp een algoritme voor Merge Sort (zie bijv. hoofdstuk 13 van Deel I) dat gebruik maakt van lineaire lijsten.
2. (stapel) Implementeer een module dat een begrensde stapel realiseert, met als operaties

```
     PROC push (TEXT t) :
     PROC pop (TEXT VAR t) :
BOOL PROC is empty :
BOOL PROC is full :
```

3. (deque) Implementeer het type `DECK` als een recursief datatype, met als operaties

```
DECK OP + (EL el, DECK d):
   {el voor d gevoegd}
DECK OP + (DECK d, EL el):
   {el achter d gevoegd}
EL OP FIRST (DECK d):
   {het voorste element van d}
EL OP LAST (DECK d):
   {het achterste element van d}
DECK OP BUTFIRST (DECK d):
   {d behalve eerste element}
DECK OP BUTLAST (DECK d):
   {d behalve laatste element}
```

4. (multilengte) Ontwerp en implementeer een aritmetiek voor een type `LINT`, integers van willekeurige lengte gerealiseerd als lineaire lijsten.

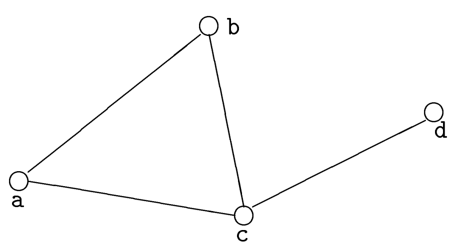
Chapter 10

Bomen

In dit hoofdstuk gaan we van lineaire lijststructuren over op *bomen*, lijststructuren waarbij meer dan een opvolger per knoop is toegelaten. We voeren enige begrippen uit de graphentheorie in en laten zien hoe bomen als recursieve datatypen gerealiseerd kunnen worden. We behandelen verschillende bijzondere vormen van bomen en geven voorbeelden van hun toepassing.

10.1 Enige begrippen

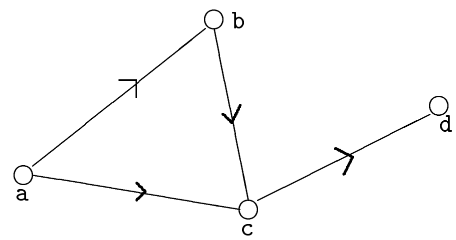
Bomen zijn een speciaal soort graphen. Een **graph** (“graaf”) is een verzameling K, waarvan de elementen **knopenknoop** of **punten** worden genoemd, samen met een verzameling (L) van paren knopen, waarvan de elementen **lijnen** heten. Een lijn is een verzameling van twee elementen van K. Een grafische voorstelling van bijvoorbeeld de graph $\langle K, L\rangle$ met $K = \{a, b, c, d\}$ en $L = \{\{a, b\}, \{b, c\}, \{a, c\}, \{c, d\}\}$ is:



Een knoop z is **bereikbaar** vanuit een knoop x, als $z = x$ of als er knoop y is, bereikbaar vanuit x, zo dat $\{y, z\}$ een element is van L. Er is dan een **pad** van x naar z. Een graph heet **samenhangend** als er (minstens) een pad is tussen elk tweetal knopen.

Onder een **boom** verstaat men in de Wiskunde een **cykelvrije** samenhangende graph (“cycle-free connected graph”), dat wil zeggen: een graph, waarin tussen elk tweetal knopen hoogstens één pad is.

Bij een **gerichte graph** graph bestaat L uit geordende paren (2-tupels) elementen van K. De lijnen hebben een richting en heten **pijlen**. Een pad volgens de pijlen heet een **weg**. De gerichte graph met $L = \{(a, b), (b, c), (a, c), (c, d)\}$ kunnen we tekenen als



In het vervolg gebruiken we de term **boom** steeds voor **gewortelde georiënteerde** gerichte bomen (“rooted oriented directed trees”), d.w.z. bomen waarvan precies één knoop (de **wortel**) geen ingaande pijl heeft (en alle andere knopen precies één) en waarbij de volgorde van de uitgaande pijlen (de **oriëntatie**) van belang is. Bij het tekenen van zo’n boom pleegt men de wortel bovenaan te tekenen, en alle pijlen neerwaarts.

De pijlen die van een knoop uitgaan wijzen naar zijn **opvolgers**. Een knoop kan nul of meer opvolgers hebben. Een knoop zonder opvolgers heet een **blad**, de andere knopen worden **interne knopen** genoemd.

We geven een kort lijstje van termen, met hun engelse en duitse equivalenten:

lijst	list	die Liste
lineaire lijst	linear list	lineare Liste
knoop, punt	node, vertex	der Knoten
pijl, lijn	edge	die Kante
boom	tree	der Baum
wortel	root	die Wurzel
blad	leaf	das Blatt

In aansluiting op deze begrippen kunnen we bomen ook recursief definiëren, als t-aire structuren over de opvolger-relatie.

Een **t-aire boom** is een boom waarin iedere knoop ten hoogste t t-aire bomen als opvolgers bezit, met een vaste volgorde. Men kan de opvolgers $k_i, i \in [1 : t]$ indiceren met i of selecteren met s_i. Vergelijk

TYPE TTREE = ROW t TTREE
TYPE TTREE = STRUCT (TTREE s_1, s_2, ..., s_t)

Bij een grafische representatie laat men de selectoren meestal impliciet.

Men kan ook de volgende, recursieve, definitie nemen:

> Een t-aire boom is óf leeg, óf hij bestaat uit een knoop en t disjuncte t-aire bomen B_1 t/m B_t, selecteerbaar met s_1 t/m s_t.

De opvolgers van een knoop worden ook zijn **onderbomen** ("subtrees") genoemd.

De knopen van een boom kunnen ieder met een waarde **gemarkeerd** zijn ("de waarde geassocieerd met die knoop").

Bomen zijn heel algemene structuren, die telkens weer optreden omdat daarmee veel ingewikkelde zaken gemodelleerd kunnen worden. Denk aan een ontleedboom die uitdrukt hoe een zin volgens een CF grammatica in elkaar zit — de knopen zijn daar gemarkeerd met nonterminale symbolen en de bladen met terminale symbolen. Denk ook aan de diagrammen waarmee men de structuur van organisaties pleegt uit te drukken.

Uit bovenstaande definitie verkrijgt men de lineaire lijst als bijzonder geval, voor $t = 1$. Een lijst is een wat saai uitgevallen boom.

Voor $t = 2$ krijgt men de **binaire boom** die we als eenvoudigste niet-triviale boom in de volgende sectie nader zullen bekijken.

Ook voor bomen met hogere t-waarden zijn toepassingen te vinden. Denk bijvoorbeeld aan een 26-aire boom voor het alfabetisch opslaan van woorden. De 4-aire bomen (**quad trees**) spelen een rol bij de opslag van 2-dimensionale plaatjes.

De gegeven recursieve definitie van t-aire boom is eenvoudig te modelleren in ELAN, via

```
TYPE TREE = STRUCT (INFO markering, ROW
t TREE onderbomen);
```

of

```
TYPE TREE = STRUCT (INFO markering, TREE
s1, s2, ..., st);
```

In beide gevallen dient de nil als denotatie voor de lege boom.

10.2 Binaire bomen

In het geval van de **binaire boom** onderscheidt men de linker- en de rechter-onderboom van een knoop:

```
TYPE TREE = STRUCT (EL info, TREE left,
right);
```

We voeren een denotatie en test in voor de lege boom:

```
TREE PROC empty tree:
  TREE: NIL
ENDPROC empty tree;
```

```
BOOL PROC is empty (TREE t):
  ISNIL CONCR t
ENDPROC is empty;
```

Een denotatie-procedure voor **singleton bomen** (d.w.z bomen die uit slechts één element bestaan) en een algemene bouw-operator voor bomen:

```
TREE PROC tree (EL e):
  tree (e, empty tree, empty tree)
ENDPROC tree;

TREE PROC tree (EL e, TREE l, r):
  TREE: [ e, l, r]
ENDPROC tree;
```

Acces-algoritmen:

```
TREE OP RIGHT (TREE t):
  assert (NOT is empty (t));
  t.right
ENDOP RIGHT;

TREE OP LEFT (TREE t):
  assert (NOT is empty (t));
  t.left
ENDOP LEFT;

EL OP INFO (TREE t):
  assert (NOT is empty (t));
  t.info
ENDOP INFO;
```

Procedures om de maximale diepte en het aantal knopen van een boom te bepalen:

```
INT PROC depth (TREE t):
  IF is empty (t)
  THEN 0
  ELSE 1+max(depth(LEFTt),depth(RIGHTt))
  FI
ENDPROC depth;

INT PROC size (TREE t):
  IF is empty (t)
  THEN 0
  ELSE 1 + size(LEFT t) + size(RIGHT t)
  FI
ENDPROC size;
```

Een simpele algoritme om de markeringen van de knopen van een boom in een kanonieke volgorde af te drukken:

```
PROC put (TREE t):
  IF NOT is empty (t)
  THEN
    put (LEFT t);
    line;
    put (INFO t);
    put (RIGHT t)
  FI
ENDPROC put;
```

Een wat slimmere algoritme geeft de boom min of meer grafisch weer, waarbij door indentatie de nest-

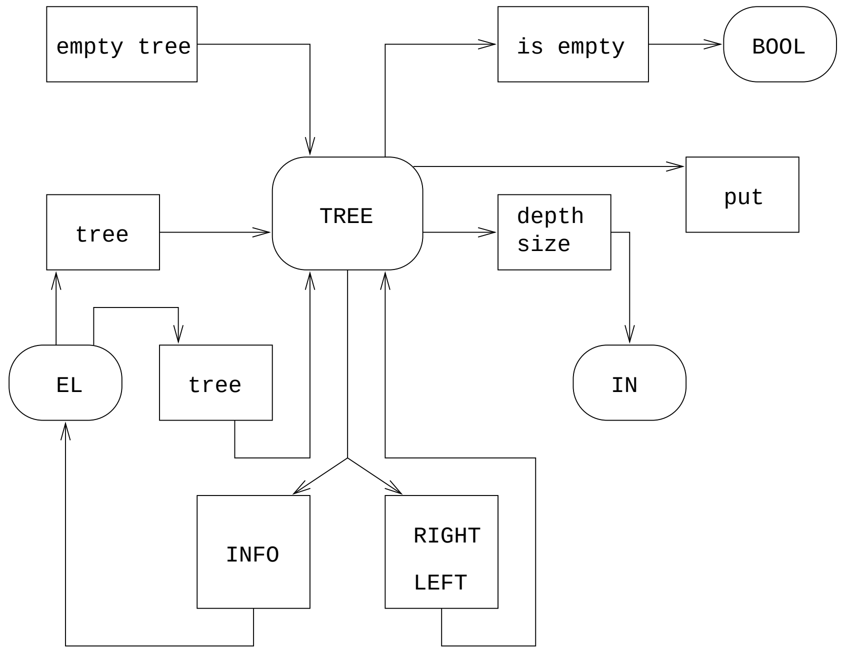


Figure 10.1: Signatuur van binaire boom

ingstructuur wordt weergegeven:

```
PROC put (INT indent, TREE t):
  IF NOT is empty (t)
  THEN put (indent + 1, LEFT t);
       line; put (indent * " ");
       put (INFO t);
       put (indent + 1, RIGHT t)
  FI
ENDPROC put;
```

Deze laatste procedure wordt opgeroepen als `put (0, boom)`.

10.2.1 Doorlópen van bomen

We introduceren enige belangrijke terminologie met betrekking tot het doorlopen van bomen (**boom traversie**).

Als we een boom recursief van links naar rechts en van boven naar beneden (**depth first**) doorlopen passeren we een t-aire knoop in beginsel $t + 1$ keer. Welke van die keren doen we iets met de `info` van de knoop? Als we bij een binaire boom de informatie van een knoop één maal verwerken zijn er drie consistente mogelijkheden:

- **preorder**:

  ```
  verwerk de informatie van de knoop;
  verwerk de linker onderboom in preorder;
  verwerk de rechter onderboom in preorder
  ```

- **inorder**:

  ```
  verwerk de linker onderboom in inorder;
  verwerk de informatie van de knoop;
  verwerk de rechteronderboom in inorder
  ```

- **postorder**:

  ```
  verwerk de linker onderboom in
  postorder;
  verwerk de rechter onderboom in
  postorder;
  verwerk de informatie van de knoop
  ```

Uiteraard zijn er nog vele andere systematische traversie-methoden te bedenken, bijvoorbeeld **breadth first**: laag voor laag van boven naar beneden en van links naar rechts.

Ga voor de procedure `put` hierboven na welke volgorde van doorlopen is toegepast.

10.3 Binaire zoekbomen

Bij het vergelijken van twee waarden a en b zijn er drie mogelijke uitkomsten:

a kleiner dan b
a gelijk aan b
a groter dan b.

In Deel I hebben we al gezien, dat we door op deze gevalsonderscheiding gebaseerde halveringsmethoden vaak efficiënte algoritmen kunnen verkrijgen. Als voorbeeld zagen we het zoeken in een woordenboek. Het algemene principe is: we zoeken in een geordende reeks $f_1, f_2, ..., f_n$ (de plaats van) het element met de waarde x. We kunnen als volgt te werk gaan:

1. een willekeurig element nemen: is dat het gezochte, dan zijn we klaar,
2. recursief verder zoeken in de deelrij links ervan (als het groter is dan het gezochte), of
3. in die rechts ervan (in het andere geval).

Bij iedere stap halveren we ongeveer het zoekgebied.

Bij de keuze van een representatie van de reeks kunnen we vooruitlopen op deze zoekmethode, door de reeks te representeren als dat willekeurige element plus de linker deelreeks en de rechter deelreeks. Recursief toegepast leidt dit idee tot een binaire boom met dat element als wortel:

```
TYPE TREE = STRUCT (EL info, TREE left,
right);
```

In dit verband kan de representatie van een geordende reeks door een geordende lineaire lijst worden gezien als een bijzonder geval: het "willekeurige" element is daar steeds het eerste, een van de takken van de boom is steeds leeg (en kan achterwege blijven).

Om met een tweedelingsalgoritme te kunnen zoeken gebruiken we een **geordende** binaire boom (**zoekboom**, "search tree"): we zorgen dat de linkeronderboom van een knoop precies alle elementen bevat die kleiner zijn dan het element in deze knoop (volgens de ordening van de elementen) en dat de rechteronderboom precies alle elementen bevat die groter zijn dan het element in de knoop. Het zoeken, bijvoorbeeld om vast te stellen of een bepaald element al of niet in de boom zit, gaat dan eenvoudig:

```
BOOL OP IN (EL el, TREE t):
  IF is empty (t) THEN false
  ELIF INFO t = el THEN true
  ELIF INFO t > el
  THEN el IN LEFT t
  ELSE el IN RIGHT t
  FI
ENDOP IN;
```

We kunnen een boom uiteraard opbouwen met de eerder ingevoerde bouwoperatie `tree`, maar daarmee is de bedoelde ordeningseigenschap niet gegarandeerd. Daarom voeren we een nieuwe procedure `enter` in die een element aan een boomvariabele toevoegt, en daarbij zorg draagt dat de genoemde ordeningsrelatie (recursief) gehandhaafd blijft. Alleen verschillende elementen worden opgenomen, dus het toevoegen van een element dat al in de boom zit heeft geen effect.

```
PROC enter (TREE VAR t, EL e):
  IF is empty (t)
  THEN t := tree (e)
  ELIF e < INFO t
  THEN enter (t.left, e)
  ELIF INFO t < e
  THEN enter (t.right, e)
  ELSE { element zit al in boom }
  FI
ENDPROC enter;
```

Op deze manier kan, uitgaand van de lege boom, iedere geordende binaire boom worden opgebouwd.

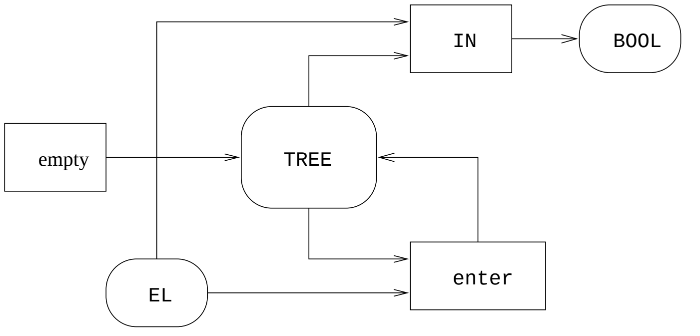


Figure 10.2: Signatuur voor zoekboom

10.3.1 Vorm en inhoud

Als we een zoekboom in inorder-volgorde doorlopen krijgen we de markeringen van de elementen te zien in een volgorde die zich houdt aan de gekozen ordening.

Een zoekboom is daarmee een alternatief voor een geordende lijst. De boom geeft de vorm, de elementen in inorder-volgorde de inhoud.

Wanneer zijn twee zoekbomen gelijk? Als ze dezelfde elementen in dezelfde volgorde bevatten, waarbij de vorm van de boom er niet toe doet. Het blijkt niet makkelijk een elegante en efficiënte gelijkheidstest voor zoekbomen te definiëren, omdat daarbij het mogelijke verschil in vorm tussen de te vergelijken bomen geen betekenis heeft.

De meest voor de hand liggende manier om vast te stellen of `a = b` is na te gaan of of alle elementen van `a` ook in `b` voorkomen en vice versa. Dit leidt tot een nogal smerig stukje programma, met een aantal hulpoperaties en variabelen, eerder een teken van vlijt dan van begrip.

Wat erger is, de complexiteit van deze test is van de orde $0(n \log n)$, waarbij n staat voor het aantal elementen in de te vergelijken bomen (de complexiteit is evenredig met het aantal elementen in de ene maal de gemiddelde lengte van de zoekweg in de andere).

Kan de gelijkheid van twee zoekbomen ook in $0(n)$ bepaald worden? Dat kan, door elk van de zoekbomen eerst om te zetten in een zoeklijst (een geordende lijst) met dezelfde elementen en dan de gelijkheidstest voor zoeklijsten te gebruiken.

```
BOOL OP = (TREE a,b):
  list (a) = list (b)
ENDOP =;
```

```
LIST PROC list (TREE x):
  IF is empty (x) THEN empty list
  ELSE list (LEFT x) + (INFO x + list
(RIGHTx))
  FI
ENDPROC list;
```

De omzetting van zoekboom naar zoeklijst en de gelijkheidstest voor zoeklijsten zijn beide lineair in het aantal elementen, zodat we nu een lineaire algoritme hebben verkregen.

In het domein van de zoeklijsten hebben twee lijsten met dezelfde inhoud ook dezelfde vorm, in het domein van de zoekbomen kan eenzelfde inhoud in geheel verschillende vormen worden gegoten.

Sommige operaties (zoals de gelijkheidstest) zijn goedkoper in het domein van de zoekbomen. In dit geval echter leidt zelfs de overhead van de afbeelding van de zoekbomen op zoeklijsten nog tot een goedkopere gelijkheidstest!

In 10.5.3 zullen we de mogelijke variatie in vorm van zoekbomen benutten om, bij gelijkblijvende inhoud, een vorm te kiezen die aan andere criteria beantwoordt (namelijk: gemiddelde zoekweg minimaal). Eerst bekijken we een toepassing van zoekbomen.

10.4 Voorbeeld: Concordantie

Een **concordantie** op een tekst is een lijst van alle woorden die in die tekst voorkomen, tezamen met hun *frequentie* (het aantal malen dat het woord in de tekst voorkomt) en eventueel nog andere gegevens, zoals een lijst van vindplaatsen. Denk aan concordanties op de bijbel. Concordanties zijn een belangrijk hulpmiddel voor tekststudie.

In dit voorbeeld construeren we een programma dat, met behulp van binaire zoekbomen, een concordantie maakt op een programmatekst. Als woorden nemen we de lexicale elementen van ELAN: identifiers, verschillende vormen van denotaties, bold symbols en operatortekens.

```
TYPE TREE = STRUCT (EL info, INT count,
TREE left, right);
```

```
LET EL = TEXT;
```

Het heeft geen zin te proberen een rij van alle symbolen van ELAN te vormen, want dat zijn er willekeurig veel. Daarom is het niet mogelijk, eerst een volledig vocabulaire op de zetten en dan alle symbolen van het invoer-programma te turven.

