Syllabus
Natuurlijke-Taalverwerking I

Gosse Bouma
Afdeling Informatiekunde
Rijksuniversiteit Groningen
gosse@let.rug.nl

Februari, 2006

Enkele aanpassingen door Gertjan van Noord (februari 2008)

Inhoudsopgave

Hoofdstuk 1

Inleiding

Veel nuttige toepassingen van de computer vereisen dat de computer op een verstandige manier met natuurlijke taal om kan gaan. Voorbeelden van zulke toepassingen zijn automatisch of computer-ondersteund vertalen, computer-ondersteund talenonderwijs, *information-retrieval* (het vinden van relevante informatie in een tekst-database of op het *World Wide Web*), automatische spellingcorrectie, en automatische spraakherkenning. Voor al deze toepassingen zijn programma's nodig die een zekere mate van taalkundige kennis bezitten. De *natuurlijke-taalverwerking* is het vakgebied dat zich bezighoudt met de constructie van dergelijke programma's. Een belangrijk onderdeel van de natuurlijke taalverwerking is automatisch ontleden, d.w.z. het automatisch toekennen van een syntactische structuur aan een zin.

In dit inleidende hoofdstuk geven we een overzicht van het vakgebied. We introduceren ook het begrip *context-vrije grammatica*. Veel programma's voor automatisch ontleden gebruiken een dergelijke grammatica. Tenslotte bespreken we een aantal toepassingen waarbij het gebruik van een computationele grammatica een rol speelt.

1.1 Taal en computer

Een computer uitgerust met de taalvaardigheid van een mens, zou een ongelooflijk nuttig hulpmiddel zijn. Zo'n machine zou elektronische post kunnen lezen, en oninteressante post kunnen verwijderen, zonder dat de geadresseerde er lastig mee wordt gevallen. Zo'n machine zou alle fouten in een tekst kunnen opsporen, en ze kunnen verbeteren. Zo'n machine zou op het *World Wide Web* alle informatie over een bepaald onderwerp kunnen opzoeken, en een samenvatting (in het Nederlands natuurlijk) kunnen geven van wat het gelezen heeft. Zo'n machine zou je kunnen toespreken (*Verwijder alle bestanden die meer dan twee jaar oud zijn en waarin het woord* `vandaag` *voorkomt.*), en ze zou daarna doen wat je gezegd hebt. Je zou ook rustig een praatje over het weer of over de krant van gisteren kunnen beginnen. Zulke machines bestaan helaas niet, en het is ook niet waarschijnlijk dat er binnenkort machines of programma's zullen bestaan die dit allemaal kunnen en even goed kunnen als mensen. Aan de andere kant wordt er op

sommige deelgebieden wel degelijk veel vooruitgang geboekt.

Het gebruik van computers als hulpmiddel om teksten te redigeren, te reproduceren, te publiceren, of te vertalen is wijd verbreid. Computers kunnen enorme hoeveelheden data opslaan en snel verwerken. Zo kunnen woorden of zinsneden efficiënt terug worden gevonden in een tekst, een woordenboek, of zelfs een electronisch tekst-*web* als het *World Wide Web*. Woordenboekmakers kunnen in grote tekstbestanden met behulp van een computer eenvoudig nieuwe woorden opsporen of een index laten maken waarin van het aantal voorkomens van ieder woord in de tekst staat vermeld. Een tekstverwerker uitgerust met een redelijke woordenlijst van het Nederlands kan een eenvoudige vorm van spellingcorrectie uitvoeren. Zo'n woordenlijst wordt door sommige gebruikers benut als hulpmiddel bij het oplossen van kruiswoordraadsels of als rijmwoordenboek. Al deze taken vereisen dat een computer met taal omgaat, maar er is geen sprake van dat de computer 'begrijpt' waar de tekst in kwestie over gaat.

De *natuurlijke-taalverwerking*[1] houdt zich bezig met de vraag hoe de kloof tussen deze twee uitersten verkleind kan worden. Er zijn toepassingen te bedenken die relatief eenvoudig lijken, maar die toch vereisen dat enige taalkundige kennis aan het programma wordt toegevoegd. Het *afbreken van woorden* vereist bijvoorbeeld, naast een woordenlijst, enige informatie over de regels voor het opdelen van (Nederlandse) woorden in samenstellende delen en lettergrepen. Het gebruik van *regels*, naast of in plaats van een lijst met woorden en hun afbreekposities, maakt het mogelijk ook onbekende woorden op de juiste manier af te breken. Een programma voor *spellingcorrectie* dat in staat is te controleren of het onderwerp van de zin in persoon en getal overeenkomt met de persoonsvorm, en dat kan herkennen waar een voltooid deelwoord hoort te staan, zou veel fouten in de schrijfwijze van werkwoorden kunnen opsporen. Daarnaast kunnen toepassingen die in principe te 'moeilijk' zijn (omdat ze tekstbegrip vereisen) soms worden opgelost met methoden waarin tekstbegrip geen rol speelt. *Elektronische post sorteren* kan bijvoorbeeld ook door te kijken naar de afzender, of door te zoeken naar bepaalde woorden in de inhoud. Soms is het ook mogelijk een programma te ontwikkelen dat het algemene, 'moeilijke' probleem weliswaar niet oplost, maar dat wel overweg kan met een specifieke instantie van het probleem. Een willekeurige tekst *automatisch vertalen* is moeilijk, maar een programma schrijven dat alleen weerberichten vertaalt is vrij eenvoudig. Een *dialoogsysteem* maken dat alleen informatie geeft over de aankomsten vertrektijden van treinen is met de huidige stand van technologie goed mogelijk, maar dat betekent niet dat er systemen bestaan waarmee dialogen over een willekeurig onderwerp mogelijk zijn. Het zijn vooral toepassingen waarbij een beperkte hoeveelheid taalkundige kennis een rol speelt, die het domein vormen van de natuurlijke-taalverwerking.

[1] Volgens Google is de schrijfwijze met en zonder verbindingsstreepje ongeveer even gangbaar. Omdat we met dit begrip (een vertaling van het Engelse *natural language processing*) de verwerking van natuurlijke taal bedoelen, en niet de natuurlijke verwerking van taal, verdient de schrijfwijze met streepje de voorkeur.

Een interdisciplinair vak

De natuurlijke-taalverwerking maakt voornamelijk gebruik van inzichten uit de *taalkunde* en de *informatica*.

De *fonologie* en de *fonetiek* zijn onderdelen van de taalkunde, die zich bezighouden met de manier waarop woorden en zinnen worden uitgesproken en met de regels die hierbij een rol spelen. Zulke kennis is van belang voor toepassingen die iets met spraak doen, zoals spraakherkenners (d.w.z. het omzetten van geluid in tekst) of programma's voor spraaksynthese (d.w.z. het omzetten van tekst in spraak).

De *morfologie* houdt zich bezig met de structuur van woorden. Zulke kennis is nuttig omdat in het algemeen geen woordenlijst volledig is. De officiële woordenlijst van het Nederlands, het Woordenlijst Nederlandse Taal, bevat slechts zo'n 125.000 woordvormen. Een woordenboek zoals Van Dale bevat misschien wel 500.000 verschillende woordvormen. Wanneer je echter de woorden in een willekeurige tekst vergelijkt met die in een woordenlijst, zul je snel ontdekken dat er woorden in de tekst staan die niet te vinden zijn in de lijst.[2] Dit wordt niet alleen veroorzaakt door het gebruik van namen (voor personen, organisaties, geografische locaties, e.d.), maar ook doordat een taal als het Nederlands samenstellingen kent. Dit zijn woorden die zijn gevormd door twee woorden aan elkaar te plakken (en te schrijven). Taalgebruikers vormen voortdurend nieuwe samenstellingen. Het woord *dialoogsysteem* is bijvoorbeeld een samenstelling van de woorden *dialoog* en *systeem*, dat waarschijnlijk in geen woordenboek te vinden is. Een programma dat kennis van de morfologie gebruikt om te bepalen of een woord eventueel een samenstelling zou kunnen zijn, is nuttig, omdat op die manier ook voor woorden die niet in het woordenboek staan kan worden bepaald of het bijvoorbeeld een werkwoord of een zelfstandig naamwoord is (de woordsoort van een samenstelling is namelijk gelijk aan de woordsoort van het achterste deel van de samenstelling).

De *syntaxis* houdt zich bezig met de regels voor de zinsbouw. Voor Nederlandse hoofdzinnen geldt bijvoorbeeld dat de persoonsvorm altijd het eerste of tweede zinsdeel is, terwijl in bijzinnen de persoonsvorm vaak achteraan in de zin staat. Met behulp van syntactische regels kan ook worden vastgesteld wat het onderwerp en het lijdend voorwerp van een zin is, wat de bijvoeglijke bepalingen en de bijwoordelijke bepalingen zijn, etc. Zulke kennis is nuttig voor het automatisch corrigeren van grammaticale fouten, voor dialoogsystemen (die de betekenis van een zin bepalen op basis van syntactische analyse), en voor automatisch vertalen. Deze toepassingen bespreken we hierna in meer detail.

De *semantiek* en de *pragmatiek* houden zich bezig met de betekenis van zinnen, en de manier waarop we taal gebruiken om informatie uit te wisselen, vragen te stellen, misverstanden op te lossen, etc. Kennis uit deze vakgebieden is met name nuttig voor dialoogsystemen.

De natuurlijke-taalverwerking maakt ook gebruik van kennis en technieken uit de

[2] Ordelman e.a. (2001) geven precieze cijfers. In een woordenlijst van de 60.000 meest frequente woorden (in een corpus van 125 miljoen woorden), zijn bijvoorbeeld 3.5% van de woorden uit een willekeurige tekst niet terug te vinden.

informatica. Om taalkundige regels te kunnen toepassen zijn algoritmen en programma's nodig. De natuurlijke-taalverwerking maakt hierbij gebruik van kennis die vaak in eerste instantie binnen de informatica is ontwikkeld. *Eindige automaten* (*finite state automata*) zijn zeer efficiente programma's voor het analyseren van strings (reeksen symbolen, bijvoorbeeld letters). Binnen de bio-informatica worden zulke automaten bijvoorbeeld gebruikt voor de analyse van DNA-strings. Binnen de natuurlijke-taalverwerking worden eindige automaten gebruikt voor het snel opzoeken van woorden in een woordenlijst, voor het analyseren en afbreken van woorden, voor het omzetten van letters in klanken, en voor een aantal andere toepassingen. Binnen de informatica is ook veel onderzoek gedaan naar de verwerking van programmeertalen. Een deel van deze kennis is ook van toepassing op de verwerking van natuurlijke talen. Een *compiler* is een programma dat programmacode omzet in machinetaal, en dat herkent waar er eventueel fouten in de code staan. Dit betekent dat expressies (zinnen) in het programma moeten worden herkend en geanalyseerd. Dit proces is in principe vergelijkbaar met het verwerken van zinnen in natuurlijke taal, alhoewel natuurlijke taal vele malen complexer is dan een programmeertaal. De *parseertechnieken* die zijn ontwikkeld voor programmeertalen zijn daarom ten dele ook bruikbaar om natuurlijke taal syntactisch te analyseren.

Naast informatica en taalkunde spelen gebieden als de *kunstmatige intelligentie* en de *informatiekunde* een rol binnen de natuurlijke-taalverwerking. Je zou kunnen stellen dat de natuurlijke-taalverwerking zich uiteindelijk ten doel stelt de menselijke taalvaardigheid zo goed mogelijk na te bootsen met behulp van een computer. Volgens die definitie zou de natuurlijke-taalverwerking een deelgebied van de kunstmatige intelligentie zijn, die zich immers ten doel stelt menselijke intelligentie in het algemeen na te bootsen. Technieken en inzichten uit de kunstmatige intelligentie spelen ook een rol in de natuurlijke taalverwerking. Om tekst te kunnen begrijpen is vaak kennis van de wereld nodig. Dit is bij uitstek een KI-onderwerp.

Het gebruik van (grote) hoeveelheden taaldata (tekst, spraak, taaldata) en het gebruik van statistische technieken om informatie aan deze data te ontlenen (over de frequentie van woorden, van woordsoorten, van bepaalde woordvolgordes, over de meest frequente vertaling van een woord, etc.) is ongetwijfeld de belangrijkste ontwikkeling binnen de natuurlijke-taalverwerking van de afgelopen 15 jaar. Veel van de statistische technieken die worden toegepast, worden ook gebruikt binnen de *machine learning*, het onderdeel van de KI dat zich bezighoudt met het leren van regels op basis van data.

De informatiekunde is vooral belangrijk omdat ze technieken levert (m.n. XML) waarmee grote hoeveelheden data (woordenboeken, grote tekstbestanden voorzien van syntactische informatie) gecodeerd en doorzocht kunnen worden. Ook veel van de mogelijke toepassingen van natuurlijke-taalverwerking liggen in de sfeer van de informatiekunde. De opkomst van internet en het World Wide Web heeft geleid tot een hernieuwde belangstelling voor natuurlijke-taalverwerking. Een belangrijke reden is de toegenomen vraag naar technieken om effectief te kunnen zoeken in grote hoeveelheden (weinig gestructureerde) informatie. Sommige van deze technieken veronderstellen natuurlijke-taalverwerking. *Informatie-extractie* is een techniek waarbij men documenten naar van tevoren bepaalde informatie, bijvoorbeeld de gegevens van vacatures die op het internet gepubliceerd zijn. Om zulke informatie (naam van de functie, vereiste opleiding, naam van het bedrijf, etc.) te vinden in tekst kan gebruik worden gemaakt van natuurlijke-taalverwerking. *Question-answering* is een vorm van zoeken in tekst of op het internet, waarbij de gebruiker een vraag stelt in natuurlijke taal, en het systeem een precies antwoord geeft (i.p.v. een lijst met URL's die wellicht het antwoord bevatten). Hierbij speelt natuurlijke-taalverwerking een cruciale rol.

Het vakgebied dat zich bezig houdt met natuurlijke taal en computers wordt soms ook wel aangeduid als *computationele taalkunde*. Dit betekent vaak hetzelfde als natuurlijke-taalverwerking. Soms wordt er echter een verschil tussen beide begrippen gemaakt, waarbij natuurlijke-taalverwerking dan staat voor toegepast onderzoek, terwijl de computationele taalkunde dan het theoretische onderzoek aanduidt. In deze visie is de computationele taalkunde de subdiscipline van de taalwetenschap die zich bezig houdt met het bestuderen van computationele eigenschappen van taal en van theorieën over taal. Een implementatie van een theorie kan vaak erg nuttig zijn voor het opsporen van fouten in de regels of ontbrekende regels. Met behulp van een implementatie kun je een theorie testen op data (woordenlijsten, willekeurige zinnen, etc.). Dit brengt meestal meteen een aantal problemen en ontbrekende regels in de theorie aan het ligt. Je zou je daarnaast bijvoorbeeld kunnen afvragen of een bepaalde syntactische theorie een verklaring biedt voor het feit dat mensen taal zeer snel kunnen verwerken. Wanneer de beste parsers voor grammatica's opgesteld volgens de regels van deze theorie computationeel zeer inefficient zijn, bestaat er een discrepantie tussen de theorie en de menselijke taalvaardigheid. De computationele taalkunde kan zulke problemen aan het licht brengen en oplossingen proberen te vinden (door bijvoorbeeld de theorie zo in te perken dat de computationele problemen verdwijnen).

In deze syllabus besteden we vooral aandacht aan de automatische syntactische analyse van taal. Dit betekent dat we ons vooral richten op computationele grammatica's en programma's voor het automatisch ontleden van zinnen met behulp van zo'n grammatica. Een zeer leesbare inleiding in de computer-taalkunde is Brandt Corstius (1978). Een breder en recenter, Engelstalig, overzicht van de natuurlijke-taalverwerking is te vinden in Jurafsky en Martin (2000).

1.2 Context-vrije Grammatica

Volgens de Nederlandse computertaalkundige Brandt Corstius is de grootste verdienste van de Amerikaanse taalkundige Chomsky dat hij ervoor gezorgd heeft dat boeken en artikelen over taalkunde voor de leek niet langer te begrijpen zijn.[3] De observatie die aan deze opmerking ten grondslag ligt is het feit dat de taalkunde in de afgelopen 40 jaar aanzienlijk abstracter is geworden, en dat het gebruik van wiskundige hulpmiddelen in

[3] Volgens anderen is de grootste bijdrage van Chomsky de introductie van zinnen voorzien van een sterretje (*) om ongrammaticaliteit aan te duiden. Door ook dergelijke zinnen in het onderzoek te betrekken is het object van onderzoek dramatisch in omvang toegenomen.

S	→	NP VP	Det	→	*een*
NP	→	Det N	Det	→	*het*
N	→	A N	N	→	*eendje*
VP	→	V	N	→	*ei*
VP	→	V NP	V	→	*legt*
			A	→	*lelijke*

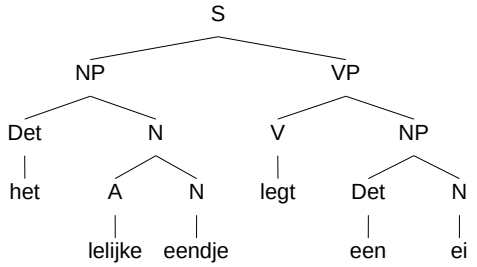
Figuur 1.1: Herschrijfregels voor een fragment van het Nederlands

veel gebieden van de taalkunde normaal is geworden. De taalkunde is daarmee van een traditioneel letterenvak geworden tot iets dat sterke overeenkomsten vertoont met sommige toegepaste exacte wetenschappen.

Voor de natuurlijke taalverwerking is deze ontwikkeling van levensbelang. Wanneer we taal willen automatisch willen analyseren met behulp van een computer, vereist dit modellen van taal die precies zijn, en waarvan de wiskundige eigenschappen duidelijk zijn. Het gebied waar het verband tussen taalkundige theorieën en computationele methoden het duidelijkst is, is de syntaxis en de computationele tegenhanger daarvan, het *automatisch ontleden*. Aan deze beide onderwerpen zullen we in deze syllabus dan ook de meeste aandacht besteden.

Doel van de *syntaxis* is het opstellen van regels die het mogelijk maken grammaticale van ongrammaticale zinnen te onderscheiden. Daarnaast probeert de syntaxis de structuur van zinnen in kaart te brengen. Onderzoek naar de structuur van taal kan bijvoorbeeld een antwoord bieden op de vraag of er zoiets als een *universele grammatica* bestaat, hoe taal door kinderen geleerd wordt, en wat de belangrijkste verschillen tussen talen zijn. Om op deze zeer abstracte vragen concrete antwoorden te krijgen is een abstract en wiskundig model van taal noodzakelijk. Toonaangevend op dit gebied is het werk van Noam Chomsky, die in de jaren vijftig en zestig een taalmodel voorstelde waarin de grammaticale zinnen van een taal worden geproduceerd (gegenereerd) op basis van een aantal abstracte regels.

De eenvoudigste van deze regels is de zogenaamde *herschrijfregel* die zegt dat een categorie C mag worden herschreven als een rijtje categorieën $\mathsf{C}_1 \ldots \mathsf{C}_n$. Voorbeelden van herschrijfregels voor een klein stukje van het Nederlands vind je in figuur 1.1. Met behulp van deze regels kunnen we bijvoorbeeld de categorie S herschrijven als de reeks NP VP. Binnen deze reeks kunnen we NP herschrijven als Det N, hetgeen de reeks Det N VP oplevert. N kunnen we herschrijven als A N, resulterend in de reeks Det A N VP. Det, A en N kunnen we herschrijven als respectievelijk *het*, *lelijke* en eendje, hetgeen de reeks *het lelijke eendje* VP oplevert. Herschrijven we nu VP als V NP, en NP als Det N, dan krijgen we de reeks *het lelijke eendje* V Det N. Herschrijven de laatste drie categorieën als respectievelijk *legt*, *een* en *ei*, dan krijgen we uiteindelijk de reeks *het lelijke eendje legt een ei*. Deze laatste reeks kunnen we beschouwen als *een zin, gegenereerd door de regels in figuur 1.1*. De reeks is ontstaan vanuit de categorie S, die we als categorie voor zinnen (*sentences*) hanteren, en bevat alleen maar woorden van het Nederlands



Figuur 1.2: Boomstructuur voor *het lelijke eendje legt een ei*

(en geen elementen meer die nog zouden kunnen worden herschreven). Het is vaak handig het proces van herschrijven niet stapsgewijs te beschrijven, maar te kiezen voor een representatie waarin wordt geabstraheerd van de precieze volgorde waarin de herschrijfstappen zijn uitgevoerd, en waarin alleen wordt aangegeven *hoe* iedere categorie is herschreven. Dit levert de bekende boomstructuren op (zie figuur 1.2).

Wanneer een grammatica slechts gebruik maakt van herschrijfregels van de vorm C $\rightarrow \mathsf{C}_1 \ldots \mathsf{C}_n$ ($n \leq 0$), waarbij precies één categorie C wordt herschreven tot 0 of meer dochters, noemen we dit een *contextvrije grammatica*. Chomsky veronderstelt dat dergelijke grammatica's te eenvoudig zijn om de grammatica's van natuurlijke taal goed te beschrijven, maar binnen de computationele grammatica zijn dergelijke grammatica's toch populair. Dit omdat er een groot aantal technieken voor het automatisch ontleden van context-vrije grammatica's bestaat. Bovendien is context-vrije grammatica vaak voldoende om een behoorlijk groot deel van een taal te kunnen beschrijven. Voor die fenomenen die niet eenvoudig of zelfs helemaal niet met context-vrije regels te beschrijven zijn, stelt Chomsky het gebruik van transformaties voor. Omdat transformaties (regels die boomstructuren omzetten in andere boomstructuren) grote problemen opleveren voor automatisch ontleden, is deze oplossing binnen de natuurlijke-taalverwerking niet populair. Als alternatief zijn er technieken ontwikkeld die geen gebruik maken van transformaties, maar die wel grammatica's opleveren die goed te parseren zijn. Vrijwel al deze technieken zijn gebaseerd op het gebruik van *unificatie*, een operatie die bijvoorbeeld ook beschikbaar is binnen het *definite clause grammar* formalisme (zie hoofdstuk 2).

1.2.1 Meer regels voor het Nederlands

In figuur 1.3 en figuur 1.4 geven we een grammatica voor een iets groter fragment van het Nederlands.

De categorie Int staat voor *intensifier*, daarmee bedoelen we bijwoorden die aan een bijvoeglijk naamwoord vooraf kunnen gaan (*erg*, *verschrikkelijk*, *enorm*, ...):

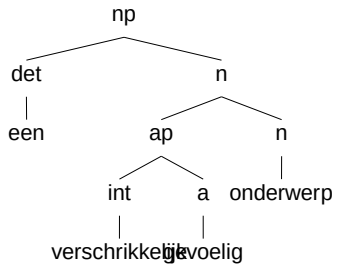
NP	→	Det N	*het eendje*
N	→	AP N	*lelijk eendje, erg vervelende toestand*
N	→	N PP	*auto met een benzinemotor*
AP	→	A	*vervelend*
AP	→	Int A	*erg vervelend, verschrikkelijk gevoelig*
PP	→	P NP	*met een auto*
VP	→	VC	*slapen, heeft geslapen*
VP	→	NP VP	*een boek lezen*
VP	→	PP VP	*van voetbal houden*
VP	→	VP PP	*houden van voetbal*
VP	→	Adv VP	*morgen voetballen*
VP	→	VP CP	*zegt dat Jan van voetbal houdt*
VC	→	V	*slaapt*
VC	→	Aux V	*heeft geslapen*
VC	→	Mod V	*moet slapen*
VC	→	V Aux	*gelapen heeft*
VC	→	V Mod	*slapen moet*
VC	→	ϵ	*(Jan leest een boek)* ϵ
CP	→	Compl NP VP	*dat Jan een boek heeft gelezen*
S	→	NP Aux VP	*Jan heeft een boek gelezen*
S	→	NP Mod VP	*Jan moet een boek lezen*
S	→	NP V VP	*Jan leest een boek*

Figuur 1.3: Herschrijfregels voor een fragment van het Nederlands.

A	→	*Engelse, gevoelig, goed, groot, politieke, ...*
Adv	→	*flink, hier, niet, wel, gisteren, steeds, zesmaal, ...*
Aux	→	*heeft, hebben, had, hadden, is, zijn, was, waren, ...*
Compl	→	*dat, of*
Det	→	*de, het, een, zijn, deze, geen, dit, ...*
Int	→	*erg, ontzettend, verschrikkelijk, ...*
Mod	→	*wil, wilde, moet, moest, ...*
N	→	*Amerikaan, dochtertje, familie, overleg, roeiers, vrede, ...*
NP	→	*Amsterdam, Feijenoord, Luis Ocaña, Wall Street, ...*
P	→	*van, in, op, voor, aan, met, door, ...*
V	→	*slaapt, opgroeit, verlopen, wonnen,...*

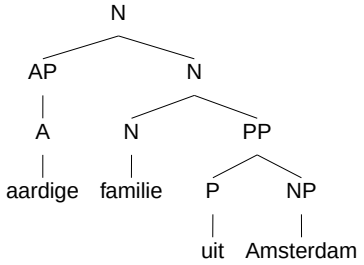
Figuur 1.4: Woordenboek voor de grammatica in figuur 1.3. Om het woordenboek overzichtelijk te houden schrijven we A → *groot, goed, nieuw, ...* i.p.v. A → *groot*, A → *goed*, A → *nieuw*, ... De comma in de rechterkant van deze regels moet dus als 'of' gelezen worden.

(1)

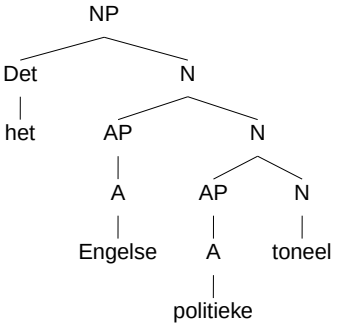


De regel die N herschrijft tot AP N en de regel die N herschrijft tot N PP, heeft N als linkerkant, maar bevat N ook als een rechterdochter. Dit betekent dat deze regels *recursief* zijn. Een N kan dus een willekeurig aantal (0 of meer) bijwoordelijke bepalingen of voorzetselbepalingen bevatten:

(2) a.



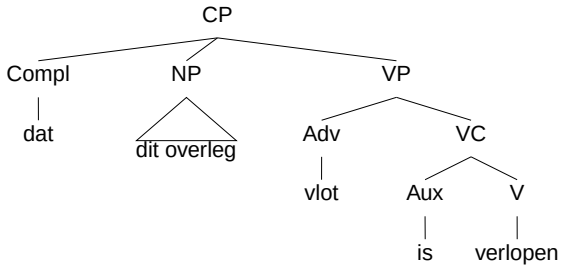
b.



VC staat voor *verb cluster*. Dit is het rijtje van één of meer werkwoorden dat achteraan in de zin staat. Een bijzin wordt i.h.a. ingeleid door een onderschikkend voegwoord, ofwel een *complementizer* (Compl). CP staat voor *complementizer phrase*, oftewel een bijzin. Aux en Mod staan voor *auxiliary* en *modal*, respectievelijk hulpwerkwoorden van tijd en

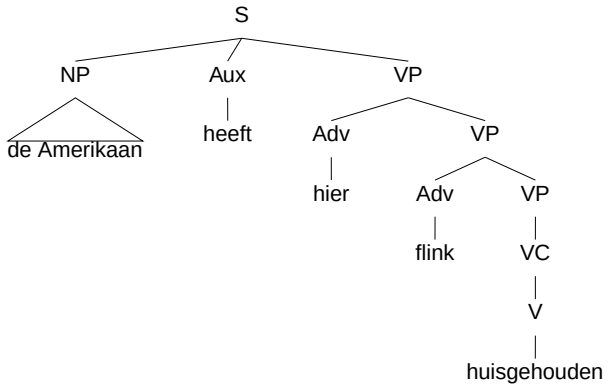
modaliteit.

(3)

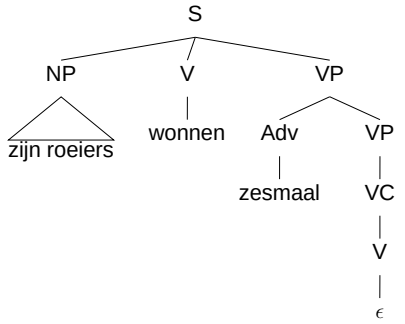


Hoofdzinnen bestaan, volgens de regels in dit fragment, uit een NP (het onderwerp), een persoonsvorm (dat een hulpwerkwoord van tijd of modaliteit of een hoofdwerkwoord kan zijn), en een VP. Wanneer de persoonsvorm een hoofdwerkwoord is, is het werkwoordscluster leeg.

