Voorwoord

Dit boek is een inleiding in de theorie van formele talen en de ontleding van zinnen uit die talen, alsook een inleiding in de techniek van de computerbouw. Deze thema's zijn om twee redenen van belang.
Ten eerste wordt het inzicht in het juiste gebruik van programmeertalen door kennis van de grondbeginselen van een compiler verdiept.
Ten tweede is het beheersen van deze thema's een voorwaarde voor het vakkundig kunnen construeren van computersystemen die op een eenvoudige instructietaal zijn gebaseerd. Het aantal van dergelijke toepassingen neemt snel toe op de terreinen van procesbesturing, gegevensverwerking en bedrijfssystemen.

Waar het de theorie van formele talen en de syntactische analyge betreft beperken wij ons tot het voor de bouw van een eenvoudige compiler minimaal noodzakelijke. Het systematisch ontwikkelen van een volledige computer voor een eenvoudige programmeertaal wordt daarentegen in alle details besproken.
De keuze van de 'brontaal' PL/0 is een compromis tussen een taal die te triviaal is om als leerzaam voorbeeld te kunnen dienen en een taal waarvan de kracht en complexiteit de werkelijke kern en de basisprincipes van de compilerbouw versluieren.
Voor alle programma's wordt de taal Pascal gebruikt. Deze taal maakt het mogelijk complexe programma's en gegevensstructuren helder voor te stellen en is derhalve voor onze thema's bij uitstek geschikt.
Dit boek is ontstaan uit een college aan de ETH Zürich. Het is een grondig bewerkte vertaling uit het Engels. De hoofdstukken 12 en 13 zijn daarbij aan de oorspronkelijke tekst toegevoegd. De fotografisch reproduceerbare (Duitse) tekst is door de auteur zelf geproduceerd. Dit was alleen mogelijk door gebruik van de modernste computersystemen voor het maken en verwerken van tekst en illustraties. Ik dank de firma Rank Xerox die deze apparatuur ter beschikking heeft gesteld.

Palo Alto, december 1976 N. Wirth

Voorwoord bij de derde druk

Het belangrijkste verschil tussen deze herziene druk en de vorige is dat voor de formulering van de in dit boek behandelde algoritmen niet langer gebruik wordt gemaakt van de programmeertaal Pascal, maar van Modula-2. Het in deze taal beschikbare moduleconcept blijkt een ideaal middel om de verschillende delen van een compilersysteem ook thematisch te scheiden.
De fotografisch reproduceerbare (Duitse) tekst werd op de computer Lilith met het tekstverwerkingspakket Lara gemaakt. Dit paar heeft niet alleen de taak aanzienlijk vereenvoudigd, maar maakte deze ook tot een genoegen. Ik dank daarom in het bijzonder de makers ervan, J. Gutknecht en H. Schaer.

Zürich, maart 1984 N.W.

Voorwoord bij de vierde druk

In deze vierde druk zijn alleen enkele schrijf- en programmeerfouten verbeterd. Ik dank allen die aan het vinden daarvan hebben bijgedragen.

Zürich, januari l986 N. W.

Inleiding

In dit boek wordt een eenvoudige, rudimentaire programmeertaal en een daarbij behorende compiler ontwikkeld. Aan de hand van dit weliswaar eenvoudige, maar niet triviale voorbeeld willen we laten zien hoe een goed gestructureerd programma systematisch wordt ontwikkeld.

Het belangrijkste doel daarvan is een inleiding in de techniek van de compiler bouw te geven. Kennis van en inzicht in de techniek van de compilerbouw zijn nuttig bij het programmeren in hogere programmeertalen.

Bovendien zijn deze vaardigheden nodig voor het construeren van invoertalen en systemen voor specifieke doelen. Omdat compilerbouw een ingewikkeld onderwerp is beperken we ons op dit terrein tot een inleiding. Wellicht is het meest fundamentele inzicht bij het construeren van compilers wel dit: de grammaticale structuur van een taal moet in de structuur van een compiler voor die taal worden weerspiegeld. Daaruit volgt onmiddellijk dat de complexiteit -of liever de eenvoud- van een taal de maatstaf is voor de complexiteit van een compiler voor die taal.

We behandelen daarom eerst taalstructuren en de formele beschrijving van die structuren in het algemeen. Daarna concentreren we ons uitsluitend op eenvoudige structuurelementen, die met een eenvoudige, efficiënte en overzichtelijke compiler zijn te behandelen. De ervaring heeft geleerd dat structureel eenvoudige elementen over het algemeen voldoende zijn
om aan bijna alle echte eisen die aan een programmeertaal worden gesteld te voldoen.

De laatste twee hoofdstukken zijn aan praktische problemen rond het construeren van een compíler gewijd. Aan de hand van een case-study wordt getoond hoe een bestaande taal en de bij die taal behorende compiler kunnen worden uitgebreid. Deze in de praktijk veel voorkomende situatie wordt door een systematische structuur van het uit te breiden systeem wezenlijk gemakkelijker gemaakt. Tenslotte worden verschillende methoden om een compiler geschikt te maken voor een andere computer uiteengezet. Al deze methoden zijn gebaseerd op de techniek van bootstrapping en veronderstellen dat de compiler in de 'te compileren taal' geschreven is.

Compilerbouw

N. Wirth

ISBN 906233234, Academic Service, 1986

Dit boek is al jaren out of print. Het wordt door sommige landelijke oplichters nog als 'leverbaar' aangeprezen. Maar je kan het nergens kopen. Soms wel lenen of huren. Daarom heb ik een boek geleend en het gescand. Dit boek is te goed om in een stoffig hoekje te laten liggen.
Ik hoop dat het nog eens opnieuw in een oplage gedrukt wordt want dan koop ik het eerste exemplaar. Tot die tijd zullen we het met deze copie moeten doen.

1: Definitie en structuur van formele talen

Elke taal is gebaseerd op een vocabulaire. Gewoonlijk worden de elementen van dat vocabulaire aangeduid als woorden. Bij formele talen is het daarentegen gebruikelijk deze elementen symbolen te noemen.
Elke taal kent reeksen symbolen die juist of goedgevormd zijn en andere die onjuist of niet goedgevormd zijn. In eerste instante bepaalt de grammatíca of syntaxis van een taal tot welke van deze twee categorieën een bepaalde reeks symbolen behoort.
We gaan hier zelfs zover een taal te definiëren als de verzameling van reeksen symbolen die op grond van de syntaxis van die taal als goedgevormd moeten worden beschouwd.
Niet goedgevormde reeksen behoren helemaal niet tot de taal, ook al bestaan deze reeksen uit symbolen die allemaal tot het vocabulaire van de taal behoren.

De eerste functie van de syntaxis is dus de beschrijving van de verzameling van die reeksen symbolen die tot de taal behoren; we noemen die reeksen symbolen zinnen.
De tweede, niet minder belangrijke, functie van de syntaxis van een taal is het definieren van een zinsstructuur. De structuur van een zin speelt een centrale rol bij de herkenning van de betekenis van die zin.
Dubbelzinnigheden in de betekenis van een zin berusten meestal op het feit dat aan dezelfde reeks woorden verschillende juiste zinsstructuren kunnen worden toegekend. Structuur (syntaxis) en betekenis (semantiek) zijn nauw met elkaar verbonden.
Syntaxis is uiteindelijk altijd middel tot een hoger doel, namelijk het coderen van een bepaalde betekenis. Dit gegeven belet ons echter niet eerst alleen de syntactische aspekten van talen te behandelen en de problemen rond betekenis en interpretatie te negeren.
Als inleiding bekijken we de zin '**katten slapen**'. Het woord 'katten' is het subject (onderwerp) van de zin, 'slapen' het predicaat. De zin behoort tot een taal die bijvoorbeeld door de volgende syntaxis is gedefinieerd:

```
zin       = subject predicaat
subject   = katten | honden
predicaat = eten | slapen
```

Deze drie regels hebben de volgende betekenis:

1. Een zin bestaat uit een subjekt gevolgd door een predicaat.
2. Een subject bestaat uit het woord katten of het woord honden.
3. Een predicaat bestaat uit het woord eten of het woord slapen.

Het centrale idee is dat elke (goedgevormde) zin kan worden afgeleid uit het startsymbool (zin) door herhaalde toepassing van substitutieregels.
Als notatie gebruiken we van nu af een variant op de zogenaamde Backus Naur Form (BNF), die voor het eerst werd toegepast bij de definitie van de taal Algol 60.

De begrippen zin, subject, en predicaat noemen we niet-terminale symbolen. De woorden die in de eigenlijke zinnen voorkomen noemen we terminale symbolen of eindsymbolen. De substitutieregels noemen we produkties. De tekens '=', '|' en '.' (punt) zijn zogenaamde metasymbolen van de BNF notatie. Als naam voor niet-terminale symbolen kiezen we zinvolle woorden die aangeven welke betekenis de bij die symbolen behorende structuur heeft. Als we in bovenstaand voorbeeld in plaats van beschrijvende woorden letters gebruiken -zoals in de wiskunde gebruikelijk is- dan is de hierna volgende, formeel beschreven syntaxis het resultaat. Om het verschil tussen terminale en niet-terminale symbolen duidelijker te laten uitkomen

gebruiken we voor niet-terminale symbolen hoofdletters, voor terminale kleine letters.

Voorbeeld 1

```
S = AB.
A = x | y.
B = z | w.
```

De zo gedefinieerde taal bestaat uit vier zinnen: xz, yz, xw, yw.

De tot nu toe informeel ingevoerde concepten gaan we nu door formele definities precies vastleggen. We duiden afzonderlijke symbolen aan met Latijnse, reeksen symbolen met Griekse letters.

1. Een taal L wordt gekarakteriseerd door een vier-tupel L(T, N, P, S).

   Het vocabulaire T van terminale symbolen.
   De verzameling N van niet-terminale symbolen.
   De verzameling P van substitutieregels, produkties genaamd.
   Het startsymbool S, een element van N.

2. De taal L(T, N, P, S) is de verzameling van reeksen terminale symbolen die met behulp van de hierna volgende regels (zie 3.) uit het startsymbool S kunnen worden afgeleid.

   $$L = \{ \xi \mid S \dashrightarrow \xi \wedge \xi \in T^* \}$$

   T* staat voor de verzameling van symboolreeksen die uit het vocabulaire T kunnen worden opgebouwd,.

3. Een reeks Sn kan uit een reeks S0 worden afgeleid dan en slechts dan als er reeksen s1, s2, ..., s,n-1 zijn, zodanig dat elke Si direct uit de voorafgaande Si-1 afgeleid kan worden (zie 4). We schrijven:

   $$\sigma_0 \xrightarrow{*} \sigma_n \equiv \sigma_{i-1} \rightarrow \sigma_i \quad \text{voor } i = 1 \ldots n$$

4. Een reeks η kan direkt afgeleid worden uit een reeks ξ dan en slechts dan als η en ξ zijn voor te stellen als de reeksen $\eta=\alpha\eta'\beta$ en $\xi=\alpha\xi'\beta$ en als $\eta'=\xi'$ een produktie in de verzameling P is.
   We zeggen dan dat de produktie wordt toegepast in de context van α en β.

   Let erop dat de notatie $\alpha=\beta1|\beta2|\beta3...\beta n$, alleen maar een afkorting is van de produkties $\alpha=\beta1$, $\alpha=\beta2$, $\alpha=\beta3$ ... $\alpha=\beta n$.

Na deze uiteenzetting kunnen we als voorbeeld de zin xz uit voorbeeld 1 in de volgende reeks stappen afleiden:

S -> AB -> xB -> xz

Hieruit volgt: S -> xz. En omdat xz behoort tot T* is xz een element van de taal d.w.z. xz behoort tot L.

Let erop dat de niet terminale symbolen A en B alleen in tussenstappen voorkomen; de laatste stap in een produktie leidt altijd tot een reeks die slechts terminale symbolen bevat. De syntactische regels worden produkties genoemd omdat zij bepalen hoe een zin kan worden geproduceerd.
Een taal noemen we (in navolging van N. Chomsky) contextvrij dan (en slechts dan) als deze door een verzameling van contextvrije produkties kan worden gedefinieerd. Een produktie is contextvrij dan (en slechts dan) als deze de volgende vorm heeft:

$$A = \sigma. \quad (A \in N, \sigma \in (N \cup T)^*)$$

De linkerkant van de produktie bestaat dus uit slechts één symbool. Dit betekent niet minder dan dat A door σ mag worden vervangen, ongeacht de context waarin A voorkomt. Als een produktie de vorm

$$\alpha A\beta = \alpha\sigma\beta.$$

heeft noemen we deze produktie contextafhankelijk. Deze produktie vertelt ons immers dat A slechts door σ mag worden vervangen als A in de omgeving van α en ß voorkomt. We zullen ons in dit boek uitsluitend met contextvrije talen bezighouden.

Het tweede voorbeeld van een syntaxis laat zien hoe door toepassing van recursie een oneindige verzameling van zinnen kan worden beschreven door een eindige verzameling produkties.

Voorbeeld 2:

```
S = xA.
A = z|yA.
```

Uit dit startsymbool zijn de volgende zinnen af te leiden:

```
xz, xyz, xyyz , xyyyz, xyyyyz, xyyyyyz ,...
```

Dit voorbeeld laat ons zien hoe door toepassing van een recursieve definitie een willekeurig aantal herhalingen kan worden verkregen. De recursivitejt is wellicht niet onmiddellijk te zien. Laten we om een herhaling zichtbaar te maken daarom de volgende notatie invoeren: $\{\sigma\}$ betekent dat de reeks σ een willekeurig aantal keren (waaronder 0 keer) mag worden herhaald. Als we de lege reeks aanduiden met de letter ε, dan betekent $\{\varepsilon\}$ hetzelfde als $\varepsilon|s|ss|sss|...$ en zijn de twee produkties uit voorbeeld 2 als volgt in één produktie samen te vatten

```
S = x{y}z.
```

waarbij het symbool A niet langer nodig is. Eveneens ter vereenvoudiging voeren we de notatie $[\sigma]$ in. $[\sigma]$ betekent hetzelfde als $\sigma|\varepsilon$. Bekijk als voorbeeld van het gebruik van $[\sigma]$ de produktie

```
S = x[y]z.
```

Deze betekent dat uit S de reeksen xyz en xz zijn af te leiden. Het ligt nu voor de hand ook uitdrukkingen tussen haken toe te staan. In plaats van xyz|xwz kunnen we dan bijvoorbeeld x(y|w)z schrijven. Denk eraan dat de verschillende soorten haakjes, {} [] (), metasymbolen zijn, net als de tekens =, | en *. Deze behoren dus tot een uitgebreide metataal, EBNF (E = extended), en niet tot de objecttaal waarvan de syntaxis wordt gedefinieerd.

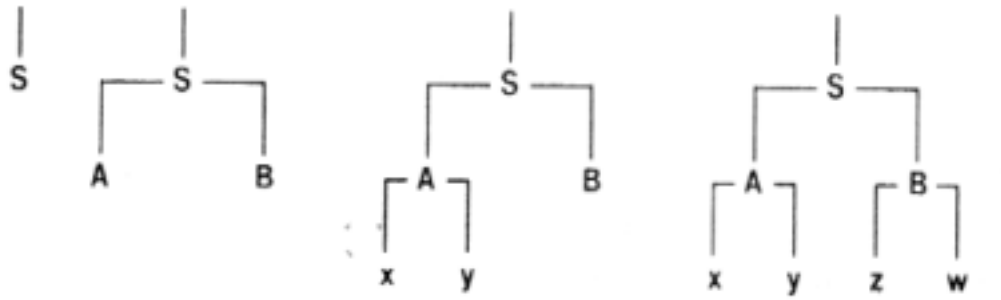
2 Zinsanalyse

Het is gewoonlijk niet de taak van een vertaler of compiler om een zin te produceren of af te leiden, maar om een zin en zijn structuur te herkennen. Dit betekent dat bij het lezen van een zin de stappen waarlangs de zin is geproduceerd opnieuw moeten worden doorlopen, nu als stappen op weg naar het herkennen van de zin. Dit is over het algemeen een ingewikkelde en in veel gevallen zelfs een onmogelijke opgave.
De complexiteit van die opgave wordt direct bepaald door de complexiteit van de produkties die de syntaxis bepalen. Er zijn talrijke herkenningsalgoritmen bekend voor verschillende klassen van talen met een verschillende structurele complexiteit.
De effectiviteit van zo'n algoritme is direct afhankelijk van de kracht ervan: naarmate een algoritme algemener toepasbaar is, zal het minder efficiënt zijn.
Wij willen hier echter alleen maar een methode ontwikkelen waarmee relatief eenvoudig talen zijn te herkennen; dit moet dan wel zo efficiënt gebeuren, dat er in de praktijk mee valt te werken.
Deze laatste eis betekent dat de tijd die nodig is om een zin te herkennen hoogstens proportioneel mag toenemen met de lengte van de te herkennen zin. Wij proberen dus niet een zo algemeen mogelijk herkenningsalgoritme -een zogenaamde ontleder of parser- te vinden maar werken vanaf de andere kant:
we poneren een eenvoudig algoritme en bepalen aan de hand van dat algoritme de klasse van talen die ermee verwerkt kan worden.
Een eerste gevolg van deze manier van werken is dat elke stap in de ontleding van een zin slechts mag worden bepaald door de stand van de analyse op dat moment en één enkel als volgende te lezen symbool.
Bovendien eisen we dat geen enkele in de analyse gemaakte stap achteraf ongedaan mag worden gemaakt (one symbol lookahead, no backtracking).

De methode die hierna beschreven wordt berust op het zogenaamde top-down principe. Het is nuttig de structuur van een zin als een boom te tekenen. Bovenaan staat het startsymbool, aan de basis van de boom staat de te herkennen zin. Ter illustratie geven we het volgende voorbeeld:

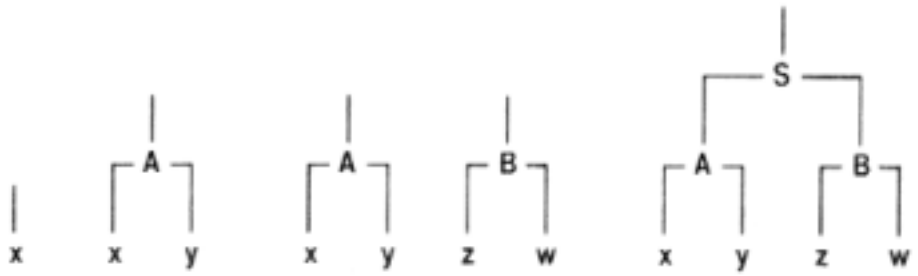
```
S = AB.
A = xy.
B = zw.
```

De stappen die de parser bij het (van links naar rechts) lezen van een zin doorloopt staan hieronder getekend; vertrekkend vanuit S wordt een structuur opgebouwd die aan de basis overeenkomt mee de te lezen zin. De boom groeit dus van boven naar beneden; vandaar de uitdrukking 'top-down'.



Bij het zogenaamde bottom-up principe gaan we prcecies omgekeerd te werk, We gaan nu niet 'speculatief' uit van een startsymbool om daaruit een reeks te produceren die met de gelezen tekst overeenkomt; nu wordt de gelezen tekst gereduceerd tot uiteindelijk de gehele tekst tot het startsymbool is teruggevoerd. In

de grafische voorstelling van dit proces groeit nu de 'structuurboom' van onder naar boven, vandaar de aanduiding 'bottom-up'.



We zullen nu aan de hand van voorbeeld 1 laten zien hoe een eenvoudige 'top-down' parser werkt. We nemen aan dat de parser de zin xw moet herkennen. De zin xw behoort alleen dan tot de taal als deze uit het startsymbool S is af te leiden. Uit S is echter alleen de reeks AB direct af te leiden. We vervangen S daarom in ieder geval door AB. Nu moet onderzocht worden of het begin yan de zin xw uit AB af te leiden is. De produktie A = x vertelt ons dat dit inderdaad mogelijk is; we kunnen x daarom, samen met A, als verwerkt doorstrepen. Nu moeten we nog aantonen dat w uit B af te leiden is.
Bij het doorzoeken van de syntaxis zien we dat de produktie B = z niet kan worden toegepast maar dat de produktie B = w wel tot ons doel leidt. We zullen nu de afzonderlijke stappen opschrijven. Daarbij plaatsen we links die symboolreeksen, uit welke de nog overgebleven rest van de te herkennen zin af te leiden moet zijn, rechts de nog te herkennen rest van de zin.

```
S     xw
AB    xw
zB    xw
B     w
W     W
```

Let erop dat in dit voorbeeld elke stap eenduidig wordt bepaald door het nu eerst na te streven doel (bijv. het verwerken van A) en het volgende te lezen symbool (bijv. x). Dit geldt helaas niet in alle gevallen, zoals het volgende voorbeeld van een syntaxis laat zien.

Voorbeeld 3

```
S =  A|B.
A = xA|y.
B = xB|z.
```

We proberen de (tot de taal behorende) zin xxz te herkennen.

```
S     xxz
d     xxz
zd    xxz
d     xz
zd    xz
A     z
```

Op deze plaats blijkt dat het onmogelijk is z uit A af te leiden. Als we in de eerste stap S niet door A maar door B hadden vervangen was het doel te bereiken geweest.

Helaas is de beslissing of we S door A of B vervangen in dit geval niet uit het te lezen symbool x alleen af te leiden. Alleen door vooruit te kijken en het symbool z te herkennen zou de juiste beslissing kunnen zijn genomen.
Het is echter gemakkelijk in te zien dat bij dìt voorbeeld van een syntaxis geen vaste grenzen voor het aantal vooruit te bekijken symbolen kan worden gegeven.
Immers, elk algoritme dat in staat zou zijn n symbolen vooruit te kijken zou door een reeks van n symbolen x, gevolgd door een z (of een y) weer falen.
We zullen daarom geen ingewikkeld herkenningsalgoritme invoeren, maar een regel poneren die de syntaxis zo inperkt dat bovenstaand geval wordt uitgesloten. We nemen aan dat steeds alle produkties met hetzelfde linkerlid in één enkele regel samengevat zijn.

Definitie 1:

We duiden de verzameling van alle symbolen die aan het begin van een uit de reeks R afleidbare reeks kunnen staan aan met first(R).

first (R) = {s | R -> s R' }

Regel 1

We eisen bij elke produktie

```
A = R1 | R2 | R3| ... | Rn.
```

dat de verzamelingen van beginsymbolen van alle termen Ri disjunct zijn.

```
first(Ri) + first(Rj) is niet leeg, voor elke i # j
```

In de syntaxis van voorbeeld 3 geldt: x behoort tot first (A) en x behoort tot first (B). De produktie S = A|B voldoet dus niet aan regel 1. Nu kan deze moeilijkheid in dit voorbeeld gemakkelijk worden omzeild door een equivalente syntaxis te definiëren die wel aan regel 1 voldoet:

```
S = C|xS.
C = y|z.
```

Helaas is regel 1 nog niet voldoende om alle problemen uit te sluiten; we laten dit in het volgende voorbeeld zien.

Voorbeeld 4

```
S = Ax.
A = [x].
```

Als we nu proberen de zin x te herkennen lopen we opnieuw vast:

```
S     x
Ax    x
xx    x
x     epsilon
```

Oorzaak van het probleem is hier de keuze van de produktie A = x in plaats van de produktie A = epsilon. Deze situatie kan alleen ontstaan als uit een symbool de lege reeks kan worden afgeleid. Let erop dat dit steeds dan het geval is als een produktie

de vorm A = {s} of de vorm A = [s] heeft. Om het in voorbeeld 4 aangeduide probleem uit te sluiten voeren we nog een beperkende regel in.

Definitie 2

De verzameling van alle symbolen die direct op een uit R afgeleide reeks kunnen volgen -de vervolgsymbolen van R- duiden we aan met follow (R).

Regel 2

Voor elk niet-terminaal symbool A waaruit de lege reeks kan worden afgeleid geldt dat de verzameling van beginsymbolen van die reeks en de verzameling van vervolgsymbolen van die reeks disjunct moeten zijn; met andere woorden, de doorsnede van de verzameling van beginsymbolen van A en de verzameling van vervolgsymbolen van A moet in dit geval leeg zijn.

In voorbeeld 4 voldoet het symbool A niet aan regel 2, omdat geldt;

$$first(A) = follow(A) = \{x\} \neq \emptyset$$

Let er ook op dat elke linksrecursieve produktie van de vorm

$$A = A\xi|\varepsilon \quad \text{of} \quad A = \{A\xi\} \quad \text{of} \quad A = [A\xi]$$

in strijd is met regel 2, omdat geldt:

$$first(A) = follow(A) = first(\xi) \neq \emptyset$$

Linksrecursieve produkties worden derhalve door onze beperkende regels uitgesloten.

Wellicht denkt u op grond van de uiteenzetting tot nog toe dat problemen met syntactische regels altijd zijn op te lossen door een equivalente syntaxis te construeren die wel aan de gestelde regels voldoet.
We wijzen er daarom nogmaals op dat de syntaxis van een zin altijd slechts een middel is om een hoger doel te bereiken, namelijk het zichtbaar maken van de betekenis van die zin. Bij elke verandering van de syntaxis moeten we daarom in het oog houden dat door die verandering niet de betekenisstructuur van de taal geweld wordt aangedaan.
Laten we dit met een voorbeeld illustreren. De volgende produkties definiëren uitdrukkingen met operanden a, b, en c en de operator '-', waarmee de aftrekking wordt bedoeld.

```
S = A|S - A.
A = a|b|c.
```

Omdat de linksrecursieve produktie voor S in strijd is met regel 1 proberen we een equivalente syntaxis op te stellen; we vinden de volgende oplossing:

```
S = AB.
B = [-S].
A = a|b|c.
```

De eerste versie geeft aan de zin a - b - c een structuur die door gebruikmaking van haakjes als volgt kan worden weergegeven; ((a-b)-c).
Dezelfde zin krijgt in de tweede versie echter impliciet de structuur (a-(b-c)) toegewezen. Als we uitgaan van de gangbare regels voor aftrekken is gemakkelijk in te zien dat de twee versies weliswaar syntactisch equivalent zijn, maar niet semantisch. Uit dit voorbeeld kunnen we leren dat bij het veranderen van de syntaxis van een taal ook rekening gehouden moet worden met de betekenisstructuur van die taal.

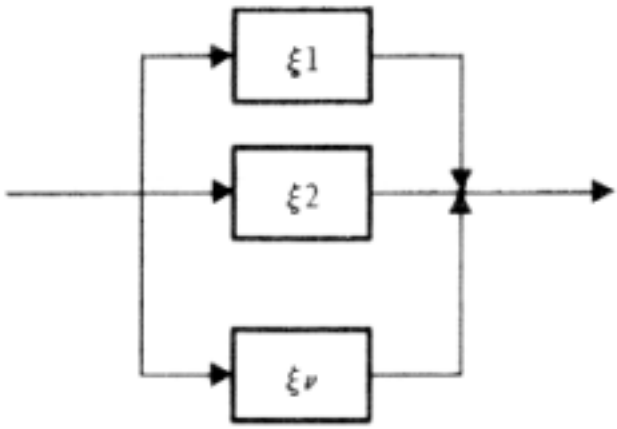
3 Syntaxisgrafen

Het voorstellen van een syntaxis in BNF is slechts een van de vele mogelijkheden. Een andere, in veel opzichten voordeliger voorstellingswijze berust op het toepassen van diagrammen of grafen. Het belangrijkste voordeel van deze voorstellingswijze is dat deze een beter overzicht biedt.
We voeren daarom hierna grafen in, waarin het verloop van de herkenning van een zin bij de top-down methode direkt te zien is. We geven een eenvoudig recept om een door BNF gedefinieerde syntaxis consequent in de daarbij behorende grafen te vertalen. De omgekeerde weg is natuurlijk even gemakkelijk.
We nemen aan dat de BNF al in die zin is genormaliseerd, dat elk niet-terminaal symbool door één reeks produkties is gedefinieerd. De vertaling wordt dan door de volgende regels vastgelegd.