Daarom bouwen we een zoekboom op van precies die symbolen die in het programma voorkomen. Elk symbool van het programma van het programma wordt in de boom-in-opbouw opgezocht. Komt het symbool reeds voor, dan wordt het (slechts) geturfd. Komt het niet voor, dan wordt het toegevoegd (met frequentie één). Om het zoeken te versnellen wordt de boom meteen geordend bijgehouden.

```
PROC enter (TREE VAR t, EL e):
  IF is empty (t)
  THEN t := TREE: [e, 1, TREE: NIL,
TREE: NIL]
  ELIF e < INFO t
  THEN enter (t.left, e)
  ELIF INFO t < e
  THEN enter (t.right, e)
  ELSE t.count INCR 1
  FI
ENDPROC enter;

PROC write (TREE t):
  IF is empty (t)
  THEN
  ELSE
    write (LEFT t);
    write (COUNT t);
    write ("  ");
    write (INFO t);
    write line;
    write (RIGHT t)
  FI
ENDPROC write;

INT OP COUNT (TREE t):
  assert( NOT is empty(t));
  t.count
ENDOP COUNT;
```

De overige operaties op bomen zijn dezelfde als in de vorige sectie.

10.4.1 Het hoofdprogramma

We construeren eerst het Top-Down gedeelte van het programma.

```
program:
  ask for file;
  initialize administration;
  count all symbols in file;
  write frequency table;
  give statistics.

ask for file:
  TEXT VAR file name;
  put ("Filename:");
  get (file name);
  old file (file name).

initialize administration:
  TEXT VAR buffer :: "";
  INT VAR line count :: 0, symb count ::
0;
  TREE VAR dict :: empty tree.

  count all symbols in file:
   REP
     TEXT CONST symbol :: next symbol;
     enter (dict, symbol)
   UNTIL symbol = "$"
   ENDREP;
   line (2).
```

```
write frequency table:
 new file ("frq" + file name);
 write (dict).

give statistics:
 line (2);
 put ("Number of lines  ");
 put (line count);
 line;
 put ("Number of symbols");
 put (symb count);
 line;
 put ("Number of tokens ");
 put (size (dict));
 line;
 put ("Depth of the tree");
 put (depth (dict)).
```

Nu moeten we nog de cruciale procedure `next symbol` definiëren, die de achtereenvolgende lexicale symbolen van de invoer leest.

10.4.2 Lexicale analyse

De **lexicale analyse** is dat deel van een vertaler of vertolker dat de invoer in lexicale eenheden (woorden, lexemen, symbolen) uiteenrafelt.

Wij zijn van plan, teksten geschreven in een programmeertaal uiteen te rafelen in symbolen van die taal. De manier waarop dat gebeurt is op zich al leerzaam, omdat dit een idee geeft van een benadering die we **syntax-gestuurd programmeren** zullen noemen: de volgende procedures zijn op een heel stereotype wijze uit een aantal syntaxregels afgeleid.

We beginnen met het definiëren van een aantal **alfabets**, de tekenverzamelingen waaruit symbolen worden opgebouwd:

```
LET letters =
"abcdefghijklmnopqrstuvwxyz";

LET letgits =
"0123456789abcdefghijklmnopqrstuvwxyz";

LET digits = "0123456789";

LET bolds =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Het hart van de zaak is de procedure die telkens een volgend symbool levert. Er zijn vijf soorten symbolen: identifiers, intreal denotations, text denotations, bold words en operatoren. Eventuele spaties tussen de symbolen moeten worden overgeslagen.

We herkennen telkens de soort van het symbool aan het eerste niet-blanke teken van de invoer.

```
TEXT PROC next symbol:
  skip spaces;
  symb count INCR 1;
  IF ahead (letters)
  THEN next identifier
  ELIF ahead (digits)
  THEN next intreal denotation
  ELIF ahead ("""")
  THEN next text denotation
  ELIF ahead (bolds)
  THEN next bold word
  ELSE next operator
  FI
ENDPROC next symbol;

PROC skip spaces:
  WHILE ahead (" ")
  REP skip char
  ENDREP
ENDPROC skip spaces;
```

We gebruiken twee herkenningsprocedures voor tekens uit een alfabet:

```
BOOL PROC is (TEXT alfabet):
  IF ahead (alfabet)
  THEN
   skip char;
   true
  ELSE false
  FI
ENDPROC is;

BOOL PROC ahead (TEXT alfabet):
  pos (alfabet, HEAD buffer) <> 0
ENDPROC ahead;
```

Tot slot definiëren we een procedure die de invoer regelsgewijs leest en opslaat in een buffer, terwijl teken voor teken van de invoer wordt opgeleverd.

```
TEXT PROC next char:
  TEXT CONST s :: HEAD buffer;
  buffer := TAIL buffer;
  WHILE buffer = ""
  REP
   IF file ended
   THEN buffer := "$ "
   ELSE
     read (buffer);
     buffer CAT " ";
     line count INCR 1
   FI;
   line;
   put (buffer)
  ENDREP;
  s
ENDPROC next char;

PROC skip char:
  TEXT CONST dummy::
     next char
ENDPROC skip char;
```

10.4.3 Herkennen van identifiers

Een identifier is te herkennen aan het feit dat hij met een letter begint. De volgende herkenningsprocedures zijn rechtstreeks af te leiden uit de syntaxregels:

```
identifier: letter, rest identifier.

rest identifier:
    letter, rest identifier;
    digit, rest identifier;
    " ", letter, rest identifier;
    " ", digit, rest identifier;
    .
```

```
TEXT PROC next identifier:
  {ahead=letter}
  TEXT VAR s :: next char;
  s + rest identifier
ENDPROC next identifier;
```

Vanwege de collaterale berekening van de operanden van de `+` en het zijeffect van `rest identifier` kunnen we niet eenvoudigweg schrijven `nextchar + rest identifier`, wat alleen goed gaat als de operanden van links naar rechts worden berekend.

```
TEXT PROC rest identifier;
  {ahead=unknown}
  IF ahead (letgits)
  THEN next identifier {gemene truuk!}
  ELIF is (" ")
  THEN
   IF ahead (letgits)
   THEN " " + next identifier
   ELSE ""
   FI
  ELSE ""
  FI
ENDPROC rest identifier;
```

10.4.4 Herkennen van intreals

Een intreal-denotatie begint met een cijfer. De (nogal gedetailleerd gekozen) syntaxregels luiden:

```
intreal denotation:
    digit, intreal denotation;
    digit, rest real denotation.

rest real denotation:
    ".", tail real denotation.

tail real denotation:
    digit, tail real denotation;
    digit, exponent.

exponent:
    "e", integer denotation.

integer denotation:
    "+", rest integer denotation;
    "-", rest integer denotation;
    rest integer denotation.

rest integer denotation:
    digit, rest integer denotation;
    digit.
```

```
TEXT PROC next intreal denotation:
  {ahead=digit}
  TEXT CONST s :: next char;
  IF ahead (digits)
  THEN s + next intreal denotation
  ELIF ahead (".")
  THEN s + rest real denotation
  ELSE s
  FI
ENDPROC next intreal denotation;

TEXT PROC rest real denotation:
  {ahead=period}
  TEXT CONST s :: next char;
  assert (ahead (digits), "Digit
expected");
  s + tail real denotation
ENDPROC rest real denotation;

TEXT PROC tail real denotation:
  {ahead=digit}
  TEXT CONST s :: next char;
  IF ahead (digits)
  THEN s + tail real denotation
  ELIF ahead ("e")
  THEN next exponent
  ELSE s
  FI
ENDPROC tail real denotation;

TEXT PROC next exponent:
  {ahead=e}
  TEXT CONST s :: next char;
  assert (ahead ("+-" + digits),
"Incorrect exponent");
  s + next integer denotation
ENDPROC next exponent;

TEXT PROC next integer denotation:
  {ahead=sign or digit}
  TEXT VAR s :: next char;
  assert (ahead (digits), "Incorrect
number");
  s + rest integer denotation
ENDPROC next integer denotation;

TEXT PROC rest integer denotation:
  {ahead=digit}
  TEXT VAR s :: next char;
  IF ahead (digits)
  THEN s + rest integer denotation
  ELSE s
  FI
ENDPROC rest integer denotation;
```

10.4.5 Herkennen van text denotations

Een text-denotatie begint met een quote. De (wat vereenvoudigde) syntax luidt:

text denotation:
quote, quote;
quote, rest text denotation.

rest text denotation:
nonquote character, quote;
nonquote character, rest text denotation.

```
TEXT PROC next text denotation:
  {ahead=quote}
  TEXT CONST s :: next char;
  IF is (quote)
  THEN s + quote
  ELSE s + rest text denotation
  FI
ENDPROC next text denotation;

TEXT PROC rest text denotation:
  {ahead=non-quote}
  TEXT CONST s :: next char;
  IF is (quote)
  THEN s + quote
  ELSE s + rest text denotation
  FI
ENDPROC rest text denotation;
```

10.4.6 Herkennen van bold words

Een bold word bestaat enkel uit hoofdletters.

bold word:
bold letter, bold word;
bold letter.

```
TEXT PROC next bold word:
  {ahead=bold}
  TEXT CONST s :: next char;
  IF ahead (bolds)
  THEN s + next bold word
  ELSE s
  FI
ENDPROC next bold word;
```

10.4.7 Herkennen van operatoren

Alle andere tekens vatten we op als operatoren, waarvan sommige op zich staan en andere zijn samengesteld, namelijk

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

```
TEXT PROC next operator:
  {ahead=some special character}
  IF is ("<")
  THEN after less
  ELIF ahead (">")
  THEN after greater
  ELIF is (":")
  THEN after colon
  ELSE next char
  FI.

    after less:
     IF is (">")
     THEN "<>"
     ELIF is ("=")
     THEN "<="
     ELSE "<"
     FI.

    after greater:
     IF is ("=")
     THEN ">="
     ELSE ">"
     FI.

    after colon:
     IF is (":")
     THEN "::"
     ELIF is ("=")
     THEN ":="
     ELSE ":"
     FI.
  ENDPROC next operator;
```

Hiermee is het concordantieprogramma compleet.

10.5 Gebalanceerde bomen

Om bij een zoekboom profijt te trekken van zijn ordening dienen zijn linker en rechter onderboom ongeveer evenveel elementen te bevatten. Als een binaire zoekboom n knopen heeft, dan duurt zoeken naar een bepaald element:

- in het gunstigste geval: $O(1)$, d.w.z. een bepaalde constante (kleine) tijdseenheid
- in het ongunstigste geval: $O(n)$ tijdseenheden
- gemiddeld: $O(\log n)$ tijdseenheden (zie bijvoorbeeld [KNU69]).

Dit gemiddelde is tevens het ongunstigste geval als de boom **gebalanceerd** is, d.w.z. als voor elke knoop geldt, dat de linker- en rechteronderboom zoveel mogelijk even groot zijn.

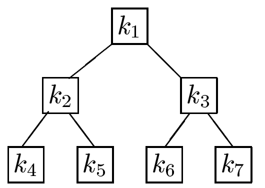
10.5.1 Volle bomen

Onder een **volle t-aire boom van diepte k** verstaat men

1. voor $k = 0$ de lege boom
2. voor $k > 0$ een knoop tezamen met precies t disjuncte volle t-aire bomen van diepte $k - 1$ als opvolgers.

Iedere t-aire boom is vervat in een volle. Volle t-aire bomen vréten uiteraard ruimte zodat men ze niet vaak gebruikt, maar ze spelen wel een rol bij de afleiding van sommige algoritmen.

Een volle binaire boom van diepte n heeft precies $2^n - 1$ elementen. Verder heeft hij de prettige eigenschap dat hij heel eenvoudig op een rij van $2^n - 1$ elementen af te beelden is. Beschouw daartoe de volgende volle binaire boom van diepte 3.



Als we de elementen in breadth-first volgorde opsommen krijgen we precies de volgorde $\boxed{k_1}\boxed{k_2}\boxed{k_3}\boxed{k_4}\boxed{k_5}\boxed{k_6}\boxed{k_7}$.

Als we de indices van de linker- en rechterzoon van een knoop k met index i bekijken vinden we de volgende regelmaat:

- $k.left$ heeft de index $2 * k$
- $k.right$ heeft de index $2 * k + 1$
- De wortel van de boom heeft index 1.

Door deze regelmaat kunnen we een selectie binnen de boom realiseren als een eenvoudige indexberekening. De basisoperaties van de binaire boom zijn hiermee te realiseren:

```
TYPE TREE = INT;

TREE PROC empty tree:
  TREE : 0
ENDPROC;

BOOL PROC is empty (TREE f):
  CONCR f = 0
ENDPROC;
```

Denotatie en bouw door declaratie:

```
LET nmb el = 127, k = 7;
{max el = 2 ** (k-1)}
ROW nmb el EL VAR tree;
```

Er bestaat slechts één exemplaar van de boom, dat wordt geconstrueerd door de rij-elementen rechtstreeks in te vullen; er zijn geen bouwoperaties om de boom uit te breiden.

```
TREE PROC root : TREE: 1 ENDPROC;
```

Access-operaties:

```
TREE OP LEFT (TREE t):
  INT CONST s :: 2 * CONCR t;
  assert (s>0);
  IF s > max el THEN empty tree ELSE s
FI
ENDOP LEFT;

TREE OP RIGHT (TREE t):
  INT CONST s :: 2 * CONCR t + 1;
  assert (s>0);
  IF s > max el THEN empty tree ELSE s
FI
ENDOP RIGHT;

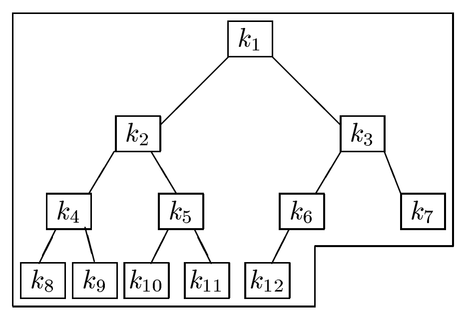
EL OP INFO (TREE t):
  tree [CONCR t]
ENDOP;
```

De overige boomoperaties (voor zover zinvol) blijven ongewijzigd, bijvoorbeeld het zoeken in de zoekboom:

```
BOOL OP IN (EL const el, TREE t):
  IF is empty (t) THEN false
  ELIF INFO t = el THEN TRUE
  ELIF INFO t > el
  THEN el IN LEFT t
  ELSE el IN RIGHT t
  FI
ENDOP;
```

Ondanks de volledig andere representatie hebben we een identieke (deel)semantiek verkregen.

Indien het aantal elementen niet een tweemacht min een bedraagt krijgen we door deze afbeelding een volle boom waarvan de onderste laag niet helemaal is opgevuld.



In de literatuur zijn verschillende algoritmen te vinden, zoals **Heapsort**, die gebaseerd zijn op het gebruik van volle bomen.

10.5.2 Gebalanceerd houden

In geval van een niet-volle boom kunnen we toch proberen de boom op zodanige wijze bij te houden dat hij niet (of nooit sterk) uit balans geraakt. Verschillende balanceringscriteria zijn denkbaar:

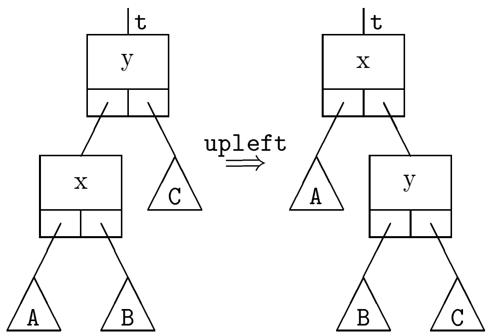
- **diepte balancering**. Bij het invoegen en verwijderen van elementen zien we erop toe dat voor iedere knoop het verschil in diepte tussen zijn linker- en rechter-onderboom ten hoogste één bedraagt (ze precies gelijk houden lukt alleen in een volle boom). De hierop gebaseerde **AVL-bomen** worden in het volgende semester behandeld.
- **frequentie balancering**. We kunnen voor elke knoop bijhouden hoe vaak hij (als onderdeel van het zoeken) wordt bezocht. Als nu de linkerdeelboom van de wortel veel vaker wordt bezocht dan de rechteronderboom verdient een herschikking (zie volgende sectie) aanbeveling. Dit moet worden afgewogen tegen de vraag hoeveel in zo'n geval invoegen, verwijderen en doorlopen kosten.

In de literatuur over datastructuren (bijvoorbeeld [DEF77] [AHU83]) vindt men verschillende typen balanceringen van bomen, die elk een bepaald compromis met betrekking tot de kosten van eerder genoemde operaties vertegenwoordigen.

10.5.3 Balanceren achteraf

Een niet gebalanceerde zoekboom (10.3) kunnen we pogen alsnog te balanceren, door knopen van de overladen tak naar de andere te verslepen, uiteraard bij behoud van de ordening.

We voeren daartoe een operatie `upleft` in die de wortel van de boom een element naar links opschuift en daarbij de ordening behoudt.




```
PROC upleft (TREE VAR t):
  assert (NOT is empty (t.left));
  t := TREE: [t.left.mark, t.left.left,
             TREE: [t.mark,
t.left.right, t.right]]
ENDPROC upleft;
```

De werking van de operatie berust op herstructureren van het topje van de boom. Analoog kan men ook een `upright` invoeren.

Daarmee kunnen we nu de boom te lijf gaan zonder dat een element verloren gaat dan wel erbij komt en zonder dat de ordening verloren gaat. Om de boom te balanceren strekken we hem eerst naar rechts tot een lineaire lijst, daarna vouwen we hem in het midden op en doen datzelfde recursief met zijn linker- en rechter-flap. In een plaatje:

```
PROC balance (TREE VAR t):
  stretch (t);
  fold (t)
ENDPROC balance;

PROC stretch (TREE VAR t):
  IF is empty (t)
  THEN
  ELSE
   IF is empty (t.left)
   THEN stretch (t.right)
   ELSE
     upleft (t);
     stretch (t)
   FI
  FI
ENDPROC stretch;
```

```
TREE PROC split (TREE VAR t, INT n):
  IF n = 0
  THEN empty
  ELSE
   EL CONST m :: t.mark;
   t := t.right;
   tree (m, empty, split (t, n - 1))
  FI
ENDPROC split;

PROC fold (TREE VAR t):
  IF NOT is empty (t)
  THEN
   t.left := split (t, (size (t) - 1)
DIV 2);
   fold (t.left);
   fold (t.right)
  FI
ENDPROC fold;
```

We proberen het balanceerproces uit door een zoekboom van willekeurig voortgebrachte integers te vergelijken met zijn gebalanceerde versie.

```
program:
  TREE VAR root :: empty;
  UPTO 12
  REP enter (root, random (0,99))
  ENDREP;
  put (root); line (2);
  balance (root); put (13, root).
```

Een willekeurig voortgebrachte zoekboom blijkt gemiddeld tweemaal zo diep als een gebalanceerde.

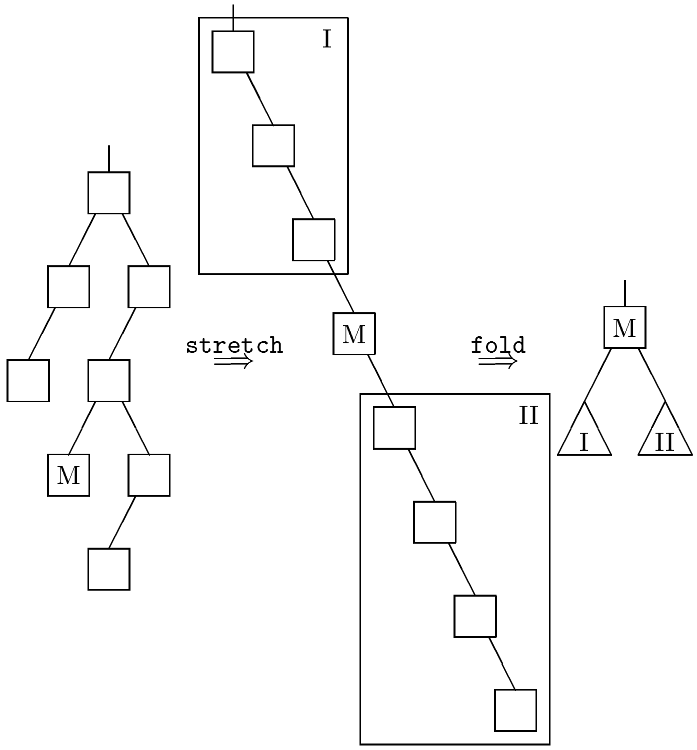
10.5.4 Heuristische balancering

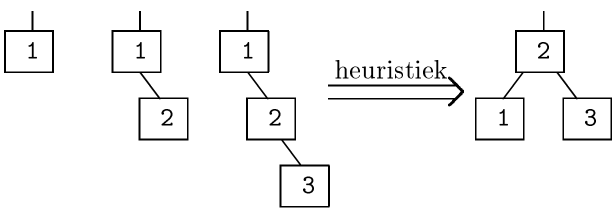
In plaats van te proberen een boom perfect te balanceren kun je ook pogen hem zo op te bouwen dat verwacht kan worden dat de onbalans niet al te groot wordt, door tijdens het invoegen van elementen in de boom lokale herstructureringen uit te voeren.

In plaats van optimale balancering krijg je dan een **heuristische balancering**, die daar een aardige slag naar slaat. Zonder dure kunstgrepen als het bijhouden van allerlei tellingen in iedere knoop is er onvoldoende informatie aanwezig om een optimale balancering te garanderen.

Bekijken we nog eens het toevoegen van elementen aan een zoekboom (10.3). Worden twee of meer elementen op volgorde van de ordening na elkaar toegevoegd, dan ontstaat een sprietige boomtak.

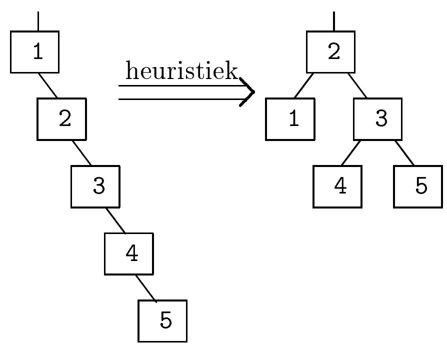
Om deze situatie te voorkomen zullen we telkens een `upleft` resp. zijn gespiegelde analogon `upright` loslaten op een deelboom die tot een spriet dreigt te verworden.