(4) a.



b.



1.2.2 Enkele eigenschappen van grammatica's

De ideale grammatica voor een taal voldoet aan het zogenaamde *all-and-only* principe. Dit wil zeggen dat de grammatica alle zinnen van de taal herkent, en van een goede analyse voorziet (de *all*-conditie) en dat de grammatica geen ongrammaticale zinnen accepteert (de *only*-conditie). Wanneer een grammatica niet aan de *only*-conditie voldoet, overgenereert de grammatica. Wanneer de grammatica niet aan de *all*-conditie voldoet, ondergenereert de grammatica. Een grammatica *overgenereert* dus wanneer er zinnen worden geaccepteerd die niet grammaticaal zijn. De regels in figuur 1.3 overgenereren. Ten eerste wordt er niets gezegd over de vorm van bijvoeglijke naamwoorden. De combinatie *lang afstanden* is echter ongrammaticaal. De regels zoals ze nu zijn gegeven maken geen onderscheid tussen *lange* en *lang* (beide vormen een AP), waardoor de grammatica overgenereert. Hetzelfde geldt voor het verschil tussen *de* en *het* of de relatie tussen de persoonsvorm en de vraag of het onderwerp enkelvoud of meervoud is. Dergelijke details kunnen in principe wel verantwoord worden door meer gedetailleerde regels te schrijven (met aparte regels voor adjectieven met en zonder *-e*-uitgang, en aparte regels voor zelfstandige naamwoorden die *de* en zelfstandige naamwoorden die *het* vereisen. In hoofdstuk 2 zullen we zien dat er een betere oplossing bestaat, waarbij het vermenigvuldigen van regels wordt vermeden.

Een grammatica *ondergenereert* wanneer niet alle grammaticale zinnen door de grammatica kunnen worden gegenereerd of herkend. Zinnen met drie opeenvolgende werkwoorden (*heb willen lezen*) kunnen door de grammatica in figuur 1.3 niet herkend worden. Ook hoofdzinnen waarin geen hulpwerkwoord of modaal werkwoord aanwezig is kunnen niet door de grammatica herkend worden. Een enigszins realistische grammatica voor het Nederlands zal veel meer regels bevatten dan het fragment in figuur 1.3. In systemen die zich strikt beperken tot het gebruik van context-vrije grammatica is het niet ongebruikelijk om te werken met grammatica's die duizenden regels bevatten. Grammatica's die gebruik maken van *unificatie* zijn in het algemeen compacter. De Alpino-grammatica voor het Nederlands maakt bijvoorbeeld gebruik van zulke re-

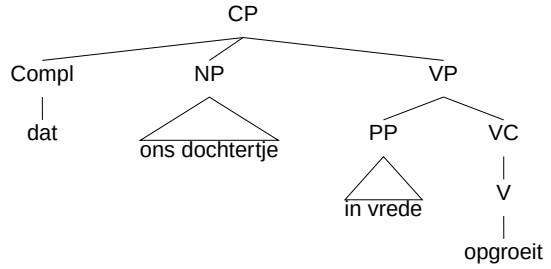
gels. De grammatica bevat ongeveer 400 regels. Hiermee kan een groot deel van het Nederlands geanalyseerd worden.

De grammatica zal sommige zinnen op meer dan één manier kunnen analyseren. Zulke zinnen zijn, volgens de grammatica, *ambigu*. Dit geldt bijvoorbeeld voor zinnen die de reeks NP PP V bevatten. Bekijk bijvoorbeeld de zinnen in (5)

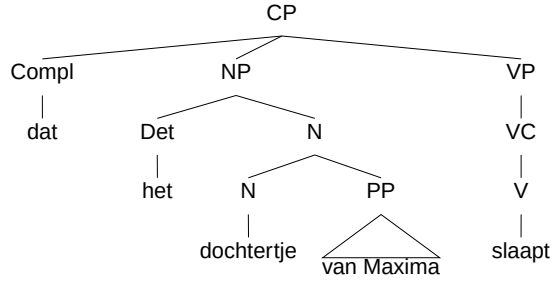
(5) a. Wij willen dat ons dochtertje in vrede opgroeit
b. Wij hopen dat het dochtertje van Maxima slaapt.

In zin (5-a) is *in vrede* een bepaling bij *opgroeien*. Voor de grammatica in 1.3 betekent dit dat de analyse in (6) de juiste is. Voor zin (5-b) geldt dat de PP *van Maxima* een bepaling bij *dochtertje* is. Dit betekent dat in dit geval de analyse in (7) de juiste is.

(6)



(7)



Een taalkundige heeft toegang tot de betekenis van woorden en zinnen, en kan op basis daarvan beargumenteren dat de analyse in (6) correct is voor zin (5-a), en dat de analyse in (7) correct is voor zin (5-b). Voor de grammatica geldt echter dat beide zinnen de reeks Compl Det N P N V bevatten. Volgens de grammatica zijn daarom voor beide zinnen beide analyses mogelijk.

Ambiguïteit is één van de centrale problemen in de natuurlijke-taalverwerking. Realistische grammatica's bevatten over het algemeen een groot aantal regels. Bovendien maken ze gebruik van grote woordenboeken, waarin veel woorden meer dan eens voorkomen (het woord *werk* kan bijvoorbeeld zowel een zelfstandig naamwoord als een

werkwoord zijn). Een onvermijdelijk gevolg hiervan is dat de ambiguïteit toeneemt. Ambiguïteit heeft bovendien de neiging zich explosief te vermeerderen. Wanneer een zin twee delen bevat die beide op twee manieren geanalyseerd kunnen worden, zal de zin als geheel 4 mogelijke analyses (ook wel *lezingen* genoemd) kennen (2×2). Wanneer een zin twee delen bevat die beide 3 mogelijke analyses bevat, zal de zin als geheel 9 analyses kennen. Een klein aantal locale ambiguïteiten kan dus leiden tot een groot aantal ambiguïteiten voor de zin als geheel.

Voor realistische grammatica's (zoals de genoemde Alpino-grammatica) is het niet ongewoon dat een zin duizenden mogelijke analyses heeft. Voor de practische toepasbaarheid van natuurlijke-taalverwerking is dit duidelijk een probleem. Het kost veel tijd alle lezingen uit te rekenen, en het blijft onduidelijk welke lezing de juiste is voor een gegeven zin. Een oplossing voor dit probleem, die in vrijwel alle grote grammatica's gebruikt wordt, maakt gebruik van statistiek. Wanneer we een grote hoeveelheid tekst syntactisch analyseren, zal blijken dat PP's die het voorzetsel *van* bevatten vrijwel altijd bepalingen bij een zelfstandig naamwoord vormen. Dit suggereert dat de *meest waarschijnlijke* analyse van *het dochtertje van Maxima slaapt* de analyse is waarbij de PP een bepaling bij *dochtertje* is. Voor PP's met *in* zijn de cijfers waarschijnlijk niet zo duidelijk wanneer we naar NP's en werkwoorden in het algemeen kijken. Wanneer we echter naar de combinatie van het werkwoord *opgroeien* en *in*-PP's kijken, zal blijken dat *opgroeien* relatief vaak met een *in*-PP voorkomt. Dat suggereert dat *ons dochtertje in vrede opgroeit* een PP bevat die als bepaling bij *opgroeien* gezien moet worden. Met behulp van statistische modellen van taal en taalgebruik, gebaseerd op de analyse van grote hoeveelheden tekst, kan dus de meest waarschijnlijke analyse van een zin gekozen worden. Natuurlijk is deze techniek niet feilloos, maar dit is, met de huidige stand van techniek, wel de meest effectieve techniek om ambiguïteit in natuurlijke taal te temmen.

1.3 Toepassingen

In deze sectie bespreken we enkele toepassingen van natuurlijke-taalverwerking waar grammaticale analyse een rol speelt. We beginnen met een overzicht van Alpino, een grote computationele grammatica voor het Nederlands.

1.3.1 De Alpino-grammatica

De Alpino-grammatica (van der Beek *et al*, 2002)[4] is een grote computationele grammatica voor het Nederlands, die het mogelijk maakt grote hoeveelheden tekst automatisch van een analyse te voorzien. De grammatica is *robuust*, dat wil zeggen dat ze geen eisen stelt aan de aard van de te analyseren tekst (dit kan een willekeurig stuk krantentekst zijn, maar ook tekst uit een encyclopedie, of zelfs (transcripties van) gesproken taal).

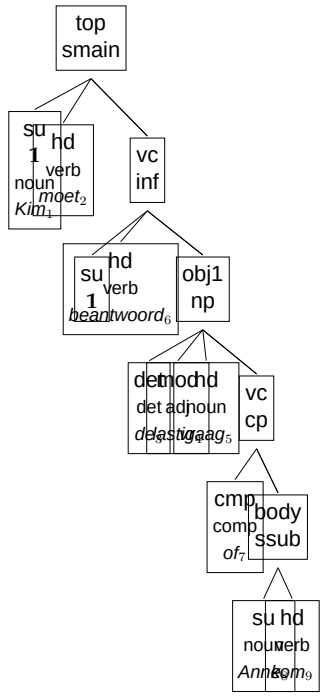
[4] Zie `www.let.rug.nl/~vannoord/alp` voor een *on-line* demo

Syntactische analyse is *efficiënt*. Korte zinnen kosten vaak minder dan een seconde cputijd. Het systeem is efficiënt genoeg om het mogelijk te maken grote tekstbestanden volledig te analyseren. Met de Alpino-parser is tot nog toe de integrale tekst van enkele volledige jaargangen van diverse kranten geanalyseerd (in totaal meer dan 100 miljoen woorden). Het systeem is *accuraat*. Dit betekent dat het systeem voor korte zinnen vaak de juiste analyse geeft, en voor lange zinnen vaak een analyse geeft dit weinig afwijkt van de juiste analyse.

Het ontwikkelen van een grammatica die het Nederlands als geheel beschrijft is geen sinecure. De meeste theoretische taalkundigen beperken zich tot grammatica's die alleen bepaalde fenomenen (bijvoorbeeld passieve zinnen, of zinnen met wederkerende voornaamwoorden) in detail beschrijven, maar die zeker niet de pretentie hebben de taal als geheel te beschrijven. Er zijn wel grammaticale beschrijvingen van het Nederlands als geheel (bijvoorbeeld in de Algemene Nederlandse Spraakkunst (ANS), Haesereyn et al., 1997), maar deze zijn vooral beschrijvend, en daardoor niet expliciet genoeg om te dienen als basis voor een computationele grammatica.

De Alpino-grammatica gebruikt inzichten uit de theoretische taalkunde om tot een grammatica te komen die wel het hele Nederlands afdekt, en waarmee bovendien efficiënt gerekend kan worden. De grammatica bestaat uit ongeveer 400 herschrijfregels, en een groot woordenboek, van ongeveer 50.000 woorden. De regels maken gebruik van een versie van *definite clause grammar*. Het voornaamste voordeel van dit formalisme boven context-vrije grammatica is het feit dat er minder (maar wel abstractere) regels nodig zijn, en dat de regels ook een aantal taalkundige fenomenen kunnen beschrijven die te moeilijk zijn voor context-vrije grammatica. Het woordenboek is gemaakt op basis van algemene, elektronische, woordenboeken. Het woordenboek bevat alleen de meest voorkomende woorden van het Nederlands. Wanneer een tekst woorden bevat die niet in het woordenboek staan, zal het systeem zelf proberen de syntactische categorie van deze woorden te bepalen.

De *output* van de Alpino-grammatica is geen gewone syntactische structuur zoals bij een context-vrije grammatica. De grammatica levert zogenaamde *dependency trees* op. Dit zijn boomstructuren waarin voor ieder zinsdeel, naast de categorie, ook de grammaticale functie is aangegeven. Een voorbeeld staat in figuur 1.5. Het woord *Kim* is het onderwerp van de zin, en heeft daarom de grammaticale functie SU. Het *hoofd* van de zin (in dit geval de persoonsvorm) is *moet*. Dit is aangegeven met de functie HD. *de vraag beantwoorden of...* wordt gezien als een verbale constituent (aangegeven met VC). Het onderwerp van de zin, *Kim*, functioneert ook als onderwerp van deze verbale constituent (degene die eventueel iets beantwoord is Kim). Dit wordt aangegeven door *Kim* de index **1** te geven, en een knoop met dezelfde index, maar zonder lexicale inhoud toe te voegen aan de verbale constituent. Het belangrijkste verschil tussen een context-vrije boom en een *dependency tree* is het feit dat constituenten in een *dependency tree* niet altijd de woordvolgorde in de zin weerspiegelen. De reeks *de lastige vraag of Anne komt* wordt in de *dependency tree* gezien als een NP, maar deze reeks komt niet als zodanig voor in de zin. De taalkundige intuïtie hierachter is dat in talen als het Nederlands zinsdelen gemakkelijk gesplitst kunnen worden. Door een *dependency tree* te geven wordt expli-



Figuur 1.5: Dependency tree voor de zin *Kim moet de lastige vraag beantwoorden of Anne komt*.

ciet gemaakt welke delen van een zin een eenheid vormen, ook als ze niet naast elkaar staan.

Het besluit om binnen de Alpino-grammatica gebruik te maken van *dependency trees* is in belangrijke mate te danken aan het Corpus Gesproken Nederlands.[5] Dit is een groot project dat tot doel heeft 10 miljoen woorden gesproken Nederlands te verzamelen en voor een deel ook te voorzien van syntactische informatie. Daarvoor wordt gebruik gemaakt van *dependency trees*. Deze aanpak is door Alpino overgenomen.

Met behulp van de Alpino-grammatica is een tekstfragment van ongeveer 140.000 woorden geanalyseerd en door taalkundigen (annotatoren) gecorrigeerd op fouten. Het resultaat is een zogenaamde *treebank*, een verzameling tekst voorzien van syntactische annotatie. De *treebank* kan worden gebruikt om te testen hoe accuraat de grammatica is.

[5] Zie `lands.kun.nl/cgn/`.

Hiervoor wordt een *dependency tree* omgezet in een verzameling *dependency relaties*. De boom in figuur 1.5 bevat bijvoorbeeld de informatie dat *Kim* het subject (SU) van *moet* is, en dat *lastig* een bepaling (*modifier*) bij *vraag* is. Een *dependency relatie* is een drieplaatsige relatie tussen een woord, een dependency label, en een ander woord. Wanneer we de hele boom in relaties vertalen levert dit de volgende verzameling op:

(8)
⟨ moet	su	Kim ⟩	⟨ moet	vc	beantwoord ⟩
⟨ beantwoord	su	Kim ⟩	⟨ beantwoord	obj1	vraag ⟩
⟨ vraag	det	de ⟩	⟨ vraag	mod	lastig ⟩
⟨ vraag	vc	of ⟩	⟨ of	body	kom ⟩
⟨ kom	su	Anne ⟩			

Om te evalueren hoe goed de analyse van Alpino overeenkomt met de analyse zoals deze moet zijn volgens de menselijke annotatoren, vertalen we beide *dependency trees* naar relaties, berekenen vervolgens de overlap tussen beide verzamelingen, en delen dit door het aantal relaties in de juiste annotatie. Dit levert een percentage op dat de accuratesse van het systeem aangeeft.

Op korte zinnen haalt het systeem een accuratesse van boven de 90%. Voor krantentekst ligt de accuratesse boven de 85%. Voor gesproken taal (ontleend aan het Corpus Gesproken Nederlands) ligt de accuratesse tussen de 65% en 75% (Vlaamse fragmenten zijn voor Alpino lastiger dan Nederlandse, en 'spontaan' gesproken fragmenten (met veel afgebroken zinnen, gestotter, etc.) zijn moeilijker dan voorgelezen tekst).

1.3.2 Spellingcorrectie en grammaticale fouten corrigeren

Wie een tekst invoert met behulp van een toetsenbord kan typefouten maken. Wie een tekst schrijft kan grammaticale fouten maken. Wie een tekst bedenkt kan stilistische fouten maken. De computer kan een belangrijk hulpmiddel zijn bij het corrigeren van dergelijke fouten. Een programma dat assisteert bij het automatisch corrigeren van teksten zou in principe alle drie soorten fouten die hierboven zijn genoemd moeten kunnen vinden. De meeste correctieprogramma's beperken zich evenwel tot het opsporen van typefouten en andere makkelijk te herkennen spelfouten. Het vinden van grammaticale fouten (zoals bijvoorbeeld het foutief gebruik van `d` en `t` bij werkwoorden) is een lastig probleem. Dit probleem is alleen op te lossen met behulp van veel grammaticale kennis. Programma's die grammaticale controle uitvoeren zijn daarom relatief zeldzaam.[6] Het controleren van de stijl van een tekst is voornamelijk lastig omdat het onduidelijk is welke criteria hier gehanteerd moeten worden. We bespreken deze drie vormen van correctie hieronder in iets meer detail.

[6] maar zie Vosse (1994) voor een programma dat zeer goede prestaties behaalt voor het Nederlands.

Spellingcorrectie

Veel tekstverwerkers zijn voorzien van de mogelijkheid om een tekst te controleren op spelfouten. De meeste tekstverwerkers maken gebruik van een programma dat ieder woord in de tekst vergelijkt met een woordenlijst. Woorden in de tekst die niet in de woordenlijst voorkomen zijn potentiële spelfouten.

Deze methode kan zeer efficiënt zijn voor het opsporen van een aantal veel voorkomende fouten. Typefouten zijn bijvoorbeeld gemakkelijk te herkennen. Veel voorkomende typefouten zijn het invoegen van een extra letter (*exctra*, het te vaak herhalen van een letter (*voorstellling*), of het omwisselen van letters (*dergleijke*). Deze fouten zijn gemakkelijk te herkennen omdat het resultaat niet in een lijst met Nederlandse woorden zal voorkomen. Ook spelfouten (in met name langere woorden) zijn eenvoudig te herkennen. Beruchte gevallen zijn bijvoorbeeld *onmiddelijk* (moet zijn *onmiddellijk*) en *enigzins* (moet zijn *enigszins*). Ook spelfouten in exotische woorden (zoals *pyjama*) zijn gemakkelijk te herkennen.

Eén van de nadelen van deze methode is dat ze werkt op basis van een woordenlijst. In sectie 1.1 (pagina 6) is al opgemerkt dat het een illusie is te denken dat zo'n lijst ooit volledig kan zijn. Aan dit bezwaar wordt door de meeste programma's tegemoet gekomen doordat ze de mogelijkheid kennen woorden toe te voegen. Een gebruiker kan op die manier een woordenlijst naast de standaardlijst opbouwen die de woorden bevat die specifiek zijn voor zijn of haar teksten. Beter zou het natuurlijk zijn wanneer het programma zou kunnen herkennen wanneer een woord of reeks woorden als naam gebruikt wordt (en opzoeken in een woordenlijst daarom weinig zin heeft) en wanneer een woord een samenstelling van bestaande woorden is. Een andere uitdaging voor spellingcorrectie op basis van woordenlijsten is het bestaan van vaste uitdrukkingen, met een eigen spelling, die uit meer dan één woord bestaan. Voorbeelden zijn *ten allen tijde, te zijner tijd, ter elfder ure, etc.*. Het woord *elfder* of *ure* zal waarschijnlijk niet in een woordenlijst staan, maar moet worden goedgekeurd wanneer het deel uitmaakt van de reeks *ter elfder ure*. Een spellingcorrector kan dus niet puur volgens een woord-voor-woord-methode werken.

Corrigeren van grammaticale fouten

Een tweede, veel lastiger, tekortkoming van programma's die spellingcorrectie op basis van alleen een woordenlijst proberen te doen is het feit dat veel fouten ontstaan door het onjuist toepassen van grammaticale regels. Grammaticale fouten zijn moeilijker te herkennen dan spelfouten, omdat in een zin met een grammaticale fout de individuele woorden correct gespeld kunnen zijn, terwijl de zin als geheel desalniettemin een fout bevat:

(9)
a. een aantal woorden ontbreken
b. Jan word ziek
c. Jan is verwent

d. Hun waren agressiever dan wij
e. Die jongen is groter dan mij
f. Het gemiddeld niveau is uitstekend

In (9-a) moet *ontbreken* bijvoorbeeld *ontbreekt* zijn, maar omdat *ontbreken* een correct Nederlands woord is, zal deze fout door een eenvoudige spellingcorrector niet ontdekt worden. Om een dergelijke fout op te sporen, moet een programma weten wat het onderwerp en de persoonsvorm van deze zin is, en wat de persoon en getal van onderwerp en persoonsvorm is. Wanneer persoon en getal van onderwerp en persoonsvorm niet overeenkomen is er sprake van een grammaticale fout. Dezelfde informatie is nodig om de fout in (9-b) te ontdekken. Voor het verbeteren van voorbeeld (9-c) is nodig dat het programma ziet dat *verwent* hier niet een persoonsvorm kan zijn (maar dat het voltooid deelwoord *verwend* nodig is). Voor (9-d) moet het programma zien dat *hun* hier foutief als onderwerp is gebruikt. In (9-e) is een vergelijking gemaakt (*dan mij*) met het onderwerp van de zin, en moet dus *ik* gebruikt worden. In (9-f) is *gemiddeld* een bijvoeglijk naamwoord dat wordt voorafgegaan door *het*. In zulke gevallen is (op een aantal uitzonderingen na) een *-e*-uitgang verplicht.

Het zal duidelijk zijn dat het opsporen van grammaticale fouten over het algemeen een syntactische analyse van de zin vereist, en dat er nogal wat taalkundige kennis nodig is om zelfs de meest voorkomende grammaticale fouten te kunnen herkennen.

Ook wanneer een programma voorzien is van uitgebreide grammaticale kennis kan het lastig blijven bepaalde fouten op te sporen, omdat niet altijd duidelijk is wat de juiste analyse van een zin moet zijn:

(10) a. Welke agent meen je gezien te hebben
b. Welke agent meent je gezien te hebben

(11) a. Zij rende naar huis
b. Zij renden naar huis

In de zin (10-a) moet het werkwoord misschien eindigen op *t*, en in zin (10-b) misschien niet, maar dit hangt af van de vraag wat het onderwerp is, en dat is in deze zinnen niet met zekerheid te zeggen. In (11-a) en (11-b) is niet duidelijk of *zij* meervoud of enkelvoud is, en dus is niet duidelijk of de persoonsvorm moet eindigen op *-e* of *-en*. Dergelijke *ambiguïteit* is een struikelblok voor automatische correctie van teksten.

Stilistische controle

Programma's die assisteren bij het corrigeren van teksten bevatten soms ook een module die uitspraken doet over de stijl van een tekst. Te denken valt bijvoorbeeld aan het opsporen van extreem lange zinnen, passieve zinnen, informeel of juist archaïsch taalgebruik, het gebruik van stoplappen, etc. Het probleem bij stilistische controle van "gewone" teksten is dat het moeilijk is aan te geven welke stilistische eigenaardigheden wel en niet zijn toegestaan. Wat in de ene tekst ontoelaatbaar is, kan in een andere tekst

juist functioneel zijn. In specialistische contexten kan het gebruik van een module voor stilistische controle wel nuttig zijn. Te denken valt bijvoorbeeld aan technische documenten. Hier worden soms strenge eisen gesteld die moeten leiden tot een consistent gebruik van terminologie en een verhoogde leesbaarheid.

1.3.3 Automatisch Vertalen

Eén van de problemen die men met behulp van computers probeerde op te lossen is het vertalen van teksten. Al in de jaren vijftig probeerde men dit proces met behulp van de computer te automatiseren. Niet gehinderd door al te veel taalkundige kennis en gewapend met een groot vertrouwen in de mogelijkheden van de net uitgevonden computer probeerde men met name in de Verenigde Staten programma's te ontwerpen die wetenschappelijke teksten van het Russisch naar het Engels zouden kunnen vertalen. Deze pogingen liepen op niets uit. Toch leeft bij veel mensen (met name buitenstaanders) nog steeds het idee dat de realisatie van een systeem voor automatisch vertalen het ultieme doel van de computationele taalkunde is. Veel computertaalkundigen (zoals bijvoorbeeld Brandt Corstius, 2003) reageren daarentegen sceptisch op het idee dat er ooit een systeem zal zijn dat een willekeurige tekst perfect kan vertalen, zonder assistentie van een menselijke vertaler.

De Europese Gemeenschap heeft in de jaren tachtig een omvangrijk project gefinancierd, VerbMobil,[7] dat als doel had een automatisch vertaalsysteem te ontwikkelen dat teksten van de EG van en naar alle negen talen van de lidstaten zou kunnen vertalen. In Duitsland is in 1993 een omvangrijk project van start gegaan dat erop gericht is een vertaalsysteem te bouwen dat in staat is gesproken taal automatisch te vertalen tussen het Duits en het Engels.

Waarom is automatisch vertalen zo moeilijk

Het automatisch vertalen van teksten vereist systemen met gedetailleerde taalkundige kennis. Een tweetalig woordenboek is bij lange na niet genoeg. Wanneer een Nederlandse zin, zoals deze, bijvoorbeeld woord voor woord in het Engels wordt vertaald levert dit in verreweg de meeste gevallen geen grammaticale Engelse zin op. De woordvolgorde van het Nederlands is anders dan die van het Engels. Bovendien zijn er woorden (zoals bijvoorbeeld voorzetsels), die niet vertaald kunnen worden zonder ook te kijken naar de manier waarop ze zijn gebruikt in een zin.

Om dergelijke problemen op te lossen maken de meeste systemen voor automatisch vertalen gebruik van de methode die wordt geschetst in figuur 1.6. In de tweede stap vindt een proces plaats dat nog het meest lijkt op het eigenlijke vertalen van een zin. De woorden uit de originele zin worden hier bijvoorbeeld vervangen door woorden uit de taal waar we naar toe vertalen. De nadelen van woord voor woord vertalen worden hier ondervangen doordat boomstructuren iets vertellen over de manier waarop woorden in

[7] verbmobil.dfki.de

1. De originele zin wordt met behulp van een grammatica automatisch ontleed.
2. Het resultaat van de ontleding (bijvoorbeeld een boomstructuur) wordt *vertaald* naar een structuur die past bij de grammatica van de taal waar naar toe wordt vertaald.
3. Op basis van de aangepaste, vertaalde, structuur, wordt met behulp van een grammatica voor de taal waar naar toe wordt vertaald, een grammaticale vertaling geproduceerd.

Figuur 1.6: **Stappen bij het automatisch vertalen**

een zin gebruikt worden, en de vertaling van een woord dus kan variëren afhankelijk van de rol van dit woord in een zin. Ook kan in de "vertaalde" structuur de volgorde van woorden en zinsdelen verschillen van die in de originele structuur.

Merk op dat het vertalen van teksten met behulp van deze methode twee grammatica's vereist (één voor het origineel en één voor de vertaling) en bovendien nog een "vertaalmodule" die een structuur geproduceerd door de éne grammatica moet kunnen verbinden met een structuur in de andere grammatica. Al met al vereist het automatisch vertalen van teksten dus zeer veel taalkundige kennis.

Het verzamelen van voldoende taalkundige kennis in een systeem kan lastig zijn, maar de echte problemen voor automatisch vertalen liggen op een gebied dat slechts ten dele met *taalkunde* te maken heeft. Sommige vertaalproblemen (zie figuur 1.7) lijken te vereisen dat men een tekst niet alleen taalkundig analyseert, maar ook *begrijpt*. Het begrijpen van teksten vereist in het algemeen echter veel kennis die het domein van de taalkunde te buiten gaat.

Alternatieven

Omdat enerzijds het volledig automatisch vertalen van willekeurige teksten vooralsnog te moeilijk is gebleken, en anderzijds de behoefte aan vertalingen enorm is, zijn er ontwikkelingen op gang gekomen waarbij men de problemen die hierboven zijn besproken probeert te omzeilen.

In de eerste plaats kan men afstappen van het idee dat de vertaling volledig automatisch moet zijn. Een systeem dat een vertaler assisteert bij het produceren van een vertaling kan het werk van een vertaler vereenvoudigen en de kwaliteit van vertalingen verbeteren. Te denken valt bijvoorbeeld aan *on-line* woordenboeken, die mogelijke vertalingen van (onbekende) woorden geven. Andere nuttige hulpmiddelen zijn bijvoorbeeld systemen die aangeven hoe een woord in het verleden (bijvoorbeeld in de voorafgaande tekst of in vergelijkbare teksten) is vertaald.