A1. Elk niet-terminaal symbool dat door een reeks produkties

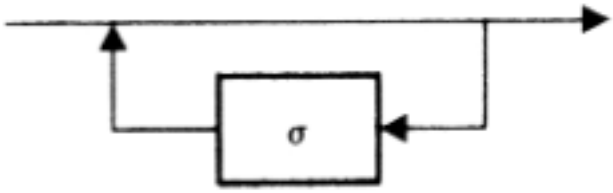
A = R1|R2|R3|...|Rn.

is gedefinieerd wordt als graaf met de volgende structuur afgebeeld:

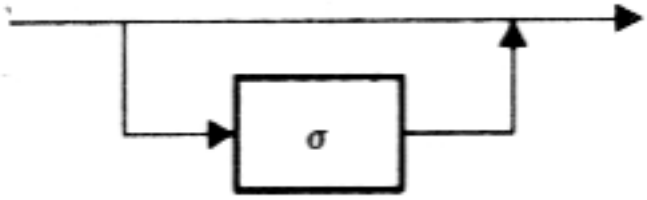


A2. Elke term Q1, Q2, Q3, ... Qn wordt in een graaf met de volgende structuur afgebeeld;

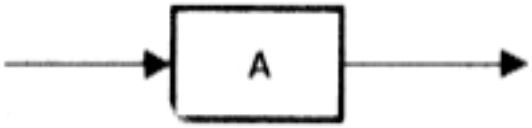
A3. Als een element de vorm {S} heeft, dan wordt het door middel van de volgende graafstructuur afgebeeld:



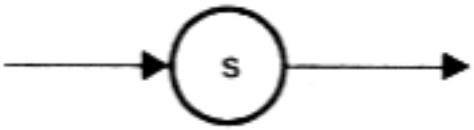
A4. Als een element de vorm [S] heeft, dan wordt het door middel van de volgende skructuur afgebeeld:



A5. Als een element een niet-terminaal symbool A is, dan wordt het omlijst door een rechthoek weergegeven.



A6. Als een element een terminaal symbool s is, dan wordt het omlijst door een cirkel weergegeven.



Voorbeeld 5

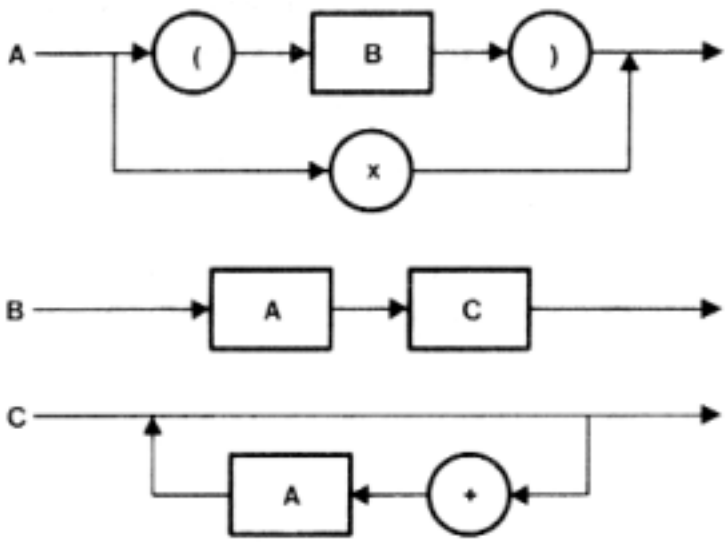
Vanaf het volgende voorbeeld plaatsen we terminale symbolen tussen dubbele aanhalingstekens; ze zijn daardoor eenduidig te herkennen. We zullen deze schrijfwijze in alle volgende voorbeelden blijven gebruiken.

```
A = "x" | "(" B ")".
B = AC.
C = {"+" A}.
```

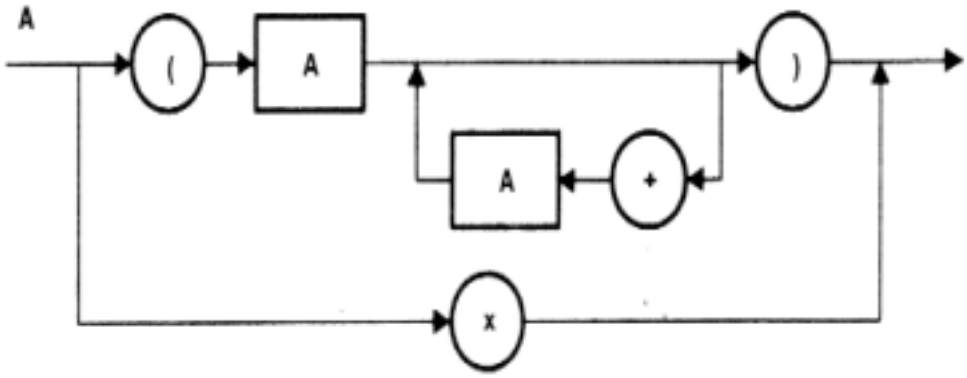
De tekens "+", "(", ")" en "x" zijn hier dus terminale symbolen van de definiërende taal, terwijl "{" en "}" als metasymbolen fungeren. De volgende zinnen behoren tot deze taal:

```
x
(x)
(x + x)
((x))
((x + (x + x)))
```

Met behulp van de eerder gegeven vertaalregels worden de drie produkties in de in figuur 1 afgebeelde grafen omgezet. Als C direkt in B en B direkt in A wordt gesubstitueerd, dan is het resultaat één graaf; deze is in figuur 2 afgebeeld.



Figuur 1: Syntaxis van voorbeeld 5



Figuur 2: Gereduceerde syntaxis van voorbeeld 5

De grafen vormen dus een alternatief voor de BNF. Ze zijn overzichtelijker dan de BNF en geven een direkte voorstelling van het proces van zinsherkenning. De vraag is derhalve hoe beide geponeerde regels grafisch zijn weer te geven. Welnu, in de grafen wordt de betekenis van deze regels pas echt duidelijk. Regel 1 vertelt ons namelijk dat bij elke vertakking in de graaf de te kiezen tak eenduidig uit de kennis van het volgende symbool moet zijn af te leiden.

Dit impliceert dat twee takken niet met hetzelfde symbool mogen beginnen. Regel 2 eist dat een graaf die zonder het lezen van een symbool kan worden doorlopen 'doorzichtig' moet zijn. Op elk punt moet eenduidig vast te stellen zijn of het volgende symbool nog tot de graaf of tot zijn opvolger behoort. De verzameling van beginsymbolen en de verzameling van mogelijke vervolgsymbolen moeten dus disjunct zijn.

Of een verzameling grafen aan deze regels voldoet is heel eenvoudig te verifiëren. Het is daarbij over het algemeen niet nodig op de weergave van de syntaxis in BNF terug te vallen. De overzichtelijke grafische voorstelling blijkt daarentegen uitermate geschikt. Uit voorzorg bepalen we weer voor elk niet-terminaal symbool A eerst de daarbij behorende verzamelingen first(A) en follow(A). We noemen een systeem van grafen dat aan de beide beperkende regels voldoet een deterministisçh systeem van grafen.

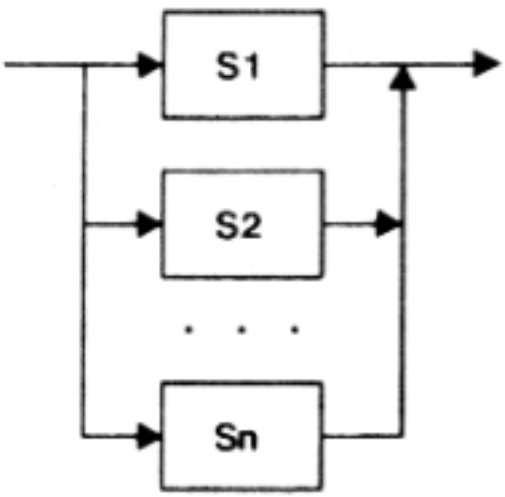
4 Een parser voor een gegeven syntaxis construeren

Als een syntaxis door een deterministisch systeem van grafen is voor te stellen dan is uit dit grafensysteem zeer systematisch een daarbijbehorende parser af te leiden. De afzonderlijke grafen corresponderen met de te herkennen syntactische categorieën en worden in afzonderlijke procedures afgebeeld. Elke graaf stelt als het ware het stroomdiagram van de ermee corresponderende procedure voor. De vertaling van het systeem van grafen in een programma kan weer door aparte regels worden beschreven, geheel analoog aan het vertalen van BNF in grafen.
De zo verkregen procedures worden in een hoofdprogramma ingebed; dit hoofdprogramma dient als omgeving. In dit programma wordt vastgelegd in welke volgorde gegevens moeten worden ingevoerd. Er moet ook een procedure in staan die steeds het volgende symbool aanlevert.
Terwille van de eenvoud gaan we er voorlopig van uit dat elk symbool uit één teken bestaat; als tekens mogen de bij moderne computers gebruikelijke tekens (ASCII-set) optreden. Het hoofdprogramma zal dus steeds de volgende vorm hebben:

```
MODULE Parser;

FROM  Terminal    IMPORT      Read;
VAR   ch          : CHAR;

(* Declaraties van afzonderlijke procedures *)

BEGIN
   Read (ch);
   S
END.
```

Hierbij staat S voor de met het startsymbool S corresponderende procedure. De afzonderlijke vertaalstappen worden door de hierna volgende regels B1 t/m B6 vastgelegd. Het is daarbij handig eerst het aantal niet-terminale symbolen te verminderen door waar mogelijk afzonderlijke grafen direct in andere grafen onder te brengen, en wel op dezelfde manier als in voorbeeld 4. We duiden het programma dat ontstaat door vertaling van een graaf S aan met P(S).

B1. Elke structuur met de vorm



wordt vertaald in een voorwaardelijke of selectie-opdracht:

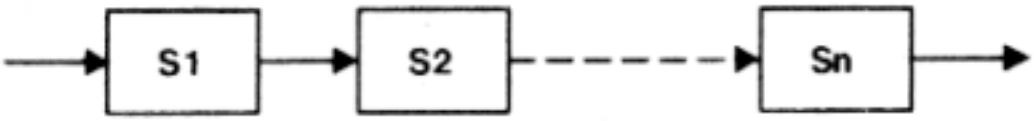
```
IF  ch IN L1  THEN  P (S1)
```

```
ELSIF  ch IN L2  THEN  P (S2)
  . . . . . . . .
ELSIF  ch IN Ln  THEN  P(Sn)
ELSE
   error
END
```

```
CASE  ch  OF
   L1 : P (S1)  |
   L2 : P (S2)  |
   ...          |
   Ln : P (Sn)
END
```

Hierbij staat Li voor de verzameling first (Si). (Aanwijzing voor het programmeren: als L uit slechts één symbool, en wel s, bestaat, dan moet de uitdrukking ch IN L natuurlijk vervangen worden door ch = s).

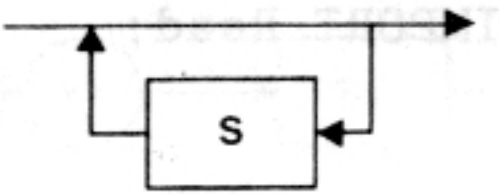
B2. Elke structuur met de vorm



wordt vertaald in een reeks opdrachten:

P (S1) ; P (S2) ; ... ; P (Sn)

B3. Elke structuur met de vorm

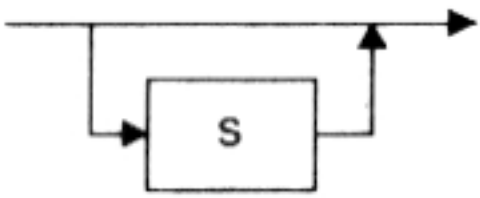


wordt vertaald in een herhalingsopdracht:

```
WHILE  ch IN L  DO  P(S)  END
```

L staat daarbij voor de verzameling first (S) (zie de aanwijzing bij B1).

B4. Elke structuur met de vorm



wordt vertaald in een voorwaardelijke opdracht:

```
IF  ch IN L  THEN  P(S)  END
```

B5. Elke verwijzing naar een graaf (rechkhoekige figuur) wordt vertaald in een aanroep van de met die graaf corresponderende procedure.

B6. Elke verwijzing naar een terminaal symbool x (ronde figuur) wordt vertaald in een leesopdracht, en wel als volgt:

```
IF  ch = "x"  THEN  Read (ch)  ELSE  error  END
```

Hierbij staat 'error' voor een niet nader gespecificeerde procedure die wordt aangeroepen zodra een niet correcte symboolreeks wordt ontdekt.
We zullen nu aan de hand van voorbeeld 5 laten zien hoe deze regels toegepast worden. We brengen daarbij enkele voor zichzelf sprekende vereenvoudigingen aan.

```
MODULE  Parser;

FROM    Terminal  IMPORT Read;

VAR   ch    : CHAR;

PROCEDURE A;

BEGIN
  IF ch = "x" THEN
    Read (ch)
  ELSIF ch = "(" THEN
    Read (ch);
    A;
    WHILE ch = "+" DO
       Read (ch);
       A
    END;
    IF ch = ")" THEN Read(ch) ELSE error END
  ELSE
    error
  END
END A;

BEGIN
  Read (ch);
  A
END Parser.
```

Programma 1: Een parser voor de taal uit voorbeeld 5

Een letterlijke vertaling zou bijvoorbeeld tot de volgende opdracht hebben geleid;

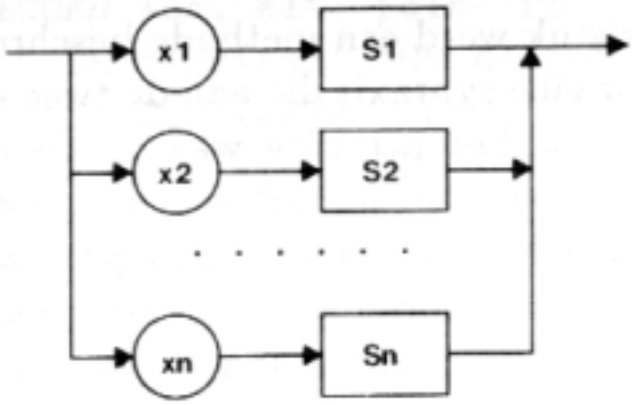
```
IF ch = "x" THEN
  IF ch = "x" THEN Read (ch) ELSE error END
ELSE
  ....
END
```

Het is onmiddelijk duidelijk dat deze opdracht is te vereenvoudigen tot;

```
IF ch = "x" THEN Read(ch) ELSE ... END
```

Zulke vereenvoudigingen zijn nogal vaak mogelijk. Twee veel voorkomende situaties leiden ons tot de volgende extra regels:

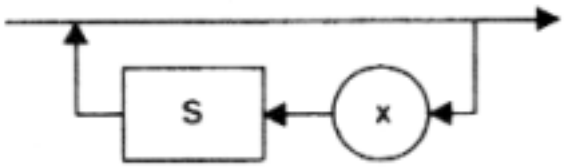
B1a. Een structuur met de vorm



wordt vertaald in de volgende opdracht:

```
IF ch = "x1" THEN
  Read (ch);
  P(S1)
ELSIF ch = "x2" THEN
  Read(ch);
  P(S2)
...
ELSIF ch = "xn" THEN
  Read (ch);
  P(Sn)
ELSE
  error
END
```

B2a. Een structuur met de vorm



wordt vertaald in de volgende WHILE opdracht:

```
WHILE ch = "x" DO Read(ch); P(S) END
```

De procedure error blijft voorlopig ongedefinieerd. Zolang we alleen maar willen uitzoeken of een gegeven tekenreeks syntactisch gezien is toegestaan of niet is het voldoende dat bij het aanroepen van 'error' het proces wordt afgebroken. In de praktijk is dit natuurlijk niet voldoende omdat gewoonlijk wordt verwacht dat de volgende tekst verder wordt onderzocht. De daaruit voortvloeiende problemen en technieken worden in hoofdstuk 9 behandeld.

5 Tabelgestuurde syntaxisanalyse

In het vorige hoofdstuk werd een methode beschreven om een parser te programmeren voor elke syntaxis die aan de twee genoemde beperkende regels voldoet. Omdat het principe waarop de analyse berust echter steeds hetzelfde blijft kunnen we ook één algemeen programma maken dat voor alle talen geschikt is. Dit algemene programma wordt, afhankelijk van de taal waarmee wordt gewerkt, voorzien van tabellen waarin de syntaxis van die taal in gecodeerde vorm wordt weergegeven. De parser wordt dan als het ware door deze tabellen -die tijdens de analyse niet veranderen- gestuurd. In deze parser is het algoritme waarmee we bij het gebruik van syntaxisgrafen een zin herkennen omgezet in een computer programma.

Om deze parser te kunnen gebruiken moet de syntaxis van de te analyseren taal uiteraard in een vorm worden voorgesteld die voor een computerprogramma toegankelijk is. De BNF en de grafen zijn hiervoor niet zo geschikt. Het ligt meer voor de hand om van grafen als stroomdiagrammen uit te gaan; deze stroomdiagrammen moeten dan in een geschikte gegevensstructuur worden vertaald.

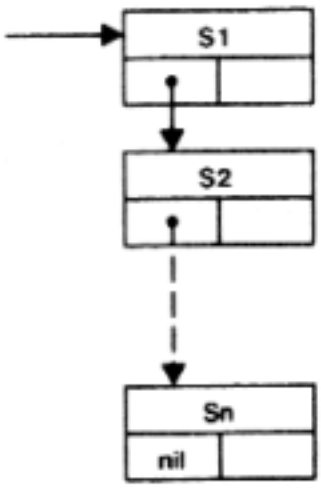
Een eenvoudige tabelvorm (array) is voor dit doel nauwelijks geschikt. We zullen veeleer een lijst structuur zonder van te voren vastliggend formaat nodig hebben. De afzonderliike symbolen corresponderen met gegevenselementen (knopen). De paden (takken) worden door pointers (wijzers) weergegeven.

In de programmeertaal Modula-2 is het record de gegchikte structuur om knopen in een lijst voor te stellen. In het onderhavige geval hebben we in elk gegevenselement (record) vier velden nodig. Het eerste veld bevat het symbool dat met de knoop correspondeert. Twee andere Velden bevatten pointers die óf naar de volgende knoop óf naar het alternatief wijzen - mits deze bestaan. Het laatste veld dient om aan te geven of het om een terminaal of om een niet-terminaal symbool gaat. Voor een record met deze structuur kunnen we het beste een recordtype definiëren waarbij voor één van de onderdelen alternatieven zijn gedefinieerd. Dit is een 'variant-record'; een recordtype met een zogenaamd 'variant part'.

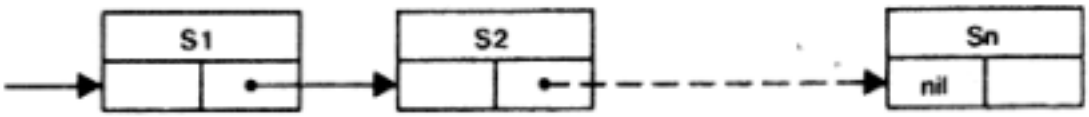
```
TYPE  pointer = POINTER TO node;

      node = RECORD
                suc , alt           : pointer;
                CASE terminal       : BOOLEAN OF
                   TRUE  : (tsym    : char)     |
                   FALSE : (nsym    : hpointer)
                END
             END
```

Het ligt voor de hand de vertaling van een systeem van grafen in een daarmee corresponderende gegevensstructuur weer te definiëren door een reeks vaste regels. Het is dan zelfs mogelijk een programma te construeren waarmee deze vertaling wordt geautomatiseerd. De regels lijken sterk op de eerder in dit boek beschreven regels.

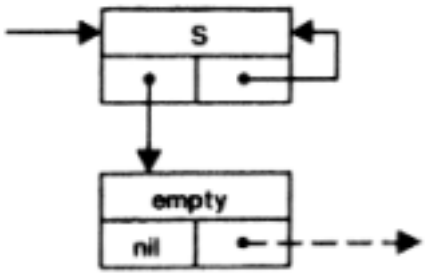
C1. Een grafenstructuur zoals onder **B1** is weergegeven wordt afgebeeld in de volgende lijststructuur:



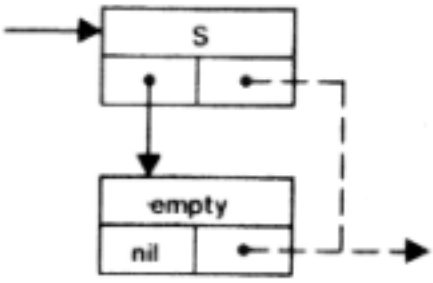
C2. Een grafenstructuur zoals onder B2 is weergegeven wordt afgebeeld in de volgende lijststructuur;



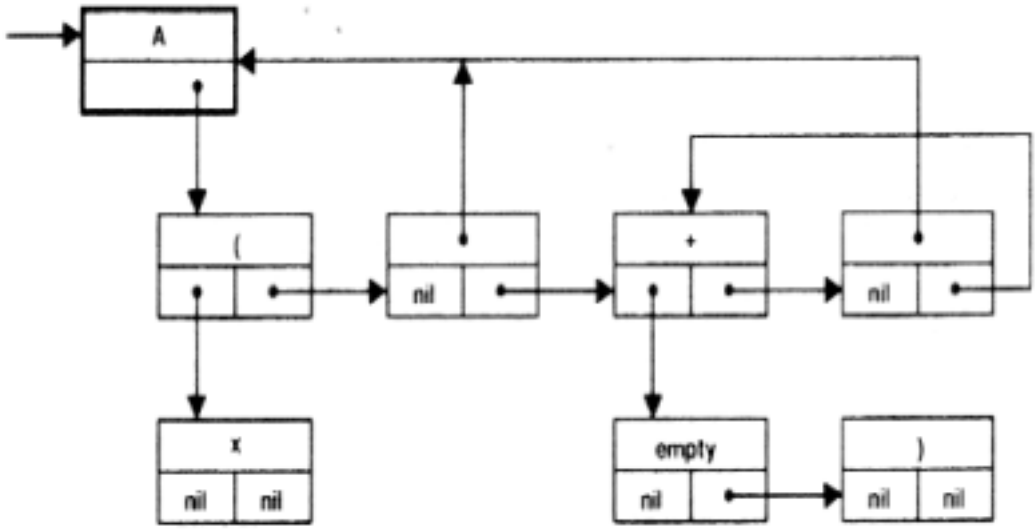
C3. Een herhalingsstructuur (zie B3) wordt als volgt 8fgebeeld:



C4. De onder B4 weergegeven structuur wordt als volgt afgebeeld:



Ter illustratie gebuiken we nogmaals voorbeeld 5 (zie figuur 2). Als we de zojuist geformuleerde regels volgen wordt deze graaf als volgt afgebeeld:



Figuur 3: Gegevensstructuur die correspondeert met de syntaxisgraaf uit fig. 2

Bovenstaande gegevensstructuur wordt geidentificeerd door een zogenaamde 'headerknoop'. Deze knoop bevat de naam van het met de knoop correspon-derende niet-terminale symbool, en wel in af te drukken vorm. Deze informatie is voor de zinsontleding in engere betekenis niet nodig, maar wordt in het later te beschrijven algoritme gebruikt om het verloop v8n de ontleding zichtbaar te maken. We spreken daarbij af dat het gegevenstype 'hpointer' er als volgt uit ziet:

```
TYPE hpointer = POINTER TO header;
TYPE header   = RECORD
                  symbol  : CHAR;
                  entry   : pointer
                END
```

Het te ontwerpen algoritme voor de ontleding van zinnen (de parser) bestaat uit een herhaalopdracht die aangeeft hoe van de ene knoop van de gegevens- structuur naar de volgende wordt gegaan.
Het programma beschrijft derhalve hoe een in de vorm van een lijststructuur weergegeven graaf geinterpreteerd moet worden. Als een knoop wordt bereikt die een niet-terminaal symbool voorstelt, dan wordt hetzelfde algoritme op de met dat symbool corresponderende gegevensstructuur toegepast vóór met de ontleding van de structuur die op dat moment wordt ontleed verder wordt gegaan.
De procedure parse -die het doorlopen van een structuur beschrijft- wordt dus recursief aangeroepen. Als een knoop een terminaal symbool voorstelt wordt gecontroleerd of het volgende invoersymbool (sym) gelijk is aan dit symbool. Als dit het geval is wordt het volgende symbool gelezen en verdergegaan met het daaropvolgende (suc).
Als dit niet het geval is wordt -als dat bestaat- het alternatief (alt) onderzocht.

```
PROCEDURE parse (goal : hpointer; VAR match: BOOLEAN);

VAR   s : pointer;

BEGIN
  s8 := goal^.entry;
  REPEAT
    IF s ^.terminal THEN
      IF s^.tsym = sym THEN
```

```
          match := TRUE;
          getsym
        ELSE
          match := (s^.tsym = empty)
        END
      ELSE
        parse (s^.nsym, match)
      END;
      IF match THEN s := s^.suc ELSE s := s^.alt END
    UNTIL  s = NIL
  END parse
```

Dit stukje programma zal een nieuw analysedoel volgen zodra zo'n doel door een ingelezen niet-terminaal symbool wordt gesteld, en wel vóór gecontroleerd wordt of het volgende invoersymbool tot de verzameling van beginsymbolen first (A) behoort. Dit betekent echter dat bij elke vertakking in een syntaxisgraaf slechts één tak mek een nietterminaal symbool mag beginnen. Als een tak overeenkomt met de lege reeks mag geen enkele parallelle taak met een niet-terminaal symbool beginnen. We formuleren deze voorwaarden hier als beperkingen van de vorm van produkties, waarbij we aannemen dat elk niet-terminaal symbool A door één samengestelde produktie is gedefinieerd.
Elke produktie moet één van de volgende drie vormen hebben:

1. $A = s_1\xi_1 | s_2\xi_2 | \ldots | s_n\xi_n$
   $A \in N,\ s_i \in T,\ \xi_i \in (N \cup T)^*,\ i \neq j \supset s_i \neq s_j$

2. $A = s_1\xi_1 | s_2\xi_2 | \ldots | s_n\xi_n | B\eta$
   $B \in N,\ \eta \in (N \cup T)^*,\ i \neq j \supset s_i \neq s_j$ en
   $s_i \notin first(B)$ voor $i = 1 \ldots n$

3. $A = s_1\xi_1 | s_2\xi_2 | \ldots | s_n\xi_n | \varepsilon$
   $i \neq j \supset s_i \neq s_j$ en $s_i \notin follow(A)$ voor $i = 1 \ldots n$

Als we minder restrictieve regels willen hanteren zijn geraffineerdere algoritmen nodig. Afhankelijk van de regels zullen die algoritmen met meer of minder moeite zijn te realiseren. We kunnen zelfs van de regel "no backtracking" afwijken door bovenstaand programma dienovereenkomstig te generaliseren. Dit leidt echter tot een gevoelig verlies aan efficiëntie; bij practische toepassingen moet zo'n verlies als het enigszins kan vermeden worden.

Het voorstellen van een syntaxis door middel van grafen heeft naast voordelen ook één belangrijk nadeel: de grafen kunnen niet direkt door een computer worden gelezen. In dit opzicht lijkt de BNF ideaal, omdat deze een syntaxis heel conventioneel weergeeft als formules die uit afzonderlijke letters en leestekens bestaan. In het volgende hoofdstuk wordt daarom de vraag behandeld hoe tabellen voor een parser direkt uit een BNF-specifikatie zijn af te leiden.

6 BNF-produkties vertalen in tabellen

Tot nog toe hebben we de transformatie van één voorstellingswijze in een andere informeel gedefinieerd. In dit hoofdstuk willen we hiervoor een formeel programma maken. Dit is niet alleen interessant omdat daardoor het vertalen door een computer kan worden gedaan -automatisch en zonder fouten-; het belang van zo'n transformatieprogramma is vooral dat het een direkte toepassing van de in hoofdstuk 4 beschreven techniek vormt.
Dit is het geval omdat de BNF -of preciezer geformuleerd: de hier toegepaste uitgebreide BNF- kan worden gebruikt om zichzelf te definiëren. We duiden deze hierna preciezer te definiëren metataal aan als uitgebreide (extended) Backus-Naur Form (EBNF). Wij stellen ons dus ten doel een parser voor deze metataal te construeren. We doen dit in vier stappen:

1. Het definiëren van de syntaxis van de EBNF, uitgedrukt in EBNF zelf.

2. Het construeren van een parser voor deze syntaxis volgens de regels B1 t/m B6.

3. Het uitbreiden van deze parser tot een vertaler die parsertabellen produceert.

4. Het integreren van deze vertaler met de tabelgestuurde parser.

Stap 1.

Geheel in overeenstemming met de tot nu toe gevolgde notatieregels moet de invoertekst -die geheel uit in EBNF geformuleerde produkties bestaat- voldoen aan de volgende voorwaarden:

1. Niet-terminale symbolen bestaan uit één hoofdletter.

2. Terminale symbolen bestaan uit één teken (niet noodzakelijkerwijze een letter) tussen aanhalingstekens (een 'literal').

De syntaxis waarmee een EBNF-produktie wordt beschreven ligt dan vast in de volgende produkties:

```
production  = identifier "=" expression "." .
expression  = term {|term}.
term        = factor {factor}.
factor      = letter | literal | "(" expression ")" | "[" expression "]" |
              "{" expression "}".
```

Stap 2

De net gegeven lexicografische regels maken de constructie van de parser tot een eenvoudige opgave. Het lezen van het volgende symbool is echter niet eenvoudigweg door de opdracht Read (ch) voor te stellen. Het moet minstens mogelijk zijn net zoveel blanco tekens (hier alleen spaties) tussen de symbolen te zetten als we willen.
Daarom maken we voor het lezen van de invoergegevens een procedure GetSym. Deze procedure zorgt ervoor dat het volgende symbool aan sym wordt toegewezen, ongeacht het aantal blanco karakters (spaties) dat tussen de symbolen staat. Deze

procedure vormt dus een triviaal voorbeeld van een mechanisme dat in algemene zin een scanner wordt genoemd. Het doel van
een scanner is het op basis van de regels van de gekozen lexicografische weergave van de tekst opsporen van het volgende symbool. Zo is het bijvoorbeeld gebruikelijk dat bepaalde symbolen uit meer tekens bestaan. De scanner maakt dan een afbeelding van tekenreeksen in symbolen.

