PROTOTYPE FOR ONTOLOGIBASERET TEKSTSØGESYSTEM

Claus Jesper Olesen

LYNGBY 2001
EKSAMENSPROJEKT
IMM-EKS-1999-894

Computer Science & Technology

IMM

Trykt af IM, DTU

Forord

Dette eksamensprojekt tjener som afslutning af min uddannelse til civilingeniør på DTU. Projektet er udført på CST IMM under vejledning af Jørgen Fischer Nilsson og Hans Bruun. Jeg vil benytte lejligheden her til at takke dem for den kompetente, engagerede og motiverende vejledning jeg har modtaget i forbindelse med mit eksamensprojekt. Derudover vil jeg takke min kæreste Hanne Engholm for at have udvist nødvendig tålmodighed og ydet positiv opbakning når det har været nødvendigt.

Claus Jesper Olesen, IM, 5 oktober 2001.

Abstract

The subject of this report is the construction and implementation of a prototype of a text-analyzer controlled by a grammar for Danish and a domain ontology in the form of a concept grammar. The text-analyzer generates descriptions of the semantics of the recognized parts of a text. The generated descriptions can be used in text search based on comparison of semantics instead of words. A built in semantic restriction of the text-analyzer proofs to be very effective in reducing the search space and thereby reducing the time consume of parsing. The implemented prototype offers a user environment for experimenting with the design of the grammars.

KEYWORDS: ontology, text search, concept grammar, parsing, attribute grammar, knowledge based systems.

Indhold

Kapitel 1

Indledning

Søgning i tekst er i dag typisk baseret på forekomster af specifikt angivne søgeord. Denne form for søgning er simpel at implementere, men kvaliteten af søgeresultatet kan være meget svingende. Derfor kræves der som regel en vis erfaring hos brugeren, hvis et godt resultat skal opnås.

For at komme ud over disse begrænsninger kan man forsøge at trække mere information ud af søgestrengen. Fx ved at undersøge sætningens sproglige opbygning i stedet for blot at se på, hvilke ord der forekommer i sætningen. Ligeledes kan man inddrage viden om det videndomæne, der søges indenfor og derved forsøge at opnå en forståelse af sætninger og søgestrenges semantiske indhold. Søgning kan derved baseres på sammenligning af semantisk indhold frem for på sammenligning af ord.

Det er denne tilgang til tekstsøgning, som ligger til grund for et ontologibaseret tekstsøgesystem.

1.1 Ontologibaseret søgning

Projektet er gennemført i tilknytning til OntoQuery- forskningsprojektet, som omhandler udvikling af teorier og metoder til indholdsbaseret tekstsøgning.

Et ontologibaseret søgesystem, som det er beskrevet i forbindelse med OntoQuery-forskningsprojektet, omfatter en tekstanalysator, der styres af

en grammatik for dansk samt en domæneontologi givet i form af en såkaldt begrebsmæssig grammatik. Tekstanalysatoren genererer ved hjælp af den begrebsmæssige grammatik beskrivelser af det semantiske indhold af de led i teksten, der kan genkendes. Sådanne beskrivelser kaldes i det følgende for beskrivere.

De dannede beskrivere kan anbringes sammen med tilhørende tekstreferencer i en database for brug til begrebsmæssig søgning i teksterne ved hjælp af en søger, der kan sammenligne beskrivere. Ved søgning danner tekstanalysatoren således beskrivere ud fra søgestrengen. De fundne beskrivere sammenlignes derefter med beskriverne i databasen, og et søgeresultat kan findes.

Søgningen er således baseret på sætningers semantiske indhold i form af beskrivere.

1.2 Projektets formål

Formålet med dette projekt er at konstruere og implementere en prototype af en tekstanalysator, der som beskrevet er i stand til at danne en beskrivelse af et tekstelements semantiske indhold. Dette gøres med baggrund i to BNF-grammatikker. Den ene afspejler den sproglige struktur af de sætningsled, som ønskes analyseret, og læner sig derfor op af en naturligtsprogsgrammatik for dansk. Grammatikken kaldes i det følgende for den lingvistiske grammatik. Den anden grammatik beskriver en ontologi opstillet for teksternes videndomæne. Denne grammatik kaldes i det følgende for den begrebsmæssige grammatik.

Prototypen skal endvidere gøre det muligt at arbejde eksperimentielt med udformningen af grammatikkerne, og dermed også med udformningen af den ontologiske struktur.

1.3 Rapportens opbygning

Indledningsvis behandles den grundliggende teori for grammatikker. Dette gøres i kapitel 2, som også indeholder en kort beskrivelse af et lingvistisk analysesystem LKB.

Kapitel 3 behandler syntaksanalyse og indeholder blandt andet en beskrivelse af de grundlæggende principper for den syntaksanalyse, som senere anvendes i tekstanalysatoren.

Derefter beskrives i kapitel 4 selve tekstanalysatorens virkemåde, og herunder hvordan de to BNF-grammatikker anvendes af tekstanalysatoren.

I kapitel 5 beskrives implementeringen af prototypen af tekstanalysatoren, hvorefter en afprøvning af tekstanalysatoren samt tilhørende køretider er beskrevet i kapitel 6.

I kapitel 7 diskuteres afslutningsvis begreberne partialitet og robusthed i relation til den beskrevne tekstanalysator.

Konklusionen findes i kapitel 8.

Som bilag findes en brugervejledning til den implementerede prototype samt den bagvedliggende kildekode.

Kapitel 2

Grammatikker

I det følgende beskrives grammatikbegrebet, samt tilhørende begreber som tvetydighed og rekursion. Desuden beskrives attributgrammatikker og sidst i kapitlet beskrives et attributbaseret analysesystem LKB.

2.1 Grammatikker generelt

En grammatik er grundlæggende en hensigtsmæssig måde at beskrive strukturen af lineære repræsentationer på og benyttes ofte, som en opskrift på hvordan man konstruerer sætninger i et af grammatikken angivet sprog.

Formelt kan en grammatik angives ved en 4-tuppel, som nedenfor.

$$(V_n, V_t, R, S)$$

V_n og V_t er endelige disjunkte mængder af henholdsvis ikke-terminale symboler og terminale symboler. De terminale symboler er de symboler, som kan optræde i sætninger i sproget. Alle sætninger består således udelukkende af terminale symboler. De ikke-terminale symboler er de øvrige symboler, som anvendes i grammatikkens regler. Symbolsekvenser, der indeholder ikke-terminale symboler, kaldes for sætningsformer. R er mængden af regler, som er par (p, q) af symbolsekvenser hvorom der gælder følgende:

$$\textit{for alle } (p,q) \in R \textit{ gælder } p \in (V_n \cup V_t)^* V_n (V_n \cup V_t)^* \ \wedge \ q \in (V_n \cup V_t)^*$$

S er det element fra V_n, som udgør startsymbolet, og som anvendes som udgangspunkt for sætninger opbygget af grammatikken.

2.1.1 Afledningsrelationen

For at give grammatikken evnen til at generere sætninger indføres afledningsrelationen, som gælder imellem sekvenser af symboler fra grammatikken, hvis den ene kan afledes over i den anden ved hjælp af en af grammatikkens regler. Relationen kan skrives $u \rightarrow v$, og får formélt følgende udseende:

$$\begin{array}{l} u \rightarrow v \\ \quad \textit{hvis og kun hvis} \\ \quad u = xy_1z \quad \wedge \\ \quad v = xy_2z \quad \wedge \\ \quad (y_1, y_2) \in R \\ \textit{for } x, z \in (V_n \cup V_t)^* \end{array}$$

Ved gentagen anvendelse af relationen kan man ud fra startsymbolet opbygge sætninger i det af grammatikken angivne sprog. Afledningsrelationen giver således en grammatik evnen til at generere sætninger.

Derved fremkommer også en af grammatikkonstruktionens væsentlige styrker, da antallet af sætninger, som kan konstrueres ud fra en grammatik ved hjælp af afledningsrelationen er potentielt uendeligt til trods for grammatikkens endelige form.[1]

Hvordan sætninger konkret opbygges, uddybes og eksemplificeres senere i afsnit 2.3 om BNF-grammatikker.

2.2 Kontekstfri grammatikker

Indenfor de beskrevne rammer findes der forskellige klasser af grammatikker. De adskiller sig typisk ved, hvilke regelkonstruktioner der er tilladte, og

[1] Bemærk i den forbindelse, at ikke alle sprog kan beskrives ved en endelig beskrivelse.

$V_n = \{S, NP, PP\}$
$V_t = \{Noun, Prep, Adj\}$
$R = \{(S,\ NP),$
$\quad (NP,\ NP\ \ PP),$
$\quad (NP,\ Adj\ \ NP),$
$\quad (NP,\ Noun),$
$\quad (PP,\ Prep\ \ NP)\}$
$S = S$

Figur 2.1: Eksempel på kontekstfri grammatik

dermed også ved hvilke typer sprog de kan generere. En af disse klasser indeholder de kontekstfri grammatikker, hvorunder BNF-grammatikker hører.

At en grammatik er kontekstfri vil sige, at grammatikkens regler er uafhængige af både venstre- og højrekonteksten. Dette giver ændrede krav til udformningen af grammatikkens regler, som beskrevet nedenfor.

for alle $(p, q) \in R$ *gælder* $p \in V_n \quad \wedge \quad q \in (V_n \cup V_t)^*$

For kontekstfri grammatikker består venstresiden i hver regel således kun af et enkelt ikke-terminalt symbol.

2.3 FBNgammatikker

BNF står for *Backus-Naur Form* og angiver en notation for kontekstfri grammatikker. I figur 2.1 ses et eksempel på en kontekstfri grammatik, og i figur 2.2 ses den på BNF-form, som vil blive anvendt i det følgende. Som det fremgår, markeres de ikke-terminale symboler med vinkelparenteser, og reglernes venstre- og højresider adskilles af "$::=$". Det er desuden muligt, at samle regler med samme venstreside til én regel med angivelse af de mulige højresider adskilt af "|". Startsymbolet er symbolet, på venstresiden i den først angivne regel.

Afledningsrelationen anvendes til at opbygge de sætninger, som grammatikken indeholder opskriften på. Dette gøres med udgangspunkt i startsymbolet S, som ved hjælp af afledningsrelationen og grammatikkens regler omdannes til sætningsformer og sætninger. Sætningen *Noun Prep Noun*

$\langle S\rangle ::= \langle NP\rangle$
$\langle NP\rangle ::= \langle NP\rangle\ \ \langle PP\rangle \mid Adj\ \ \langle NP\rangle \mid Noun$
$\langle PP\rangle ::= Prep\ \ \langle NP\rangle$

Figur 2.2: Eksempel på BNF-grammatik

	Anvendt regel:
$\langle S\rangle$	$\langle S\rangle ::= \langle NP\rangle$
$\langle NP\rangle$	$\langle NP\rangle ::= \langle NP\rangle\ \ \langle PP\rangle$
$\langle NP\rangle\ \ \langle PP\rangle$	$\langle PP\rangle ::= Prep\ \ \langle NP\rangle$
$\langle NP\rangle\ \ Prep\ \ \langle NP\rangle$	$\langle NP\rangle ::= \langle NP\rangle\ \ \langle PP\rangle$
$\langle NP\rangle\ \ Prep\ \ \langle NP\rangle\ \ \langle PP\rangle$	$\langle PP\rangle ::= Prep\ \ \langle NP\rangle$
$\langle NP\rangle\ \ Prep\ \ \langle NP\rangle\ \ Prep\ \ \langle NP\rangle$	$\langle NP\rangle ::= Adj\ \ \langle NP\rangle$
$\langle NP\rangle\ \ Prep\ \ \langle NP\rangle\ \ Prep\ \ Adj\ \ \langle NP\rangle$	$\langle NP\rangle ::= Noun$
$\langle NP\rangle\ \ Prep\ \ \langle NP\rangle\ \ Prep\ \ Adj\ \ Noun$	$\langle NP\rangle ::= Noun$
$\langle NP\rangle\ \ Prep\ \ Noun\ \ Prep\ \ Adj\ \ Noun$	$\langle NP\rangle ::= Noun$
$Noun\ Prep\ \ Noun\ \ Prep\ \ Adj\ \ Noun$	

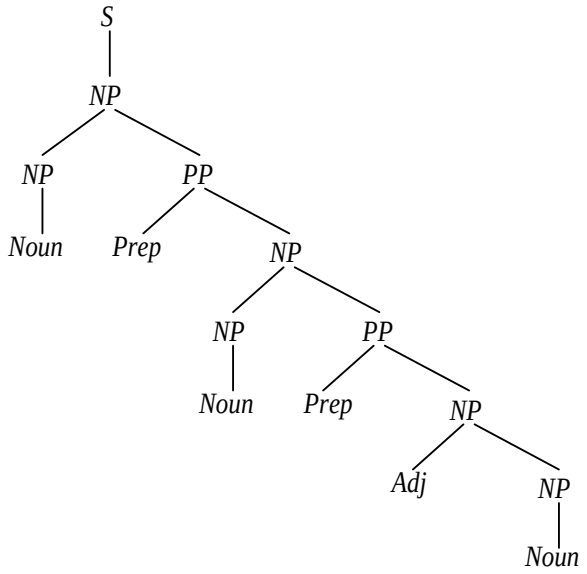
Figur 2.3: Generering af sætningen *Noun Prep Noun Prep Adj Noun*

Prep Adj Noun kan fx opbygges ved brug af den sekvens af regler, som er vist i figur 2.3.

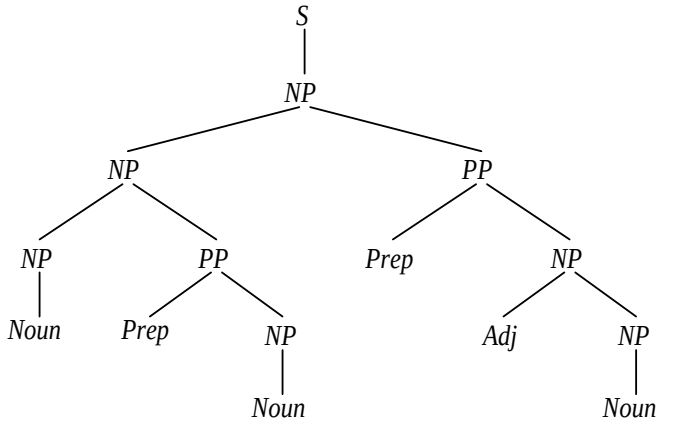
2.4 Syntakstræer

De afledninger som gennemføres, når sætninger i sproget genereres ud fra grammatikkens regler, kan illustreres ved hjælp af syntakstræer. Syntakstræer viser på overskuelig vis sætningens syntaktiske struktur med hensyn til grammatikken, og forbinder på sin vis sætningen og grammatikken med hinanden. Strukturen er træformet på grund af reglernes forholdsvis simple opbygning som en følge af kontekstfriheden. I figur 2.4 ses et syntakstræ for sætningen fra før.

Syntakstræer spiller en væsentlig rolle, når man ønsker at bruge grammatikken ikke kun som generator af sætninger, men også i forbindelse med genkendelse af sæninger i et sprog. Det som man derved ønsker, er netop



Figur 2.4: Syntakstræ for *Noun Prep Noun Prep Adj Noun*



Figur 2.5: Syntakstræ for *Noun Prep Noun Prep Adj Noun*

at opbygge et syntakstræ for en given sætning med hensyn til en grammatik. Lykkes dette er det et udtryk for at den givne sætning tilhører det sprog som grammatikken angiver. Til dette formål kan man foretage en syntaksanalyse (engelsk: *Parsing*) af en given sætning med hensyn til en given grammatik.

2.5 Tvetydighed

For nogle grammatikker kan der opstilles flere forskellige syntakstræer for den samme sætning. Sådanne grammatikker kaldes tvetydige, da den pågældende sætning således ikke har en entydig forbindelse til grammatikken. Et eksempel på en tvetydig sætning er sætningen fra før, som udover det viste syntakstræ i figur 2.4 også har syntakstræet i figur 2.5. Grammatikken fra figur 2.2 er således tvetydig.

Da tvetydighed har afsæt i på hvilke måder en sætning kan afledes ved hjælp af grammatikkens regler, er det selvfølgelig reglernes udformning, der er afgørende for om en grammatik er tvetydig eller ej.

Der findes overordnet to typer af tvetydighed.

- Der er tale om bundet tvetydighed, hvis der for tvetydige sætninger altid findes et endeligt antal forskellige syntakstræer.

- Ubundet tvetydighed forekommer, hvis der findes sætninger, som har et uendeligt antal forskellige syntakstræer i forhold til grammatikken.

Da sætningen i figur 2.4 og figur 2.5 kun har de to viste syntakstræer i forhold til grammatikken er der således tale om bundet tvetydighed.

Ubundet tvetydighed kan forekomme ved, at symboler kan afledes over i sig selv, som fx via reglerne:

$$\langle NP \rangle ::= \langle PP \rangle$$
$$\langle PP \rangle ::= \langle NP \rangle$$

En afledning fra $\langle NP \rangle$ til $\langle NP \rangle$ kan foretages et uendeligt antal gange, uden at syntaksanalysen bringes fremad i form af ændringer i den aktuelle sætningsform. Den ubundne tvetydighed fremgår ikke altid så tydeligt af grammatikken, som i eksemplet her, da den også kan være skjult i et større antal mere komplicerede regler. Desuden kan ubundet tvetydighed forekomme, hvis et eller flere symboler kan afledes over i den tomme streng.

Tvetydighed kan give problemer i forbindelse med syntaksanalyse, hvis der ikke i syntaksanalysen tages hensyn til den tvetydighed, som en grammatik besidder. Vil man arbejde med tvetydige grammatikker, må man derfor sørge for at indrette sin syntaksanalyse, så den tager hensyn til den tvetydighed, der forekommer i de grammatikker, man ønsker at anvende.

2.6 Rekursion

En anden egenskab, som udspringer af reglernes udformning, er rekursion. En regel er rekursiv, hvis symbolet på venstresiden af reglen også optræder i reglens højreside. Derudover kan en regel være venstrerekursiv eller højrerekursiv.

En regel er venstrerekursiv, hvis venstresidens symbol optræder som det første symbol i højresiden og højrerekursiv, hvis symbolet optræder som det sidste symbol i højresiden.

Reglen $\langle NP \rangle ::= \langle NP \rangle \ \langle PP \rangle$ er et eksempel på en venstrerekursiv regel.

Ligesom tvetydighed kan også rekursion optræde indirekte igennem flere regler.