```
PROC enter (TREE VAR t, EL e):
  IF is empty (t)
  THEN t := tree (e)
  ELIF e < INFO t
  THEN IF is empty (t.left)
      THEN t.left := tree (e)
      ELSE enter (t.left, e);
           IF is empty (t.right)
           THEN upleft (t)
           FI { het kan nog wel wat
efficienter.. }
      FI
  ELIF INFO t < e
  THEN IF is empty (t.right)
      THEN t.right := tree (e)
      ELSE enter (t.right, e);
           IF is empty (t.left)
           THEN upright (t)
           FI
      FI
  ELSE { element zit al in boom }
  FI
ENDPROC enter;
```

Het effect hiervan is dat geen kale takken zonder zijscheuten ontstaan. Zonder de heuristiek wordt door alle elementen op volgorde aan te bieden een lineaire lijst gebouwd. Door de heuristiek ontstaat een **harige boom** ("hairy tree"), een binaire boom waarin geen knoop voorkomt die als enige onderknoop een knoop met maar één onderknoop heeft.



Uit metingen blijkt dat de diepte van een zoekboom met willekeurige getallen gevuld gemiddeld 30% groter is dan die van een harige zoekboom met dezelfde elementen (1.301 ± 0.092, als resultaat van 311 metingen aan zoekbomen van 3260 willekeurige elementen). Een harige zoekboom van willekeurige elementen is dus gemiddeld anderhalf keer zo diep als een gebalanceerde. Een zelfde effect is te verwachten op de gemiddelde lengte van zoekwegen.

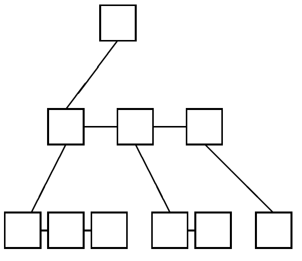
Het effect van de heuristiek is nogal variabel, afhankelijk van de volgorde van aanbieden. In verhouding tot zijn geringe prijs (in rekentijd en programmeertijd) is dit echter een hele aardige heuristiek.

Eenzelfde heuristiek zou kunnen worden toegepast tijdens het zoeken in de boom (met de operator `IN`). Wordt tijdens het zoeken de structuur van de boom zodanig gewijzigd dat telkens het gezochte element (ten koste van andere) een plaats dichter naar de wortel klimt, dan ontstaat een min of meer op zoekfrequentie gebalanceerde boom: vaak gezochte elementen zitten dichter bij de wortel.

10.6 Onregelmatige bomen

Een binaire boom, met zijn rigoreuze beperking tot hoogstens twee onderbomen per knoop, heeft een nogal regelmatige structuur. De volle binaire boom vertoont zelfs een volkomen regelmaat.

De t-aire bomen met een grote of zelfs onbegrensde waarde van t vormen het andere uiterste. Het aantal onderbomen kan per knoop sterk verschillen. Zo'n **onregelmatige boom** kan eenvoudig gepresenteerd worden als een lijst van lijsten.



Een op deze representatie gebaseerd voorbeeld wordt uitgewerkt in 12.4.

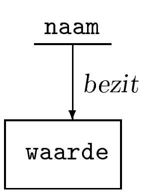
Chapter 11

Graphen

In dit hoofdstuk beschouwen we de implementatie van graphen en de operaties daarop. Daartoe introduceren we eerst het begrip *verwijzing*. We zullen laten zien hoe met behulp van verwijzingen algemene en flexibele representaties van vele mathematische objecten kunnen worden gegeven.

11.1 De bezit-relatie

In onze terminologie is een object de combinatie van een naam met een waarde. We zeggen dat die naam die waarde **bezit**. Deze **bezit relatie** ontstaat door een declaratie en geldt tot het verlaten van het geldigheidsgebied van de betrokken relatie. Daarbinnen kan deze relatie eventueel worden gewijzigd door een toewijzing.



Bij een geïnitialiseerde declaratie wordt er, net als bij de toewijzing, een kopie van de waarde van de initialisatie toegewezen. Nou ziet de ene kopie van de waarde drie er niet anders uit dan de andere, maar voor samengestelde waarden heeft deze subtiliteit consequenties.

```
ROW 10 INT VAR r1;
geef r1 een waarde;
ROW 10 INT VAR r2 :: r1;
```

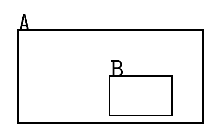
Als `r2` nu dezelfde waarde bezat als `r1` zou een toewijzing aan een element van `r2` het corresponderende element van `r1` mee wijzigen! Gelukkig bezit hij echter slechts een kopie daarvan.

11.2 De bevat-relatie

Een object is samengesteld als zijn waarde samengesteld is. Een waarde noemen we samengesteld als er minstens één andere waarde is die daarvan een **component** (d.w.z. element of veld) is. Een type, tot slot, noemen we samengesteld als de waarden ervan samengesteld zijn.

Het feit, dat een waarde van type `B` een component is van een waarde van type `A`, kunnen we uitdrukken door een `A` te realiseren als een structuur met een veld van type `B`:

```
LET A = STRUCT (..., B x, ...);
```



Iedere waarde van type `A` **bevat** een waarde van type `B`. Deze **bevat relatie** geldt ook voor een rij en zijn elemanten.

Een toewijzing aan het `x`-veld van een `A`-variabele

```
A VAR a; B VAR b; ... ; a.x:= b
```

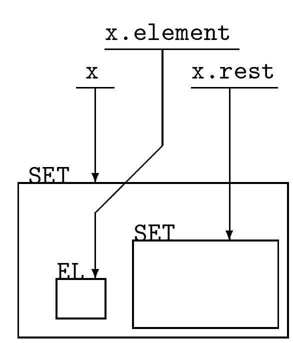
is dan tevens een toewijzing aan die variabele. Je kunt `a.x` zien als een alias voor een deelvariabele van `a`.

Interessant wordt het als waarden van type `B` op hun beurt weer waarden van type `A` kunnen bevatten of, meer direct, `A` waarden zelf weer `A` waarden kunnen bevatten (recursieve typen).

Als we bij voorbeeld een verzameling willen representeren als een element en een verzameling (van de overige elementen), komen we tot typen als

```
LET SET = STRUCT (EL element, SET rest);
```

met de paradoxale eigenschap dat een `SET` op zijn beurt een `SET` bevat.



Waarom onstaan dan geen oneindig grote objecten?

11.3 Verwijzingen

De oplossing ligt, net zoals bij recursieve algoritmen, in het gebruik van een naam voor een waarde in plaats van die waarde. Om aan te geven dat het object `a` het object `b` bevat, nemen we niet `b` zélf in `a` op, maar een **verwijzing** ("reference") naar `b`, een waarde (verschillend van `b` zelf), die ons toegang tot `b` kan geven, bijvoorbeeld een index van een rij-element of (in machinetalen) een adres.

We zeggen dat een waarde a **verwijst naar** een waarde b als a hetzij een verwijzing naar b is, hetzij een verwijzing naar b bevat. Nu kunnen we ook precies aangeven wat we bedoelen met een **lijststructuur**: een structuur (van waarden) over de verwijst-naar-relatie.

11.3.1 Concrete verwijzingen

In sommige programmeertalen als C, maar niet in ELAN, bestaat een ingebouwd mechanisme dat ons in staat stelt met de waarde van een variabele (van zeg type `EL`) een verwijzing naar die waarde te associëren (van bijv. type `REF (EL)`), met de mogelijkheid die verwijzing als waarde te manipuleren, en daardoor indirect ook die variabele.

In de terminologie die wij gebruiken, zeggen we dat die `REF (EL)`-waarde een **concrete verwijzing** naar dat object is, een abstractie van de begrippen **adres** of pointer. Deze in de taal ingebouwde (concrete) verwijzingen vormen een veilig en elegant mechanisme om met de adressen van objecten en hun waarden om te gaan.

Door de concrete verwijzing naar `b` op te nemen in `a` en niet een kopie van `b` zelf, kunnen we objecten recursief definiëren, zonder te komen tot oneindig grote objecten.

Iedere waarde van een abstract type wordt in ELAN door een verwijzing gerepresenteerd, die echter onzichtbaar blijft. Recursieve typen zijn daardoor mogelijk. Een constructor als `REF` is echter niet beschikbaar. Daarom kan `NIL` (een speciale verwijzing) ook alleen als representatie van een speciale waarde van een abstract type optreden.

Hoe zien die concrete verwijzingen er nu eigenlijk uit? Voor een deel moeten we die vraag onbeantwoord laten. We kunnen concrete verwijzingen slechts beschrijven door hun eigenschappen te noemen, net zoals we de gehele getallen eigenlijk alleen maar door hun eigenschappen (operaties!) kunnen beschrijven.

Wel kunnen we zeggen, dat in de meeste implementaties verwijzingen qua grootte ongeveer overeenkomen met integers. Immers, in principe kan elke verwijzing worden gerealiseerd door elke variabele te identificeren met een willekeurig, maar uniek nummer. Dit betekent, dat zelfs voor zeer grote aantallen variabelen de verwijzingen klein kunnen blijven. Overigens zijn de verwijzingen geen integers! Twee verwijzingen kunnen niet bij elkaar worden opgeteld of anderszins operaties ondergaan. In feite kunnen verwijzingen alleen maar met de speciale verwijzing `NIL` worden vergeleken.

11.3.2 Abstracte verwijzingen

We hoeven helemaal geen nieuwe taalconstructen te verzinnen om verwijzingen in te voeren. Zo kunnen we bijvoorbeeld de index van een element uit een rij gebruiken als verwijzing naar dat element en deze opnemen in andere objecten. Dit is een vorm van **abstracte** (d.w.z.: door ons zelf ingevoerde, niet tot de concrete taal behorende) verwijzing. Zulke representatietruuks en conventies zijn echter nogal onveilig (een bron van fouten), omdat we onze eigen conventies gemakkelijk kunnen vergeten: indices zien er net zo uit als andere getallen, we kunnen de verkeerde getallen voor indices aanzien, we kunnen op indices bewerkingen uitvoeren die op de bedoelde objecten zinloos zijn.

Veiligheid kunnen we krijgen door de verwijzing in een abstract datatype te verbergen. Als voorbeeld voeren we een type `LIST` in.

```
PACKET lijst type
DEFINES LIST, empty list, is empty,
        HEAD, TAIL, list, put, +, IN:

  TYPE LIST = INT; { namelijk een index
in de volgende rij }
  LET maxlist = ... ;
  ROW maxlist STRUCT (EL hd, LIST tl)
VAR elementen;
  INT VAR plist :: 1; { index van eerste
vrije element }
```

Hierbij bepaalt `maxlist` het maximaal aantal lijstelementen dat tegelijkertijd kan bestaan, terwijl we in `plist` expliciet bijhouden hoeveel daarvan er al in gebruik zijn.

De index nul gebruiken we als nil-element.

```
  LIST CONST empty list :: LIST:  0;

BOOL PROC is empty (LIST l):
  CONCR l = 0
ENDPROC is empty;
```

De bouw-algoritmen `list` en `+` nemen telkens een nieuw element van de rij in gebruik.

```
LIST PROC list (EL el):
  el + empty list
ENDPROC list;

LIST OP + (EL el, LIST l):
  assert (plist <= maxlist);
  elementen [plist]:= [el, l];
  plist INCR 1;
  LIST : (plist - 1)
ENDOP +;
```

Nu nog de acces-operatoren en we zijn in business.

```
EL OP HEAD (LIST l):
  assert (NOT is empty(l));
  elementen [CONCRl].hd
ENDOP HEAD;


LIST OP TAIL (LIST l):
  assert(NOT is empty(l));
  elementen [CONCR l].tl
ENDOP TAIL;
```

Alle overige lijst-operaties blijven bij deze representatie dezelfde als eerder (in 9.2) beschreven.

```
ENDPACKET lijst type;
```

Ieder toepassingsprogramma dat werkt met lijsten gerepresenteerd als recursief datatype zal ook kunnen werken met deze nieuwe representatie. Het is frappant te zien dat, dank zij het sterke abstractiebegrip, met eenvoudige middelen een veilige en relatief efficiënte lijst-abstractie kan worden gerealiseerd.

11.3.3 Geheugenbeheer

Bij even doordenken is de bovenstaande representatie niet zo tevredenstellend, omdat na `maxlist` keer gebruik van `list` deze denotatie-procedure niet meer mag worden opgeroepen — de lijstelementen zijn op. Bij de eerder ingevoerde representatie via een recursief type `LIST` kan natuurlijk ook slechts een eindig aantal lijstelementen worden gedenoteerd (afhankelijk van de geheugengrootte van de gebruikte computer) maar daar werd wel automatisch ieder element hergebruikt dat niet meer toegankelijk was (bijvoorbeeld door een overschrijvende toewijzing). De representatie met abstracte verwijzingen biedt deze luxe niet. We hebben de verplichting zelf een vorm van **geheugenbeheer** te voeren, met alle voor- en nadelen van dien.

We houden steeds bij welke lijstelementen nog vrij zijn (in de vrij-lijst `available`, verborgen in het pakket) en wijzigen de bouwalgoritme zodanig dat hij zijn elementen van de vrij-lijst haalt.

```
LIST OP + (EL el, LIST l):
  assert (NOT is empty (available));
  INT CONST p :: CONCR available;
{eerste vrije element}
  available:= TAIL available;
  elementen [p]:= [el,l];
  LIST : p
ENDOP +;
```

Om de vrij-lijst op te bouwen geven we het pakket de volgende initialisatie:

```
INT VAR p;
FOR p UPTO maxlist - 1
REP
  elementen [p].tl:= LIST: (p+1)
ENDREP;
elementen [maxlist].tl:= empty list;
LIST VAR available :: LIST: 1.
```

Nu kunnen we ook een procedure invoeren om een lijst die niet meer nodig is te recyclen door al zijn elementen weer in de vrij-lijst op te nemen.

```
PROC recycle (LIST v):
  IF NOT is empty (v)
  THEN
    recycle (TAIL v);
    elementen [CONCR v].tl:= available;
    available:= v
  FI
ENDPROC recycle;
```

Het hoeft geen betoog dat het recyclen van waarden die nog wél nodig blijken een pracht-manier is om ongelukken te maken. Het hier beschreven geheugenbeheer is dan ook niet zo robuust, er kan niet worden voorkomen dat de gebruiker ervan ongelukken maakt.

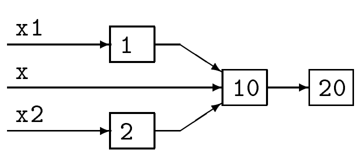
Het liefst zouden we met problemen zoals het geheugenbeheer niets te maken willen hebben (zoals bij recursieve lijsten). Wij behandelen dit mechanisme hier dan ook voornamelijk om te laten zien hoe spannend en gevaarlijk het leven is bij gebruik van abstracte verwijzingen.

11.3.4 Confluente lijsten

Door het expliciet gebruik van verwijzingen is het met de hier gekozen representatie ook mogelijk, **confluentie** (samenvloeiing) van lijsten uit te drukken. In het voorbeeld

```
LET EL = INT;
LIST VAR x :: 10 + list (20);
LIST VAR x1 :: 1 + x;
LIST CONST x2 :: 2 + x
```

hebben de lijsten `x1` en `x2` alle elementen behalve het eerste gemeen.



We definiëren een algoritme om het n^e element van een lijstvariabele te wijzigen.

```
PROC replace (LIST VAR l, INT n, EL el)
:
  assert (NOT is empty (l));
  IF n = 1
  THEN elementen [CONCR l].hd:= el
  ELSE replace (elementen [CONCR l].tl),
n-1, el)
  FI
ENDPROC replace;
```

Wordt nu het eerste element van x gewijzigd, bijvoorbeeld door de oproep

```
replace (x, 1, -111)
```

dan veranderen de tweede elementen van x1 en x2 mee! De staart van de variable x1 en zelfs die van de constante x2 zijn allebei een alias voor x.

Willen we bij deze representatie een toewijzing met de gebruikelijke semantiek hebben ("linkerkant krijgt als waarde een kopie van de waarde van de rechterkant") dan moeten we ook de kopie zelf maken.

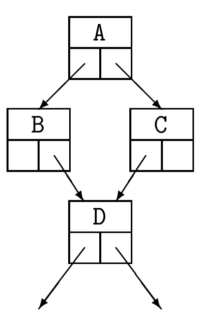
```
LIST OP COPY (LIST l):
  IF is empty (l)
  THEN l
  ELSE HEAD l + COPY TAIL l
  FI
ENDOP COPY;
```

Om nu alle aliassen uit te bannen schrijven we, gebruik makend van de zojuist ingevoerde operator:

```
LIST VAR x :: 10 + list (20);
LIST VAR x1 :: 1 + COPY x;
LIST x2 :: 2 + COPY x
```

Het geheugenbeheer van een ELAN-implementering werkt ongeveer zoals hierboven beschreven, maar dan automatisch en volkomen veilig, met wat meer raffinement waardoor vele kopieën vermeden worden. Het is gebaseerd op verborgen verwijzingen.

Wij kunnen van confluente lijsten ook een nuttig gebruik maken, namelijk om algemenere graphstructuren dan lineaire lijsten en bomen te representeren, bijvoorbeeld een **DOAG** (Directed Oriented Acyclic Graph, een gerichte georiënteerde acyclische graph), zoals de volgende ruitstructuur:



De knopen B en C in de graph hebben al hun opvolgers (D en diens opvolgers) gemeenschappelijk, waardoor deze minder geheugenruimte innemen dan ingeval B en C daarvan kopieën bezaten ("sharing"). Ook circulaire structuren zijn daarmee mogelijk, door tussenkomst van een verwijzing.

11.3.5 Impliciete en expliciete datastructuren

Bij een lineaire lijst kan de opvolger-relatie expliciet worden gerealiseerd door aan ieder element een verwijzing naar het volgende element te koppelen. Maar voor begrensde lijsten kan diezelfde relatie natuurlijk ook worden berekend door gebruik te maken van de natuurlijke volgorde van de elementen van een rij. Ditzelfde geldt voor bomen — zo kan een volle binaire boom elegant op een rij worden afgebeeld (10.5.1).

In geval van realisering van een datastructuur door middel van (abstracte of concrete) verwijzingen spreekt men van een **expliciete datastructuur**. Bij realisering als een rij spreekt men van een **impliciete datastructuur**, omdat daar de expliciete verwijzingen achterwege blijven.

11.4 Voorbeeld: Dijkstra's algoritme

Als voorbeeld realiseren we gerichte graphen in de vorm van een impliciete datastructuur met daarbij de algoritme van Dijkstra (zie [DIJ59]) voor het bepalen van de kortste weg tussen twee punten in een graph. De algoritme zelf wordt hier niet uitgelegd, het gaat ons enkel om de gebruikte datastructuur. Zowel een knoop als een pijl wordt voorgesteld door een geheel getal, namelijk een index in resp. de rij van knopen en de rij van pijlen. De pijlen die van éénzelfde knoop uitgaan worden als een lineaire lijst met abstracte verwijzingen gerepresenteerd.

Iedere pijl heeft zijn prijs, en de bedoeling is een weg te vinden met minimale totale prijs.