Een andere benadering die succesvol is gebleken is het ontwikkelen van systemen

Conceptuele verschillen. Niet voor ieder woord bestaat er een goede vertaling in andere talen. Een bekend voorbeeld is het Nederlandse woord *gezellig*. Voor dit typisch Nederlandse begrip is in veel andere talen geen precieze equivalent te vinden. Woorden als *glaznost*, *klùne* en *debriefing* maken op een gegeven moment deel uit van het Nederlands, juist omdat er geen goede Nederlandse vertaling voor valt te geven.

Ambiguïteit. Woorden en zinnen kunnen vaak verschillende betekenissen hebben. Voor het produceren van de juiste vertaling is het nodig te weten wat er bedoeld wordt. De zin *het stomme kind lachte* zou in het Engels kunnen luiden: *the stupid child laughed*, maar ook *the mute child laughed*. De NP *de tomaat die Jan eet* kan worden vertaald als *the tomato John is eating*, maar in bepaalde bizarre contexten zou ook de vertaling *the tomato eating John* juist kunnen zijn.

Verschil in grammaticale distincties. Veel talen maken gebruik van morfologische markeringen om bijvoorbeeld naamvallen, meervouden, aanspreekvormen, werkwoordstijden, etc. uit te drukken. Wanneer men vertaalt van een taal waarin zulke verschillen niet worden gemaakt, naar een taal waarin dit wel belangrijk is, kunnen problemen ontstaan. Het Spaanse *llegó* betekent bijvoorbeeld *hij arriveert* of *zij arriveert*. Op basis van het Spaanse origineel zal niet altijd duidelijk zijn welke van beide vertalingen de juiste is. Het Engels kent alleen de aanspreekvorm *you*. In het Nederlands moet gekozen worden voor *u* of *jij*.

Idiomatische uitdrukkingen. Soms is het onjuist om te proberen de woorden in een zin één voor één van een vertaling te voorzien. Een uitdrukking als *Gedane zaken nemen geen keer* kan niet worden vertaald door een Engelse zin te produceren met daarin de vertaling van de individuele woorden. In plaats daarvan moet deze zin als een *idiomatische* uitdrukking worden herkend, die door een vergelijkbare uitdurkking in het Engels vertaald kan worden (*No use crying over spilt milk*).

Figuur 1.7: **Problemen voor automatisch vertalen**

die bedoeld zijn voor het vertalen van een zeer bepaald type tekst. Het bekendste voorbeeld op dit gebied is een Canadees systeem dat weerberichten automatisch vertaalt tussen het Engels en het Frans. Zulke beperkte systemen zijn veel eenvoudiger te bouwen dan algemene vertaalmachines, omdat het probleem veel overzichtelijker is. Het vocabulaire, en ook het soort informatie dat in een weerbericht kan worden verstrekt, is beperkt. Dit betekent dat voor woorden en zinnen die in principe ambigu zijn vrij eenvoudig de juiste lezing valt te kiezen. Ook de hoeveelheid grammaticale variatie in weerberichten is te overzien.

1.3.4 Google Sets

De prestaties van systemen voor natuurijke-taalverwerking zijn de afgelopen 15 jaar flink toegenomen. Dit is voor een groot deel te danken aan het feit dat bij de ontwikkeling en het testen van de meeste systemen gebruik wordt gemaakt van grote hoeveelheden echte taaldata (en niet enkel van geconstrueerde voorbeeldzinnen). Computers bezitten vooralsnog bij lange na niet de capaciteit die mensen bezitten om natuurlijke taal te verwerken en te interpreteren, maar ze kunnen wel goed omgaan met grote hoeveelheden data. Omdat er meer en meer tekst elektronisch beschikbaar komt, zijn veel onderzoekers gaan zoeken naar oplossingen voor problemen op het gebied van de natuurijke-taalverwerking waarbij intensief gebruik wordt gemaakt van data en statistische technieken. Een uitgebreid overzicht van dit belangrijke vakgebied is te vinden in Manning and Schütze (1999).

Een aardig voorbeeld van wat statistische technieken in combinatie met een grote hoeveelheid data op kunnen leveren is *Google Sets*.[8] Op deze web-site kan de gebruiker een aantal begrippen invoeren, waarna het programma andere begrippen probeert te vinden die tot dezelfde categorie behoren. Wanneer de gebruiker bijvoorbeeld *oak* en *willow* invoert vindt het programma *maple, pine, poplar, elm, etc.*.

De techniek achter Google Sets is, voor zover wij weten, niet bekend.[9] Omdat Google Sets vooral goed voor het Engels lijkt te werken, probeerden we een Nederlandse versie te maken. Deze werkt als volgt. Het basisidee is dat een woord w_1 waarschijnlijk tot dezelfde categorie behoort als woord w_2, wanneer ze vaak samen in nevenschikkingen (coördinaties) voorkomen. De kans dat *pvda* en *vvd* tot dezelfde categorie behoren is bijvoorbeeld groot, omdat ze vaak samen in een nevenschikking voorkomen:

(12) a. . . . en Bolkestein en Kok hun afkeer voor respectievelijk de PvdA en de VVD hadden overwonnen . . .
b. Vandaag en maandag spreken zij de PvdA en de VVD.
c. PvdA en VVD zijn behoorlijk tevreden.
d. PvdA , VVD en D66 denken de nieuwe belastingkorting van enkele honderden guldens per jaar te betalen uit . . .

[8] `http://labs.google.com/sets`

[9] Maar zie de discussie-pagina op de site voor enkele suggesties, en zie `http://www.cs.ualberta.ca/~lindek/demos.htm` voor een mogelijke techniek.

145	pvda d66	24	pvda groenlinks
129	pvda vvd	23	pvda cda
80	pvda vvd d66	19	cda pvda
34	vvd pvda	16	pvda vvd cda
26	d66 pvda	16	pvda cda d66

Figuur 1.8: Meest frequente nevenschikkingen met *PvdA*. De eerste kolom geeft de frequentie van de combinatie, de rest van de velden de elementen van de nevenschikking. Merk op dat *PvdA en VVD* en *VVD en PvdA* apart geteld worden, en dat ook *PvdA, VVD, en D66* een aparte score oplevert.

We namen drie jaargangen Nederlandse krantentekst en de tekst van een encyclopedie (samen ongeveer 50 miljoen woorden), die reeds door Alpino syntactisch geanalyseerd was. Syntactische analyse maakt het mogelijk nevenschikkingen terug te vinden, en bovendien het zelfstandig naamwoord te vinden binnen de delen van een nevenschikking. We extraheerden ongeveer 350.000 nevenschikkingen. Je zou kunnen denken dat syntactische analyse hier niet erg belangrijk is, omdat je ook domweg kan zoeken naar patronen van drie woorden van de vorm w_1 *en* w_2. Die strategie zal echter veel fouten opleveren. Op basis van een voorbeeld als

(13) het aanreiken van bier en rode wijn

wil je niet concluderen dat *bier* en *rode* verwante woorden zijn. Anderzijds zal zo'n techniek ook veel goede voorbeelden missen:

(14) er is eten , ' soft ' (limonade) , bier , wijn en whiskey.

Op basis van het bovenstaande voorbeeld zou je niet alleen willen besluiten dat *wijn* en *whiskey* verwant zijn, maar bijvoorbeeld ook dat *bier* en *wijn*, en *bier* en *whiskey* verwant zijn. Op basis van een volledige syntactische analyse worden veel van deze fouten vermeden (alhoewel syntactische analyse natuurlijk ook nooit geheel foutloos is).

De meest frequente coordinaties met *pvda* op basis van onze analyse van het corpus staan in figuur 1.8. Hieruit kunnen we gemakkelijk concluderen dat *vvd, cda, d66* en *groenlinks* tot dezelfde categorie behoren als *pvda*. In totaal komt `vvd` bijvoorbeeld zo'n 250 maal in coordinaties met *pvda* voor.

Alleen maar kijken naar frequenties is echter niet helemaal zonder gevaren. De woorden *verkeer* en *christenunie* komen beide zo'n 25 keer in coordinaties met *pvda* voor, maar alleen *christenunie* is een politieke partij.

Een iets slimmere techniek vergelijkt daarom voor ieder paar van woorden, met welke andere woorden ze vaak een nevenschikking vormen. Wanneer dit voldoende overeenkomst vertoont (er zijn verschillende wiskundige maten verzonnen om de overeenkomst uit te drukken), zijn de twee woorden vergelijkbaar. Het idee is nu dat *pvda* en *christenunie* waarschijnlijk met vergelijkbare andere woorden gecoordineerd worden

115	verkeer waterstaat	8	cda christenunie sgp
19	verkeer vervoer	4	pvda christenunie
11	verkeer pvda	4	christenunie sgp
6	pvda verkeer	4	cda sgp christenunie
4	verkeer industrie	3	groenlinks christenunie

Figuur 1.9: Meest frequente nevenschikkingen met *verkeer* en *ChristenUnie*

.

cda	0.575	gpv	0.112
d66	0.542	zaken	0.104
vvd	0.537	justitie	0.089
groenlinks	0.419	volksgezondheid	0.078
sp	0.307	minister	0.070
sgp	0.205	partijen	0.064
christenunie	0.128	financiën	0.064
rpf	0.114		

Figuur 1.10: Woorden die het meest verwant zijn aan *PvdA*, op basis van nevenschikking.

(*vvd, cda, d66*, etc.), terwijl dit voor *pvda* en *verkeer* veel minder het geval is (omdat *verkeer* ook met heel andere woorden dan *pvda* coordineert). Figuur 1.9 zien dat dit, voor dit voorbeeld, de juiste voorspellingen lijkt te doen. *Christenunie* is veel minder frequent dan *pvda*, maar komt wel met bijna precies dezelfde woorden voor, terwijl dit bij *verkeer* veel minder duidelijk is.[10]

Figuur 1.10 laat zien welke woorden volgens het programma het meest verwant zijn aan *PvdA*. De eerste negen zijn andere politieke partijen, daarna verschijnen andere politieke begrippen.

Je vraagt je wellicht af waarom een programma als dit van nut is voor natuurlijke-taalverwerking. Eén van de grote problemen van de natuurlijke-taalverwerking is dat goede woordenboeken die iets zeggen over de betekenis van woorden schaars zijn. Natuurlijk zijn er genoeg goede woordenboeken, maar die zijn gemaakt voor mensen, en geven de betekenis van een woord bijvoorbeeld in de vorm van een definitie, in natuurlijke taal. Een computer-programma kan hier weinig mee. Er zijn wel een aantal woordenboeken met betekenis speciaal voor natuurlijke-taalverwerking gemaakt, maar deze zijn vaak alleen beschikbaar voor het Engels, en altijd onvolledig. Er is dus een grote behoefte aan programma's die automatisch leren wat woorden betekenen en hoe ze met elkaar in verband staan. Om de vraag:

[10] De reden dat *verkeer* zo vaak met *pvda* opduikt is overigens voornamelijk een gevolg van zinnen als: *Minister Netelenbos (verkeer , PvdA) gaat Schiphol behandelen als een gewoon bedrijf.*

(15) Welke partij verloor in mei 2002 de verkiezingen?

te beantwoorden, op basis van de volgende tekst:

(16) Onder leiding van Ad Melkert verloor de PvdA tijdens de verkiezingen van 15 mei 2002 22 zetels

is het handig om te weten dat *PvdA* een politieke partij is, en *Ad Melkert* niet. Zulke kennis kan, to op zekere hoogte, automatisch uit tekstbestanden gehaald worden met behulp van programma's zoals *Google Sets*.

1.4 Oefeningen

1. (a) Geef een reden waarom de natuurlijke-taalverwerking een onderdeel van de kunstmatige intelligentie is.
   (b) Beschrijf waarom de natuurlijke-taalverwerking en de computationele taalkunde een bijdrage kunnen leveren aan de taalkunde.
2. Waarom is het gebruik van een woordenlijst niet voldoende voor het opsporen van grammaticale fouten in tekst?
3. Laat een stuk Nederlandse tekst vertalen door Systran (`www.systransoft.com/`), en bespreek aan de hand van het resultaat welke problemen op het gebied van automatisch vertalen nog niet opgelost zijn.
4. (a) Beschrijf wat het *all-and-only*-principe voor grammatica's inhoudt.
   (b) Beschrijf een toepassing of situatie waarin het niet erg is wanneer niet aan het *all*-deel van het principe voldaan wordt.
   (c) Beschrijf een toepassing of situatie waarin het niet erg is wanneer niet aan het *only*-deel van het principe voldaan wordt.
5. In een recent Engels boek over natuurlijke taalverwerking wordt Groucho Marx geciteerd:

   One morning I shot an elephant in my pajamas. How he got into my pajamas I don't know.

   (a) Geef een context-vrije grammatica voor een fragment van het Engels dat de eerste zin op tenminste twee manieren produceert.
   (b) Leg de grap uit in termen van de grammatica.

Hoofdstuk 2

Definite Clause Grammatica

Prolog is uitermate geschikt voor het implemeteren van grammatica's. In dit hoofdstuk introduceren we *Definite Clause Grammatica* (DCG), Prolog's ingebouwde grammaticaformalisme. DCG's maken het mogelijk context-vrije grammatica's rechtstreeks in Prologprogramma's om te zetten. Doordat DCG het gebruik van *unificatie* ondersteund, kunnen grammatica's bovendien compact geïmplementeerd worden, en kunnen grammatica's geïmplementeerd worden die krachtiger zijn dan context-vrije grammatica.

2.1 Zinsontleden in Prolog

Herschrijfregels laten zich bijna direct als Prolog-regels lezen. Een regel als

(1) S → NP VP

betekent dat een S kan worden herschreven als een NP gevolgd door een VP. We kunnen ook zeggen dat we kunnen bewijzen dat iets een S is door te bewijzen dat het een NP gevolgd door een VP is. In Prolog zouden we schrijven:

(2)
```
s :- np, vp.
```

Er is nog wel een verschil tussen de herschrijfregel en bovenstaande logische regel. De herschrijfregel gaat over zinnen en zinsdelen, oftewel: rijtjes woorden waarvan moet worden vastgesteld of het al dan niet zinnen of zinsdelen van de taal zijn. Een bepaalde reeks woorden is alleen een S, op basis van de NP-VP-regel, als er een NP is die het correspondeert met het eerste deel van de invoer, en een VP die correspondeert met het tweede deel. De volgende Prolog-regel drukt ook deze extra condities uit:

(3)
```
s(P0,P2) :- np(P0,P1), vp(P1,P2).
```

De variabelen `P0, P1` en `P2` staan hier voor posities in de invoer. Voor het moment nemen we aan dat een invoer als

```
s(P0,P2) :- np(P0,P1), vp(P1,P2).
np(P0,P2) :- det(P0,P1), n(P1,P2).
np(P0,P3) :- det(P0,P1), a(P1,P2), n(P2,P3).
vp(P0,P1) :- v(P0,P1).
vp(P0,P2) :- v(P0,P1), np(P1,P2).

a(P0,P1) :- word(P0,oude,P1).

det(P0,P1) :- word(P0,de,P1).
det(P0,P1) :- word(P0,het,P1).
det(P0,P0).

n(P0,P1) :- word(P0,man,P1).
n(P0,P1) :- word(P0,huis,P1).

v(P0,P1) :- word(P0,verlaat,P1).
v(P0,P1) :- word(P0,schreeuwt,P1).
```

Figuur 2.1: Een context-vrije grammatica in Prolog

(4) de man verlaat het huis

aanleiding geeft tot de volgende Prolog-feiten:

(5)
```
word(0,de,1).
word(1,man,2).
word(2,verlaat,3).
word(3,het,4).
word(4,huis,5).
```

Herschrijfregels die een categorie herschrijven als een woord (zoals N → *man*) kunnen we nu als volgt implementeren:

(6) `n(P0,P1) :- word(P0,man,P1).`

Bewijzen dat de invoer in (4) een zin is, betekent nu dat het doel `s(0,5)` moet slagen. Eén manier om dat doel te bereiken is te bewijzen dat `np(0,P1)` en `vp(P1,5)` een oplossing heeft.

Figuur 2.1 bevat een grammaticafragment dat is gecodeerd volgens bovenstaande methode.

Een context-vrije herschrijfregel kan nul of meer dochters hebben. De grammatica in 2.1 bevat voornamelijk regels met twee dochters, of regels met één dochter, waarbij de dochter een woord is. Andere mogelijkheden worden echter niet door de notatie uitgelsoten. De volgende regel heeft bijvoorbeeld drie dochters:

(7) NP → Det A N

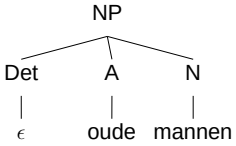
In dit geval moeten 4 variabelen gebruikt worden om de relatieve posities van de dochters ten opzichte van elkaar te coderen. Regel 3 van de grammatica in figuur 2.1 laat zien hoe deze regel gecodeerd kan worden.

Een regel die geen dochters introduceert is bijvoorbeeld:

(8) Det → ϵ.

De ϵ geeft hier aan dat de rechterkant van de regel leeg is. Zo'n regel zouden we kunnen gebruiken om een NP als oude mannen te ontleden als een reeks die bestaat uit een 'lege' determiner, gevolgd door een adjectief en een zelfstandig naamwoord:

(9)



In de notatie met posities correspondeert een dergelijke regel met het feit `det(P0,P0)`. Dit betekent dat we voor iedere positie in de invoer kunnen aannemen dat daar een 'lege' determiner staat. Het hangt natuurlijk van de rest van de grammatica af of het ook zinvol is om op die plek een determiner aan te nemen. Stel bijvoorbeeld dat we `np(0,P2)` willen bewijzen voor een invoer die begint met *oude mannen* .... We kunnen hiervoor de regel

(10) `np(P0,P3) :- det(P0,P1), a(P1,P2), n(P2,P3).`

gebruiken. `np(0,P2)` kan nu worden bewezen (met `P2=2`), omdat `det(0,0)` slaagt (op basis van het prolog-feit `det(P0,P0)`), en `a(0,1)` en `n(1,2)` ook slagen (mits *oude* en *mannen* als `a` en `n` gedefinieerd zijn).

Het gebruik van het predicaat `word` met drie argumenten is omslachtig. Je zou voor elke zin die je wilt ontleden de woorden van de zin opnieuw in je programma moeten coderen. In de volgende paragraaf zullen we een techniek bespreken die deze stap overbodig maakt. Toch kan het expliciet representeren van de input als feiten in het programma nuttig zijn. Er zijn bijvoorbeeld situaties te bedenken waar de relatie tussen de invoer en de feiten die aan het programma moeten worden toegevoegd niet triviaal is. Zo zou je bijvoorbeeld een versie van `input/2` kunnen schrijven die combinaties van woorden als één feit weergeeft (ik probeer 's nachts te slapen zou onder andere `word(2,[s,nachts],4` en `word(4,[te,slapen],6)` op kunnen leveren), of die een enkel woord in de invoer juist als twee feiten representeert (ik ben opgebeld leidt tot `word(2,op,3)` en `word(3,gebeld,4)`. Ook andere vormen van morfologische analyse (waarbij een verbogen woord bijvoorbeeld wordt geanalyseerd als een stam plus een uitgang) zouden in dit stadium kunnen plaatsvinden.

2.1.1 Invoerposities als Prolog-lijsten

Een parser voor een context-vrije grammatica zal meestal als doel hebben van een reeks woorden vast te stellen of het al dan niet een grammaticale zin (of grammaticaal zinsdeel) is. Wanneer we de invoer van zo'n programma als een Prolog-lijst coderen, kan de stap die de invoer omzet in Prolog-feiten geheel worden overgeslagen.

De eliminatie van een de stap die de invoer omzet in Prolog-feiten berust op het volgende inzicht. Posities in de invoer coderen we normaalgesproken met getallen: 0, 1, 2,.... In plaats van getallen kunnen we echter ook Prolog-lijsten gebruiken. We spreken bijvoorbeeld af dat, gegeven de invoer `[de, oude, man, schreeuwt]`, de lijst `[de, oude, man, schreeuwt]` met positie 0 correspondeert, `[oude, man, schreeuwt]` met positie 1, `[schreeuwt]` met positie 3, en `[ ]` met de eindpositie (4 in dit geval). Wanneer we de ontleder (met name `input` en `make_words`) zo aanpassen dat ze met lijsten in plaats van getallen werkt, zal het effect van `input` zijn dat, gegeven de invoer `[de,oude,man,schreeuwt]`, de volgende feiten worden toegevoegd:

(11)
```
word([de,oude,man,schreeuwt],de,[oude,man,schreeuwt]).
word([oude,man,schreeuwt],oude,[man,schreeuwt]).
word([man,schreeuwt],man,[schreeuwt]).
word([schreeuwt],schreeuwt,[]).
```

Met deze feiten is iets vreemds aan de hand. Ze hebben allemaal dezelfde structuur:

(12)
```
word([Word|Words],Word,Words).
```

Merk bovendien op dat `input` niet meer nodig is om de eindpositie van de invoer te berekenen. In de lijst-notatie is de eindpositie namelijk altijd de lege lijst (`[ ]`). Maar als dat zo is, dan kunnen we ook volstaan met het toevoegen van dit enkele feit aan de ontleder, en de invoer-stap geheel en al overslaan.


```
parse(Words) :-
  s(Words,[]).

word([Word|Words],Word,Words).
```

Figuur 2.2: Ontleden met behulp van posities als Prolog-lijsten

Wanneer we in het vervolg invoerposities als lijsten beschouwen, hoeven we niets te veranderen aan de manier waarop grammatica-regels zijn gecodeerd. Het fragment in figuur 2.1 blijft dus ongewijzigd.

2.2 Notatie: De DCG-pijl

De vertaling van een context-vrije grammatica naar een Prolog-programma vereist dat alle regels worden omgezet in corresponderende Prolog-regels, waarbij aan iedere categorie twee variabelen worden toegevoegd die dienen om de posities van de invoer te coderen. Daarnaast moet iedere regel die een woord introduceert worden omgezet in een Prolog-regel die het predicaat `word/3` aanroept. Je zou kunnen volhouden dat dit nog steeds een vrij omslachtige manier is om een grammatica te implementeren. De Prolog-notatie voor *definite clause grammars* (DCG) komt aan dit bezwaar tegemoet, doordat ze het expliciet definiëren van posities en het expliciet aanroepen van `word` (of een variant daarvan) overbodig maakt.

Kern van de DCG-notatie is de DCG-pijl `'-->'`. Deze pijl is te vergelijken met het Prolog *if*-teken `':-'`. Een herschrijfregel als

(13) $S \rightarrow NP\ VP$

schrijven we in DCG-notatie als:

(14)
```
s --> np, vp.
```

Prolog zet deze regel om in:

(15)
```
s(P0,P2) :- np(P0,P1), vp(P1,P2).
```

Deze omzetting gebeurt automatisch wanneer je een programma in Prolog consulteert. Wanneer je dus van de DCG-pijl gebruik maakt in je programma's, zul je, wanneer je om een *listing* vraagt, of wanneer je een *trace* doet, altijd de regel met de extra variabelen zien.

```
s --> np, vp.          det --> [de].
                       det --> [het].
np --> det, n.         det --> [].
np --> det, a, n.
                       n --> [man].
vp --> v.              n --> [huis].
vp --> v, np.
                       v --> [verlaat].
a --> [oude].          v --> [schreeuwt].
```

Figuur 2.3: Een Definite Clause Grammatica

Regels die een woord introduceren, schrijven we in DCG-notatie als volgt:

(16) `n --> [man].`

Deze regel wordt (in Sicstus Prolog[1]) omgezet in:

(17) `n(P0,P1) :- 'C'(P0,man,P1).`

Merk op dat het speciale (*built-in*) predicaat `'C'` hier de rol speelt van `word/3` in het voorbeeld hierboven.

In figuur 2.3 staat de grammatica uit figuur 2.1 in DCG-notatie. Om te bewijzen dat een bepaalde invoer een zin of zinsdeel is, roepen we niet langer het predicaat `parse` aan, maar roepen we rechtstreeks het predicaat aan dat correspondeert met de categorie die we aan de invoer proberen toe te kennen. Om te bewijzen dat

(18) de man verlaat het huis

een zin is, roepen we dus

(19) `s([de,man,verlaat,het,huis],[])`

aan. Het eerste argument van `s` is altijd de hele invoer (gecodeerd als prolog-lijst) en het tweede argument is de lege lijst.

Regels die geen dochter introduceren schrijven we in DCG-notatie als volgt:

(20) `det --> [].`

Dit leidt tot:

(21) `det(P0,P1) :- P0 = P1.`

Het is ook mogelijk meer dan één woord als dochter te introduceren:

(22) `det --> [bijna,alle].`

Dit leidt tot:

(23) `det(P0,P2) :- 'C'(P0,bijna,P1), 'C'(P1,alle,P2).`

Tenslotte kun je regels schrijven die zowel een syntactische categorie als een woord introduceren:

(24) `a --> [erg], a.`

Hiermee kun je bijvoorbeeld *erg oude* (of *erg erg oude*) als een `a` analyseren.

[1] De manier waarop *terminals* (regels die een `[Woord]` introduceren) worden behandeld is niet in alle Prolog-inplementaties gelijk. De voorbeelden hieronder volgen Sictus-Prolog. In SWI-Prolog worden deze voorbeelden vertaald als:

```
n([man|P1],P1).
det(P0,P0).
det([bijna,alle|P1],P1).
```

2.3 Prolog-termen als Categorieën

De DCG-notatie lijkt op het eerste gezicht slechts een handige manier om context-vrije grammatica's om te zetten in Prolog. In werkelijkheid biedt de DCG-notatie echter veel meer mogelijkheden dan een context-vrije grammatica.

In een context-vrije grammatica zijn de categorieën van de grammatica atomaire symbolen. Dit betekent dat de categorieën zelf geen interne structuur hebben. In een DCG worden de categorieën echter weergegeven door Prolog-termen. Een term kan variabelen bevatten. Door gebruik te maken van variabelen kunnen we taalkundige fenomenen, zoals overeenstemming in persoon en getal van persoonsvorm en het onderwerp van de zin, naamvallen, werkwoordstijden, het verschil tussen hoofd- en bijzinnen, etc. gemakkelijk coderen.

2.3.1 Congruentie van persoon en getal

Het klassieke voorbeeld van congruentie is overeenstemming in persoon en getal tussen het onderwerp van de zin en de persoonsvorm:

(25) a. ik denk aan Henk.
b. hij denkt aan Henk.
c. wij denken aan Henk.
d. *ik denken aan Henk.
e. *hij denk aan Henk.
f. *wij denkt aan Henk.

In een context-vrije herschrijfgrammatica kan dit alleen beschreven worden door categorieën als NP_enk_1, NP_enk_2, NP_mv, enzovoort, in te voeren, en regels als

(26) a. S → NP_enk_1 VP_enk_1
b. S → NP_enk_2 VP_enk_2
c. ...

Deze oplossing heeft echter als nadeel dat er veel extra categorieën ontstaan, en, als gevolg hiervan, dat veel regels verdubbeld moeten worden. In plaats van één regel voor NP-VP-zinnen zullen er nu verschillende zulke regels zijn.

In een DCG is een elegantere oplossing mogelijk. De volgende regel drukt uit dat er overeenkomst in persoon en getal moet zijn tussen onderwerp en (persoonsvorm van de) VP:


```
s --> np(P,G), vp(P,G).  vp(P,G) --> v(P,G).
                         vp(P,G) --> v(P,G), pp.
np(1,enk) --> [ik].
np(2,enk) --> [jij].     pp --> p, np(_,_).
np(3,enk) --> [hij].
np(3,enk) --> [zij].     v(1,enk) --> [denk].
np(1,mv) --> [wij].      v(2,enk) --> [denkt].
np(2,mv) --> [jullie].   v(3,enk) --> [denkt].
np(3,mv) --> [zij].      v(_,mv) --> [denken].
```

Figuur 2.4: DCG met regels voor overeenstemming in persoon en getal tussen onderwerp en persoonsvorm.

(27) `s --> np(P,G), vp(P,G).`

De variabelen `P` en `G` staan voor, respectievelijk, persoon en getal. Een fragment van een grammatica die overeenkomst in persoon en getal tussen onderwerp en persoonsvorm garandeert staat in figuur 2.4.