Også rekursion kan give problemer i forhold til syntaksanalyse, hvorfor der også må tages hensyn til denne i syntaksanalyse med rekursive grammatikker eller i udformningen af grammatikker.

Der findes dog metoder til at fjerne venstrerekursionen fra en grammatik, uden at det sprog grammatikken genererer forandres.

2.7 Attributgrammatikker

En attributgrammatik er en grammatik, hvor de enkelte knuder i syntakstræerne er tilknyttet attributter i form af værdier af en type, som er relevant i den pågældende anvendelse. Det betyder, at der til hver regel i grammatikken er tilknyttet information om, hvordan eventuelle attributværdier for de ikke-terminale symboler i reglen kan beregnes i form af attributfunktioner. Attributfunktionernes beregning er baseret på værdien af de andre attributværdier i reglen. Bemærk at ikke-terminale symboler i en regel svarer til knuder i et syntakstræ.

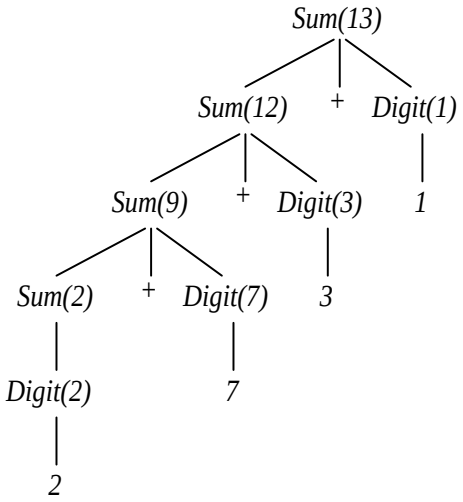
Hvis en attributværdi beregnes for et ikke-terminalt symbol N i reglens højreside, siges værdien at være nedarvet af N. Værdien vil i dette tilfælde bevæge sig nedad i et tilhørende syntakstræ.

Hvis en værdi beregnes for det ikke-terminale symbol i reglens venstreside, siges værdien at være syntetiseret. Værdien vil i dette tilfælde bevæge sig opad i et tilhørende syntakstræ.

I figur 2.6 ses et eksempel på en attributgrammatik til summe af en-cifrede tal. For hver regel er der således angivet hvordan attributværdierne beregnes. Grammatikken har én attribut for hvert ikke-terminale symbol. Attributten anvendes i dette tilfælde til at beregne summen af de udtryk, som grammatikken kan opbygge. Da alle attributfunktioner udelukkende beregner attributværdien for symbolet på venstresiden, vil alle værdier bevæge sig opad i de syntakstræer, som opbygges på baggrund af grammatikken. Dette er illustreret i figur 2.7, som viser syntakstræet for sætningen $2 + 7 + 3 + 1$.

$$
\begin{array}{ll}
\langle Sum\rangle ::= \langle Digit\rangle & \{Sum_{value} := Digit_{value}\} \\
\langle Sum\rangle_1 ::= \langle Sum\rangle_2 + \langle Digit\rangle & \{Sum_{1value} := Sum_{2value} + Digit_{value}\} \\
\langle Digit\rangle ::= 0 & \{Digit_{value} := 0\} \\
\langle Digit\rangle ::= 1 & \{Digit_{value} := 1\} \\
\quad : & \quad : \\
\langle Digit\rangle ::= 9 & \{Digit_{value} := 9\}
\end{array}
$$

Figur 2.6: Eksempel på attributgrammatik



Figur 2.7: Syntakstræ og attributværdier for $2 + 7 + 3 + 1$

$$
\begin{array}{ll}
\langle Sum\rangle ::= \langle Digit\rangle & \{Digit.i := Sum.i\} \\
 & \{Sum.s := Digit.s\} \\
\langle Sum\rangle_1 ::= \langle Sum\rangle_2 + \langle Digit\rangle & \{Sum_2.i := Sum_1.i\} \\
 & \{Digit.i := Sum_2.s\} \\
 & \{Sum_1.s := Digit.s\} \\
\langle Digit\rangle ::= 0 & \{Digit.s := 0 + Digit.i\} \\
\langle Digit\rangle ::= 1 & \{Digit.s := 1 + Digit.i\} \\
\quad : & \quad : \\
\langle Digit\rangle ::= 9 & \{Digit.s := 9 + Digit.i\}
\end{array}
$$

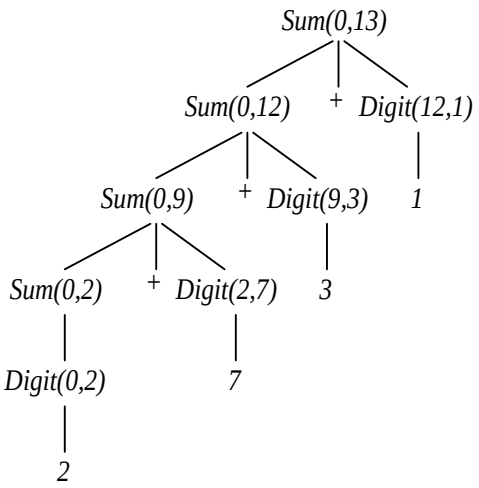
Figur 2.8: Eksempel på attributgrammatik med flere attributter

Fra starten af kendes kun værdierne fra de terminale symboler i bunden af syntakstræet. Værdierne længere oppe i træet beregnes på baggrund af attributfunktioner, som er knyttet til grammatikkens regler. Hvilke beregninger der konkret udføres, afgøres selvfølgelig af hvilke regler, der er anvendt i opbygningen af syntakstræet. Det ses, at syntetiserede værdier fra underliggende dele i træet på denne måde sendes opad i træet. Ved startknuden i toppen af syntakstræet findes således det samlede resultat af hele udtrykket, når samtlige beregninger er udført.

2.7.1 Flere attributter

I grammatikken i figur 2.6 bliver der kun benyttet en enkelt attribut, ligesom der kun arbejdes med syntetiserede attributværdier. Der kan også arbejdes med flere attributter, og værdier, der både syntetiseres og nedarves i den samme grammatik. Da dette anvendes i den senere beskrevne tekstanalysator, udvides grammatikken fra før derfor med endnu en attribut for, at eksemplificere dette. Den nye attributværdi skal afspejle det hidtil beregnede resultat. Da den ene af de nu to attributværdier hele tiden nedarves, og den anden hele tiden syntetiseres, kaldes de for henholdsvis i (*inherited*) og s (*synthesized*). Grammatikken ses i figur 2.8.

Attributten i er det hidtil beregnede resultat fra længere oppe i syntakstræet, det er således en nedarvet værdi, som derfor skal beregnes for ikke-terminale symboler i reglernes højresider. Den anden attribut s repræsenterer den samme værdi som ved den tidligere grammtik. I dette tilfælde beregnes den blot på en anden måde på baggrund af attributten med det



Figur 2.9: Andet syntakstræ og attributværdier for $2 + 7 + 3 + 1$

foreløbige resultat. Syntakstræet og de nye attributværdier for sætningen $2 + 7 + 3 + 1$ kan ses i figur 2.9. Det ses, at der nu sendes værdier både op og ned i syntakstræet. Da den øverste knude i syntakstræet også forventes at have nedarvet en værdi, sættes denne naturligvis til 0, da attributten repræsenterer den hidtil beregnede værdi.

Eksemplet illustrerer hvordan der kan medbringes information nedad fra længere oppe i syntakstræet, hvilket som nævnt vil blive anvendt i forbindelse med tekstanalysatoren.

2.7.2 Beregning af attributværdier

Den rækkefølge, som attributværdierne kan beregnes i, er ikke vilkårlig, men afgøres af hvordan de forskellige værdier afhænger af hinanden igennem attributfunktionerne.

Hvis beregningerne er baseret på venstrekonteksten som i dette tilfælde, og som de vil være i forbindelse med tekstanalysatoren, kan attributfunktionernes beregninger selvsagt kun være baseret på de attributværdier, som i

$$
\begin{aligned}
S_0 ::= S_1\ S_2\ S_3\ ...\ S_n\ \ & \{S_1.i = f_1(S_0.i)\} \\
& \{S_2.i = f_2(S_0.i,\ S_1.s)\} \\
& \{S_3.i = f_3(S_0.i,\ S_1.s,\ S_2.s)\} \\
& \quad : \\
& \{S_n.i = f_n(S_0.i,\ S_1.s,\ S_2.s,\ ...,\ S_{n-1}.s)\} \\
& \{S_0.s = f_{n+1}(S_0.i,\ S_1.s,\ S_2.s,\ ...,\ S_n.s)\}
\end{aligned}
$$

Figur 2.10: Attributafhængigheder

reglen forekommer "til venstre" for den attributværdi som ønskes beregnet. Grammatikker, hvis attributfunktioner opfylder dette, kaldes *leftattributed*.

Afhængigheden, imellem attributværdierne vil for en regel i en *leftattributed* grammatik, følge mønstret angivet i figur 2.10.

$S_1 ... S_n$ er ikke-terminale symboler. Højresiden kunne dog også indeholde terminale symboler, som i så fald ikke ville have tilknyttet en attributfunktion, men blot vil returnere en syntetiseret værdi s.

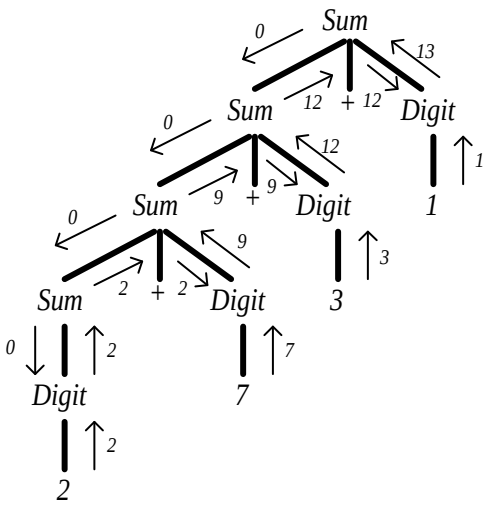
Som det ses, må attributfunktionerne for de forskellige ikke-terminale symboler således basere deres beregninger på de attributværdier, der findes i venstrekonteksten.

For en *leftattributed* grammatik vil attributværdierne kunne beregnes i et gennemløb af syntakstræet fra venstre mod højre, hvor forældre-knuder besøges før børn i et såkaldt *preorder traversal*. Værdiernes vej igennem syntakstræet, i et sådant gennemløb er vist i figur 2.11.

2.8 LKB - et attributbaseret grammatiksystem

LKB står for *Linguistic Knowledge Building* og er navnet på et system til forskning og udvikling indenfor naturligsprogsgrammatikker og leksika. LKB er bl.a. udviklet med henblik på, at lingvister uden specielt stor erfaring med computere kan arbejde med grammatikker og syntaksanalyse.

Systemet er baseret på en attributbaseret datastruktur *typed feature structures*, som blandt andet gør det muligt at tilknytte leksikal information til grammatikker.



Figur 2.11: Syntakstræ og attributværdiernes retning for $2 + 7 + 3 + 1$

En kort beskrivelse af LKB-systemet er medtaget, da systemet også kan anvendes i forbindelse med tekstanalyse i sammenhæng med en begrebsmæssig modellering.

For en mere tilbundsgående beskrivelse af LKB-systemet henvises til [14].

2.8.1 *Typed Feature Structures*

Systemet er udelukkende baseret på datastrukturen *Typed feature structures*, som er en formél måde at angive lingvistisk information og strukturer på.

I figur 2.12 vises en simpel grammatik med leksikal information. Denne grammatiks definition ved hjælp af *typed feature structures* i LKB-systemet kan ses i figur 2.13. Eksemplet er taget fra [14].

Definitionen består af tre dele: Et type-system, leksikale indgange og grammatikkens regler. Alt angives ved hjælp af *typed feature structures*.

Type-systemet definerer rammerne for den øvrige grammatik, og angiver

$\langle S \rangle ::= \langle NP \rangle \;\; VP$
$\langle NP \rangle ::= Det \;\; N$

Leksikon:
dog : N
dogs: N
this : Det
these: Det
sleeps : VP
sleep : VP

Figur 2.12: Simpel grammatik med leksikal information

helt fra bunden det type-hierarki, som anvendes i forbindelse med angivelsen af reglerne og de leksikale informationer. I typesystemet definerer en *typed feature structure* en type, som peger på en forældre-type, og som kan være tilknyttet et antal attributter i form af rolle-værdi par, hvor rollen på sin vis navngiver en relation til en anden defineret type. Rolle-værdi parrene skal ses som restriktioner på den tilknyttede type.

Desuden nedarves rolle-værdi par fra overliggende typedefinitioner i hierarkiet, så når fx de leksikale indgange defineres, som værende af typen `lexeme`, skal de udfylde alle de rolle-værdi par, som er defineret og nedarvet for typen `lexeme`. I dette tilfælde værdier for rollerne `ORTH` og `CATEGORY`.

De leksikale indgange giver en sammenhæng imellem terminalsymbolerne i grammatikken og ordene i sætningerne. Definitionerne gør det således muligt at genkende sætninger som "these dogs sleeps" såvel som "this dog sleeps".

Reglerne for grammatikken defineres tilsvarende med udgangspunkt i typen `phrase`, som dog er af en mere kompliceret struktur. Det skyldes blandt andet, at en regel er defineret ved en kategori, som afspejler venstresiden af reglen og ved en liste af kategorier, der afspejler højresiden.

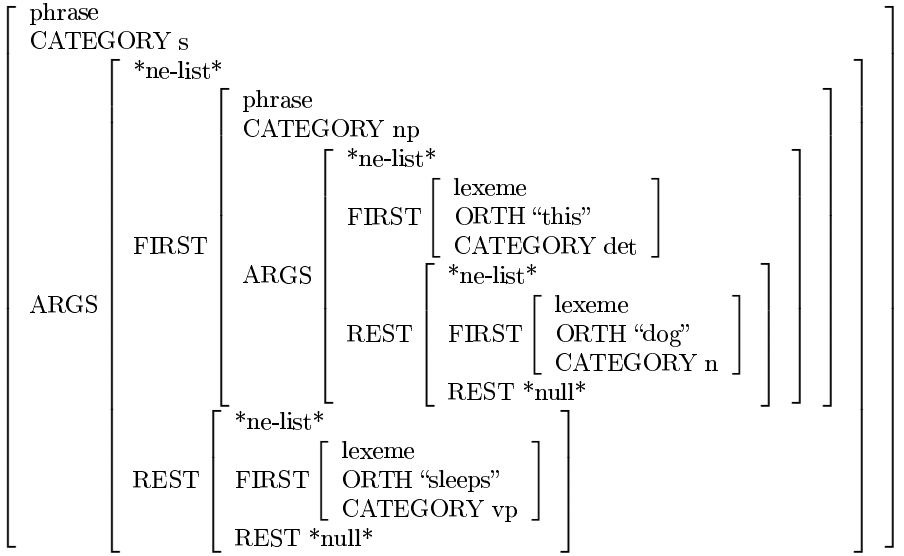
2.8.2 Unifikation

Syntaksanalyse foretages ved hjælp af unifikation. Unifikationen finder ud fra to *typed feature structures* den mest generelle *typed feature structure*, som rummer al informationen fra de to givne. Er en unifikation, der omfatter alle de angivne *typed feature structures*, ikke mulig, fejler syntaksanalysen.

```
;;; Types
string := *top*.

*list* := *top*.

*ne-list* := *list* &
[ FIRST *top*,
 REST *list* ].

*null* := *list*.

synsem-struc := *top* &
[ CATEGORY cat ].

cat := *top*.

s := cat.

np := cat.

vp := cat.

det := cat.

n := cat.

phrase := synsem-struc &
[ ARGS *list* ].

lexeme := synsem-struc &
[ ORTH string ].

root := phrase &
[ CATEGORY s ].
```

```
;;; Lexicon
this := lexeme &
[ ORTH "this",
 CATEGORY det ].

these := lexeme &
[ ORTH "these",
 CATEGORY det ].

sleep := lexeme &
[ ORTH "sleep",
 CATEGORY vp ].

sleeps := lexeme &
[ ORTH "sleeps",
 CATEGORY vp ].

dog := lexeme &
[ ORTH "dog",
 CATEGORY n ].

dogs := lexeme &
[ ORTH "dogs",
 CATEGORY n ].

;;; Rules
s_rule := phrase &
[ CATEGORY s,
 ARGS [ FIRST [ CATEGORY np ],
        REST [ FIRST [ CATEGO-
RY vp ],
               REST *null* ]]].

np_rule := phrase &
[ CATEGORY np,
 ARGS [ FIRST [ CATEGORY det ],
        REST [ FIRST [ CATEGO-
RY n ],
               REST *null* ]]].
```

Figur 2.13: Simpel grammatik ved hjælp af *typed feature structures* i LKB



Figur 2.14: AVM for *This dog sleeps* ved brug af *typed feature structures*

For sætningen "this dog sleeps" fås som resultat den *typed feature structure*, som ses i figur 2.14, på baggrund af den angivne grammatik. For overskuelighedens skyld er den vist i form af en *attribute value matrix* (AVM).

Som det ses er det relativt omfattende strukturer der arbejdes med på trods af grammatikken og sætningens enkelhed.

2.8.3 Ontologisk modellering

Det her viste eksempel, indeholder ingen ontologisk modellering af et tilhørende videndomæne, svarende til den begrebsmæssige grammatiks rolle i tekstanalysatoren. En sådan ontologisk struktur kan dog modelleres i LKB ved hjælp af et yderligere antal *typed feature structures*, hvorved systemet kan anvendes til, at generere beskrivelser af semantisk indhold ligesom den beskrevne tekstanalysator.

I afsnit 4.2.4 uddybes beskrivelsen af en begrebsmæssig modellering i LKB i forbindelse med beskrivelsen af den begrebsmæssige grammatik.

Kapitel 3

Syntaksanalyse

I dette kapitel beskrives først de analyse- og søgestrategier, som ligger til grund for forskellige syntaksanalysealgoritmer. Derefter beskrives en *breadth-first bottom-up* syntaksanalysator, og som en udbygning af denne beskrives en modificeret udgave af Earleys syntaksanalysator, som senere vil ligge til grund for den bekrevne tekstanalysator. Sidst i kapitlet beskrives syntaksanalyse af attributgrammatikker kort.

Målet med en syntaksanalyse af en sætning med hensyn til en given grammatik er, som tidligere beskrevet, at få opbygget et syntakstræ, der beskriver forbindelsen imellem sætningen og grammatikken. Selvfølgelig forudsat at det overhovedet er muligt, at danne sætningen ud fra grammatikken.