```
LET NODE = INT;

LET ARC = INT;

start with empty graph:
  LET max nodes = 20, max arcs = 100;
  ROW max nodes STRUCT
   (TEXT name, ARC neighbours) VAR
nodes;
  ROW max arcs STRUCT
   (NODE target, INT cost, ARC next)
               VAR arcs;
  INT VAR nmb nodes :: 0, nmb arcs :: 0;
  LET nil = 0.
```

Het voordeel van deze representatie is dat we heel gemakkelijk een begrensde herhaling over alle knopen kunnen doen. Daar zullen we op vele plaatsen gebruik van maken.

Eerst verzoeten we het leven met een aantal accesoperaties, om verderop niet allerlei abstruse selecties uit de rij van knopen te hoeven schrijven.

```
TEXT OP NAME (NODE n):
  nodes [n].name
ENDOP NAME;


ARC OP NBRS (NODE n):
  nodes [n].neighbours
ENDOP NBRS;
```

Hetzelfde voor de pijlen:

```
NODE OP TARGET (ARC a):
  arcs [a].target
ENDOP TARGET;


INT OP COST (ARC a):
  arcs [a].cost
ENDOP COST;


ARC OP NEXT (ARC a):
  arcs [a].next
ENDOP NEXT;
```

De knopen zijn gemarkeerd met namen. Om de knoop met een bepaalde naam op te zoeken gebruiken we de volgende operator.

```
NODE OP INDEX (TEXT node name):
  NODE VAR i;
  FOR i  UPTO nmb nodes
  REP
    IF NAME i = node name
    THEN LEAVE INDEX WITH i
    FI
  ENDREP;
  nil
ENDOP INDEX;
```

De graph wordt gelezen uit een bestand `graph.data`, waarin de graph genoteerd staat als een aantal regels van de vorm

```
startknoop --> eindknoop : prijs
```

De volgorde waarin de pijlen van de graph worden ingevoerd is geheel vrij.

```
PROC read graph (TEXT fname):
  old file (fname);
  WHILE NOT file ended
  REP get arc
  ENDREP;
  close file.

get arc:
  TEXT VAR record;
  read (record);
  INT CONST
     arrow pos :: pos(record, "-->"),
     colon pos :: pos(record, ":");
  IF arrow pos = 0 OR colon pos = 0
  THEN
    LEAVE get arc
  FI;
  add arc (from, to, cost).


from:
  node (compress (text (record, arrow
pos - 1))).


to:
  node (compress (subtext (record,
         arrow pos + 3, colon pos -
1))).


cost:
  int (subtext (record, colon pos + 1)).
ENDPROC read graph;
```

Voor het opbouwen van de graph dienen de volgende twee operaties.

```
PROC add arc (NODE from, to, INT cost):
  assert (nmb arcs < max arcs);
  assert (cost >= 0, "Only nonnegative
costs, please.");
  nmb arcs INCR 1;
  arcs [nmb arcs].target:= to;
  arcs [nmb arcs].cost:= cost;
  arcs [nmb arcs].next:= NBRS from;
  nodes [from].neighbours:= nmb arcs
ENDPROC add arc;


NODE PROC node (TEXT node name):
  NODE VAR k :: INDEX node name;
  IF k = nil
  THEN
    assert (nmb nodes < max nodes);
    nmb nodes INCR 1;
    nodes [nmb nodes].name:= node name;
    nodes [nmb nodes].neighbours:= nil;
    nmb nodes
  ELSE k
  FI
ENDPROC node;
```

Het weergeven van de graph op het beeldscherm gebeurt in dezelfde stijl als de representatie in het bestand.

```
PROC show graph:
  NODE VAR from;
  FOR from UPTO nmb nodes
  REP
    ARC VAR next :: NBRS from;
    WHILE next <> nil
    REP
      put (NAME from + " --> "
         + NAME TARGET next
         + " :" + text (COST next));
      line;
      next:= NEXT next
    ENDREP
  ENDREP
ENDPROC show graph;
```

De algoritme van Dijkstra luidt bij de gekozen representatie als volgt:

```
INT PROC ewd (NODE start, goal):
  build initial costs and visit vector;
  mark start node as visited;
  UPTO nmb nodes - 1
  REP
    NODE CONST k ::
      unvisited node with smallest cost;
    mark k as visited;
    adjust total costs via k
  ENDREP;
  total cost [goal].

build initial costs and visit vector:
  ROW max nodes BOOL VAR visited;
  ROW max nodes INT VAR total cost;
  NODE VAR i;
  FOR i UPTO nmb nodes
  REP
    visited [i]:= FALSE;
    total cost [i]:= direct path cost
(start, i)
  ENDREP.

mark start node as visited:
  visited [start]:= TRUE;
  total cost [start]:= 0.

unvisited node with smallest cost:
  NODE VAR j, min node;
  INT VAR min cost :: maxint;
  FOR j  UPTO nmb nodes
  REP
    IF NOT visited [j]
    THEN
      IF total cost [j] < min cost
      THEN
        min node:= j;
        min cost:= total cost [j]
      FI
    FI
  ENDREP;
  min node.

mark k as visited:
  visited [k]:= TRUE.

adjust total costs via k:
  FOR j  UPTO nmb nodes
  REP
    IF NOT visited [j]
    THEN
      INT CONST nw :: cost from start to
j via k;
      IF nw < total cost [j]
      THEN total cost [j]:= nw
      FI
    FI
  ENDREP.


cost from start to j via k:
  IF total cost [k] = maxint
  THEN maxint
  ELSE
    INT CONST dpc kj :: direct path cost
(k, j);
    IF dpc kj = maxint
    THEN maxint
    ELSE total cost [k] + dpc kj
    FI
  FI.
ENDPROC ewd;


INT PROC direct path cost (NODE from,
to):
  ARC VAR n :: NBRS from;
  WHILE n <> nil
  REP
    IF TARGET n = to
    THEN
      LEAVE direct path cost WITH (COST
n)
    ELSE n:= NEXT n
    FI
  ENDREP;
  maxint
ENDPROC direct path cost;
```

De volgende procedure dient voor het interactief invoeren van een knoop-naam.

```
NODE PROC ask node (TEXT prompt text):
  REP
    put (prompt text);
    TEXT VAR name;
    get (name);
    NODE CONST ident :: INDEX compress
(name)
  UNTIL ident <> nil
  ENDREP;
  ident
ENDPROC ask node;
```

Tot slot volgt een simpel hoofdprogramma, dat al deze stukken aan het werk zet.

```
program:
  start with empty graph;
  read graph ("graph.data");
  show graph;
  NODE CONST
    from node :: ask node ("From?"),
      to node :: ask node ("To?");
  line;
  put (ewd (from node, to node));
  put (ewd (to node, from node)).
```

Hiermee is de basis gelegd voor het bestuderen van graph-algoritmen, een onderwerp dat hier niet verder kan worden uitgediept.

In de volgende sectie wordt een andere representatie van graphen (als een matrix) behandeld, met een toepassing die wat lijkt op het kortste-weg probleem.

11.5 Voorbeeld: Warshall's algoritme

Het kortste-afstand-probleem is verwant aan een ander probleem op graphen, het vinden van de *transitieve afsluiting* van een gegeven relatie. Een relatie R tussen een aantal dingen denken we ons daartoe voorgesteld als een gerichte graph, met die dingen als knopen, en met een pijl van knoop p naar knoop q precies dan als de bedoelde relatie geldt tussen ding p en ding q. Is de relatie symmetrisch, dan is er voor iedere pijl van p naar q ook een pijl van q naar p.

Onder de **transitieve afsluiting** van R verstaan we een relatie R^+, met als graph dezelfde verzameling knopen en een pijl tussen ieder tweetal knopen waartussen een weg is in R.

Beschouw het volgende voorbeeld: een compiler moet bepalen of een procedure `P` een procedure `Q` (eventueel zichzelf recursief) kan aanroepen. De te beschouwen dingen zijn dan de procedures in een programma. Tussen twee dingen `P` en `Q` kan een (directe) relatie bestaan. In ons voorbeeld: `P` roept `Q` direct, d.w.z., de body van `P` bevat een aanroep van `Q`. De afsluiting van deze relatie is de relatie die aangeeft of `P` direct of indirect, desnoods via vele andere procedures, `Q` kan aanroepen.

We nummeren de knopen willekeurig van 1 tot en met n. Een directe relatie van p_s met p_t geven we aan met $p_s \rightarrow p_t$. We schrijven $p_s \rightarrow^+ p_t$ als er een stel knopen $p_{i_1}, \ldots, p_{i_k}$ is te vinden ($k \geq 0$), zo dat $p_s \rightarrow p_{i_1} \rightarrow \ldots \rightarrow p_{i_k} \rightarrow p_t$.

We willen nu voor elk tweetal s en t bepalen, of $p_s \rightarrow^+ p_t$ ($\rightarrow^+$ is de transitieve afsluiting van $\rightarrow$).

We representeren de graphen van $\rightarrow$ en $\rightarrow^+$ ieder door een matrix van waarheidswaarden (**adjacency matrix**):

```
TYPE REL = ROW n ROW n BOOL;
```

We willen een procedure schrijven

```
REL PROC warshall (REL direct):
  de transitieve afsluiting.
```

waarvoor geldt, dat `direct [s][t]` dan en slechts dan als $p_s \rightarrow p_t$ en `warshall (direct) [s][t]` dan en slechts dan als $p_s \rightarrow^+ p_t$.

Een voor de hand liggende aanpak is, alle mogelijke wegen van s naar t af te zoeken, in essentie zoals dat werd gedaan in Deel I [KOS84] bij de bepaling van de kortste weg tussen steden (een equivalent probleem). Dit leidt tot een exponentieel algoritme met complexiteit $O(d^n)$ waarin d een gemiddelde is voor het aantal knopen waarmee enige knoop een directe relatie heeft.

Warshall heeft echter een polynomiaal algoritme aangegeven dat op de volgende gedachtengang berust. De relatie $\rightarrow^+$ wordt in n iteraties opgebouwd, waarbij telkens meer informatie wordt toegevoegd. We definiëren het predicaat $R(s,t,m)$ = "er is een stel $p_{i_1}, ..., p_{i_j}, ..., p_{i_k}$ met $k \geq 0$ en $i_j \leq m$ (!) zodat $p_s \rightarrow \ldots \rightarrow p_{i_j} \rightarrow \ldots \rightarrow p_t$".

Ga na, dat geldt

- $p_a \rightarrow p_b = R(a,b,0)$
- $p_a \rightarrow^+ p_b = R(a,b,n)$

De algoritme berust op

$$R(s,t,m) = R(s,t,m-1) \vee (R(s,m,m-1) \wedge R(m,t,m-1))$$

d.w.z., als er (al) een weg is van s naar t via knopen met nummer hoogstens $m-1$ of als er een weg is van s naar m via knopen hoogstens $m-1$ en een weg van naar t via knopen hoogstens $m-1$, dan is er een weg van s naar t via knopen hoogstens m; die weg loopt dus van s via 0 of meer knopen $< m$ naar een knoop $= m$ en vervolgens via 0 of meer knopen $< m$ naar t. We zeggen hierbij voor het gemak $p_x < p_y$ als $x < y$, en $p_x = p_y$ als $x = y$.

In iteratie m van de algoritme bouwen we een `REL`-matrix `nieuw` op, waarvoor geldt `nieuw [s][t]` dan en alleen dan als $R(s,t,m)$. Met deze voorbereidingen kunnen we de algoritme direct opschrijven:

```
de transitieve afsluiting:
  REL VAR oud :: direct;
  INT VAR m;
  FOR m UPTO n
  REP {oud [s][t] = R (s, t, m-1) voor
elk tweetal s][t}
    REL VAR nieuw;
    bouw nieuw op;
    oud:= nieuw
    { oud [s][t] = R (s, t, m) voor elk
tweetal s][t}
  ENDREP;
  oud.
```

```
bouw nieuw op:
  INT VAR s, t;
  FOR s UPTO n
  REP FOR t UPTO n
    REP nieuw [s][t]:=
          oud [s][t] OR (oud [s][m] AND
oud [m][t])
    ENDREP
  ENDREP.
```

De complexiteit van deze algoritme is onmiddellijk duidelijk: de driedubbel geneste herhaling maakt hem $O(n^3)$. De initialisatie `oud:= direct` en de rijtoewijzing `oud:= nieuw` zijn $O(n^2)$, dus te verwaarlozen.

Overigens is het opbouwen van een afzonderlijke nieuwe matrix overbodig. De “met m bijgewerkte relatie” kan direct in de “oude” worden genoteerd. Voor sommige paren s, t maken we dan gebruik van “te nieuwe” informatie, maar dat maakt niet uit: de definitie van $R(s, t, m)$ laat toe, dat de weg van s naar t meer dan eens via de knoop m loopt. Ga dat na.

Daarmee wordt dus

```
de transitieve afsluiting:
  INT VAR m, s, t;
  REL VAR rel :: direct;
  FOR m UPTO n
  REP FOR s UPTO n
    REP FOR t UPTO n
      REP rel [s][t]:=
          rel [s][t] OR (rel [s][m] AND
rel [m][t])
      ENDREP
    ENDREP
  ENDREP;
  rel
ENDPROC warshall;
```

In volgende secties zullen we zien, hoe deze algoritme nog efficiënter gemaakt kan worden.

11.5.1 Een efficiëntere versie

In de kern van de algoritme hadden we

```
rel [s][t]:= rel [s][t] OR (rel [s][m]
AND rel [m][t])
```

We merken op, dat de waarde van `rel [s][m]` niet afhangt van `t` zodat dus de berekening daarvan uit de binnenste lus kan worden verwijderd. Als `rel [s][m]` false is, kan de `AND` geen `TRUE` leveren en wordt de hele toewijzing overbodig. De binnenste lus kan dus voorwaardelijk worden gemaakt:

```
IF rel [s][m]
THEN FOR t UPTO n
  REP rel [s][t]:= rel [s][t] OR rel
[m][t]
  ENDREP
FI
```

Hierbij keren we gedeeltelijk terug van het gebruik van “te nieuwe” informatie, maar dat maakt voor het resultaat geen verschil, omdat die niet nodig is. De complexiteit is nog steeds $O(n^3)$, maar wel met een lagere evenredigheidsconstante, vooral als de directe relatie-matrix weinig elementen `TRUE` bevat.

11.5.2 Nogmaals Warshall’s algoritme

Een variant die er op het eerste gezicht heel anders uitziet gaat uit van een type `SET` dat de verzameling $\{1, 2, \ldots, maxsize\}$ realiseert, met ten minste de operaties uit fig. 11.1.

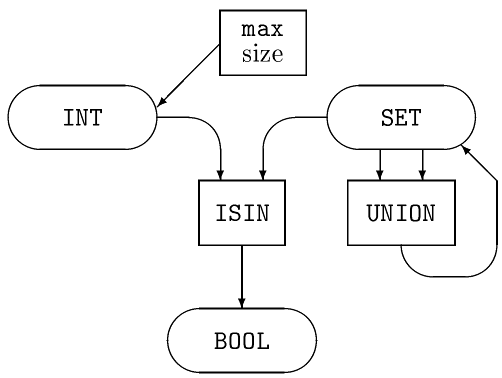


Figure 11.1: Signatuur van `SET`

Zolang `n` niet groter is dan `max size` kunnen we een relatie realiseren als een rij van verzamelingen

```
TYPE REL = ROW n SET;
```

waarbij voor de relatie `direct` geldt

$$i \to k \Longleftrightarrow \texttt{k ISIN direct [i]}$$

In vele programmeertalen is zo’n type ingebouwd, omdat dat type en zijn operaties zich heel makkelijk laten afbeelden op de bitrijen en binaire operaties van allerlei machines. Daarom is `max size` ook meestal 32 of 64.

Hiervan gebruik makend kunnen we Warshall als volgt formuleren:

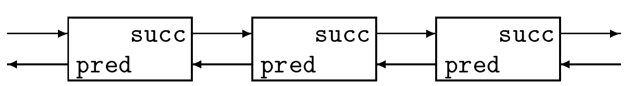
```
REL PROC warshall (REL direct):
  REL VAR rel :: direct;
  INT VAR m, s;
  FOR m UPTO n
  REP
    FOR s UPTO n
    REP
      IF m ISIN rel [s]
      THEN rel[s]:= rel[s] UNION rel[m]
      FI
    ENDREP
  ENDREP;
  rel
ENDPROC warshall;
```

Nu is de algoritme $O(n^2)$ geworden! Dat komt, omdat de bewerkingen op een set parallel, voor alle bits tegelijk, plaatsvinden (zolang tenminste n niet groter is dan `max size`!). Wordt echter `n` groter dan `max size` dan moeten we een relatie als $\lceil n \div maxsize \rceil * n$ sets realiseren, en komt $O(n^3)$ toch weer uit de mouw.

De hier beschreven Warshall-algoritme is in het hart van vele compilers en aanverwante systemen terug te vinden.

11.6 Opgaven

1. (geheugenbeheer) Hoe kunnen twee stapels op zuinige wijze worden gerealiseerd binnen eenzelfde rij? Kan hetzelfde met twee queue's? Hoe realiseer je meer dan twee stapels binnen eenzelfde rij?
2. (Merge Sort) Verzin een algoritme voor het in situ mengen van twee lijsten, in de aan het begin van dit hoofdstuk gegeven lijst-representatie met abstracte verwijzingen. Geef daarmee een O($nlogn$) in situ sorteeralgoritme.
3. (dubbelgeketende lijst) realiseer met behulp van abstracte verwijzingen een type `DLIST` ("doubly linked list"), een lineaire lijst waarvan ieder element behalve naar zijn opvolger ook naar zijn voorganger verwijst.

   De signatuur is dezelfde als die van `LIST`, met een extra operatie `PRED` die bij een gegeven lijstelement de lijst beginnend met het voorgangerelement daarvan oplevert. Implementeer ook een operatie om het k-de element uit de lijst te verwijderen.
4. (graphen) Breid het gegeven voorbeeld uit met een voorziening voor het wijzigen en weer uitschrijven van de graph. Schrijf een programma dat vaststelt welke cykels in de graph aanwezig zijn.
5. (kortste afstand) Schrijf een programma om volgens de methode van Warshall in O(n^3) een tabel te maken van de kortste afstanden tussen elk van n steden.

Chapter 12

Polymorfie

polymorfie is een bovenbegrip van twee begrippen:

- **genericiteit** — Bij een generische naam hoort een opsomming van definities, die voldoende van type verschillen om bij iedere applicatie van de naam te kunnen vaststellen welke definitie van toepassing is. Was er oorspronkelijk (in ALGOL68, ADA) slechts sprake van "overloading" van operatoren, in ELAN is genericiteit ook uitgebreid tot procedures.
- **Modaliteit** — bij een modale naam hoort één enkele definitie die op een (min of meer ruime) klasse van parametertypen van toepassing is.

In dit hoofdstuk gaan we in op het begrip polymorfie en zijn consequenties. Daartoe behandelen we eerst een uitbreiding van ELAN met taalmiddelen voor polymorfie: de mogelijkheid typen, procedures en operatoren met een type te parametriseren.

Polymorfie is in de eerste plaats een abstractiemiddel: het stelt ons in staat typen en operaties te generaliseren. Maar daardoor is het tevens een belangrijk hulpmiddel voor de bouw van algemeen bruikbare programmatuur.

12.1 Type-constructoren

We voeren weer een uitbreiding van ELAN in. Een type-declaratie kan worden voorzien van een of meer formele type parameters, zoals