De DCG-regel in (27) wordt door Prolog als volgt vertaald:

(28) `s(P0,P2) :- np(P,G,P0,P1), vp(P,G,P1,P2).`

Merk op dat aan de termen `np(P,G)` en `vp(P,G)` twee extra variabelen worden toegevoegd, die corresponderen met de twee extra argumenten die steeds nodig zijn om informatie over de invoer weer te geven. In het algemeen geldt dat bij de vertaling van DCG naar Prolog de positie-variabelen altijd als de twee laatste argumenten van de term worden toegevoegd.

2.3.2 Lidwoorden

Het Nederlands maakt onderscheid tussen de lidwoorden *het* en *de*. Het eerste gaat alleen samen met *onzijdige* zelfstandige naamwoorden, terwijl het tweede alleen samen gaat met *vrouwelijke* of *mannelijke* zelfstandige naamwoorden. Omdat het onderscheid tussen vrouwelijke en mannelijke zelfstandige naamwoorden hier verder geen rol speelt, zullen we in het vervolg eenvoudigweg spreken over *de*-woorden en *het*-woorden.

```
np --> det, n.  a --> [gevaarlijke].
n --> a, n.     a --> [bruine].

det --> [het].  n --> [beest].
det --> [de].   n --> [beer].
```

Figuur 2.5: Eenvoudige DCG voor NP's.

```
np --> det(Lidw), n(Lidw).  a --> [gevaarlijke].
n(Lidw) --> a, n(Lidw).     a --> [bruine].

det(het) --> [het].         n(het) --> [beest].
det(de) --> [de].           n(de) --> [beer].
```

Figuur 2.6: DCG voor NP's met onderscheid tussen *de* en *het*

Met behulp van de regels in figuur 2.5 kunnen we NP's herkennen als *de beer, de bruine beer, het gevaarlijke bruine beest*, enzovoort. Merk op dat de regel die N herschrijft als A N, recursief is. Op deze manier kunnen we in principe een willekeurig lange reeks bijvoeglijke naamwoorden aan N vooraf laten gaan.

De bovenstaande grammatica herkent evenwel ook ongrammaticale reeksen als **het beer*, **het bruine beer*, en **de gevaarlijke bruine beest*. Om deze ongrammaticale reeksen uit te sluiten voegen we een extra argument aan de categorieën `det` en `n` toe, waarmee we informatie over het soort zelfstandig naamwoord (*de* of *het*) representeren. De aangepaste grammatica staat in figuur 2.6.

2.3.3 Hoofd- en bijzinsvolgorde

In sectie 1.2, in figuur 1.3, presenteerden we een context-vrije grammatica voor een deel van het Nederlands, met enkele opvallende tekortkomingen:

- Er werd geen verschil gemaakt tussen de tegenwoordige tijd, de voltooide tijd, en de infinitief vorm van een werkwoord.
- De regels voor VP's in hoofdzinnen, en voor VP's in bijzinnen waren grotendeels identiek.
- Hoofdzinnen zonder hulpwerkwoord werden niet verantwoord.

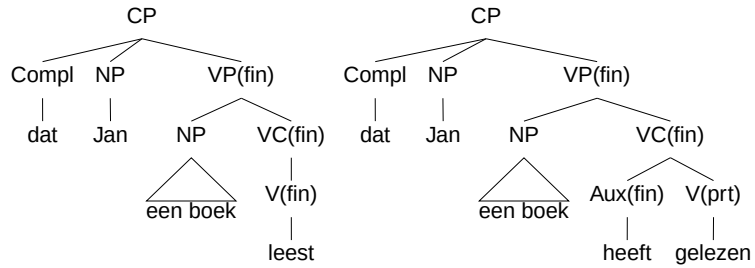
Het eerste probleem is vrij eenvoudig op te lossen met behulp van de technieken die we hierboven voor congruentie van persoon en getal en voor het onderscheid tussen *de*- en *het*-woorden hebben gebruikt. Hieronder laten we zien dat ook de twee andere tekortkomingen met deze techniek (het toevoegen van een variabele die informatie over de vorm van een woord bevat, en die wordt doorgegeven in de regels) kunnen worden opgelost.


```
vp(V) --> vc(V).
vp(V) --> np, vp(V).
vp(V) --> pp, vp(V).
vp(V) --> vp(V), pp.
vp(V) --> adv, vp(V).
vp(V) --> vp(V), cp.
vc(V) --> v(V).
vc(V) --> aux(V), v(prt).
vc(V) --> mod(V), v(inf).
vc(V) --> v(prt), aux(V).
vc(V) --> v(inf), mod(V).
cp --> compl, np, vp(fin).
s --> np, aux(fin), vp(no-aux).
s --> np, mod(fin), vp(no-mod).
s --> np, v(fin), vp(no-v).
aux(fin) --> [heeft].
aux(no-aux) --> [].
mod(fin) --> [wil].
mod(no-mod) --> [].
v(fin) --> [denkt].
v(prt) --> [gedacht].
v(inf) --> [denken].
v(no-v) --> [].
```

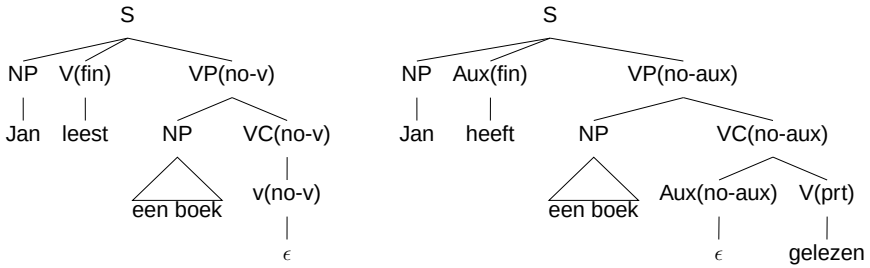
Figuur 2.7: DCG voor hoofd- en bijzinsvolgorde

Het enige verschil tussen bijzinnen en hoofdzinnen (in de mededelende vorm, en waarin het onderwerp voorop staat) is de positie van de persoonsvorm.In bijzinnen maakt de persoonsvorm deel uit van het werkwoordscluster aan het eind van de zin, en in (mededelende) hoofdzinnen staat de persoonsvorm op de tweede plaats. De context-vrije grammatica kan deze eigenschap van de Nederlandse syntaxis alleen maar verantwoorden door regels te verdubbelen.

De grammatica in figuur 2.7 bevat een betere analyse van de woordvolgorde in hoofd- en bijzinnen. Er wordt slechts één verzameling regels voor de `vp` gegeven. Wel bevatten deze regels allemaal de variabele `V`, die gebruikt wordt om de status van het werkwoord (*verb*) te coderen. In bijzinnen heeft `V` altijd de waarde `fin` (voor *finiet*). Het werkwoordscluster bestaat in dit geval namelijk altijd een finiet werkwoord: ofwel hoofdwerkwoord (`v(fin)`), een hulpwerkwoord van tijd (*auxiliary*, `aux(fin)`) in combinatie met een voltooid deelwoord (*participle*, `v(prt)`), of een hulpwerkwoord van modaliteit (`mod(fin)`) in combinatie met een infinitief (`v(inf)`). Enkele voorbeelden staan in figuur 2.8.



Figuur 2.8: Bijzinsvolgorde



Figuur 2.9: Hoofdzinsvolgorde

In hoofdzinnen is de situatie interessanter. Een hoofdzin kan een hulpwerkwoord van tijd, van modaliteit, of een hoofdwerkwoord als persoonsvorm bevatten. Deze persoonsvorm staat altijd op de tweede plaats. Bovendien ontbreekt in deze gevallen een persoonsvorm in de VP. Deze informatie coderen we door V de waarde `no-aux`, `no-mod` of `no-v` te geven. Een VC die geen persoonsvorm bevat kan worden herschreven op dezelfde manier als een VC met persoonsvorm, *behalve* dat de knoop die de normaalgesproken de persoonsvorm bevat in dit geval *leeg* is. Dit coderen we met behulp van een DCG-regel die een lege rechterkant heeft. Figuur 2.9 bevat enkele voorbeelden. We hebben drie verschillende `no-` waarden nodig, om ervoor te zorgen dat een hoofdzin die bijvoorbeeld een hoofdwerkwoord op de tweede plaats bevat, niet een VC bevat met daarin een infinitief, of dat een hoofdzin met een hulpwerkwoord van tijd ook een VC bevat met een infinitief in plaats van een voltooid deelwoord:

(29) a. *Jan leest het boek lezen
b. *Jan heeft het boek lezen

2.4 Notatie: De DCG-accolades

Soms kan het handig zijn een DCG-fragment te combineren met 'gewone' Prolog-regels. Wanneer je een grammatica schrijft met een groot woordenboek, is het handig om de grammatica en het woordenboek enigszins gescheiden te houden. Een aparte woordenboek-'module' kan helpen om de informatie in het woordenboek overzichtelijk en consistent te houden. Ook het omzetten van informatie uit een algemeen woordenboek naar een woordenboek in Prolog-formaat is vaak eenvoudiger wanneer de informatie kan worden omgezet naar algemene Prolog-feiten, en niet naar DCG-regels.

Veel woorden kennen verschillende vormen. Zelfstandige naamwoorden kennen meestal een enkelvouds- en meervoudsvorm, bijvoeglijke naamwoorden kunnen een *-e* uitgang krijgen, in de vergelijkende en overtreffende trap staan (*groot, groter, grootste*), en werkwoorden kunnen in het enkelvoud en meervoud staan, in de tegenwoordige of verleden tijd staan, en als voltooid deelwoord voorkomen. Dit betekent dat voor ieder woord dat aan de grammatica wordt toegevoegd een aantal vergelijkbare regels moet worden geïntroduceerd. Wanneer we bijvoorbeeld onderscheid maken in persoon en getal van het werkwoord, hebben we de volgende regels nodig om de tegenwoordige tijd van de werkwoorden lopen en denken te beschrijven:

(30)
```
v(1,enk) --> [loop].   v(1,enk) --> [denk].
v(2,enk) --> [loopt].  v(2,enk) --> [denkt].
v(3,enk) --> [loopt].  v(3,enk) --> [denkt].
v(_,mv) --> [lopen].   v(_,mv) --> [denken].
```

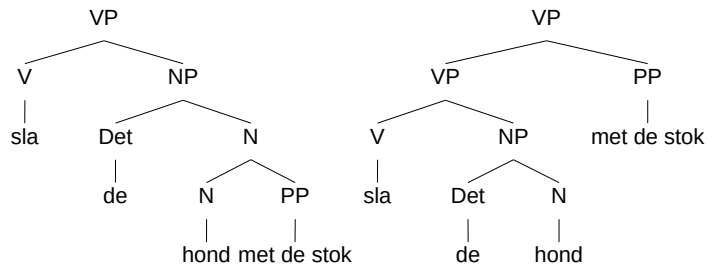
Deze notatie is nogal omslachtig. Een compactere notatie verkrijgen we door woordenboek en grammaticaregels te scheiden. Het idee is dat een `v(1,sg)` een willekeurig woord kan zijn, maar dat we vervolgens wel in het woordenboek kijken of dit woord inderdaad een eerste persoon enkelvoud van een werkwoord is. De implementatie van dit idee werkt als volgt:

(31)
```
v(1,enk) --> [Woord], {ww(Woord,_,_)}.
v(2,enk) --> [Woord], {ww(_,Woord,_)}.
v(3,enk) --> [Woord], {ww(_,Woord,_)}.
v(_,mv) --> [Woord], {ww(_,_,Woord)}.

ww(loop,loopt,lopen).
ww(denk,denkt,denken).
```

De accolades zijn hier essentieel. Het deel van de regel tussen accolades maakt geen deel uit van de DCG-regel, maar bevat een aanroep van een 'gewone' regel. Prolog vertaalt normaalgesproken een DCG-regel door aan alle termen in de regel twee extra argumenten toe te voegen, die gebruikt worden om de input te coderen. De accolades geven aan dat dit voor het deel van de code tussen accolades achterwege moet blijven. De eerste regel vertaalt dus als:

(32)
```
v(1,enk,A,B) :- 'C'(A,Woord,B), ww(Woord,_,_).
```

Het voordeel van deze techniek is dat het toevoegen van een nieuw werkwoord betekent dat er alleen maar een extra feit voor `woord/3` moet worden toegevoegd. Deze aanpak levert daarom, vooral bij grotere aantallen woorden, of wanneer één lemma in vele vormen kan voorkomen, een duidelijk compacter en overzichtelijker woordenboek op. Een ander voordeel is dat het woordenboek zelf (i.e. een verzameling feiten zoals `ww` hierboven) geen informatie bevat over de vraag hoe, bijvoorbeeld, persoon en getal precies in de grammatica gecodeerd zijn. Dit betekent dat veranderingen in de grammatica niet direct gevolgen hoeven te hebben voor het woordenboek, en dat verschillende grammatica's gebruik kunnen maken van hetzelfde woordenboek.



Figuur 2.10: sla de hond met de stok

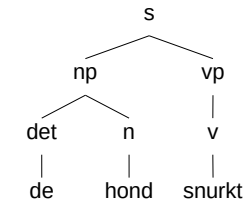
2.5 Grammatica's die boomstructuren opleveren

Een programma dat alleen zegt of een gegeven invoer al dan niet grammaticaal is, is van beperkt nut. Veel vaker willen we niet alleen een ja/nee antwoord, maar ook enige informatie over de syntactische structuur of de betekenis van de geanalyseerde invoer.

Een grammatica die boomstructuren aflevert kunnen we bijvoorbeeld gebruiken om te onderzoeken in hoeverre een grammatica ambiguïteiten bevat. Een grammatica is (syntactisch) ambigu wanneer voor één uiting verschillende analyses, corresponderend met verschillende boomstructuren, mogelijk zijn. Een bekend voorbeeld van ambiguïteit heeft te maken met PP's. Een PP kan optreden als deel van een NP (*een hond met korte haren*) of als deel van een VP (ga naar school op de fiets). In sommige gevallen is het moeilijk uit te maken of een PP deel uitmaakt van een NP of een VP (*sla de hond met de stok*). De boomstructuren voor de twee verschillende analyses van dit voorbeeld staan in figuur 2.10.

Het construeren van boomstructuren als onderdeel van het ontleedproces vereist normaalgesproken dat de *parser* wordt uitgebreid met voorzieningen die het mogelijk maken informatie over de toegepaste regels bij te houden. In een DCG is het echter ook mogelijk de boomstructuur als onderdeel van de grammatica zelf te beschouwen. Dit kan worden bereikt door het toevoegen van een extra argument aan de regels in de grammatica. Dit argument wordt vervolgens gebruikt om een Prolog-term op te bouwen die de boomstructuur representeert.

In de voorafgaande secties zijn een aantal voorbeelden behandeld waar variabelen aan de categorieën van een DCG werden toegevoegd. De mogelijke waarden van deze variabelen waren steeds atomair. De waarde van een Prolog-variabele kan echter ook een complexe Prolog-term zijn. Deze eigenschap kunnen we gebruiken om aan alle regels in de grammatica een argument toe te voegen dat de syntactische boom representeert van het deel van de invoer dat wordt geanalyseerd met de betreffende regel.




```
b(s,[b(np,[b(det,[w(de)]),b(n,[w(hond)])]),b(vp,b(v,[w(snurkt)]))])
```

Figuur 2.11: Syntactische boom en bijbehorende Prolog-term.

```
s(b(s,[NP,VP])) --> np(NP), vp(VP).

np(b(np,[Det,N])) --> det(Det), n(N).

vp(b(vp,[V])) --> v(V).

det(b(det,[w(de)])) --> [de].

n(b(n,[w(hond)])) --> [hond].

v(b(v,[w(snurkt)])) --> [snurkt].
```

Figuur 2.12: Dcg met bomen.

Een boom kunnen we in Prolog bijvoorbeeld coderen als een term `b(Moeder, Dochters)`. Het eerste argument is een syntactische categorie, het tweede argument is een lijst van dochters, waarbij iedere dochter ook zelf weer een boom kan zijn. Bladeren in de boom (de woorden) coderen we als `w(Woord)`, waar `Woord` het woord zelf is. In figuur 2.11 geven we een voorbeeld van een boom en de corresponderende term.

In figuur 2.12 wordt een grammatica-fragment gegeven dat dergelijke Prolog-termen opbouwt. Aan iedere categorie is een extra argument toegevoegd. Dit argument bevat de boomstructuur van een constituent die op basis van deze regel geanalyseerd is.[2]

Wanneer we `s(Boom,[de,hond,snurkt],[]` aanroepen, zal dit slagen met `Boom` geïnstantieerd als de Prolog-term in figuur 2.11.

[2]Merk op dat het woordenboek voor deze grammatica ook aanzienlijk vereenvoudigd kan worden door het scheiden van woordenboek en regels met behulp van accolades:

(i)
```
n(b(n,[w(Woord)])) --> [Woord], {noun(Woord)}.

noun(hond).
.....
```

```
s  --  np  --  det  --  de
           --  n    --  hond
   --  vp  --  v    --  slaapt
```

Figuur 2.13: ascii-representatie van boomstructuren.

```
datum(d(D,M)) --> dag(D), maand(M).

dag(1) --> [een].
dag(2) --> [twee].
...
dag(31) --> [eenendertig].

maand(1) --> [januari].
maand(2) --> [februari].
...
maand(12) --> [december].
```

Figuur 2.14: Dcg voor data.

Voor langere zinnen worden de Prolog-termen voor de bijbehorende boomstructuren snel moeilijk leesbaar. Wanneer je een grammatica maakt die boomstructuren oplevert, wil je waarschijnlijk ook code schrijven die het mogelijk maakt boomstructuren op een leesbare wijze te presenteren. Een veel gebruikte methode presenteert bomen overdwars, met de stam links en de bladeren rechts (figuur 2.13).

Een boom met de stam boven en de bladeren onder vereist meestal iets meer rekenwerk. Het is ook mogelijk Prolog-term voor een boom om te zetten in input voor een grafisch programma (zoals Tcl/Tk of Latex), en dit te gebruiken om bomen te vertonen.

2.6 Betekenis

In veel toepassingen, zoals dialoogsystemen en automatisch vertalen, is de betekenis van een uitdrukking van belang. Betekenissen kunnen we op dezelfde manier aan de grammatica toevoegen als boomstructuren, namelijk door een argument te introduceren dat de betekenis van een uitdrukking bevat.

Een eenvoudig voorbeeld wordt gegeven in figuur 2.14. Data kunnen we weergeven als termen van de vorm `d(Dag,Maand)`, waarbij `Dag` een getal tussen 1 en 31 is, en `Maand` een getal tussen 1 en 12. Een uitdrukking als *twee oktober* wordt door de grammatica de 'betekenis' `d(2,10)` toegekend.

Voor toepassingen waarbij aan hele zinnen een betekenis moet worden toegekend, kan het berekenen van de betekenis van zinnen erg complex worden. In Blackburn en Bos (1998) wordt een techniek besproken om ook de meest complexe betekenissen met behulp van DCG te berekenen.

2.7 Links-recursie

Links-recursie kan de oorzaak zijn van het niet-termineren van een Prolog-programma:

(33)
```
ouder(jan,marie).
voorouder(X,Y) :-
  ouder(X,Y).
voorouder(X,Y) :-
  voorouder(X,Z),
  ouder(Z,Y).
```

Wanneer we `voorouder(jan,henk)` proberen te bewijzen, zal het bovenstaande programma niet termineren, omdat het predicaat `voorouder` recursief kan worden aangeroepen, zonder dat die wezenlijke veranderingen in het te bewijzen *goal* oplevert. De *goal* `voorouder(jan,A)` leidt bijvoorbeeld tot de *goal* `voorouder(jan,B)`, enzovoort. In dit geval is de oplossing eenvoudig: de *goals* in de *body* van de tweede definitie voor `voorouder` moeten worden omgedraaid.

Omdat DCG-regels worden vertaald in Prolog-regels, en er bij het ontleden van een zin gebruik wordt gemaakt van de Prolog-bewijsstrategie, zijn links-recursieve regels in een grammatica ook een probleem. Links-recursie doet zich bijvoorbeeld voor wanneer we de volgende regel aan een grammatica toevoegen:

(34)
```
n --> n, pp.
```

Deze regel laat een N bestaan uit een N gevolgd door een PP. Een recursieve regel (vergelijkbaar met N → AP N) lijkt hier op zijn plaats, omdat er in principe geen grens is aan het aantal PP's dat een zelfstandig naamwoord kan volgen: *bal in de hoek, bal met rode stippen in de hoek, bal van de hond met rode stippen in de hoek*, enzovoort.

Wanneer we de invoer `[bal, met, rode, stippen]` echter gaan ontleden zal vroeger of later

(35) `n([bal, met, rode, stippen], X)`

worden aangeroepen, hetgeen zal leiden tot een *goal*

(36) `n([bal, met, rode, stippen], Y)`

enzovoort.

Merk op dat links-recursie niet altijd veroorzaakt hoeft te worden door een regel waarin de moeder en linkerdochter identiek zijn. Links-recursie kan ook veroorzaakt worden door een regel waarin de linkerdochter een knoop is die een linkerdochter kan hebben (enz.) die identiek is aan de moederknoop, zoals in het voorbeeld hieronder.

(37)
```
np --> det, n.
det --> np, [s].
```

Deze regels, die nodig zouden kunnen zijn voor NP's met een genitief, zoals *Jan s huis, Jan s broer s huis*, introduceren links-recursie doordat een Det een NP als linker-dochter kan hebben.

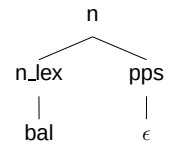
Er zijn twee oplossingen voor dit probleem. De eerste bestaat eruit dat de grammatica wordt aangepast, zodat links-recursie zich niet meer voordoet. De volgende regels maken het bijvoorbeeld ook mogelijk een N te laten volgen door een willekeurige reeks PP's.

(38)
```
n --> n_lex, pps.

n_lex --> [bal].

pps --> pp, pps.
pps --> [].
```

De categorie N wordt hier herschreven in twee categorieën N_lex en PPs. N_lex kan alleen maar worden herschreven als een woord. PPs is een (mogelijk) lege reeks PP's. De recursie betreft in dit geval niet langer de meest linkse dochter, zodat er niet langer gevaar voor een niet-terminerende lus bestaat.

De oplossing waarbij de grammatica wordt aangepast heeft als nadeel dat het vaak minder intuitieve analyses oplevert. De invoer `[bal]` zal bijvoorbeeld geanalyseerd worden als N bestaande uit een N_lex gevolgd door een lege PPs.

(39)



Overigens zouden we ook de meer gangbare boom kunnen opleveren:

(40)
```
n(B) --> n_lex(N), pps(N,B).

n_lex(b(n,[w(bal)])) --> [bal].

pps(N,PP) --> pp(N,PP1), pps(PP1,PP).
pps(N,N) --> [].
pp(N,b(n,[N,b(pp,[P,NP])])) --> p(P), np(NP).
```

De boom voor N_lex wordt hier doorgegeven aan PPs en aan PP. Binnen de regel voor PP wordt de boom N gebruikt om een boom te maken voor een N met als dochters een N en een PP. Deze boom wordt doorgegeven aan PPs, en uiteindelijk aan N. Door de opbouw van de boomstructuur niet meer geheel parallel te laten lopen aan de regels, kan een boomstructuur worden opgebouwd die correspondeert met een links-recursieve analyse, terwijl de regels niet links-recursief zijn.

Het bezwaar tegen deze methode voor het oplossen van links-recursie zit hem dus vooral in het feit dat ze het lastig maakt een grammatica op een overzichtelijke wijze te coderen. Dit geldt vooral ook wanneer de recursie niet onstaat binnen een enkele regel, maar het gevolg is van de interactie van verschillende regels. Dit bezwaar is minder van belang wanneer grammatica's met links-recursie automatisch kunnen worden omgezet in grammatica's zonder links-recursie. Zulke algoritmen bestaan, maar ze hebben als nadeel dat het aantal regels in de niet-linksrecursieve grammatica vele malen groter kan zijn dan in de originele grammatica.

De tweede oplossing bestaat uit het vervangen van de Prolog-bewijsstrategie door een parseermethode die minder gevoelig is voor links-recursie. Deze oplossing bespreken we in het volgende hoofdstuk.

2.8 Oefeningen

1. Waarom moet (om te bewijzen dat *de keeper mist de bal* een zin (`s`) is) een *definite clause grammar* (DCG) worden aanroepen met (41-a) en niet met (41-b)?

   (41) a. `s([de,keeper,mist,de,bal],[])`
   b. `s([de,keeper,mist,de,bal],A)`

2. Geef een *definite clause grammar* die het verschil tussen *de* en *het*, en tussen enkelvoud en meervoud in onderstaande zinnen verantwoordt. De grammatica mag *slechts 2 herschrijfregels* bevatten (en een willekeurig aantal regels die woorden introduceren).

het kind slaapt	*de kind slaapt
de kinderen slapen	*de kinderen slaapt
de jongen slaapt	*de jongen slapen
de jongens slapen	*het jongens slapen

3. Geef een definite clause grammar die hoogstens 3 regels bevat (afgezien van regels die een woord introduceren) die de zinnen in de linker kolom herkent en geen van de zinnen in de rechter kolom:

ik slaap	ik slaapt
hij slaapt	hij slapen
ik moet slapen	hij slaap
hij moet slapen	hij moet slaapt

4. De volgende *definite clause grammar* herkent de taal $a^n\ b^n$ voor $n \geq 1$, de taal die bestaat uit een rijtje a's gevolgd door een evenlang rijtje b's.

```
s --> a(I), b(I).

a(i(I)) --> a, a(I).
a(i) --> a.

b(i(I)) --> b, b(I).
b(i) --> b.
```

   Geef een DCG voor de taal *perm($a^n\ b^n$)* voor $n \geq 1$, de taal die bestaat reeksen met een *gelijk aantal* a's en b's maar waarbij de a's en de b's *in willekeurige volgorde* mogen staan.

Hoofdstuk 3

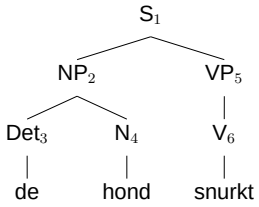
Automatisch ontleden

Automatisch ontleden (*parsing*) is één van de kerngebieden van de computationele taalkunde. Een ontleed-methode (*parser*) is een programma dat, gegeven een invoer en een formele grammatica, vaststelt welke analyses door de grammatica aan de invoer worden toegekend. Bij het automatisch analyseren van zinnen op basis van een CFG of DCG betekent dit dat het programma vaststelt welke *boomstructuren* door de grammatica aan een gegeven zin kunnen worden toegekend.

Er is een groot aantal verschillende ontleed-methoden voor natuurlijke taal. Dit is voor een deel het gevolg van het feit dat er verschillende grammatica-formalismen bestaan. Een grammatica gebaseerd op unificatie van (Prolog-)termen vereist een andere *parser* dan een CFG. Naast het formalisme kunnen ook 'toevallige' eigenschappen van de grammatica van belang zijn. Een ontleed-methode die efficient is voor grammatica G, hoeft dat niet te zijn voor G', zelfs als beide grammatica's hetzelfde formalisme gebruiken.

Van groot belang bij het analyseren van natuurlijke taal is het feit dat we meestal geïnteresseerd zijn in het vinden van *alle* analyses van een bepaalde invoer. Omdat uitingen in natuurlijke taal (zeer) ambigu kunnen zijn, betekent dit dat de meeste algoritmen die in de informatica gebruikt worden voor het analyseren van programmeertalen (als onderdeel van het compileren van een programma) niet geschikt zijn voor de analyse van natuurlijke taal. Programmeertalen zijn zorgvuldig ontworpen talen waarin ambiguïteit juist vermeden moet worden. Bovendien wordt bij het ontwerp van een programmeertaal rekening gehouden met het feit dat de taal automatisch verwerkt moet kunnen worden. Omdat er voor bepaalde deelverzamelingen van CFG zeer efficiente ontleed-methoden bestaan zijn de meeste programmeertalen dan ook zo opgezet dat ze met behulp van zo'n beperkte CFG gedefinieerd kunnen worden. Kenmerkend voor zulke grammatica's is echter het feit dat er geen ambiguïteiten kunnen optreden. Voor natuurlijke taal is dat een onrealistische eis. Dit maakt dat de algoritmen die gebruikt worden het ontleden van natuurlijke taal wezenlijk anders zijn dan de algoritmen die worden gebruikt bij het ontleden van programmeertaal.

S
NP VP
Det N VP
de N VP
de hond VP
de hond V
de hond snurkt



Figuur 3.1: Herschrijven van S tot de reeks *de hond snurkt*. De indices in de corresponderende boomstructuur geven de volgorde aan waarin symbolen worden herschreven.