Hier zien we wat het verschil is tussen tekens en symbolen: tekens zijn de 'atomen' van de tekst die wordt ingevoerd, symbolen zijn de bouwstenen van de door de syntaxis gedefinieerde taal; deze taal is onafhankelijk van de tekens waaruit de symbolen in die taal zijn opgebouwd.
Voor we overgaan tot het construeren van een parser moeten we nog nagaan of de hier gegeven syntaxis wel aan de beperkende regels 1 en 2 voldoet. Daarom passen we de vertaalregels B1 t/m B6 'mechanisch' toe op de syntaxis. Programma 2 is het resultaat:

```
MODULE EBNF;

FROM  Terminal IMPORT   Read, Write, WriteLn, WriteString;

CONST empty   = " ";

VAR   sym     : CHAR;

PROCEDURE GetSym;

BEGIN
  REPEAT
    Read (sym);
    Write (sym)
  UNTIL sym # empty
END GetSym;

PROCEDURE error;

BEGIN
  WriteLn;
  WriteString ("Syntax incorrect");
  HALT
END error;

PROCEDURE expression;

  PROCEDURE term;

    PROCEDURE factor;

    BEGIN
      IF ("A" <= sym) & (sym <="Z") THEN
        (* nonterminal symbol *)
        GetSym
      ELSIF sym = '"' THEN
        (* terminal symbol      *)
        GetSym;
        GetSym;
        IF sym = '"' THEN GetSym ELSE error END
      ELSIF sym = "(" THEN
        GetSym;
```

```
        expression;
        IF sym = ")" THEN GetSym ELSE error END
      ELSIF sym = "[" THEN
        GetSym;
        expression;
        IF sym = "]" THEN GetSym ELSE error END
      ELSIF sym = "{" THEN
        GetSym;
        expression;
        IF sym = "}" THEN GetSym ELSE error END
      ELSE
        error
      END
    END factor;

  BEGIN                              (* term *)
    factor;
    WHILE ("A" < sym) & (sym < "Z") OR (sym = '"') OR
            (sym ="(") OR (sym = "[") OR (sym = "{") DO
      factor
    END
  END term;

  BEGIN                              (* expression *)
    term;
    WHILE sym = "|" DO
       GetSym;
       term
    END
  END expression;

BEGIN                                (* main program   *)
  GetSym;
  WHILE sym # "." D0
    IF ("A" <= sym) & (sym <= "Z") THEN GetSym ELSE error END;
    IF sym = "=" THEN GetSym ELSE error END;
    expression;
    IF sym = "." THEN
      WriteLn;
      GetSym
    ELSE
      error
    END;
  END
END EBNF.
```

Programma 2: Syntaxisanalyse van EBNF

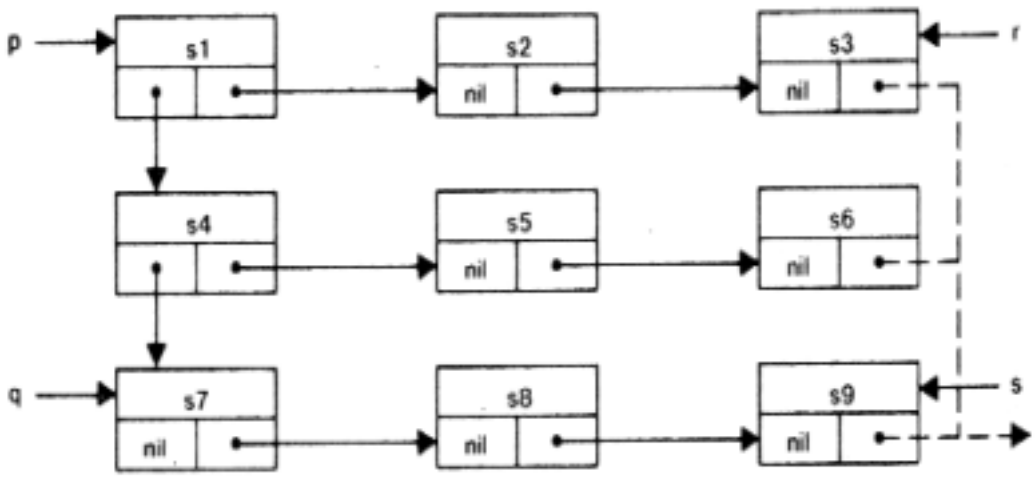
Stap 3

Het volgende doel is de uitbreiding van het zo verkregen programma tot een vertaler die de gewenste gegevensstructuur produceert. Helaas is deze stap niet op dezelfde manier als de vorige te normaliseren of te systematiseren. Bij gebrek aan formele constructieregels passen we het volgende, voor de hand liggende recept toe: we schrijven de met elke syntactische eenheid corresponderende structuur nogmaals op en definiëren precies de parameters van de procedure die deze structuur produceert.
Elke aangeroepen procedure produceert dus een gegevensstructuur die corre-spondeert met de door die procedure gelezen tekst. Deze gegevensstructuur wordt

vervolgens door de aanroepende procedure als bouwsteen in de door de aanroeper geconstrueerde gegevensstructuur ingevoegd. Dit proces is uiteraard over het algemeen recursief.
Omdat het bij de zo gemaakte produkten om onderling door pointers aan elkaar gekoppelde gegevensstruckuren gaat ligt het voor de hand niet direkt met deze structuren zelf te werken, maar met pointers naar die structuren. Elke van de vier genereringsprocedures factor, term en expression krijgt vier parameters die met p, q, r en s aangeduid worden. Hun betekenis wordt duidelijk uit de hieronder afgebeelde gegevensstructuur, die correspondeert met de syntactische uitdrukking:

s1 s2 s3 | s4 s5 s6 | s7 s8 s9

De vier pointers worden door de procedure expression gegenereerd.



Hoofddoel van de procedure factor is het produceren van nieuwe elementen (knopen) in de gegevensstructuur; deze gegevenselementen corresponderen met de gelezen symbolen. De beide procedures term en expression hebben daarentegen de taak deze elementen aan elkaar te koppelen.
Daartoe wordt door term het pointerveld suc en door expression het veld alt gebruikt. Verdere bijzonderheden zijn uit programma 3 af te lezen.

We vestigen hier nog de aandacht op een detail bij het verwerken van niet-terminale symbolen. Zo'n symbool kan binnen een factor voorkomen vóór het als linkerdeel van een produktie optreedt en dus vóór het gedefinieerd is. Het is daarom nodig te onthouden of een symbool al gedefinieerd is.
Daartoe worden alle niet-terminale symbolen in een record van het type Header geregistreerd. Het veld entry geeft aan of al een met het symbool corresponderende gegevensstructuur is gemaakt. Deze records worden in een lineaire lijst aan elkaar gekoppeld. De variabele 'list' stelt het begin, 'sentinel' het einde van deze lijst voor. Dit einde is een fictief element dat dient om het zoekproces te vereenvoudigen. Dit zoekproces is vastgelegd in de procedure find (sym, h). Als het symbool sym wordt gevonden, dan wordt de pointer naar het met dit symbool correspon-derende element toegekend aan de variabele h. Komt het symbool niet in de lijst voor, dan wordt een element toegevoegd. Verdere bijzonderheden zijn in het eerste deel van programma 3 te vinden.

Stap 4

Nu worden de vertaler en de algemene parser uit hoofdstuk 5 tot één geheel verenigd. Zowel het programma als de invoergegevens bestaan uit drie delen. Eerst

komen de parser voor EBNF, respectievelijk de syntaxis van de te behandelen taal uitgedrukt in EBNF-produkties.
Daarna volgen een opdracht waamee het startsymbool wordt gelezen, resp. het startsymbool. Tenslotte volgen de algemene parser, resp. de tot de eerder gedefinieerde taal behorende en te herkennen zinnen. (Het startsymbool moet door een $-teken worden voorafgegaan).

```
MODULE GeneralParser;

FRDM  SYSTEM      IMPORT      TSIZE;
FROM  InOut       IMPORT      OpenInput, OpenOutput, Done, Read,
                              Write, WriteLn, WriteString,
                              CloseInput, CloseOutput;
FROM  Storage     IMPORT      ALLOCATE;

CONST empty       = " ";
      ESC         = 33C;

TYPE  NodePtr     = POINTER TO Node;
      HeadPtr     = POINTER TO Header;
      Node        = RECORD
                      suc, alt      : NodePtr;
                      CASE terminal : BOOLEAN OF
                         TRUE: tsym : CHAR      |
                        FALSE: nsym : HeadPtr
                      END
                    END;
      Header      = RECORD
                      sym     : CHAR;
                      entry   : NodePtr;
                      suc     : HeadPtr
                    END;

VAR   list, sentinel, h : HeadPtr;
      q, r, s           : NodePtr;
      sym               : CHAR;
      noerr             : BOOLEAN;

PROCEDURE GetSym;

BEGIN
  REPEAT
    Read (sym);
    Write (sym)
  UNTIL (sym > empty) OR (sym = ESC)
END Getsym;

PROCEDURE find (s  : CHAR; VAR h  : HeadPtr);

VAR   h1    : HeadPtr;

BEGIN
  h1 := list;
  sentinel^.sym := s;
  WHILE h1^.sym # s DO h1 := h1^.suc END;
  IF h1 = sentinel THEN                    (* insert new *)
    ALLOCATE (sentinel, TSIZE (Header));
    h1^.suc := sentinel;
    h1^.entry := NIL
```

```
  END;
  h := h1
END find;

PROCEDURE error;

BEGIN
  WriteLn;
  writeString (" inconect syntax ");
  noerr := FALSE
END error;

PROCEDURE link (p,q  : NodePtr);

VAR  t  : NodePtr;

BEGIN         (* insert q in places indicated by linked chain p *)
  WHILE p # NIL DO
    t := p;
    p := t^.suc;
    t^.suc := q
  END
END link;

PR0CEDURE expression (VAR  p,q,r,s   : NodePtr);

VAR   q1, s1   : NodePtr;

  PROCEDURE term (VAR  p,q,r,s  : NodePtr);

  VAR   p1, q1, r1, s1   :    NodePtr;

    PROCEDURE factor (VAR  p, q, r, s   : NodePtr);

    VAR  a  : NodePtr;
         h  : HeadPtr;

    BEGIN
      IF ("A" <= sym) & (sym <= "Z") THEN
        ALLOCATE (a, TSIZE (Node));
        find (sym, h);
        WITH a^ DO
          terminal := FALSE;
          nsym := h;
          alt := NIL;
          suc := NIL
        END;
        p := a;
        q := a;
        r := a;
        s := a;
        GetSym
      ELSIF sym = '"' THEN          (* terminal symbol *)
        GetSym;
        ALLOCATE (a, TSIZE (Node));
        WITH a^ DO
          Terminal := TRUE;
          tsym := sym;
           alt := NIL;
           suc := NIL
```

```
        END;
        p := a;
        q := a;
        r := a;
        s := a;
        GetSym;
        IF sym = '"' THEN GetSym ELSE error END
      ELSIF  sym = "("  THEN
        GetSym;
        expression (p, q, r, s);
        IF sym = ")" THEN GetSym ELSE error END
      ELSIF sym = "[" THEN
        Getsym;
        expression (p, q, r, s);
        ALLOCATE (a, TSIZE (Node));
        WITH  a^  DO
          terminal := TRUE;
          tsym := empty;
          alt := NIL;
          suc := NIL
        END;
        q^.alt := a;
        a^.suc := a;
        q := a;
        s := a;
        IF sym = "]" THEN GetSym ELSE error END
      ELSIF  sym = "{"  THEN
        GetSym;
        expression (p, q, r, s);
        link (r, p);
        ALLOCATE (a, TSIZE (Node));
        WITH a^ DO
          terminal := TRUE;
          tsym := empty;
          alt := NIL;
          suc := NIL
        END;
        q^.alt:= a;
        q := a;
        r := a;
        s := a;
        IF sym = "}" THEN GetSym ELSE error END
      ELSE
        error
      END
    END factor;

  BEGIN                  (* term *)
    factor (p, q, r, s);
    WHILE ("A" <= CAP (sym)) & (CAP (sym) <= "Z") OR (sym = '"') OR
            (sym = "(") OR (sym = "[") OR (sym = "{")  DO
      factor (p1, q1, r1, s1);
      link (r, p1);
      r := r1;
      s := s1
    END
  END term;

BEGIN                          (* expression     *)
  term (p, q, r, s);
  WHILE sym = "|" DO
```

```
    GetSym;
    term (q^.alt, q1, s^.suc, s1);
    q := q1;
    s := s1
  END
END expression;

PROCEDURE parse (goal  : HeadPtr;  VAR  match  : BOOLEAN);

VAR   s  : NodePtr;

BEGIN
  a := goal^.entry;
  REPEAT
    IF s^.terminal THEN
      IF s^.tsym = sym THEN
        match := TRUE;
        GetSym
      ELSE
        match := (s^.tsym = empty)
      END
    ELSE
      parse (s^.nsym, match)
    END;
    IF match THEN s := s^.suc ELSE s := s^.alt END;
  UNTIL s = NIL
END parse;

BEGIN                    (* main program   *)
  OpenInput ("EBNF");
  IF Done THEN
    noerr := TRUE;
    ALLOCATE (sentinel, TSIZE (Header));
    list := sentinel;

    (* read productions and build data structure *)

    LOOP
      GetSym;
      IF  sym = "$"  THEN  EXIT  END;
      find (sym, h);
      GetSym;
      IF sym = "="  THEN  GetSym  ELSE  error  END;
      expression (h^.entry, q, r, s);
      link (r, NIL);
      IF  sym # "."  THEN  error  END
    END;
    IF noerr THEN
      h:= list;
      (* check whether all symbols are defined *)
      WHILE  h # sentinel  DO
        IF  h^.entry = NIL  THEN
          WriteString (" undefined symbol ");
          Write (h^.sym);
          WriteLn ;
          noerr := FALSE
        END;
      h := h^.suc
      END
    END;
```

```
    IF noerr THEN             (* read goal symbol *)
      GetSym;
      find (sym, h);
      WriteLn;
      CloseInput;
      LOOP              (* read and parse sentences *)
        Write (">");
        GetSym;
        IF sym = ESC THEN EXIT END;
        parse (h, noerr);
        IF noerr & (sym = ESC) THEN
          WriteString ("   correct")
        ELSE
          WriteString (" incorrect")
        END;
        WriteLn
      END
    ELSE
      CloseInput
    END;
    WriteLn
  END
END GeneralParser.
```

Programma 3: Algemene parser

We vestigen er de aandacht op dat dit programma is geproduceerd door alleen maar aanvullende opdrachten in een bestaand programma in te voegen. Het bestaande programma diende alleen om zinnen te herkennen en kan als raamwerk worden gezien voor een nieuw programma dat zinnen niet alleen herkent, maar ook vertaalt in gegevensstructuren.
Deze methode van ontwikkelen van programma's is sterk aan te bevelen. Bij gebruik van deze methode is het namelijk mogelijk zich op één bepaald aspect (bijvoorbeeld het herkennen van zinnen) te concentreren; pas daarna komen andere aspecten (bijvoorbeeld het produceren van gegevenens structuren) aan de orde. Door deze manier van werken wordt het systemayisch construeren en controleren van programma's heel wat eenvoudiger. Deze methode stelt de programmeur in staat voortdurend te controleren of een complex programma juist is geprogrammeerd.
We noemen deze methode de methode van de stapsgewijze verfijning (step-wise refinement) of stapsgewijze uitbreiding. In ons nogal eenvoudige voorbeeld van deze techniek bestaat de uitbreiding uit slechts twee stappen.
Gecompliceerde talen en complexere vertaaltaken vereisen vaak een aanzienlijk groter aantal van zulke stappen. In de hoofdstukken 8 t/m 11 wordt de ontwikkeling van een programma in drie stappen beschreven; het daar beschreven proces lijkt sterk op het hier gepresenteerde.

Programma 3 laat duidelijk zien dat een door tabellen -of veeleer door gegevensstructuren- gestuurd algoritme voor zinsontleding veel flexibeler en eenvoudiger is dan een direct geprogrammeerde parser. Deze extra flexibiliteit -die echter over het algemeen niet nodig is- vormt de kern van het idee van uitbreidbare talen (extensíble languages). Dit zijn talen -of vertalers- die door het toevoegen van extra syntaxis regels gemakkelijk zijn uit te breiden. De programmeur kan naar believen voor elk programma taaluitbreidingen aangeven. Dit moet dan in de vorm van produkties die de vertaaltabellen vergroten. Een nog ambitieuzer plan stelt ons zelfs in staat de taal voor lokale gebieden binnen een programma naar believen uit te breiden (of in te krimpen).
Dit wordt mogelijk als programma 3 zodanig wordt veranderd dat afwisselend

groepen produkties en zinnen (programmadelen) worden ingevoerd. Het is echter zeer de vraag of zo'n flexibiliteit in de praktijk wenselijk of nuttig is. Het ontwikkelen van dergelíjke uitbreidbare talen en de daarbij behorende vertalers is tot nog toe niet erg succesvol geweest.
De belangrijkste oorzaak daarvan is dat de structuur van de toegevoegde uitbreidingen nog wel is aan te geven, maar dat het heel wat moeilijker blijkt te zijn hun betekenis aan te geven. Tot nog toe is elke poging om een gemakkelijk te begrijpen formalisering van dit onderdeel te maken gestrand. Nog afgezien van deze problemen, het doel van talen is dat ze wederzijds (door maker en gebruiker) worden begrepen.
Daarvoor zijn conventies (afspraken) en consensus over die conventies nodig. Het naar believen bedenken van eigen talen is duidelijk in strijd met dit einddoel.
We trekken daaruit de conclusies en concentreren ong in de rest van dit boek uitsluitend op het ontwikkelen van een eenvoudige vertaler voor één, van te voren bekende, kleine programeertaal.

7: De programmeertaal PL/0

In de nu volgende twee hoofdstukken worden een elementaire programmeertaal en een compiler voor die taal ontwikkeld. We noemen deze taal PL/0. Om binnen de perken van dit boek te bliiven moeten we deze taal en de compiler voor die taal wel eenvoudig houden.

Toch willen we graag de belangrijkste beginselen van programeertalen en compileer technieken uitleggen. Uit deze twee gegevenheden volgen de randvoorwaarden die ten aanzien van de te kiezen progrganneertaal moeten gelden. We zouden ongetwjjfeld een eenvoudigere of juist ingewikkeldere taal kunnen kiezen.

PL/0 is slechts één van de mogelijke compromissen tussen enerzijds voldoende eenvoud terwille van een overzichtelijke uiteenzetting en anderzijds voldoende complexiteit om het hele project realistisch en dus de moeíte waard te maken.

PL/0 is wat betreft programma structuren naar verhouding goed ontwikkeld. Als elementaire opdracht vinden we de waardetoekenning. De samengestelde opdrachten laten zien hoe reeksen opdrachten, voorwaardelijke opdrachten en herhalingsopdrachten in elkaar zitten.

Als voorwaardelijke en herhalingsopdrachten bevat PL/0 de conventionele structuren IF en WHILE. PL/0 bevat verder het belangrijke concept subprogramma, in de taal gerepresenteerd door proceduredeclaraties en -aanroepen.

Waar het gaat om gegevenstypen en gegevensstructuren is PL/0 een buitengewoon karig toegeruste taal. Het enige type in de taal is het gehele getal (integer), terwijl gegevensstructuren zelfs helemaal ontbreken.

Het is dus mogelijk integerconstanten en -variabelen te declareren. Ook kunnen met rekenkundige basisoperaties expressies worden geconstrueerd.

Procedures zijn logisch gezien eenheden van bewerkingen. Het ligt daarom voor de hand het belangrijke concept van lokaliteit van objecten (constanten, variabelen en procedures) in de taal op te nemen. PL/0 biedt de mogelijkheid objecten als lokale objecten te declareren. Dit betekent dat deze objecten alleen binnen de procedure waarin ze zijn gedeclareerd gelden.

Verder bevat PL/0 eenvoudige opdrachten voor de in- en uitvoer van gegevens. Het lezen en vervolgens aan een varìabele toekennen van een waarde v wordt aangeduid met **?v**; de uitvoer van een waarde x met **!x**.

Bovenstaande korte karakterisering van de taal heeft als belangrijkste doel de lezer voldoende intuitief inzicht te geven om de hieronder gedefinieerde syntaxis te doorzien en de met de afzonderlijke structuren corresponderende betekenis te begrijpen.
De syntaxis wordt hieronder in EBNF weergegeven. Aan het einde van dit hoofdstuk vindt u de syntaxis afgebeeld in grafen.

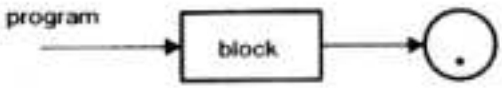
Syntaxis van PL/0:

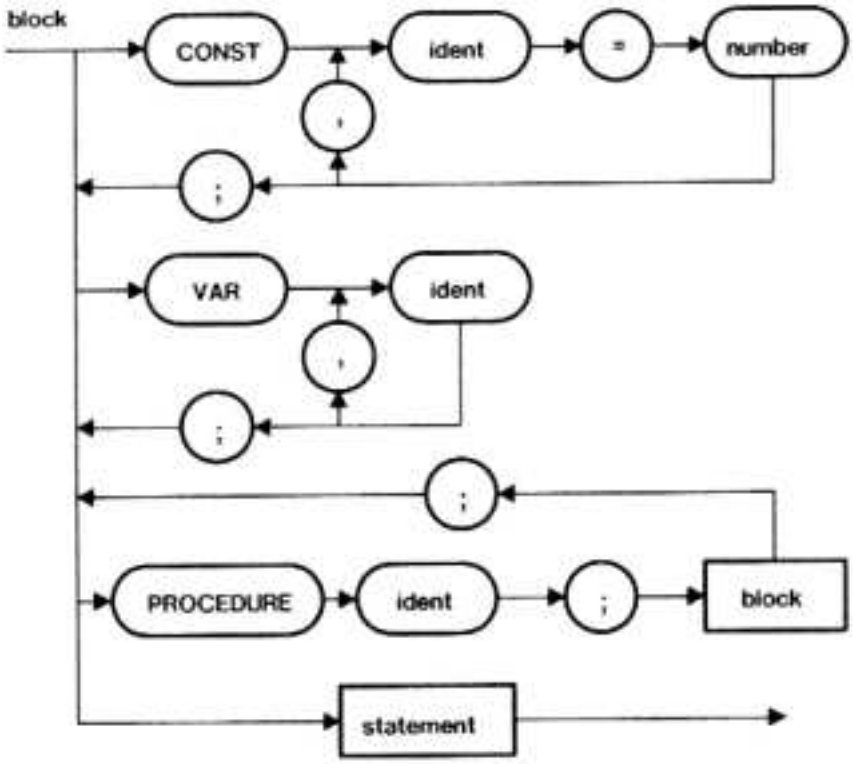
```
program     = block "." .
block       = ["CONST" ident "=" number { "," ident "=" number} ";"]
              ["VAR"   ident {"," ident} ";"]
              {"PROCEDURE" ident ";" block ";"} statement.
statement   = [ident ":=" expression | "CALL" ident | "?" ident
              | "!" expression
              | "BEGIN" statement {";" statement } "END"
              | "IF" condition "THEN" statement
              | "WHILE" condition "DO" statement ].
condition   = "ODD" expression
              | expression ("="|"#"|"<="|"<"|">"|">=") expression .
expression  = ["+"|"-"] term {("+"|"-") term}.
term        = factor {("*"|"/") factor}.
factor      = ident | number | "(" expression ")".
```

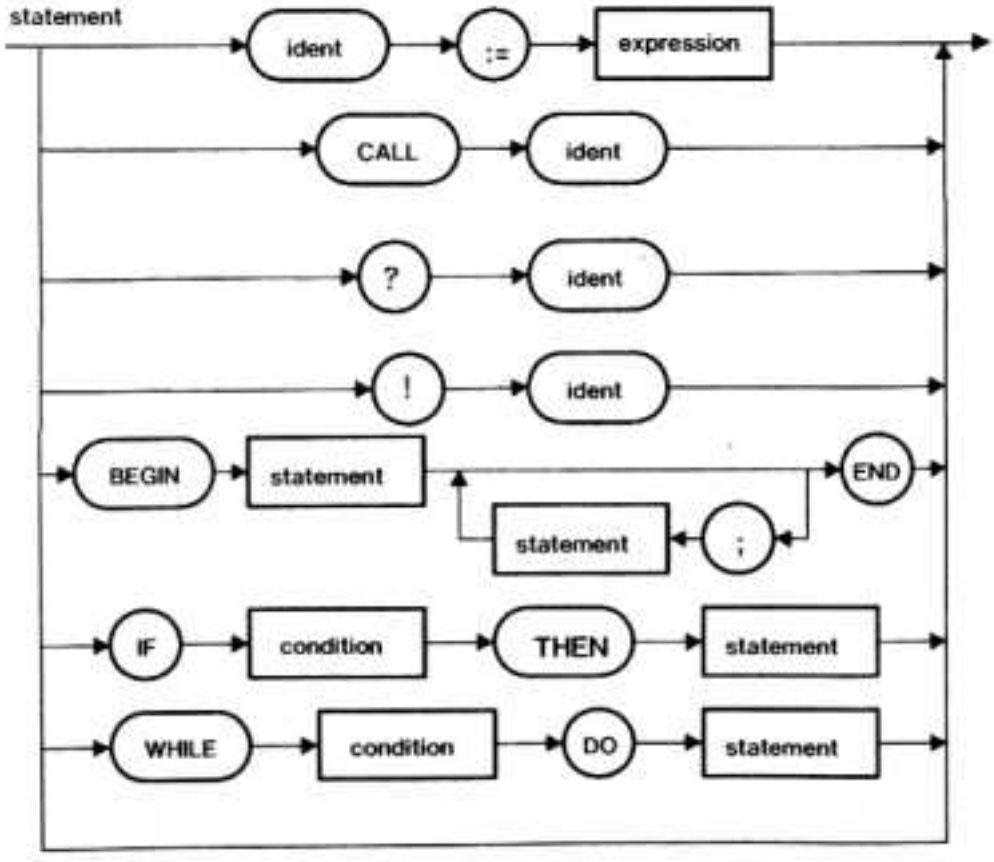
We geven één voorbeeld van een PL/0-programma om te laten zien wat de elementen van de taal zijn en hoe een in die taal geschreven programma eruit ziet. Het gekozen programma bevat procedures voor het vermenigvuldigen en delen van twee natuurlijke getallen, voor het bepalen van de grootste gemeenschappelijke deler van twee getallen en voor het berekenen van de faculteit yan een getal door middel van recursie.