3.1 Analysestrategier

Når en syntaksanalysator skal konstrueres, kan analysen angribes på flere forskellige måder, men grundlæggende har man valget imellem to tilgange til problemet. Det kan gøres *top-down* eller *bottom-up*. Betegnelserne dækker over, hvordan det ønskede syntakstræ forsøges opbygget.

3.1.1 *Top-down*

En syntaksanalysator, der tager udgangspunkt i en *top-down* tilgang, vil prøve at opbygge det ønskede syntakstræ "fra toppen". Det vil sige, at analysatoren bruger grammatikkens startsymbol som udgangspunkt, for derefter ved hjælp af grammatikkens regler at aflede den sætning som ønskes analyseret.

Denne taktik stiller store krav til analysatoren, da det sjældent vil være oplagt, hvilke regler der bør vælges for at ende op med den givne sætning.

3.1.2 *Bottom-up*

En syntaksanalysator, der tager udgangspunkt i en *bottom-up* tilgang, vil i stedet prøve at opbygge det ønskede syntakstræ "fra bunden". Det vil sige, at analysatoren bruger symbolerne i sætningen, der ønskes analyseret, som udgangspunkt for reduktioner på baggrund af grammatikkens regler. Reglerne vil således blive benyttet "baglæns", hvorved den givne sætning, efter et antal reduktioner vil være reduceret til grammatikkens startsymbol.

Denne taktik giver på nogle punkter analysatoren et lettere arbejde end med *top-down* tilgangen, da symbolerne i den givne sætning blot kan sammenlignes med reglernes højresider, når regler til reduktion skal vælges. Denne fremgangsmåde sikrer dog ikke nødvendigvis, at de valgte regler fører til startsymbolet.

3.2 Søgestrategier

Uanset, hvilken af de beskrevne analysestrategier der anvendes, vil syntaksanalysatoren løbende skulle vælge imellem et antal mulige træk i form af afledninger/reduktioner, som vil afhænge af tidligere foretagne træk. Som løsning på det problem anvendes en afsøgning af mulighederne, som grundlæggende kan følge en af to søgestrategier: *Depth-first* og *breadth-first*. Søgestrategierne er generelle og anvendes ikke kun i forbindelse med syntaksanalyse.

3.2.1 *Depth-first*

I en *depth-first* søgning forfølges en potentiel løsning indtil, det kan afgøres om, den ér en løsning eller ej. Hvis den ikke viser sig at være en løsning, findes der tilbage til det seneste sted, hvor der findes en endnu ikke afsøgt gren, som så afsøges. Denne mekanisme er også kendt som *backtracking*. Dette fortsættes indtil en løsning er fundet eller hele løsningsrummet er afsøgt.

Den primære styrke ved en *depth-first* søgning er det begrænsede hukommelsesbehov, som er proportionalt med problemets størrelse, i modsætning til en *breadth-first* søgning som i værste fald skal bruge en eksponentiel mængde hukommelse.

Depth-first afsøgning kan dog give problemer i forbindelse med ubundet tvetydighed, da søgningen kan havne i en uendelig løkke på grund af symboler, der kan afledes over i sig selv. Samme problem kan opstå, hvis søgestrategien anvendes i forbindelse med en top-down analyse af en rekursiv grammatik.

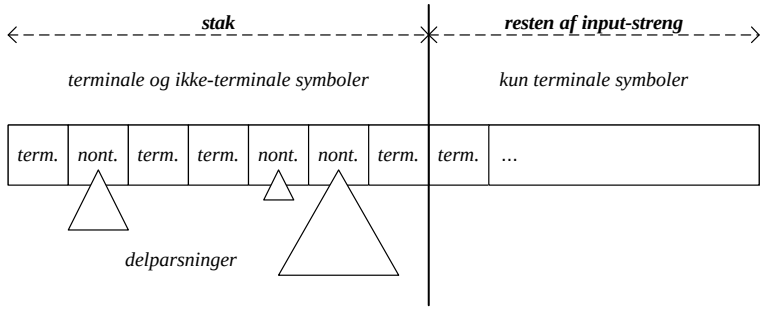
3.2.2 *Breadth-first*

I en *breadth-first* søgning arbejdes der parallelt på alle muligheder. Det vil sige, at der i hvert skridt gås et skridt i alle mulige retninger. Dette fortsættes indtil en løsning er fundet eller indtil, der ikke kan kommes længere i nogen retning.

Styrken ved en *breadth-first* søgning er, at den første løsning der findes også er den mest enkle, i modsætning til *depth-first* søgning, som i værste fald når hele søgerummet igennem, før den mest enkle løsning findes. Desuden har en *breadth-first* søgning ikke samme tendens til at gå i uendelig løkke, da der arbejdes med en mængde mulige løsninger sideløbende.

3.3 *Breadth-first bottom-up* syntaksanalyse

Den syntaksanalyse, som ligger til grund for tekstanalysatoren i prototypen, er baseret på en *breadth-first* søgestrategi kombineret med en *bottom-up* analyse. Principperne for en sådan syntaksanalysator gennemgås derfor i det følgende.



Figur 3.1: Repræsentation af sætningsform

Analysatoren vil med baggrund i analysestrategien forsøge at opbygge syntakstræet nedefra og op. Syntaksanalysen tager således udgangspunkt i den sætning, som ønskes analyseret. Analysatoren vil forsøge at reducere sætningen ved hjælp af grammatikkens regler, efterhånden som symbolerne i sætningen indlæses. Derved vil der opstå nye sætningsformer, som så igen forsøges reduceret. Dette fortsættes indtil alle symboler fra den givne sætning er indlæst, og der forhåbentligt er reduceret til grammatikkens startsymbol. Da søgestrategien er *breadth-first*, vil flere mulige reduktioner af en given sætningsform betyde, at analysatoren vil oprette kopier af sætningsformen og forfølge alle muligheder og efterfølgende arbejde parallelt med de nyreducerede sætningsformer.

I en konkret udformning af en *breadth-first bottom-up* syntaksanalysator er der således behov for at kunne repræsentere sætninger og sætningsformer, samt at kunne udføre operationer som indlæsning af symboler og reduktion af sætningsformer.

3.3.1 Sætningsformer som stakke

En sætningsform kan passende repræsenteres ved en stak af terminale og ikke-terminale symboler svarende til de symboler som udgør sætningsformen. Når sætningsformen skal "forlænges" med det næste indlæste symbol fra den givne sætning, indskiftes symbolet blot på stakken.

På denne måde ligger de endnu ikke indlæste terminale symboler fra den givne sætning på sin vis i forlængelse af stakken, som det er vist i figur 3.1.

Som det er illustreret vil hvert ikke-terminale symbol i en sætningsform, og dermed også i stakken, repræsentere et underliggende syntakstræ, som hidrører fra tidligere foretagne reduktioner.

3.3.2 *Shift* og *reduce*

Forløbet af syntaksanalysen vil indeholde et antal gentagne anvendelser af to hovedoperationer kaldet *shift* og *reduce*.

Shift operationens opgave er at indlæse det næste symbol fra den givne sætning. Dette indebærer, at alle de sætningsformer i form af stakke, som de hidtil indlæste symboler har kunnet reduceres til, skal undvides med det indlæste symbol. Dette gøres ved, at symbolet indskiftes på hver stak.

Reduce operationens opgave er at gennemgå stakkene, som netop har fået indskiftet næste symbol, med henblik på reduktion. Da analysen foregår *bottom-up* sker det ved at sammenholde stakindholdene med grammatikreglernes højresider, hvilket typisk vil medføre mulighed for et antal reduktioner. Jvf. *breadth-first* søgestrategien oprettes der for hver mulig reduktion nye stakke så alle de sætningsformer, der er mulige på baggrund af de hidtil indlæste symboler, er repræsenteret.

Det er i denne operation at dele af det ønskede syntakstræ opbygges, da grammatikkens regler anvendes til at reducere symbolsekvenser til enkelte symboler, hvilket afspejler syntakstræets opbygning nedefra.

3.3.3 Analysens forløb

Forløbet af en *breadth-first bottom-up* syntaksanalyse vil udgøres af de beskrevne operationer anvendt på sætningsformer repræsenteret ved stakke som beskrevet.

Overordnet vil syntaksanalysatoren gennemløbe følgende:

```
while (moreSymbolsToRead) {
    shift
    reduce
}
```


I starten af analysen vil der kun være en tom stak, afspejlende at ingen symboler fra den givne sætning er indlæst. Efterhånden som symboler indlæses og nye stakke oprettes i forbindelse med reduktioner, vil antallet af stakke vokse. Når alle symboler i den givne sætning er indlæst, og den sidste reduktions-operation er gennemført, vil man således stå med et antal stakke, som hver især afspejler en sætningsform, som den givne sætning har kunnet reduceres til. Hvis sætningen har kunnet accepteres af grammatikken, vil en eller flere af stakkene udelukkende indeholde grammatikkens startsymbol. Dette vil afspejle, at et syntakstræ der forbinder den givne sætning og startsymbolet eksisterer. Hvis der er flere stakke, som udelukkende indeholder startsymbolet, er det et udtryk for at grammatikken er tvetydig, da der således eksisterer flere syntakstræer for den givne sætning med hensyn til grammatikken. Bemærk at *breadth-first* søgestrategiens natur sikrer, at alle løsninger findes, hvorfor der eksisterer præcis de syntakstræer, som syntaksanalysatoren finder. Analysen kan dog ikke håndtere ubundet tvetydighed, da denne ikke kan repræsenteres af den underliggende datastruktur, som er baseret på opbygning af syntakstræer.

3.3.4 Ulemper

En ulempe ved *breadth-first bottom-up* syntaksanalysen er, at mange af de sætningsformer som er fundet til sidst, og som viser sig *ikke* at være ført til en løsning, kunne være stoppet før. Problemet med dem er, at de ikke kan afledes af startsymbolet. Derfor kan de selvsagt heller aldrig reduceres til startsymbolet, som ønsket. Problemet hænger sammen med *bottom-up* analysestrategien, da problemet er, at sætningsformerne ikke er kompatible med en *top-down* analyse, med udgangspunkt i grammatikkens startsymbol.

Konsekvensen er at syntaksanalysatoren kommer til at arbejde forgæves med sætningsformer, som alligevel aldrig ville kunne blive til gyldige løsninger. Et eksempel på dette er illustreret i figur 3.2.

Allerede fra det først indlæste symbol, kan det ses, at den givne sætning ikke kan accepteres, da den ikke starter med et af symbolerne *Noun* eller *Adj*. En løsning på problemet er at reducere syntaksanalysatorens råderum, så den kun får lov at arbejde med sætningsformer, der også er gyldige i en *top-down* analyse. Det vil sige at alle sætningsformer under opbygning også skal kunne afledes af startsymbolet.

Grammatik:	$\langle S \rangle ::= \langle NP \rangle$
	$\langle NP \rangle ::= \langle NP \rangle\ \langle PP \rangle \mid Adj\ \langle NP \rangle \mid Noun$
	$\langle PP \rangle ::= Prep\ \langle NP \rangle$
Sætning:	$Prep\ Noun\ Prep\ Noun$
	Stakindhold:
Start	
$shift$	$Prep$
$reduce$	$Prep$
$shift$	$Prep\ Noun$
$reduce$	$Prep\ \langle NP \rangle$
	$\langle PP \rangle$
$shift$	$\langle PP \rangle\ Prep$
$reduce$	$\langle PP \rangle\ Prep$
$shift$	$\langle PP \rangle\ Prep\ Noun$
$reduce$	$\langle PP \rangle\ Prep\ \langle NP \rangle$
	$\langle PP \rangle\ \langle PP \rangle$
Slut	

Figur 3.2: Mislykket *bottom-up* syntaksanalyse

I næste afsnit beskrives en modificeret udgave af Earleys algoritme, som netop tager hånd om dette problem.

3.4 Earleys syntaksanalyse - modificeret

Earleys algoritme er en syntaksanalysator beregnet for kontekstfri grammatikker. Algoritmen kan kaldes en *top-down* begrænset *breadth-first bottom-up* analysator, da den er baseret på en *breadth-first bottom-up* strategi som tidligere beskrevet, men med en *top-down* begrænser, der kun gør det muligt at arbejde med sætningsformer, som også kan afledes fra startsymbolet. Earleys algoritme løser således det tidligere beskrevne problem ved en almindelig *breadth-first bottom-up* analyse ved også at inddrage *top-down* analysestrategien i syntaksanalysen. Hvordan *top-down* begrænsningen gør sin indflydelse gældende, forklares i det følgende.

Det skal bemærkes, at den her beskrevne algoritme er en modificeret version af Earleys algoritme. Den adskiller sig således på få punkter i forhold til Earleys algortime. De igangværende syntaksanalyser i form af forskellige sætningsformer er i denne analysator repræsenteret ved et antal stakke som i *breadth-first bottom-up* analysatoren. Earleys algortime benytter i stedet én længere stak, der indeholder alle igangværende sætningsformer flettet ind i hinanden ved hjælp af pegere frem og tilbage i stakken. Derved er algoritmen mindre hukommelsekrævende, ligesom den er i stand til at håndtere alle kontekstfri grammatikker. Det er den her beskrevne modificerede version af algoritmen derimod ikke, da den ikke kan håndtere ubundet tvetydighed. Denne restriktion opstår på grund af den anderledes underliggende datastruktur, som er baseret på syntakstræer under opbygning, på samme måde som i den beskrevne *breadth-first bottom-up* analysator. Der findes således ikke samme cykliske struktur som i Earleys algoritme, og som er grundlaget for at kunne håndtere den ubundne tvetydighed.

Dette har dog ingen betydning i den her påtænkte anvendelse, hvor kun bundet tvetydighed er relevant. Det skal dog haves i tankerne, når grammatikker udformes. Den mere hukommelseskrævende men implementeringsmæssigt mere simple taktik med én stak pr. sætningsform er desuden at foretrække i forhold til en senere udvidelse af algoritmen. Udvidelsen gør algoritmen i stand til også at håndtere attributter, som det er beskrevet i afsnit 4.6.

3.4.1 Repræsentation af sætningsformer - *items*

For som ønsket at kunne afgøre om nye sætningsformer, som dannes undervejs i analysen, er acceptable, dvs. om de også kan afledes fra startsymbolet, er det nødvendigt at have mere information om de enkelte sætningsformer end blot, hvilke symboler de består af. Derfor ændres de elementer som ligger i stakkene fra at være symboler til i stedet at være mængder af såkaldte *items*.

Et *item* er en regel fra grammatikken kombineret med en position iblandt højresidens symboler. Positionen markeres med en prik.

$$\langle NP \rangle ::= \langle NP \rangle \bullet \langle PP \rangle$$

Prikkens betydning er, at symbolerne til venstre for prikken er blevet genkendt, og symbolerne til højre for prikken forventes at blive genkendt efterfølgende. Et *item* angiver således, hvor i en given regel syntaksanalysen kan være nået til. En mængde af *items* viser tilsvarende hvor, og i hvilke regler syntaksanalysen kan være nået til.

Sætningsformerne repræsenteres altså nu af stakke, som indeholder mængder af *items*, hvor de før indeholdt symboler. Opgaven for de indgående *items* er at vise hvilke muligheder, der er til rådighed indenfor rammerne af en *top-down* tilgang.

Den mængde af *items*, som udgør det øverste element i en stak for en sætningsform, vil således altid indeholde oplysninger om hvor og i hvilke regler syntaksanalysen kan være nået til. Og vigtigst af alt: Hvilke symboler der kan forventes efterfølgende. Dette benyttes fx til at afgøre, om et nyt indlæst symbol kan accepteres i forlængelse af de allerede indlæste symboler.

Fx angiver

$$\langle PP \rangle ::= Prep \bullet \langle NP \rangle$$

at der netop er læst terminal-symbolet $Prep$, og at ikke-terminalsymbolet $\langle NP \rangle$ forventes efterfølgende. Da $\langle NP \rangle$ er et ikke-terminalt symbol, må grammatikken indeholde én eller flere regler med et $\langle NP \rangle$ på venstresiden. Et forventet $\langle NP \rangle$ kan derfor afledes til en anden symbolsekvens, hvis første

Item:

$$\langle PP \rangle ::= Prep \bullet \langle NP \rangle$$

Prædikterede items:

$$\langle NP \rangle ::= \bullet Noun$$
$$\langle NP \rangle ::= \bullet \langle NP \rangle \ \langle PP \rangle$$
$$\langle NP \rangle ::= \bullet Adj \ \langle NP \rangle$$

Figur 3.3: Eksempel på prædikterede *items*

symbol dermed også er blandt de symboler, som forventes efterfølgende. Af denne grund medtages et yderligere antal *items*, såkaldte *prædikterede items*, som genereres rekursivt ud fra de *items*, som allerede er indeholdt i mængden. Den færdige mængde af *items*, som kan ses i figur 3.3, beskriver således hvilke symboler - terminale som ikke-terminale - som forventes og kan accepteres efterfølgende. De prædikterede *items* er baseret på grammatikken i figur 3.2.

Bemærk at forandringen fra symboler til *items* i stakkene ikke ændrer på den overordnede *breadth-first bottom-up* analyse, men blot ændrer repræsentationen af de symboler som ligger i stakken.

3.4.2 *Shift* og *reduce* - igen

Algoritmen arbejder stadig med de to operationer *shift* og *reduce*, men selvfølgelig i en lidt anderledes form, da stakkene ikke længere består af symboler men af mængder af *items*.

Shift operationen skal ikke længere blot indskifte det nyindlæste symbol på alle stakkene, men skal for hver stak indskifte en individuel mængde af *items*, som er opbygget på baggrund af den enkelte stak og det indskiftede symbol. For hver stak gøres dette ved at sammenholde det nye symbol med det øverste stakelements mængde af *items*. Det nye symbol skulle jo gerne være iblandt de af stakken forventede symboler. Dette undersøges ved at sammenligne symbolet efter prikken i alle stakelementets *items* med det nye symbol. Derefter kopieres alle de matchende *items* over i en ny mængde og opdateres ved at flytte prikken et skridt mod højre, hvorefter der tilføjes prædikterede *items* som tidligere beskrevet.

Hvis det nye symbol ikke er blandt de forventede ifølge den på stakken øverste mængde af *items* kan det nye symbol således ikke accepteres ud fra en *top-down* tilgang. Det betyder, at den pågældende stak kan smides væk, da den alligevel aldrig vil kunne reduceres til startsymbolet. Derved frasorteres nogle af de uønskede sætningsformer, som fik lov at fortsætte i den almindelige *breadth-first bottom-up* analyse.