Tenslotte speelt de toepassing waar de grammatica voor bedoeld is een rol. Naast het vinden van analyses voor grammaticale uitingen is soms ook de behandeling van ongrammaticale invoer essentieel. Een programma dat gesproken taal analyseert, bijvoorbeeld als onderdeel van een dialoogsysteem, zal regelmatig uitingen moeten analyseren die niet voldoen aan de regels van de grammatica. In zulke gevallen is vaak toch een analyse van delen van de invoer mogelijk, die kan worden gebruikt om de bedoeling van de spreker te achterhalen. Dit veronderstelt evenwel dat ontleden, naast een analyse van de hele uiting (als dit mogelijk is), altijd een analyse oplevert van mogelijke zinsdelen binnen een uiting. Deze veronderstelling is juist voor sommige, maar lang niet voor alle, ontleed-methoden.

In dit hoofdstuk bespreken we het automatisch ontleden van zinnen op basis van CFG. We introduceren een *bottom-up* alternatief voor de *top-down* parseer-strategie die standaard is voor DCG. Daarnaast besteden we aandacht aan *tabellen*, een veel gebruikte techniek voor het vergroten van de efficientie van het parseer-proces.

3.1 Top-down ontleden

Een CFG wordt soms gepresenteerd als een herschijfgrammatica. Een invoer $i_1 \ldots i_n$ voldoet aan de regels van de grammatica wanneer we het startsymbool van de grammatica (meestal S) kunnen herschrijven tot de reeks symbolen die zijn gegeven in de invoer. Iedere stap in het herschrijfproces correspondeert met een regel in de grammatica. We mogen een reeks $r_1 \ldots r_n$ herschrijven tot $r_1 \ldots r_{i-1} d_1 \ldots d_m r_{i+1} \ldots r_n$ wanneer er een regel $r_i \rightarrow d_1 \ldots d_m$ bestaat. Een voorbeeld, met de bijbehorende analyse-boom, wordt gegeven in figuur 3.1.

De indices van de boomstructuur in figuur 3.1 geven de volgorde aan waarin symbolen worden herschreven. De nummering geeft duidelijk aan dat er hier sprake is van een *top-down* proces. Dit wil zeggen dat moeders eerder worden herschreven dan dochters. Verder is er in dit geval sprake van een *links-naar-rechts* volgorde. Dit betekent dat tijdens het herschrijven altijd de meest linkse syntactische categorie als de te herschrijven categorie wordt geselecteerd.

Wanneer we een CFG implementeren als een DCG, levert de bewijsstrategie van Prolog een ontleed-methode op die identiek is aan *top-down* van links-naar-rechts herschrijven. Om te bewijzen dat `[de, hond, snurkt]` een zin is, moeten we achtereenvolgens het volgende bewijzen (aanroepen van het predicaat `'C'` (of een ander predicaat dat de introductie van woorden regelt) zijn weggelaten):

(1)
```
1. s([de, hond, snurkt], []).
2. np([de, hond, snurkt], P1), vp(P1, []).
3. det([de, hond, snurkt], P2), n(P2, P1), vp(P1, []).
4. n([hond, snurkt], P1), vp(P1, []).
5. vp([snurkt], []).
6. v ([snurkt], [])
```

Omdat de bewijsstrategie van Prolog top-down werkt, en omdat resolutie altijd probeert het meest linkse element van de *goal-stack* te *matchen* met een regel of feit uit het programma, levert de implementatie van een CFG als een DCG een *top-down*, links-naar-rechts, ontleder op.

```
parse(Cat,[Word|Tail],Tail) :-
  lex(Word,Cat).
parse(Cat,In,Out) :-
  rule(Cat,Dghtrs),
  parse_daughters(Dghtrs,In,Out).

parse_daughters([],List,List).
parse_daughters([D1|Dghtrs],In,Out) :-
  parse(D1,In,Mid),
  parse_daughters(Dghtrs,Mid,Out).
```

Figuur 3.2: Een top-down parser vergelijkbaar met de Prolog-bewijsstrategie voor DCG's.

Wanneer we willen experimenteren met verschillende parseer-strategieën, is het in het algemeen nuttig een scheiding aan te brengen tussen de grammatica (de regels en het lexicon, de data) en het parseer-algoritme. Bij het normale gebruik van een DCG is dit niet geval, omdat daar de regels ook rechtstreeks het parseer-algoritme opleveren. Regels en woorden zullen we daarom vanaf nu implementeren als Prolog-feiten van de vorm:

(2) a. `rule(Mother,List_of_Daughters)`
b. `rule(np,[det,n]).`
c. `lex(Word,Cat).`
d. `lex(aap,n).`

Een regel als np → det n correspondeert met het Prolog-feit in (2-b). Het predicaat `rule` heeft als eerste argument de categorie van de moeder, en als tweede argument een lijst met (de categorieën van de) dochters. Het predicaat `lex` neemt als eerste argument een woord, en als tweede argument de categorie van dit woord. Een *top-down* ontleder voor een grammatica in dit formaat staat in figuur 3.2. De goal `parse(s,[de,man,slaapt],[])` zal slagen wanneer de reeks *de man slaapt* volgens de regels van de grammatica van categorie `s` is.

3.2 Bottom-up ontleden

Een *bottom-up* ontleder bouwt een boomstructuur van onderen naar boven op, waarbij eerst de categorieën van de woorden in de invoer worden bepaald, en daarna de verschillende delen van de analyse worden samengenomen tot grote delen (op basis van grammaticaregels). Op deze manier worden knopen lager in de boom altijd eerder opgebouwd dan knopen daarboven.

De *top-down* ontleed-methode die hierboven is geschetst heeft, net als de normale bewijsmethode voor DCG's, als nadeel dat links-recursieve grammaticaregels leiden tot een niet-terminerende lus. *Bottom-up* ontleed-methoden hebben geen probleem met links-recursie.

3.2.1 Shift-reduce parsing

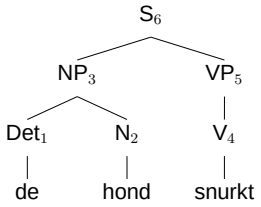
Een eenvoudige *bottom-up* ontleed-methode is de zogenaamde *shift-reduce*-methode. Deze methode maakt gebruik van een hulpstructuur, de *stapel* (*stack*). Het ontleden van een zin met behulp van de *shift-reduce*-methode houdt in dat de invoer van links naar rechts doorlopen wordt, en er ondertussen met behulp van de stapel wordt bijgehouden hoe de invoer geanalyseerd kan worden. Er zijn twee soorten acties mogelijk die de toestand van de stapel veranderen. De *shift*-actie houdt in dat een woord van de invoer wordt gelezen, de categorie van dit woord wordt bepaald, en dat deze categorie wordt toegevoegd aan de stapel. De *reduce*-actie houdt in dat een aantal elementen op de stapel wordt vervangen door één nieuw element. Een *reduce*-actie komt overeen met het toepassen van een grammaticaregel. Wanneer de elementen $D_1 \ldots D_n$ de top van de stapel vormen, mogen ze worden vervangen door een element M als er een grammaticaregel $M \rightarrow D_1 \ldots D_n$ is.

Het ontleden van de zin *de hond snurkt* met behulp van de *shift-reduce*-methode is schematisch weergegeven in figuur 3.3. De eerste twee stappen bestaan uit een *shift*-actie, waarbij *de* en *hond* worden ingelezen, en de corresponderende categorieën op de stapel gezet. Daarna is een *reduce*-actie mogelijk, op basis van de grammaticaregel NP → DET N. Vervolgens wordt *snurkt* verwerkt, en kunnen twee *reduce*-acties worden uitgevoerd. Uiteindelijk is de gehele invoer verwerkt, en is het enige symbool op de stapel S. Dit betekent dat de invoer is geanalyseerd als een reeks van categorie S.

De *shift-reduce* methode levert een *bottom-up* ontleder op. Figuur 3.4 geeft de boom die correspondeert met de afleiding in figuur 3.3. Indices geven ditmaal de volgorde aan waarin categorieën worden geïntroduceerd (d.w.z. op de stapel worden gezet) volgens de *shift-reduce*-methode. In dit geval geldt dat dochterknopen altijd geïntroduceerd worden voor moederknopen.

stap	invoer	stapel	actie
1.	*de hond snurkt*	[ ]	*shift*
2.	*hond snurkt*	[Det]	*shift*
3.	*snurkt*	[Det N]	*reduce*
4.	*snurkt*	[NP]	*shift*
5.		[NP V]	*reduce*
6.		[NP VP]	*reduce*
7.		[S]	

Figuur 3.3: Analyse van *de hond snurkt* met behulp van de *shift-reduce*-methode.



Figuur 3.4: Boomstructuur voor de afleiding in figuur 3.3. Indices geven de volgorde aan waarin categorieën op de stapel worden geplaatst.

NP	→	Det N	*het eendje*
N	→	N PP	*auto met een benzinemotor*
PP	→	P NP	*met een auto*
VP	→	VC	*slapen, heeft geslapen*
VP	→	PP VP	*van voetbal houden*
VC	→	V	*slaapt*
CP	→	Compl NP VP	*dat Jan een boek heeft gelezen*

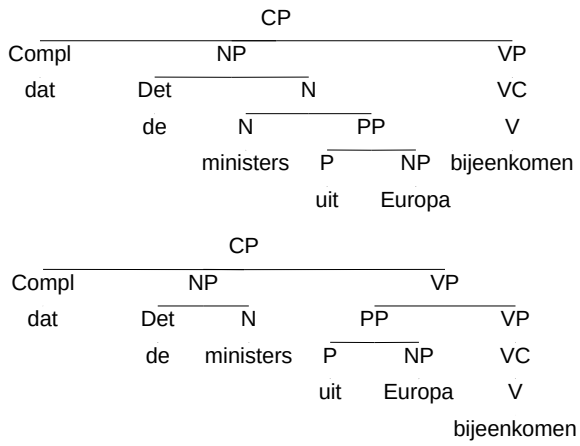
Figuur 3.5: Herschrijfregels voor een fragment van het Nederlands.

Tijdens het ontleden met behulp van de *shift-reduce*-methode moet er vaak worden gekozen tussen twee acties. Zolang niet de gehele invoer is verwerkt kan er altijd een *shift*-actie worden uitgevoerd. Daarnaast kan er een *reduce*-actie worden uitgevoerd wanneer de elementen op de stapel corresponderen met de dochters in een grammaticaregel. Wanneer er meerdere regels zijn die toegepast kunnen worden, moet er gekozen worden tussen verschillende *reduce*-acties. In het algemeen is niet van tevoren te bepalen welke keuze de juiste is. Dit betekent dat alle mogelijkheden onderzocht moeten worden wanneer we willen vaststellen of een zin geaccepteerd wordt door de grammatica, dan wel, wanneer we alle mogelijke analyses van een zin willen vinden.

Een illustratie van de manier waarop verschillende analyses corresponderen met verschillende acties van de ontleder kunnen we geven aan de hand van de regels in figuur 3.5 (een selectie van de regels in figuur 1.3, pag. 11). Deze grammatica kenmerkt zich onder andere door het feit dat er PP-ambiguïteiten mogelijk zijn. Een PP kan zuster zijn van een N, of een VP. Dit betekent o.a. dat, volgens de grammatica, zowel zin (3-a) als zin (3-b) twee mogelijke analyses heeft.

(3) a. (Juppé heeft vandaag gezegd) dat de ministers uit Europa bijeenkomen.
b. (Juppé heeft vandaag gezegd) dat de ministers uit voorzorg bijeenkomen.

De *betekenis* van de woorden in de zin leert ons dat in zin (3-a) de PP *uit Europa* waarschijnlijk een zuster van N is (omdat *uit Europa* gelezen wordt als een bepaling bij *ministers*), terwijl de PP *uit voorzorg* in (3-b) een zuster van VP is (omdat *uit voorzorg* wordt gelezen als een bepaling bij *bijeenkomen*).



Figuur 3.6: Twee analyses voor zin (3-a).

De afleidingen die corresponderen met deze bomen zijn gegeven in figuur 3.7. In toestand 5 zijn er twee mogelijkheden. Een *reduce*-actie (5a) betekent dat Det en N samen tot een NP vormen. De PP is in dat geval dus geen zuster van N. Een *shift*-actie (5b) leidt er uiteindelijk toe dat de PP als zuster van N geanalyseerd wordt. De eerste mogelijkheid leidt in stap 7 weer tot een situatie waarin het ontleedproces op twee verschillende manieren (succesvol) voortgezet kan worden. Kiezen voor een *shift*-actie (7aa) leidt ertoe dat de PP als zuster van de VP geanalyseerd wordt. Kiezen voor een *reduce*-actie (7ab) leidt ertoe dat de PP als zuster van S aangehecht wordt.

In het bovenstaande voorbeeld is sprake van een keuze tussen een *reduce*- en een *shift*-actie. Zo'n keuze noemen we ook wel een *shift-reduce*-conflict. Een keuze tussen twee verschillende *reduce*-acties is soms ook mogelijk. Dit doet zich voor wanneer twee verschillende regels kunnen worden gebruikt. Wanneer we de reeks *dat Brinkman een ramkoers heeft ingezet* ontleden, ontstaat op een gegeven moment de stapel ... Aux V. De grammatica bevat o.a. de volgende regels:

(4) a. VC → V
b. VC → Aux V

In principe kan nu een *reduce*-actie worden uitgevoerd op basis van de eerste regel (met als resultaat een stapel ... Aux VC) of op basis van de tweede regel (met als resultaat een stapel ... VC). Natuurlijk zal alleen de laatste actie tot een volledige afleiding leiden, maar het *shift-reduce*-algoritme zal toch beide mogelijkheden moeten onderzoeken. Hier is dus sprake van een *reduce-reduce*-conflict.

stap	invoer	stapel	actie
1.	*dat de ministers uit Europa bijeenkomen*	[ ]	*shift*
2.	*de ministers uit Europa bijeenkomen*	[Compl]	*shift*
3.	*ministers uit Europa bijeenkomen*	[Compl Det]	*shift*
4.	*uit Europa bijeenkomen*	[Compl Det N]	*shift*
5a.	*uit Europa bijeenkomen*	[Compl NP]	*reduce*
6a.	*Europa bijeenkomen*	[Compl NP P]	*shift*
7a.	*bijeenkomen*	[Compl NP P NP]	*reduce*
8a.	*bijeenkomen*	[Compl NP PP]	*shift*
9a.		[Compl NP PP V]	*reduce*
10a.		[Compl NP PP VC]	*reduce*
11a.		[Compl NP PP VP]	*reduce*
12a.		[Compl NP VP]	*reduce*
13a.		[CP]	
5b.	*uit Europa bijeenkomen*	[Compl Det N]	*shift*
6b.	*Europa bijeenkomen*	[Compl Det N P]	*shift*
7b.	*bijeenkomen*	[Compl Det N P NP]	*reduce*
8b.	*bijeenkomen*	[Compl Det N PP]	*reduce*
9b.	*bijeenkomen*	[Compl Det N]	*reduce*
9b.	*bijeenkomen*	[Compl NP]	*shift*
10b.		[Compl NP V]	*reduce*
11b.		[Compl NP VC]	*reduce*
12b.		[Compl NP VP]	*reduce*
13b.		[CP]	

Figuur 3.7: Twee analyses van (3-a) met behulp van de *shift-reduce*-methode.

3.2.2 Implementatie in Prolog

De *shift-reduce*-methode is eenvoudig in Prolog te implementeren. Zoals reeds werd opgemerkt in sectie 3.1, kunnen regels en het woordenboek als Prolog feiten worden gerepresenteerd:

(5) a. `rule(np,[det,n]).`
b. `lex(minister,n).`

Een stapel kunnen we beschouwen als een lijst. Een niet onbelangrijk verschil met de voorbeelden is dat we het begin van de lijst als de 'top' van de stapel zullen beschouwen. Elementen worden dus **vooraan** toegevoegd. Dit betekent dat de elementen op de stapel altijd in 'omgekeerde' volgorde staan. De *lijst* `[n, det]` correspondeert dus met de *stapel* [det n]. Om te bepalen of een *reduce*-actie mogelijk is, moeten de dochters in een regel worden vergeleken met de bovenste elementen op de stapel, in dit geval dus de eerste elementen van een lijst. Omdat de volgorde van elementen op de lijst het omgekeerde is van de volgorde van de dochters in een regel, wordt het predicaat `reverse/2` gebruikt.

Figuur 3.8 geeft een implementatie van de *shift-reduce*-methode in Prolog. Wanneer we bijvoorbeeld willen bepalen of *de oude man* een np is, beginnen we met de aanroep:

(6) `?- shift_reduce([de,oude,man],np).`

Ontleden begint met een lege stapel. Daarna wordt steeds een *reduce*-actie of een *shift*-actie uitgevoerd. De invoer is succesvol ontleed als de gehele invoer kan worden verwerkt, en de stapel kan worden gereduceerd tot `TopCat`.

De bovenstaande implementatie is minder efficient dan nodig, doordat in het predicaat `reduce/2` steeds een aanroep van `reverse/2` en `append/3` nodig is (om te bepalen of de n dochters van een regel in omgekeerde volgorde overeenkomen met de eerste n elementen van de lijst). Wanneer een shift-reduce-parser wordt gebruikt om alle afleidingen van een zin te vinden, of wanneer de invoer ongrammaticaal was, zal de parser tijdens iedere stap in de afleiding alle regels proberen toe te passen. Het omdraaien van de dochters in een regel is echter geheel onafhankelijk van de afleiding, en kan dus beter éénmalig, van tevoren, worden uitgevoerd. Ook het gebruik van `append` om een lijst in twee delen te splitsen is in het algemeen inefficient.

Deze inefficientie kan dan ook eenvoudig worden vermeden, door de manier waarop regels worden gecodeerd (als `rule(Mother,Daughters)`) aan te passen aan het gebruik binnen het shift-reduce-algoritme. Een regel als np → det n maakt het mogelijk om een stapel (lijst) `[n,det|Rest]` te reduceren tot een stapel `[np|Rest]`. Je zou dus kunnen zeggen dat de regel de volgende `reduce`-relatie waar maakt:


```
shift_reduce(Words,TopCat) :-
  shift_reduce(Words,[],TopCat).

shift_reduce([],[TopCat],TopCat).
shift_reduce(Words,Stack,TopCat) :-
  reduce(Stack,NewStack),
  shift_reduce(Words,NewStack,TopCat).
shift_reduce(Words,Stack,TopCat) :-
  shift(Words,NewWords,Cat),
  shift_reduce(NewWords,[Cat|Stack],TopCat).

reduce(Stack,[Mother|Rest]) :-
  rule(Mother,Daughters),
  reverse(Daughters,ReversedDaughters),
  append(ReversedDaughters,Rest,Stack).

shift([Word|Words],Words,Cat) :-
  lex(Word,Cat).
```

Figuur 3.8: Implementatie van het shift-reduce-algoritme in Prolog.

(7) `reduce([n,det|Rest],[np|Rest]).`

Wanneer we alle regels opschrijven zoals in (7), kan de definitie van `reduce` in figuur 3.8 worden geëlimineerd. Hiermee zijn ook alle aanroepen van `reverse` en `append` geëlimineerd.

Je zou kunnen volhouden dat het formaat in (7) lastiger te lezen en te begrijpen is (en wellicht gemakkelijker tot fouten leidt) dan de implementatie met `rule`. In dat geval kun je een predicaat schrijven dat automatisch `reduce`-feiten afleidt uit `rule`-feiten. Deze compilatie-stap hoeft alleen te worden uitgevoerd wanneer de grammatica wordt geladen, en niet tijdens het ontleden zelf.

3.2.3 Links-recursie en Epsilons

Links-recursieve regels zijn geen probleem voor de *shift-reduce*-methode, omdat de methode *bottom-up* werkt. Dit betekent onder andere dat een grammaticaregel pas kan worden toegepast nadat alle dochters van de regel zijn gevonden. Een regel als N → N PP kan dus k maal worden toegepast, als er ook inderdaad k PP's voorhanden zijn. Omdat de methode *bottom-up* werkt moeten die PP's afgeleid worden voordat de regel kan worden toegepast. Wanneer een N gevolgd wordt door twee PP's wordt de regel dus ook hoogstens twee maal toegepast.

De *bottom-up* strategie heeft helaas een andere zwakke plek. In een *top-down* methode, zoals de Prolog-behandeling van DCG's vormen regels die een leeg element introduceren in het algemeen geen probleem. Voor de *shift-reduce*-methode ligt dat anders. Stel dat we aan onze voorbeeld grammatica de volgende regel toevoegen:

(8) $\mathsf{Det} \rightarrow \epsilon$

Met behulp van deze regel zouden we bijvoorbeeld de reeks *oude boeken* als een NP (bestaande uit een (lege) *determiner*, een adjectief, en een zelfstandig naamwoord) kunnen analyseren. Wanneer deze regel echter als *reduce*-actie wordt gebruikt betekent dit dat er een Det aan de stapel kan worden toegevoegd, zonder dat er een bepaald element (corresponderend met de dochter van de regel) van de stapel verwijderd wordt. Immers, het corresponderende `reduce`-feit is:

(9)
```
reduce(Stack,[det|Stack]).
```

Deze regel is op iedere stapel toepasbaar, met name ook op een stapel die is gecreeerd door toepassen van dezelfde regel. De lus die hierdoor onstaat leidt tot een niet-terminerend programma, net zoals links-recursie dat veroorzaakt bij een *top-down* strategie.

Een oplossing die voor dit geval wellicht effectief is, is het verbieden van stapels die twee of meer achtereenvolgende Det's bevatten. Zo'n oplossing vereist echter dat de ontwerper van de grammatica bedenkt welke patronen van categorieën wel en niet voor kunnen komen in de taal. Dit kan een niet-triviale opgave zijn.

Een betere oplossing is het 'wegcompileren' van ϵ-regels. Dit gaat op basis van de volgende overweging:

> Een grammatica G herkent dezelfde taal als grammatica G' (met ϵ-regels) als G alle niet-ϵ-regels van G' bevat en bovendien voor iedere regel $\mathrm{D}_i \rightarrow \epsilon$ in G' en iedere regel $\mathsf{M} \rightarrow \mathsf{D}_1 \ldots \mathsf{D}_i \ldots \mathsf{D}_n$ in G' geldt dat er een corresponderende regel $\mathsf{M} \rightarrow \mathsf{D}_1 \ldots \mathsf{D}_{i-1}, \mathsf{D}_{i+1} \ldots \mathsf{D}_n$ in G is.

Wanneer G' bijvoorbeeld de regels $\mathsf{NP} \rightarrow \mathsf{Det\ N}$ en $\mathrm{DET} \rightarrow \epsilon$ bevat, zal G de regel $\mathsf{NP} \rightarrow \mathsf{N}$ bevatten. Op basis van dit inzicht kunnen we iedere CFG waarin gebruik wordt gemaakt van ϵ-regels omzetten in een CFG waarin geen gebruik wordt gemaakt van ϵ-regels, maar die wel dezelfde taal accepteert. In de meeste gevallen zal het éénmaal toepassen van bovenstaande overweging een grammatica G opleveren die geen ϵ-regels bevat. In speciale gevallen kan het echter gebeuren dat de extra regels die in G nodig zijn, zelf weer ϵ-regels zijn. In dat geval moet de compilatiestap worden herhaald tot er geen ϵ-regels meer worden toegevoegd.

3.3 Breadth-first ontleden

De *shift-reduce*-methode die we hierboven hebben besproken, en ook de *top-down*-methode die normaalgesproken voor DCG's wordt gebruikt, maken gebruik van een *depth-first*, *backtracking*, strategie om de verschillende analyses van een zin te onderzoeken. Wanneer bijvoorbeeld gekozen kan worden tussen een *shift*- en een *reduce*-actie, worden beide mogelijkheden één voor één onderzocht.

Een ontleed-methode gebaseerd op *backtracking* heeft als nadeel dat er veel werk dubbel gedaan wordt. Neem bijvoorbeeld de *shift-reduce* analyse van de zin marie ziet de jongen met de hond. Er zijn, volgens de grammatica gegeven in de vorige sectie, drie analyses van deze zin mogelijk. Alle drie de analyses bevatten de constituent met de hond. De *shift-reduce*-analyse leidt deze constituent dan ook drie maal af. Dit betekent dat er werk dubbel gedaan wordt. In een eenvoudig geval zoals lijkt dit misschien een detail, maar in de praktijk is het, met name voor grotere grammatica's en langere zinnen (een combinatie de gemakkelijk leidt tot een explosieve toename van het aantal ambiguïteiten), essentieel dat deze inefficientie vermeden wordt.

Ambiguïteit is niet de enige oorzaak van dubbel werk. Aangezien we in het algemeen alle mogelijke analyses van een zin willen vinden, zullen alle mogelijke combinaties van *shift*- en *reduce*-acties onderzocht moeten worden. Hierbij zullen nogal wat analyses zitten die niet tot succes leiden. Ondertussen kunnen echter wel grote delen van de invoer op een correcte manier geanalyseerd worden. Deze partiele analyses zullen ook in succesvolle analyses gebruikt worden. Neem bijvoorbeeld de zin

(10) In het voorstel van de commissie-Donner, die over de toekomst van de WAO heeft geadviseerd, is daarvan veel terug te vinden.

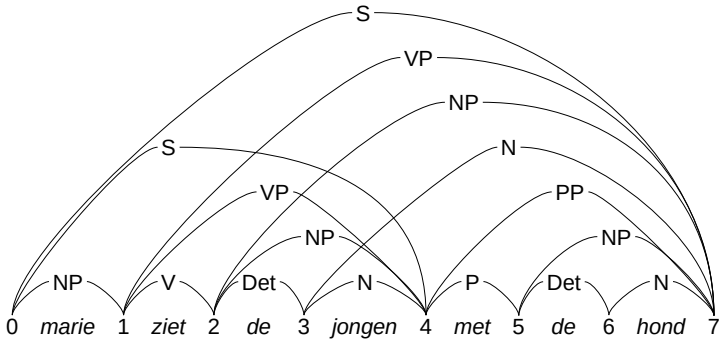
Na het verwerken van de eerste drie woorden van de zin bestaat de stapel bijvoorbeeld uit [P Det N]. Wanneer op dit moment een *reduce*-actie wordt uitgevoerd, is geen succesvolle analyse mogelijk. De PP *van de commissie-Donner, die...geadviseerd* moet namelijk eerst met N gecombineerd worden tot een grotere N. Het algoritme heeft hier echter geen weet van, en zal proberen de rest van de input te analyseren. Hierbij wordt hoogswaarschijnlijk de reeks *van de commissie-Donner, die...geadviseerd* op enig moment correct tot een PP gereduceerd. Om de fout die aan het begin is gemaakt echter te herstellen moet *backtracking* plaatsvinden tot het moment waar slechts [P Det N] op de stapel staan. Tijdens *backtracking* gaat de correcte analyse van de PP *van de commissie-Donner, die...geadviseerd* dus verloren. Ook het onderzoeken van analyses die uiteindelijk geen succes opleveren, leidt dus tot dubbel werk.

De meeste practische ontleed-methoden voor natuurlijke taal maken dan ook gebruik van een andere strategie. In plaats van *depth-first* wordt gebruik gemaakt van een *breadth-first* zoekstrategie. Alle mogelijke analyses van de invoer worden parallel onderzocht, waarbij gedeeltelijke analyses steeds voor hergebruik in aanmerking komen.

Ontleed-methoden die niet gebaseerd zijn op *backtracking* maken allemaal gebruik van een *tabel* (*chart*) waarin gedeeltelijke resultaten worden opgeslagen. Gedeeltelijke resultaten zijn analyses van losse zinsdelen. Grotere analyses kunnen worden afgeleid door kleinere delen te combineren (op basis van de regels van de grammatica). Omdat alle analyses in een tabel worden opgeslagen, hoeft de analyse van ieder deel echter slechts éénmaal te worden uitgevoerd.

De rol van een tabel met zinsdelen (*wellformed substring table*) kan worden geïllustreerd aan de hand van het volgende voorbeeld.

(11) a. marie ziet de jongen met de hond.
b.
S → NP VP	PP → P NP
S → S PP	VP → V NP
NP → Det N	VP → VP PP
N → N PP	



Figuur 3.9: Zinsdelen in marie ziet de jongen met de hond

In figuur 3.9 geven we een overzicht van de zinsdelen die we in zin (11-a) kunnen ontdekken, op basis van de regels in (11-b). Van positie 0 tot 1 is er een zinsdeel NP. Van positie 2 tot 4 is er een zinsdeel NP. Van positie 4 tot 7 een PP, etc. In plaats van een grafische weergave kunnen zinsdelen ook als een lijst of tabel weergeven.