```
TYPE LIST (M) = STRUCT (M hd, LIST (M)
tl)
```

Hierin is `M` een *formele type parameter*, waarvoor iedere actuele type-declarer kan worden gesubstitueerd, zoals bijvoorbeeld in `LIST (INT)` of `LIST (TEXT)` of zelfs `LIST (ROW 2 REAL)` of `LIST (LIST (BOOL))`.

Door deze geparametriseerde type-declaratie is `LIST` niet een apart type geworden, maar een **type constructor** oftewel **data structuur** een mechanisme voor het samenstellen van typen, waarden en objecten, analoog aan de bekende concrete datastructuren `ROW` en `STRUCT`. Pas door parametrisering met een type (aangeduid met een type-declarer) wordt zo'n constructor tot een (abstract) type. De syntax van ELAN wordt daartoe op enkele plaatsen uitgebreid:

type-declaration:
  type—token, type-association—list.

type-association:
  type-name, equal—token, type-declarer;
  type-constructor, formal-type-parameter—list—pack,
    equal—token, type-declarer.

type-constructor:
  bold-identifier.

formal-type-parameter:
  bold-identifier.

abstract-type-declarer:
  type-name;
  formal-type-name;
  type-constructor, type-declarer—list—pack.

We kunnen nu zelf algemene constructoren invoeren, uiteraard gebaseerd op de concrete constructoren, om onze uitdrukkingskracht te verhogen door een abstracter taalgebruik.

De datastructuren uit de vorige hoofdstukken zoals lijsten en binaire zoekbomen hoeven we slechts éénmaal in een pakket als constructoren met de bijbehorende operaties te definiëren om daarvan bij wijze van spreken een levenlang profijt te trekken ("re-usability", **herbruikbaarheid**). Polymorfie in combinatie met modulariteit stelt ons in staat, software te schrijven die in hoge mate herbruikbaar is: We hoeven niet iedere keer als we een lijst of boom nodig hebben bij Adam en Eva aan te vangen. Een eenmaal correcte implementering van een datastructuur blijft correct.

Merk op dat we niet de mogelijkheid invoeren, een type met een waarde te parametriseren, zoals `ROW` behalve met het type van elementen ook met hun aantal te parametriseren is. In dat opzicht is de `ROW`-constructor een anomalie, omdat dit de enige vorm van declarer is die een denoter als typebepalende component bevat (net als in PASCAL, maar in tegenstelling tot ALGOL60 en ALGOL68).

Niets weerhoudt U er dus van, te definiëren wat U bedoelt met

```
TYPE BOX (M) = ... ;
BOX (PASTILLES) VAR ...
```

maar als U het aantal pastilles in een doos wilt fixeren doet U er goed aan, dat aantal bij de denotatieprocedure voor `BOX`en mee te geven. Alle boxen van eenzelfde type pastilles hebben nu eenmaal hetzelfde type.

12.2 Modale algoritmen

Om ten volle profijt te kunnen trekken van een abstracte constructor als `LIST` moeten we in staat zijn, algoritmen te definiëren die op alle soorten van lijsten toepasbaar zijn. Daartoe kunnen ook operatoren en procedures impliciet met een formeel type worden uitgerust, dat daartoe (in een allesbehalve fraaie notatie) achter het symbool `OP` of `PROC` moet worden aangegeven.

procedure-head:
  result—option, proc—token,
      formal-type-parameter—list—pack—option,
procedure-name,
    formal-parameter-specification—pack—option,
colon—token.

Als voorbeeld een algemene denotatie-procedure voor lege lijsten:

```
PROC (EL) make empty (LIST (EL) VAR l):
  l := LIST (EL): NIL
ENDPROC;
```

Bij oproep van deze procedure wordt uit het type van de actuele parameter corresponderend met `l` het type afgeleid waaraan `EL` voor de duur van de oproep gebonden wordt. In de oproep

```
  LIST (INT) VAR mylist; ... ; make
empty (mylist) ; ...
```

is dit het type `INT`.

Let wel dat het te binden type alleen uit de actuele parameters van de operatie kan worden afgeleid, niet uit het door de context verlangde resultaat, zoals operaties ook alleen op de parameters generisch kunnen zijn. Het formele type wordt *impliciet* meegegeven, in zoverre als het bedoelde actuele type niet expliciet in de oproep wordt aanduid maar volgt uit de identificatie van de formele parameters. Een dergelijke algoritme met een formele type parameter noemen we een **modale algoritme**. (Deze vreemde term is afkomstig uit de ALGOL68 werkgroep, waarin reeds in 1972 gediscussieerd werd over "mode alternatives", kortweg modals.)

Het is helaas niet mogelijk een modale procedure

```
M PROC (M) empty:
  M: NIL
ENDPROC empty;
```

in te voeren omdat (door de afwezigheid van een formele parameter) het formele type `M` nergens aan gebonden kan worden.

Laten we, als oefening, de typische operaties voor de constructor `LIST` verder definiëren.

```
BOOL PROC (M) is empty (LIST (M) l):
  ISNIL CONCR l
ENDPROC is empty;

EL OP (EL) HEAD (LIST (EL) l):
  l.head
ENDOP;
```

We maken hier gebruik van de kennis (alleen beschikbaar binnen het pakket waarin `LIST` gedefinieerd wordt) dat een van de velden van `LIST` de naam `head` heeft. In plaats van de formele typenaam `EL` hadden we natuurlijk ook iedere andere vrije typenaam kunnen gebruiken, bijvoorbeeld `M`. Zijn binding aan een bepaald actueel type geldt telkens binnen de romp van een algoritme, voor de duur van de uitvoering daarvan.

```
LIST (M) OP (M) TAIL (LIST (M) l):
  l.tail
ENDOP TAIL;
```

Nu schrijven we nog een denotatie-procedure voor de singleton list en een linkse toevoeging van een element aan een lijst:

```
LIST (EL) PROC (EL) list (EL a)
  LIST (EL): [a, LIST (EL): NIL]
ENDPROC list;

LIST (EL) OP (EL) + (EL a, LIST (EL)
const b):
  LIST (EL): [a,b]
ENDOP +;
```

We kunnen een lijst van drie teksten opbouwen door:

```
bouw textlijst:
  LIST (TEXT) VAR l;
  l := "a" + ("b" + list("c")).
```

Merk op dat hier de haakjes na de eerste `+` absoluut noodzakelijk zijn omdat anders (vanwege de associatieregels van de operatoren) de betekenis zou zijn:

```
  l := ("a" + "b") + list ("c")
```

waardoor de lijst slechts twee elementen (`ab` en `c`) zou hebben!

Met hetzelfde gemak bouwen we een lijst van de eerste 100 gehele getallen:

```
bouw intlijst:
  LIST (INT) VAR l;
  make empty (l);
  INT VAR i;
  FOR i UPTO 100
  REP l := i+l
  ENDREP.
```

In welke volgorde staan de elementen van deze laatste lijst?

12.2.1 Noodzakelijke operaties

We geven een algemene afdrukprocedure voor lijsten.

```
PROC (M) put (LIST (M) l):
  IF is empty (l) THEN
  ELSE line;
       put(HEAD l);
       put(TAIL l)
  FI
ENDPROC put;
```

Merk op dat deze algemene uitvoerprocedure alleen werkt voor lijsten waarvan de elementen zelf op hun beurt een procedure `put` kennen. We kunnen bijvoorbeeld een lijst van Booleans niet hiermee afdrukken (tenzij we eerst een `put` voor Booleans definiëren). Een modale algoritme legt vaak eisen op aan het type van zijn parameters, omdat het verlangt dat voor dat type bepaalde operaties gedefinieerd moeten zijn, de **noodzakelijke operaties** van die algoritme. Een modale algoritme is alleen toepasbaar op een type waar alle noodzakelijke operaties in de signatuur voorkomen.

Als laatste geven we een algemene sorteeralgoritme voor lijsten.

```
PROC (M) sort (LIST (M) VAR l):
  LIST (M) VAR result;
  make empty (result);
  WHILE NOT is empty (l)
  REP insert (result, HEAD l);
      l:= TAIL l
  ENDREP;
  l := result
ENDPROC sort;

PROC (M) insert (LIST (M) VAR l, M x):
  IF HEAD l >= x
  THEN l:= x + l
  ELSE insert (l.tail, x)
  FI
ENDPROC insert;
```

Welke noodzakelijke operaties op het elementtype dienen daartoe gedefiniëerd te zijn?

12.2.2 Volgorde van identificatie

Als we van polymorfie gebruik maken zullen er van bepaalde operatoren vele definities bestaan, die zich onderscheiden in de mate aan **specificiteit** van hun parametrisaties.

De generische exemplaren zijn het meest specifiek, omdat ze ieder precies één specifieke parametrisatie kennen. Bovendien zijn hun parametrisaties vanzelf disjunct, omdat deze parametrisaties alle moeten verschillen.

De modale exemplaren zijn heel wat minder specifiek, omdat ze op een open klasse van parametrisaties kunnen werken. Met name kunnen ze overlap vertonen, met generische exemplaren of andere modale exemplaren. Daarom speelt bij **polymorfe identificatie** de volgorde een belangrijke rol: van de operaties met eenzelfde naam wordt de meest specifieke genomen die op de gegeven parameters toepasbaar is. Generische operaties gaan voor modale operaties, en meer specifieke modale operaties voor minder specifieke.

Beschouw, als voorbeeld, de toewijzing. In elan is het "wordt"-teken `:=` een operatie als iedere andere. In de standaardpakketten is voor ieder concreet type een toewijzing gedeclareerd. Bovendien bestaat er een modale toewijzing

```
OP (T) := (T VAR destination, T source):
  { kopie van de waarde van source }
  { wordt aan destination toegewezen }
```

Voor die typen waarvoor een eigen toewijzingsoperatie is gedeclareerd, zoals de concrete typen, geldt dat deze operaties meer specifiek zijn dan de modale toewijzing. Voor de typen die zijn samengesteld uit concrete typen geldt de modale operatie. Voor abstracte typen geldt de modale operatie, tenzij er een specifieke toewijzing voor dit type is gedeclareerd. Dit geeft de gelegenheid, een andere semantiek aan de toewijzing te geven wanneer dat voor dit type nodig is.

12.2.3 Nogmaals verwijzingen

Als ELAN verwijzingen (11) kende zouden deze in de taal zijn ingebed als een concrete constructor `REF`, met bijvoorbeeld de volgende operaties:

```
M OP (M) DEREF (REF (M) r):
  { levert de waarde waar r naar
verwijst }

REF (M) OP (M) ADDR (M VAR v):
  { levert het adres dat verwijst }
  { naar de waarde van v }

OP (M) := (REF (M) dest, M source):
  { vervang de waarde waar dest naar
verwijst }
  { door een kopie van de waarde van
source }
```

Helaas ontbreken ons de taalmiddelen om deze zelf in te voeren.

12.3 Voorbeeld: lists, bags en sets

Bij wijze van voorbeeld van de definitie van typeconstructoren en polymorfe operaties geven we een ruiker van (verwante) datastructuren.

Sets onderscheiden zich van lijsten, doordat gelijke elementen slechts eenmaal worden opgenomen. Daartoe wordt de set hier als een geordende lijst gerealiseerd; de volgorde speelt echter strikt genomen geen rol.

Een bag, een zak vol elementen, kan net als een lijst wel gelijke elementen bevatten. In tegenstelling echter tot een lijst speelt de volgorde van toevoegen geen rol, twee bags zijn gelijk als zij dezelfde elementen bevatten. Om dit te bereiken, en om het zoeken van een element in een bag te vereenvoudigen wordt een bag gemeenlijk als geordende lineaire lijst gerealiseerd. Overigens speelt de volgorde der elementen geen rol.

Een set onderscheidt zich van een bag doordat hij geen duplicaten van elementen bevat — voeg je een element toe dat al aanwezig is dan heeft de toevoeging geen effect.

We voeren hier uitsluitend die operaties in die de drie constructoren `LIST`, `BAG` en `SET` gemeen hebben.

Merk op hoe voor sommige operaties (zoals `make empty`, `TAIL` en `put`) de drie constructoren over één kam geschoren kunnen worden, terwijl voor andere (met name de `+`) drie verschillende generische definities noodzakelijk zijn.

```
TYPE SET (EL) = STRUCT (EL head, SET
(EL) tail);
```

We kiezen eens een andere formele typenaam dan het afgezaagde `M`. Ieder woord in grote letters dat nog geen andere betekenis heeft kan worden gekozen. Hoewel de formele typenamen geen betekenis hebben zullen we in dit voorbeeld het elementtype aanduiden met `EL` en de naam `TY` gebruiken waar zowel lijsten als bags als sets kunnen staan.

```
TYPE LIST (EL) = STRUCT (EL head, LIST
(EL) tail);

TYPE BAG (EL) = STRUCT (EL head, BAG
(EL) tail);

PROC (TY) make empty (TY VAR t):
  CONCR t := NIL
ENDPROC make empty;

BOOL PROC is empty (TY t):
  ISNIL CONCR t
ENDPROC is empty;
```

De vorige twee operaties zijn niet alleen van toepassing voor de drie bedoelde constructoren, maar zelfs voor ieder abstract type. Ze zijn dus wel erg algemeen — misschien hadden we beter telkens drie meer specifieke definities kunnen geven, zoals

```
PROC (EL) make empty (LIST (EL) VAR t):
  CONCR t := NIL
ENDPROC make empty;
```

maar het gemak dient de mens.

De operator `HEAD` kunnen we het best voor elk van de drie constructoren apart definiëren.

```
EL OP (EL) HEAD (LIST (EL) l):
  CONCR l.head
ENDOP HEAD;

EL OP (EL) HEAD (BAG (EL) l):
  CONCR l.head
ENDOP HEAD;

EL OP (EL) HEAD (SET (EL) l):
  CONCR l.head
ENDOP HEAD;
```

Voor de operator `TAIL` kunnen we volstaan met één declaratie, waarbij de operator echter tevens wordt ingevoerd voor alle andere abstracte typen die een veld `tail` bezitten.

```
TY OP (TY) TAIL (TY l):
  CONCR l.tail
ENDOP TAIL;

PROC (TY) put (TY a):
  IF NOT is empty (a)
  THEN put (HEAD a); put (TAIL a) FI
ENDPROC put;

INT OP (TY) LENGTH (TY a):
  IF is empty (a)
  THEN 0 ELSE 1 + LENGTH TAIL a FI
ENDOP LENGTH;
```

De volgende operator heeft liefst twee formele typen `TY` en `EL`, hoewel hij uiteraard alleen voor bepaalde combinaties daarvan een betekenis heeft, namelijk die waarvoor een `+` bestaat.

```
OP (TY, EL) INCR (TY VAR a, EL b):
  a := b + a
ENDOP INCR;
```

De links-concatenatie is voor ieder van de drie constructoren verschillend. Hier worden de verschillen tussen lijsten, zakken en verzamelingen duidelijk.

```
SET (EL) OP (EL) + (EL a, SET (EL) b):
  IF is empty (b)
  THEN SET (EL): [a, NIL]
  ELIF a > HEAD b
  THEN SET (EL): [HEAD b, a + TAIL b]
  ELIF HEAD b > a
  THEN SET (EL): [a, b]
  ELSE b
  FI
ENDOP +;

BAG (EL) OP (EL) + (EL a, BAG (EL) b):
  IF is empty (b)
  THEN BAG (EL): [a, NIL]
  ELIF a > HEAD b
  THEN BAG (EL): [HEAD b, a + TAIL b]
  ELSE BAG (EL): [a, b]
  FI
ENDOP +;
```

```
LIST (EL) OP (EL) + (EL a, LIST (EL) b):
  LIST (EL): [a, b]
ENDOP +;
```

De volledige concatenatie van lijsten gaat als volgt:

```
LIST (EL) OP (EL) + (LIST (EL) a, b):
  IF is empty (a) THEN b
  ELSE HEAD a + (TAIL a + b)
  FI
ENDOP +;
```

De concatenatie gaat voor de andere twee constructoren analoog, al zal het duidelijk zijn dat daarbij andere links-concatenaties worden aangesproken.

Tot slot van dit voorbeeld volgt een programmaatje dat het verschil in gedrag van de drie constructoren proefondervindelijk aantoont. Probeer het maar eens uit.

```
program:
  LIST (INT) VAR intlist;
  BAG (INT) VAR intbag;
  SET (INT) VAR intset;
  make empty (intlist);
  make empty (intbag);
  make empty (intset);

  UPTO 10
  REP INT CONST i :: random (1, 10);
      intlist INCR i;
      intbag INCR i;
      intset INCR i
  ENDREP;
  put (LENGTH intlist); line;
  put (intlist); line;
  put (intbag); line;
  put (intset); line.
```

12.4 Voorbeeld: grammatica's

Als toepassing van de polymorfe lijsten schrijven we een programma om een contextvrije grammatica (zie Deel I) in te lezen, op te bergen en weer leesbaar uit te schrijven.

We recapituleren enige begrippen. Een **contextvrije grammatica** bestaat uit een of meer *regels*. Een regel bestaat uit een *linkerkant* ("left-hand-side", "lhs"), namelijk een enkel *symbool*, gevolgd door een dubbele punt, en een *rechterkant* ("right-hand-side", "rhs"), namelijk een of meer *alternatieven* gescheiden door puntkomma's en afgesloten door een punt. Een alternatief mag leeg zijn, dan wel bestaan uit een of meer symbolen, van elkaar gescheiden door komma's. De symbolen die in de linkerkant van een regel voorkomen heten *niet-terminaal*, de andere heten *terminaal*.

De terminale symbolen zijn de woorden van de door de grammatica beschreven taal, de niet-terminale zijn de begrippen in termen waarvan de taal beschreven wordt. Om ze te onderscheiden van andere symbolen zullen we terminale symbolen steeds tussen aanhalingstekens ("quote"s) zetten.

Als voorbeeld van het gebruik van dit formalisme geven we een contextvrije grammatica die beschrijft hoe contextvrije grammatica's eruit zien.

cf grammar: rule, cf grammar; rule.

rule: lhs, ":", rhs, "." .

lhs: symbol.

rhs: alternative, ";", rhs; alternative.

alternative: symbol; symbol, ",", alternative; .

Grammatica's in de zojuist beschreven notatie willen we kunnen inlezen, opslaan en weer uitgeven. Daartoe moeten we een manier van opslaan (een interne representatie) van de ingelezen grammatica bedenken. We hoeven alleen de essentiële delen van de regels metterdaad op te slaan, niet de leestekens maar wel de symbolen, de lijsten daarvan die de alternatieven vormen, de lijsten van alternatieven die de rechterkanten vormen en de lijsten van regels. We bereiken dit doel door met ieder van de bedoelde constructen een type te associëren. De volgende typedeclaraties zijn als het ware een parafrase van de eerder gegeven definitie van cf grammar.

```
LET GRAMMAR = LIST (RULE);

LET RULE = STRUCT (SYMB lhs, LIST (ALT)
rhs);

LET ALT = LIST (SYMB);

LET SYMB = STRUCT (TEXT t, BOOL is
term);
```

Merk op dat de ingelezen grammatica als een lijst van lijsten wordt voorgesteld, het is een onregelmatige boom (zie 10.6).

12.4.1 Lexicaal interface

Voor het lezen maken we gebruik van eenzelfde lexicaal interface als in hoofdstuk 10.4.

```
PROC skip:
  buffer := TAIL buffer;
  IF buffer = ""
  THEN read (buffer)
  FI;
  IF ahead (" ")
  THEN skip
  FI
ENDPROC skip;

PROC skip (TEXT t):
  assert (ahead (t));
  skip
ENDPROC skip;
```

```
TEXT PROC next identifier:
  TEXT VAR idf :: "";
  {letter ahead}
  REP
    idf CAT HEAD buffer;
    skip
  UNTIL NOT ahead (letters + digits)
  ENDREP;
  idf
ENDPROC next identifier;

BOOL PROC is (TEXT s):
  IF ahead (s)
  THEN
    skip;
    true
  ELSE false
  FI
ENDPROC is;

BOOL PROC ahead (TEXT alfabet):
  pos (alfabet, HEAD buffer) > 0
ENDPROC ahead;
```

We kiezen als wortel van het programma

```
program:
  LET letters =
"abcdefghijklmnopqrstuvwxyz",
      digits = "0123456789";
  old file ("input");
  new file ("output");
  GRAMMAR VAR g;
  TEXT VAR buffer :: " ";
  read (g);
  write (g).
```

12.4.2 Inlezen van CF grammatica's

Het einde van een grammatica zullen we herkennen aan een dollarteken na de laatste regel daarvan. We hanteren dus eigenlijk een licht gewijzigde definitie

cf grammar: rule, cf grammar; rule, " "$".

De volgende leesprocedures zijn met wat knutselwerk vrij direct uit de gegeven syntax af te leiden (weer een voorbeeld van **syntax-gestuurde programmering**).