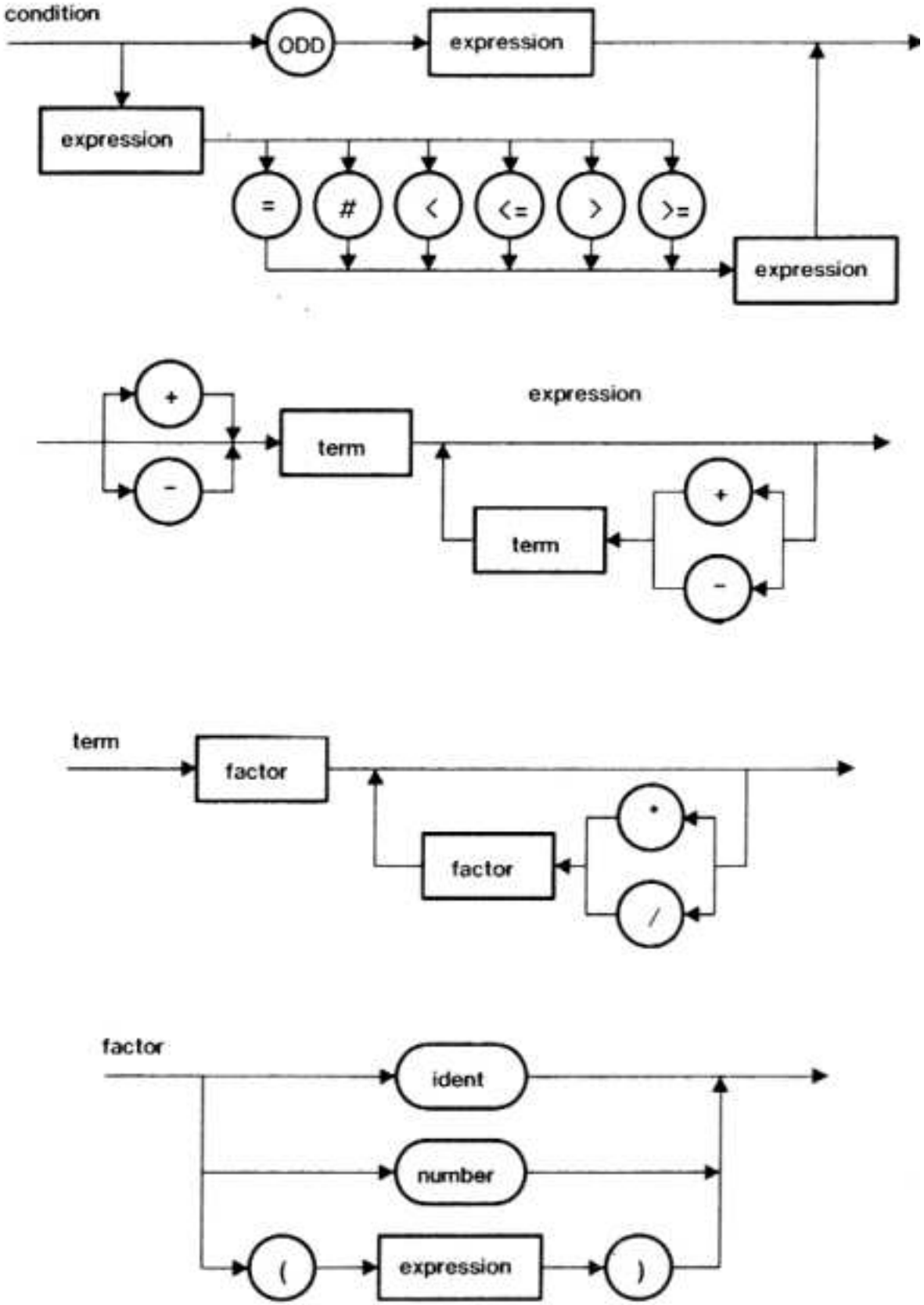
```
VAR  x, y, z, q, r, n, f;

PROCEDURE multiply;

VAR  a, b;

BEGIN
  a := x;
  b := y;
  z := 0;
  WHILE b>0 DO
  BEGIN
    IF ODD z THEN z := z + a;
    a := 2 * a;
    b := b / 2
  END
END;

PROCEDURE divide;

VAR  w;

BEGIN
  r := x ;
  q := 0;
  w := y;
  WHILE w <= r DO w := q * w;
  WHILE w >  y DO
  BEGIN
    q := a*q;
    w := w/a;
    IF w <= r THEN
    BEGIN
```

```
      r := r - w ;
      q := q + 1
    END
  END
END;
PRDCEDURE gcd ;

VAR f, g;

BEGIN
  f := x;
  g := y;
  WHILE f # g DO
  BEGIN
    IF f < g THEN g := g - f;
    IF g < f THEN f := f - g;
  END;
  z := f
END;

PROCEDURE fact;

BEGIN
  IF n > 1 THEN
  BEGIN
    f := n*f;
    n := n-1;
    CALL fact
  END
END;

BEGIN
  ?x; ?y; CALL multiply; !z;
  ?x; ?y; CALL divide; !q; !r;
  ?x; ?y; CALL gcd; !z;
  ?n; f := 1; CALL fact; !f
END.
```


program
block
.


block
CONST
ident
=
number
,
;
VAR
ident
,
;
;
PROCEDURE
ident
;
block
statement


statement
ident
:=
expression
CALL
ident
?
ident
!
ident
BEGIN
statement
END
statement
;
IF
condition
THEN
statement
WHILE
condition
DO
statement



Figuur 4: Syntaxisdiagrammen van de taal PL/0

8 Een parser voor PL/0

Als eerste stap construeren we een parser voor PL/0. Deze parser breiden we dan later via de methode van de stapsgewijze verfijning uit tot een compiler. Deze eerste stap kan strikt volgens de gegeven regels B1 t/m B6 worden uitgevoerd. Eerst moet echter nog worden gecontroleerd of de syntaxis van PL/0 wel voldoet aan de regels 1 en 2 uit hoofdstuk 2.
Zodra voor elk van de hiervoor gegeven syntaxisgrafen (d.w.z. voor elk niet-terminaal symbool) ìs vastgesteld wat de verzameling begìnsymbolen en wat de verzameling vervolgsymbolen is kan deze controle gemakkelijk aan de hand van die grafen worden uitgevoerd. In de tabel hieronder vindt u die verzamelingen begin- en vervolgsymbolen.

symbool S	Beginsymbolen(S)	Vervolgsymbolen(S)
Block	**CONST VAR PROCEDURE ident ? ! CALL IF BEGIN WHILE**	**. ;**
Statement	**ident ? ! CALL BEGIN IF WHILE**	**. ; END**
Condition	**ODD + - (ident number**	**THEN DO**
Expression	**+ - (ident number**	**. ;) END THEN DO = # < <= > >=**
Term	**(ident number**	**. ;) END THEN DO = # < <= > >=**
Factor	**(ident number**	**. ;) END THEN DO = # < <= > >=**

Begin- en vervolgsymbolen in PL/0

De oplettende lezer heeft ongetwijfeld al gezien dat de basissymbolen van PL/0 -in tegenstelling tot de eerdere voorbeelden- niet alleen uit tekens, maar over het algemeen uit tekenreeksen bestaan.
Zo worden de tekenreeksen 'BEGIN', ':=' en zelfs hele identifiers als symbolen geinterpreteerd. We gebruiken net als in programma 3 een scanner om uit de reeks ingevoerde tekens de afzonderlijke symbolen te lezen. In deze scanner zijn de zogenaamde lexicografische regels vastgelegd: de regels die bepalen hoe afzonderlijke symbolen zijn opgebouwd. We noemen de scanner weer GetSym. Dit zijn de taken van de scanner:

1. Blanco tekens en tekens die het einde van een zin aangeven worden overgeslagen.
2. Zogenaamde gereserveerde woorden -zoals BEGIN, END, enz.- worden herkend en als symbolen aan de parser doorgegeven.
3. Reeksen letters en cijfers die geen gereserveerde woorden vormen worden als identifiers herkend. De variabele sym krijgt de waarde ident. De identifier wordt zelf aan de variabele id toegekend.
4. Reeksen cijfers worden herkend als getallen. De variabele sym krijgt de waarde number (getal); het getal wordt aan de variabele num toegekend.

5 Combinaties van bijzondere tekens (bijvoorbeeld :=) worden herkend en als symbool aan de parser doorgegeven.
6 Commentaar -in PL/0 voorgesteld als tekenreeks die begint met (* en eindigt met *)- wordt overgeslagen.
7 U vindt de tekst van de scanner in programma 4. De scanner is opgezet als module en als zodanig een klassiek voorbeeld van het gebruik van het idee van modules.

Door modules te gebruiken kunnen details van een programma voor de gebruiker worden verborgen. Alleen die aspecten, die voor het gebruik van de module relevant zijn worden voor de gebruiker zichtbaar gemaakt (geimporteerd). Deze aspecten worden in het zogenaamde definitiegedeelte samengevat.
Zo'n definitiemodule wordt gewoonlijk als afzonderlijk tekstbestand (file) opgeslagen en kan apart worden gecompileerd.

De procedure Getch leest het eerstvolgende teken uit de te verwerken programmatekst en kopieert dit teken in de uitvoer. Voor dit doel wordt in ons voorbeeld niet -zoals gebruikelijk- een nieuw (uitvoer)bestand gebruikt; de uitvoer wordt in een venster (window) zichtbaar gemaakt.
We gaan er daarbij vanuit dat als uitvoerapp%aat een beeldscherm beschikbaar is waarop met behulp van een module verschillende uitvoerstromen in verschilende vensters kunnen worden afgedrukt.
Hoewel in dit hoofdstuk nog maar één uitvoerstroom voorkomt zullen bij de verdere ontwikkeling van de compiler hieraan nog andere stromen worden toegevoegd.
In programma 4 worden de noodzakelijke hulpprocedures geimmporteerd uit een module TextWindows; deze module wordt hier niet verder besproken. Plaats en grootte van de vensters worden bij de opening van die vensters door de procedure OpenTextWindow bepaald.

Met de procedure Mark(n) worden ontdekte fouten zichtbaar gemaakt. Terwille van de eenvoud worden alleen foutnummers in de in een venster afgedrukte programmatekst ingevoegd. De fouten worden in het venster geinverteerd weergegeven (bijvoorbeeld zwart op wit in plaats van wit op zwart).
De betekenis van de afzonderlijke foutnummers is in de tabel aan het einde van hoofdstuk 8 te vinden. De scanner kijkt precies één symbool in de tekst vooruit.

De beslissingen van de parser worden genomen op grond van dit eerstvolgende symbool, voorgesteld door de variabele sym. De procedure GetCh kijkt echter nog een karacter verder vooruit. Dit is hier nodig omdat symbolen uit meer tekens kunnen bestaan. De details van deze routines zijn uit programma 4 af te lezen; dit programma vormt een volledige parser voor PL/0.

```
DEFINITION MODULE PL0Scanner;

FROM  TextIO      IMPORT      File;
TYPE  Symbol      = (null, odd, times, div, plus, minus, eql, neq, lss,
                     leq, gtr, geq, comma, rparen, then, do, lparen,
                     becomes, number, ident, semi, end, call, if, while,
                     begin, read, write, const, var, proc, period, eof);
VAR   symbol            : Symbol;
      id, number        : CARDINAL;
      source            : File;

PROCEDURE Diff (u, v    : CARDINAL) : INTEGER;
```

```
      (* Difference between identifiers at buf [u] and buf [v]    *)

PROCEDURE KeepId;

PROCEDURE Mark (n : CARDINAL);

      (* Mark error 'n' in sourcetext                             *)

PROCEDURE GetSymbol;

      (* Get next symbol; results in 'symbol', 'id' and 'number'  *)

PROCEDURE InitScanner;

PROCEDURE CloseScanner;

END PL0Scanner.


IMPLEMENTATION MODULE PL0Scanner;

FROM  InOut       IMPORT      Read;
FROM  TextIO      IMPORT      ReadChar;
FROM  TextWindows IMPORT      Window, OpenTextWindow, Write, WriteCard,
                              Invert, CloseTextWindow;

CONST maxCard     = 177777B;  (* = FFFF in hexadecimal           *)
      bufLen      = 1024;

TYPE  SymbolRecord  = RECORD
                        symbol      : Symbol;
                        ind         : CARDINAL
                      END;

VAR   ch          : CHAR;     (* last character read             *)
      K,                      (* number of keywords              *)
      id0, id1    : CARDINAL; (* Indexes into identifier buffer   *)
      win         : Window;
      keyTab      : ARRAY [1..20] OF SymbolRecord;
      buf         : ARRAY [0..bufLen-1] OF CHAR;
                              (* character buffer; identifiers are *)
                              (* stored with leading length count  *)

PROCEDURE Mark (n  : CARDINAL);

BEGIN
   Invert (win, TRUE);
   WriteCard (win, n, 1);
   Invert (win, FALSE)
END Mark;



PROCEDURE GetCh;

BEGIN
   ReadChar (source, ch);
   Write (win, ch)
```

```
END Getch;


PROCEDURE Diff (u, v  : CARDINAL) : INTEGER;

VAR  w      : CARDINAL;
     i      : INTEGER;

BEGIN
   w := ORD (buf [u]);
   LOOP
      IF  w = 0  THEN  RETURN 0
      ELSIF  buf [u] # buf [v]  THEN
         RETURN INTEGER (ORD (buf [u])) - INTEGER (ORD (buf [v]))
      ELSE
         INC (u);
       INC (v);
       DEC (w)
      END
   END
END Diff;


PROCEDURE KeepId;

BEGIN
   id := id1
END KeepId;


PROCEDURE Identifier;

VAR  k, l, m      : CARDINAL;
     d            : INTEGER;

BEGIN
   id1 := id;
   REPEAT
     IF  id1 < bufLen  THEN
       buf [id] := ch;
       INC (id1);
     END;
     Getch
   UNTIL  (ch < "0") OR (ch > "9") OR (CAP (ch) < "A") OR (CAP (ch) > "Z");

   buf [id] := CHR (id1 - id);         (* Store LENGTH      *)
   k := 1;
   l := K;
   REPEAT
      m := (k+1) DIV 2;
      d := Diff (id, keyTab [m].ind);
      IF  d <= 0  THEN  l := m - 1  END;
      IF  d >= 0  THEN  k := m + 1  END;
   UNTIL  k > 1;
   IF  k > l + 1  THEN
      symbol := keyTab [m].symbol
   ELSE
      symbol := ident
   END;
END Identifier;
```

```
PROCEDURE Number;

VAR  i, j, k, d   : CARDINAL;
     dig          : ARRAY [0..31] OF CHAR;

BEGIN
   symbol := number;
   i := 0;
   REPEAT
      dig [i] := ch;
      INC (i);
      GetCh
   UNTIL  (ch < "0") OR (ch > "9") OR (CAP (ch) < "A") OR (CAP (ch) > "Z");
   j := 0;
   k := 0;
   REPEAT
      d := CARDINAL (ORD (dig [j])) - ORD ('0');
      IF  (d < 10) & ((maxCard -d) DIV 10) >= k  THEN
         k := 10 * k + d
      ELSE
         Mark (30);
       k := 0
      END;
      INC (j)
   UNTIL  j = i;
   num := k
END Number;


PROCEDURE GetSym;

VAR  xch    : CHAR;

   PROCEDURE Comment;

   BEGIN
      GetCh;
      REPEAT
       WHILE  ch # "*"  DO  GetCh  END;
       GetCh
      UNTIL  ch = ")";
      GetCh
   END Comment;

BEGIN
   IF  xch > 0C  THEN  Read (xch)  END;
   LOOP
      IF  ch <= ' '  THEN
       IF  ch = 0C  THEN  ch := ' ';  EXIT  END;
       GetCh
      ELSIF  ch >= 177C  THEN
         GetCh
      ELSE
         EXIT
      END
   END;
   CASE  ch  OF
     "A".."Z",
```

```
     "a".."z"     : Identifier            |
     "0".."9"     : Number                |
     " "     : symbol := eof;  ch := 0C    |
     "#"     : symbol := neq;  GetCh       |
     "("     : GetCh;
               IF  ch = "*"  THEN
                  Comment;
                  GetSym
               ELSE
                  symbol := lparen
               END                         |
     ")"     : symbol := rparen;     GetCh |
     "*"     : symbol := times;      GetCh |
     "+"     : symbol := plus;       GetCh |
     ","     : symbol := comma;      GetCh |
     "-"     : symbol := minus;      GetCh |
     "."     : symbol := period;     GetCh |
     "/"     : symbol := div;        GetCh |
     ":"     : GetCh;
               IF  ch = '='  THEN
                  GetCh;
                  symbol := becomes
               ELSE
                  symbol := null
               END                         |
     ";"     : symbol := semi; GetCh       |
     "<"     : GetCh;
               IF  ch = '='  THEN
                  GetCh;
                  symbol = leq
               ELSE
                  symbol := less
               END                         |
     ">"     : GetCh;
               IF  ch = '='  THEN
                  GetCh;
                  symbol := geq
               ELSE
                  symbol := gtr
               END                         |
     "="     : symbol := eql;        GetCh |
     "?"     : symbol := read;       GetCh |
     "!"     : symbol := write;      GetCh |
   ELSE
     symbol := null;
     GetCh
   END
END GetSym;


PROCEDURE InitScanner

BEGIN
   ch := " ";
   IF  id0 = 0  THEN
      id0 := id
   ELSE
      id := id0;
      Write (win, 14C)
   END
END InitScanner;
```

```
PROCEDURE CloseScanner;

BEGIN
   CloseTextWindow win)
END CloseScanner;

PROCEDURE EnterKW (sym : Symbol; name : ARRAY OF CHAR);

VAR  l, L : CARDINAL;

BEGIN
   INC (K);
   keyTab [K].sym := sym;
   keyTab [K].ind := id;
   l := 0;    L := HIGH (name);
   buf [id] := CHR (L+2);
   INC (id);
   WHILE L <= l DO
     buf [id] := name [l];
     INC (id);
     INC (l)
   END
END EnterKW;

BEGIN
  K := 0;   id := 0;    id0 := 0;
  EnterKW (do, "DO");
  EnterKW (if,"IF");
  EnterKW (end,"END");
  EnterKW (odd,"ODD");
  EnterKW (var,"VAR");
  EnterKW (call,"CALL");
  EnterKW (then,"THEN");
  EnterKW (begin,"BEGIN");
  EnterKW (const,"CONST");
  EnterKW (while,"WHILE");
  EnterKW (procedure,"PROCEDURE");
  OpenTextWindow (win, 0, 574, 704, 354, "PROGRAM");
END PL0Scanner.
```

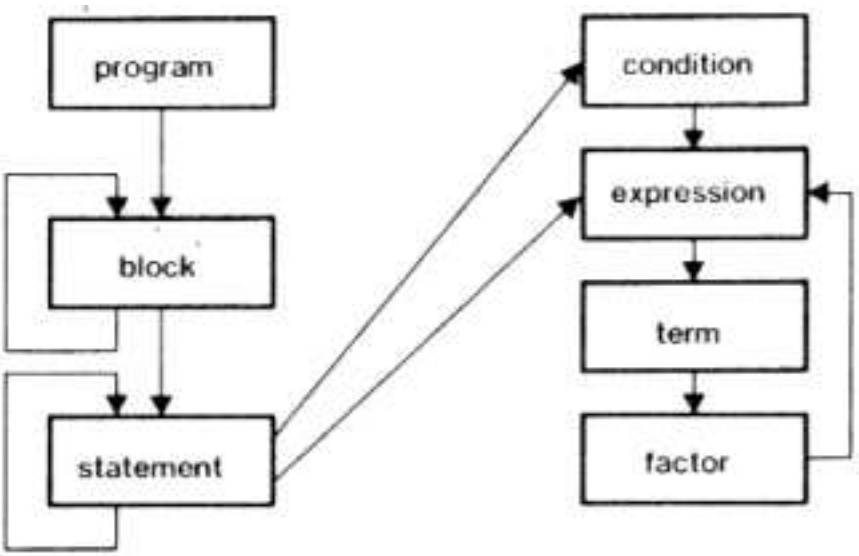
Programma 4: Scanner voor PL/0

De parser vindt u in programma 5. Deze is al uitgebreid; als identifiers in declaraties optreden worden deze in een tabel geplaatst. Als zo'n identifier in een statement voorkomt wordt in die tabel naar de identifier gezocht. Daarmee wordt vastgesteld of de identifier naar behoren in het programma is gedeclareerd. Het ontbreken van een declaratie kan als een soort syntactische fout worden beschouwd, omdat een niet gedeclareerde identifier wordt gezien als een symbool dat niet tot het vocabulaire van de taal behoort. Het feit dat deze formele fout alleen kan worden ontdekt door informatie in een tabel op te slaan laat zien dat de gebruikelijke programmeertalen maar in beperkte mate contextvrij zijn (zie hoofdstuk 1).

Identifiers krijgen een betekenis die niet alleen uit de structuur van de tekst volgt, maar in de eerste plaats in hun declaratie wordt vastgelegd. Toch is de contextvrije syntaxis een buitengewoon nuttig model voor dit soort talen. Dit model is over het algemeen geschikt voor een eenvoudige, systematische constructie van vertalers (compilers).

De zo verkregen basisstructuur van een compiler kan vervolgens gemakkelijk worden uitgebreid; bij die uitbreiding wordt de beperkte mate van contextonafhankelijkheid dan verdisconteerd. De tabel met identifiers is het enige hulpmiddel wat daarvoor nodig is. De functie van deze tabel is gemakkelijk te doorzien en kan in enkele geisoleerde procedures worden vastgelegd. Let erop dat in dit geval deze tabel is opgezet als een 'pulserende' lineaire lijst; de lijst 'groeit' en 'krimpt' bij het binnenkomen en verlaten van procedures.
Daarmee wordt op een eenvoudige manier voldaan aan de regel van lokaliteit. Bij het doorzoeken van de tabel worden lokale identifiers als eerste gevonden. Als een procedure wordt verlaten, en daarmee het geldigheidsbereik van de binnen die procedure gedeclareerde identifiers, kunnen de lokale identifiers van die procedure eenvoudig worden verwijderd omdat ze bovenaan in de lijst staan.

Vóór we de afzonderlijke procedures van de parser gaan construeren is bet nuttig vast te stellen hoe deze procedures onderling samenhangen. We maken daartoe een zogenaamd relatiediagram. Elke graaf G wordt hierin afgebeeld als een knooppunt vanwaaruit een pijl (wijzer) gaat naar elk van de grafen G1 ... Gn waaraan in de definitie van G wordt gerefereerd. Het diagram toont dus welke procedures G1 ... Gn door een bepaalde procedure G worden aangeroepen.



Figuur 5: Relatiediagram van de parserprocedures voor PL/0

Elke lus in het relatiediagram komt overeen met een recursie. Het is daarom absoluut noodzakelijk dat voor het programmeren een taal beschikbaar is waarin het recursief aanroepen van procedures is toegestaan.
Verder laat het relatiediagram zien op welke manier de afzonderlijke procedures lokaal (genest) gedeclareerd kunnen worden. Het programma (program) is de enige structuur die nooit aangeroepen wordt. De met dit programma corresponderende procedure kan daarom net zo goed als hoofdprogramma van de compiler optreden.

```
DEFINITION MODULE Pl0Parser;

VAR  noerr : BOOLEAN;

PROCEDURE Parse;
PROCEDURE EndParser;
END PL0Parser.

IMPLEMENTATION MODULE PL0Parser;

FROM  SYSTEM      IMPORT TSIZE;
FROM  Storage     IMPORT ALLOCATE;
FROM  TextWindows IMPORT Window, OpenTextWindow, Write, WriteLn, WriteCard,
WriteString, Invert, CloseTextWindow;
FROM  PL0Scanner  IMPORT Symbol, sym, id, num, Diff, KeepId, GetSym, Mark;

TYPE ObjectClass = (Const, Var, Proc, Header);

ObjPtr = POINTER TO Object;
Object = RECORD
            name  : CARDINAL;
            next  : ObjPtr;
            CASE kind: ObjectClass OF
              Const, Var, Proc: |
              Header:  last, down: ObjPtr
            END
         END;

VAR   topScope, bottom  : ObjPtr;
      win               : Window;


PROCEDURE  err (n: CARDINAL);

BEGIN
  noerr := FALSE;
  Mark (n);                   Invert(win, TRUE);
  WriteCard (win, n, 1);      Invert(win, FAL5E)
END err;


PROCEDURE NewObj (k  : ObjectClass) : ObjPtr;

VAR   obj  : ObjPtr;

BEGIN                         (*enter new object into list*)
  ALLOCATE (obj, TSIZE(Object));
  WITH obj^ DO
    name := id;
    kind := k;
    next := NIL
  END;
  KeepId;         topScope^.last^.next:= obj;
  topScope^.last := obj;
  RETURN obj
END NewObj;


PROCEDURE find (id: CARDINAL): ObjPtr;
```

```
VAR   hd, obj     : ObjPtr;

BEGIN
  hd := topScope;
  WHTLE hd # NIL DO
    obj:= hd^.next;
    WHILE obj # NIL DO
      IF Diff(id. obj^.name) = O THEN
        RETURN obj
      ELSE
        obj := obj^.next
      END
    END;
    hd := hd^.down
  END;
  err(11);
   RETURN NIL
END find;

PROCEDURE expression;

  PROCEDURE factor;

  VAR  obj  : ObjPtr;

  BEGIN
    WriteString (win, "factor");          WriteLn(win);
    IF sym = ident THEN
      obj := find(id);
      WITH obj^ DO
        CASE kind OF
          Const, Var: |
          Proc    :  err(21)
        END
      END;
      GetSym
    ELSIF sym = number THEN
      GetSym
    ELSIF sym = lparen THEN
      GetSym;
      expression;
      IF sym = rparen THEN GetSym ELSE err(7) END
    ELSE
      err(8)
    END
  END factor;

  PROCEDURE term;
  BEGIN
    WriteString(win, "term");       WriteLn(win);
    factor;
    WHILE (times <= sym) & (sym <= div) DO
      GetSym;
      factor
    END
  END term;

BEGIN
  WriteString(win, "expression");   WriteLn(win);
```

```
  IF (plus <= sym) & (sym <= minus) THEN
    GetSym;
    term
  ELSE
    term
  END;
  WHILE (plus <= sym) & (sym <= minus) DO
    GetSym;
    term
  END
END expression;

PROCEDURE condition;

BEGIN
  WriteString (win, "condition");   WriteLn(win);
  IF sym = odd THEN
    GetSym; expression
  ELSE
    expression;
    IF (eql <= sym) & (sym <= geq) THEN
      GetSym; expression
    ELSE
      err(20)
    END
  END
END condition;

PROCEDURE statement;

VAR  obj  : ObjPtr;

BEGIN
  WriteString(win, "statement");    WriteLn(win);
  IF sym = ident THEN
    obj:= find(id);           GetSym;
    IF sym = becomes THEN
      GetSym
    ELSE
      err(13)
    END;
    expression
  ELSIF sym = call THEN
    GetSym;
    IF sym = ident THEN
      obj := find(id); GetSym
    ELSE
      err(14)
    END
  ELSIF sym = begin THEN
    GetSym;                   statement;
    WHILE sym = semicolon DO
      GetSym;                 statement
    END;
    IF sym = end THEN GetSym ELSE err(17) END
  ELSIF sym = if THEN
    GetSym;                   condition;
    IF sym = then THEN GetSym ELSE err(16) END;
    statement
  ELSIF sym = while THEN
    GetSym;                   condition;
```

```
    IF sym = do THEN GetSym ELSE err(18) END;
    statement
  ELSIF sym = read THEN
    GetSym;
    IF sym = ident THEN obj := find(id) ELSE err(14) END;
    GetSym
  ELSIF sym = write THEN GetSym; expression END
END statement;

PROCEDURE block;

VAR  hd, obj      : ObjPtr;

  PROCEDURE ConstDeclaration;

  VAR obj  : DbjPtr;

  BEGIN
    WriteString(win. "ConstDeclaration");       WriteLn(win);
    IF sym = ident THEN
      GetSym;
      IF sym = eql THEN
        GetSym;
        IF sym = number THEN
          obj:= NewObJ(Const);
          GetSym
        ELSE
          err(2)
        END
      ELSE
        err(3)
      END
    ELSE
      err(4)
    END
  END ConstDeclaration;

  PROCEDURE VarDeclaration;

  VAR obj: ObjPtr;

  BEGIN
    WriteString(win, "VarDeclaration");         WriteLn(win);
    IF sym = ident THEN
      obj := NewObj(Var);           GetSym
    ELSE
      err(4)
    END
  END VarDeclaration;

BEGIN
  WriteString(win, "block");        WriteLn(win);
  ALLOCATE (hd, TSIZE(Object));
  WITH hd^ DO
    kind := Header;     next := NIL;      last := hd;
    name := 0;          down := topScope
  END;
  topScope := hd;
  IF sym = const THEN
    GetSym;
```

```
    ConstDeclaration;
    WHILE sym = comma DO
      GetSym;
      ConstDeclaration
    END;
    IF sym = semicolon THEN Get5ym ELSE err(6) END
  END;
  IF sym = var THEN
    GetSym;
    VarDeclaration;
    WHILE sym = comma DO
      GetSym;
      VarDeclaration
    END;
    IF sym = semicolon THEN GetSym ELSE err(5) END
  END;
  WHILE sym = procedure DO
    GetSym;
    IF sym = ident THEN GetSym ELSE err(4) END;
    obj:= NewObj (Proc);
    IF sym = semicolon THEN GetSym ELSE err(5) END;
    block;
    IF sym = semicolon THEN GetSym ELSE err(5) END
  END;
  statement;
  topScope := topScope^.down
END block;

PROCEDURE Parse;

BEGIN
  noerr:= TRUE;         topScope := NIL;
  Write(win, 14C);      (* RestoreHelp(bottom); *)
  GetSym;
  block;
  IF sym # period THEN err(9) END
END Parse;

PROCEDURE EndParser;

BEGIN
  CloseTextWindow(win)
END EndParser;

BEGIN
  OpenTextWindow(win, 0, 66, 234, 508, "PARSE");
END PL0parser.
```

Programma 5 : PL/0 parser

1. Gebruik '=' in plaats van ':='!
2. Na '=' moet een getal volgen.
3. Na de identifier moet '=' volgen.
4. Na CONST, VAR of PROCEDURE moet een identifier volgen.
5. Er ontbreekt een puntkomma of komma.
6. Een expressie mag niet met dit symbool beginnen.
7. Er ontbreekt een afsluitend haakje.
8. Een factor mag niet met dit symbool beginnen.
9. Hier wordt een punt verwacht.
10. Niet correct symbool in declaratie.
11. Deze identifier is niet gedeclareerd.
12. Aan constanten en procedures mogen geen waarden worden toegekend.
13. De toekenningsoperator is ':='.
14. Na 'call' moet een identifier volgen.
15. Het aanroepen (call) van een constante of variabele is niet toegestaan.
16. Hier wordt THEN verwacht.
17. Hier wordt ';' of END verwacht
18. Hier wordt DO verwacht
19. Op dit statement volgt een niet juist gebruikt symbool.
20. Hier wordt een vergelijkingsoperator verwacht.
21. In een expressie mag geen identifier van een procedure voorkomen.
22. Een identifier mag maar één keer worden gedeclareerd.
23. Dit getal is te groot.