(12) ⟨0,1,NP⟩
⟨1,2,V⟩
⟨2,4,NP⟩
⟨4,7,PP⟩
...

De elementen in de tabel hebben drie argumenten: een begin- en eindpositie, en een categorie.

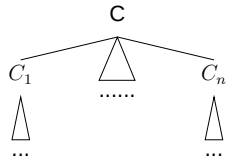
Een chart-parser onderzoekt systematisch welke zinsdelen er in een zin ontdekt kunnen worden, en plaatst deze in een tabel. Nieuwe zinsdelen worden toegevoegd op basis van zinsdelen die al in de chart staan. Elementen in de chart blijven (gedurende de parse van een zin) altijd bestaan, en dus hoeft een constituent nooit tweemaal te worden afgeleid. Wanneer we zin (10) bijvoorbeeld met een chart-parser analyseren, zal op een gegeven moment in de chart staan:

(13) ⟨0,1,P⟩
⟨1,2,Det⟩
⟨2,3,N⟩
⟨3,15,PP⟩
...

Op dit moment kan ⟨1,3,NP⟩ aan de chart worden toegevoegd, maar ook ⟨2,15,N⟩ (en daarna ⟨1,15,NP⟩). Hiervoor hoeft de PP *van de commissie-Donner, die...geadviseerd* echter niet opnieuw geanalyseerd te worden.

Alle mogelijke analyses van de invoer zijn impliciet aanwezig in de tabel met zinsdelen. Om bijvoorbeeld de boomstructuur voor de gehele invoer van (11-a) te reconstrueren zoeken we naar een zinsdeel met categorie C van positie 0 tot 7. Als de grammatica een regel $C \rightarrow C_1 ... C_n$ bevat, kunnen we vervolgens kijken of er ook een reeks kleinere zinsdelen C_1 tot en met C_n te vinden is, die samen ook de gehele invoer beslaan. Als dit het geval is, bestaat er een analyse die als top de volgende boomstructuur heeft:

(14)



Vervolgens kunnen de mogelijke analyses van de delen van de zin op dezelfde wijze gereconstrueerd worden. Eén constituent correspondeert soms met meerdere mogelijke analyses. De constituent ⟨5,7,NP⟩ heeft maar één mogelijke analyse. De constituent ⟨1,7,VP⟩ kan echter op twee manieren zijn opgebouwd. Ze kan zijn gevormd door de combinatie ⟨ 1,2,V⟩ en ⟨2,7,NP⟩, maar ook door de combinatie van ⟨1,4,VP⟩ en ⟨4,7,PP⟩. De constituent ⟨0,7,S⟩ kan zelfs op drie manieren worden geanalyseerd. Eén analyse maakt gebruik van de zinsdelen ⟨0,4,S⟩ en ⟨4,7,PP⟩. De twee andere analyses maken gebruik van de zinsdelen ⟨0,1,NP⟩ en ⟨1,7,VP⟩. Omdat de VP hier zelf weer twee mogelijke analyses heeft, levert dit dus in totaal drie mogelijke analyses voor S op.

Merk op dat het reconstrueren van bomen op basis van een chart zelf weer een zekere hoeveelheid zoeken vereist. Zoeken kan worden vermeden wanneer we enige extra administratie aan de chart toevoegen. Wanneer we items nummeren, kunnen we (in een aparte tabel) voor ieder item ook bijhouden op basis van welke andere items het geconstrueerd werd. Bij het reconstrueren van bomen hoeven we dan alleen maar in deze chart te kijken welke dochters zijn gebruikt voor de constructie van een bepaald chart item.

Het gebruik van een chart heeft nog een belangrijk voordeel. In veel toepassingen komt het regelmatig voor dat een bepaalde invoer niet geanalyseerd kan worden. Een zin kan bijvoorbeeld typefouten bevatten, of grammaticale fouten. In zulke gevallen kan het toch nuttig zijn delen van de zin als een zinsdeel te kunnen herkennen. Een chart bevat altijd alle constituenten die in een bepaalde invoer ontdekt kunnen worden, en kan dus worden gebruikt om zinsdelen in een invoer te ontdekken, zelfs wanneer de invoer als geheel niet geanalyseerd kan worden. Zonder chart is dit vrijwel onmogelijk.

3.3.1 Implementatie in Prolog

Een *breadth-first*, *bottom-up* ontleed-methode die gebruik maakt van een chart of een tabel, kan in Prolog als volgt geïmplementeerd worden.

Zinsdelen geven we weer als feiten van de vorm

(15)
```
item(0,1,np).
```

De tabel is een verzameling van dergelijke feiten. Tijdens het parsen wordt de tabel gevuld. Dit betekent dat `item`-feiten dynamisch aan het programma moeten kunnen worden toegevoegd. In Prolog bereiken we dit met behulp van het (*built-in*) predicaat `assert/1`.

Regels hebben de vorm

(16)
```
rule([vp,np],s).
```

Merk op dat de volgorde van de dochters van de regel weer is omgedraaid, net zoals in de *shift-reduce*-methode. Voor het woordenboek wordt het predicaat `lex(Word,Cat)` gebruikt.

```
chart(Words) :-
  retractall(item(_,_,_)),
  chart(0,Words).

chart(Pos0,[Word|Words]) :-
  Pos1 is Pos0 + 1,
  shift(Pos0,Pos1,Word),
  chart(Pos1,Words).
chart(_,[]).

shift(Pos0,Pos1,Word) :-
  lex(Word,Cat),
  make_item(Pos0,Pos1,Cat),
  fail.
shift(_,_,_).

make_item(Pos0,Pos1,Cat) :-
  item(Pos0,Pos1,Cat),
  !.
make_item(Pos0,Pos1,Cat) :-
  assert(item(Pos0,Pos1,Cat)),
  reduce(Pos0,Pos1,Cat).

reduce(Pos1,Pos2,Cat) :-
  rule([Cat|Cats],Mother),
  find_sisters(Pos0,Pos1,Cats),
  make_item(Pos0,Pos2,Mother).
  fail.
reduce(_,_,_).

find_sisters(Pos0,Pos0,[]).
find_sisters(Pos0,Pos2,[Cat|Cats]):-
  item(Pos1,Pos2,Cat),
  find_sisters(Pos0,Pos1,Cats).
```

Figuur 3.10: Een *bottom-up chart parser* in Prolog.

De opzet van de chart-parser is vergelijkbaar met de *shift-reduce*-methode omdat we tijdens het verwerken van de invoer weer *bottom-up* zullen werken, en er steeds *shift*- en *reduce*-acties mogelijk zijn. Er zijn twee belangrijke verschillen. In de methode hieronder wordt geen gebruik gemaakt van een stapel. De rol van de stapel wordt nu overgenomen door de chart. Daarnaast worden de invoer slechts éénmaal, van links naar rechts, doorlopen. Na ieder woord, worden steeds alle *shift*- en *reduce*-acties die mogelijk zijn op dit punt in de invoer uitgevoerd. Het laatse is essentieel voor een *breadth-first* strategie. In Prolog kiezen we ervoor het effect van *breadth-first* zoeken te implementeren met behulp van een zogenaamde *failure-driven loop*.

De code voor de *breadth-first, shift-reduce*-methode staat in figuur 3.10. Ontleden van een zin begint met een aanroep als:

(17) `chart([marie,ziet,de,jongen]).`

Als eerste stap wordt de tabel met zinsdelen schoongemaakt. Dit betekent dat alle feiten van de vorm `item(P1,P2,Cat)` gewist worden.[1] Daarna begint het woord voor woord verwerken van de invoer. De positie in de invoer wordt bijgehouden met een teller (`Pos`), die in eerste instantie gelijk aan 0 is.

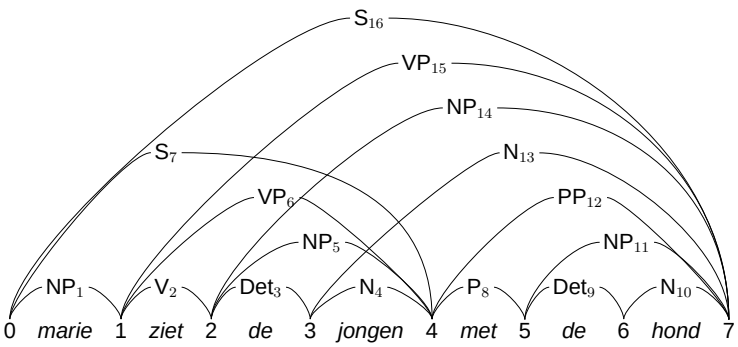
Zowel de *shift*- als de *reduce*-acties zijn *breadth-first* geïmplementeerd. Dit bereiken we door de eigenlijke *shift*- en *reduce*-acties in een zogenaamde *failure-driven loop* in te bedden. Alle mogelijke manieren om een *shift*- of een *reduce*-actie uit te voeren worden dus steeds uitgeprobeerd.

Het verwerken van de invoer gebeurt door steeds (*breadth-first*) *shift*-acties uit te voeren. Een *shift*-actie betekent in dit geval dat de categorie van een woord wordt opgezocht, en dat een `item(`$\texttt{Pos}_0$`,`$\texttt{Pos}_1$`,Cat)` aan de tabel wordt toegevoegd (waarbij $\texttt{Pos}_0$ en $\texttt{Pos}_1$ de begin- en eindpositie weergeven, en `Cat` de categorie van het woord). Toevoegen van een zinsdeel gebeurt door het predicaat `make_item`. Wanneer de tabel al een `item` bevat met dezelfde begin- en eindposities en categorie als het nieuw toe te voegen zinsdeel, doen we niets. Wanneer dit niet het geval is, wordt het nieuwe zinsdeel met behulp van `assert` toegevoegd. Bovendien wordt op dit moment (*breadth-first*) *reduce* aangeroepen. Iedere keer wanneer een nieuw item wordt toegevoegd, kan dit namelijk één of meer nieuwe *reduce*-acties mogelijk maken.

[1] In Sicstus Prolog werkt `retract` en `retractall` alleen voor predicaten die dynamisch zijn gemaakt. Dit kun je doen door de declaratie

```
:- dynamic item/3.
```

aan het programma toe te voegen.



Figuur 3.11: Items voor de zin *marie ziet de jongen met de hond*

Een *reduce*-actie wordt in dit geval uitgevoerd als de nieuw toegevoegde constituent correspondeert met de meest rechtse dochter van een regel (het eerste element in de lijst dochters van een `rule`), en bovendien alle andere dochters van de regel al in de tabel aanwezig zijn. Dit laatste wordt gecontroleerd door het predicaat `zusters`. Dit predicaat zoekt naar de zusters van het nieuw gevonden zinsdeel. Merk op dat de dochters van een `bf_reduce_regel` in omgekeerde volgorde staan, zodat bij het vinden van zusters van rechts naar links gewerkt moet worden. Dit heeft als voordeel dat de eindpositie van een zuster steeds bekend is. Als alle zusters gevonden zijn, wordt een zinsdeel $\langle \texttt{Pos}_0, \texttt{Pos}_1, \texttt{M} \rangle$ toegevoegd, waarbij $\texttt{Pos}_0$ de beginpositie van de meest linkse dochter is, $\texttt{Pos}_1$ de eindpositie van de meest rechtse dochter, en `M` de categorie van de moeder. Merk op dat het toevoegen van dit zinsdeel op zich weer aanleiding kan geven tot nieuwe *reduce*-acties.

De volgorde waarin zinsdelen worden gevonden door `bf_shift_reduce` staat aangegeven in figuur 3.11. De volgorde waarin de zinsdelen S_{16}, VP_{15} en N_{13} worden gevonden hangt af van de volgorde waarin de regels $\mathsf{N} \rightarrow \mathsf{N\ PP}$, $\mathsf{VP} \rightarrow \mathsf{VP\ PP}$ en $\mathsf{S} \rightarrow \mathsf{S\ PP}$ worden toegepast. In het voorbeeld wordt ervan uitgegaan dat deze regels worden geprobeerd in de zojuist gegeven volgorde. Wanneer we een andere volgorde kiezen, zullen deze zinsdelen ook in een andere volgorde worden toegevoegd.

3.4 Literatuur

De implementatie van verschillende ontleed-methoden voor natuurlijke taal in Prolog, waarbij verschillende die we hierboven niet besproken hebben (de *left-corner*-methode, *active chart*-methoden, en Earley's algoritme) worden besproken in Pereira en Shieber (1987). Van Noord (1997) bespreekt een efficiënte implementatie in Prolog van een variant van de bottom-up chart-parsing methode.

3.5 Oefeningen

1. (a) Wanneer is een grammatica links-recursief?
   (b) Geef een grammatica-fragment dat links-recursie bevat, en laat zien hoe dit fragment omgezet kan worden in een fragment dat niet langer links-recursief is.
   (c) Waarom is een links-recursieve DCG normaalgesproken problematisch?
   (d) Waarom is links-recursie geen probleem voor een bottom-up parser?

2. Gegeven is de volgende contextvrije grammatica G:

S → NP VP	Det → *de*
NP → Det N	N → *aanvallers*
N → A N	N → *goals*
N → N PP	N → *minuut*
PP → P NP	V → *scoren*
VP → V	V → *juichen*
VP → V NP	P → *in*
VP → VP PP	A → *laatste*

   (a) Beschrijf wat een *shift-reduce conflict* is aan de hand van grammatica G en een voorbeeldzin.
   (b) Geef alle zindelen of *chart items* die een *bottom-up chart parser* op basis van deze grammatica zou produceren voor de zin *De aanvallers scoren de goals in de laatste minuut*.
   (c) De grammatica E bevat naast alle regels van G ook de regel: $\mathsf{Det} \rightarrow \epsilon$. Welk effect heeft het toevoegen van zo'n regel op een *chart parser*? (Je mag ervan uitgaan dat de parser geen items toevoegt die identiek zijn aan items die reeds op de chart staan.)
   (d) Hoe ziet een grammatica eruit die dezelfde zinnen herkent als grammatica E maar die niet de regel $\mathsf{Det} \rightarrow \epsilon$ bevat?

Hoofdstuk 4

Unificatie-grammatica

4.1 Inleiding

Taalkundigen maken graag gebruik van kenmerken (*features*) om klanken en categorieën te beschrijven. Het gebruik van kenmerken maakt het mogelijk om, bijvoorbeeld, klanken in klassen in te delen. De medeklinkers b, m en n behoren bijvoorbeeld tot de klasse van de stemhebbende klinkers, maar p niet. Uit de matrices hieronder blijkt ook dat b en p nauw verwante klanken zijn (die slechts in het kenmerk STEMHEBBEND van elkaar verschillen).

(1) a. $\mathsf{b} = \begin{bmatrix} \text{sonorant} & - \\ \text{stemhebbend} & + \\ \text{nasaal} & - \\ \text{labiaal} & + \\ \text{alveolair} & - \end{bmatrix}$ $\mathsf{m} = \begin{bmatrix} \text{sonorant} & + \\ \text{stemhebbend} & + \\ \text{nasaal} & + \\ \text{labiaal} & + \\ \text{alveolair} & - \end{bmatrix}$

b. $\mathsf{n} = \begin{bmatrix} \text{sonorant} & + \\ \text{stemhebbend} & + \\ \text{nasaal} & + \\ \text{labiaal} & - \\ \text{alveolair} & + \end{bmatrix}$ $\mathsf{p} = \begin{bmatrix} \text{sonorant} & - \\ \text{stemhebbend} & - \\ \text{nasaal} & - \\ \text{labiaal} & + \\ \text{alveolair} & - \end{bmatrix}$

Klanken die nauw verwant zijn verwachten we vaak samen in een regel aan te treffen. Het Nederlands kent bijvoorbeeld een regel die niet-sonorante (stemhebbende) medeklinkers aan het eind van een woord omzet in stemloze medeklinkers (b wordt p, d wordt t, g wordt ch, v wordt f, en z wordt s). In zo'n regel spelen kenmerken op twee manieren een rol. Ten eerste geldt de regel voor alle medeklinkers die [SONORANT -] zijn. Een systeem dat dit kenmerk ontbeert, zal deze regel dus minder direct (als een verzameling regels, of als een regel met een ingewikkelder 'inputconditie') moeten uitdrukken. Ten tweede wordt iedere medeklinker M waarop de regel van toepassing is, omgezet in de medeklinker M' die slechts van M verschilt in het kenmerk STEMHEBBEND (b wordt dus p en niet, bijvoorbeeld, t). Een systeem waarin deze verwantschappen niet volgen uit de gebruikte kenmerken zal haar toevlucht moeten nemen tot minder elegante regels.

Het gebruik van kenmerken maakt het dus mogelijk *generalisaties* uit te drukken. Binnen de syntaxis maakt men om die reden ook veel gebruik van kenmerken. De categorieën NP, VP, AP, en PP worden bijvoorbeeld vaak geanalyseerd in termen van de kenmerken N en V. Daarnaast kan de verwantschap tussen NP, VP ,AP, PP en N, V, A en P worden uitgedrukt met behulp van het kenmerk BAR.

(2)
a. $\text{np} = \begin{bmatrix} \text{n} & + \\ \text{v} & - \\ \text{bar} & 2 \end{bmatrix} \quad \text{n} = \begin{bmatrix} \text{n} & + \\ \text{v} & - \\ \text{bar} & 0 \end{bmatrix}$

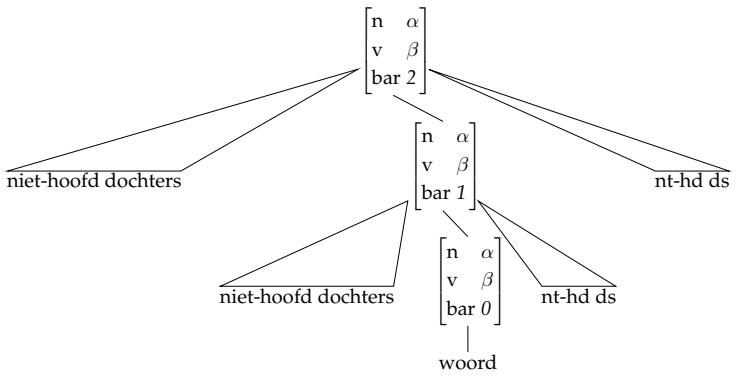
b. $\text{ap} = \begin{bmatrix} \text{n} & + \\ \text{v} & + \\ \text{bar} & 2 \end{bmatrix} \quad \text{a} = \begin{bmatrix} \text{n} & + \\ \text{v} & + \\ \text{bar} & 0 \end{bmatrix}$

c. $\text{vp} = \begin{bmatrix} \text{n} & - \\ \text{v} & + \\ \text{bar} & 2 \end{bmatrix} \quad \text{v} = \begin{bmatrix} \text{n} & - \\ \text{v} & + \\ \text{bar} & 0 \end{bmatrix}$

d. $\text{pp} = \begin{bmatrix} \text{n} & - \\ \text{v} & - \\ \text{bar} & 2 \end{bmatrix} \quad \text{p} = \begin{bmatrix} \text{n} & - \\ \text{v} & - \\ \text{bar} & 0 \end{bmatrix}$

Deze kenmerken spelen bijvoorbeeld in de *X-bar*-theorie een belangrijke rol. Binnen deze theorie gaat men uit van de stelling dat constituenten altijd de structuur in figuur 4.1 hebben. Een constituent heeft een woord als kern of *hoofd*. Woorden hebben altijd het kenmerk [BAR 0]. Een grotere constituent wordt gevormd door een hoofd te combineren met andere zinsdelen. Deze andere zinsdelen zijn *complementen* (wanneer een werkwoord combineert met een lijdend of meewerkend voorwerp is de laatste het complement) of *adjuncten* (bijvoegelijke bepalingen bij een werkwoord of bijvoegelijke naamwoorden bij een zelfstandig naamwoord). De categorie van de resulterende constituent heeft dezelfde waarden voor de kenmerken N en V als het hoofd, maar die is gespecificeerd als [BAR 1]. Daarna kan vaak nog een grotere constituent worden gevormd door te combineren met een *specifier* (bijvoorbeeld een lidwoord bij een zelfstandig naamwoordgroep) of een *subject* (het onderwerp van een werkwoordsgroep). Door gebruik te maken van kenmerken kan men dus binnen de X-bar theorie uitdrukken wat de verschillende zinsdelen gemeenschappelijk hebben, en kan men bij de formulering van regels volstaan met een klein aantal regelschema's in plaats van aparte regels voor iedere categorie.

Er zijn een aantal taalkundige theorieën die gebaseerd zijn op het idee dat alle taalkundig relevante informatie in de vorm van (complexe) matrices van kenmerken kan worden uitgedrukt. Belangrijke vertegenwoordigers van deze stroming zijn *Generalized Phrase Structure Grammar* (GPSG), *Lexical-Functional Grammar* (LFG) en *Head-driven Phrase Structure Grammar* (HPSG). Naast het feit dat men zeer veel aandacht besteed aan de rol van kenmerken hebben deze theorieën ook gemeenschappelijk dat ze *non-transformationeel* zijn. Dit wil zeggen dat ze geen gebruik maken van transformaties, dit in tegenstelling tot de transformationele grammatica en haar nakomelingen (zoals de *Government and Binding*-theorie en het *Minimalisme*).



Figuur 4.1: Constituentenstructuur volgens de X-bar theorie

Non-transformationele theorieën zijn populair in de computationele taalkunde. De belangrijkste reden hiervoor is dat grammatica's die gebruik maken van transformaties lastig te implementeren zijn, en vrijwel nooit leiden tot robuuste of efficiente systemen. Non-transformationele theorieën voldoen in dit opzicht beter. De meeste van deze theorieën maken gebruik van een kern die bestaat uit een context-vrije grammatica. Aan de regels van deze grammatica worden extra condities toegevoegd, bijvoorbeeld doordat de symbolen in de regels worden voorzien van kenmerken. De computationele aspecten van ontleden op basis van een context-vrije grammatica zijn goed onderzocht. Het toevoegen van kenmerken aan dergelijke grammatica's blijkt bovendien geen onoverkomelijk probleem voor zinsontleden op te leveren. Dit maakt dat non-transformationele grammatica's bij uitstek geschikt zijn voor computationele toepassingen.

In feite hebben we al kennis gemaakt met een voorbeeld van een nontransformationele theorie. *Definite-clause grammar* (DCG) maakt gebruik van regels die overeenkomen met de herschrijfregels van een context-vrije grammatica. Aan de symbolen in de regels kunnen extra argumenten worden toegevoegd die overeenkomen met het gebruik van kenmerken in taalkundige theorieën. DCG's maken bovendien gebruik van *unificatie* om de waarde van bepaalde argumenten te instantiëren, dan wel om uit te drukken dat twee argumenten in waarde overeen moeten stemmen. Bovengenoemde nontransformationele theorieën maken ook gebruik van unificatie, zij het dat in dit geval (matrices van) kenmerken geunificeerd worden. Gezien de centrale rol van unificatie in theorieën als GPSG, LFG en HPSG, en in computationele formalismen als DCG wordt deze familie van taalkundige formalismen ook wel aangeduid met de verzamelterm *unificatie grammatica* (UG).

In de volgende sectie bespreken we de belangrijkste theoretische concepten van unificatie-grammatica. Daarna behandelen we de taalkundige adekwaatheid van unificatie-grammatica. UG is niet alleen een handige en taalkundig aansprekende notatie voor context-vrije grammatica, maar ook een alternatief voor transformationele theorieën. Fenomenen die doorgaans in termen van transformaties beschreven worden, zoals vraagwoordzinnen en werkwoordsverplaatsing in Nederlandse hoofdzinnen, kunnen ook binnen een nontransformationele UG geanalyseerd worden. Tenslotte bespreken we een implementatie van UG in Prolog.

4.2 Unificatie van Kenmerken

Een UG maakt gebruik van matrices van kenmerken en waarden (in de Engelstalige literatuur meestal aangeduid met de term *feature-structure* of *attribute-value matrix* (AVM). Eenvoudige voorbeelden daarvan hebben we in de inleiding gegeven. Taalkundigen werken vaak met de assumptie dat waarden altijd binair (+ of -, 0 of 1) zijn, of anders worden gekozen uit een klein aantal atomaire symbolen (bijvoorbeeld *nom, acc, dat* en *gen* als waarden voor CASE of 0,1, en 2 als waarden voor BAR). Binnen de UG staat men ook toe dat de waarde van een kenmerk zelf weer een matrix is. Een voorbeeld is:

(3)
$$\begin{bmatrix} \textsf{bar} & 0 \\ \textsf{head} & \begin{bmatrix} \textsf{cat} & v \\ \textsf{vform} & fin \end{bmatrix} \\ \textsf{subj} & \begin{bmatrix} \textsf{head} & \begin{bmatrix} \textsf{cat} & n \\ \textsf{case} & nom \\ \textsf{agr} & \begin{bmatrix} \textsf{per} & 1 \\ \textsf{num} & pl \end{bmatrix} \end{bmatrix} \\ \textsf{bar} & 2 \end{bmatrix} \\ \textsf{obj} & nil \end{bmatrix}$$

Deze matrix zou bijvoorbeeld een werkwoord kunnen beschrijven. De kenmerken BAR, HEAD, SUBJ en OBJ geven respectievelijk het *bar* of *projectie*-niveau, de hoofdkenmerken, de eigenschappen van het onderwerp (*subject*) en de eigenschappen van het lijdend voorwerp (*object*). Het kenmerk BAR heeft de atomaire waarde 0. Het gaat dus, volgens de regels van de X-bar-theorie, om een woord. Het kenmerk HEAD geeft aan wat de hoofdkenmerken van het woord zijn. Hoofdkenmerken zijn die kenmerken die, weer in overeenstemming met de X-bar theorie, identiek zijn voor de moeder van een complexe constituent en de dochter die het hoofd vormt van die constituent. In dit geval gaat het om een woord met categorie *v(erb)*.[1]

[1] Eventueel zou het kenmerk CAT kunnen worden vervangen door de kenmerken N en V, zoals in het voorbeeld in de inleiding.

De werkwoordsvorm wordt gegeven door het kenmerk VFORM. In dit geval is de waarde *finite*. Het gaat dus om een persoonsvorm. Het onderwerp SUBJ is in dit geval een NP ([CAT N, BAR 2], die bovendien de *agreement*-kenmerken eerste persoon en meervoud heeft. Het onderwerp is dus waarschijnlijk wij of we. De waarde van OBJ tenslotte is domweg *nil*. Dit betekent dat het werkwoord geen lijdend voorwerp toestaat. Het werkwoord in kwestie zou dus de (eerste persoon) meervoudsvorm van een intransitief werkwoord, bijvoorbeeld *werken* in de zin wij werken of *liepen* in de zin we liepen.

Naast complexe waarden voor kenmerken maakt UG ook gebruik van gedeelde waarden (*reentrancies*). Wanneer twee kenmerken een waarde delen betekent dit dat ze beide dezelfde waarde hebben, en bovendien, dat iedere verandering van die waarde ook voor beide kenmerken geldt. Gedeelde waarden geeft men meestal aan met een index: $\boxed{1}$. De bovenstaande beschrijving van een werkwoord kunnen we bijvoorbeeld uitbreiden met een kenmerk SEM, dat de betekenis (semantiek) representeert. In dit geval nemen we aan dat het woord bijvoorbeeld een vorm van lopen is. De betekenis is een éénplaatsige relatie `lopen`. De waarde van het argument van deze relatie, komt overeen met de waarde van de semantiek van het onderwerp. Deze overeenstemming geven we weer door middel van de gedeelde waarde $\boxed{1}$. Het idee achter deze analyse is dat wanneer lopen met een onderwerp combineert, dit een waarde voor de semantiek van SUBJ op zal leveren, en daarmee voor ARG1.