```
PROC read (GRAMMAR VAR g):
  IF ahead ("$")
  THEN make empty (g)
  ELSE
    RULE VAR r;
    read (r);
    read (g);
    g := r + g
  FI
ENDPROC read;
```

```
PROC read (RULE VAR r):
  SYMB VAR lhs;
  LIST (ALT) VAR rhs;
  read (lhs);
  skip (":");
  read (rhs);
  skip (".");
  r := [lhs, rhs]
ENDPROC read;

PROC read (SYMB VAR s):
  IF ahead (letters)
  THEN s := [next identifier, false]
  ELSE
    skip (quote);
    s := [next identifier, true];
    skip (quote)
  FI
ENDPROC read;
```

Merk op dat bij het inlezen ieder symbool wordt voorzien van een markering die aangeeft of het terminaal of niet-terminaal is.

```
PROC read (LIST (ALT) VAR rhs):
  LIST (SYMB) VAR alt;
  read (alt);
  IF is (";")
  THEN
    read (rhs);
    rhs := alt + rhs
  ELSE rhs := list (alt)
  FI
ENDPROC read;

PROC read (LIST (SYMB) VAR alt):
  IF ahead (";.")
  THEN make empty (alt)
  ELSE
    SYMB VAR m;
    read (m);
    IF is (",")
    THEN
      read (alt);
      alt := m + alt
    ELSE alt := list (m)
    FI
  FI
ENDPROC read;
```

We hebben hiermee een **generische familie** van procedures verkregen, die samen de taak van het inlezen en opbergen van de grammatica vervullen.

12.4.3 Afdrukken van de grammatica

Bij wijze van contrôle op het inlezen willen we de opgeslagen grammatica in een keurige gestandaardiseerde layout afdrukken (**pretty print**). Voor het inlezen is overigens iedere layout aanvaardbaar, al is ie nog zo rommelig — spaties en regelovergangen worden immers door onze lexicale procedures overgeslagen.

We schrijven weer een generische familie van procedures.

```
PROC write (GRAMMAR g):
  IF NOT is empty (g)
  THEN
    write (g.hd);
    write line;
    write (g.tl)
  FI
ENDPROC write;

PROC write (RULE r):
  write (r.lhs);
  write (":");
  write (r.rhs);
  write (".");
  write line
ENDPROC write;

PROC write (LIST (ALT) rhs):
  IF NOT is empty (rhs)
  THEN
    write line;
    write ("    ");
    write (rhs.hd);
    IF NOT is empty (rhs.tl)
    THEN
      write (";");
      write (rhs.tl)
    FI
  FI
ENDPROC write;

PROC write (LIST (SYMB) alt):
  IF NOT is empty (alt)
  THEN
    write (alt.hd);
    IF NOT is empty (alt.tl)
    THEN
      write (",");
      write (alt.tl)
    FI
  FI
ENDPROC write;

PROC write (SYMB s):
  IF s.is term
  THEN write (quote + s.t + quote)
  ELSE write (s.t)
  FI
ENDPROC write;
```

Wie het probeert zal merken dat zo'n generische familie van algoritmen over een gedetailleerde hiërarchie van typen (zoals `GRAMMAR`, `RULE`, `LIST(ALT)`, `ALT`, `SYMB`) dankzij de beveiliging door het typemechanisme vrij gemakkelijk zonder grote vergissingen geschreven kan worden.

12.5 Eindige homogene verzamelingen

We gaan nog wat nader in op de implementering van verzamelingen en de typische operaties daarop.

Iedere implementering op een computer van het wiskundig begrip *verzameling* is noodzakelijkerwijze eindig, beperkt door het eindige geheugen van de computer. Het aantal elementen in de verzameling moet eindig zijn, zelfs al is er geen specifiek maximum aan gesteld. Bovendien kunnen we niet zomaar verzamelingen van willekeurige elementen representeren en kiezen daarom ervoor dat alle elementen van de verzameling eenzelfde type hebben. Kortom, we beperken ons tot eindige **homogene verzamelingen**.

Voor eindige homogene verzamelingen liggen drie mogelijke implementaties voor de hand:

- Een implementatie als een rij
- Een implementatie als een geordende lijst (zie ook 9.4.4)
- een implementatie als een geordende boom (zie ook 10.3).

Voor de eerste twee spreekt de eenvoud, van de derde is te verwachten dat deze voor grote verzamelingen voordeel oplevert.

12.5.1 Verzamelingen geïmplementeerd als rijen

Alle deelverzamelingen van een eindige homogene verzameling met `aantal` elementen kunnen we representeren als een rij

```
TYPE SET = ROW aantal BOOL;
```

waarvan de elementen aangeven of het corresponderende element in de verzameling aanwezig is. Zoals we al zagen in 11.5.2 kan hierbij soms met vrucht van de machinestructuur gebruik gemaakt worden.

Is alleen het maximum aantal elementen bekend maar kunnen ze verder uit een veel grotere verzameling van elementen van bijv. type `EL` gekozen worden, dan kunnen we gebruik maken van een stack-achtige representatie

```
  TYPE SET = STRUCT (ROW max aantal EL
elementen, INT aantal);
```

Het gebruik en de consequenties van deze representaties spreken voor zich.

12.5.2 Verzamelingen geïmplementeerd als lijsten

Een interessante implementering van verzamelingen is gebaseerd op lijsten. We hoeven geen expliciete aannamen te maken over het aantal elementen, de grootte van het beschikbare geheugen legt daaraan vanzelfsprekende beperkingen op.

Ter versnelling van het opzoeken van elementen verdient het aanbeveling dat zulke lijsten geordend worden bijgehouden. Daarbij komt iedere ordening op de elementen in aanmerking die voldoende eenvoudig berekend kan worden: we maken er immers geen ander gebruik van dan de indeling van de lijsten, dus het hoeft niet een intuïtief inzichtelijke ordening te zijn.

Geordende lijsten zijn behandeld in 9.4.4, een daarop gebaseerde constructor `SET` in 12.3.

12.5.3 Verzamelingen geïmplementeerd als bomen

Worden de verzamelingen waarin elementen moeten worden ingevoegd en opgezocht groot, dan ligt de overgang naar geordende bomen voor de hand. We maken gebruik van dezelfde ordening als bij de lijsten.

```
PACKET setpacktree
DEFINES SET, make empty, is empty, set,
        INCR, IN, U, I, -,
        card, put, apply all:
```

Hier staan `U` en `I` voor resp. de vereniging ("union") en de doorsnede ("intersection"); de beperkte tekenset van ELAN stelt ons helaas niet in staat $\cup$ en $\cap$ te gebruiken.

```
TYPE SET (M) = STRUCT (M info, SET (M)
left, right);

PROC (M) make empty (SET (M) VAR s):
  CONCR s := NIL
ENDPROC make empty;

BOOL PROC (M) is empty (SET (M) s):
  ISNIL CONCR s
ENDPROC is empty;
```

Denotatie voor de singleton set:

```
SET (M) PROC (M) set (M el):
  SET (M): [el, SET (M): NIL, SET (M):
NIL]
ENDPROC set;
```

De bouwoperator `tree` ziet er zelf niet op toe dat de bedoelde ordeningsrelatie tussen de deelbomen gegarandeerd wordt. Hij moet dan ook met voorzichtigheid worden genoten. Hij wordt zeker niet uit dit pakket geëxporteerd!

```
SET (M) PROC (M) tree
          (M info, SET (M) left, right):
  SET (M): [info, left, right]
ENDPROC tree;
```

Bij wijze van links-concatenatie kiezen we voor de verandering eens de toevoeging van een element aan een set-variabele.

```
OP (M) INCR (SET (M) VAR s, M el):
  IF is empty (s)
  THEN s := set (el)
  ELIF el < s.info
  THEN
    s.left INCR el
  ELIF s.info < el
  THEN
    s.right INCR el
  FI
ENDOP INCR;
```

Dank zij de ordening is het eenvoudig te bepalen of een bepaald element in de verzameling aanwezig is.

```
BOOL OP (M) IN (M el, SET (M) s):
  IF is empty (s)
  THEN false
  ELIF s.info < el
  THEN el IN s.right
  ELIF el < s.info
  THEN el IN s.left
  ELSE true
  FI
ENDOP IN;
```

Nog wat eenvoudige operaties voor we aan de moeilijkere gevallen komen.

```
INT PROC (M) card (SET (M) s):
  IF is empty (s)
  THEN 0
  ELSE 1 + card (s.left) + card
(s.right)
  FI
ENDPROC card;
```

```
PROC (M) put (SET (M) s):
  IF is empty (s)
  THEN
  ELSE
    put (s.left);
    put (" ");
    put (s.info);
    put (s.right)
  FI
ENDPROC put;
```

Nu komen we aan de set-operaties *vereniging* `U`, *doorsnede* `I` en het asymmetrisch *verschil* `-`. Op het eerste gezicht moeten deze operatoren prachtige staaltjes van recursief programmeren opleveren, maar dat valt tegen: Omdat de structuur van de bomen zo weinig met de inhoud van de verzamelingen te maken heeft kunnen deze operaties de structuur nauwelijks uitbuiten.

Een vrij eenvoudige realisering van de vereniging is

```
SET (M) OP (M) U (SET (M) s1, s2):
  IF is empty (s1)
  THEN s2
  ELIF is empty (s2)
  THEN s1
  ELSE
    SET (M) VAR s :: s1;
    s INCR s2.info;
    s := s U s2.left;
    s := s U s2.right;
    s
  FI
ENDOP U;
```

Bij nadere beschouwing is dit een verschrikkelijke oplossing: het resultaat wordt op de meest omslachtige wijze opgebouwd uit de elementen van de twee deelverzamelingen. Bomen worden in grote aantallen afgebroken en weer opgebouwd. Het lijkt sterk de vraag of de resulterende boom enigszins gebalanceerd is, of juist gedegeneerd tot een dure lijst.

Bovendien is dit, door het gebruik van een niet bepaald nodige variabele, een staaltje van de voetgangers-stijl bij het programmeren: “Als je het niet meer weet dan introduceer je maar een variabele, en als je het dan nog niet weet nog maar een”.

Laten we in plaats daarvan zoeken naar een recursieve oplossing zonder variabelen, die naar mogelijkheid de structuur van de bomen uitbuit. Iets is daar wel uit te halen:

- als `s1.info` gelijk is aan `s2.info` kun je de linker- en rechter-deelbomen onafhankelijk verenigen
- als `s1.info` kleiner is dan `s2.info` dan behoren de elementen in de linker onderboom van `s1` ook alle links van `s2`
- en het omgekeerde geldt bij de groter-dan-relatie.

```
SET (M) OP (M) U (SET (M) s1, s2):
  IF is empty (s1)
  THEN s2
  ELIF is empty (s2)
  THEN s1
  ELIF s1.info < s2.info
  THEN tree (s2.info, s1.info + (s1.left
U s2.left), s2.right)
       U s1.right
  ELIF s2.info < s1.info
  THEN tree (s1.info, s2.info + (s2.left
U s1.left), s1.right)
       U s2.right
  ELSE tree (s2.info, s1.left U s2.left,
s1.right U s2.right)
  FI
ENDOP U;
```

Ook deze oplossing laat nog een nare smaak achter: door een diepgaander analyse moet toch een eficiëntere oplossing te vinden zijn! We hebben hier weer een situatie waar de vorm de inhoud in de weg zit (zie 10.3.1).

De volgende twee operatoren zijn iets overtuigender:

```
SET (M) OP (M) I (SET (M) s1, s2):
  IF is empty (s1)
  THEN s1
  ELIF is empty (s2)
  THEN s2
  ELIF s1.info IN s2
  THEN SET (M): [s1.info, s1.left I s2,
s1.right I s2]
  ELSE (s1.left I s2) U (s1.right I s2)
  FI
ENDOP I;
```

```
SET (M) OP (M) - (SET (M) s1, s2):
  IF is empty (s1) OR is empty (s2)
  THEN s1
  ELIF s1.info IN s2
  THEN (s1.left - s2) U (s1.right - s2)
  ELSE SET (M): [s1.info, s1.left - s2,
s1.right - s2]
  FI
ENDOP -;
```

Tot slot definiëren we een procedure die een meegegeven actie `p` toepast op alle elementen van de verzameling, tijdens een in-order traversie van de betrokken boom.

```
PROC (M) apply all (SET (M) s,
                    PROC (M) p):
  IF is empty (s)
  THEN
  ELSE
    apply all (s.left, PROC (M) p);
    p (s.info);
    apply all (s.right, PROC (M) p)
  FI
ENDPROC apply all;
```

Als toepassing hiervan kan de boven gegeven definitie van `put` worden vervangen door

```
PROC (M) put (SET (M) s):
  apply all (s, PROC (M) put)
ENDPROC;

ENDPACKET setpacktree;
```

In het volgende hoofdstuk geven we een uitgebreide toepassing van dit verzameling-pakket.

12.6 Conclusie

ELAN is een van de weinige algoritmische programmeertalen die de veiligheid, bereikt door preciese en eenduidige typering van alle constructen (**strong typing**) verenigt met de flexibiliteit verkregen door polymorfie (zowel genericiteit als modaliteit). Daarbij is er naar gestreefd, de notatie zo beknopt en doorzichtig mogelijk te houden.

Een modale procedure of operator heeft een of meer formele typen. Deze worden niet vermeld als formele parameters in de heading van de algoritme, zoals wellicht voor de hand zou liggen. Een type is in ELAN namelijk geen waarde, er bestaan geen typeleverende uitdrukkingen die als actuele parameter zouden kunnen dienen.

In plaats daarvan wordt het `PROC`- of `OP`-symbool in de definitie gemarkeerd met de namen van zijn formele typen. Deze markering kan echter met evenveel recht worden gezien als een prefixering van de naam van de algoritme met een of meer typen (en sommige programmeertalen noteren dit ook zo). Het actuele type bepaalt welke operatie zich achter de operator of procedurenaam verschuilt. Deze prefix wordt in ELAN uitsluitend bij de definitie aangegeven. Ook de noodzakelijke operaties hoeven niet in het programma te worden opgesomd.

Door deze conventies is bereikt dat het gebruik van polymorfe typen en operaties niet (zoals in ADA) met veel bureaucratie behept is.

12.7 Opgaven

De eerste drie opgaven sluiten aan bij het voorbeeld van grammatica's, dat naar believen kan worden uitgebreid tot een hulpmiddel voor de analyse van de eigenschappen van contextvrije grammatica's.

1. (nonfalse) Een nonterminal wordt *nonfalse* genoemd als hij een lege terminale productie heeft. Dat is het geval als hij een leeg alternatief heeft, dan wel een alternatief dat uitsluitend nonfalse nonterminals bevat (en geen terminal symbols). Schrijf een generieke familie van algoritmen die bepaalt of een nonterminal nonfalse is.
2. (linksrecursie) Een nonterminal `n` is linksrecursief als hij een (directe of indirecte) productie bezit die met `n` begint; `n` roept als het ware zichzelf linksrecursief op. Schrijf een generische familie van algoritmen die bepaalt welke nonterminals linksrecursief zijn, uitgaande van de wetenschap welke nonterminals nonfalse zijn.
3. (herkennen) Schrijf een generische familie van (backtrack) algoritmen met als wortel

   ```
   BOOL PROC recognize (TEXT input, GRAMMAR
   g):
   ```

   die berekent of de *input* een terminale productie is van de eerste nonterminal uit de context-vrije grammatica `g`. Dit is een pittige opgave die enige kennis van formele grammatica's vereist.
4. (symbolisch rekenen) Een veelterm is een som van eentermen. Een eenterm is het product van een constante en een aantal factoren. Een factor is een variable tot een zekere macht verheven. Een variabele wordt aangeduid door een letter. Ontwerp een abstract datatype `POLY`, gebaseerd op `LIST`, dat het elementaire rekenen met veeltermen realiseert. Definieer in ieder geval de volgende operaties:

   ```
   POLY OP (POLY a, b) +, -, *, /
   POLY PROC poly (TEXT variabele, INT
   macht)
   POLY PROC poly (REAL constante)
        PROC assign (TEXT variabele, REAL
   waarde)
   REAL PROC evaluate (POLY p)
        PROC put (POLY p)
   POLY PROC diff (POLY p, TEXT VAR
   variabele)
   ```

12.8 Probleemoplossen

Dit hoofdstuk betreft een toepassing van polymorfe datastructuren en algoritmen, waarbij het bereiken van flexibiliteit centraal staat. We definiëren een raamwerk dat voor een wijde klasse van opgaven gebruikt kan worden.

12.9 Kunstmatige intelligentie

De **kunstmatige intelligentie** ("artificial intelligence", "AI") is de wetenschap van de bestudering en nabootsing van typisch menselijke denkprocessen met behulp van automaten. De AI is vrijwel net zo oud als de informatica (zie bijv. [MIC86]), beide vakgebieden groeiden sinds 1950 hand in hand op. Iedere toename in de snelheid en opslagcapaciteit van de computer voerde tot een toename in mogelijkheden van de AI en telkens weer verlevendigden methoden en problemen uit de AI het informatica-landschap, dat bijzonder saai zou zijn als het enkel door administratieve en numerieke toepassingen bepaald werd. Verder speelt de AI traditioneel de rol van kweekvijver voor ideeën, technieken en methoden van de informatica.

Van oudsher onderzoekt de AI het menselijk vermogen, problemen op te lossen. De AI probeert daartoe met behulp van automaten processen te realiseren die mensen dankzij hun intelligentie kunnen doen, en langs die weg te achterhalen hoe onze intelligentie in elkaar zit. Want we weten maar weinig van typisch menselijke activiteiten als: patroonherkenning; leerprocessen; het spreken, verstaan en vertalen van menselijke spraak; het bedrijven van wiskunde; het oplossen van puzzels en het spelen van schaak, bridge en andere ingewikkelde meerpersoons spelen.

In dit hoofdstuk richten we ons op het oplossen van puzzels. De inspiratie voor dit hoofdstuk is goeddeels afkomstig van een zeer oud verhaal van Lambert Meertens [MEE62], waarbij we de daar behandelde problemen met wat moderner middelen aanvatten.

Veel puzzels en spelletjes zijn op de volgende wijze te beschrijven: gegeven is een *beginsituatie*, een *eindsituatie* en verder de *spelregels* die bepalen welke zetten in een bepaalde situatie geoorloofd zijn. De opgave is, van de beginsituatie tot de eindsituatie te komen door het toepassen van geoorloofde zetten. Daarbij zijn we met name geïnteresseerd in een *optimale* oplossing: een oplossing met het geringste aantal zetten. Enige voorbeelden van dit soort problemen:

A) <u>beginsituatie</u> pppp–qqqq
<u>eindsituatie</u> qqqq–pppp
In een tunnel staan 4 tegenover 4 kamelen met daartussen een lege plaats. "p" stelt voor een kameel die naar rechts, "q" een kameel die naar links kijkt, "–" de lege plaats.
Iedere kameel kan één plaats vooruit, of twee plaatsen (daarbij over een kameel springend), mits hij dan op de lege plaats terecht komt.

B) <u>beginsituatie</u> geit kool wolf — man
<u>eindsituatie</u> man — geit kool wolf
Aan een riviertje staan een geit, een kool en een wolf, die naar de overkant willen. Er is een man met een bootje, waarin hij slechts één van de drie tegelijk kan overroeien.
Wanneer hij er niet bij is, probeert de wolf de geit, en de geit de kool te verslinden.

C) <u>beginsituatie</u> (12) (7) (5)
<u>eindsituatie</u> (6) (6) (0)
Bij een inbraak hebben een stel dieven enkele volle vaten met wijn meegenomen (bijv. 3 vaten van 12, 7 en 5 liter).
Gevraagd wordt de wijn op een bepaalde manier te verdelen, waarbij slechts van deze vaten gebruik mag worden gemaakt.

D) <u>beginsituatie</u> $\neg P \vee Q$
<u>eindsituatie</u> $\neg Q \supset \neg P$
De beginsituatie moet in de eindsituatie getransformeerd worden door stellingen of axioma's toe te passen.

Het gaat hier om problemen die in de AI een lange traditie hebben; samen met enige andere zijn ze terug te vinden in een boek als [SLA71], en verder in menig PROLOG leerboek.

In al deze gevallen gaat het om de vraag: hoe laten we de machine zo'n probleem oplossen?

De meest voor de hand liggende methode, nl. gewoon proberen tot een oplossing is gevonden, blijkt al gauw teveel tijd te vereisen; bovendien hoeft een gevonden oplossing nog helemaal niet de beste te zijn. Wij moeten de zaak dus systematischer aanpakken.

12.10 Depth-first search

De toestanden die vanuit de beginsituatie volgens de spelregels bereikbaar zijn kunnen wij ons als een boom voorstellen, de *toestandsboom* ("state tree"), waarin de knopen toestanden voorstellen, en iedere knoop verbonden is met de toestanden die van daaruit beschikbaar zijn. De toestandsboom bestaat uit lagen, waarbij de k-de laag die toestanden bevat die in k stappen vanuit de begintoestand bereikbaar zijn.

De toestandsboom kan systematisch worden doorlopen door middel van een **backtrack-algoritme** (zie Deel I) met de volgende schematische structuur:

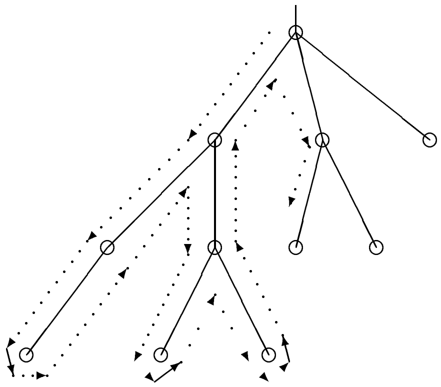


Figure 12.1: Depth-first search