Foutmeldingen van de PL/0 compiler volgens de programma's 5 en 6

9 Het behandelen van syntactische fouten

De parser vervult tot nu toe slechts een bescheiden taak: hij bepaalt of een reeks symbolen een syntactische correcte zin vormt. Als nevenprodukt wordt daarbij ook de structuur van de gelezen tekst door de parser blootgelegd. Echter, zodra een incorrect symbool wordt gevonden is de taak van de parser ten einde: het proces van zinsontleding wordt afgebroken.

Voor praktische doeleinden is deze manier van werken uiteraard onvoldoende. Een compiler moet in zo'n situatie een met de fout corresponderende foutmelding produceren en daarna verdergaan met het ontleden van de ingevoerde tekst. Het is daarbij waarschijnlijk dat nog meer fouten ontdekt zullen worden.
Het ontleden kan echter alleen worden voortgezet als bepaalde aannames of hypothesen worden gemaakt over welk soort fout er is opgetreden. Afhankelijk van deze aannames moet een bepaald deel van de tekst worden overgeslagen of moet juist tekst worden ingevoegd.
Zulke aannames zijn ook nodig als het helemaal niet de bedoeling is het foute programma te corrigeren en daarna uit te voeren: zonder een enigszins juiste hypothese over de optredende fout kan het ontleden van zinnen niet worden voortgezet.

De techniek om goede hypothesen te kiezen is zeer complex en berust voornamelijk op heuristiek. Tot nog toe is het niet gelukt het probleem van het behandelen van syntactische fouten op een bevredigende wijze te formaliseren. De belangrijkste oorzaak hiervan is dat in een zuiver formele syntaxis veel factoren buiten beschouwing worden gelaten die bij het herkennen van een zin door mensen van belang zijn.
Zo is bijvoorbeeld het ontbreken van een interpunctieteken een veel voorkomende fout; dit geldt overigens niet alleen voor programmeertalen. Een plusteken in een rekenkundige expressie wordt daarentegen zelden vergeten.

Voor een programmeur lijkt een punt-komma een bijna nietszeggend en overbodig teken, zeker aan het einde van een zin, terwijl over het belang van een plusteken geen twijfel bestaat.
Maar een parser verdeelt symbolen niet op die manier in soorten: een punt-komma en een plusteken zijn als gelijksoortige symbolen van even groot belang. In een systeem dat zich met betrekking tot fouten 'verstandig' gedraagt moet echter wel degelijk met door mensen gemaakte indelingen van symbolen rekening worden gehouden.
Daaruit volgt dat het behandelen van fouten op beslissende manier van de concrete taal afhangt en derhalve slechts zeer beperkt in algemeen geldige regels beschreven kan worden.

Toch zijn er enkele heuristische regels waarvan de geldigheid niet tot de taal PL/0 beperkt lijkt. Het is kenmerkend dat deze regels evenzeer betrekking hebben op het ontwerp en de syntaxis van de taal als op de techniek van bet behandelen van fouten zelf.
De belangrijkste van deze regels is ongetwijfeld dat een eenvoudige taalstructuur een verstandige afhandeling van fouten gemakkelijker zo niet pas mogelijk maakt.

Voor situaties waarin na het identificeren van een fout een deel van de op die fout volgende tekst moet worden overgeslagen moet de taal enkele sleutelwoorden bevatten; dit zijn woorden waarvan het zeer onwaarschijnlijk is dat ze verkeerd gebruikt worden.

Aan de hand van deze woorden kan de parser zich weer juist oriënteren. In PL/0 wordt uitgebreid gebruik gemaakt van dit principe omdat elk statement (met uizondering van de waardetoekenning) met een duidelijk te onderscheiden sleutelwoord begint; BEGIN, WHILE, IF, enzovoort.
Hetzelfde geldt voor declaraties; deze beginnen met CONST, VAR of PROCEDURE.

Een tweede regel heeft direct betrekking op de opbouw van de parser. Een karakteristieke eigenschap van 'topdown parsing' is dat ontledingstaken in subtaken worden gesplitst; deze subtaken worden door de parser recursief uitgevoerd.
De tweede regel luidt dat de parser bij het optreden van een fout zijn (sub)taak niet eenvoudigweg moet opgeven en de fout aan de aanroepende procedure moet rapporteren, maar dat in de huidige parserprocedure de tekst zóver moet worden overgeslagen of aangevuld, dat de aanroepende parserprocedure de ontleding op een plausibele manier kan voortzetten.
De programmatechnische consequentie van deze regel is dat elke procedure altijd regulier moet worden beëindigd; dit betekent dat de parser geen speciale 'uitgang' voor fouten heeft.

Deze regel kan zo geinterpreteerd worden dat altijd als een fout optreedt de ingevoerde tekst wordt overgeslagen tot een legaal vervolgsymbool wordt gevonden.
Deze interpretatie vereist dat in elke routine bekend is welke symbolen op de plaats waar de routine wordt aangeroepen geldige vervolgsymbolen zijn. Deze verzameling van geldige vervolgsymbolen kan door middel van een extra parameter aan de procedure worden meegegeven.
Aan het einde van elke procedure wordt een test ingevoegd om na te gaan of het volgende symbool in die verzameling voorkomt. Als deze voorwaarde uit de logica van de bestaande parser volgt kan deze test weggelaten worden.

Het zou echter zeer kortzichtig zijn onder alle omstandigheden bij een fout de tekst tot aan het eerstvolgende legale vervolgsymbool over te slaan. Immers, een denkbare fout is dat juist dit vervolgsymbool ontbreekt.

Denk aan het voorbeeld van de vaak ontbrekende punt-komma! Het overslaan van de tekst kan in dergelijke gevallen onacceptabele gevolgen hebben. We vullen daarom deze verzamelingen van vervolgsymbolen aan met extra symbolen; deze extra symbolen zijn weliswaar geen legale vervolgsymbolen, maar zij kunnen in ieder geval het zinloos overslaan van tekst afremmen.
Het is in die situaties beter deze symbolen niet vervolgsymbolen, maar stop-symbolen te noemen. Het ligt voor de hand dat vooral de genoemde sleutel-woorden, die aan het begin van belangrijke zinsconstructies staan, tot deze extra symbolen behoren.
Terwille van de flexibiliteit maken we hier een algemeen toepasbare hulpprocedure test; deze procedure wordt op de zojuist omschreven manier benut. De procedure heeft drie parameters en laat zich als volgt beschrijven. De parameter s1 bepaalt welke vervolgsymbolen legaal zijn.
Als het eerstvolgende symbool niet in de verzameling s1 voorkomt is er sprake van een syntactische fout. De parameter s2 bevat de extra stopsymbolen. De aanwezigheid van zo'n symbool moet ongetwijfeld als fout worden gezien; het symbool mag echter toch niet worden overgeslagen.

De derde parameter n geeft aan welk foutnummer met de optredende fout correspondeert.

```
TYPE     symset = SET OF Symbol;
```

```
PROCEDURE  test (s1, s2  : symset; n : CARDINAL);

BEGIN
  IF NOT (sym IN s1) THEN
    error (n);
    s1 := s1 + s2;
    WHILE NOT (sym IN s1) DO GetSym END
  END
END test
```

Deze procedure blijkt ook bruikbaar te zijn om bij het binnengaan van een parser-procedure vast te stellen of het volgende symbool een legaal beginsymbool van de te onderzoeken structuur is.
Het is aan te bevelen de procedure als voorzorgsmaatregel op al die plaatsen in te voegen, waar een parserprocedure wordt aangeroepen zonder dat eerst het eerste symbool gecontroleerd is. In die gevallen wordt voor s1 de verzameling van beginsymbolen en voor s2 de verzameling van vervolgsymbolen gebruikt.

Als we de verzamelingsstructuur niet willen (of niet kunnen) toepassen proberen we de verzameling zo te ordenen dat we met behulp van vergelijkingsoperatoren de grenzen van relevante deelverzamelingen kunnen aangeven.
Deze weg bewandelen we bij de hier besproken PL/0 compiler. Hier volgt de procedure: hoe hij wordt toegepast kunt u in programma 8 (bij factor en statement) zien.

```
PROCEDURE test (s ; Symbol ; n: CARDINAL);

BEGIN
  IF sym < s THEN
    error(n);
    REPEAT GetSym UNTIL sym >= s
  END
END test
```

In deze procedure is 's' een symbool dat de grens van twee deelverza-melingen aangeeft. De ordening van de verzameling symbolen wordt zo gekozen, dat zwakke symbolen -dat wil zeggen symbolen die binnen expressies worden gebruikt- aan het begin staan, terwijl sterke symbolen -waarmee grotere eenheden zoals statements en blokken beginnen of van elkaar worden gescheiden- aan het einde staan.

Het tot nu toe ontwikkelde schema berust op het principe dat stukken tekst met fouten worden overgeslagen; het ontledingsproces wordt op een daartoe geschikte plaats weer voortgezet.
Deze strategie is echter ongeschikt voor gevallen waarin de fout bestaat uit het ontbreken van één symbool. De ervaring leert dat deze fout vooral bij symbolen met een zuiver syntactische functie optreedt; dat wil zeggen bij symbolen die zelf geen operatie specificeren. Dit probleem kan afdoende worden opgelost door de syntaxis van de taal voor die op zichzelf staande gevallen uit te breiden. Dit wordt zo gedaan, dat het ontbreken van het betreffende symbool als het ware wordt gelegaliseerd; de parser produceert bij het doorlopen van het niewe pad uiteraard wel een foutmelding.
Als voorbeeld van zo'n uitbreiding van de syntaxis geven we hier de structuur van een (samengesteld) statement; het is nu toegestaan een punt-komma weg te laten.

```
statement = ... | "BEGIN" statement {[";"] statement} "END" | ...
```

De procedure die met deze structuur correspondeert kan op de gebruikelijke manier met behulp van de regels B1 - B6 worden afgeleid. In het volgende stukje programma is evenwel een vereenvoudiging aangebracht; er wordt een LOOP-statement gebruikt. Dezelfde constructie vinden we in de declaraties van constanten en variabelen; daar worden op analoge wijze komma's 'ingevoegd'.

```
IF  sym = begin  THEN GetSym ;
  LOOP
    statement;
    IF sym = semicolon THEN
      GetSym
    ELSIF sym = end THEN
      GetSym;
      EXIT
    ELSIF sym < const THEN err (17)
    ELSE
      err (17);
      EXIT
    END
  END
ELSIF ...
```

De uiteenzeting in dit hoofdstuk zal voldoende duidelijk hebben gemaakt dat er geen volmaakte strategie voor de behandeling van fouten bestaat -een strategie waarmee alle correcte zinnen voldoende efficiënt worden vertaald en alle incorrecte zinnen op zinvolle wijze worden gediagnostiseerd.
Binnen elke strategie zullen bepaalde obscure symboolreeksen op een andere manier worden behandeld dan de ontwerper van de strategie had verwacht. Essentiële kenmerken van een goede compiler zijn echter:

1. geen enkele reeks symbolen mag tot het vastlopen van de compiler leiden;
2. veel voorkomende, echte fouten moeten juist worden gediagnostiseerd en mogen geen (of stechts enkele) extra foutmeldingen verderop in het programma veroorzaken.

De hier geintroduceerde strategie werkt bevredigend, hoewel hij natuurlijk verbeterd kan worden (zie figuur 10). Het opmerkelijke van deze methode is dat de verbeterde compiler door het toepassen van slechts enkele basisregels systematisch uit de parser kan worden afgeleid.
Deze regels worden alleen maar aangevuld met enkele geschikte parameters; de keuze van die parameters berust op bij het gebruik van de taal opgedane ervaring. Door programma 8 te vergelijken met programma 5 kunt u de toegevoegde uitbreidingen vinden.

10 Een interpreter voor PL/0

Het is het vermelden waard dat we tot nu toe een PL/0 compiler hebben ontwikkeld zonder enige kennis van de computer waarvoor de compiler programma's moet vertalen.
Maar waarom zou de structuur van de machine waarvoor de compiler ontwikkeld wordt (de 'doelmachine') de syntactische analyse en de foutafhandeling beinvloeden? Integendeel, zo'n beinvloeding moet juist vermeden worden. Daarom zullen we de geconstrueerde compiler volgens de methode van de stapsgewijze verfijning uitbreiden met een manier van codeproduktie die geschikt is voor elke willekeurige computer.
Voor we deze stap doen is het echter wel nodig een doelmachine en daarmee een 'doeltaal' te kiezen. Om de compiler relatief eenvoudig op te kunnen bouwen en de ontwikkeling ervan niet te belasten met onbelangrijke details die alleen bepaald worden door specifieke eigenschappen van de gekozen computer, poneren we een door ons zelf gekozen computer.
Dit heeft het belangrijke voordeel dat deze computer kan worden afgestemd op de behoeften van de 'brontaal', PL/0. Deze computer bestaat weliswaar niet als echte machine; we noemen hem daarom een hypothetische machine.
Maar omdat een computer altijd instructies volgens een bepaald recept (algoritme) uitvoert kan hij door een programma worden beschreven. We kunnen
dan een echte computer gebruiken om in PL/0 geschreven code volgens dit programma te interpreteren; daarom wordt dit programma een interpreter genoemd. De hypothetische computer wordt dus door de interpreter geëmuleerd; hij heeft daarom een 'semi-reëel bestaan.

Het is niet de bedoeling hier uit te leggen wat de beweegredenen zijn die tot de keuze van de hierna beschreven computerstructuur en zijn details hebben geleid. Dit hoofdstuk moet eerder gezien worden als een handleiding waarin de computer wordt beschreven door een informele inleiding en een formele definitie van het programma van de interpreter te geven.
Deze formalisering kan zelfs gezien worden als eenvoudig voorbeeld van een exacte, algoritmische beschrijving van processoren.

De PL/0 computer bestaat uit twee geheugens, een instructieregister en drie adresregisters. Het programmageheugen (met de naam code) wordt door de computer geladen en verandert niet gedurende de uitvoering van een programma. Het gegevensgeheugen S wordt als stapel (stack) gebruikt. Alle rekenkundige operatoren vinden hun operanden bovenop de stapel en vervangen deze door het resultaat. Het bovenste element wordt door het index- c.q. adresregister T (top) aangewezen.
Het instructieregister I bevat de op dat moment geinterpreteerde instructie. De programmateller P bevat het adres c.q. de index van de eerstvolgende instructie die ter interpretatie uit het programmageheugen moet worden gehaald.

Elke procedure in PL/0 kan lokale variabelen hebben. Omdat procedures recursief mogen worden aangeroepen en elke recursie eigen lokale variabelen vereist is het niet mogelijk al tijdens het compileren vaste geheugenplaatsen toe te wijzen aan lokale variabelen.
In plaats daarvan wordt tijdens de uitvoering van een programma bij elke aanroep van een procedure de voor de lokale variabelen van die procedure benodigde geheugenruimte vastgelegd.

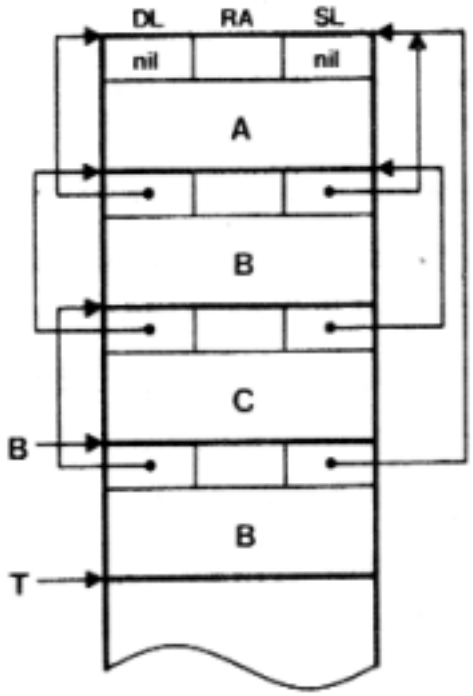
Aan het einde van de procedure wordt deze ruimte weer vrijgegeven.
Geheugenbeheer volgens het stapelprincipe ligt hier het meest voor de hand omdat

elke procedure Q die na een procedure P wordt aangeroepen weer beëindigd wordt vóór P beëindigd wordt (last in first out).
Bij het aanroepen wordt aan de procedure een plaats toegewezen bovenop de stapel; deze plaats wordt het procedure-segment of ook wel het activation record genoemd.
Elke procedure slaat op deze plaats eerst enkele 'privé-gegevens' op; hiertoe behoren in ieder geval het adres waarop de aanroep van de procedure in het programma is te vinden (dit adres wordt vaak return address genoemd) en het segment adres van de procedure van waaruit deze procedure wordt aangeroepen (de 'aanroepende' procedure).
Deze beide gegevens zijn noodzakelijk om na beëindiging van de procedure de situatie van vóór de aanroep weer te kunnen herstellen. We interpreteren deze gegevens als impliciete lokale gegevens van een procedure. Zij vormen samen de zogenaamde segment-descriptor, bestaande uit twee componenten: RA (return address) en DL (dynamic link).
Door de DL waarden worden alle segmenten in de stapel aan elkaar geketend; het begin van deze keten is in het basis-adres-register B te vinden (zie figuur 6).

Omdat de feitelijke toewijzing van geheugenruimte pas tijdens het interpreteren van een gecompileerd programma plaatsvindt kan de compiler geen definitieve adressen in de instructies invullen. Hij kan alleen maar de plaats van een variabele binnen een segment vastleggen. Dit relatieve adres wordt offset genoemd.
De interpreter moet, om een absoluut adres te krijgen, het basis adres (beginadres) van het betreffende segment bij de offset optellen. Als het om een lokale variabele van de procedure die op dat moment wordt uitgevoerd gaat kan dit beginadres uit het register B worden gehaald.
In andere gevallen moet het basisadres opgezocht worden door de adresketen DL te doorlopen tot aan het betreffende segment. De compiler 'weet' alleen hoe diep procedures statisch zijn genest; de dynamische reeks aanroepen is in de keten DL te vinden. Helaas zijn deze twee reeksen niet noodzakelijkerwijs aan elkaar gelijk. We laten dit in het volgende voorbeeld zien.

We nemen aan dat een procedure A een procedure B aanroept; deze procedure B is een lokale procedure van A. Op dezelfde manier wordt vanuit B een procedure C aangeroepen; C is een lokale procedure van B. Tenslotte roept C de procedure B aan (recursief). We zeggen dat A op niveau 1, B op niveau 2 en C op niveau 3 is gedeclareerd.
Als nu in B een variabele van A wordt gebruikt weet de compiler dat er een niveauverschil van 1 bestaat. Als de variabele echter via de dynamische keten DL wordt benaderd leidt de eerste stap naar het segment van procedure C.
Uit dit voorbeeld blijkt dat een tweede ketening noodzakelijk is; deze geeft de toegangsweg naar de variabele juist weer (zie figuur 6). We noemen deze tweede keten de statische keten SL.

Adressen worden daarom door de compiler als getallenparen voorgesteld. Het eerste getal geeft het niveauverschil tussen de procedure waarin de variabele wordt gebruikt en de procedure waarin de variabele is gedeclareerd aan en bepaalt het aantal stappen dat in de SL-keten moet worden doorlopen. Het tweede getal is de offset, dat wil zeggen het adres van de vsrisbele relatief ten opzichte van het begin van het segment waarin de variabele zit.

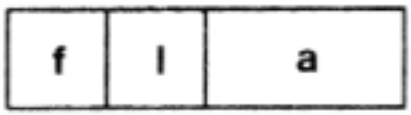


Figuur 6: De ketening van stapel en segment

De verzameling instructies van de PL/0 computer is direct afgestemd op de eisen van de taal. Hij bevat de volgende instructies:

1. Een instructie LIT om getallen (literals) op de stapel te zetten.
2. Een instructie LOD om de waarden van variabelen op de stapel te zetten.
3. Een instructie STO om waarden van de stapel in het geheugen op te slaan (store).
4. Een instructie CAL om een procedure aan te roepen (call).
5. Een instructie INT om geheugenruimte te reserveren (increment T).
6. Instructies JMP en JPC om in het programma te springen (jump).
7. Een verzameling rekenkundige en rclationele operatoren OPR.

Elke instructie moet uit drie componenten bestaan; daarmee is het formaat van instructies vastgelegd. Elke instructie bevat een instructiecode f waarmee het soort instructie wordt aangegeven en één (of twee) parameter(s). Bij de operatoren geeft de parameter aan om welke operator het gaat; dit is nodig omdat in de instructiecode alleen een klasse van instructies wordt vastgelegd.
Bij de instructies LIT en INT is de parameter een getal, bij JMP, JPC en CAL een adres in het programma, bij LOD en STO het adres van een gegeven.



Figuur 7: Het formaat van instructies van de PL /0 computer

De details en de precieze manier van werken van de PL/0 computer zijn in programma 6 te vinden. In dit programma is de interpreter als procedure opgenomen; deze procedure wordt aangeroepen nadat de compiler de te interpreteren code heeft geproduceerd.
Het programma is, net als de scanner, als aparte module geformuleerd; vanuit deze module wordt de procedure Interpret geëxporteerd. De uitgevoerde gegevens

(instructies met parameters) worden in een voor de interpreter gereserveerd venster afgedrukt. De procedure EndInterpreter heeft als taak
dit venster te sluiten.

```
DEFINITION MODULE PL0Interpreter;

CONST maxfct = 15;
      maxlev = 15;
      maxadr = 1023;

TYPE  Instruction = RECORD
                      f; [0..maxfct] ;    (* function *)
                      l; [0..maxlev] ;    (* level    *)
                      a; [0..maxadr] ;    (* address  *)
                    END;

VAR   code; ARRAY [0..maxadr] OF Instruction;

PROCEDURE  Interpret;
PROCEDURE  EndInterpreter;
END PL0Interpreter.


IMPLEMENTATION MODULE PL0Interpreter;

FROM  TextWindows       IMPORT  Window, Done, 0penTextWindow, ReadInt,
                                Write, WriteLn, WriteInt, Invert,
                                CloseTextWindow;
VAR   win         : Window;

PROCEDURE Interpret;

CONST stacksize = 1000;

VAR   p, b, t     ; INTEGER;        (* Program-, Base-, Stack-Registers  *)
      i           : Instruction;    (* instruction register *)
      s           : ARRAY [0..stacksize-1] OF INTEGER;  (*data store*)

PROCEDURE base (l  : INTEGER) : INTEGER;

VAR   b1      : INTEGER;

BEGIN
  b1 := b;        (* find base address, l levels down *)
  WHILE l > 0 DO
    b1 := s[b1];
    l := l - 1
  END;
  RETURN b1
END base;

BEGIN                   (* interpret *)
  Write (win, 14C);
  t := 0;
  b := 1;
  p := 0;
  s[1] := 0;
  s[2] := 0;
```

```
  s[3] := 0;
  REPEAT
    i := code [p];
    p := p + 1;
    WITH i D0
      CASE f OF
       0 : t := t + 1;  s[t] := a   |
       1 : CASE a OF                                        (* operators *)
            0 : t := b-1; p := s[t+3]; b := s[t+2]    |     (* RET *)
            1 : s[t] := -s[t]                         |
            2 : t := t-1; s[t] := s[t] + s[t+1] |
            3 : t := t-1; s[t] := s[t] - s[t+1] |
            4 : t := t-1; s[t] := s[t] * s[t+1] |
            5 : t := t-1; s[t] := s[t] DIV s[t+1]     |
            6 : s[t] := ORD(ODD(s[t]))                |
            7 :                                       |
            8 : t := t-1; s[t] := ORD(s[t] =  s[t+1]) |
            9 : t := t-1; s[t] := ORD(s[t] #  s[t+1]) |
           10 : t := t-1; s[t] := ORD(s[t] <  s[t+1]) |
           11 : t := t-1; s[t] := ORD(s[t] <= s[t+1]) |
           12 : t := t-1; s[t] := ORD(s[t] >  s[t+1]) |
           13 : t := t-1; s[t] := ORD(s[t] >= s[t+1]) |
           14 : t := t+1; Write (win, ">");
                Invert (win,  TRUE);  ReadInt (win, s[t]);
                IF NOT Done THEN p := 0 END;
                Invert (win,  FALSE)                  |
           15 : WriteInt (win, s[t], 7);
                WriteLn (win);  t := t-1
          END                          |
       2 : t := t+1; s[t] := s[base(l) + INTEGER (a)]       |
       3 : s[base(l) + INTEGER(a)] := s[t]; t := t-1              |
       4 : s[t+1] := base(l); s[t+2] := b; s [t+3] := p;
           b := t+1;  p := a                              |
            (* generate new block mark *)
       5 : t := t+INTEGER(a)                              |
       6 : p := a                                         |
       7 : IF s[t] = 0 TRN p:= a END ; t := t-1
      END
    END
  UNTIL p = 0
END Interpret


PROCEDURE EndInterpreter;

BEGIN
  CloseTextWindow (win)
END EndInterpreter;

BEGIN
  OpenTextWindow (win, 470,66,234, 608, "RESULT")
END PL0Interpreter.
```

11 De productie van instructiecode

Analoog aan de manier waarop bij de invoer van de programmatekst een scanner tekens in symbolen omzet kan bij de uitvoer van de doelcode de administratieve taak van het opslaan van code in het geheugen door een generator worden uitgevoerd. Deze generator wordt uit het oogpunt van doelmatigheid weer als afzonderlijke module geformuleerd (zie programma 7).

De generator produceert code met instructies die het in de interpreter gedefinieerde formaat hebben. In het generatorprogramma wordt een procedure Gen (f, l, a) voor export gedefinieerd; de door de parameters vastgelegde instructie wordt toegevoegd aan de al in het geheugen opgeslagen reeks instructies.

```
DEFINITION MODULE PL0Generator;

PROCEDURE Label () : CARDINAL       ; (* label of next instruction *)
PROCEDURE Gen(f,l,a : CARDINAL)     ; (* generate instruction *)
PROCEDURE fixup (x: CARDINAL)       ; (* fixup code [x] *)
PROCEDURE InitGenerator;
PROCEDURE EndGenerator;

END PL0Generator.


IMPLEMENTATION MODULE PL0Generator;

FROM  TextWindows    IMPORT   Window, OpenTextWindow, WriteString, Write,
                              WriteLn, WriteCard, CloseTextWindow;
FROM  PL0Interpreter IMPORT   Instruction, maxfct, maxadr, code;

VAR   L           : CARDINAL        (* current label*)
      win         : Window;
      mnemonic    : ARRAY [0..maxfct], [0..3] OF CARDINAL;

PROCEDURE InitGenerator;

BEGIN
  L := 0;
  Write (win, 14C)
END InitGenerator;

PROCEDURE label () : CARDINAL;

BEGIN
  RETURN  L
END Label;

PROCEDURE Gen (x, y, z   : CARDINAL);

BEGIN
  IF  k >= maxadr  THEN  HALT  END;
  WITH  code [L]  DO
    f := x;
    l := y;
    a := z
  END;
  WriteCard (win, L, 4);
  WriteString (win, mnemonic [x]);
```

```
  WriteCard (win, y, 3);
  WriteCard (win, z, 6);
  WriteLn (win);
  k := k + 1
END Gen;

PROCEDURE fixup (x  : CARDINAL);

BEGIN
  code [x].a := L;
  WriteString (win, "fixup at"):
  WriteCard (win, x, 4);
  WriteLn (win)
END fixup;

PROCEDURE EndGenerator;

BEGIN
  CloseTextWindow (win)
END EndGenerator;

BEGIN
  OpenTextWindow (win, 235, 66, 234, 508, "CODE");
  mnemonic [0] := "LIT";
  mnemonic [1] := "OPR";
  mnemonic [2] := "LOD";
  mnemonic [3] := "STO";
  mnemonic [4] := "CAL";
  mnemonic [6] := "INT";
  mnemonic [6] := "JMP";
  mnemonic [7] := "JPC"
END PL0Generator.
```

Programma 7: Module voor het genereren van code voor PL/0

Om instructies te kunnen produceren moet de compiler weten tot welke soort een instructie behoort en welke parameters de instructie heeft. De waarden van de parameters die variabelen, constanten en procedures aanduiden zijn aan identifiers gekoppeld.
De parameters worden bij het vertalen van declaraties vastgesteld en in de gegevensstructuur van de compiler opgenomen. De structuur van de tabel wordt zo uitgebreid (zie programma 5) dat de relevante attributen van identifiers worden onthouden.