(4)
$$\begin{bmatrix} \textsf{bar} & 0 \\ \textsf{head} & \begin{bmatrix} \textsf{cat} & v \\ \textsf{vform} & fin \end{bmatrix} \\ \textsf{subj} & \begin{bmatrix} \textsf{head} & \begin{bmatrix} \textsf{cat} & n \\ \textsf{case} & nom \\ \textsf{agr} & \begin{bmatrix} \textsf{per} & 1 \\ \textsf{num} & pl \end{bmatrix} \end{bmatrix} \\ \textsf{bar} & 2 \\ \textsf{sem} & \boxed{1} \end{bmatrix} \\ \textsf{obj} & nil \\ \textsf{sem} & \begin{bmatrix} \textsf{pred} & lopen \\ \textsf{arg1} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

De belangrijkste operatie in een UG is *unificatie*. Unificatie van matrixen (*feature-unification*) is vergelijkbaar met unificatie van termen in Prolog. Net als termen, kunnen matrices ordenen in termen van *subsumptie*. Een matrix M subsumeert M' als M' alle informatie bevat die M bevat. Voorbeelden zijn :

(5) a. $\begin{bmatrix} \text{bar} & 0 \\ \text{head} & \begin{bmatrix} \text{cat} & v \end{bmatrix} \end{bmatrix}$ subsumeert $\begin{bmatrix} \text{bar} & 0 \\ \text{head} & \begin{bmatrix} \text{cat} & v \\ \text{vform} & \textit{fin} \end{bmatrix} \end{bmatrix}$

b. $\begin{bmatrix} \text{bar} & 0 \\ \text{head} & \begin{bmatrix} \text{cat} & v \end{bmatrix} \end{bmatrix}$ subsumeert $\begin{bmatrix} \text{bar} & 0 \\ \text{head} & \begin{bmatrix} \text{cat} & v \end{bmatrix} \\ \text{obj} & \textit{nil} \end{bmatrix}$

Niet alle matrices staan in een subsumptie-relatie tot elkaar (subsumptie is dus een partiële ordening op matrices). De volgende twee matrices staan bijvoorbeeld niet in een subsumptie-relatie tot elkaar:

(6) $\begin{bmatrix} \text{bar} & 0 \\ \text{head} & \begin{bmatrix} \text{cat} & v \\ \text{vform} & \textit{fin} \end{bmatrix} \end{bmatrix}$ $\begin{bmatrix} \text{bar} & 0 \\ \text{head} & \begin{bmatrix} \text{cat} & v \end{bmatrix} \\ \text{obj} & \textit{nil} \end{bmatrix}$

Een matrix die een gedeelde waarde bevat, subsumeert alleen matrices waarin dezelfde waarde gedeeld wordt:

(7) $\begin{bmatrix} \text{subj} & \begin{bmatrix} \text{sem} & \boxed{1} \end{bmatrix} \\ \text{sem} & \begin{bmatrix} \text{pred} & \textit{lopen} \\ \text{arg1} & \boxed{1} \end{bmatrix} \end{bmatrix}$ subsumeert $\begin{bmatrix} \text{head} & \begin{bmatrix} \text{cat} & v \end{bmatrix} \\ \text{subj} & \begin{bmatrix} \text{sem} & \boxed{1}\textit{jan} \end{bmatrix} \\ \text{sem} & \begin{bmatrix} \text{pred} & \textit{lopen} \\ \text{arg1} & \boxed{1} \end{bmatrix} \end{bmatrix}$

Alle informatie in de linker matrix ook aanwezig in de rechter matrix. In de tweede matrix wordt *jan* overigens maar eenmaal als waarde van $\boxed{1}$ vermeld. Dit is een notationele conventie, bedoeld om het herhalen van identieke informatie te voorkomen (hetgeen de leesbaarheid en, met name, de omvang van grotere voorbeelden ten goede komt). We hadden dus net zo goed het volgende kunnen schrijven:

(8) $\begin{bmatrix} \text{head} & \begin{bmatrix} \text{cat} & v \end{bmatrix} \\ \text{subj} & \begin{bmatrix} \text{sem} & \boxed{1}\textit{jan} \end{bmatrix} \\ \text{sem} & \begin{bmatrix} \text{pred} & \textit{lopen} \\ \text{arg1} & \boxed{1}\textit{jan} \end{bmatrix} \end{bmatrix}$ of $\begin{bmatrix} \text{head} & \begin{bmatrix} \text{cat} & v \end{bmatrix} \\ \text{subj} & \begin{bmatrix} \text{sem} & \boxed{1} \end{bmatrix} \\ \text{sem} & \begin{bmatrix} \text{pred} & \textit{lopen} \\ \text{arg1} & \boxed{1}\textit{jan} \end{bmatrix} \end{bmatrix}$

In het volgende voorbeeld is geen sprake van een subsumptie-relatie, omdat de informatie dat de waardes van HEAD SEM en SEM identiek zijn niet aanwezig is in de rechter matrix.

(9) $\begin{bmatrix} \text{head} & \begin{bmatrix} \text{cat} & v \\ \text{sem} & \boxed{1} \end{bmatrix} \\ \text{sem} & \boxed{1} \end{bmatrix}$ subsumeert niet $\begin{bmatrix} \text{head} & \begin{bmatrix} \text{cat} & v \\ \text{sem} & \begin{bmatrix} \text{pred} & \textit{lopen} \end{bmatrix} \end{bmatrix} \\ \text{sem} & \begin{bmatrix} \text{pred} & \textit{lopen} \end{bmatrix} \end{bmatrix}$

Unificatie laat zich definieren in termen van subsumptie. De unificatie (*most general unifier*) van twee matrices van kenmerken M_1 en M_2 is de meest algemene matrix die wordt gesubsumeerd door M_1 en M_2. Intuitief betekent dit dat de unificatie van twee matrices een matrix oplevert die precies alle informatie bevat die in ofwel het ene, ofwel de andere matrix aanwezig is, en niets meer dan dat. Unificatie faalt wanneer er geen matrix bestaat die door beide argumenten van de operatie wordt gesubsumeerd. Hieronder volgen enkele voorbeelden, waarbij we de operator $\sqcup$ gebruiken om unificatie uit te drukken.

(10) $\begin{bmatrix} \text{agr} & \begin{bmatrix} \text{per} & 3 \end{bmatrix} \\ \text{case} & \textit{nom} \end{bmatrix} \sqcup \begin{bmatrix} \text{agr} & \begin{bmatrix} \text{num} & \textit{pl} \end{bmatrix} \\ \text{bar} & 0 \end{bmatrix} = \begin{bmatrix} \text{agr} & \begin{bmatrix} \text{num} & \textit{pl} \\ \text{per} & 3 \end{bmatrix} \\ \text{case} & \textit{nom} \\ \text{bar} & 0 \end{bmatrix}$

(11) $\begin{bmatrix} \text{head} & \begin{bmatrix} \text{cat} & v \\ \text{sem} & \boxed{1} \end{bmatrix} \\ \text{sem} & \boxed{1} \end{bmatrix} \sqcup \begin{bmatrix} \text{sem} & \begin{bmatrix} \text{pred} & \textit{lopen} \\ \text{arg1} & \textit{jan} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{head} & \begin{bmatrix} \text{cat} & v \\ \text{sem} & \boxed{1}\begin{bmatrix} \text{pred} & \textit{lopen} \\ \text{arg1} & \textit{jan} \end{bmatrix} \end{bmatrix} \\ \text{sem} & \boxed{1} \end{bmatrix}$

De unificatie van de volgende twee matrices mislukt:

(12) $\begin{bmatrix} \text{agr} & \begin{bmatrix} \text{per} & 3 \\ \text{num} & \textit{sg} \end{bmatrix} \\ \text{case} & \textit{nom} \end{bmatrix} \not\sqcup \begin{bmatrix} \text{agr} & \begin{bmatrix} \text{num} & \textit{pl} \end{bmatrix} \\ \text{bar} & 0 \end{bmatrix}$

Een UG drukt alle taalkundig relevante informatie in de vorm van kenmerken en waarden uit. Een niet-triviale grammatica zal daarom al snel enige tientallen verschillende kenmerken gebruiken. Om het overzicht niet te verliezen, is het handig verschillende groepen kenmerken samen te brengen. In de bovenstaande voorbeelden gebruikten we bijvoorbeeld SUBJ als het kenmerk dat alle informatie over het subject bevat, en SEM als kenmerk dat alle semantische informatie bevat. Het samenbrengen van groepen kenmerken onder een bepaald label heeft nog een betekenis.

$$\begin{bmatrix}\text{cat} & s\\ \text{head} & \boxed{1}\end{bmatrix} \rightarrow \begin{bmatrix}\text{cat} & np\\ \text{head} & \begin{bmatrix}\text{agr} & \boxed{2}\end{bmatrix}\\ \text{case} & nom\end{bmatrix} \begin{bmatrix}\text{cat} & vp\\ \text{head} & \boxed{1}\begin{bmatrix}\text{agr} & \boxed{2}\\ \text{vform} & fin\end{bmatrix}\end{bmatrix}$$

$$\begin{bmatrix}\text{cat} & vp\\ \text{head} & \boxed{1}\end{bmatrix} \rightarrow \begin{bmatrix}\text{cat} & v\\ \text{head} & \boxed{1}\\ \text{comp} & nil\end{bmatrix}$$

$$\begin{bmatrix}\text{cat} & vp\\ \text{head} & \boxed{1}\end{bmatrix} \rightarrow \begin{bmatrix}\text{cat} & np\\ \text{case} & acc\end{bmatrix} \begin{bmatrix}\text{cat} & v\\ \text{head} & \boxed{1}\\ \text{comp} & np\end{bmatrix}$$

Figuur 4.2: Regels in een unificatie-grammatica.

Net als in de fonologische voorbeelden waarmee we dit hoofstuk begonnen, verwachten we dat de juiste combinatie van kenmerken het mogelijk maakt om generalisaties eenvoudig en direct uit te drukken. Het kenmerk HEAD bevat bijvoorbeeld zowel informatie over de categorie van een constituent als informatie over zaken als werkwoordsvorm of naamval. Op zich is er geen reden om precies deze kenmerken samen te brengen. De kenmerken onder HEAD zijn echter precies die kenmerken die altijd identiek zijn voor de moeder en de dochter die het hoofd is van een complexe constituent. Door deze kenmerken te bundelen onder HEAD kunnen we deze generalisatie aan een regel toevoegen als de conditie dat de waarde van HEAD op de moeder en het hoofd identiek moet zijn.

4.3 Een eenvoudig fragment

Een UG bestaat uit een verzameling regels en een woordenboek. Regels hebben de vorm `Moeder` → `Dochters`, waarbij de `Moeder` een feature-structure is, en de dochters een lijst van feature-structuren. Drie voorbeelden van zulke regels staan in figuur 4.2.

Dit eerste regel in figuur 4.2 is de bekende regel S → NP VP. Aan de regel is de informatie toegevoegd dat de VP finiet moet zijn, dat de NP de nominatieve (eerste) naamval moet hebben, en dat er overeenstemming moet zijn in de kenmerken voor persoon en getal (gebundeld onder AGR) tussen VP en NP. Bovendien zin heeft het kenmerk HEAD dezelfde waarde op S en VP.

De twee volgende regels in figuur 4.2 corresponderen met de CFG-regels VP → V en VP → NP V. Wederom geldt dat HEAD dezelfde waarde heeft op moeder en (hoofd-) dochter. Bovendien moet de dochter een *intransitief*, resp. *transitief* werkwoord zijn. Dit drukken we uit door het kenmerk COMP (*complement*) de waarde *nil*, resp. *np* te geven.

$$\text{bert} \mapsto \begin{bmatrix}\text{cat} & np\\ \text{head} & \begin{bmatrix}\text{agr} & \begin{bmatrix}\text{per} & 3\\ \text{num} & sg\end{bmatrix}\end{bmatrix}\end{bmatrix} \quad \text{duiven} \mapsto \begin{bmatrix}\text{cat} & np\\ \text{head} & \begin{bmatrix}\text{agr} & \begin{bmatrix}\text{per} & 3\\ \text{num} & pl\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

$$\text{lacht} \mapsto \begin{bmatrix}\text{cat} & v\\ \text{head} & \begin{bmatrix}\text{agr} & \begin{bmatrix}\text{per} & 3\\ \text{num} & sg\end{bmatrix}\\ \text{vform} & fin\end{bmatrix}\\ \text{comp} & nil\end{bmatrix} \quad \text{bezit} \mapsto \begin{bmatrix}\text{cat} & v\\ \text{head} & \begin{bmatrix}\text{agr} & \begin{bmatrix}\text{per} & 3\\ \text{num} & sg\end{bmatrix}\\ \text{vform} & fin\end{bmatrix}\\ \text{comp} & np\end{bmatrix}$$

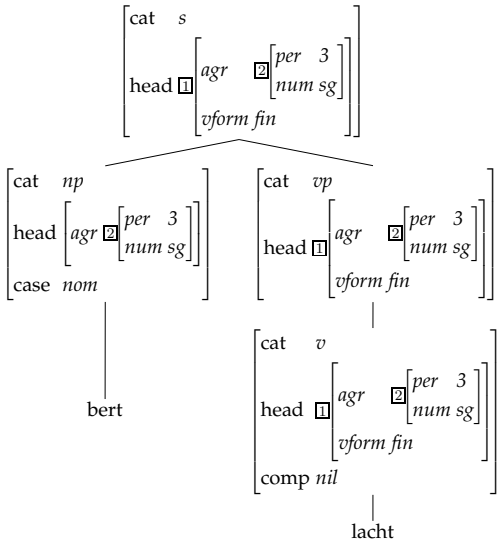
Figuur 4.3: Fragment van het woordenboek.

Het woordenboek bestaat uit paren van een woord en een feature-structuur, die we kunnen opschrijven als `Woord` ↦ `FS`. Enkele voorbeelden worden gegeven in figuur 4.3. Twee boomstructuren die volgens dit fragment afleidbaar zijn, worden gegeven in figuur 4.4 en 4.5. Merk op dat de waarde van het kenmerk HEAD wordt gedeeld door V, VP, en S.

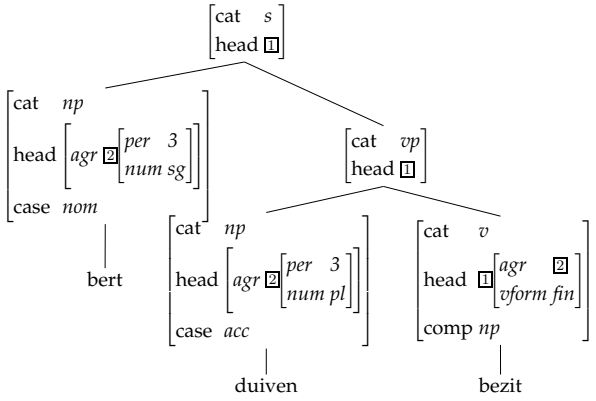
In regels worden soms waarden gedeeld tussen verschillende matrices (tussen een moeder en een dochter, of tussen dochters). Dit is mogelijk omdat we een regel zelf als een feature-structuur kunnen zien. De regel S → NP VP kunnen we bijvoorbeeld ook representeren als:

(13)
$$\begin{bmatrix}\text{moeder} & \begin{bmatrix}\text{cat} & s\\ \text{head} & \boxed{1}\end{bmatrix}\\ \text{dochters} & \begin{bmatrix}\text{first} & \begin{bmatrix}\text{cat} & np\\ \text{head} & \begin{bmatrix}\text{agr} & \boxed{2}\end{bmatrix}\\ \text{case} & nom\end{bmatrix}\\ \text{rest} & \begin{bmatrix}\text{first} & \begin{bmatrix}\text{cat} & vp\\ \text{head} & \boxed{1}\begin{bmatrix}\text{agr} & \boxed{2}\\ \text{vform} & fin\end{bmatrix}\end{bmatrix}\\ \text{rest} & nil\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

De lijst dochters van een regel is hier gecodeerd met behulp van de kenmerken FIRST en REST.



Figuur 4.4: Bert lacht



Figuur 4.5: (Ernie meent dat) Bert duiven bezit

4.4 Implementatie en notatie

Unificatie-grammatica kan zonder veel moeite in Prolog worden geïmplementeerd. Het is mogelijk feature-structuren als Prolog-termen te coderen op een manier die het mogelijk maakt de 'gewone' Prolog term-unificatie te gebruiken voor unificatie van feature-structuren.

De definitie van een UG in Prolog bevat in het algemeen drie componenten. Naast regels en een woordenboek is er een definitie nodig van de kenmerken die in de grammatica gebruikt worden. Het laatste is nodig om de vertaling van feature-structuren in Prolog termen te ondersteunen. In deze sectie bespreken we de notatie en onderliggende implementatie van het Hdrug systeem,[2] een ontwikkelomgeving voor unificatie-grammatica's.

4.4.1 Definitie van feature-structuren

Voor de vertaling van feature-structuren naar termen is het noodzakelijk dat alle gebruikte features, en combinaties van features, bekend zijn. Voor een gegeven grammatica, is het meestal niet moeilijk zo'n definitie te maken.

Het idee is dat iedere feature-structuur die in de grammatica wordt gebruikt, van een bepaald *type* is. Het feature AGR, bijvoorbeeld, heeft altijd een feature-structuur als waarde die iets over de *agreement*-eigenschappen van een woord of zinsdeel zegt. De enige features in het fragment uit sectie 4.3. die in *agreement* een rol spelen, zijn PER en NUM. Dit betekent dat we een type *agr* kunnen definieren, waarvoor de features PER en NUM gedefinieerd zijn. In figuur 4.6 gebeurt dit door het drie-plaatsige predicaat `type`. Het eerste argument is de naam van het type, het tweede argument noemt alle subtypes van dit type,[3] en het derde argument noemt alle features die voor dit type gedefinieerd zijn. In de eenvoudige grammatica die we hier als voorbeeld gebruiken, wordt geen gebruik gemaakt van subtypes, en is het tweede argument van `type` dus altijd de lege lijst.

Het effect van de definitie van het type *agr* in figuur 4.6 is dat het systeem iedere feature-structuur van dit type representeert als een prolog-term van de vorm `agr(Per, Num, Index)`. De variabelen `Per` en `Num` worden hier gebruikt om de waarden van de features PER en NUM te coderen. De laatste variabele van iedere feature-term is een index-variabele, die wordt gebruikt om *reentrancy* van feature-structuren te coderen.

De twee overige type-definities zijn het type *head*, voor de waarde van het kenmerk HEAD, en *sign* voor feature structuren die corresponderen met een heel woord of een heel zinsdeel. Voor zulke feature-structuren zijn altijd de features CAT, HEAD en COMP gedefinieerd. De naam *sign* is gebruikelijk in HPSG, en is ontleed aan het werk van de Franse taalkundige de Saussure (i.e. diens *signe*).

[2] `www.let.rug.nl/~vannoord/Hdrug`

[3] Voor een subtype S van een type T zijn dezelfde features gedefinieerd als voor T, maar kunnen bovendien nog extra features gedefinieerd zijn.

```
top([sign,head,agr]).

type(head,[],[agr,case,vform]).
type(agr, [],[per,num]).
type(sign,[],[cat,head,comp]).
```

Figuur 4.6: Type-definities voor het grammatica fragment.

Het predicaat `top/1`, tenslotte, definieert de verzameling (als een lijst) van types die in de grammatica gebruikt worden.

De type definitie ondersteunt de compilatie van (beschrijvingen van) feature-structuren naar Prolog-termen. De feature-structuur in (14-a) wordt bijvoorbeeld vertaald naar de Prolog-term in (14-b).

(14) a.
$$\begin{bmatrix} \textsf{cat} & v \\ \textsf{head} & \begin{bmatrix} \textsf{agr} & \begin{bmatrix} \textsf{per} & 3 \\ \textsf{num} & sg \end{bmatrix} \\ \textsf{vform} & fin \end{bmatrix} \\ \textsf{comp} & nil \end{bmatrix}$$

b. `sign(v,head(agr(3,sg,_),_,fin,_),nil,_)`

De type-definitie legt ook restricties op aan de grammatica. Feature-structuren die niet voldoen aan de definitie kunnen in de grammatica niet gebruikt worden. Voorbeeld (15-a) is niet toegestaan, omdat er een kenmerk in gebruikt wordt (BAR), die in de type-definitie niet terug te vinden is. Een feature structuur waarin HEAD en AGR naast elkaar voorkomen, zoals in (15-b) is niet toegestaan omdat er geen type te vinden is waar deze structuur mee correspondeert.

(15) a.
$$\begin{bmatrix} \textsf{head} & \begin{bmatrix} \textsf{case} & nom \end{bmatrix} \\ \textsf{bar} & 2 \end{bmatrix}$$

b.
$$\begin{bmatrix} \textsf{head} & \begin{bmatrix} \textsf{case} & nom \end{bmatrix} \\ \textsf{agr} & \begin{bmatrix} \textsf{num} & 3 \end{bmatrix} \end{bmatrix}$$

4.4.2 Beschrijving van feature-structuren

In het fragment in sectie 4.3. hebben we regels en woordenboek-items gedefinieerd door feature-structuren als matrices op te schrijven. Deze notatie is vooral nuttig omdat ze gemakkelijk leesbaar is. Voor een implementatie is deze notatie echter minder geschikt. In implementaties kiest men er meestal voor om feature-structuren door middel van vergelijkingen te definiëren.

Wanneer `V` de feature-structuur voor `lacht` is, kunnen we deze, in het Hdrug-systeem, bijvoorbeeld als volgt definiëren:

(16)
```
woord(lacht, V) :-
    V:cat <=> v,
    V:head:agr:per <=> 3,
    V:head:agr:num <=> sg,
    V:head:vform <=> fin,
    V:comp <=> nil.
```

Links van de pijl staat hier steeds een *pad*, een reeks van kenmerken (gescheiden door ':'). Het pad `V:head:agr:num` duidt bijvoorbeeld de waarde van het kenmerk NUM onder AGR onder HEAD in de feature-structuur `V` aan.

Gedeelde waarden geven we weer door een pad links en rechts van de pijl. Wanneer `S, NP` en `VP` de feature-structuren voor de corresponderende knopen in de regel S → NP VP zijn, kunnen we deze feature structuren bijvoorbeeld als volgt definiëren:

(17)
```
regel(s_np_vp,S,[NP,VP]) :-
    S:cat <=> s,
    S:head <=> VP:head,
    NP:cat <=> np,
    NP:case <=> nom,
    NP:head:agr <=> VP:head:agr,
    VP:cat <=> vp,
    VP:head:vform <=> fin,
    VP:comp <=> nil.
```

De tweede regel zegt dat de waarde voor HEAD binnen `S` en `VP` identiek is, en de vijfde regel unificeert de waarden voor HEAD:AGR binnen `NP` en `VP`.

In figuur 4.7 en 4.8 wordt een klein fragment van een unificatie-grammatica in Hdrug-notatie gegeven. Regels worden gedefinieerd als `regel(Naam, Moeder, Dochters)`. `Naam` is hier de naam van de regel; deze is alleen voor de leesbaarheid van de grammatica van belang. Het lexicon wordt gedefinieerd door `woord(Woord, Feature-Structuur)`. Het tweede argument heeft hier dezelfde vorm en betekenis als de knopen in de regels.

```
regel(s_np_vp, S,[ NP, VP ]) :-
    S:cat         <=> s,
    VP:cat        <=> vp,
    S:head        <=> VP:head,
    NP:cat        <=> np,
    NP:head:case  <=> nom,
    NP:head:agr   <=> VP:head:agr,
    VP:head:vform <=> fin.

regel(vp_v, VP  ,[ V ]) :-
    VP:cat  <=> vp,
    V:cat   <=> v,
    VP:head <=> V:head,
    V:comp  <=> nil.

regel(vp_np_v, VP  ,[ NP, V ]) :-
    VP:cat       <=> vp,
    V:cat        <=> v,
    VP:head      <=> V:head,
    NP:cat       <=> np,
    NP:head:case <=> acc,
    V:comp       <=> np.
```

Figuur 4.7: Implementatie van een UG-fragment in Hdrug (I).


```
woord(bert, NP) :-
    NP:cat          <=> np,
    NP:head:agr:per <=> 3,
    NP:head:agr:num <=> sg.

woord(duiven, NP) :-
    NP:cat          <=> np,
    NP:head:agr:per <=> 3,
    NP:head:agr:num <=> pl.

woord(lacht, V) :-
    V:cat           <=> v,
    V:head:agr:per  <=> 3,
    V:head:agr:num  <=> sg,
    V:head:vform    <=> fin,
    V:comp          <=> nil.

woord(bezit, V) :-
    V:cat           <=> v,
    V:head:agr:num  <=> sg,
    V:head:vform    <=> fin,
    V:comp          <=> np.
```

Figuur 4.8: Implementatie van een UG-fragment in Hdrug (II).

Wanneer een regel wordt aangeroepen, worden alle vergelijkingen die bij een regel aangeroepen, hetgeen tot gevolg heeft dat alle variabelen die voor een feature-structuur staan, worden geïnstantieerd als Prolog-termen. Bijvoorbeeld:

```
?- regel(s_np_vp, S, [NP, VP]).

S  = sign(s,head(agr(A,B,C),D,fin,E),_)
NP = sign(np,head(agr(A,B,C),nom,_,_),_)
VP = sign(v,head(agr(A,B,C),D,fin,E),_)
```

4.5 Het gebruik van Macro's

Het specificeren van grotere grammaticafragmenten in bovenstaande notatie levert al vrij snel een situatie op waarin identieke of vergelijkbare informatie in verschillende regels of woordenboek-definities herhaald moet worden. Dit kan worden vermeden door informatie die vaak herhaald moet worden te definiëren als een *macro* (of *template*). In Prolog kunnen we *macro's* eenvoudigweg als extra Prolog-clauses beschouwen die een (deel van een) feature-structuur definiëren, en die kunnen worden aangeroepen in de definitie van andere feature-structuren.

Wanneer we de waarden voor persoon en getal voor een woord moeten definiëren, zijn er twee vergelijkingen nodig die iets zeggen over de paden HEAD:PER en HEAD:NUM. Dit paar vergelijkingen kunnen we ook als volgt vastleggen:

```
agr(X,Per,Num) :-
    X:head:agr:per <=> Per,
    X:head:agr:num <=> Num.
```

Het predicaat `agr/3` kunnen we vervolgens aanroepen in de definitie van een woord:

```
woord(bert, NP) :-
    NP:cat <=> np,
    agr(NP,3,sg).
```

Naast een direct practisch voordeel, heeft deze methode ook als voordeel dat ze het mogelijk maakt taalkundige generalisaties uit te drukken. In theorieën als GPSG en HPSG neemt men bijvoorbeeld een *head-feature-principle* (HFP aan. Dit principe zegt dat de HEAD-features op moeder en (hoofd-)dochter identiek zijn. In het huidige fragment laat dit principe zich als volgt coderen:

```
hfp(Moeder,Hoofd) :-
   Moeder:head <=> Hoofd:head.
```

In een regel kunnen we nu gebruik maken van dit principe:

```
regel(vp_v, VP  ,[ V ]) :-
    VP:cat <=> vp,
    V:cat  <=> v,
    hfp(VP,V),
    V:comp <=> nil.
```

In dit geval is de winst vooral taalkundig: door gebruik te maken van namen voor bepaalde stukken informatie wordt duidelijk gemaakt hoe de implementatie samenhangt met de theorie.


```
cat(X,Cat) :-
    X:cat <=> Cat.

agr(X,Per,Num) :-
    X:head:agr:per <=> Per,
    X:head:agr:num <=> Num.

case(X,Case) :-
    X:head:case <=> Case.

vform(X,Vform) :-
    X:head:vform <=> Vform.

agreement(NP,VP) :-
    NP:head:agr <=> VP:head:agr.

hfp(Moeder,Hoofd) :-
   Moeder:head <=> Hoofd:head.

head_subject_structure(Moeder,Hoofd,Subj) :-
    hfp(Moeder,Hoofd),
    case(Subj,nom),
    agreement(Subj,Hoofd),
    vform(Hoofd,fin)

head_complement_structure(Moeder,Hoofd,Comp) :-
    hfp(Moeder,Hoofd),
    Hoofd:comp <=> Comp.

regel(s_np_vp, S,[ NP, VP ]) :-
    cat(S,s), cat(NP,np), cat(VP,vp),
    head_subject_structure(S,VP,NP).

regel(vp_v, VP  ,[ V ]) :-
    cat(VP,vp), cat(V,v),
    head_complement_structure(VP,V,nil).

regel(vp_np_v, VP  ,[ NP, V ]) :-
    cat(VP,vp), cat(NP,np), cat(V,v),
    head_complement_structure(VP,V,np),
    case(NP,acc).
```

Figuur 4.9: Een fragment met macro's

```
regel(vp_v_bijzin, VP  ,[ V, Bijzin ]) :-
    cat(VP,vp), cat(V,v), cat(Bijzin,bijzin),
    head_complement_structure(VP,V,bijzin).

regel(vp_np_pp_v, VP  ,[ NP, PP, V,]) :-
    cat(VP,vp), cat(V,v), cat(NP,np), cat(PP,pp),
    head_complement_structure(VP,V,np_pp),
    case(NP,acc),
    PP:head:pform <=> aan.

regel(vp_np_np_v, VP  ,[ MeeVwp, LijdVwp, V,]) :-
    cat(VP,vp), cat(V,v), cat(MeeVwp,np), cat(LijdVwp,np),
    head_complement_structure(VP,V,np_pp),
    case(MeeVwp, acc),
    case(LijdVwp,acc).

regel(vp_pp_v, VP  ,[ PP, V ]) :-
    cat(VP,vp), cat(V,v), cat(PP,pp),
    head_complement_structure(VP, V, pp(Pform)),
    PP:head:pform <=> Pform.
```

Figuur 4.10: Meer regels voor de VP

In figuur 4.9 geven we een versie van de regels uit figuur 4.7, waarin zoveel mogelijk informatie in macros is vastgelegd. De macros `agr, case` en `vform` zijn voornamelijk afkortingen voor de paden naar één of meer features. `Agreement` en `hfp` leggen een *reentrancy* vast. De macros `head_subject_structure` en `head_complement_structure` drukken generalisaties over regels uit. De eerste geeft aan dat de combinatie van een hoofd met een subject vereist dat er agreement is, dat het subject nominatief is, en dat het hoofd een finiet werkwoord bevat. Naast de regel S → NP VP zullen we hierna nog enkele regels zien die van deze structuur gebruik maakt. Een `head_complement_structure` is de combinatie van een hoofd met een complement. In dit geval geldt altijd het *head feature principle*. Bovendien moet het kenmerk COMP op het hoofd altijd een bepaalde waarde hebben. Een randgeval van een `head_complement_structure` is de regel die een VP herschrijft als een intransitief werkwoord, dat geen complement selecteert.