```
PROC doorzoek boom
       (INT k, SIT huidige toestand,
                   eind toestand):
  IF huidige toestand = eindtoestand
  THEN meld succes in k stappen
  ELSE
    SET (SIT) CONST bereikbaar ::
     alle hiervandaan bereikbare
toestanden;
    SIT VAR volgende;
    FOR volgende ∈ bereikbaar
    REP IF legale toestand (volgende)
        THEN doorzoek boom (k+1,
volgende eindtoestand)
        FI
    ENDREP
  FI
ENDPROC doorzoek boom;
```

Hierbij geeft het predicaat `legale toestand` de mogelijkheid, bepaalde *heuristieken* toe te passen om het zoekproces in te perken.

De resulterende manier van doorlopen heeft als karakteristiek dat de takken van de toestandsboom in een bepaalde kanonieke volgorde worden doorlopen, waarbij iedere tak ten einde toe wordt doorlopen voordat aan de volgende wordt begonnen. Het zoekproces gaat dus allereerst in de diepte ().

Een gunstige eigenschap van deze benadering is dat slechts een klein deel van de toestandsboom (nl. het pad van de wortel naar de huidige toestand) moet worden opgeslagen, zodat het geheugenbeslag bescheiden is.

Daartegenover staan 3 specifieke nadelen van depth-first search:

1. De bedoelde probleemstelling moet zodanig zijn dat de toestandsboom een eindige diepte heeft (zo niet, dan moeten heuristieken daarvoor zorgen) omdat anders een eindeloze recursie ontstaat.
2. Eenzelfde toestand kan vaak langs verschillende reeksen van zetten en op verschillende diepten in de boom bereikt worden. Het is daardoor bij depth-first search erg moeilijk te voorkomen dat eenzelfde deelboom vele malen doorlopen wordt.
3. Wordt een oplossing gevonden dan is nog lang niet zeker dat het een optimale oplossing betreft. Daartoe moet in principe de hele toestandsboom worden doorlopen. Bijzondere heuristieken (als de *branch-and-bound* heuristiek, zie Deel I) moeten worden ingezet om het resterende zoekproces te beperken.

Daarnaast is er nog een algemeen probleem, dat deze benadering van het zoekproces gemeenschappelijk heeft met andere: Meestal beschouwen we *exponentiële problemen*, d.w.z. problemen waarbij de toestandsboom in afhankelijkheid van de bereikte diepte k een omvang $O(a^k)$ heeft, waarbij a een probleemafhankelijke constante is die aangeeft hoeveel opvolgers een situatie gemiddeld heeft. Die karakteristieke factor heet de *vertakkingsfactor* (“branching factor”) van de boom.

Bij het in 12.9 geschetste kamelenprobleem (probleem A) heeft a de waarde 4; voor schaakproblemen (bij toepassing van geschikte heuristieken) wordt veelal een vertakkingsfactor 10 aangegeven.

De complexiteit van de probleemoplossing wordt dus bepaald door twee constanten, die allebei zeer zwaar aantikken. We moeten alles in het werk stellen deze constanten klein te houden.

De (maximale) diepte van de toestandsboom hangt meestal direct samen met een bepaalde parameter van het probleem: het aantal kamelen, het aantal wijnvaten, het aantal vrije velden, etc. Voor backtrack-algoritmen geldt vrij algemeen de ervaring dat ze tot een zekere probleemgrootte vlot werken terwijl een iets groter probleem door het exponentiëel gedrag prohibitief veel rekentijd en geheugencapaciteit vergt.

De vertakkingsfactor van de toestandsboom kan via allerlei slimme heuristieken beperkt worden, hoewel de heuristieken al gauw zo ingewikkeld worden dat de daardoor veroorzaakte overlast uiteindelijk verdere verbetering onmogelijk maakt.

Het probleem van de meervoudig optredende toestanden kan in principe worden opgelost door het onthouden van de reeds voorgekomen toestanden (**memoizing**) maar het aantal te onthouden toestanden kan snel uit de hand lopen (waardoor een te groot geheugenbeslag en tijdsbeslag optreedt). Klaarblijkelijk is de representatie van de toestandsboom als een enkel pad niet de meest gelukkige, en moeten we gaan overwegen met *verzamelingen* van toestanden te werken, en daardoor doublures voorkomen.

In het boven weergegeven schema van de backtrack-algoritme is deze gedachte gedeeltelijk al gerealiseerd, doordat de k-de laag als een verzameling van toestanden wordt opgeslagen, zodat tenminste de direct op een bepaalde toestand volgende aan elkaar gelijke toestanden maar één keer worden opgeslagen. Desondanks kan natuurlijk nog zeer veel dubbel werk over blijven.

12.11 Breadth-first search

Laten wij overgaan op een andere volgorde van doorlopen van de toestandsboom: we bepalen telkens, uitgaande van de situaties in de k-de laag, de verzameling van situaties bereikbaar in de $(k+1)$-ste laag totdat in die nieuwe laag de eindsituatie voorkomt. Hiermee garanderen we dat iedere laag op zich geen doublures bevat en dat de eerste gevonden oplossing optimaal is.

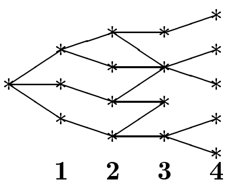


Figure 12.2: Breadth-first search

We zoeken dus in de breedte alvorens in de diepte te gaan (**breadth-first search**). Verschillende situaties in de k-de laag kunnen (afhankelijk van de spelregels) voeren tot éénzelfde situatie in de $(k+1)$-ste laag. De zoekstructuur is nu niet meer een boom maar een *DOAG:* een gerichte georiënteerde acyclische graaf (zie 11.3.4).

Daarbij steekt er nog addertje onder het gras: in de $(k+1)$-ste laag zou een situatie kunnen voorkomen die al eerder is opgetreden (in een van de lagen $0 \ldots k-1$). We moeten dus expliciet die situaties wegschrappen uit de k-de laag die al eerder zijn opgetreden.

Te dien einde beschouwen we de toestandsboom als bestaande uit twee delen, de *bulk*, d.w.z. de verzameling van situaties behorende tot de lagen $0 \ldots k-1$, en het *front*, de verzameling van situaties die in de k-de laag voor het eerst optreden. Hierbij maken we expliciet gebruik van de eigenschap van verzamelingen, dat gelijke elementen daarin slechts één keer optreden.

Op basis van deze gedachten is de kern van het breadth-first zoekproces:

```
SET (SIT) VAR bulk, front, next;
make empty (bulk);
front:= set (begintoestand);
WHILE NOT eindtoestand IN front
REP next:=
      {situaties bereikbaar vanuit front};
   bulk:= bulk U front;
   front:= next - bulk
ENDREP
```

We hebben hier een prachtige toepassing van verzamelingen te pakken, die we nog wat nader gaan uitwerken.

12.12 General problem solver (1)

De verschillende puzzels die we willen bekijken hebben alle een sterk overeenkomstige structuur. Dit brengt ons op de gedachte, één raamwerk te maken waarin verschillende probleembeschrijvingen als het ware kunnen worden ingevoegd. We maken daartoe gebruik van pakketten die polymorfe typen en algoritmen definiëren.

We willen mikken op de volgende laagsgewijze structuur van pakketten:

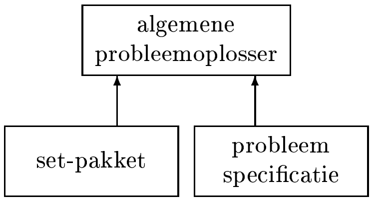


Figure 12.3: Laagsgewijze structuur van GPS

Een probleem kan worden gespecificeerd door het definiëren van de volgende zaken:

1. een type `SIT`, geschikt voor het bedoelde probleem
2. de begin- en eindtoestand
3. een procedure `from`, die bij een gegeven toestand berekent welke toestanden daaruit volgens de spelregels bereikbaar zijn, van het type

```
SET (SIT) PROC from (SIT current state):
  { all successors of current state }
```

De realisering van een en ander zullen wij dadelijk voor enige problemen concretiseren, eerst beschouwen we de gepostuleerde "algemene probleem oplosser".

Het idee, een algemene probleemoplosser (**General Problem Solver**, GPS) te ontwikkelen voor een brede klasse van problemen, gespecificeerd als hierboven, is terug te voeren tot [NSS59].

Om van de schets uit de vorige sectie een robuuste probleemoploser te maken moeten we rekening houden met een aantal zaken:

- Natuurlijk kan er ook een onoplosbaar probleem gesteld worden, d.w.z. een probleem waarbij de eindsituatie niet in de toestandsboom voorkomt. Daarbij moeten we onderscheiden tussen eindige problemen (waarbij de toestandsboom begrensd is en dus op een bepaalde diepte geen nieuwe situaties meer worden gevonden) en eindeloze problemen (of ten minste die problemen waarbij de toestandsboom zo uitgebreid is dat de geheugencapaciteit van de computer wordt overschreden).
- We willen méér uitvoer geven dan alleen de vaststelling of de oplossing gelukt dan wel mislukt is, bijvoorbeeld een pad van situaties tussen begin- en eindsituatie. Ook willen we de computer aan

het werk zien, om de voortgang van het zoekproces te kunnen volgen.

Voor eindeloze of te grote problemen treffen we geen bijzondere voorziening — als het computergeheugen vol zit merken we dat wel. Voor eindige problemen zullen we ook het mislukken netjes signaleren.

Wat betreft de voortgangsrapportage beperken we ons tot het melden van het laagnummer alsmede per laag de omvang van bulk en front. Ook zullen we het front zelf afdrukken, wat met zich mee brengt dat voor het type `SIT` een procedure `put` moet bestaan. Uiteindelijk melden we het succes of de mislukking. (Het bij succes aangeven van een kortste pad tussen begin- en eindsituatie laten we als opgave aan de lezer).

We gaan uit van een set-pakket zoals beschreven in 12.5.3. We implementeren de General Problem Solver als een (recursieve) procedure `search` zonder parameters, waarbij `k` (het nummer van de huidige laag), `bulk`, `front` en `next` globale variabelen zijn. We gebruiken bij alle problemen hetzelfde hoofdprogramma:

```
program:
  SIT VAR initial, final;
  initial ASK "Initial state=";
  final ASK "Final   state=";
  SET (SIT) VAR bulk, front, next;
  bulk:= set (initial);
  front:= set (initial);
  INT VAR k :: 0, incount :: 0;
  search;
  line; put (incount).

OP ASK (SIT VAR state, TEXT message):
  line;
  put (message);
  get (state)
ENDOP ASK;
```

De variabele `incount` dient om een oogje op de complexiteit van het zoekproces te houden. Binnen het gebruikte set-pakket turven we hoe vaak de operator `IN` gebruikt wordt.

De kern van de GPS luidt

```
  next:= situaties bereikbaar
          vanuit front;
  bulk:= bulk U front;
  front:= next - bulk
```

Dit is wel fraai, maar niet erg efficiënt. In plaats van `next` eerst op te bouwen en daarna het verschil met `bulk` te nemen kunnen we ook direct alle oude situaties uit `next` elimineren en de nieuwe situaties in `bulk` opnemen, volgens het schema

```
  next:= situaties bereikbaar
          vanuit front - bulk;
  bulk:= bulk U next;
  front:= next
```

Bovendien kunnen we de eerste twee regels combineren door, uitgaande van `front`, alle nieuwe bereikbare situaties meteen zowel aan `bulk` als aan `next` toe te voegen. We hebben dan de operatoren `U` en `-` niet meer nodig en volstaan met `INCR` en `IN`.

In plaats van de functie `from` voeren we daartoe een actie `extend from` in, die (op een van het probleem afhankelijke wijze) de vanuit zijn argument bereikbare situaties probeert toe te voegen aan zowel `bulk` als `next`, via een actie `extend` die weer in de GPS wordt opgenomen. De rest van het GPS-pakket luidt daarmee als volgt:

```
PROC search:
  report;
  IF is empty (front)
  THEN line;
       put ("Failure!")
  ELIF final IN front
  THEN line;
       put ("Success!")
  ELSE
   make empty (next);
   apply all (front, PROC (SIT) from);
   front:= next;
   search
  FI.

report:
  line (2);
  put (k);
  k INCR 1;
  put (card (bulk));
  put (card (front));
  put (":");
  line;
  put (front).
ENDPROC search;

PROC extend (SIT s):
  IF s IN bulk
  THEN
  ELSE
    next INCR s;
    bulk INCR s
  FI
ENDPROC extend;
```

In 12.15 gaan we nog wat nader in op het benodigde `SET`-pakket. Eerst een voorbeeld van een toepassing.

12.13 Voorbeeld: Tekstpermutaties

Als eerste voorbeeld behandelen we het probleem, een gegeven tekst om te zetten in een andere door toepassing van de volgende twee transformaties:

- Het kop-element van de tekst wordt achteraan gezet:

  $\boxed{t_1}\boxed{t_2}\ \cdots\ \boxed{t_n} \quad \Rightarrow \quad \boxed{t_2}\ \cdots\ \boxed{t_n}\boxed{t_1}$

- De twee eerste elementen worden verwisseld:
  $\boxed{t_1}\boxed{t_2}\boxed{t_3}\ \cdots\ \boxed{t_n} \quad \Rightarrow \quad \boxed{t_2}\boxed{t_1}\boxed{t_3}\ \cdots\ \boxed{t_n}$

Dit gelukt uiteraard alleen als de ene tekst een permutatie is van de andere.

We kiezen de voor de hand liggende representatie voor situaties:

```
LET SIT = TEXT;

PROC extend from (SIT state):
  extend (TAIL state + HEAD state);
  extend (HEAD TAIL state + HEAD state +
TAIL TAIL state)
ENDPROC extend from;
```

Het is instructief, aan het voorbeeld *abcd* $\Rightarrow$ *dcba*, voor achtereenvolgende waarden van k het aantal elementen van bulk en front te bekijken.

```
Initial state= abcd
Final   state= dcba
        0           1           1 :
abcd
        1           3           2 :
bacd   bcda
        2           6           3 :
acdb   cbda   cdab
        3           11          5 :
bdac   cadb   cdba   dabc   dcab
        4           16          5 :
adbc   cabd   dacb   dbac   dcba
Success!
        189
```

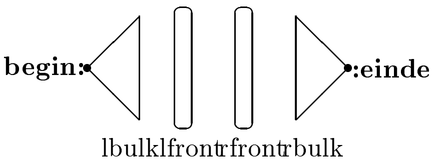
12.14 General problem solver (2)

Waarom werken we bij het zoeken van oplossing eigenlijk van het beginpunt toe naar het eindpunt (**forward reasoning**)? Misschien kunnen we voor sommige puzzels veel beter van het eindpunt toewerken naar het beginpunt, daarbij de spelregels in omgekeerde richting toepassend (**backward reasoning**), bijvoorbeeld wanneer de vertakkingsfactor dan lager uitvalt.

Daartoe hebben we bij de probleem-specificatie ook de inverse nodig van de procedure `from`, n.l.:

```
SET (SIT) PROC to (SIT current state):
  { all predecessors of current state }
```

In zijn algemeenheid heeft geen van beide richtingen de voorkeur, want voor ieder probleem waarvoor de ene richting voordelig is bestaat er ook een waarvoor de andere richting de voorkeur verdient. Daarom ligt het voor de hand een gemengde strategie te kiezen: er wordt zowel vanuit het beginpunt als vanuit het eindpunt een toestandsboom opgebouwd, totdat een gemeenschappelijke tussensituatie is gevonden.



Noem het aantal te onderzoeken situaties uitgaand van de beginsituatie I_B en uitgaand van de eindsituatie I_E. We weten niet a priori welk van de twee het grootst is.

Noem het aantal te onderzoeken situaties wanneer men met de gekozen zoekrichting juist geen geluk heeft I_{ong} en het aantal situaties waar men geen pech heeft I_{gun}. Bij keuze van een van de twee zoekrichtingen geldt dan $I_{ong} = \max(I_B, I_E)$ en $I_{gun} = \min(I_B, I_E)$. Wanneer nu van twee kanten tegelijk wordt gewerkt geldt voor het totaal aantal onderzochte situaties I_{tot}, dat het deel dat uit de beginsituatie wordt afgeleid hoogstens gelijk is aan I_B, terwijl het deel dat uit de eindsituatie wordt afgeleid hoogstens I_E bedraagt. Daarom geldt

$$I_{tot} \leq I_B + I_E \leq 2I_{ong}$$

Dit is een soort garantie: wanneer het zoeken vanuit twee kanten al twee keer zo lang duurt als vanuit één kant, dan hebben degenen die vanuit één kant zochten "toevallig" geluk gehad.

Het is echter niet mogelijk een garantie andersom te geven. Voor ieder willekeurig getal m is een probleem aan te geven, zodat de bewering $I_{gun} < mI_{tot}$ (dus zeker $I_{ong} < mI_{tot}$) wordt tegengesproken. Daarom geven we voorkeur aan het zoeken vanuit twee kanten voor die problemen waar het voldoende eenvoudig is, achterwaarts te werk te gaan.

We kunnen de zaak nog wat anders bekijken: als het zoekproces zich zuiver exponentieel, met een vaste vertakkingsfactor a zou gedragen, dan is de complexiteit van het enkelvoudig zoekproces $O(a^n)$, terwijl deze voor het tweezijdig zoekproces $O(2a^{\frac{n}{2}})$ bedraagt.

Bij een vertakkingsfactor groter dan 2 verwachten we dus een netto tijdwinst (en ruimtewinst) terwijl voor zoekprocessen met een beperkt blijvend front we hoogstens een verslechteringsfactor 2 verwachten.

We zullen niet stoer van beide kanten tegelijk uitgaan maar telkens één van de twee zoekfronten uitbreiden, en wel datgene dat het minste aantal elementen bevat. Bij heel asymmetrische problemen geeft deze heuristiek nog een geringe verbetering.

Overigens is de tweede versie van onze probleemoplosser eenvoudig af te leiden uit de eerste. Vanwege de overzichtelijkheid geven we bij iedere stap aan welk van de twee richtingen wordt gekozen.

Ook hebben we de (nogal triviale) recursie in de procedure `search` vervangen door iteratie.