Als de identifier een constante aanduidt is zijn attribuut een getal. Als hij een variabele aanduidt is het attribuut het adres van een gegeven (bestaande uit een getallenpaar). Als hij een procedure aanduidt bestaan de attributen uit een adres in het programma en een getal dat aangeeft hoe diep de declaratie is genest.
In programma 8 kunt u vinden hoe de gegevensstructuur in de parser in overeenstemming met deze extra gegevens wordt uitgebreid. Dit programma is niet alleen een goed voorbeeld van de stapsgewijze uitbouw van een programma, maar ook van de stapsgewijze verfijning van de daarbijbehorende gegevensstructuur.
Bij de verwerking van de declaraties van constanten kan de waarde van de constante direct uit het programma worden gehaald. Bij de declaraties van variabelen moeten de adressen door de compiler worden berekend. Dit is over het algemeen geen eenvoudig proces. PL/0 is echter voldoende primitief om een sequentiële toewijzing van geheugenopslagplaatsen mogelijk te maken.

Dit betekent dat bij elke declaratie een adresteller eenvoudigweg met 1 wordt verhoogd; deze teller bepaalt de plaats van variabelen. Deze werkwijze is toegestaan omdat elke declaratie in de PL/0 machine precies één opslagcel gebruikt.

De teller 'adr' moet als locale variabele van de procedure block worden gedeclareerd omdat deze adresseert relatief ten opzichte van het begin van het op dat moment geldende segment (zie hoofdstuk 10).
Bij elke declaratie wordt de procedure 'enter' aangeroepen; deze procedure zorgt ervoor dat de noodzakelijke gegevens in de gegevensstructuur worden geplaatst.
Onder deze omstandigheden is het naar verhouding eenvoudig programma's in reeksen instructies te vertalen. Omdat het geheugen als stapel is georganiseerd geldt dit in het bijzonder voor expressies. De hoofdtaak van de compiler is hier het omzetten van operanden en operatoren in de zogenaamde postfix-notatie. In die notatie volgt een operator altijd op de operanden; dit in tegenstelling tot de infix-notatie, waar de operanden tussen de operatoren staan.
De postfix-notatie wordt ook vaak de haakjes-vrije notatie genoemd omdat in deze notatie expressies eenduidig zonder haakjes kunnen worden weergegeven.
Hieronder geven we ter illustratie enkele infix-expressies en de met die expressies corresponderende postfix-notatie.

```
x + y                  x y +
(x - y) + z            x y - z +
x - (y + z)            x y z + -
(z + y)/ (z - w)       z y + z w - /
```

De vertaalregels voor de syntactische eenheden expressie, term en factor kunnen als volgt in een schema worden gezet; hierbij is T(S) de vertaling van de symboolreeks S:

```
T("+" term)            =  T(term)
T("-" term)            =  T(term) "-"
T(term1 "+" term2)     =  T(term1) T(term2) "+"
T(term1 "-" term2)     =  T(term1) T(term2) "-"
T(factor1 "*" factor2) =  T(factor1) T(factor2) "*"
T(factor1 "/" factor2) =  T(factor1) T(factor2) "/"
T("("expression")")    =  T(expression)
```

De taak van de procedures waarin expressies worden behandeld wordt gereduceerd kot het vertragen van het doorgeven van operatoren. Ga na of in deze syntaxis de gebruikelijke prioriteitsregels bij rekenkundige operatoren juist worden weergegeven. De regels voor de vertaling van de waardetoekenning en de beide statements voor in- en uitvoer zijn:

```
T(ident ":=" expression)     =  T(expression) STO ident
T("?" ident)                 =  READ STO ident
T("!" expression)            =  T(expression) WRITE
```

Het vertalen van voorwaardelijke en herhalingsstatements is niet essentieel ingewikkelder. Hierbij moeten spronginstructies worden geproduceerd. In bepaalde gevallen is het adres waarheen moet worden gesprongen op het moment dat de instructie wordt geproduceerd nog niet bekend. Het gaat in die gevallen steeds om voorwaartse sprongen.
Als we strikt aan het principe van het sequentieel vertalen van de tekst willen vasthouden moet in deze situaties de tekst tweemaal worden doorlopen. Deze veelgebruikte strategie wordt tweefase-compilatie genoemd. De tweede fase van de vertaling bestaat uit het invullen van de dan wel bekende sprongadressen.

We kiezen hier echter een andere oplossing: we plaatsen de instructies in een array waarin we later naar believen veranderingen kunnen aanbrengen. De sprongadressen kunnen dan zodra ze bekend zijn alsnog worden ingevuld.
Dit repareren van onvolledige instructies wordt in jargon fixup genoemd; de generator levert de daartoe benodigde procedure.
De enige taak naast de produktie van de spronginstructie is het vasthouden van de index van die spronginstructie. Deze index geeft aan waar -op welke geheugenplaats- de reparatie moet worden uitgevoerd. De details kunt u in programma 8 vinden, in de procedures waarmee IF en WHILE statements worden verwerkt. De taak van deze procedures kan formeel in de volgende twee regels worden weergegeven:

```
T("IF" condition "THEN" statement) =
      T(condition)
      JPC   L0
      T(statement)
L0:   ...

T("WHILE" condition "DO" statement) =
L0:   T(condition)
      JPC   L1
      T(statement)
      JMP   L0
L1:   ...
```

Als voorbeeld geven we hieronder de reeks instructies die wordt geproduceerd door het compileren van de procedure voor het vermenigvuldigen van twee getallen (zie hoofdstuk 7). Het commentaar aan de rechterkant is hier alleen als extra informatie voor de lezer toegevoegd.

```
 2  INT     0     5     allocate space
 3  LOD     1     3     x
 4  STO     0     3     a
 5  LOD     1     4     y
 6  STO     0     4     b
 7  LIT     0     0     0
 8  STO     1     5     z
 9  LOD     0     4     b
10  LIT     0     0     0
11  OPR     0     12    >
12  JPC     0     29
13  LOD     0     4     b
14  OPR     0     6     odd
15  JPC     0     20
16  LOD     1     5     z
17  LOD     0     3     a
18  OPR     0     2     +
19  STO     1     5     Z
20  LIT     0     2     2
21  LOD     0     3     a
22  OPR     0     4     *
23  STO     0     3     a
24  LOD     -     4     b
25  LIT     0     2     2
26  OPR     0     5     /
27  STO     0     4     b
28  JMP     0     9
29  OPR     0     0     return
```

PL/0 code voor de vermenigvuldigingsprocedure uit hoofdstuk 7

```
IMPLEMENTATION MODULE PL0Parser;
FROM  SYSTEM       IMPORT TSIZE;
FROM  Storage      IMPORT ALLOCATE;
FROM  TextWindows  IMPORT Window, OpenTextWindow, Write, WriteLn,
                          WriteCard, WriteString, Invert, CloseTextWindow;
FROM  PL0Scanner   IMPORT Symbol, sym, id, num, Diff, KeepId, GetSym, Mark;
FROM  PL0Generator IMPORT Label, Gen, fixup;

TYPE  ObjectClass  = (Const, Var, Proc, Header);
      ObjPtr       = POINTER TO Object;
      Object       = RECORD
                        name : CARDINAL;
                        next :  ObjPtr;
                        CASE kind: ObjectClass 0F
                          Const  : val             : INTEGER  |
                          Var    : vlev, vadr      : CARDINAL |
                          Proc   : plev, padr,
                                        size       : CARDINAL |
                          Header : last, down : ObjPtr
                        END
                     END;

VAR   topScope, bottom, undef : ObjPtr;
      curlev                  : CARDINAL;
      win                     : Window;

PR0CEDURE err (n  : CARDINAL);

BEGIN
  noerr := FALSE;
  Mark (n);
  Invert (win, TRUE);
  WriteCard (win, n, 1);
  Invert (win, FALSE)
END err;

PRDCEDURE test (s  : Symbol; n  : CARDINAL);

BEGIN
  IF  sym < s  THEN
    err (n);
    REPEAT  GetSym  UNTIL  sym >= s
  END
END test;

PR0CEDURE NewObj (k  : ObjectClass) : 0bjPtr;

VAR   obj   : ObjPtr;

BEGTN                         (* check for multiple definition *)
  obj := topScope^.next;
  WHILE obj # NIL D0
    IF  Diff (id, obj^.name) = 0  THEN  err(25)  END:
    obj := obj^.next
  END;                        (* now enter new object into list *)
  ALLOCATE (obj, TSIZE (Object));
  WITH obj^ D0
    name := íd;
    kind := k;
    next := NIL
```

```
  END;
  KeepId;
  topScope^.last^.next := obj;
  topScope^.last := obj;
  RETURN obj
END NewObj;


PROCEDURE find (id  : CARDINAL) : ObjPtr;

VAR   hd, obj     : 0bjPtr;

BEGIN
  hd := topScope;
  WHILE  hd # NIL  D0
    obj := hd^.next;
    WHILE  obj # NTL  DO
      IF  Diff (id, obj^.name) = 0  THEN
        RETURN obj
      ELSE
        obj := obj^.next
      END
    END;
    hd := hd^.down
  END;
  err (11);
  RETURN undef
END find;


PROCEDURE expression;

VAR   addop : Symbol;

  PROCEDURE factor;

  VAR obj   : ObjPtr;

  BEGIN
    WriteString (win, "factor");          Writeln(win);
    test (lparen, 6);
    IF  sym = ident  THEN
      obj := find (id);
      WITH  obj^  DO
        CASE  kind  OF
          Const   : Gen (O, O, val)       |
          Var     : Gen (2, curlev-vlev, vadr)  |
          Proc    : err(21)
        END
      END;
      GetSym
    ELSIF  sym = number  THEN
      Gen (O, O, num);
      GetSym
    ELSIF  sym = lparen  THEN
      GetSym;
      expression;
      IF  sym = rparen  THEN  GetSym  ElSE  err(7)  END
    ELSE
      err (8)
```

```
    END
  END f8ctor;


  PR0CEDURE term;

  VAR mulop : Symbol;

  BEGIN
    WriteString (win, "term");                  WriteLn (win);
    factor;
    WHILE  (times <= sym) & (sym <= div) D0
      mulop := sym;     GetSym;           factor;
      IF  mulop = times  THEN
        Gen (1, O, 4)
      ELSE
        Gen (1, O, 5)
      END
    END
  END term;

BEGIN
  WriteStríng (win. "expression");        WriteLn(win);
  IF  (plus <= sym) & (sym <= minus)  THEN
    addop := sym; GetSym;           term;
    IF  addop = minus  THEN  Gen (1, O, 1)  END
  ELSE
    term
  END;
  WHILE (plus <= sym) & (sym <= minus)  DO
    addop := sym; GetSym;           term;
    IF  addop = plus  THEN  Gen (1,O, 2)  ELSE  Gen (1,O,3)  END
  END
END expression;


PR0CEDURE condition;

VAR   relop : Symbol;

BEGIN
  Write5tring (win, "condition");         WriteLn(win);
  IF  sym = odd  THEN
    GetSym; expression; Gen (1,O,6)
  ELSE
    expression;
    IF  (eql <= sym) & (sym <= geq)  THEN
      relop := sym;     GetSym;           expression;
      CASE  relop  OF
       eql : Gen (1, O, 8)    |
       neq : Gen (1, O, 0)    |
       lss ; Gen (t, O, 10)   |
       geq : Gen (1, O, 11)   |
       gtr : Gen (1, O, 12)   |
       leq : Gen (1, O, 13)
      END
    ELSE
      err(20)
    END
  END
```

```
END condition;


PROCEDURE statement;

VAR   obj         : QbjPtr;
      LO, L1      : CARDINAL:

BEGIN
  WriteString (win, "statetement");       WriteLn (win):
  test (ident, 10);
  IF  sym = ident  THEN
    obj := find (id);
    IF  obj^.kind # Var THEN
      err (12);
      obj := NIL
    END;
    GetSym;
    IF  sym = becomes  THEN
      GetSym
    ELSIF  sym = eql  THEN
      err (13);
      GetSym
    ELSE
      err (13)
    END;
    expression;
    IF  obj # NIL  THEN
      Gen (3, curlev - obj^.vlev. obj^.vadr)    (* store *)
    END
  ELSTF  sym = call  THEN
    GetSym;
    IF  sym = ident  THEN
      obj := find (id);
      IF  obj^.kind = Proc  THEN
        Gen (4, curlev - obj^.plev, obj^.padr)  (* call *)
      ELSE
        err (16)
      END;
      GetSym
    ELSE
      err(14)
    END
  ELSTF sym = begin THEN
    GetSym;
    LOOP
      statement;
      IF  sym = semicolon  THEN
        GetSym
      ELSIF  sym = end  THEN
        GetSym;
      EXIT
      ELSIF  sym < const  THEN
        err (17)
      ELSE
        err (17);
      EXIT
      END
    END
  ELSIF  sym = if  THEN
    GetSym;
```

```
    condition;
    IF  sym = then  THEN  GetSym  ELSE  err (16)  END;
    L0 := Label ();
    Gen (7, 0, 0);
    statement;
    fixup (L0)
  ELSIF  sym = while  THEN
    L0 := Label ();
    GetSym;
    condition;
    L1 := Label ();
    Gen (7, 0, 0);
    IF  sym = do  THEN  GetSym  ELSE  err (18)  END;
    statement;
    Gen (6, 0, L0);
    fixup (L1)
  ELSIF  sym = read  THEN
    GetSym;
    IF  sym = ident  THEN
      obj := find (id);
      IF  obj^.kind = Var  THEN
        Gen (1, 0, 14);
      Gen (3, curlev - obj^.vlev, obj^.vadr)
      ELSE
      err (12)
      END
    ELSE
      err (14)
    END;
    GetSym
  ELSIF  sym = write  THEN
    GetSym;
    expression;
    Gen (1, 0, 15)
  END;
  test (ident, 19)
END statement;



PROCEDURE block;

VAR   adr         : CARDINAL;       (* data address *)
      LO          : CARDINAL;
      hd, obj     : ObjPtr;

  PROCEDURE ConstDeclaration;

  VAR       obj   : ObjPtr;

  BEGIN
    WriteString (win, "ConstDeclaration");      WriteLn (win);
    IF  sym = ident  THEN
      GetSym;
      IF  (sym = eql) OR (sym = becomes)  THEN
        IF  sym = becomes  THEN  err (1)  END;
      GetSym;
      IF  sym = number  THEN
        obj := NewObj(Const);
        obj^.val := num;
```

```
        GetSym
      ELSE
        err (2)
      END
      ELSE
        err (3)
      END
    ELSE
      err (4)
    END
  END ConstDeclaration;


  PR0CEDURE VarDeclaration;

  VAR       obj   : ObjPtr;

  BEGIN
    WriteString (win. "VarDeclaration");        WriteLn (win);
    IF  sym = ident  THEN
      obj := NewObj (Var);    GetSym;
      obj^.vlev := curlev;    obj^.vadr := adr; adr := adr+1
    ELSE
      err (4)
    END
  END VarDeclaration;

BEGIN
  WriteString (win, "block");             WriteLn (win);
  curlev ;= curlev + 1;                   adr := 3;
  ALL0CATE (hd, TSIZE (Object));
  WITH  hd^  DO
    kind := Header;
    next := NIL;
    last := hd;
    name := 0; down := topScope
  END;
  topscope := hd; L0 := Label();          Gen(6, 0, 0);     (* jump *)
  IF  sym = const  THEN
    GetSym;
    LOOP
      ConstDeclaration;
      IF  sym = comma  THEN
        GetSym
      ELSIF  sym = semicolon  THEN
        GetSym;
      EXIT
      ELSIF  sym = ident  THEN
        err (5)
      ELSE
        err (5);
      EXIT
      END
    END
  END;
  IF  sym = var  THEN
    GetSym;
    LOOP
      VarDeclaration;
        IF  sym = comma  THEN
        GetSym
```

```
      ELSIF  sym = semicolon  THEN
        Getsym;
        EXIT
        ELSIF  sym = ident  THEN
        err (5)
      ELSE
        err(5);
        EXIT
      END
      END
    END;
  WHILE  sym = procedure  DO
    GetSym;
    IF  sym = ident  THEN  GetSym  ELSE  err (4)  END;
    obj := New0bj (Proc);
    obj^.plev := curlev;
    obj^.padr := label ();
    IF  sym = semicolon  THEN  Getsym  ELSE  err (5)  END;
    block;
    IF  sym = semicolon  THEN  GetSym  ELSE  err (5)  END
  END;
  fixup( L0);
  Gen (5, 0, adr);            (* enter *)
  statement;
  Gen (1, 0, 0);        (* return *)
  topScope := topScope^.down;
  curlev := curlev - 1
END block;

PROCEDURE Parse;

BEGIN
  noerr := TRUE;
  topScope : = NIL;
  curlev := O;
  Write (win, 14C);           (* RestoreHeap (bottom); *)
  GetSym;
  block;
  IF  sym # period  THEN  err (9)  END
END Parse;


PROCEDURE EndParser;
BEGIN
  CloseTextWindow (win)
END EndParser;

BEGIN
  ALLOCATE (undef, TSIZE (Object));
  ALLOCATE (bottom, O);
  WITH  undef^  DO
    name := 0;
    next := NIL;
    kind := Var;
    vlev := 0;
    vadr := 0
  END;
  0penTextWindow (win, 0, 68, 234, 508, "PARSE");
END PL0Parser.
```

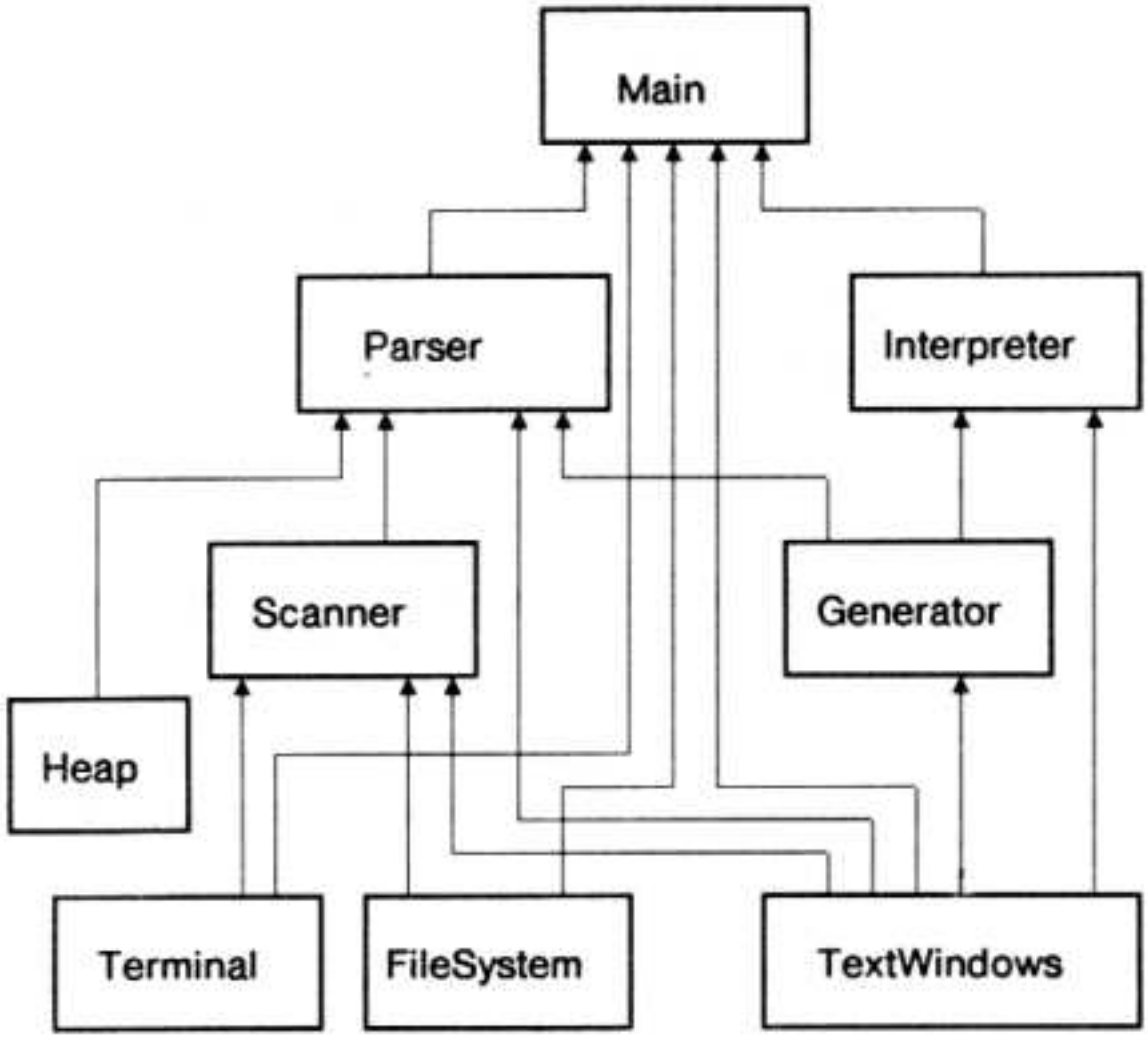
Programma 8: PL/0-parser

De structuur van het volledige compiler-interpreter-systeem kan uit figuur 8 worden afgelezen. De pijlen in die figuur geven aan in welke richting objecten uit modules worden geimporteerd.
Het systeem van modules wordt gestuurd door een hoofdmodule PL/0; in deze module wordt eerst de naam van het bestand waarin de programmatekst staat gevraagd.
Daarna wordt de compiler aan het werk gezet. Als er geen fouten optreden wordt tenslotte de interpreter aangeroepen om het programma uit te voeren.
In de figuren 9 en 10 ziet u welke gegevens bij het compileren en uitvoeren van twee korte programma's op het scherm verschijnen.

```
MODULE PL0;             (* NW WS 83/84 *)

FROM  Terminal        IMP0RT  Read;
FROM  FileSystem      IMPORT  Lookup, Response, Close;
FROM  TextWindows     IMPORT  Window, OpenTextWindow, Write, WriteLn,
                              WriteString, CloseTextWindow;
FROM  PL0Scanner      IMPORT  InitScanner, source, CloseScanner;
FROM  PL0Parser       IMPORT  Parse, noerr, Endparser;
FROM  PL0Generator    IMPORT  InitGenerator, EndGenerator;
FROM  PL0Interpreter  IMPORT  Interpret, EndInterpreter;

CONST NL    = 27;       (* max file name length *)

VAR   ch          : CHAR;
      win         : Window;
      FileName    : ARRAY [O..NL] OF CHAR;

PROCEDURE ReadName;

CONST DEL   = 10C;

VAR   i     : CARDINAL;

BEGIN
  Read (ch);
  FileName := "";
  i := 0;
  WHILE  (CAP (ch) >= "A") & (CAP (ch) <= "Z") OR (ch >= "O") & (ch <= "9")
       OR (ch = ".") OR (ch = DEL)  DO
    IF  ch = DEL  THEN
      IF  i > O  THEN
        Write (win, DEL);
      i := i-1
      END
    ELSIF  i < NL  THEN
      Write (win, ch);
      FileName [i] := ch;
      i := i+1
    END ;
    Read (ch)
  END;
  IF  (1 < i) & (i < NL) & (FileName [i-1] = ".")  THEN
    FileName [i] := "P";      i := i+1;
    FileName [i] := "L";      i := i+1;
    FileName [i] := "0";      i := i+1;   WriteString (win, "PL0")
  END;
  FileName [i] := 0C
END ReadName;
```

```
BEGIN
  OpenTextWindow (wìn, 0, 0, 704, 66, "DIALOG");
  LOOP
    WriteString (win, " in> ");
    ReadName;
    IF  ch = 33C  THEN  EXIT  END;
    Lookup (source, FileName, FALSE);
    IF  source.res = done  THEN
      InitScanner;
      InitGenerator;
      parse;
      Close (source);
      IF  noerr  THEN
        WriteString (win, " interpreting");  Interpret
      ELSE
        WriteString (win, " incorrect")
      END
    ELSE
      WriteString (win, " not found: ")
    END;
    WriteLn (win)
  END;
  CloseScanner;
  EndParser;
  EndGenerator;
  EndInterpreter;
  CloseTextWindow (win)
END PL0.
```

Programma 9: Hoofdmodule van het compiler interpreter systeem.



Figuur 8: Modulestructuur van het PL/0 systeem

Moet nog gemaakt worden

Figuur 9: Voorbeeldprogramma grootste gemene deler

Moet nog gemaakt worden

Figuur 10: Voorbeeldprogramma met syntactische fouten

Kijk vooral ook op http://fruttenboel.verhoeven272.nl/m4m/index.html voor nieuwe ontwikkelingen.

12 Een taaluitbreiding: processen

Heel algemeen gezien zijn programma's beschrijvingen van automaten die zich strikt volgens voorgeschreven regels gedragen. We beschouwen de computer als een apparaat dat door elk ander programma in een andere automaat wordt veranderd. Binnen deze zienswijze is programmeren het met behulp van bouwstenen construeren van machines; de bouwstenen worden door de gebruikte programeertaal aangeleverd.
In geen enkele andere tak van technologie is het zo gemakkelijk een eenmaal geconstrueerde machine te veranderen, uit te breiden, of aan veranderde behoeften aan te passen. Dit heeft tot gevolg dat de programmeur er meer dan ingenieurs uit andere vakgebieden op vertrouwt dat zijn produkten gemakkelijk en ogenschijnlijk kostenloos zijn te veranderen.
Daarom vinden we in de praktijk ook zelden geprogrammeerde systemen die definitief 'af' zijn. Het is vooral bij de constructie van complexe systemen belangrijk deze systemen zo te ontwerpen, dat ze op zo veel mogelijk verschillende manieren uitgebouwd kunnen worden.
Daarbij verdient 'modularisering', dat wil zeggen het opsplitsen van het ontwerp in modules, in het bijzonder aandacht. Als de basisstructuur van een systeem goed is gekozen kunnen veranderingen op enkele plaatsen in het systeem worden gelokaliseerd; dit is van groot belang voor de overzichtelijkheid en betrouwbaarheid van het systeem.
Voortdurende evolutie is echter niet alleen een eigenschap van programma's, maar ook van programmeertalen. Dit is gemakkelijk te verklaren: ook de vertalers (compilers) van deze programmeertalen zijn weer gemakkelijk te veranderen programma's. In dit hoofdstuk besteden we daarom aandacht aan een taaluitbreiding en de realisering daarvan.
Het gaat daarbij slechts om één voorbeeld uit veel denkbare voorbeelden op het veel algemenere terrein van de systeemuitbouw. De hier eerst te formuleren regels voor het maken van uitbreidingen hebben dan ook een veel bredere geldigheid dan het daarna te geven voorbeeld. In de opgaven aan het einde van dit boek zijn meer soortgelijke projecten te vinden.

Als we een programmeertaal willen uitbreiden is het aan te bevelen eerst aandacht te schenken aan de conceptuele aspecten van die uitbreiding. De nieuwe elementen van de taal moeten zo algemeen mogelijk worden opgevat en gedefinieerd. Het toespitsen van de uitbreiding op een bepaald soort toepassingen moet zoveel mogelijk worden vermeden.
Vanzelfsprekend kent de ervaren constructeur bepaalde implementatietechnieken; hij kan zelfs de voor- en nadelen van die technieken enigszins inschatten. Het is de kunst niet al te vroeg een bepaalde oplossing te kiezen. We maken daarom een schema waarin staat waarop onze hoofdaandacht achtereenvolgens gericht moet zijn.

1. Het definiëren van de nieuw in te voeren basisconcepten.
2. Het definiëren van de nieuwe elementen van de taal. Formele syntaxis en semantiek.
3. Het vastleggen van de voor de nieuwe taalelementen te produceren reeksen instructies. Indien nodig: het definiëren van de uitbreidingen van de interpreterende machine.
4. Het bepalen van de gegevenstypen en gegevensstructuren in de compiler die voor het weergeven van de nieuwe taalelementen nodig zijn.
5. Het programmeren van de algoritmen waarmee de nieuwe taalelementen worden verwerkt. Het integreren van de nieuwe delen van de compiler.

6. Controle van de juistheid van de implementatie aan de hand van testprogramma's; deze controle moet met en zonder de nieuwe taalelementen worden uitgevoerd.
7. We doorlopen hierna de eerste vier stappen van dit schema aan de hand van een voorbeeld; we willen PL/0 uitbreiden van een zuiver sequentiële taal tot een taal waarin stukken programma kunnen worden beschreven die gelijktijdig uitgevoerd worden.