4.6 Meer regels voor de VP

In figuur 4.10 geven we een aantal extra regels voor de VP (in bijzinnen). De eerste regel maakt het mogelijk VP's af te leiden waarin een (ondergeschikte) bijzin voorkomt. Bijzinnen volgen altijd op het werkwoord:

(18) (dat ernie) beweert/meent/zegt/gelooft dat bert duiven bezit

De tweede regel dient voor werkwoorden die zowel een lijdend als een meewerkend voorwerp selecteren:

(19) (dat ernie) duiven aan bert geeft/overhandigt/schenkt

Merk op dat in deze regel een nieuw HEAD-kenmerk wordt gebruikt. PFORM is een kenmerk dat de vorm van de prepositie aangeeft, die het hoofd is van een PP. De prepositie aan heeft dus als waarde voor PFORM *aan*, *tegen* heeft als PFORM *tegen*, etc. Het introduceren van een nieuw kenmerk betekent overigens dat de definitie van het type *head* moet worden aangepast:

```
type(head,[],[agr,case,vform,pform]).
```

Werkwoorden die een lijdend en meewerkend voorwerp selecteren, kunnen ook altijd gebruikt worden in VP's waarin het meewerkend voorwerp aan het lijdend voorwerp vooraf gaat. In dat geval vervalt het voorzetzel *aan* (meestal), en is het meewerkend voorwerp dus een NP. Merk op dat in dit geval de waarde van COMP weer np_pp is. Dit betekent dat de werkwoorden die in regel `vp_np_pp_v` gebruikt kunnen worden ook in de regel `vp_np_np_v` gebruikt kunnen worden.

De regel `vp_pp_v` dient voor werkwoorden die een voorzetselvoorwerp selecteren.

(20) a. (dat bert) aan duiven denkt
b. (dat bert) op de duiven wacht
c. (dat bert) van duiven droomt

Een werkwoord dat een voorzetselvoorwerp selecteert, legt ook restricties op aan het voorzetsel dat gebruikt wordt. Dit kunnen we implementeren door gebruik te maken van het kenmerk PFORM, dat codeert welk voorzetsel het hoofd van een PP is. Omdat er in dit geval niet één complementstype is, maar eigenlijk een ander type voor ieder voorzetselvoorwerp, is de waarde van COMP in dit geval een term `pp(Pform)`, waar het argument `Pform` overeenstemt met de waarde van PFORM op de PP.

```
regel(v_v_v, VC, [v/Hlpww, v/Hfdww]) :-
    cat(VC,vc), cat(Hlpww,aux), cat(Hfdww,v),
    head_complement_structure(V,Hlpww,v(Vform)),
    vform(Hfdww,Vform),
    VC:comp <=> Hfdww:comp.
```

Figuur 4.11: Werkwoordsclusters

4.7 Werkwoordsclusters

De regel in figuur 4.11 is geen VP regel, maar een regel die een werkwoordscluster (*verb cluster*) vormt. De regel combineert twee werkwoorden tot één constituent van categorie V. Deze regel speelt een rol in zinnen als:

(21) a. (dat bert) moet lachen
b. (dat bert) heeft gezegd dat ernie duiven bezit

De reeksen moet lachen en heeft gezegd vormen in dit geval een cluster. Dit betekent o.a. dat er geen andere constituenten tussen deze elementen geplaatst kunnen worden. Het hoofd van een werkwoordcluster is het hulpwerkwoord. Een hulpwerkwoord is bijvoorbeeld een werkwoord van modaliteit, zoals moeten of willen, of een hulpwerkwoord van tijd zoals hebben of zijn. De modale werkwoorden nemen een infinitief (zoals lachen of bezitten) als argument. De hulpwerkwoorden van tijd nemen een voltooid deelwoord als argument. Dit verschil wordt geïmplementeerd door COMP de waarde `v(Vform)` te geven, waar `Vform` overeenkomt met de waarde van VFORM op het (geselecteerde) hoofdwerkwoord.[4]

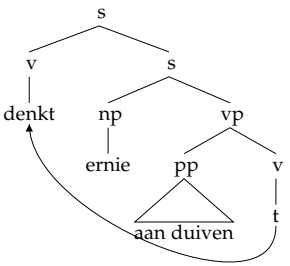
[4]Merk op dat in sectie 2.3.3 een DCG is besproken voor dit verschijnsel. De oplossing die hier wordt gegeven integreert de selectie-eigenschappen van hulpwerkwoorden met de selectie-eigenschappen van werkwoorden in het algemeen.

De regel voor werkwoordsclusters heeft nog een bijzonder aspect. De waarde van COMP op de moeder is in dit geval identiek aan de waarde van COMP op de infinitief, c.q. het deelwoord. Dit betekent dat werkwoordsclusters in precies dezelfde VP-regels kunnen worden gebruikt als de infinitief of het deelwoord:

(22) a. dat bert aan duiven denkt
b. dat bert aan duiven heeft gedacht
c. dat bert ernie duiven geeft
d. dat bert ernie duiven wil geven

4.8 Hoofd- en bijzinsvolgorde

In het voorafgaande hebben we ons vooral gericht op kwesties die met de morfologische vorm van woorden en constituenten te maken hebben, en met selectie van complementen. Deze verschijnselen laten zich in het algemeen goed beschrijven met een herschrijfgrammatica. Het gebruik van kenmerken en unificatie is hier vooral nuttig omdat het leidt tot elegante grammatica's, waarin voor de hand liggende generalisaties kunnen worden uitgedrukt. In deze en de volgende sectie concentreren we ons op verschijnselen die gewoonlijk in termen van transformaties beschreven worden. In deze sectie behandelen we de UG-variant van de transformatie werkwoordsverplaatsing, die een rol speelt in de afleiding van hoofdzinnen. De regels die we hiervoor geven zijn een variant van de DCG-regels uit sectie 2.3.3. In de volgende sectie behandelen we de UG-variant van vraagwoord- of WH-verplaatsing. Deze regel speelt een rol in vraagzinnen.



Figuur 4.12: Werkwoordsverplaatsing in transformationele grammatica.

In de transformationele grammatica is het gebruikelijk Nederlandse hoofdzinnen te beschouwen als zinnen die zijn afgeleid van bijzinnen door middel van de transformatie *werkwoordverplaatsing* (*v-movement*). Deze regel verplaatst een (finiet) werkwoord uit de VP naar de eerste positie in de zin. De zin denkt ernie aan duiven zou dan bijvoorbeeld als worden afgeleid als in figuur 4.12.

Deze transformatie wordt gemotiveerd met het argument dat op deze manier de overeenkomsten tussen de volgorde in hoofd- en bijzinnen duidelijk wordt gemaakt. De volgorde binnen de VP is in beide gevallen identiek, met uitzondering van het finiete werkwoord, dat in hoofdzinnen altijd een positie voor in de zin inneemt. Dat de bijzinsvolgorde meer 'basaal' is dan de hoofdzinsvolgorde volgt uit zinnen als:

(23) a. dat bert ernie wil opbellen
b. belt bert ernie op
c. dat ernie niet uitslapen kan
d. slaapt ernie zondags uit

Werkwoorden zoals opbellen of uitslapen bevatten een zogenaamd *scheidbaar prefix* of *partikel*, een voorvoegsel dat soms wel en soms niet deel uitmaakt van het werkwoord. In hoofdzinnen staat dit partikel altijd achteraan. Dit maakt aannemelijk dat de volgorde in bijzinnen eigenlijk de 'onderliggende' volgorde is, en dat de hoofdzinsvolgorde daar met een transformatie uit is afgeleid.


```
vtrace(V,YesNo) :-
        V:head:vgap:vtrace <=> YesNo.

verb_fronting(V,VP) :-
        vtrace(V,no),
        vtrace(VP,yes),
        VP:head:vgap:vcomp  <=> V:comp.

regel(vraagzin, S, [V, np/NP, vp/VP]) :-
       cat(S,s), cat(V,v), cat(NP,np), cat(VP,vp),
       head_subject_structure(S,V,NP),
       verb_fronting(V,VP).

regel(hoofdzin, S, [NP, V, VP]) :-
       cat(S,s), cat(V,v), cat(NP,np), cat(VP,vp),
       head_subject_structure(S,V,NP),
       verb_fronting(V,VP).

regel(vgap, V, [] ) :-
       cat(V,v),
       vform(V,fin),
       vtrace(V,yes),
       V:head:vgap:vcomp <=> V:comp.

woord(denkt,V) :-
       cat(V,v),
       vtrace(V,no),
       vform(V,fin),
       agr(V,3,sg),
       V:comp <=> nil.
```

Figuur 4.13: Hoofd- en bijzinsvolgorde met VGAP.

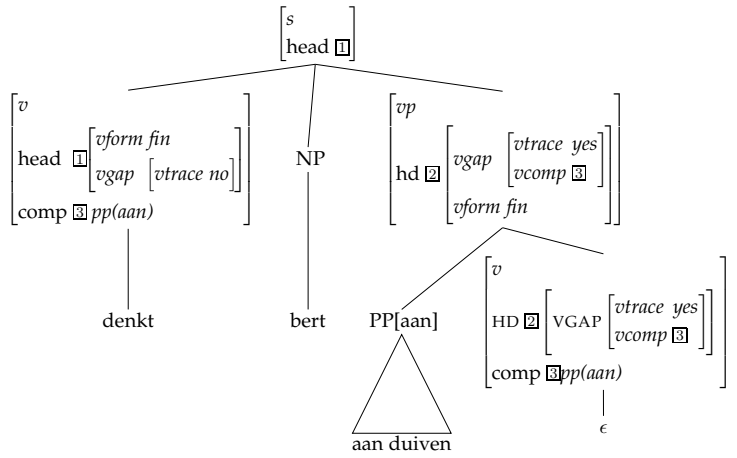
In UG wordt geen gebruik gemaakt van een transformatie om dit verband af te leiden. In plaats daarvan kan de relatie tussen het vooropgeplaatse werkwoord en de lege positie binnen de VP door middel van een kenmerk worden uitgedrukt. Hiervoor gebruiken we het kenmerk VTRACE. Daarnaast zal blijken dat informatie over het type complement(-en) waarmee het werkwoord in de VP kan combineren moet worden gedeeld tussen het voorop geplaatste werkwoord en het verbale 'gat' in de VP. Hiervoor introduceren we het kenmerk VCOMP. Beide kenmerken bundelen we onder het kenmerk VGAP. Het kenmerk VGAP is een HEAD-kenmerk.

Figuur 4.13 geeft een fragment van een grammatica met werkwoordsverplaatsing. De `vraagzin`-regel geldt voor zinnen als denkt ernie aan duiven en de `hoofdzin`-regel geldt voor zinnen als ernie denkt aan duiven. Merk op dat deze regels drie dochters hebben. Dit betekent dat de tweede S-knoop uit de afleiding in figuur 4.12 wordt geëlimineerd.

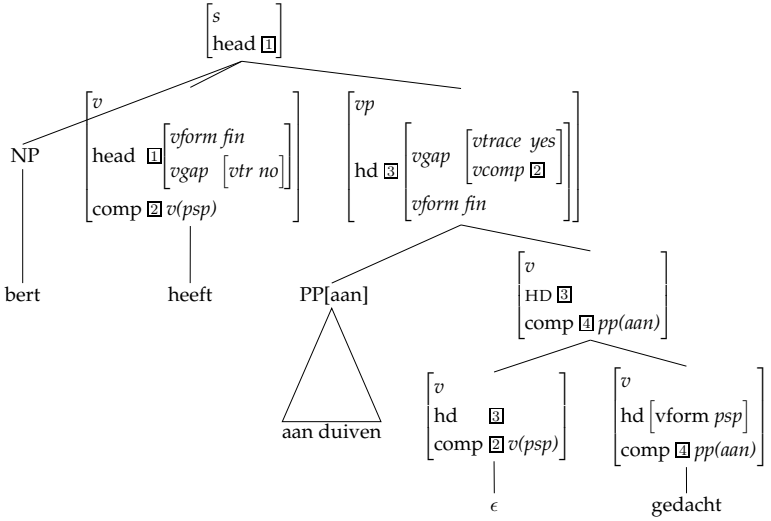
In beide regels wordt een 'voorop geplaatst' werkwoord gebruikt. De condities die hiervoor gelden staan in de macro `verbal_fronting`. In de eerste plaats moet gelden dat het voorop geplaatste werkwoord zelf geen *verbal trace* is. Dit geven we aan door aan VTRACE de waarde *no* toe te kennen. In de tweede plaats moet de VP wel een *verbal trace* bevatten. Daarnaast moet er nog een stukje informatie aan de VP moet worden doorgegeven.

Het voorop geplaatste werkwoord kan alleen voorkomen met een bepaald type VP. Meer in het bijzonder geldt dat de COMP-waarde van V bepaalt welke complementen er in de VP kunnen voorkomen. Die informatie wordt aan de VP doorgegeven via het kenmerk VCOMP.

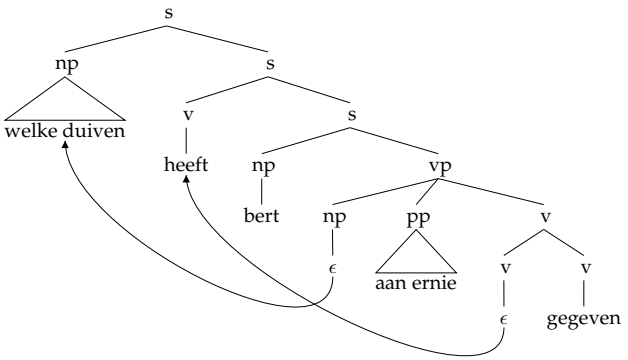
Omdat VGAP een HEAD-kenmerk is, wordt informatie over VGAP altijd doorgegeven aan het hoofd van de VP. Wanneer het werkwoord voorop geplaatst is, is het hoofd van de VP een *verbal trace*. In UG introduceren we een leeg element als een regel met een lege lijst dochters (d.w.z. als een ϵ-regel). De regel `vgap` geeft aan dat een *verbal gap* of *trace* altijd een finiet werkwoord is (alleen finiete werkwoorden kunnen verplaatst worden). Bovendien geldt dat de waarde van VCOMP identiek is aan de waarde van COMP. Omdat de waarde van VCOMP gelijk is aan de COMP-waarde van het voorop geplaatste werkwoord, betekent dit dat de COMP-waarde van het (lege) hoofd van de VP gelijk wordt gemaakt aan de COMP waarde van het voorop geplaatste werkwoord. Dit garandeert dat een voorop geplaatst werkwoord alleen met een 'passende' VP voorkomt.



Figuur 4.14: denkt bert aan duiven



Figuur 4.15: bert heeft aan duiven gedacht



Figuur 4.16: Vraagwoordverplaatsing in transformationele grammatica.

In figuur 4.14 en 4.15 staan twee voorbeelden van afleidingen. (Categorieën staan hier cursief in de linkerbovenhoek vermeld.)

4.9 Vraagwoord-zinnen

In de vorige sectie hebben we reeds een analyse gegeven van ja/nee-vragen. Vraagwoord-zinnen (WH-*questions*) zijn zinnen die beginnen met een NP waarin een vraagwoord voorkomt:

(24) a. wat bezit bert
b. welke duiven heeft bert aan ernie gegeven
c. wie zegt bert dat ernie gezien heeft

De transformationele analyse van deze zinnen maakt gebruik van een transformatie die constituenten met daarin een vraagwoord voorop plaatst (WH-*movement*). Een voorbeeld is te vinden in figuur 4.16.

De verplaatsing van vraagwoord-constituenten kan in UG geanalyseerd worden op een manier die grote overeenkomsten vertoont met de analyse van werkwoordsverplaatsing in de vorige sectie. Het belangrijkste verschil tussen beide verschijnselen is dat werkwoordsverplaatsing een lokaal verschijnsel is: het gaat altijd om de relatie tussen een voorop geplaatst werkwoord en een *gap* dat de positie van het hoofd van de VP inneemt. Vraagwoord-verplaatsing is daarentegen een niet-lokaal verschijnsel: de voorop geplaatste constituent kan corresponderen met een complement in de VP, maar ook met een constituent in een bijzin. Men spreekt daarom ook wel van een *unbounded dependency* tussen het vraagwoord en het corresponderende gat.

We introduceren het kenmerk WHTRACE om te coderen of er een vraagwoord-gat aanwezig is in een constituent. Het kenmerk WHHEAD dient om de waarde van de HEAD-kenmerken op het verplaatste element en het corresponderende *trace* te unificeren. Beide kenmerken worden gebundeld onder WHGAP. In tegenstelling tot VGAP is WHGAP geen HEAD-kenmerk.

Enkele regels voor een fragment met vraagwoord-verplaatsing zijn gegeven in figuur 4.17 en 4.18. De `whvraag`-regel is een variant van de `vraagzin`-regel uit fragment 4.13. In deze regel wordt een vraagwoord-constituent, een (voorop geplaatst) werkwoord, een onderwerp, en een VP geïntroduceerd. Omdat vier dochters in één regel worden geïntroduceerd, zijn de tussenliggende S-knopen uit de transformationele afleiding in figuur 4.16 overbodig.

De macro `wh_fronting` legt de relatie tussen de voorop geplaatste constituent en de VP waar het corresponderende gat zich bevindt. In dit geval unificeren we de HEAD-kenmerken van de vraagwoord-constituent met WHHEAD. Een *wh-trace*, oftewel een 'gat' dat correspondeert met een voorop geplaatste vraagwoord-constituent, unificeert WHHEAD met HEAD. Op deze manier worden de HEAD-kenmerken van het voorop geplaatste element en het corresponderende gat geunificeerd. Dit is vooral van belang wanneer de vraagwoord-constituent een onderwerp is. In dat geval moeten met name de AGR-kenmerken worden gedeeld met het 'gat':

(25) a. wiens duif denkt bert dat de wedstrijd heeft gewonnen?
b. *wiens duif denkt bert dat de wedstrijd hebben gewonnen?

Een verschil tussen VGAP en WHGAP is dat het eerste een HEAD-kenmerk is, maar het tweede niet. Binnen een VP die een NP bevat moet de waarde van WHGAP bijvoorbeeld gedeeld worden tussen de VP en de NP, en niet tussen de VP en V. Omdat de waarde van WHGAP niet via het *head feature principle* (HFP) kan worden geregeld, is de macro `pass_wh_gap` toegevoegd. Dit unificeert de WHGAP waarden van een moeder en één van de dochters.

In alle gevallen waarin zeker is dat een constituent geen gat kan bevatten dat correspondeert met een vraagwoord-constituent, moet de waarde van WHTRACE op *nil* gezet worden. Dit geldt met name voor NP's en VP's.

Een voorbeeld afleiding is te vinden in figuur 4.19.

4.10 Literatuur

De klassieke verhandeling over UG is Shieber (1986). Een recente inleiding, met de nadruk op taalkundige aspecten, is Sag & Wasow (1999).

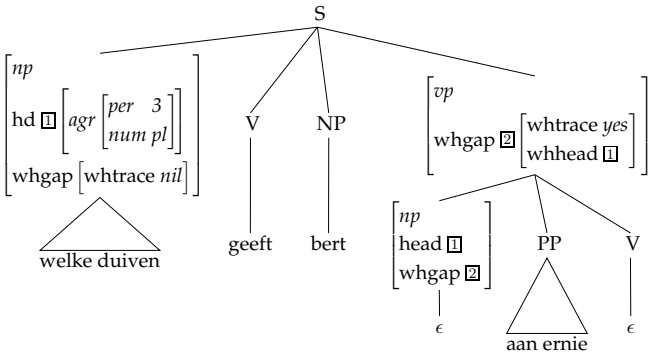
Veel van wat we in dit hoofdstuk behandeld hebben, vindt zijn oorsprong in *Generalized Phrase Structure Grammar* (Gazdar, Klein, Pullum & Sag 1985) en *Head-driven Phrase Structure Grammar* (Pollard & Sag 1994).

```
whtrace(XP,YesNo) :-
    XP:whgap:whtrace <=> YesNo.

wh_fronting(Wh,VP) :-
    whtrace(Wh,no),
    whtrace(VP,yes),
    VP:whgap:whhead <=> Wh:head.

pass_wh_gap(M,D) :-
    M:whgap <=> D:whgap.

regel(whvraag, S, [ WH, V, NP, VP]) :-
    cat(S,s), cat(WH,np), cat(V,v), cat(NP,np), cat(VP,vp),
    head_subject_structure(S,V,NP),
    verb_fronting(V,VP),
    wh_fronting(WH,VP),
    whtrace(NP,no).

regel(vp_np_v, VP  ,[ NP, V ]) :-
    cat(V,v), cat(NP,np), cat(VP,vp),
    head_complement_structure(VP,V,np),
    case(NP,acc),
    pass_wh_gap(VP,NP).

regel(vp_np_pp_v, VP  ,[ NP, PP, V,]) :-
    cat(PP,pp), cat(V,v), cat(NP,np), cat(VP,vp),
    head_complement_structure(VP,V,np_pp),
    case(NP,acc),
    PP:head:pform <=> aan,
    pass_wh_gap(VP,NP).

regel(whgap, np/WH, []) :-
    WH:whgap <=> WH:head.
```

Figuur 4.17: Fragment van een UG met vraagwoord-verplaatsing (I).


```
regel(vraagzin, S, [V, NP, VP]) :-
    cat(S,s), cat(V,v), cat(NP,np), cat(VP,vp),
    head_subject_structure(S,V,NP),
    verb_fronting(V,VP),
    whtrace(NP,no),
    whtrace(VP,no).

regel(hoofdzin, S, [NP, V, VP]) :-
    cat(S,s), cat(V,v), cat(NP,np), cat(VP,vp),
    head_subject_structure(S,V,NP),
    verb_fronting(V,VP),
    wh_trace(VP,no).

woord(bert, NP) :-
    cat(NP,np),
    agr(NP,3,sg),
    whtrace(NP,no).

woord(wie, NP) :-
    cat(NP,np),
    whtrace(NP,no).
```

Figuur 4.18: Fragment van een UG met vraagwoord-verplaatsing (II).



Figuur 4.19: welke duiven geeft bert aan ernie

Een transformationele behandeling van de syntaxis van het Nederlands is te vinden in Model(1991). De ANS (Haesereyn, Romijn, Geerts, De Rooy & Van den Toorn 1997) is een descriptieve grammatica van het Nederlands.

4.11 Opgaven

1. - Pas de grammatica zo aan dat PP's afgeleid kunnen worden.
   - Voeg een aantal werkwoorden toe die een voorzetzelvoorwerp als complement selecteren.

2. - Pas de grammatica zo aan dat bijzinnen die beginnen met dat afgeleid kunnen worden (bijvoorbeeld (bert meent) dat ernie slaapt).
   - Voeg een aantal werkwoorden toe die een dat-zin als complement nemen.

3. In de ANS (Geerts, Haeseryn, De Rooij & Van den Toorn 1984) wordt gesteld dat, in schrijftaal, hun alleen gebruikt kan worden als meewerkend voorwerp dat niet vooraf wordt gegaan door aan:

   (26) a. dat ernie hun de duiven gaf
   b. dat ernie de duiven aan hen gaf

   - Voeg de voornaamwoorden hen (dat natuurlijk ook als lijdend voorwerp gebruikt kan worden) en hun toe aan de grammatica, en zorg ervoor dat genoemde regel wordt gerespecteerd.

4. - Voeg regels voor werkwoordclusters bestaande uit twee werkwoorden, toe.
   - Voeg enkele modale werkwoorden (WILLEN, MOETEN) en de hulpwerkwoorden van tijd (zijn en hebben) toe.
   - Zorg ervoor dat modale werkwoorden alleen met infinitieven en hulpwerkwoorden van tijd alleen met voltooide deelwoorden combineren.
   - Zorg ervoor dat ieder hoofdwerkwoord met het juiste hulpwerkwoord van tijd combineert:

   (27) a. bert heeft aan duiven gedacht
   b. ernie heeft gelachen
   c. bert is gekomen
   d. ernie is gevallen

5. - Voeg de vraagwoord NP's wie en wat en de vragende lidwoorden welke en hoeveel aan de grammatica toe.
   - Pas de grammatica zo aan dat vraagzinnen die beginnen met wie of wat, of met welke ... of hoeveel ... kunnen worden afgeleid.

Bibliografie

Van der Beek, L., G. Bouma & G. van Noord. (**2002**). Een brede computationele grammatica voor het Nederlands. *Nederlandse Taalkunde*, 7(4):353–374.

Blackburn, P. & J. Bos. (**1998**). Representation and inference for natural language: A first course in computational semantics. Technisch rapport, Ms., Department of Computational Linguistics, University of Saarland, Saarbrücken. <http://www.comsem.org/>

Brandt Corstius, H. (**1978**). *Computer-taalkunde*. Coutinho, Muiderberg.

Brandt Corstius, H. (**2003**). De desillusie van mijn leven of remember november. In: T. Gaustad, (red.), *Computational Linguistics in the Netherlands 2002*, 1–8. Rodopi, Amsterdam.

Gazdar, G., E. Klein, G. Pullum & I. Sag. (**1985**). *Generalized Phrase Structure Grammar*. Blackwell.

Geerts, G., W. Haeseryn, J. de Rooij & M. van den Toorn. (**1984**). *Algemene Nederlandse Spraakkunst*. Wolters-Noordhoff, Groningen.

Haesereyn, W., K. Romijn, G. Geerts, J. De Rooy & M. Van den Toorn. (**1997**). *Algemene Nederlandse Spraakkunst*. Martinus Nijhoff Uitgevers Groningen / Wolters Plantyn Deurne. Tweede, geheel herziene druk.

Jurafsky, D. & J. Martin. (**2000**). *Speech and Language Processing: An introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, Upper Saddle River, NJ.

Manning, C. D. & H. Schütze. (**1999**). *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts.

Model, J. (**1991**). *Grammatische Analyse*. ICG Publications, Dordrecht.

Van Noord, G. (**1997**). An efficient implementation of the head-corner parser. *Computational Linguistics*, 23-3.

Ordelman, R., A. van Hessen & F. de Jong. (**2001**). Lexicon optimization for dutch speech recognition in spoken document retrieval. In: *Eurospeech 2001*, 1085–1088. Aalborg.

Pereira, F. C. & S. M. Shieber. (**1987**). *Prolog and Natural Language Analysis*. Center for the Study of Language and Information Stanford.

Pollard, C. & I. Sag. (**1994**). *Head-driven Phrase Structure Grammar*. Center for the Study of Language and Information Stanford.

Sag, I. A. & T. Wasow. (**1999**). *Syntactic Theory: A Formal Introduction*. CSLI Publications, Stanford CA.

Shieber, S. M. (**1986**). *Introduction to Unification-Based Approaches to Grammar*. Center for the Study of Language and Information Stanford.

Vosse, T. (**1994**). *The Word Connection*. proefschrift, Rijksuniversiteit Leiden, (**1994**).