Reduce operationen, som stadig følger efter hver *shift*-operation, undersøger som før hver stak om reduktioner er mulige. Dette kan nu gøres ved blot at betragte den øverste mængde af *items* på stakken, for at se om prikken i nogle *items* er nået til enden af højresiden. Dette betyder nemlig, at en hel højreside er genkendt, hvorved der er basis for at reducere den genkendte højreside til symbolet i reglens venstreside. En reduktion betyder som før, at et eller flere af de øverste stakelementer fjernes og erstattes af et nyt. Forskellen er blot, at det nu gælder mængder af *items*, og ikke symboler.

Efter at de reducerbare øverste stakelementer er fjernet, findes den nye mængde af *items*, som skal erstatte dem på helt samme måde, som når et nyt symbol skal indskiftes på stakken i *shift* operationen. I dette tilfælde er der blot tale om et ikke-terminalt symbol. Ligesom i *shift* operationen kan det her ske, at det indskiftede symbol ikke er blandt de forventede. Dette medfører som i *shift* operationen, at stakken fjernes, da sætningsformen således ikke kan afledes fra startsymbolet. Ved indskiftningen kan der også opstå nye afsluttede *items*, som således kan danne basis for flere reduktioner, hvilket håndteres som i den almindelige *breadth-first bottom-up* analysator ved at kopiere stakke og sideløbende arbejde videre på de mulige sætningsformer.

3.4.3 Analysens forløb

Det overordnede forløb af algoritmen er ligesom for *breadth-first bottom-up* analysen uden *top-down* begrænsning. Et antal gentagne kald af *shift* og *reduce*, indtil alle symboler i den givne sætning er indlæst.

Startbetingelserne er dog anderledes, da der ikke lægges ud med en tom stak som før men med en initel mængde af *items*. Denne mængde findes ved at indføre et nyt startsymbol $\langle S_n \rangle$ og en ny regel, der afleder det nye startsymbol over i grammatikkens oprindelige startsymbol:

$$\langle S_n \rangle ::= \langle S \rangle$$

Den initielle *item*-mængde udgøres så af et *item* $\langle S_n \rangle ::= \bullet\langle S \rangle$ og de tilhørende prædikterede *items*. På den måde sikres fra start, at det første indlæste symbol også er afledeligt fra startsymbolet.

I modsætning til den almindelige *breadth-first bottom-up* analyse vil antallet af stakke/sætningsformer ikke udelukkende vokse igennem syntaksanalysen, da der er åbnet mulighed for at kassere ubrugelige løsninger undervejs. Derfor vil antallet af stakke bevæge sig op og ned, alt efter hvor mange reduktionsmuligheder der opstår, og hvor mange stakke der må droppes på baggrund af *top-down* begrænsningen.

I forbindelse med afslutningen af algoritmen spiller den specielt oprettede regel igen en væsentlig rolle, da øverste *item*-mængde i de stakke, som er nået hele vejen igennem algoritmen, undersøges for et *item* med udseendet:

$$\langle S_n \rangle ::= \langle S \rangle\bullet$$

Dette viser nemlig, at grammatikkens oprindelige startsymbol er blever genkendt, og dermed at det har været muligt at reducere den givne sætning til startsymbolet. Ligesom før kan der for tvetydige grammatikker være flere stakke med det pågældende *item*, som udtryk for at der findes flere forskellige syntakstræer for den analyserede sætning.

3.5 Syntaksanalyse af attributgrammatikker

Syntaksanalyse for attributgrammatikker er ikke nødvendigvis anderledes end for grammatikker uden attributter. Målet kan i begge tilfælde være at få opbygget et syntakstræ, der forbinder en given sætning med grammatikken. At der så for attributgrammatikkens vedkommende efterfølgende udføres nogle beregninger på baggrund af træets indhold og opbygning har ikke nødvendigvis nogen betydning i selve syntaksanalysen. Forskellen opstår først når attributterne inddrages direkte i syntaksanalysen.

Attributterne kan inddrages ved, at attributværdierne beregnes løbende i takt med at syntakstræet opbygges. Det kræver selvfølgelig, at syntaksanalysatoren opbygger syntakstræet på en måde som harmonerer med de

forskellige attributters afhængigheder af hinanden. Fx vil en *breadth-first top-down* analyse ikke fungere specielt godt med den tidligere bekskrevne attributgrammatik fra figur 2.6, som udelukkende sender værdier opad i parsetræet. Udgangspunktet for attributberegningerne er således bunden af syntakstræet, men syntaksanalysatoren opbygger træet oppefra.

Hvis attributværdierne lader sig beregne i takt med, at syntakstræet opbygges, vil man ved en afsluttet syntaksanalyse ikke kun stå med et syntakstræ men også med de beregnede attributværdier. Derudover er det muligt at give den semantiske udbygning - som attributterne udgør - indflydelse på analysens forløb. Dette kunne fx være som en begrænsende faktor i forhold til, hvilke sætninger der ønskes at gennemføre en analyse for. For den udvidede attributgrammatik for simple summer i figur 2.8 kunne man forestille sig, at man kun ønskede at genkende sætninger, hvis semantiske værdi var en-cifret. Da der for hver knude i syntakstræerne findes en attribut, som angiver det hidtil beregnede resultat, er dette blot et spørgsmål om at holde øje med denne attributs værdi. Bliver værdien større end 9, afbrydes syntaksanalysen for dette deltræ. Denne mekanisme anvendes senere i den beskrevne tekstanalysator, som en semantisk begrænsning for de sætninger der genkendes.

Kapitel 4

Tekstanalysatoren

I dette kapitel beskrives opbygningen af teksatanalysatoren samt de involverede BNF-grammatikker. Først gives et kort overblik over tekstanalysatoren hvorefter, den ontologiske modellering i form af den begrebsmæssige grammatik beskrives.

Derefter beskrives den lingvistiske grammatik samt hvordan, den udstyret med attributter bliver i stand til at håndtere den semantiske information. Desuden defineres et antal basisfunktioner, som benyttes til at beskrive attributfunktionerne for de lingvistiske regler.

Til sidst beskrives hvordan, den modificerede udgave af Earleys algoritme modificeres yderligere, for at understøtte den beskrevne funktionalitet.

4.1 Overblik

Tekstanaysatorens opgave er på baggrund af en sætning med angivelse af ordklasser, at finde nul, et eller flere bud på det semantiske indhold i sætningen i form af de såkaldte beskrivere. Eksempler på ind- og uddata for tekstanalysatoren kunne være:

mangel{Noun} på{Prep} C-vitamin{Noun}

$$\downarrow$$

$$\langle Lack \rangle [\langle WRT \rangle : \langle VitaminC \rangle]$$

symptom{Noun} på{Prep} mangel{Noun} på{Prep} C-vitamin{Noun}

$$\downarrow$$

$$\langle Symptom \rangle [\langle CBY \rangle : \langle Lack \rangle [\langle WRT \rangle : \langle VitaminC \rangle]]$$

En beskriver er en sætningsform i en BNF-grammatik, kaldet den begrebsmæssige grammatik. Den begrebsmæssige grammatik repræsenterer en ontologi for et videndomæne, og tilbyder på den måde en struktur som den betydning, der hentes ud af sætningerne, kan hænges op på.

Ud over den begrebsmæssige grammatik arbejder tekstanalysatoren med endnu en BNF-grammatik kaldet den lingvistiske grammatik. Den lingvistiske grammatik spiller en mere direkte rolle i syntaksanalysen, da den benyttes til at forstå sætningernes sproglige opbygning. I tekstanalysatoren benyttes grammatikken som udgangspunkt for en traditionel syntaksanalyse ved hjælp af den modificerede version af Earleys algoritme. Da tekstanalysen også skal omfatte opbygningen af bud på sætningens semantiske indhold, udstyres grammatikken med attributter til håndtering af den semantiske information, som opbygges undervejs i tekstanalysen i form af beskrivere.

Helt overordnet kan man sige, at den lingvistiske grammatik anvendes som udgangspunkt for at forstå de indlæste sætninger, og den begrebsmæssige grammatik anvendes til at beskrive deres semantiske indhold.

4.2 Den begrebsmæssige grammatik

Den begrebsmæssige grammatik er en BNF-grammatik, som på én gang udgør en beskrivelse af en ontologi for et videndomæne samtidig med, at den anvendes til at generere sætningsformer, som udgør de omtalte beskrivere, som afspejler meningen af indlæste sætninger. Grammatikken anvendes således både som en definition af videndomænets struktur og til at beskrive den betydning, som de analyserede sætninger tillægges.

Ontologien specificerer de væsentlige begreber indenfor et videndomæne samt tilladte relationer imellem dem. Begreberne knyttes primært sammen

af den taxonomiske ISA relation, som afspejler begrebsmæssig inklusion. Relationen afspejler, at et begreb er en generalisering af et andet begreb, og omvendt at det andet begreb er en specialisering af det første. Fx er begrebet *C-vitamin* en specialisering af begrebet *vitamin*, og omvendt er *vitamin* en generalisering af begrebet *C-vitamin*. ISA relationen knytter således alle de centrale begreber indenfor det aktuelle videndomæne sammen og placerer dem i en endelig struktur, der afspejler en klassificering af begreberne.

I den begrebsmæssige grammatik er de enkelte begreber modelleret ved ikke-terminale symboler, og begrebsinklusionen er repræsenteret ved hjælp af reglerne i grammatikken. Grammatikkens regler afspejler på den måde direkte den ontologiske opbygning.

Fx som nedenfor:

$$\langle UNIV \rangle ::= \langle CONCR \rangle \mid \langle OCCUR \rangle \mid ...$$
$$\langle CONCR \rangle ::= \langle STUFF \rangle \mid ...$$
$$\langle STUFF \rangle ::= \langle Vitamin \rangle \mid ...$$
$$\langle Vitamin \rangle ::= \langle VitaminA \rangle \mid \langle VitaminB \rangle \mid \langle VitaminC \rangle \mid ...$$

Hver regel svarer til brug af ISA relationen, hvor grammatikkens startsymbol svarer til det mest generelle begreb. En regel, der angiver at et begreb kan afledes over i et andet, afspejler således, at begrebet på højresiden udgør en specialisering af begrebet på venstresiden. Betydningen af de viste regler er bl.a., at kategorien $\langle Vitamin \rangle$ udgør en specialisering af kategorien $\langle STUFF \rangle$. Ligeledes er $\langle VitaminA \rangle$, $\langle VitaminB \rangle$ og $\langle VitaminC \rangle$ alle specialiseringer af $\langle Vitamin \rangle$, men forskellige da de er placeret som alternativer til hinanden.

Hvert begreb er således tilknyttet et antal generaliseringer af begrebet og et antal specialiseringer af begrebet. Det mest generelle begreb i ontologien er selvfølgelig kun tilknyttet specialiseringer og udgøres som nævnt af startsymbolet. De mest specialiserede begreber er omvendt kun tilknyttet generaliseringer. Disse begreber afspejler typisk konkrete ord som fx C-vitamin.

Den begrebsmæssige struktur, som kan opbygges på denne baggrund, kaldes for skeletontologien.

Afledningen af et begreb fra startsymbolet afspejler en sti ned igennem ontologien, hvilket igen afspejles af det tilhørende syntakstræ. Syntakstræerne

består således blot af en linie fra startsymbolet, der passerer mere og mere specialiserede begreber. Dette hænger sammen med, at skeletontologien dannes af regler, som kun indeholder afledninger over i netop ét symbol. Et eksempel på en sådan afledning ses nedenfor.

$$\langle UNIV \rangle$$
$$\downarrow$$
$$\langle CONCR \rangle$$
$$\downarrow$$
$$\langle STUFF \rangle$$
$$\downarrow$$
$$\langle Vitamin \rangle$$
$$\downarrow$$
$$\langle VitaminC \rangle$$

En begrebsmæssig grammatik, der afspejler en skeletontologi, må ikke være rekursiv, da grammatikken derved ikke afspejler begrebsinklusionen som ønsket.

Grammatikken må derimod gerne indeholde bundet tvetydighed, da det kan afspejle polysemi. Udvides de viste regler for eksempel med

$$\langle STUFF \rangle ::= \langle Diet \rangle$$
$$\langle OCCUR \rangle ::= \langle Diet \rangle$$

afspejles det således, at begrebet $\langle Diet \rangle$ har forskellige generaliseringer, begrundet i at begrebet kan opfattes på forskellige måder i forskellige sammenhænge.

Ubundet tvetydighed opstår ikke, da der ingen rekursion må være, og da regler der afledes over i den tomme streng ikke giver mening i forhold til den begrebsmæssige moddellering.

4.2.1 Semantiske relationer

Den begrebsmæssige grammatik skal ikke kun være i stand til at håndtere enkeltstående begreber, men også sammenhænge imellem dem som de forekommer i de sætninger, som er målet for tekstanalysen.

Derfor udvides grammatikken til også at kunne beskrive en attributtering af begreberne. Dette gøres ved at udbygge grammatikken med semantiske relationer, som kobles på begreberne i skeletontologien.

De semantiske relationer består i muligheden for at tilknytte et begreb i skeletontologien et eller flere rolle-værdi par, som angiver henholdsvis relationens art, samt hvilket andet begreb relationen knytter sig til.

Relationens art eller rolle repræsenteres i grammatikken ved nogle ikke terminale symboler, som ikke er en del af skeletontologien, men som udelukkende repræsenterer de forskellige typer af relationer, som kan forekomme i attributteringen. Nedenfor ses eksempler på roller, som afspejler nogle almindelige relationer, der vil blive anvendt i det følgende.

Rolle	Beskrivelse
WRT	med hensyn til (with respect to)
CHR	karakteriseret ved (characteristic)
CBY	skyldes (caused by)

Udvides de tidligere viste regler med semantiske relationer kunne de få følgende udseende :

$$\langle OCCUR\rangle ::= \langle STATE\rangle \ [\ \langle CBY\rangle : \langle OCCUR\rangle\ ]$$
$$\langle STATE\rangle ::= \langle Lack\rangle \ [\ \langle WRT\rangle : \langle CONCR\rangle\ ] \mid \langle Symptom\rangle$$

De angivne rolle-værdi par skal betragtes som en mulig specialisering af de tilhørende begreber og ikke som en obligatorisk udvidelse. Det vil sige, at betydningen af den første af reglerne egentlig er:

$$\langle OCCUR\rangle ::= \langle STATE\rangle \mid \langle STATECBY\rangle$$
$$\langle STATECBY\rangle ::= \langle STATE\rangle \ [\ \langle CBY\rangle : \langle OCCUR\rangle\ ]$$

Rolle-værdi parret skal i dette tilfælde opfattes som obligatorisk. Bemærk at de firkantede parenteser også optræder som terminale symboler, og ikke kun angiver optionaliteten.

Hvad der også afspejles er, at en tilknyttet semantisk relation tjener som en specialisering af begreber, og bemærk at de mulige semantiske relationer

$$\langle UNIV\rangle$$
$$\downarrow$$
$$\langle OCCUR\rangle$$
$$\downarrow$$
$$\langle STATE\rangle \ [\ \langle CBY\rangle : \langle OCCUR\rangle\ ]$$
$$\downarrow$$
$$\langle Symptom\rangle \ [\ \langle CBY\rangle : \langle OCCUR\rangle\ ]$$
$$\downarrow$$
$$\langle Symptom\rangle \ [\ \langle CBY\rangle : \langle STATE\rangle\ ]$$
$$\downarrow$$
$$\langle Symptom\rangle \ [\ \langle CBY\rangle : \langle Lack\rangle \ [\ \langle WRT\rangle : \langle CONCR\rangle\ ]\ ]$$
$$\downarrow$$
$$\langle Symptom\rangle \ [\ \langle CBY\rangle : \langle Lack\rangle \ [\ \langle WRT\rangle : \langle STUFF\rangle\ ]\ ]$$
$$\downarrow$$
$$\langle Symptom\rangle \ [\ \langle CBY\rangle : \langle Lack\rangle \ [\ \langle WRT\rangle : \langle Vitamin\rangle\ ]\ ]$$
$$\downarrow$$
$$\langle Symptom\rangle \ [\ \langle CBY\rangle : \langle Lack\rangle \ [\ \langle WRT\rangle : \langle VitaminC\rangle\ ]\ ]$$

Figur 4.1: Eksempel på afledning af beskriver

nedarves af specialiserede begreber. De semantiske relationer giver således i form af *nesting* mulighed for uendelige specialiseringer i form af uendligt mange forskellige sætningsformer. Grammatikken får derved den ønskede generative evne ved hjælp af de semantiske relationer. Rekursion er således ønskelig så længe den ikke findes i skeletontologien.

Da rekursionen således udspringer af de semantiske relationer opstår der stadig ikke ubundet tvetydighed i grammatikken, da enhver rekursiv afledning ledsages af terminale symboler i form af de firkantede parenteser.

4.2.2 Afledning af beskrivere

Et konkret eksempel på anvendelse er fx sætningen symptom på mangel på C-vitamin, hvis mening kan repræsenteres af sætningsformen eller beskriveren $\langle Symptom\rangle \ [\ \langle CBY\rangle : \langle Lack\rangle \ [\ \langle WRT\rangle : \langle Vitamin\rangle\ ]\ ]$. Afledningen af beskriveren er gjort med udgangspunkt i de tidligere viste regler, og kan ses i figur 4.1.

I tekstanalysatoren er det netop beskrivere som denne, der ønskes fundet. Udgangspunktet for at gøre dette er de ord, der forekommer i de givne sætninger. Ordet symptom skal føre til begrebet $\langle Symptom \rangle$, ordet mangel skal føre til begrebet $\langle Lack \rangle$ etc. For at gøre dette har tekstanalysatoren behov for at kunne knytte konkrete ord til begreber.

Denne sammenhæng placeres i den begrebsmæssige grammatik ved at lade de terminale symboler i grammatikken være ord i stedet for de mest specialiserede begreber. Ordene bliver så tilknyttet deres tilhørende begreber ved hjælp af regler som:

$\langle Symptom \rangle ::= "symptom"$
$\langle VitaminA \rangle ::= "A\text{-}vitamin"$
$\langle CBY \rangle ::= "på" \mid "skyldes"$

Det er ikke meningen, at alle variationer, bøjningsformer og synonymer af et ord skal være repræsenteret, da disses sammenhæng med det tilsvarende begreb er baseret på opslag i en ordbog, før sætningen forsøges analyseret. De sætninger, som tages ind til analyse, er således renset for bøjninger etc.

Et begreb bør således ikke tilknyttes flere end ét ord. Modsat gælder der en mange til mange relation imellem de semantiske roller og deres tekstuelle former.