Stap 1 : Basisconcepten

We wijken af van het axioma dat programma's zuiver sequentieel moeten worden uitgevoerd door het declareren van meer processen toe te staan; deze processen zijn op zich sequentieel, maar kunnen, als ze eenmaal gestart zijn, gelijktijdig uitgevoerd worden. Dit concept is echter alleen interessant als deze processen op één of andere manier met elkaar kunnen communiceren.
De eenvoudigste vorm van communicatie is die waarin gemeenschappelijke variabelen worden gebruikt. Daarmee zijn processen te synchroniseren; dit bekekent dat ze zo zijn te programmeren, dat ze op bepaalde punten op elkaar wachten.
Laten we als voorbeeld aannemen dat een proces P na het statement A moet wachten op het beëindigen van het statement B door een proces Q. We kunnen deze situatie programmeren door een gemeenschappelijke variabele w als volgt te gebruiken; w heeft aan het begin de waarde TRUE.

```
P:  ...  A; WHILE w DO END; ...
Q:  ...  B; w := FALSE; ...
```

Deze techniek van synchroniseren heeft, ook als we toestaan dat zulke synchronisaties maar een enkele keer worden uitgevoerd -we spreken dan van losjes gekoppelde processen-, een beslissend nadeel: de techniek vraagt er bijna om elk proces aan een eigen processor toe te wijzen.
Waarom? Het concentreren van al het werk op enkele processoren, of zelfs op één processor, is niet realistisch, omdat de processen, ook als ze alleen maar wachten, 'bezig zijn'. De uitvoering van het statement "WHILE w DO END" bestaat uit een perverse vorm van 'bezig zijn met wachten' (busy waiting).
Het werkelijke doel van de programmeur wordt door dit statement versluierd. Deze oplossing is gekozen omdat in de taal geen passende structuurelementen voorhanden zijn. We corrigeren dit gebrek door een nieuw statement voor wachten in te voeren.

Het invoeren van zo'n nieuw statement vraagt op zich weer om het invoeren van nieuwe objecten: voortgangsdoelen waarop moet worden gewacht. Afhankelijk van hoe we naar deze doelen kijken worden ze voorwaarden (conditions) of gebeurtenissen (events) genoemd; er wordt gewacht op het voldaan zijn aan voorwaarden of het optreden van een gebeurtenis. Het is niet zo zinvol hiervoor de in de literatuur gebruikte aanduiding wachtrij (queue) te gebruiken; deze term duidt op een mogelijke implementatie van deze objecten. We kiezen hier de neutrale aanduiding signaal: een proces wacht op het binnenkomen van een signaal dat door een ander proces wordt gestuurd.
Daarmee wordt ook aangegeven dat bij signalen twee operaties nodig zijn: het zenden van een signaal en het ontvangen van een signaal. Onder bepaalde omstandigheden wacht de ontvanger tot het signaal binnenkomt.

Het synchroniseren van processen met behulp van deze nieuwe objecten heeft ten opzichte van het werken met conventionele variabelen het grote voordeel dat processen niet elkaars namen behoeven te 'kennen'. Een proces wacht op een

signaal dat, onafhankelijk van de identiteit van de afzender van dat signaal, een bepaalde, voor de voortgang noodzakelijke toestand van het systeem meldt.
Dit ís in het bijzonder voordelig bij het simuleren van discrete gebeurtenissen. De hier geplande uitbreiding ontsluit dus een veelzijdig terrein van nieuwe toepassingen. Beslissend is voor ons echter dat het door het expliciet invoeren van het concept signaal mogelijk wordt een systeem met meer processen te implementeren op een computer met één processor. Weliswaar wordt het tegelijkertijd verlopen van processen daarbij slechts gesimuleerd, maar als een proces in de wachttoestand komt kan de processor wel onverwijld worden ingezet voor de uitvoering van andere processen.
Het is niet nodig steeds te controleren of het wachten kan worden beëindigd. De wachttoestand kan namelijk op maar één manier worden opgeheven; door het zenden van het betreffende signaal door het proces dat op dat moment uitgevoerd wordt. Er hoeft daarom niet op wachtende processen gelet te worden; deze processen kosten ook geen processortijd.
Eigenlijk zou naast processen en signalen nóg een ander concept moeten worden ingevoerd om met enig nut processen simultaan te kunnen laten verlopen: een concept waarmee processen elkaar in kritische onderdelen van programma's wederzijds uitsluiten. Omdat we ons hier bezighouden met een simulatie van gelijktijdige processen door één processor kunnen we dit probleem van het elkaar wederzijds uitsluiten gemakkelijk oplossen: in kritische gedeelten van het programma plaatsen we geen statements waarmee de processor van één proces naar een ander zou kunnen overgaan.
We kunnen daarom afzien van de invoering van een speciale taalstructuur om kritische stukken van een programma mee aan te duiden.

Stap 2 : Formele definitie van syntaxis en semantiek.

We beperken ons tot het aangeven van de uitgebreide syntaxis in de vorm van EBNF-produkties. De veranderingen betreffen de structuren block en statement. Het zenden van een signaal s duiden we aan met !s, het wachten op en ontvangen van dat signaal met ?s. Met het toepassen van dezelfde schrijfwijze als bij de invoer en uitvoer van gegevens geven we aan dat het om nauw verwante concepten gaat.

```
block =
  ["CONST" ident "=" number {"," ident "=" number} ";"]
  ["VAR" ident {"," ident} ";"]
  ["SIGNAL" ident {"," ident} ";"]
  {("PROCEDURE"|"PROCESS") ident ";" block ":"} statement.

statement = ["!" ident | "?" ident | ident ":=" expression
             | "CALL ident | ... ].
```

Declaraties van processen hebben dezelfde vorm als declaraties van procedures. Als in een aanroep (call) de identifier van een proces wordt gebruikt, wordt met het betreffende statement een nieuw proces gecreeerd; dit proces zal simultaan (of quasi-simultaan) met het aanroepende proces worden uitgevoerd.
Dit in tegenstelling tot het aanroepen van een procedure; bij het aanroepen van een procedure wordt pas verder gegaan met de uitvoering van de aanroepende procedure als de aangeroepen procedure beëindigd is. De zend- en ontvang-statements hebben natuurlijk betrekking op een signaal. De betekenis van het overzenden van een signaal is dat aan het ontvangende proces wordt gemeld dat het systeem (van processen) een bepaalde toestand heeft bereikt.
Dit is de eigenlijke informatie waarop het signaal betrekking heeft; we duiden deze informatie aan met het predicaat Ps. De semantiek van beide nieuwe statements

kan nu door middel van een voorwaarde voor het zenden en een consequentie van het ontvangen van een signaal axiomatisch worden gedefinieerd.

{Ps} !s ?s {Ps}

Stap 3 : Uitbreiding van de PL/0-machine en definiëring van de geproduceerde code.

Het is duidelijk dat de conceptuele uitbreiding van PL/0 met simultane processen ook een daarmee corresponderende uitbouw van de PL/0-machine vereist. We gaan er in het volgende nadrukkelijk van uit dat de gelijktijdigheid niet alleen gesimuleerd kan worden, maar ook gesimuleerd moet worden. We kunnen de PL/0-machine dus als sequentieel programma blijven beschrijven.
We houden ons vooralsnog niet met de vraag bezig hoe één processor meer processen moet bedienen, maar bespreken eerst de problemen die bij het creëren van een proces optreden. De problematiek van het beheer van het geheugen komt daarbij onmiddellijk naar voren.
Als er tegelijkertijd meer processen bestaan en er geen eenvoudige relatie bestaat tussen de opeenvolging van tijdstippen waarop deze worden gecreëerd en weer beëindigd, kan de beproefde stapel-strategie niet meer worden gehanteerd. We willen hier echter niet ingaan op algemene methoden van geheugenbeheer; we poneren daarom enkele vereenvoudigende regels zonder enige aanspraak op efficiënte toepasbaarheid van deze regels in voor de praktijk te ontwerpen systemen.

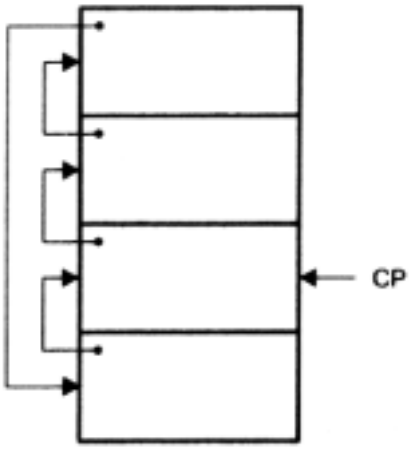
1. Processen worden alleen in het hoofdprogramma gedeclareerd. Daarmee wordt vermeden dat in een proces Q, zelf gedeclareerd binnen een proces P, lokale variabelen van het proces P worden gebruikt op een moment dat P niet meer bestaat.

2. Telkens als een nieuw proces wordt gecreëerd wordt aan dat proces geheugen-ruimte van vaste omvang toegewezen. In die ruimte heeft dat proces zijn eigen stapel (workspace).

3. Aan het einde van een proces wordt de geheugenruimte weer vrijgegeven. Als het geheugenbeheer goed is georganiseerd kan deze ruimte, als later weer een proces wordt gecreëerd, opnieuw gebruikt worden.

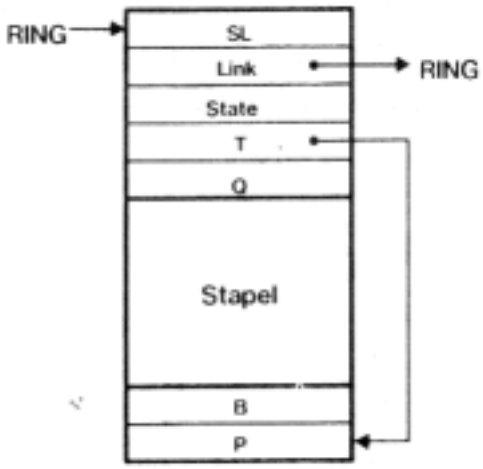
Het belangrijkste nadeel van deze primitieve strategie is dat de stapel in een vaste geheugenruimte moet worden ondergebracht, terwijl niet van te voren bekend is wat de maximum ruimte is die een stapel nodig heeft. Daar staat als voordeel tegenover dat de strategie eenvoudig is. Het geheugen kan nog steeds als array worden voorgesteld; de afzonderlijke stapels zijn onderling samenhangende stukken van dit array.
We delen het geheugen S voor gegevens op in stukken; elk stuk behoort bij een proces en in elk stuk is het eerste element een proces-descriptor. Dit stuk noemen we een proces-segment. Elk proces-segment bestaat uit afzonderlijke procedure-segmenten en wordt beheerd als stapel.
De afzonderlijke proces-segmenten zijn op de in fuguur 11 aangegeven manier met wijzers (indices) aan elkaar geketend. De keten van alle actieve segmenten vormt de zogenaamde procesring. Het nieuwe register CP (current process) geeft het proces aan dat op dat moment wordt uitgevoerd. Het is het eenyoudigst het hoofdprogramma zelf ook als proces te beschouwen. Het is verstandig voor dit hoofdprogramma een grotere geheugenruimte te reserveren, waarbij rekening wordt gehouden met het aantal globale variabelen.

De registers P, T en B (zie hoofdstuk 10) bevatten dezelfde waarden als in hoofdstuk 10 is beschreven, maar dan voor het proces dat op dat moment wordt uitgevoerd. De registerwaarden van alle andere processen moeten in hun segment worden opgeslagen. In figuur 12 vindt u een schema van een segment met daarbijbehorende gegevensvelden.



Figuur 11: Geheugenruimte met proces-segmenten

SL duidt net als eerst de statische keten van procedure-segmenten aan (zie hoofdstuk 10); link heeft betrekking op de procesring en state geeft de toestand van een proces aan. Het proces is Chapter12verwerkings-gereed, kortweg 'gereed' (ready; state = 0) of wacht op een signaal (waiting; state > O). Het ligt eigenlijk voor de hand aan de wachttoestand als extra attribuut een verwijzing naar het signaal waarop wordt gewacht toe te kennen. Dit is echter niet erg zinvol omdat het wachtende proces zelf helemaal niet in staat is op dit signaal te reageren.
Het is veel doelmatiger om omgekeerd aan het signaal als attribuut een verwijzing naar het proces toe te kennen. Omdat meer processen op hetzelfde signaal kunnen wachten is een ketening van deze processen in het algemeen een nog betere oplossing.
Het begin van deze wachtrij vormt als het ware de 'waarde' die aan het signaal wordt toegekend. Aan elk gedeclareerd signaal wordt daarom door de compiler op dezelfde manier als aan variabelen een geheugenplaats toegewezen. Voor de programmeur heeft een signaal echter niet de betekenis van een waarde die gelezen en door een toekenning veranderd kan worden.
Het veld Q in de proces-descriptor dient als opslagplaats voor de schakel in deze keten (queue). Het laatste proces in de keten bevat de pointerwaarde Q = NIL.



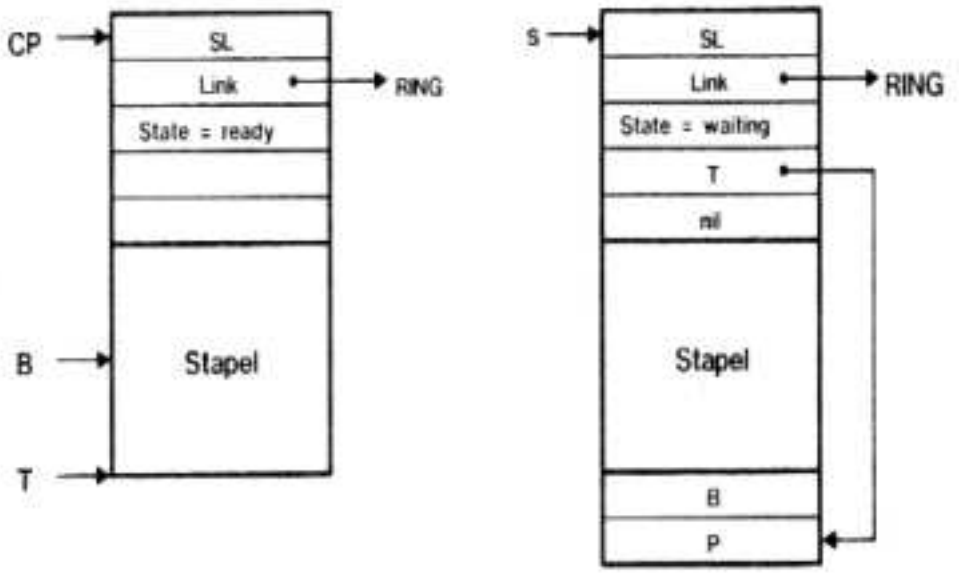
Figuur 12: Segment van één proces

Nu de zogenaamde architectuur of structuur van de nieuwe PL/0-machine vaststaat moeten de instructies nog worden gedefinieerd. Het gaat om vier instructies, namelijk

- de instructie STRT om een nieuw proces te creëren en te starten
- de instructie SEND om een signaal te zenden
- de jnstructie RECEIVE om een signaal te ontvangen
- de instructie STOP om een proces op te heffen

Analoog aan CAL heeft de instructie STRT als parameter het beginadres van een proces. SEND en RECEIVE hebben als parameter het adres yan het signaal (in het gegevensgeheugen), terwijl STOP in de klasse operatoren zonder parameters kan worden ondergebracht,
Nu ook de uitbreidingen van de instructieset zijn gedefinieerd moeten we nog de algoritmen voor de afzonderlijke instructies beschrijven. We zien hier af van een uitvoerige programmering van die algoritmen. We beperken ons tot enkele essentiële punten. Als het beschreven principe van geheugentoewijzing wordt gevolgd is de implementatie van de instructies STRT en STOP heel eenvoudig.
We richten ons daarom onmiddellijk op de instructies SEND en RECEIVE door de toestanden n processegment vóór en na de uitvoering van deze instructies te beschrijven.
Het proces wordt ingevoegd in een pointerketen die bij s begint; deze keten stelt de wachtrij met processen die op s wachten voor (zie figuur 13). Na het opschorten van een proces kan in principe met elk willekeurig proces dat in de toestand gereed verkeert worden verdergegaan. Omdat alle processen in de ring aan elkaar geketend zijn ligt het voor de hand deze ring te doorzoeken en het eerste proces te kiezen dat 'gereed' is. Als zo'n proces niet bestaat is het systeem in een patstelling geraakt, waarbij elk proces op het signaal van een ander proces wacht. In deze toestand is het systeem vastgelopen; in jargon spreken we van een deadlock.
In een goed geprogrammeerd systeem doet zich zo'n toestand niet voor. Een foutmelding is daarom in zo'n situatie op zijn plaats.

Bij de implementatie van de instructie SEND is men wellicht geneigd alleen het eerste proces uit de wachtrij te pakken en de toestand van dat proces als 'gereed' te markeren. Dit is echter niet geoorloofd; de axioma's die voor signaaloperaties zijn geponeerd verlangen immers dat het ontvangende proces dezelfde toestand aantreft als door de afzender wordt gesignaleerd.



Figuur 13: Procesegment vóór en na ?s

Als de processor niet direct door het proces dat het signaal stuurt aan het wachtende proces zou worden toegewezen zou een derde proces de door het signaal aangegeven toestand intussen weer ongedaan kunnen maken. Daaruit volgt dat de instructie SEND op precies dezelfde manier moet worden geimplementeerd als RECEIVE; alleen blijft nu de zender in de toestand 'gereed' en de ontvanger wordt expliciet door de instructie aangewezen. De ontvanger hoeft daarom niet eerst in de ring opgezocht te worden.

Tenslotte moeten we nog de situatie behandelen waarin de wachtrij met op het signaal wachtende processen leeg is. Hoe moeten we het effect van een signaal waarop geen enkel proces wacht definiëren? Het is gebruikelijk het signaal in zo'n situatie gewoon als niet-gegeven te beschouwen en met het proces dat het signaal zendt verder te gaan.
Er zijn echter ook argumenten om deze situatie te zien als een niet bedoelde programmeerfout; het signaal moet dan in overeenstemming daarmee worden geinterpreteerd. Het is op zijn minst zinvol in dat geval de processor vrij te geven.

Stap 4 : Uitbreidingen in het declaratiegedeelte van de compiler.

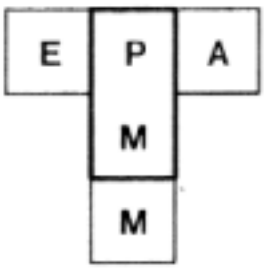
De uitbreidingen in het declaratiegedeelte van de compiler zijn relatief beperkt en eenvoudig. Het waardebereik van het type Symbol moet zeker worden uitgebreid: de nieuwe symbolen moeten erin worden opgenomen (zie programma 4). Verder krijgt het type ObjectClass de twee nieuwe waarden process en signal. Het type Instruction wordt zo uitgebreid, dat de vier nieuwe instructies erin kunnen worden ondergebracht.
De interpreter krijgt een nieuwe variabele: het procesregister CP; dit register wijst naar de descriptor van het proces dat op dat moment verwerkt wordt.

We slaan een uitvoerige programmering van de niet zo ingewikkelde uitbreidingen van de compiler over. In de praktijk wordt deze techniek van procesbesturing gebruikt in systemen met meer processen (tìme-sharing). Meestal worden daarbij geraffineerde strategieën toegepast voor het beheer van het geheugen en voor het inzetten van de processor (process scheduling). Omdat het thema van dit hoofdstuk de uitbreiding van talen en de implementatie van die talen is gaan we niet in op de mogelijkheden om dit ene voorbeeld te verbeteren.

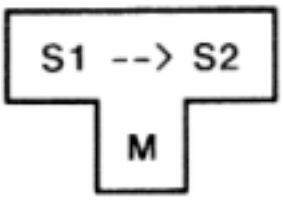
13 Compìlers ontwikkelen en transporteren

In de voorgaande hoofdstukken hebben we een PL/0-compiler ontwikkeld en geprogrammeerd. Daarbij werd duidelijk dat een compiler over het algemeen een ingewikkeld programma is; het is daarom zeer aan te bevelen bij de implementatie van een compiler een hogere programeertaal te gebruiken.

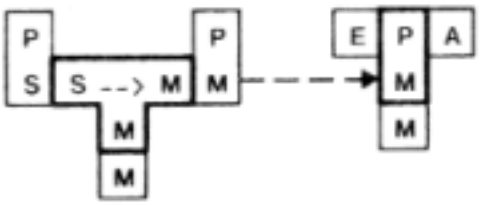
Uiteraard gelden bij de ontwikkeling van een compiler dezelfde regels en technieken voor systematische ontwikkeling van programma's als bij de programmering van elk ander gecompliceerd algoritme. Toch gelden een aantal punten specifiek voor compilers; daarom worden die hier besproken.
Deze punten hebben vooral betrekking op de veel voorkomende situatie dat een compiler moet worden uitgebreid. Bij het ontwikkelen van een compiler, en zelfs van een programmeertaal, wordt aan het begin bijna nooit direct aan de definitieve vorm gewerkt.
Eerst wordt de kern van de taal geimplementeerd; voortbouwend daarop wordt het einddoel stapsgewijs benaderd. We gaan in dit hoofdstuk verder op deze techniek in.
Eerst voeren we echter een notatie in waarmee we de ontwikkeling van programma's in beeld brengen. Met de figuur:



geven we aan dat uit de invoergegevens E door middel van een programma P de uitvoergegevens A worden herleid; het programma P is geformuleerd in de taal M en derhalve interpreteerbaar door de computer M.
Als het programma P een compiler is die zinnen uit de taal S1 vertaalt in de taal S2, dan wordt deze compiler -die zelf in de taal M is geformuleerd- voorgesteld door de figuur;

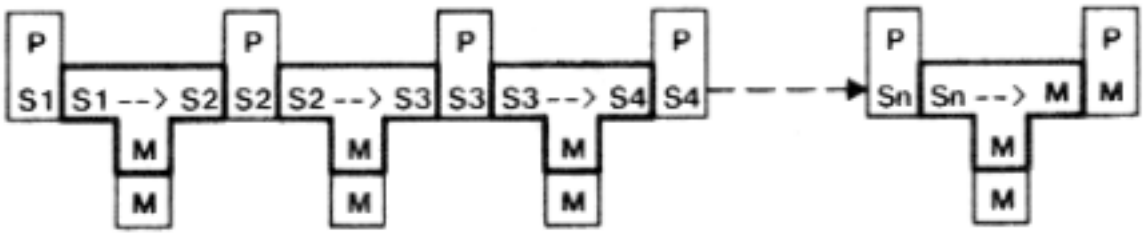


Omdat deze figuur de vorm van een T heeft wordt er de naam T-diagram aan gegeven. Met behulp van deze diagrammen kunnen we de gebruikelijke gang van zaken bij het compileren en daarna uitvoeren van een programma P schematisch weergeven (zie figuur 14); daarbij wordt in beide stappen dezelfde computer M gebruikt.



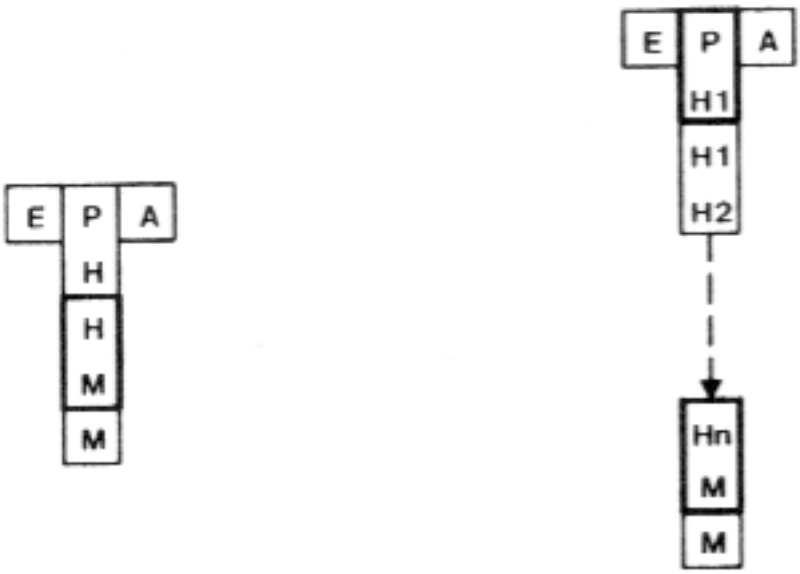
Figuur 14: Schematische.voorstelling van compilatie en uitvoering

In deze figuur wordt de karakteristieke eigenschap van T-diagrammen duidelijk: ze kunnen als dominostenen tegen elkaar worden gelegd. Daarbij moet in de velden die tegen elkaar liggen dezelfde identifier staan.
De figuur leert ons dan dat de compiler het programma P, geformuleerd in de 'brontaal' S, vertaalt in hetzelfde programma P, maar nu geformuleerd in de 'doeltaal' M.
We geven in figuur 15 nog een voorbeeld van het gebruik van T-diagrammen: het verloop van een meerfasencompilatie.



Figuur 15: Compilatie van een programma P in n fasen

In figuur 16 brengen we het verloop van het interpretatief uitvoeren van een programma P in beeld; P is geformuleerd in een taal (of code) H en wordt met een op computer M geimplementeerde interpreter uitgevoerd.
De algemene vorm van deze figuur is interpretatie in meer stappen; ook deze algemene vorm staat in figuur l6. Een voorbeeld van deze algemene vorm zou de PL/0-code voor H1 zijn, geimplementeerd op een machine die geprogrammeerd is ín de (micro)code H2.



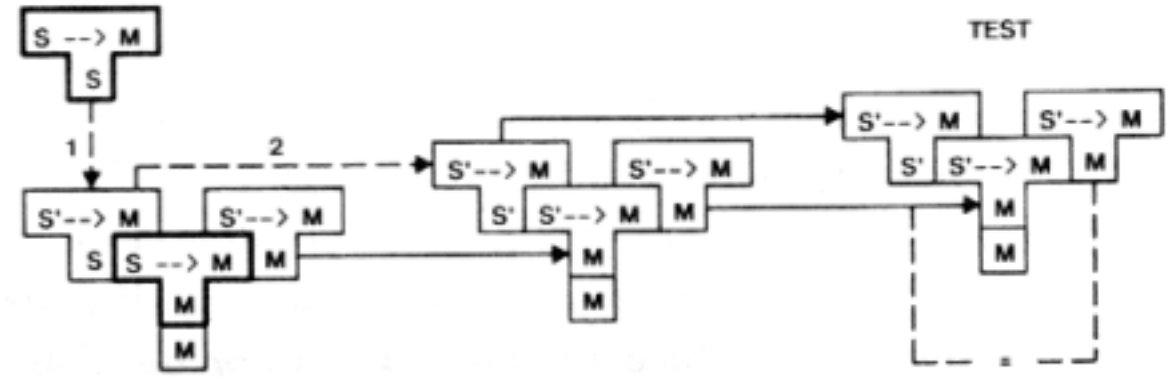
figuur 16: Interpretatief uitvoeren van een programma P

Nu we T-diagrammen hebben geintroduceerd kunnen we ons bezig gaan houden met de techniek van de stapsgewijze ontwikkeling van compilers. Het heeft grote voordelen als een compiler voor de 'brontaal' S zelf eveneens in S is geformuleerd. Laten we vooralsnog aannemen dat er een compiler voor de taal S bestaat; deze compiler bestaat zowel geformuleerd in S als in de machinetaal van de beschikbare computer M. Nu moet S verder worden ontwikkeld tot versie S'.

Dit gebeurt in twee stappen, namelijk

1. door het programmeren van de uitbreidingen in S (zie hoofdstuk 12) en
2. door het opnieuw formuleren van de compiler, waarbij de nieuwe taalinstrumenten nuttig worden toegepast.