```
PROC search:
  REP
    report;
    IF is empty(lfront) OR is
empty(rfront)
    THEN line;
         put ("Failure!");
         LEAVE search
    ELIF NOT is empty (lfront I rfront)
    THEN line;
          solved:= true;
          put ("Success!")
    ELIF card (lfront) <= card (rfront)
    THEN report ("left ", lfront);
          make empty (next);
          apply all(lfront, PROC(SIT)
from);
          lfront:= next
    ELSE report ("right", rfront);
          make empty (next);
          apply all(rfront, PROC(SIT)
to);
          rfront:= next
    FI
  UNTIL solved
  ENDREP
ENDPROC search;

PROC report:
  line; put (k);
  k INCR 1;
  put (card (lbulk));
  put (card (lfront));
  put (card (rfront));
  put (card (rbulk));
  put (":")
ENDPROC report;

PROC (SIT) report
   (TEXT direction, SET (SIT) VAR
front):
  put (direction); line;
  put (front); line
ENDPROC report;

PROC extend to (SIT s):
  IF NOT (s IN rbulk)
  THEN
    next INCR next;
    rbulk INCR rbulk
  FI
ENDPROC extend to;

PROC extend (SIT s):
  IF NOT (s IN lbulk)
  THEN
    next INCR next;
    lbulk INCR lbulk
  FI
ENDPROC extend;
```

```
program:
  SIT VAR initial, final;
  SET (SIT) VAR lbulk, lfront, rbulk,
rfront, next;
  initial ASK "Initial state=";
  lbulk:= set (initial);
  lfront:= set (initial);
  final ASK "Final   state=";
  rfront:= set (final);
  rbulk:= set (final);
  BOOL VAR solved :: false;
  INT VAR k :: 0, incount::0;
  search;
  line; put(incount).
```

Losgelaten op het tekstpermutatieprobleem

$$\overbrace{abc\cdots z}^{k\text{ letters}} \Rightarrow \overbrace{z\cdots cba}^{\text{gespiegelde}}$$

leidt het gebruik van `gps2` inderdaad tot een aanmerkelijke verbetering ten opzichte van `gps1`, zoals de volgende tabel aangeeft.

Telkens wordt het aantal elementen van de bulk(en) aangegeven alsmede het aantal keren dat de operator `IN` is toegepast. Dit geeft een beter overzicht dan het meten van uitvoeringstijden, die sterk machineafhankelijk en moeilijk te reproduceren zijn.

	$\lvert$bulk$\rvert$	incount
$abcd \Rightarrow dcba (k = 4)$		
gps1	16	189
gps2	6+6	77
$abcde \Rightarrow edcba (k = 10)$		
gps1	118	11613
gps2	31+31	1581
$abcdef \Rightarrow fedcba (k = 15)$		
gps1		> 30000
gps2	124+82	17204

12.15 Het set-pakket

In eerste instantie hebben we gebruik gemaakt van de simpelste implementering van verzamelingen, namelijk als lijsten (zie 12.3). Zoals aangegeven in 12.5.3 kunnen we verzamelingen ook als zoekbomen representeren, waarbij we hopen dat er een wat grotere efficiëntie bereikt wordt: Omdat vooral de bulk een verzameling met vele honderden elementen kan zijn verwachten we dat een representatie als lijst minder makkelijk te doorzoeken is dan een representatie als boom.

Daarbij schuilt er echter een addertje onder het gras: het ligt niet in de verwachting dat de volgorde waarin de elementen aan de verzameling worden toegevoegd willekeurig zal zijn. Daarvoor is er in ons voorbeeld een te grote verwantschap tussen een bepaalde situatie, voorgesteld als een tekst, en de van daaruit bereikbare situaties. Het is dus mogelijk

		incount			
				heur.	gebal.
	|bulk|	lijsten	bomen	bomen	bomen
$abcd \Rightarrow dcba (k = 4)$					
gps1	16	189	108	99	104
gps2	6+6	77	62	62	61
$abcde \Rightarrow edcba (k = 10)$					
gps1	118	11613	1815	1606	1367
gps2	31+31	1581	668	648	587
$abcdef \Rightarrow fedcba (k = 15)$					
gps1	683	> 30000	16275	14415	11073
gps2	124+82	17204	3522	3304	2840

Figure 12.4: 3 set-representaties vergeleken

dat de boom degradeert tot een lijst, waardoor de zaak alleen nog maar trager wordt.

We zullen daarom drie varianten vergelijken van de implementatie van sets als bomen, en wel:

1. een recht-door-zee implementatie, waarbij geen enkele maatregel wordt getroffen om de optredende bomen te balanceren,
2. een totale herbalancering van de bulk na iedere slag (zoals in 10.5.3), en
3. een heuristische balancering, zoals beschreven in 10.5.4.

Deze drie onderscheiden zich vooral in de operator `INCR`, terwijl we overigens hetzelfde interface aanhouden als bij het vorige set-pakket, inclusief het turven van `incount`.

Tabel 12.4 geeft een vergelijing van de efficiëntie van deze drie paketten. Als maatstaf wordt genomen het aantal keren dat de operator `IN` wordt uitgevoerd bij toepassing op het tekstpermutatie-probleem.

Uit de tabel blijkt dat de keuze voor een boomrepresentatie van de verzamelingen een enorme invloed heeft op de efficiëntie, terwijl het balanceren een kleiner maar merkbaar gunstig effect heeft. Van twee kanten zoeken blijkt echter de grootste verbetering te geven.

12.16 Meer voorbeelden

We geven nog een aantal typische voorbeelden.

12.16.1 Voorbeeld: Kamelen

In een tunnel staan k kamelen tegenover k kamelen, een "p" stelt een kameel voor die naar rechts kijkt en een "q" een kameel die naar links kijkt. Het minteken staat voor de lege plaats.

$$\overbrace{ppp\cdots p}^{k} - \overbrace{q\cdots qqq}^{k}$$

Iedere kameel kan of één plaats vooruit in de richting waarin hij kijkt, of twéé plaatsen (daarbij over een andere kameel springend) mits hij dan op de lege plaats terecht komt.

```
LET SIT = TEXT;

PROC try from (SIT state, TEXT before,
after):
  SIT VAR s :: state;
  change (s, before, after);
  IF s <> state
  THEN extend (s)
  FI
ENDPROC try from;

PROC try to (SIT state, TEXT before,
after):
  SIT VAR s :: state;
  change (s, before, after);
  IF s <> state
  THEN extend to (s)
  FI
ENDPROC try to;

PROC to (SIT state):
  try to (state, "q-", "-q");
  try to (state, "-p", "p-");
  try to (state, "-pp", "pp-");
  try to (state, "-qp", "pq-");
  try to (state, "qp-", "-pq");
  try to (state, "qq-", "-qq")
ENDPROC to;
```

De vertakkingsfactor is in dit voorbeeld hoogstens 6.

```
PROC from (SIT state):
  try from (state, "-q", "q-");
  try from (state, "p-", "-p");
  try from (state, "pp-", "-pp");
  try from (state, "pq-", "-qp");
  try from (state, "-pq", "qp-");
  try from (state, "-qq", "qq-")
ENDPROC from;
```

Een vergelijking van de twee zoekstrategieën en representaties levert het volgende resultaat:

		incount			
				heur.	gebal.
	|bulk|	lijsten	bomen	bomen	bomen
$ppp - qqq \Rightarrow qqq - ppp (k = 15)$					
gps1	72	3164	1046	817	573
gps2	53+32	2859	1163	989	876
$pppp - qqqq \Rightarrow qqqq - pppp (k = 24)$					
gps1	195	21446	3899	3227	1767
gps2	150+77	18997	4488	3805	2943

Merk op dat de twee GPS'en hier elkaar niet veel ontlopen. Een nadere beschouwing van het verloop van `front` toont aan dat dit komt omdat voorbij de helft van het zoekproces de fronten weer gaan krimpen. Het effect van balancering van de bulk blijkt uitgesproken sterk te zijn.

12.16.2 Voorbeeld: Kruiken

Als laatste voorbeeld behandelen we het probleem van de n *kruiken*.

We beschikken over n kruiken (engels: "jugs"), geheel of gedeeltelijk gevuld met water, ieder met een bepaalde maximale inhoud (een geheel getal). De toestand kunnen we karakteriseren door n gehele getallen (ieder niet groter dan de corresponderende maximuminhoud). In de begintoestand zijn alle kruiken vol.

Telkens kunnen we nu één van de volgende handelingen verrichten:

- een kruik geheel leeg gieten
- vanuit een kruik die voldoende water bevat een andere kruik volgieten
- de hele inhoud van een kruik overgieten in een andere kruik die daartoe voldoende plaats heeft

totdat een bepaalde eindtoestand is bereikt.

In tegenstelling tot de vorige voorbeelden, waar de toestand als een tekst werd gerepresenteerd, stellen we de toestand deze keer voor als een rij van gehele getallen (de momentane inhoud van de verschillende kruiken), waarbij we het aantal kruiken gelijk aan drie kiezen.

```
LET SIT = ROW njugs INT;
```

```
LET njugs = 3;
```

Vanwege de ietwat exotische representatie moeten we zelf de vergelijkings operatoren definieren die het set-pakket verlangt.

```
BOOL OP = (SIT a, b):
  INT VAR i;
  FOR i UPTO njugs
  REP
    IF a [i] <> b [i]
    THEN
      LEAVE = WITH false
    FI
  ENDREP;
  true
ENDOP =;

BOOL OP < (SIT a, b):
  INT VAR i;
  FOR i UPTO njugs
  REP
    IF a [i] < b [i]
    THEN
      LEAVE < WITH true
    ELIF a [i] > b [i]
    THEN
      LEAVE < WITH false
    FI
  ENDREP;
  false
ENDOP <;
```

Ook zien we ons verplicht, zelf aan te geven hoe een toestand moet worden ingelezen en afgedrukt.

```
PROC get (SIT VAR jug):
  INT VAR i;
  FOR i UPTO njugs
  REP get (jug [i])
  ENDREP
ENDPROC get;

PROC put (SIT jug):
  INT VAR i;
  put (" (");
  FOR i UPTO njugs
  REP put (text (jug [i], 2))
  ENDREP;
  put (")")
ENDPROC put;

PROC from (SIT state):
  INT VAR i, j;
  FOR i UPTO njugs
  REP
    IF state [i] > 0
    THEN
      FOR j UPTO njugs
      REP pour (state, i, j)
      ENDREP
    FI
  ENDREP
ENDPROC from;
```

We proberen alle mogelijke combinaties van kruiknummers i en j, waarbij we $i = j$ laten overeenkomen met het geval dat kruik i wordt leeggegoten.

```
PROC pour (SIT s, INT i, j):
  IF i = j
  THEN
    SIT VAR u :: s;
    u [i]:= 0;
    extend (u)
  ELIF s [j] = jugs [j]
  THEN
  ELSE
    SIT VAR t :: s;
    INT CONST d :: min (s [i], jugs [j]
- s [j]);
    t [i] DECR d;
    t [j] INCR d;
    extend (t)
  FI
ENDPROC pour;
```

Bij wijze van heuristiek zouden we die situaties buiten beschouwing kunnen laten waarbij de kruiken tezamen niet meer genoeg water bevatten om de doelsituatie te bereiken. We laten dit over aan de lezer.

```
jugs problem:
  ROW njugs INT CONST jugs :: [12,7,5];
  program.
```

		incount			
				heur.	gebal.
	|bulk|	lijsten	bomen	bomen	bomen
5 5 5⇒*5 3 1* (faalt bij k=14)					
gps1	242	>30000	10865	10342	9217
5 5 5⇒*0 1 2* (k=8)					
gps1	223	>30000	8788	8443	7541

Weer blijkt de (gebalanceerde) boom een veel betere representatie voor verzamelingen dan de lineaire lijst.

De in dit hoofdstuk behandelde technieken bieden ruime gelegenheid tot experimenteren en verbeteren. Zij dienen echter vooral om de kracht te laten zien van abstracte datatypen met polymorfe typering in een algoritmische programmeertaal.

12.17 Opgaven

1. (rapportage 1) Wijzig de eenzijdige probleemoplosser zodanig, dat bij succes precies die toestanden worden afgedrukt via welke het optimale pad verloopt.
2. (rapportage 2) Wijzig de tweezijdige probleemoplosser zodanig dat bij succes precies die toestanden worden afgedrukt via welke het optimale pad verloopt (hint: los daartoe recursief het deelprobleem $\langle BEGIN, MIDDEN\rangle$ en het deelprobleem $\langle MIDDEN, EIND\rangle$).
3. (spelen) Analyseer de problematiek van 2-persoons spelen, waarbij de spelers om beurten een zet doen. Ontwerp daarvoor een Algemene Speler. Pas hem toe op problemen als Boter, Kaas en Eieren. Laat hem ook tegen zichzelf spelen.
4. (bewijzen) Probleem D uit het begin van dit hoofdstuk. De transformatieregels voor 1e orde predicatencalculus zijn in ieder logica leerboek te vinden.

Bibliography

[AHU83] A.V.Aho, J.E. Hopcroft, J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley Publishing Company, 1983.
Een standaardwerk over datastructuren.

[BAY81] M. Bayer, B. Böhringer, J.-P. Dehottay, H. Feuerhahn, J. Jasper, C.H.A. Koster, U. Schmiedecke, *Software development in the CDL2 laboratory*, in: [HUN81].
Bevat onder andere een (summiere) beschrijving van de systeemimplementeringstaal CDL2, waarin de auteur zijn opvattingen over modulariteit heeft beschreven.

[CHU60] R.V. Churchill, *Complex variables and applications*, Mc Graw-Hill 1960.

[CRA87] D. Craemer, *Abstrakte Maschinen und modulares Programmieren in ELAN*, GMD ITF Bericht, Bonn 1987.

[vDALE] van Dale, *Groot woordenboek der Nederlandse taal*, Elfde herziene druk, 1984.

[DVB85] R. Danckwerts, D. Vogel, K. Bovermann, *Elementäre Methoden der Kombinatorik, mit Programmbeispielen in Elan*, Teubner Verlag, Stuutgart 1985.
Een schoolboek met combinatorische voorbeelden in Elan.

[DEF77] E. Denert, R. Franck, *Datenstrukturen*, Bibliographisches Institut (Reihe Informatik Bd. 22), Mannheim 1977.

[DIF84] E.W. Dijkstra, W.H.J. Feijen, *Een methode van programmeren*, Academic Service, 1984.
Een collegedictaat voor eerstejaars studenten over dezelfde stof als [DIJ76].

[DIJ59] E.W. Dijkstra, *A note on two problems in connection with graphs*, Numerische Mathematik Vol. 1, 1959.

[DIJ76] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.
Een fundamentele en rijkelijk toegelichte behandeling van het afleiden van sequentiële algoritmen vanuit een doelrelatie.

[Eumel] R. Hahn, D. Heinrichs, P. Heyderhoff, J. Liedtke, *EUMEL Benutzerhandbuch Version 1.7*, Universität Bielefeld / Gesellschaft für Mathematik und Datenverarbeitung, Bonn -St. Augustin, 1984.
Handboek bij het in Duitsland op vele plaatsen gebruikte Extendable multi User Microprocessor ELAN-system.

[FAI85] R. Fairley, *Software Engineering Concepts*, McGraw-Hill, 1985.
Leerboek Programmatuurkunde.

[FLO67] R.W. Floyd, *Assigning Meanings to Programs*, Proceedings Symposium on Applied Mathematics, Volume 19 American Mathematical Society, Providence, Rhode Island, 1967.
Het klassieke artikel waarin de “inductive assertions” worden geïntroduceerd.

[GAN75] J.D. Gannon, *Language Design to Enhance Programming Reliability*, Technical Report CSRG-47 Computer Systems Research Group, University of Toronto, 1975.
Een gedetailleerde analyse over de invloed van een welgekozen verzameling wijzigingen aangebracht in de (overigens oninteressante) taal TOPSI op het foutgedrag van zijn gebruikers.

[GER76] S.L. Gerhart, L. Yelowitz, *Observations on Fallability in Applications of Modern Programming Methodologies*, IEEE transactions on Software Engineering Volume 2, Number 3, September 1976.
Een commentaar op fouten, gevonden in door vaak gerenommeerde auteurs gepubliceerde algoritmen of programma’s.

[GRPH] I.S. Grant, W.R. Phillips, *Electromagnetism*, John Wiley publ. cy.
Een solide leerboek Electrotechniek.

[HOA69] C.A.R. Hoare, *An Axiomatic Basis of Computer Programming*, Communications of the ACM, Volume 12, October 1969.
Behelst een axiomatische aanpak van bewijsregels, mede in relatie tot de semantische definitie van programmeertalen.

[HEA79] G. Hommel, J. Jäckel, S. Jähnichen, K. Kleine, W. Koch, C.H.A. Koster, *ELAN Sprachbeschreibung*, Akademische Verlagsgesellschaft, Wiesbaden 1979.
De officiële definitie van de taal Elan.

[HUN81] H. Hünke, (ed.): *Software Engineering Environments*, North Holland Publ. Cy., 1981.

[KLI85] L.H. Klingen, J. Liedtke, *Elan in 100 Beispielen*, Teubner Verlag, Stuttgart 1985.

Zeer inspirerende bron van voorbeelden voor het Informatica-onderwijs aan middelbare scholen.

[KME79] C.H.A. Koster, H. Meijer, *Systematisch Programmeren in Algol 68, Deel 2: Recursieve Algorithmen en Datastructuren*, Kluwer, Deventer 1979. De voorloper van dit boek.

[KNU69] D.E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley Publishing Company, 1969.
Hoofdstuk 1 behandelt (vrijwel) alle mathematische middelen, nodig voor de bepaling van de complexiteit van algoritmen.

[KNU74] D.E. Knuth, *Structured Programming with Goto Statements*, Computing Surveys, Volume 6, Number 4, 1974.
Over de morele verwerpelijkheid en materiele noodzaak van het gebruik van sprongen zijn heel wat vlammende artikelen verschenen; een conclusie uit deze discussie is te vinden in dit artikel van Knuth.

[KNU81] D.E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, tweede uitgave, Addison-Wesley Publishing Company, 1981.
Een bron voor vele abstracte typen met een wiskundig karakter.

[KOS84] C.H.A. Koster, *Systematisch Leren Programmeren, deel 1: Top-Down programmering*, Educaboek, Culemborg 1984; Academic Service, Rijswijk 1988.
Het eerste deel van deze reeks.

[KZO78] C.H.A. Koster, T.A. Zoethout, *Systematisch Programmeren in Algol 68, Deel1: Inleiding in de Informatica*, Kluwer, Deventer 1978.

[LIS74] B. Liskov, S. Zilles, *Programming with Abstract Data Types*, SIGPLAN Notices Volume 9, Number 4, pp 50-59, April 1974.
Een van de klassieke artikelen over relevantie en toepasbaarheid van een scheiding van eigenschappen en representatie van objecten.

[MEE62] L. Meertens, *Kunstmatige intelligentie: een programma voor het optimaal oplossen van een klasse van problemen*, verslag van de lezing van de Heer L.G.L.Th. Meertens, gehouden op 19-10-62.
Een kopie van dit verslag is verkrijgbaar bij de auteur van dit leerboek.

[MEU75] S.G. van der Meulen, C.H. Lindsey, *Informal Introduction to Algol 68, Revised Version*, North-Holland Publishing Company, Amsterdam, 1975.

[MIC86] D. Michie, *On Machine Intelligence*, Ellis Horwood Series in Artificial Intelligence, Second Edition, 1986.
Een (nogal persoonlijk) overzicht over de ontwikkeling van de AI.

[MIL79] R.A. De Millo, R.J. Lipton, A.J. Perlis, *Social Processes and Proofs of Theorems and Programs*, Communications of the ACM, Volume 22, Number 5, May 1979.
Een zeer leesbaar artikel waarin twijfel werd uitgesproken over de rol van formele verificatie in de praktijk; het artikel heeft een brede stroom reacties (voor en tegen) opgeroepen.

[NSS59] A. Newell, J.C. Shaw, H.A. Simon, *(Report on) A General Problem Solving Program for a Computer*, Computers and Automation, July 1959.

[OTT85] *Analysis mit dem Computer*, Teubner Verlag, Stuttgart, 1985.
Invoering in de analyse met gebruikmaking van Elan.

[PAR90] H.A. Partsch, *Specification and Transformation of Programs, a formal approach to software development*, Springer-Verlag, 1990.
Een leerboek voor gevorderden over transformationele programmering.

[PEP84] P. Pepper (ed.), *Program Transformation and Programming Environments*, NATO ASI Series F: Computer and Systems Sciences Vol. 8, Springer-Verlag, 1984.
Een zeer gevarieerde verzameling artikelen over Programmatuurkunde, waarbij ook de levendige discussie is vastgelegd.

[SLA71] J.R. Slagle, *Artificial Intelligence, The Heuristic Programming Approach*, McGraw Hill Series in Systems Science, 1971.

[SOM89] I. Sommerville, Software Engineering, 3rd Edition, Addison-Wesley, 1989.
Klassiek leerboek Programmatuurkunde.

[TvdW90] Th.P. van de Weide, *Van Datastructuur tot Informatiesysteem*, Academic Service, 1991.
Vervolg op dit boek, derde en laatste deel van deze reeks.

[VERV92] W.A. Vervoort, Een hink-stap-sprong flowchart techniek, Memoranda Informatica 92-38, Dept. of Computer Science, Universiteit van Twente, Mei 1992. Een pleidooi voor het gebruik van diagrammen voor het afleiden en vastleggen van herhalings-invarianten.

[VRI48] L. De Vries, *Het jongens radioboek*, deel 2, 1948.
De Vries heeft een hele generatie van

jongens (en meisjes!) met de techniek vertrouwd gemaakt.

[WEBST] Webster's New Collegiate Dictionary, 1980.

[WEI71] G.M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.
Bevat veel interessants over de folklore van het fouten maken en de vreemde denkprocessen van programmeurs.

Index