Alle de begreber, som via en regel i grammatikken er tilknyttet et konkret ord, dvs. et terminalt symbol, kaldes for terminale begreber, da de afspejler begreber, som i kraft af de tilknyttede ord kan optræde direkte i de sætninger, som indlæses til analyse og dermed også i beskrivere.

De terminale begreber er også dem, som optræder i afledte beskrivere.

De terminale symboler, i form af ordene som de forekommer i de indlæste sætninger, benyttes således kun til at forbinde ordene med de relevante terminale begreber.

For den begrebsmæssige grammatik skelnes der således imellem de terminale *symboler* og de terminale *begreber*.

Bemærk at et terminalt begreb ikke nødvendigvis er et fuldt specialiseret begreb, men godt kan være en generalisering af et eller flere andre begreber. Fx som begrebet $\langle Vitamin \rangle$, der både er en generalisering af de forskellige typer vitamin, ligesom det afspejler ordet *"vitamin"*.

organisk C-vitamin
$\downarrow$
$Vitamin\ [\ CHR : Organic\ ]$

mangel på vitamin
$\downarrow$
$Lack\ [\ WRT : Vitamin\ ]$

sygdom skyldes mangel på vitamin
$\downarrow$
$Disease\ [\ CBY : Lack\ [\ WRT : Vitamin\ ]\ ]$

symptom på mangel på organisk C-vitamin
$\downarrow$
$Symptom\ [\ CBY : Lack\ [\ WRT : VitaminC\ [\ CHR : Organic\ ]\ ]\ ]$

Figur 4.2: Eksempler på beskrivere

4.2.3 Udformning af en begrebsmæssige grammatik

En begrebsmæssig grammatik opstilles med henblik på et bestemt videndomæne, som har tilknytning til de sætninger, som ønskes analyseret. I OntoQuery forskningsprojektet er der hidtil blevet arbejdet med videndomænet ernæring, hvilket også er benyttet i de angivne eksempler.

Trods grammatikkens skræddersyede karakter vil den mest generelle del af grammatikken i nogen grad være uafhængig af, hvilket videndomæne der modelleres.

Der vil desuden findes generelle måder at modellere visse konstruktioner på. Fx vil Xmangel blive til $\langle Lack \rangle\ [\ \langle WRT \rangle : \langle X \rangle\ ]$.

De sætningstyper, som der grundlæggende ønskes at arbejde, er navneled og udsagnsled med adjektiver og præpositioner. Eksempler på disse vises i figur 4.2. I dette og de følgende eksempler udelades vinkelparenteserne omkring de ikke-terminale symboler for overskuelighedens skyld.

Udsagnsord, præpositioner og adjektiver kan håndteres ved hjælp af semantiske relationer, som det fremgår. Bemærk at de semantiske roller ikke nødvendigvis har en tekstuel repæsentation i den givne sætning. Beskriveren for organisk C-vitamin benytter en rolle, som udelukkende er baseret på den sproglige konstruktion. Rollen CHR knytter sig i det her tilfælde

$UNIV ::= CONCR \mid OCCUR \mid PROPERTY$
$CONCR ::= STUFF\ [\ CHR : PROPERTY\ ]$
$OCCUR ::= STATE\ [\ CBY : OCCUR\ ]$
$PROPERTY ::= Organic$
$STATE ::= Lack\ [\ WRT : CONCR\ ]\ \mid Symptom \mid Disease$
$STUFF ::= Vitamin$
$Vitamin ::= VitaminA \mid VitaminB \mid VitaminC \mid$ "$vitamin$"
$Symptom ::=$ "$symptom$"
$Disease ::=$ "$sygdom$"
$Lack ::=$ "$mangel$"
$Organic ::=$ "$organisk$"
$VitaminA ::=$ "A-$vitamin$"
$VitaminB ::=$ "B-$vitamin$"
$VitaminC ::=$ "C-$vitamin$"
$WRT ::=$ "$på$"
$CBY ::=$ "$på$" | "$skyldes$"
$CHR ::=$ "_"

Figur 4.3: Eksempel på begrebsmæssig grammatik

kun til, at der er tale om et navneord tilknyttet et adjektiv og ikke til et bestemt ord i sætningen.

I det følgende vil der i eksempler og forklaringer blive benyttet den begrebsmæssige grammatik, som er vist i figur 4.3. Grammatikken er en udbygning af de tidligere viste regler og eksempler.

Bemærk at rollen CHR ikke er tilknyttet noget specielt ord, men alligevel har sin berettigelse, da rollen kan anvendes i forbindelse med adjektiver, som beskrevet for sætningen organisk C-vitamin.

4.2.4 Begrebsmæssig modellering i LKB

Som tidligere beskrevet kan LKB-systemet også anvendes i forbindelse med en begrebsmæssig modellering af et videndomæne.

De strukturer, som en *typed feature structure* modellerer i LKB-systemet, synes umiddelbart ikke at ligge så langt fra den ontologiske struktur, der beskrives af den begrebsmæssige grammatik: Et hierarki af begreber, som kan kombineres i form af nedarvede semantiske relationer, der til forveksling

ligner de rolle-værdi par, som indgår i *typed feature structures*. En væsentlig forskel er dog, at for *typed feature structures* er rolle-værdi parrene obligatoriske som restriktioner, hvorimod de i den begrebsmæssige grammatik fungerer som mulige specialiseringer. En begrebsmæssig modellering, som den der i dette tilfælde er ønskelig, kan således ikke direkte repræsenteres ved *typed feature structures* i LKB, men må også baseres på yderligere type-definitioner til håndtering af valgfriheden mht. mulige rolle-værdi par.

Dette komplicerer selvsagt notationen for en begrebsmæssig modellering i LKB, som i forvejen har en syntaktisk tung repræsentation af den øvrige grammatiske information.

4.3 Den lingvistiske grammatik

Den lingvistiske grammatik anvendes til at forstå den grammatiske opbygning af de sætninger, som ønskes analyseret. Reglerne i grammatikken skal derfor afspejle den grammatiske opbygning af de sætningskonstruktioner, som ønskes indfanget til semantisk analyse af tekstanalysatoren. Der vil i den forbindelse ikke blive forsøgt en fuldstændig lingvistisk analyse af sætningerne, hvorfor den beskrevne grammatik vil være stærkt abstraheret.

4.3.1 Ordklasser

Udgangspunktet for at grammatikken kan analysere den grammatiske opbygning af en sætning er ordklasserne for de ord, som optræder i sætningen. Fx substantiv, præposition, adjektiv, navneord etc. De forskellige ordklasser optræder derfor som de terminale symboler i den lingvistiske grammatik, ligesom de er en del af de sætninger, som indlæses. Angivelsen af ordklasser for en sætning er ligesom filtreringen for bøjningsformer baseret på ordbogsopslag, som ligger udenfor den her bekrevne tekstanalysators arbejdsområde.

Angivelsen af ordklasser er ikke nødvendigvis entydig, hvorfor tekstanalysatoren også må tage højde for tvetydige angivelser af ordklasser, som fx for ordet have, der både kan være et navneord og et substantiv.

4.3.2 Grammatikkens udformning

Som tidligere nævnt er målet i dette tilfælde ikke en tilbundsgående lingvistisk analyse af teksten, som vil være uhyre kompliceret. Målet er i stedet en analyse baseret på en forenklet naturligtsprogs-grammatik, der sigter efter de tidligere nævnte konstruktioner; navneled og udsagnsled med adjektiver og præpositioner. Som beskrevet kan disse konstruktioner også håndteres af den opstillede begrebsmæssige grammatik, som ligger bag den semantiske analyse. Udformningen af den her anvendte lingvistiske grammatik tager derfor udgangspunkt i de samme konkrete sætninger, som eksemplificerer de relevante sætningskonstruktioner.

I dette tilfælde lægges vægten selvfølgelig på ordklasserne:

organisk{Adj} C-vitamin{Noun}

mangel{Noun} på{Prep} vitamin{Noun}

sygdom{Noun} skyldes{Verb} mangel{Noun} på{Prep} vitamin{Noun}

symptom{Noun} på{Prep} mangel{Noun} på{Prep} organisk{Adj} C-vitamin{Noun}

Som udgangspunkt for grammatikken opstilles følgende regler til håndtering af præpositioner som fx mangel{Noun} på{Prep} C-vitamin{Noun}.

$$
\begin{aligned}
&S ::= NP \\
&NP ::= NP \;\; PP \mid Noun \\
&PP ::= Prep \;\; NP
\end{aligned}
$$

NP står for *Noun Phrase*, PP for *Preposition Phrase* og $Prep$ for *Preposition*. Som det ses, gør de opstillede regler grammatikken stærkt tvetydig. Tvetydigheden er tilsigtet og afspejler, hvordan sætningens indhold kan være struktureret. Hvilke af mulige syntakstræer, der bedst beskriver indholdet afgøres senere af tekstanalysatoren på baggrund af, hvordan den begrebsmæssige grammatik er udformet. Hvordan dette foregår, uddybes senere.

På helt tilsvarende vis udvides grammatikken til også at håndtere udsagnsled. Dette resulterer i følgende grammatik:

$$
\begin{aligned}
&S ::= NP \\
&NP ::= NP \;\; PP \mid NP \;\; VP \mid Noun \\
&PP ::= Prep \;\; NP \\
&VP ::= Verb \;\; NP
\end{aligned}
$$

Grammatikken kan nu håndtere fx sætningen sygdom{Noun} skyldes{Verb} mangel{Noun} på{Prep} vitamin{Noun}. Udsagnsled forekommer dog ofte i forbindelse med et relativt pronomen, som i sygdom{Noun} der{RelPron} skyldes{Verb} mangel{Noun} på{Prep} vitamin{Noun}, hvilket giver anledning til, at den sidste regel udvides til følgende:

$$S ::= NP$$
$$NP ::= NP \;\; PP \mid NP \;\; VP \mid Adj \;\; NP \mid Noun$$
$$PP ::= Prep \;\; NP$$
$$VP ::= Verb \;\; NP \mid RelPron \;\; Verb \;\; NP$$

Figur 4.4: Lingvistisk grammatik

$$VP ::= Verb \;\; NP \mid RelPron \;\; Verb \;\; NP$$

Adjektiver håndteres ved hjælp af en simpel udvidelse, der blot gør det muligt at sætte et Adj foran en NP. Derved fremkommer den lingvistiske grammatik, som ses i figur 4.4.

Også her er der tale om en stærk tvetydighed, der har forbindelse med hvilke dele af en sætning adjektivet tilknyttes. I sætningen langvarig{Adj} sygdom{Noun} skyldes{Noun} mangel{Noun} på{Prep} A-vitamin{Noun} vil adjektivet ifølge grammatikken således både kunne knyttes til ordet sygdom og til hele resten af sætningen, hvilket ikke giver nogen mening. Igen vil afgørelsen om, hvilken konstruktion der kan bruges, blive taget af tekstanalysatoren med udgangspunkt i, hvilke konstruktioner den begrebsmæsige grammatik tillader.

Som det ses er grammatikken stærkt abstraheret i forhold til en "ægte" naturligsprogs-grammatik for dansk. Bl.a. tages der ikke hensyn til artikler, og mere avancerede sætningskonstruktioner. Det er således ikke al den lingvistiske information, som er til stede i sætningerne, der udnyttes, men at opbygge en grammatik der udnytter informationen til fulde er en opgave for lingvister og ikke for ingeniører. At tekstanalyse er en yderst kompliceret opgave illustreres af eksemplet: på Jupiters måner mener videnskaben nu at der er fundet vand. At få knyttet sætningens sidste ord vand til de tre første ord, som det meningsmæssigt hænger sammen med, er selvsagt ikke trivielt.

4.3.3 Tilsigtet tvetydighed

Som beskrevet indeholder den lingvistiske grammatik i høj grad bundet tvetydig, hvilket ikke er tilfældigt. Tvetydigheden anvendes, som tidligere antydet, til at nå alle de måder sætningens meningsindhold kunne være struktureret på. Forventningen er så, at de fleste kan sorteres fra, på bag-

grund af hvilke meningskonstruktioner der er tilladelige ifølge den begrebsmæssige grammatik.

Betragt sætningen symptom{Noun} på{Prep} mangel{Noun} på{Prep} C-vitamin{Noun}. Sætningen har to gyldige syntakstræer i forhold til den sproglige grammatik, som det ses i figur 4.5. Syntakstræerne afspejler hver sin strukturering af sætningens indhold. Overføres denne strukturering, til hvordan beskrivere for sætningen kunne se ud, fås følgende to muligheder.

$$Symptom\ [\ CBY : Lack\ [\ WRT : VitaminC\ ]\ ]$$
$$Symptom\ [\ CBY : Lack\ ]\ [\ WRT : VitaminC\ ]$$

Tager man derimod udgangspunkt i den begrebsmæssige grammatik, der blev opstillet i figur 4.3, er det kun den første, der er mulig igennem afledningen, der blev vist i figur 4.1.

Den anden beskriver er derimod ikke gyldig i forhold til den begrebsmæssige grammatik, da begrebet $Symptom$ ikke kan tilknyttes en semantisk relation som $[WRT : Vitamin]$.

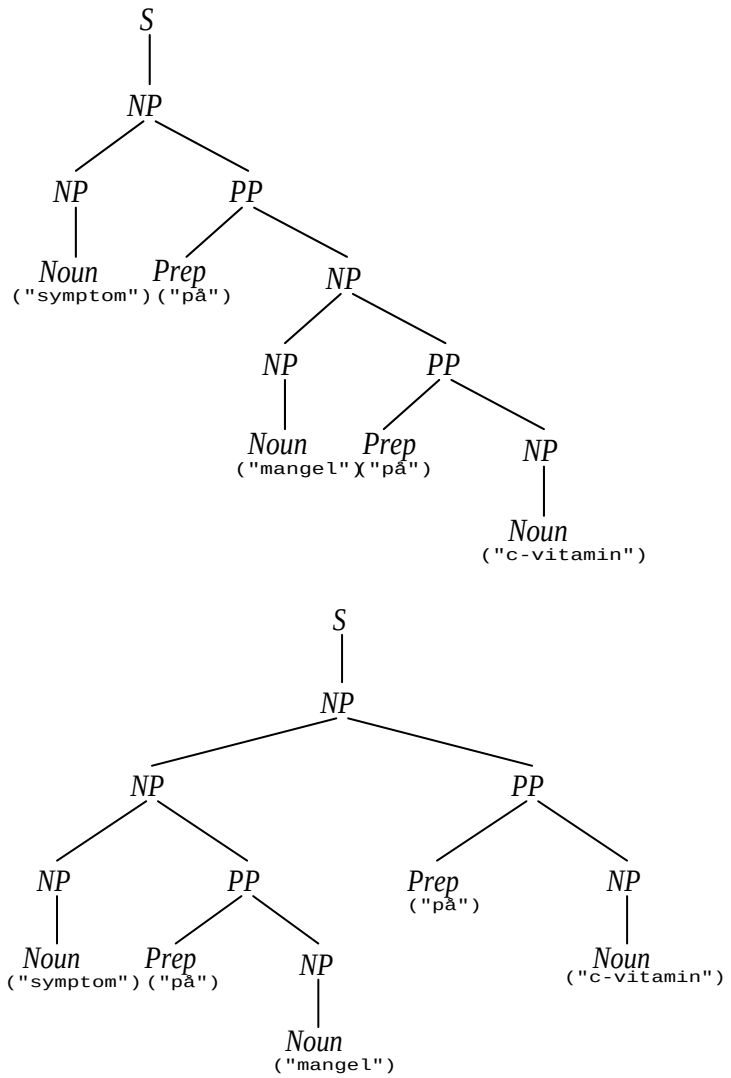
Tvetydigheden i den lingvistiske grammatik anvendes til, at nå frem til en masse bud på sætningens lingvistiske opbygning. Tvetydigheden løses så på baggrund af, hvilke semantiske konstruktioner, der er mulige ifølge den begrebsmæssige grammatik. Det er dog ikke udelukket, at en sætning der er tvetydig i forhold til den lingvistiske grammatik, kan give anledning til flere end én beskrivere. Igen afgøres det af, hvilke semantiske konstruktioner den begrebsmæssige grammatik tillader.

Da den lingvistiske grammatik skal danne grundlag for en syntaksanalyse på baggrund af den tidligere beskrevne modificerede version af Earleys algoritme, må grammatikken ikke indeholde ubundet tvetydighed.

4.4 Attributtering

Som det fremgår i forrige afsnit, er der en forbindelse imellem de to grammatikker, da beskriverne i den begrebsmæssige grammatik dannes med udgangspunkt i sætningernes lingvistiske opbygning.

I tekstanalysatoren er denne forbindelse sat i system ved at udstyre den lingvistiske grammatik med to attributter. Attributterne anvendes til hånd-



Figur 4.5: Syntakstræer for symptom{Noun} på{Prep} mangel{Noun} på{Prep} C-vitamin{Noun}

tering af den semantiske information, som opbygges i form af beskrivere med udgangspunkt i den begrebsmæssige grammatik.

Attributternes opgave er, at lede mængder af mulige beskrivere rundt i de lingvistiske syntakstræer.

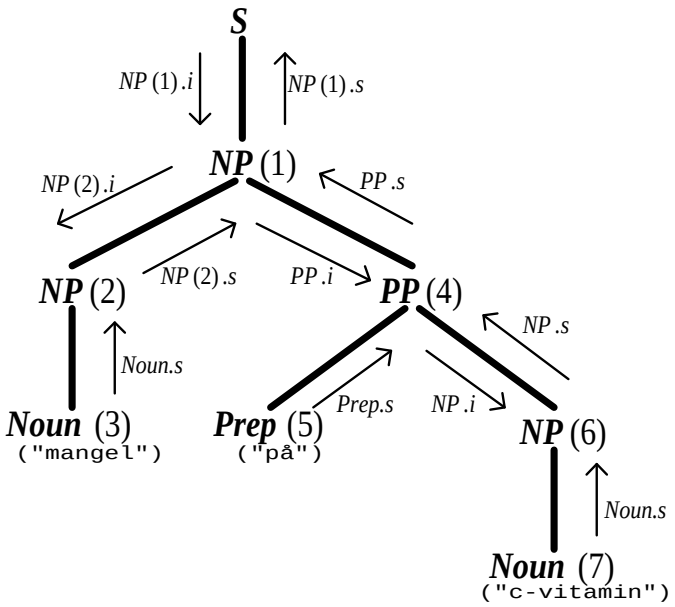
Attributteringen har samme struktur som attributgrammatikken i figur 2.8. Det vil sige, at der hele tiden sendes en værdi med en mængde mulige beskrivere nedad i syntakstræet, og på den baggrund returneres en opdateret mængde opad i syntakstræet tilsvarende eksemplet i figur 2.11, der i stedet rundsender en opdateret sum.