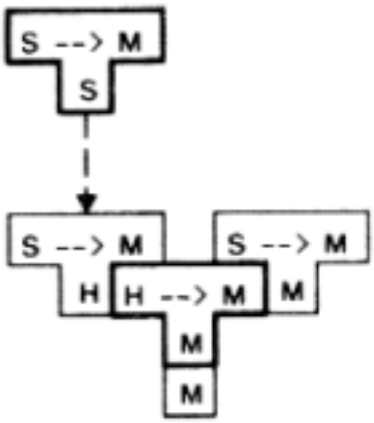
Het grote voordeel van het formuleren van een compiler in zijn eigen 'brontaal' is dus dat de compiler zelf direct van een uitbreiding kan profiteren. Daarom kan het ontwikkelen van een compiler beginnen met een relatief eenvoudige taal; door herhaling van bovengenoemde stappen wordt naar het eindprodukt toegewerkt.
In vaktaal heet dit zichzelf 'aan de veters van zijn eigen laarzen uit het moeras optrekken' ofwel **bootstrapping**. In figuur 17 ziet u wat in een stap uit dat proces gebeurt; deze stap kan zo vaak worden herhaald als nodig is.



figuur 17; Bootstrap van een computer van S naar S'

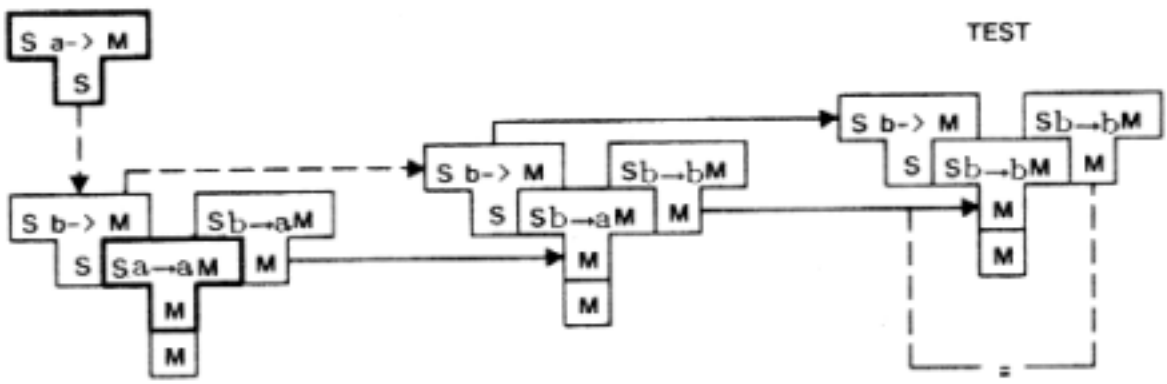
Ook de derde in figuur 17 afgebeelde stap is nog het vermelden waard. Deze stap is toegevoegd om te controleren of de verandering van de compiler van S naar S' juist is verlopen. Deze controle vindt zuiver mechanisch plaats door vast te stellen of de resultaten van de stappen 2 en 3 precies aan elkaar gelijk zijn.

Het bestaan van zo'n rigoreuze test is in de praktijk van onschatbare waarde. Nu rijst natuurlijk onmiddellijk de vraag wat de beginstap moet zijn. Hoe maken we een compiler voor een eerste versie van S, en wel geformuleerd in S?
Gewoonlijk wordt als oplossing van dit probleem gekozen voor het 'met de hand' vertalen van S naar een andere op de machine M beschikbare taal N. In de praktijk blijkt het voordelen op te leveren daarvoor geen assembleertaal, maar een hogere programmeertaal te gebruiken. De beginstap vindt u in figuur 18.



Figuur 18: Realisering van een compiler voor S

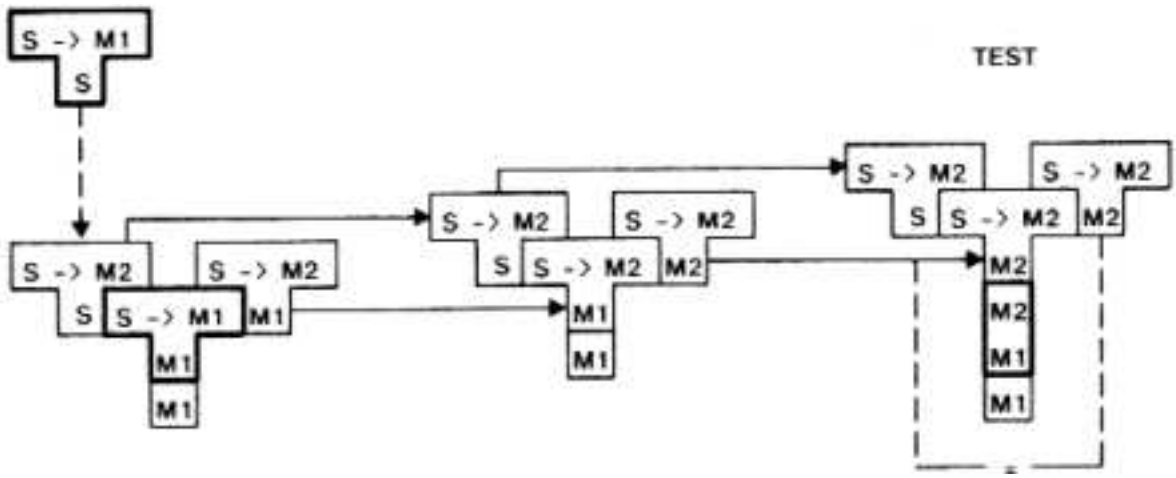
De beginstap wordt aanzienlijk gemakkelijker gemaakt door de gelukkige omstandigheid dat over het algemeen bij het programmeren in de 'vreemde' hulptaal H geen rekening hoeft te worden gehouden met efficiency.
Als het allereerste produkt inefficiënt is heeft dit geen gevolgen voor de op basis daarvan ontwikkelde compiler; alleen in de eerste bootstrap zal er iets van zijn te merken, daarna kan dit hulpprodukt worden weggegooid.
Een bootstrap hoeft niet altijd een uitbreiding van de taal in te houden. We kunnen ook een bootstrap maken om de methode van compileren en de geproduceerde code te verbeteren. Zo'n bootstrap ziet u in figuur 19. Als in de formule S x --> y M geldt 'x = 0', dan wordt de oude compileermethode gebruikt.
Als x = b wordt de verbeterde compileermethode gebruikt. Als y = 0 wordt de compiler zelf door de oude versie van de compiler geproduceerd, als y = b door de nieuwe versie, Het is duidelijk dat het bij de methode van bootstrapping toegestaan is in het begin eenvoudige, maar niet optimale code te produceren: later kan deze code worden verbeterd. Als de compiler in de eigen 'brontaal' is geschreven kan hij als eerste programma van deze verbeteringen profiteren. In de toegevoegde testfase kan weer worden gecontroleerd of de verbeterde versie van de compiler juist is en zichzelf kan reproduceren.



Figuur 19: Bootstrap ter verbeterjng van een compiler

Ter afsluiting gaan we nog in op de problemen rond compilertransport. Daarbij gaat het erom dat een taal S op een computer M2 moet worden geimplementeerd door een bestaande compiler van M1 'over te brengen' naar M2.
Compilers vormen onder alle programma's juist op dit punt een uitzondering. De toepassing van een taal die zowel voor M1 als voor M2 bestaat biedt namelijk geen oplossing. Een compiler produceert immers reeksen instructies van zijn 'doeltaal'; hij is dus 'van nature' afgestemd op één bepaalde computer en daarom niet zonder meer overdraagbaar (portabel).
Transport van een compiler betekent daarom altijd het maken van een principieel nieuw programma. Weliswaar kunnen, als de compiler goed is opgebouwd, grote delen -zoals de syntaxis analyse en de foutafhandeling- onveranderd worden overgenomen (zie de hoofdstukken 8 en 9), maar het blijft onvermijdelijk een deel echt opnieuw te programmeren.
Het ligt voor de hand om het overblijvende werk tot een minimum te reduceren door een gemeenschappelijke programmeertaal te gebruiken. Een vaak gebruikte kandidaat voor deze rol is Fortran. In de praktijk blijkt echter steeds weer dat in Fortran geschreven programma's veel minder portabel zijn als aanvankelijk werd aangenomen.
De oorzaak hiervan is dat de systeemprogrammeur gedwongen is uitbreidingen te gebruiken die alleen op zijn machine en voor zijn Fortran-versie gelden. Bovendien biedt Fortran weinig bescherming tegen het gebruik van bepaalde specifieke

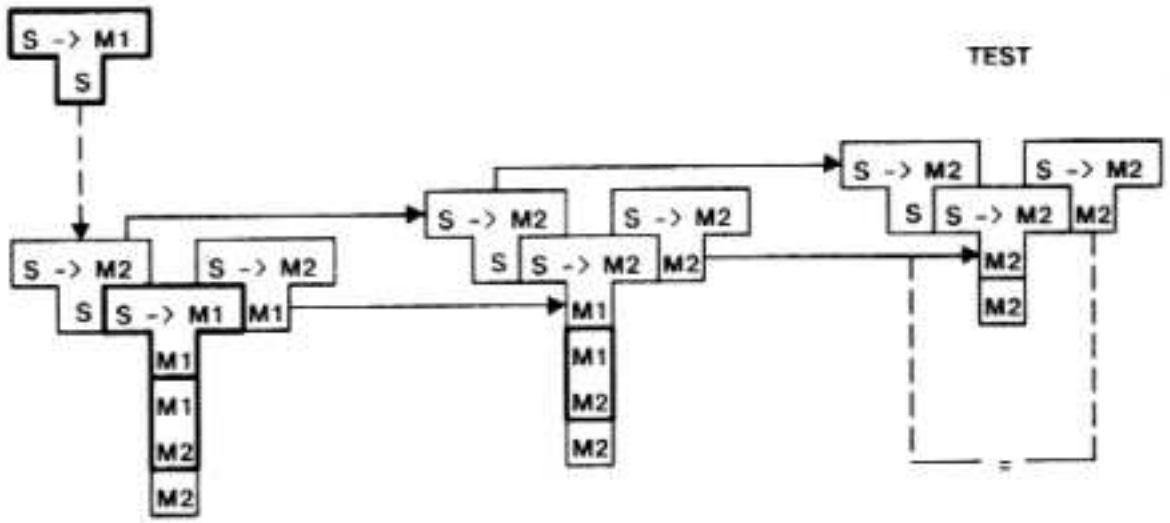
eigenschappen van de apparatuur; dit geldt met name voor die eigenschappen die door de structuur van Fortran heen zichtbaar blijven. Het gevolg daarvan is dat een latere aanpassing van een programma aan een ander apparaat vaak een ingewikkelde onderneming is.
We laten daarom zien welke technieken gebruikt kunnen worden als de compiler weer in zijn eigen taal geformuleerd is. We onderscheiden twee situaties waarin fundamenteel anders te werk moet worden gegaan.
In de eerste situatie wordt het omzetwerk uitsluitend op de 'bronmachine' M1 uitgevoerd. Dit zal met name gebruikelijk zijn als M1 de machine is van de leverancier en M2 de machine van de opdrachtgever.
In de tweede situatie nemen we aan dat de ontvanger zelf het noodzakelijke werk op M2 uitvoert.
Als we aannemen dat al het werk op M1 moet worden uitgevoerd hebben we als beginprodukten nodig: de compiler geschreven in de brontaal (S) en in binaire vorm (M1). De manier van werken wordt in figuur 20 weergegeyen. Voor de testfase is een interpreter (emulator) van M2 op M1 vereist.



Figuur 20; Transport van M1 naar M2 bij gebruik van M1

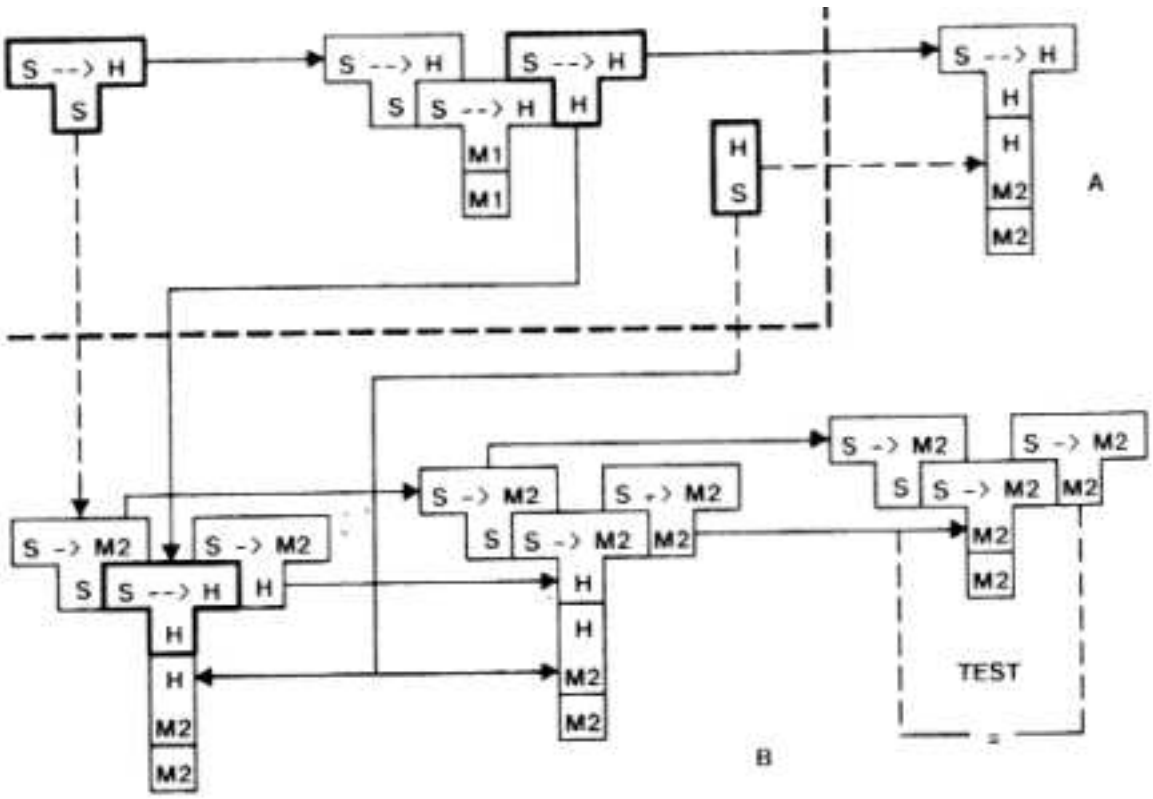
Als het werk uitsluitend op de 'doelcomputer' M2 moet worden uitgevoerd wordt veel nadrukkelijker gebruik gemaakt van de emulatietechniek. Hier is een interpretator (emulator) van M1 op M2 nodig (zie figuur 21). Het nadeel van deze methode is dat uitsluitend interpreterend wordt gecompileerd. Bedenk daarbij dat een goede interpreter voor elke geinterpreteerde instructie toch gemiddeld tussen de 20 en 30 machine instructies uitvoert.
Als we er bovendien nog rekening mee houden dat in de praktijk altijd fouten in programma's sluipen, waardoor elke stap in werkelijkheid een aantal malen wordt herhaald, dan kunnen we concluderen dat hier erg veel rekentijd wordt gebruikt. Deze methode is daarom in de prakkijk niet aantrekkelijk. Dan kan beter via een dataverbinding gebruik worden gemaakt van een ander type computer; er komen steeds meer service-centra waar dergelijke computers voor dat doel beschikbaar zijn. Als de interpretatieve methode op de juiste plaats wordt gebruikt heeft deze echter ook zijn voordelen. Zo is het bij erg korte programma's niet zo belangrijk dat ze relatief langzaam worden uitgevoerd, omdat bij die programma's het compileren, laden en de in- en uitvoer van gegevens de eigenlijke rekentijd bepalen. Dit heeft tot gevolg dat de keus van een hypothetische computer H en de bouw van een compiler voor S op H het probleem van de overdraagbaarheid essentieel kan vergemakkelijken.
Het enige werk dat in dit geval door de ontvanger moet worden gedaan is het programmeren van een efficiënte interpreter voor H op de 'doelmachine'. De compiler geschreven in de brontaal (S) en in gecompileerde vorm (H) vormen

tesamen met een precieze beschrijving van de hypothetische machine H als het ware een bouwdoos (compiler kit). Het transporteren van een compiler met behulp van zo'n bouwdoos is een relatief eenvoudige zaak.



Figuur 21: Transport van M1 naar M2 bij gebruik van M2

In het onderste gedeelte van figuur 22 ziet u tenslotte hoe door gebruik van de interpreter van H op M2 weer een echte compiler geproduceerd kan worden. De lijnen vanuit de compilerkit H/S naar H/M2 beginnen als stippellijnen.
Daarmee wordt aangegeven dat het hier niet gaat om gewone stappen: het compileerprogramma moet door middel van een interpretator van H op M2 worden gecompileerd. Deze stap hoeft uiteraard maar eenmaal uitgevoerd te worden. Aan deze manier van werken kleven helaas weer de nadelen van het tijdverslindende interpretatieve compileren van een groot programma: van de nieuwe compiler zelf.



Figuur 22: Transport van M1 naar M2 via een hypothetische machine H

14 Opgaven

1 Onderzoek de volgende syntaxis:

```
S = A.
A = B | "IF" A "THEN "A" ELSE "A".
B = C | B "+" C | "+" C.
C = D | C "&" D | "&" D.
D = "X" | "(" A ")" "%" D.
```

Bepaal de verzameling van terminale en de verzameling van niet-terminale symbolen.
Bepaal voor elk niet-terminaal symbool de verzameling van begin- en de verzameling van vervolgsymbolen. Construeer de gehele reeks ontledingsstappen voor de volgende zinnen:

```
x + x
(x + x) & (+ & x)
(x & % + x)
IF x + x THEN x & x ELSE % x
TF x THEN IF % x THEN x ELSE x + x ELSE x & x
IF % x THEN x ELSE IF x THEN x + x ELSE % x + x
```

2 Voldoet de syntaxis van opgave 1 aan de regels 1 en 2 (zie hoofdstuk 2) voor het ontleden volgens het 'top-down' principe met een 'lookahead' van slechts één symbool? Zo niet, ontwerp dan een equivalente syntaxis die wel aan deze regels voldoet. Geef deze syntaxis in de vorm van EBNF-produkties, als syntaxisgrafen en als gegevensstructuur voor programma 3.

3 Ontleed het PL/0-programma uit hoofdstuk 7 in zijn syntactische componenten, en wel volgens het principe van de 'top-down parsing'.

4 De volgende produkties definiëren een deel van de eerste versie van de taal Algol 60. In dit deel komen enkele passages voor die voor meer dan één uitleg vatbaar zijn; deze zijn in de latere versie geëlimineerd. Geef deze dubbelzinnigheden met enkele zinnen aan en maak een alternatieve syntaxis waarmee de taal een eenduidige structuur krijgt.

```
expression       = SimpleExpression |
                   SimpleExpression relation SimpleExpression |
                   IF expression "THEN" expression "ELSE" expression.
relation         = "=" | "#".
SimpleExpression = "a" | "b" | "c" | ... "| "z".
statement        = BasicStatement | IfStatement | ForStatement.
IfStatement      = "IF" expression "THEN" statement |
                   "IF" expression "THEN" statement "ELSE" statement.
ForStatement     = "FOR" ForList "DO" statement.
BasicStatement   = "A" | "B" | ... | "Z".
ForList          = "10" | "11" | ... | "19".
```

Onderzoek de volgende zinnén:

```
IF a THEN b ELSE c = d
IF a THEN IF b THEN A ELSE B
IF a THEN FOR 10 DO IF b THEN A ELSE B
```

5 Algol 60 bevat een meervoudige waardetoekenning van de vorm

```
v1 := v2 :=  ... := e
```

We nemen aan dat deze door de volgende syntaxis is gedefinieerd;

```
assignment    = leftpart expression.
leftpartlist  = leftpart | leftpartlist leftpart.
leftpart      = variable ":=".
expression    = variable | expression "+" variable.
variable      = ident | ident "[" expression "]".
```

Hoever moet vooruitgekeken worden (het niveau van 'loookahead') om zinnen met gebruik van deze syntaxis volgens het 'top-down' principe te kunnen analyseren? Maak een alternatieve notatie voor de meervoudige waarde-toekenning waarin de 'lookahead' maar één symbool is.

6 Construeer een programma waarmee een door EBNF-produkties voorgestelde syntaxis wordt gelezen en de verzamelingen van begin-, vervolg- en eind symbolen worden geproduceerd. Neem daarbij aan dat een eindsymbool van een niet-terminaal symbool A een symbool is dat aan het einde van een uit A geproduceerde zin mag staan. Aanwijzing: gebruik het volgende algoritme van Warshall:

```
TYPE  matrix      = ARRAY [1..n], [1..n] OF BOOLEAN;

PROCEDURE ancestor (VAR m : matrix; n : CARDINAL);

(* Initially m [i, j] is TRUE, if individual i is a parent of individual j.
   At completion. m [i, j] is TRUE if i is an ancestor of j *)

VAR   i, j, k    : CARDINAL;

BEGIN
  FOR i := 1 TO n DO
    F0R j := 1 T0 n DO
      TF m[j, i] THEN
        F0R k:=1 TO n DO
          IF m [i, k] THEN m[j, k] := TRUE END
        END
      END
    END
  END
END ancestor
```

7 Verander programma 3 zodanig, dat gecontroleerd wordt of de gelezen syntaxis aan de beperkende regels 1 en 2 voldoet. Programma 3 zou dan bijvoorbeeld niet alleen de syntaxis

```
A = B C.
B = "x" "y".
C = "x" "z".
```

maar ook `A = {"x"} | {"y"}.`

accepteren. Onderzoek eerst op welke manier de parser het in deze gevallen af laat weten.

8 Maak de PL/0-code die door de compiler (programma 8) wordt geproduceerd voor de procedures divide en gcd (hoofdstuk 7).

9 Verbeter het PL/0-systeem uit programma 8 zodanig, dat geen enkele programmeerfout tot het vastlopen van het systeem kan leiden. Let daarbij vooral op de volgende mogelijke problemen: de geheugenruimte is te klein of bij het rekenen komt een te groot getal voor (arithmetic overflow).

10 Verwijder het symbool CALL uit de syntaxis van het PL/0-systeem. Nu wordt niet langer voldaan aan regel 1; pas de syntaxis aan de nieuwe situatie aan. Op grond waarvan wordt beslist or een statement een waardetoekenning of de aanroep van een procedure is?

11 Verander de taal PL/0 in een variant PL/0' waarin het voorwaardelijke en het herhalingsstatement anders zijn geformuleerd:

```
statement      =  ident ":=" expression | "CALL" ident |
                  "IF" guardedStatements {"|" guardedStatements} "FI" |
                  "DO" guardedStatements {"|" guardedStatements} "OD".

guardedStatements  = condition "<-" statement { ";" statement}.
```

Met het nieuwe statement:

```
IF B0 <- S0 | B1 <- S1 | ... | Bn <- Sn FI
```

geven we aan dat een willekeurige voorwaarde wordt gekozen uit alle voorwaarden Bi waaraan voldaan is; het bij die voorwaarde behorende statement wordt uitgevoerd. Als aan geen enkele van de voorwaarden is voldaan wordt het programma afgebroken. Elke reeks statements Si wordt dus alleen maar uitgevoerd als aan de voorwaarde Bi is voldaan.
In zo'n situatie zeggen we dat Si door Bi wordt beschermd; Bi wordt de guard (beschermer) van Si genoemd. Laten we met het statement:

```
   DO   B0 <- S0 | B1 <- S1 | ... | Bn <- Sn   OD
```

aangeven dat één van de statements Si waarvoor aan de conditie (guard) voldaan is wordt uitgevoerd, en wel net zo lang tot aan geen voorwaarde Bi meer is voldaan. De structuur DO ... OD is dus een herhalingsstatement. Pas de syntaxis bij het declareren van procedures op de volgende manier aan de nieuwe situatie aan.

```
 "PROCEDURE" ident ";" statement {";" statement} "END" ";".
```

12 Breid de taal en de compiler PL/0 uit met het gegevenstype BOOLEAN en de logische operatoren AND, OR en NOT. De syntaxis moet zodanig worden aangepast, dat in plaats van voorwaarden (conditions) Boolse uitdrukkingen (expressies) zijn toegestaan. Relaties kunnen dan als Boolse operanden in expressies voorkomen.

13 Breid de taal en de compiler PL/0 uit met het gegevenstype REAL, waarvoor de rekenkundige operatoren +, - , * en / moeten worden gedefinieerd. Kies daarbij uit de volgende alternatieven:

1. Het resultaat van een operatie behoort steeds tot hetzelfde type als de operanden. De typen INTEGER en REAL zijn dan niet te 'combineren'. Er zijn wel conversie-operatoren tussen beide typen.

2. Operanden van de typen INTEGER en REAL zijn naar believen te combineren. Bij waardetoekenningen aan integervariabelen wordt automatisch afgerond. Vergelijk de complexiteit van de twee implementaties.

14 Breid de taal PL/0 uit met arrays; neem daarbij aan dat de grenzen van de indices van een array in de declaratie van de array-variabele worden aangegeven. Neem aan dat de grenzen op de volgende manier als constanten m en n zijn gespecificeerd:

```
VAR a [m : n]
```

15 Breid de taal PL/0 uit met procedures met parameters. Onderzoek de volgende twee mogelijkheden en kies één van de twee voor de implementatie.

1. Waardenparameters (call by value): de actuele parameters in de aanroep van de procedure zijn expressies. Bij de aanroep van de procedure worden deze geëvalueerd, De formele parameters zijn lokale variabelen van de procedure; aan deze variabelen worden bij de aanroep de waarden van de actuele parameters toegekend.

2. Adresparameters (of VAR-parameters; call by reference): de actuele parameters zijn variabelen. Bij de aanroep van de procedure worden de formele parameters vervangen door deze variabelen.
   Zulke parameters worden geimplementeerd door de adressen van de actuele parameters (de variabelen) op te slaan in de geheugencellen die aan de formele parameters zijn toegewezen. De variabelen worden dan indirect geadresseerd. Het is daardoor mogelijk variabelen buiten de procedure door middel van de parameters te veranderen.
   Het is daarom zinvol de regels waarin in de taal het bereik (scope) van variabelen wordt geregeld aan de nieuwe situatie aan te passen: binnen procedures mogen alleen lokale grootheden direct toegankelijk zijn; niet-lokale variabelen moeten via parameters worden bereikt.

16 Verbeter het PL/0 systeem (programma 8) door het invoeren van een zogenaamd 'display'. Daarmee moet de toegang tot variabelen versneld worden. Het display is een array D van indices; de componenten D[i] van dit array geven de plaats (index) van het procedure-segment op niveau i in de stapel aan. Door invoering van dit display is het niet meer nodig de SL-keten te doorlopen om variabelen te vinden: de eenvoudige formule D[i] + a levert het adres van een variabele; hierbij is i het niveau en a de offset van het adres van de variabele.
Daar staat tegenover dat nu bij het aanroepen en verlaten van elke procedure het display bijgewerkt moet worden. Kan het display de SL-keten vervangen of blijft deze keten nodig?

17 Voer de in hoofdstuk 12 behandelde uitbreiding van PL/0 met processen uit. Om een bruikbaar systeem te krijgen moeten tegelijkertijd procesparameters en arrays worden ingevoerd.

18 Verander het PL/0-systeem zodanig, dat in plaats van een stapel in de interpretator een conventionele computer-architectuur wordt gebruikt. Dit betekent in het bijzonder dat er niet langer op het mechanisme van de stapel afgestemde instructies beschikbaar zijn. Neem in plaats daarvan aan dat er meer registers beschikbaar zijn en dat de instructies naast een geheugenadres ook nog een registernummer bevatten dat aangeeft welk register gebruikt moet worden.

19 Verander de compiler zodanig, dat code voor een bestaande computer wordt geproduceerd. Het is voor de eenvoud aan te bevelen om -zeker in de eerste stap- symbolische assembleercode te produceren.

20 Breid programma 5 zodanig uit, dat door het programma elk willekeurig PL/0-programma wordt vertaald in een daarmee corresponderend PL/0'-programma. De syntaxis van PL/0' is te vinden in opgave 11.
Is een eenvoudjge vertaling in de omgekeerde richting eveneens mogelijk?

Literatuur

Aho, A.V. en Ullman, J.D., Principles of Compiler Desjgn, Addison-Wesley, Reading, 1977.

Alblas, H. e.a., Vertalerbouw, Academic Service, Den Haag, 1981

Barron, D.W., Pascal, The Language and its Implementation, Wiley & Sons, Chichester, 1981

Brinch Hanson, P., Operating Systems Principles, Prentice-Hall, Englewood Cliffs, 1973.

Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, 1976

Dijkstra, E.W. en Feijen, W.H.J., Een Methode van Programmeren, Academic Service, Den Haag, 1984

Dijkstra, E.W., Dahl, O.-J. en Hoare, C.A.R., Structured Programming, Academic Press, London and New York, 1974

Early, J. en Strurgis, H., A formalism for translator interactions, Comm. ACM 13, 10 (Oct. 1970), 607-617.

Gries, D., Compiler Construction for Digital Computers, McGraw-Hill, New York, 1971

Kaubisch, W.H., Perrott, R.H. en Hoare, C.A.R., Quasiparallel programming, Software - Practice and Experience 6, 3 (July 1976) 341-356

Knuth, D.E. Top-down syntax analysis, Acta Informatica 1 (1971) 79-110

Pemberton, S. en Daniels, M.C., Pascal Implementation: The P4 Compiler, Ellis Horwood, Chichester, 1982

Wirth, N., Algorithmen und Datenstrukturen, Teubner Verlag, Stuttgart, 1983.

Wirth, N. Programmjng in Modula-2, Springer-Verlag, New York, 1982

Wulf, W. et al., The Design of an Optimizing Compiler, American Elsevier, New York, 1975

Zima, H., Compilerbau, Bibliographisches Institut, Mannheim, 1982