Udgangspunktet for de værdier, der syntetiseres opad i syntakstræet, er de værdier, der syntetiseres fra de terminale symboler i grammatikken. For grammatikken i figur 2.8 var det de genkendte tals værdi, som blev sendt opad fra de terminale symboler. Syntetiserede værdier fra de terminale symboler var således af samme type som de syntetiserede værdier fra ikke-terminale symboler. For den lingvistiske grammatik er dette anderledes, da de ikke-terminale symboler som nævnt syntetiserer mængder af beskrivere. De terminale symboler syntetiserer derimod de konkrete ord, som forekommer i sætningen.

I det følgende gennemgås attributværdierne for knuderne i det lingvistiske syntakstræ for sætningen mangel{Noun} på{Prep} C-vitamin{Noun}, som er vist i figur 4.6. Værdierne forklares i den rækkefølge de dannes ved et *preorder traversal* af syntakstræet. De to attributter kaldes som tidligere i og s for henholdsvis den nedarvede og den syntetiserede værdi.

Indledningsvis er venstrekonteksten tom, hvilket principielt gør mængden af mulige beskrivere uendelig stor. Da dette selvsagt ikke er et anvendeligt udgangspunkt, findes blot de beskrivere, som er mulige på baggrund af skeletontologien. Det vil sige de terminale begreber, som kan afledes fra startsymbolet uden, at de mulige semantiske relationer tages i brug. De nedarvede mulige semantiske relationer vedhæftes dog, som et udtryk for hvilke semantiske relationer det er muligt at udvide det tilhørende begreb med. Dette benyttes længere fremme i analysen til opbygning af den/de endelige beskrivere. Denne operation anvendes også senere under betegnelsen *HeadDerive*.

På baggrund af den begrebsmæssige grammatik fås derfor følgende indledende mængde:



Figur 4.6: Syntakstræ og attributter for mangel{Noun} på{Prep} C-vitamin{Noun}

$$
\begin{aligned}
(1)NP.i = \{ & \\
& (Lack,\ [\ CBY : OCCUR\ ]\ [\ WRT : CONCR\ ]), \\
& (Symptom,\ [\ CBY : OCCUR\ ]), \\
& (Disease,\ [\ CBY : OCCUR\ ]), \\
& (VitaminA,\ [\ CHR : PROPERTY\ ]), \\
& (VitaminB,\ [\ CHR : PROPERTY\ ]), \\
& (VitaminC,\ [\ CHR : PROPERTY\ ]), \\
& (Vitamin,\ [\ CHR : PROPERTY\ ]), \\
& (Organisk, \_) \\
\}
\end{aligned}
$$

Som det ses, består elementerne i mængden af to dele: En beskriver og de semantiske relationer som det er tilladt at udvide beskriveren med jvf. den begrebsmæssige grammatik.

Mængden er udgangspunktet for, hvilke beskrivere der forventes at kunne genkendes. Mængden sendes videre ned til (2) uden ændringer, da der ikke er fremkommet yderligere information, som kune give anledning til at ændre den.

$$(2)NP.i = (1)NP.i$$

Ved (2) nås det første terminale symbol og dermed kontakt med sætningens indhold. Fra $Noun$ syntetiseres således ordet "$mangel$".

$$(3)Noun.s = \text{"}mangel\text{"}$$

På den baggrund kan den mængde $(2)NP.i$, som blev nedarvet, nu opdateres. Dette gøres ved at sammenholde de til rådighed værende attributværdier $(2)NP.i$ og $(3)Noun.s$, og finde frem til hvilken mængde der skal sendes opad i syntakstræet som $(2)NP.s$. Hvordan denne mængde findes, er individuelt for hver regel i den lingvistiske grammatik og afgøres af de attributfunktioner, som er tilknyttet den anvendte regel. Hvordan, attributfunktionerne for de forskellige regler er opbygget, uddybes senere. I dette her tilfælde er der blot tale om et genkendt navnord, som forventes at være tilknyttet et af de mulige terminale begreber. Værdien af $(2)NP.s$ bliver derfor:

$$(2)NP.s = fa_{NP::=Noun}((2)NP.i, (3)Noun.s) = \{$$
$$(Lack,\ [\ CBY : OCCUR\ ]\ [\ WRT : CONCR\ ])$$
$$\}$$

$fa_{NP::=Noun}$ henviser i dette her tilfælde til en ikke nærmere defineret attributfunktion for reglen $NP ::= Noun$.

Sætningens semantiske indhold i form af ordet *"mangel"* medvirker således til at finde frem til den/de gyldige beskrivere for sætningen. Den syntetiserede mængde sendes uforandret videre til $(4)PP.i$:

$$(4)PP.i = (2)NP.s$$

Fra $(5)Prep$ syntetiseres det ord der udgør præpositionen:

$$(5)Prep.s = "på"$$

På dette sted i syntakstræet er vi ifølge venstrekonteksten midt i at genkende et præpositionsled(PP). Vi har en beskriver, som afspejler det navneled, som præpositionsledet skal knytte sig til $(4)PP.i$, og vi har selve præpositionen $(5)Prep$. Det giver tilsammen nogle afgrænsede muligheder for, hvad det efterfølgende navneled $(6)NP$ må indeholde.

Præpositioner repræsenteres i den begrebsmæssige grammatik ved semantiske relationer. Derfor består afgrænsningen i at finde ud af, hvilke semantiske relationer det er muligt at knytte til det navneled, som ér genkendt og indeholdt i $(4)PP.i$. Som det ses, er der i dette tilfælde to muligheder: $[\ CBY : OCCUR\ ]$ og $[\ WRT : CONCR\ ]$. Mulighederne kan endvidere yderligere begrænses, da rollen skal være forenelig med præpositionen *"på"*, hvilket dog ikke udelukker nogen i dette tilfælde, da både WRT og CBY kan afledes over i *"på"*.

Konklusionen er således, at hvis det efterfølgende navneled (6) skal give mening i forhold til den begrebsmæssige grammatik, skal det kunne afledes af et af begreberne $CONCR$ og $OCCUR$.

Den værdi, der således sendes nedad som (6)NP.i, er den tidligere beskrevne operation *HeadDerive* med udgangspunkt i både begrebet $CONCR$ og begrebet $OCCUR$. Det giver mængden indeholdende de beskrivere, som kan

afledes af $CONCR$ og $OCCUR$:

$$(6)NP.i = \{$$
$$(Lack,\ [\ CBY : OCCUR\ ]\ [\ WRT : CONCR\ ]),$$
$$(Symptom,\ [\ CBY : OCCUR\ ]),$$
$$(Disease,\ [\ CBY : OCCUR\ ]),$$
$$(VitaminA,\ [\ CHR : PROPERTY\ ]),$$
$$(VitaminB,\ [\ CHR : PROPERTY\ ]),$$
$$(VitaminC,\ [\ CHR : PROPERTY\ ]),$$
$$(Vitamin,\ [\ CHR : PROPERTY\ ])$$
$$\}$$

$(7)Noun$ syntetiserer, som det tidligere $Noun$ det tilhørende ord. I dette tilfælde *"C-vitamin"*.

$$(7)Noun.s = "c\text{-}vitamin"$$

Dette benyttes ligesom tidligere til sammenligning med de mulige beskrivere, der er blevet nedarvet. Den beskriver, der returners opad i syntakstræet, er således:

$$(6)NP.s = \{$$
$$(VitaminC,\ [\ CHR : PROPERTY\ ])$$
$$\}$$

Hele præpositionsledet $(4)PP$ er nu genkendt, og der skal findes en værdi, som skal returneres opad i syntakstræet, som udtryk for hvilke beskrivere der er mulige på baggrund af den værdi, som blev nedarvet, og de værdier som er blevet syntetiseret fra $(5)Prep$ og $(6)NP$. Da det lykkedes at finde en beskriver, der kunne afledes af $CONCR$ er den mulige semantiske relation $[\ WRT : CONCR\ ]$ genkendt i form af $[\ WRT : VitaminC\ ]$, som tilføjes den nedarvede beskriver $Lack$ i $(4)PP.i$. Derfor fås:

$$(4)PP.s = \{$$
$$(Lack\ [\ WRT : VitaminC\ ],\ [\ CBY : OCCUR\ ])$$
$$\}$$

Denne værdi videresendes opad til startsymbolet, så

$$
\begin{aligned}
&(2) NP.s = \{ \\
&\qquad (Lack\ [\ WRT : VitaminC\ ],\ [\ CBY : OCCUR\ ]) \\
&\}
\end{aligned}
$$

Der er således fundet en beskriver, der afspejler sætningens semantiske indhold. Den medsendte mulige semantiske relation, som ikke fandt nogen anvendelse, smides væk og tilbage haves beskriveren:

$$Lack\ [\ WRT : VitaminC\ ]$$

Det her gennemgåede eksempel illustrerer, hvordan beskriverne, med udgangspunkt i den begrebsmæssige grammatik, opbygges på baggrund af det lingvistiske syntakstræs udformning og de ord, som indgår i sætningen.

Som det fremgår, afledes de fulde beskriver ikke direkte fra startsymbolet for den begrebsmæssige grammatik i en *top-down* afledning som i figur 4.1. De stykkes derimod sammen af mindre beskrivere, som genkendes i forskellige dele af det lingvistiske syntakstræ. Delene stykkes derefter sammen af attributfunktionerne for den lingvistiske grammatik på baggrund af de til rådighed værende semantiske relationer.

4.5 Strukturering af attributfunktioner

De operationer, der blev beskrevet undervejs i forrige eksempel, og som kombinerer attributværdierne til at opbygge gyldige beskrivere, er som nævnt placeret i attributfunktionerne. Attributfunktionerne beskriver således, hvordan hver enkelt lingvistisk regel skal kombinere de til rådighed værende attributværdier. Dette gøres med henblik på at danne beskrivere, der afspejler den linvistiske konstruktion, som reglen afspejler. I det her viste eksempel var det præposition og dermed attributfunktioner for reglerne $NP ::= Noun \mid NP\ \ PP$ og $PP ::= Prep\ \ NP$, der blev anvendt og til dels beskrevet.

Da alle attributberegninger er baseret på venstrekonteksten, har attributfunktionerne selvfølgelig kun de attributter, som allerede er syntetiseret til rådighed i beregningerne svarende til beskrivelsen i afsnit 2.7.2.

Der er principielt ingen grænser for, hvilke operationer attributfunktionerne kan udføre for at finde frem til et resultat, da funktionerne som nævnt udformes individuelt til hver enkelt regel i den lingvistiske grammatik. For hver ny regel man kunne finde på, vil der derfor være nye og anderledes attributfunktioner.

Dette har betydning i forhold til opbygningen af en prototype af tekstanalysatoren, da det skal være muligt at angive attributfunktionerne sammen med den lingvistiske grammatik. Det kræver en eller anden form for generalisering af attributfunktionernes operationer, som samtidigt gør det forholdsvist enkelt at angive attributfunktionerne sammen med grammatikken. Generaliseringen skal være bred nok til at omfatte de mest oplagte konstruktioner uden at blive unødigt kompliceret. Der ønskes derfor indført nogle basisfunktioner, som kan anvendes til at beskrive og angive attributfunktionerne for den lingvistiske grammatik.

Bemærk i den forbindelse at basisfunktionerne kun indføres af hensyn til brugergrænsefladen for prototypen, ligesom de benyttes i beskrivelsen af attributfunktionerne for de øvrige regler i den lingvistiske grammatik. De skal derfor ikke ses, som en generel afgrænsning af hvilke attributfunktioner, der er mulige.

Som udgangspunkt for vurderingen af hvilke mekanismer, der bør understøttes, er benyttet de hidtil anvendte grammatikker. Den tidligere beskrevne lingvistiske grammatik og den begrebsmæssige grammatik fra figur 4.3. Derudover er der ikke mindst skelet til strukturen af beskriverne. Det har givet anledning til følgende formelle krav til basisfunktionerne:

- Argumenttyperne skal svare til de typer, som de rundsendte attributværdier har. Dvs. mængder af beskrivere med mulige semantiske relationer samt tekststrenge.

- Basisfunktionerne skal alle returnere en mængde af beskrivere med angivelse af de mulige semantiske relationer, så funktionerne kan anvendes rekursivt i kombination eller direkte som attributfunktioner.

Der er desuden følgende uformelle krav til den tilbudte funktionalitet:

- Det skal være muligt at finde de beskrivere i en mængde, som afspejler et givet ord eller indeholder mulige semantiske relationer, hvis rolle eller begreb afspejler et givent ord. På den måde kan ikke gyldige

beskrivere sorteres fra, når fx et konkret ord syntetiseres fra et terminalt symbol.

- Det skal være muligt at finde de beskrivere, som et givet begreb kan afledes over i svarende til den omtalte *HeadDerive* operation. Derved er det muligt at opbygge mængder af mulige beskrivere på baggrund af bl.a de begreber, som optræder i de semantiske relationer.

- Det skal for beskrivere med mulige semantiske relationer være muligt at overføre mulige semantiske relationer til beskriveren. Dette gøres på baggrund af et begreb og en rolle, som afspejler den derved genkendte semantiske relation. Denne funktionalitet er helt afgørende i forhold til at opbygge beskriverne ved brug af de mulige semantiske relationer.

På denne baggrrund indføres de tre basisfunktioner *Match*, *FeatureAccept* og *HeadDerive*.

Til forklaring af deres funktionalitet indføres følgende typestruktur, som afspejler beskrivernes opbygning:

`Role`
Ikke terminalt symbol der afspejler en rolle i en semantisk relation
`Concept`
Ikke terminalt symbol der repræsenterer begreb i den begrebsmæssige grammatik
`TConcept`
Ikke terminalt symbol der repræsenterer et terminalt begreb
`Feature: Role × Concept`
Mulig semantisk relation
`AcceptedFeature: Role × Descriptor`
Accepteret semantisk relation
`Descriptor: TConcept × AcceptedFeature-set`
Beskriver
`DescriptorWPF: Descriptor × Feature-set`
Beskriver med mulige semantiske relationer

WPF står for *with possible features* og hentyder til de beskrivere med vedhæftede mulige semantiske relationer, som anvendes i attributværdierne.

Beskrivere med denne tilhørende mængde af mulige semantiske relationer kaldes i det følgende for *WPF-beskrivere*.

4.5.1 Match

Match operationen benyttes til at finde frem til de WPF-beskrivere i en mængde, hvis indhold afspejles af de givne streng-argumenter. Der kan vælges at sammenligne beskriverne med en streng afspejlende et terminalt begreb, og/eller der kan vælges at sammenligne med de mulige semantiske relationers indhold. Dette gøres ved angivelse af en streng afspejlende en semantisk rolle og/eller en streng, som skal være en afledning af den semantiske relations begreb.

Funktionen tager således en mængde af mulige beskrivere, som gennemsøges for elementer, hvis indhold passer til de øvrige argumenter. Det er tilstrækkeligt, at mindst ét af de tre sidste argumenter er til stede.

En mere formel specifikation har følgende udseende.

```
Match:
   DescriptorWPF-set × String × String × String →
   DescriptorWPF-set

Match(DS, word1, role, word2) = {
   D |
      D ∈ DS ∧
      f ∈ D.Feature-set ∧
      (Defined(word1) ⇒
         D.Descriptor.Tconcept DerivesDirectly[1] word1) ∧
      (Defined(role) ⇒ f.Role Derives role) ∧
      (Defined(word2) ⇒ f.Concept Derives word2)
}
```

[1] Operationen *Derives* afspejler afledningsrelationen for grammatikker. A *derives* B er således sandt, hvis det ikke-terminale symbol A kan afledes over i symbolet B ifølge den begrebsmæssige grammatik. Operationen afgør således om det terminale begreb, som ordet B repræsenterer, hører under begrebet A hvad angår begrebsinklusion. Operationen *DerivesDirectly* afgør om der findes en regel i grammatikken der afleder et symbol over i et andet. *DerivesDirectly* benyttes her til at afgøre, om en streng afspejler et terminalt begreb.

De følgende eksempler på *Match* er baseret på de mængder, som er defineret i i figur 4.7.

$$
\begin{aligned}
&Match(possibleDescriptorsWPF1, "mangel", \_, \_) = \{ \\
&\qquad (Lack,\ [\ CBY : OCCUR\ ]\ [\ WRT : CONCR\ ]) \\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&Match(possibleDescriptorsWPF1, \_, "skyldes", "disease") = \{ \\
&\qquad (Lack,\ [\ CBY : OCCUR\ ]\ [\ WRT : CONCR\ ]), \\
&\qquad (Symptom,\ [\ CBY : OCCUR\ ]), \\
&\qquad (Disease,\ [\ CBY : OCCUR\ ]) \\
&\}
\end{aligned}
$$

Bemærk at de semantiske relationer der sammenlignes med, er de *mulige* semantiske relationer, og ikke de allerede genkendte/accepterede, hvilket illustreres af eksemplet nedenfor.

$$
\begin{aligned}
&Match(possibleDescriptorsWPF3, \_, \_, "disease") = \{ \\
&\qquad (Lack\ [\ WRT : VitaminC\ ],\ [\ CBY : OCCUR\ ]) \\
&\}
\end{aligned}
$$

$$Match(possibleDescriptorsWPF3, \_, \_, "mangel") = \{\}$$

4.5.2 FeatureAccept

Formålet med *FeatureAccept* er at gennemgå en given mængde af WPF-beskrivere med henblik på at få genkendt mulige semantiske relationer, som dermed kan overføres til beskriveren. For at en mulig semantisk relation kan accepteres som genkendt, skal dens indhold være i overensstemmelse med de øvrige to argumenter. De øvrige to argumenter afspejler henholdsvis den rolle og det begreb, som en semantisk relation skal indeholde for at blive accepteret. Da argumentet, der afspejler det begreb, som ønskes genkendt i en semantisk relation, kan være af to forskellige typer, findes der to versioner af *FeatureAccept*.

$$
\begin{aligned}
&possibleDescriptorsWPF1 = \{ \\
&\qquad (Lack,\ [\ CBY : OCCUR\ ]\ [\ WRT : CONCR\ ]), \\
&\qquad (Disease,\ [\ CBY : OCCUR\ ]), \\
&\qquad (Symptom,\ [\ CBY : OCCUR\ ]), \\
&\qquad (VitaminA,\ [\ CHR : PROPERTY\ ]), \\
&\qquad (VitaminB,\ [\ CHR : PROPERTY\ ]), \\
&\qquad (VitaminC,\ [\ CHR : PROPERTY\ ]), \\
&\qquad (Vitamin,\ [\ CHR : PROPERTY\ ]), \\
&\qquad (Organic, \_) \\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&possibleDescriptorsWPF2 = \{ \\
&\qquad (VitaminA,\ [\ CHR : PROPERTY\ ]), \\
&\qquad (VitaminB,\ [\ CHR : PROPERTY\ ]), \\
&\qquad (VitaminC,\ [\ CHR : PROPERTY\ ]), \\
&\qquad (Vitamin,\ [\ CHR : PROPERTY\ ]) \\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&possibleDescriptorsWPF3 = \{ \\
&\qquad (Lack\ [\ WRT : VitaminC\ ],\ [\ CBY : OCCUR\ ]) \\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&possibleDescriptorsWPF4 = \{ \\
&\qquad (Disease,\ [\ CBY : OCCUR\ ]) \\
&\}
\end{aligned}
$$

Figur 4.7: Mængder af beskrivere med angivelse af mulige sematiske relationer.

I den første version skal begreberne i de mulige semantiske relationer sammenlignes med et ord i form af en tekststreng, der afspejler et terminalt begreb. I dette tilfælde gennemgås de mulige semantiske relationer for hver WPF-beskriver, og hvis den semantiske rolle passer med den streng, som afspejler rollen, og relationens begreb kan afledes over i den anden streng, er der bid. Der er derved fundet en mulig semantisk relation, som kan overføres til den tilhørende beskriver. Overførslen foregår ved at den semantiske relation kobles på beskriveren, og relationens begreb erstattes af det terminale begreb, som den givne begrebs-streng afspejler.

En mere formel beskrivelse ses nedenfor:

```
FeatureAccept v1:
   DescriptorsWPF-set × String × String →
   DescriptorWPF-set

FeatureAccept(DS, role, word) = {
   D.accept[2](f,word) |
      D ∈ DS ∧
      f ∈ D.feature-set ∧
      (Defined(role) ⇒ f.Role Derives role) ∧
      f.Concept Derives word
}
```

Eksemplet nedenfor illustrerer funktionen. Eksemplet er baseret på mængderne defineret i figur 4.7.

$$FeatureAccept1(possibleDescriptorsWPF1, \text{"på"}, \text{"}C\text{-}vitamin\text{"}) = \{$$
$$(Lack\ [\ WRT : VitaminC\ ],\ [\ CBY : OCCUR\ ])$$
$$\}$$

Blandt de mulige semantiske relationer i mængden $possibleDescriptorsWPF1$ er det kun $[\ WRT : CONCR\ ]$ og $[\ CBY : OCCUR\ ]$, hvis roller kan afledes over i strengen "*på*". Af de tilhørende begreber $CONCR$ og $OCCUR$ er det dog kun $CONCR$, som kan afledes over i "C-*vitamin*". Den mulige semantiske relation $[\ WRT : CONCR\ ]$ er derfor genkendt, og den kan gøres til en del af beskriveren. Den semantiske relation kobles således på beskriveren, og $CONCR$ erstattes af det terminale begreb $VitaminC$.

[2] *Accept* afspejler at den pågældende beskriver overfører en af sine mulige semantiske relationer til beskriveren.

I den anden version af *FeatureAccept* er den streng, som før blev sammenlignet med de mulige semantiske relationers begreb, erstattet af en mængde af WPF-beskrivere. Ønsket er så at få genkendt semantiske relationer som før, men ikke kun med baggrund i et terminalt begreb, men med baggrund i hele beskrivere. Mere formelt fås:

```
FeatureAccept v2:
   DescriptorsWPF-set × String × DescriptorsWPF-set →
   DescriptorWPF-set

FeatureAccept(DS1, role, DS2) = {
   D1.accept(f, D2) |
      D1 ∈ DS1 ∧
      f ∈ D1.Feature-set ∧
      D2 ∈ DS2 ∧
      (Defined(role) ⇒ f.Role Derives role) ∧
      f.Concept Derives D2.Descriptor.TConcept
}
```

Nedenfor ses et eksempel på anvendelse af funktionen.

$$FeatureAccept2$$
$$(possibleDescriptorsWPF1, \text{"på"}, possibleDescriptorsWPF2) = \{$$
$$(Lack\ [\ WRT : VitaminA\ ],\ [\ CBY : OCCUR\ ]),$$
$$(Lack\ [\ WRT : VitaminC\ ],\ [\ CBY : OCCUR\ ]),$$
$$(Lack\ [\ WRT : Vitamin\ ],\ [\ CBY : OCCUR\ ])$$
$$\}$$

Som før er det både den mulige relation $[\ WRT : CONCR\ ]$ og $[\ CBY : OCCUR\ ]$, hvis roller kan afledes over i strengen "*på*". Begreberne $CONCR$ og $OCCUR$ sammenholdes derfor med beskriverne i mængden $possibleDescriptorsWPF2$. Sammenligningen foregår ved, at undersøge om beskrivernes terminale begreb kan afledes fra $CONCR$ eller $OCCUR$.

Er dette muligt erstattes det pågældende begreb af beskriveren, og den semantiske relation er genkendt. Der oprettes naturligvis nye WPF-beskrivere, alt efter hvor mange det er muligt at danne ud fra de givne argumenter.

I dette eksempel indeholdt ingen af beskriverne i mængden $possibleDescriptorsWPF2$ allerede accepterede semantiske relationer. Hvis dette havde været tilfældet, var de naturligvis fulgt med over i de semantiske relationer fra $possibleDescriptorsWPF1$, som var blevet genkendt. Dette illustreres af følgende eksempel.

$$
\begin{aligned}
&FeatureAccept2(\\
&\quad possibleDescriptorsWPF4,, possibleDescriptorsWPF3) = \{\\
&\qquad (Disease\ [\ CBY : Lack\ [\ WRT : VitaminC\ ]\ ], \_)\\
&\}
\end{aligned}
$$

Eksemplet viser endvidere, at rolle-argumentet ikke behøver at være angivet. Hvis det mangler, accepteres alle roller, og sammenligningen med de mulige semantiske relationer vil kun omfatte relationernes begreb.

FeatureAccept kan således kombinere genkendte beskrivere ved hjælp af de til rådighed værende semantiske relationer og på den måde anvendes til at opbygge større og mere komplekse beskrivere.

4.5.3 HeadDerive

HeadDerive finder med udgangspunkt i et begreb alle de terminale begreber, som det givne begreb kan afledes over i ifølge den begrebsmæssige grammatik. Undervejs i afledningerne opsamles selvfølgelig de nedarvede mulige semantiske relationer. *HeadDerive* benyttes således til at finde den mængde af WPF-beskrivere, som det givne begreb kan afledes over i.

Nedenfor ses resultatet af, at anvende *HeadDerive* på startsymbolet for den begrebsmæssige grammatik.

$$
\begin{aligned}
&HeadDerive1(\text{"}UNIV\text{"}) = \{\\
&\qquad (Lack,\ [\ CBY : OCCUR\ ]\ [\ WRT : CONCR\ ]),\\
&\qquad (Disease,\ [\ CBY : OCCUR\ ]),\\
&\qquad (Symptom,\ [\ CBY : OCCUR\ ]),\\
&\qquad (VitaminA,\ [\ CHR : PROPERTY\ ]),\\
&\qquad (VitaminB,\ [\ CHR : PROPERTY\ ]),\\
&\qquad (VitaminC,\ [\ CHR : PROPERTY\ ]),\\
&\qquad (Vitamin,\ [\ CHR : PROPERTY\ ]),\\
&\qquad (Organic, \_)\\
&\}
\end{aligned}
$$

Ligesom FeatureAccept findes HeadDerive også i to versioner, hvoraf den første netop er beskrevet. Den anden version knytter sig til samme funktionalitet, men anvendes på begreber der er placeret i mulige semantiske relationer. For at nå ind til disse er der dog behov for angivelse af den mængde af WPF-beskrivere, de mulige semantiske relationer skal findes i, samt angivelse af en valgfri begrænsning i form af hvilken rolle de berørte relationer skal indeholde.

HeadDerive har derfor følgende to specifikationer:

```
HeadDerive v1:
   Concept → DescriptorWPF-set

HeadDerive v2:
   DescriptorsWPF-set × String → DescriptorWPF-set
```

Et par eksempler på den sidst beskrevne version baseret på mængderne i figur 4.7 ses nedenfor.

$$
\begin{aligned}
&HeadDerive2(possibleDescriptorsWPF1, \text{"}skyldes\text{"}) = \{\\
&\qquad (Lack,\ [\ CBY : OCCUR\ ]\ [\ WRT : CONCR\ ]),\\
&\qquad (Symptom,\ [\ CBY : OCCUR\ ]),\\
&\qquad (Disease,\ [\ CBY : OCCUR\ ])\\
&\}
\end{aligned}
$$

De mulige semantiske relationer i mængden $possibleDescriptorsWPF1$ gennemgås, og som det ses er det kun $[\ CBY : OCCUR\ ]$, hvis rolle kan afledes over i "*skyldes*". Derfor anvendes den først beskrevne version af *HeadDerive* på $OCCUR$, og det fundne resultat returneres.

Var rollen "*skyldes*" ikke blevet angivet, var den første version af *HeadDerive* blevet anvendt på alle de begreber, der forekommer i de mulige semantiske relationer, dvs. både $OCCUR$ og $CONCR$. Resulatet var i så

fald blevet som nedenfor.

$$
\begin{aligned}
&HeadDerive2(possibleDescriptorsWPF1, \_) = \{\\
&\quad (Lack,\ [\ CBY : OCCUR\ ]\ [\ WRT : CONCR\ ]),\\
&\quad (Symptom,\ [\ CBY : OCCUR\ ]),\\
&\quad (Disease,\ [\ CBY : OCCUR\ ]),\\
&\quad (VitaminA,\ [\ CHR : PROPERTY\ ]),\\
&\quad (VitaminB,\ [\ CHR : PROPERTY\ ]),\\
&\quad (VitaminC,\ [\ CHR : PROPERTY\ ]),\\
&\quad (Vitamin,\ [\ CHR : PROPERTY\ ])\\
&\}
\end{aligned}
$$

Bemærk igen, at det er de *mulige* semantiske relationer, som er i søgelyset og ikke de allerede genkendte, hvilket også illustreres i følgende eksemepel.

$$HeadDerive2(possibleDescriptorsWPF3, "på") = \{\}$$

4.5.4 Beskrivelse af attributfunktioner

I form af de indførte basisfunktioner haves nu et redskab til at beskrive attributfunktionerne. De attributfunktioner, som i løse vendinger blev beskrevet for nogle af reglerne i den lingvistiske grammatik i forbindelse med sætningen mangel på C-vitamin i afsnit 4.4, kan nu beskrives ved hjælp af de indførte baisisfunktioner. For den første regel i den lingvistiske grammatik fås følgende:

$$
\begin{aligned}
S ::= NP \quad & NP.i = S.i\\
& S.s = NP.s
\end{aligned}
$$

Reglen forbinder startsymbolet med en eventuelt genkendt NP. Der skal således ikke foregå nogen bearbejdning af de mulige beskrivere, som er syntetiseret af NP. Attributfunktionerne sender derfor blot mængden videre op eller ned i syntakstræet.

$$NP ::= Noun \quad NP.s = Match(NP.i, Noun.s, \_, \_)$$

Terminalsymbolet $Noun$ giver kontakt med sætningens indhold, som derfor benyttes til at sortere og udelukke de mulige beskrivere, som ikke er kompatible med det syntetiserede ord. De beskrivere, der sendes opad i syntakstræet er således dem, hvis terminale begreb er tilknyttet det syntetiserede ord.

$$
\begin{aligned}
NP_1 ::= NP_2 \ \ PP \quad & NP_2.i = NP_1.i\\
& PP.i = NP_2.s\\
& NP_1.s = PP.s
\end{aligned}
$$

Denne regel knytter et præpositionsled til et navneled. Fra NP'et i højresiden syntetiseres de beskrivere, som er mulige efter at dette er gendkendt. Da et præpositionsled i den begrebsmæssige grammatik består af en udbygning i form af en semantisk relation, sendes de syntetiserede beskrivere blot ned til PP'et i form af attributten $PP.i$. PP syntetiserer så de beskrivere, som det har været muligt at tilknytte det genkendte præpositionsled. Denne mængde kan sendes ubehandlet videre op i form af $NP1.s$.

$$
\begin{aligned}
PP ::= Prep \ \ NP \quad & NP.i = HeadDerive2(PP.i, Prep.s)\\
& PP.s = FeatureAccept2(PP.i, Prep.s, NP.s)
\end{aligned}
$$

Reglen opbygger et præpositionsled ud fra en genkendt præposition og et navnled i form af en NP.

Formålet med NP leddet er, at få genkendt nogle beskrivere, som kan anvendes til at fylde i nogle af de mulige attributter, som er nedarvet i $PP.i$. De skal dog ligeledes passe med den/de roller, som angives af den syntetiserede præposition i form af ordet $Prep.s$.

De mulige beskrivere, som således sendes ned i $NP.i$, er baseret på de mulige semantiske relationer fra $PP.i$, hvis rolle kan forenes med $Prep.s$.

Til sidst kombineres den syntetiserede præposition og de syntetiserede navneled i de mulige semantiske relationer fra $PP.i$.

Eksempler på attributfunktionernes anvendelse kan for så vidt ses i det tidligere eksempel i afsnit 4.4, da de viste værdier er de samme som de nu nærmere definerede attributfunktioner vil finde frem til.

Attributfunktionerne for de øvrige regler i den lingvistiske grammatik beskrives i det følgende på tilsvarende vis med udgangspunkt i basisfunktionerne.

For den del af den lingvistiske grammatik, som afspejler udsagnsled er attributfunktionerne magen til de netop beskrevne, da præpositionsled og udsagnsled håndteres på samme måde i både den lingvistiske og den begrebsmæssige grammatik.

$$\begin{aligned} NP_1 ::= NP_2 \;\; VP \quad & NP_2.i = NP_1.i \\ & VP.i = NP_2.s \\ & NP_1.s = VP.s \end{aligned}$$

$$\begin{aligned} VP ::= Verb \;\; NP \quad & NP.i = HeadDerive2(VP.i, Verb.s) \\ & VP.s = FeatureAccept2(VP.i, Verb.s, NP.s) \end{aligned}$$

Der er dog reglen $VP ::= RelPron \;\; Verb \;\; NP$ til at tage sig af et eventuelt relativt pronomen. Attributfunktionerne for reglen er dog de samme som uden det relative pronomen, da tilstedeværelsen af dette ikke forandrer den tilhørende semantiske beskrivelse.

$$\begin{aligned} VP ::= RelPron \;\; Verb \;\; NP \quad & NP.i = HeadDerive2(VP.i, Verb.s) \\ & VP.s = FeatureAccept2(VP.i, Verb.s, NP.s) \end{aligned}$$

I forbindelse med håndteringen af adjektiver indførtes reglen $NP ::= Adj \;\; NP$. Attributfunktionerne for denne ser ud som følger.

$$\begin{aligned} NP_1 ::= Adj \;\; NP_2 \quad & NP_2.i = FeatureAccept2(NP_1.i, \_, Adj.s) \\ & NP_1.s = NP_2.s \end{aligned}$$

Denne konstruktion er anderledes end de tidligere beskrevne, da det første symbol i reglens højreside Adj ikke er det primære udgangspunkt for hvilke beskrivere der kan genkendes. Det knytter sig i stedet til det efterfølgende NP. Der er ligeledes heller ikke et konkret ord i sætningen, som knytter sig til en bestemt rolle i den begrebsmæssige grammatik. Adjektivet afspejles i den begrebsmæssige grammatik af et begreb, som i form af en semantiske relation er tilknyttet det efterfølgende NP.

I mængden $NP_1.i$ ledes der således efter mulige semantiske relationer, hvis tilhørende begreb afspejles af adjektivet $Adj.s$, uanset hvad relationens rolle er.

Anvendes reglen på organisk{Adj} C-vitamin{Noun}, vil attributværdierne være følgende:

$$\begin{aligned} NP_1.i = \{ & \\ & (Lack,\ [\ CBY : OCCUR\ ]\ [\ WRT : CONCR\ ]), \\ & (Symptom,\ [\ CBY : OCCUR\ ]), \\ & (Disease,\ [\ CBY : OCCUR\ ]), \\ & (VitaminA,\ [\ CHR : PROPERTY\ ]), \\ & (VitaminB,\ [\ CHR : PROPERTY\ ]), \\ & (VitaminC,\ [\ CHR : PROPERTY\ ]), \\ & (Vitamin,\ [\ CHR : PROPERTY\ ]), \\ & (Organisk, \_) \\ \} & \end{aligned}$$

$$Adj.s = \text{"}organisk\text{"}$$

$$\begin{aligned} NP_2.i = FeatureAccept2(NP_1.i, \_, Adj.s) = \{ & \\ & (VitaminA\ [\ CHR : Organic\ ], \_), \\ & (VitaminB\ [\ CHR : Organic\ ], \_), \\ & (VitaminC\ [\ CHR : Organic\ ], \_), \\ & (Vitamin\ [\ CHR : Organic\ ], \_) \\ \} & \end{aligned}$$

$$\begin{aligned} NP_2.s = \{ & \\ & (VitaminC\ [\ CHR : Organic\ ], \_) \\ \} & \end{aligned}$$

$$NP_1.s = NP_2.s$$

4.6 Tekstanalysatoren

Tekstanalysatoren, som udfører de mekanismer der er beskrevet tidligere i kapitlet, er grundlæggende baseret på den modificerede version af Earleys algoritme, som er beskrevet i afsnit 3.4. Algoritmen virker umiddelbart på ordklasserne og den lingvistiske grammatik, men er udvidet med henblik på beregning og håndtering af de indførte attributværdier som er baseret på den begrebsmæssige grammatik.

Udvidelsen af den Early-inspirerede algoritme til også at kunne håndtere og udnytte attributter som de beskrevne kræver selvsagt flere ændringer af algoritmen. Ændringerne er beskrevet nærmere i det følgende.

4.6.1 Flere stakke

Før attributværdierne kan beregnes for de indgående stakelementer, er en grundlæggende ændring nødvendig i forhold til den tidligere beskrevne algoritme. I visse tilfælde vil det være nødvendigt at opdele den mængde af *items*, som repræsenterer et stakelement. Det drejer sig om de tilfælde, hvor flere *items* er i gang med at blive genkendt. Fx i følgende situation:

$$NP ::= NP \bullet PP$$
$$NP ::= NP \bullet VP$$
$$PP ::= \bullet Prep \;\; NP$$
$$VP ::= \bullet Verb \;\; NP$$

De viste *items* afspejler, at et NP netop er genkendt. Det vides blot ikke, hvilken af de to øverste regler der er under genkendelse. Det giver et problem, når man vil beregne de tilhørende attributværdier, da det ikke kan afgøres, hvilke attributfunktioner der skal anvendes til beregningerne.[3] Attributfunktionerne er som bekendt forskellige fra regel til regel. Stakelementer som dette vil derfor blive opdelt i to og derved give anledning til endnu en stak i syntaksanalysen. Indholdet af de øverste stakelementer på de to stakke vil således være:

[3] At attributfunktionerne for de viste regler i dette tilfælde er ens ændrer ikke ved det generelle problem.

$$NP ::= NP \bullet PP$$
$$PP ::= \bullet Prep \;\; NP$$

$$NP ::= NP \bullet VP$$
$$VP ::= \bullet Verb \;\; NP$$

De to stakke vil således repræsentere det samme lingvistiske syntakstræ, men med hver sine forventninger, om hvad der er under genkendelse, og hver sine attributværdier.

De tilføjede prædikterede *items* udgør ikke samme problem, da nedarvede attributværdier til det første symbol i reglernes højresider aldrig beregnes af en attributfunktion, men blot sendes ubehandlet videre fra venstresidens nedarvede værdi. Det skyldes at venstrekonteksten er uforandret i forhold til venstresidens nedarvede værdi. Det vil sige at attributfunktionen svarende til f_1 i figur 2.10 altid blot sender argumentet videre som resultat.

Ændringen betyder at hvert stakelement og dets tilhørende mængde af *items* nu indeholder et primært *item*, som beskriver hvilken produktion der er under genkendelse. De øvrige *items* i et stakelement vil udelukkende hidrøre fra prædiktion.

4.6.2 Håndtering af attributværdier

Attributværdierne ønskes beregnet sideløbende med at syntaksanalysen skrider frem. Dette lader sig gøre, da attributfunktionernes beregninger kun er baseret på attributværdier, som er til stede i venstrekonteksten, da den lingvistiske grammatik er *leftattributed*. Dette harmonererer med den måde den modificerede Earley-algoritme opbygger de lingvistiske syntakstræer på, svarende til et *preorder traversal*.

Som antydet ovenfor integreres attributværdierne i de stakelementer, som repræsenterer de lingvistiske syntakkstræer, der er under opbygning i syntaksanalysen.

Stakelementerne, der før udelukkende bestod af en mængde af *items*, udvides således til også at indeholde to attributværdier. De to attributværdier har betydning i forhold til de *items*, som udgør stakelementet.

$NP_1 ::= NP_2 \bullet PP$
$PP ::= \bullet Prep \;\; NP$

$lastSynthValue = NP_2.s$
$possibleParses = PP.i = f_a(NP_1.i, NP_2.s)$

$NP_1 ::= NP_2 \;\; PP\bullet$

$lastSynthValue = PP.s$
$possibleParses = NP_1.s = f_a(NP_1.i, NP_2.s, PP.s)$

Figur 4.8: Eksempler på værdier af *lastSynthValue* og *possibleParses*

Den ene værdi kaldes *lastSynthValue*, og er den værdi, som er syntetiseret fra det senest accepterede symbol. Det vil sige fra symbolet lige før prikken i det primære *item*. Denne værdi bruges som argument i fremtidige kald af attributfunktioner. For eksempel i forbindelse med en senere fuld genkendelse af det primære *item* og den deraf følgende reduktion.

Den anden attributværdi kaldes *possibleParses* og afspejler den værdi, som nedarves af symbolet efter prikken i det primære *item*. Det er således den samme værdi, som nedarves af symbolerne efter prikken i de prædikterede *items*. Værdien er desuden den, som bringes med fremad i syntaksanalysen og nedad i syntakstræet, når et nyt symbol indskiftes. Værdien er resultatet af en attributfunktion for reglen i det primære *item*.

Hvis hele højresiden er genkendt, og prikken i det primære *item* derved er nået til enden af højresiden, indeholder *possibleParses* i stedet den syntetiserede værdi for symbolet på venstresiden af reglen i det primære *item*. Det vil sige den værdi, som returneres opad i syntakstræet, efter at hele reglen er genkendt.

I figur 4.8 vises eksempler på de to værdier for forskellige stakelementer.

I forbindelse med et terminalt symbol efter prikken i det primære *item* er værdien af *possibleParses* blot tom, da de terminale symboler ikke nedarver nogen værdi.

4.6.3 Den semantiske begrænsning

Den sideløbende beregning af sætningernes semantiske indhold åbner op for, at den semantiske information i attributværdierne kan anvendes til at påvirke den igangværende syntaksanalyse. Dette udnyttes ved at stoppe den videre opbygning af et lingvistisk syntakstræ, hvis resultatet af en attributfunktion bliver en tom mængde. Dette afspejler nemlig, at det lingvistiske syntakstræ, som er under opbygning ikke er kompatibelt med nogle sætningsformer i den begrebsmæssige grammatik. Hvilklet igen betyder, at der ikke kan findes en gyldig beskriver for syntakstræet. Der er således ingen grund til at ofre energi på at bygge videre på syntakstræet.

Attributværdierne kan således sørge for, at kun de lingvistiske syntakstræer, som giver mening i forhold til den begrebsmæssige grammatik, opbygges fuldt ud. Opbygningen af de øvrige stoppes så snart, at det er afgjort, at de ikke kan føre til en gyldig beskriver.

Denne mekanisme kan have stor betydning for sætninger, som er stærkt tvetydige i forhold til den lingvistiske grammatik.

4.6.4 *Shift* og *reduce* - igen

Shift og *reduce*-operationerne fungerer grundlæggende stadig på samme måde som tidligere beskrevet, men er udvidet med hensyn til den semantiske information i attributværdierne. Når et nyt terminalsymbol indskiftes fra den sætning, som ønskes analyseret, og samtidig kan accepteres af de *items*, der tilhører det øverste stakelement, lægges der som bekendt et nyt element på stakken.

For dette nye stakelement skal de to attributværdier beregnes. Da det i sagens natur er et terminalt symbol, der netop er genkendt, vil *lastSynthValue* blot blive sat lig den til symbolet hørende tekststreng. Fx *"mangel"*. Værdien af *possibleParses* vil være resultatet af et kald af en attributfunktion, der beregner det følgende symbols nedarvede værdi i det nye stakelements primære *item*. Argumenterne til kaldet af funktionen findes ved at kigge på et antal af de øverste stakelementer. Antallet af stakelementer, der skal involveres, afhænger af hvor langt henne i højresiden prikken i det nye primære *item* er. Argumenterne i en attributfunktion er som tidligere beskrevet de syntetiserede værdier som er blevet returneret af symbolerne

i venstrekonteksten, hvilket netop er værdien af *possibleParses* for de involverede stakelementer.

Reduce tages som bekendt i brug, når prikken i det primære *item* har nået enden af højresiden. Det medfører, at et antal af de øverste stakelementer fjernes, og et nyt lægges på som erstatning. Beregningen af *possibleParses* for det nye stakelement, sker på samme måde som ved *shift*, blot med udgangspunkt i den reducerede stak. Da det indskiftede symbol her er et ikke-terminalt symbol, får *lastSynthValue* sin værdi på baggrund af et kald af den relevante attribut-funktion for den regel, som anvendes til reduktionen. Argumenterne til kaldet hentes hos de stakelementer, som fjernes fra stakken i forbindelse med reduktionen.

Den begrebsmæssige grammatiks indflydelse, som begrænser, gøres gældende ved hvert kald af en attributfunktion, da den tilhørende stak fjernes, hvis resultatet bliver tomt. Den semantiske begrænsning virker således i forbindelse med både *shift* og *reduce*.

Efter endt analyse kigges som før efter startsymbolet. Denne gang i form af det specielle *item* $S_n ::= S\bullet$, der afspejler at den indlæste sætning kunne genkendes. I dette tilfælde vil værdien af *possibleParses* for det tilhørende stakelement være de beskrivere, som det har været muligt at opbygge i den begrebsmæssige grammatik.

Kapitel 5

Implementering

I dette kapitel beskrives implementeringen af en prototype, der gør det muligt at arbejde med tekstanalysatoren og de tilhørende grammatikker.

Først beskrives programmets overordnede struktur, hvorefter implementeringen af de forskellige elementer i tekstanalysatoren beskrives. Sidst i kapitlet er de øvrige dele af programmet kort beskrevet.

5.1 Overordnet opbygning

Prototypen er implementeret i programmeringssproget JAVA version 1.3.

Programmet giver mulighed for at angive og redigere en begrebsmæssig og en lingvistisk grammatik med tilhørende attributfunktioner samt at udføre tekstanalyser på baggrund af de to grammatikker ved hjælp af den beskrevne tekstanalysator. De grammatikker og sætninger til analyse, som anvendes, kan indlæses fra filsystemet. For en nærmere beskrivelse af systemet henvises til brugervejledningen i bilag A, hvor prototypens brugergrænseflade også kan ses.

Programmet er opbygget objektorienteret med henblik på en modulær opbygning, som gør det enklere at udskifte eller ændre enkeltdele at programmet. For hovedparten af objekterne er deres interne repræsentation af data er således skjult for de øvrige klasser og tilgås udelukkende igennem, de metoder de enkelte objekter tilbyder.

Programmet er bygget op omkring den statiske klasse `GUI` som udover at repræsentere og administrere den grafiske brugergrænseflade, forbinder de forskellige dele af programmaet samt deres forskellige funktionaliteter. Den arbejder således med instanser af følgende væsentlige klasser.

- **Grammar**
  En instans af klassen repræsenterer en lingvistisk grammatik.
- **ConceptGrammar**
  En instans af klassen repræsenterer en begrebsmæssig grammatik.
- **SRTRBFBUParser**[1]
  En instans af klassen repræsenterer en tekstanalysator.
- **FileHandler**
  En statisk klasse, som håndterer kontakten med filsystemet, og bl.a. indlæser grammatikfiler til instanser af ovennævnte klasser.
- **GUIListener**
  Klasse som tager sig af de *events* som genereres af den grafiske brugergrænseflade.

Klassen `GUI` holder således blandt andet styr på, hvilke filer der er åbne, samt hvilke grammatikker der er aktive. Derudover sættes tekstanalyser i gang på baggrund af brugerens interaktion med brugergrænsefladen, ligesom resultater og andre informationer fra prototypen formidles til brugeren via brugergrænsefladen.

Bag de viste klasser findes et yderligere antal klasser, som repræsenterer underliggende datastrukturer og funktionaliteter.

I det følgende beskrives de forskellige klasser med udgangspunkt i hvordan de indgår i tekstanalysatoren.

[1] **S**emantic **R**estricted **T**op-down **R**estricted **B**readth **F**irst **B**ottom-**U**p **P**arser.

5.2 Den Lingvistiske grammatik

Grammar	
−startSym	:NonTerminal
−nonTerminals	:Hashtable
−terminals	:Hashtable
−productionRules	:Vector
+**getSymbol**(String name)	:Symbol
+**rhsToLhs**(Vector rhs)	:Vector
+**getStartSymbol**()	:NonTerminal
+**getProductionRules**()	:Vector

En lingvistisk grammatik er repræsenteret ved en instans af klassen `Grammar`. `Grammar` indeholder felter og metoder[2] som vist i UML-klassespecifikationen ovenfor.

Som det ses, indeholder klassen felter, der afspejler en BNF-grammatiks opbygning. En mængde ikke-terminale symboler, en mængde terminale symboler, et startsymbol og en mængde regler. Valget af JAVA-datastrukturerne `Hashtable` og `Vector` er baseret på behovet for opslag i og gennemløb af de pågældende mængder. `Hashtable` tilbyder gode muligheder for opslag, og `Vector` tilbyder gode muligheder for gennemløb.

Metoderne benyttes i af tekstanalysatoren i forbindelse med syntaksanalyse med hensyn til grammatikken.

I det følgende beskrives repræsentationen af grammatikkens symboler og regler med attributfunktioner.

[2]For overskuelighedens skyld er kun de centrale metoder medtaget i dette og de følgende UML-klassespecifikationer. De udeladte metoder er typisk til udskrivning og kopiering af objekter. For en komplet oversigt over klasseindholdet henvises til kildekoden i bilag B

5.2.1 Symboler

Terminal	
−name	:String
+**isTerminalSymbol**()	:boolean
+**getName**()	:String
+**equals**(NonTerminal sym)	:boolean
+**equals**(Terminal sym)	:boolean

NonTerminal	
−name	:String
+**isTerminalSymbol**()	:boolean
+**getName**()	:String
+**equals**(NonTerminal sym)	:boolean
+**equals**(Terminal sym)	:boolean

De symboler som indgår i den lingvistiske grammatik er repræsenteret ved instanser af klasserne `Terminal` og `NonTerminal`, som begge implementerer metoderne i *interfacet* `Symbol`. Klasserne indeholder en tekststreng med symbolets tekstuelle repræsentation samt metoder til sammenligning.

5.2.2 Regler i den lingvistiske grammatik

ProductionRule	
−lhs	:NonTerminal
−rhs	:Vector
−attributeFunctions	:AttributeFunctionDef[]
+**rhsMatch**(Vector string)	:boolean
+**getRhsSymbol**(int position)	:Symbol
+**getLhs**()	:Symbol
+**rhsSize**()	:int
+**rhsContains**(Symbol symbol)	:int
+**addFunDef**(int funPos, AttributeFunctionDef funDef)	
+**getFunDef**(int funNr)	:AttributeFunctionDef

Den lingvistiske grammatiks regler er repræsenteret ved instanser af klassen `ProductionRule`.

Klassen indeholder felter, der afspejler symbolet i reglens venstreside og symbolerne i højresiden samt de for reglen definerede attributfunktioner. Attributfunktionerne er repræsenteret ved instanser af den indre klasse `AttributeFunctionDef`.

Metoderne i `ProductionRule` giver tekstanalysatoren mulighed for at arbejde med reglerne i grammatikken. Blandt andet ved hjælp af metoden `rhsMatch()`, som anvendes, når det skal afgøres om en given symbolsekvens svarer til en regels højreside i *bottom-up* analysen. Derudover findes metoder til håndtering af attributfunktionerne.

5.2.3 Definitioner af attributfunktioner

ProductionRule.AttributeDefinition	
−operation	:int
−arguments	:Vector
+**addArgument**(int argument)	
+**addArgument**(AttributeFunctionDef argument)	
+**getOperation**()	:int
+**getArguments**()	:Vector

De angivne attributfunktioner for reglerne i en lingvistisk grammatik er som nævnt repræsenteret ved den i `ProductionRule` indre klasse `AttributeDefinition`. Instanser af klassen repræsenterer således definitionen af en attributfunktion, som er indlæst fra den tekstuelle repræsentation af grammatik og attributfunktioner, som anvendes i brugergrænsefladen.

Feltet `operation` afspejler, hvilken af de beskrevne basisfunktioner der repræsenteres. Dette gøres ved at tildele feltet værdien af én af et antal konstanter, der afspejler de forskellige operationer.

Konstanterne er defineret i den statiske klasse `AttributeCalculator`, som også varetager beregningerne, når attributfunktionerne tages i brug. Klassen beskrives i afsnit 5.5.2.

Blandt konstanterne er der desuden en værdi, der afspejler at en attributfunktion ikke er blevet angivet[3], samt en værdi, der anvendes til at angive

[3]Dette medfører at den senest nedarvede attributværdi sendes ubehandlet videre som resultat, når den ikke definerede funktionen skal anvendes.

et argument, som ikke er blevet angivet, som i følgende attributfunktionsdefinition:

$$FeatureAccept1(s0.i, \_, s1.s)$$

Vektoren `arguments`, hvis indhold afspejler de angivne argumenter, kan indeholde elementer af to typer. Der kan være heltal, som afspejler positioner for symboler i den tilhørende regels højreside. Positioner henviser til hvor de attributværdier, som skal indgå i attributfunktionens beregning skal findes.

Derudover kan argumenterne være andre instanser af klassen `AttributeFunctionDef`, hvilket afspejler at det pågældende argument er resultatet af en basisfunktion. Det svarer fx til det første argument i den nedenfor angivne attributfunktion.

$$FeatureAccept1(Match(s0.i, s2.s), \_, s1.s)$$

5.3 Den begrebsmæssige grammatik

ConceptGrammar	
−startSym	:NonTerminal
−nonTerminals	:Hashtable
−terminals	:Hashtable
−productionRules	:Vector
+**getSymbol**(String name)	:Symbol
+**getStartSymbol**()	:NonTerminal
+**getProductionRules**()	:Vector
+**findRuleWithLhs**(NonTerminal lhs)	:ConceptProductionRule
+**getLhsTo**(Terminal t)	:NonTerminal
+**isSubConceptOf**(NonTerminal sub, NonTerminal concept)	:boolean

Klassen `ConceptGrammar` repræsenterer en begrebsmæssig grammatik, og benytter sig ligesom `Grammar` af klasserne `Terminal` og `NonTerminal`, og

indeholder som det ses også grundlæggende de samme felter. Derudover indeholder `ConceptGrammar` metoder, der gør det muligt at arbejde med den begrebsmæssige grammatik i forbindelse med tekstanalysen.

Selvom klassen på overfladen ligner klassen, der repræsenterer den lingvistiske grammatik, er de underliggende datastrukturer anderledes, da det afspejles at den begrebsmæssige grammatik ikke er en vilkårligt udformet BNF-grammatik, men at den er baseret på en ontologisk struktur med semantiske relationer. `ConceptGrammar` indeholder derfor også et antal indre klasser, som bl.a. tager sig af denne anderledes struktur. De indre klasser repræsenterer blandt andet regler i den begrebsmæssige grammatik og beskrivere.

5.3.1 Regler i den begresmæssige grammatik

ConceptGrammar.ConceptProductionRule	
−lhs	:NonTerminal
−alternatives	:Vector
+**getLhs**()	:NonTerminal
+**getAlternatives**()	:Vector
+**containsRhs**(Symbol s)	:boolean
+**findRuleWithLhs**(NonTerminal lhs)	:ConceptProductionRule
+**getLhsTo**(Terminal t)	:NonTerminal
+**isSubConceptOf**(NonTerminal sub, NonTerminal concept)	:boolean

Reglerne i den begrebsmæssige grammatik er repræsenteret ved instanser af den indre klasse `ConceptProductionRule`. Den adskiller sig fra `ProductionRule`, som blev anvendt ved den lingvistiske grammatik, ved at den `Vector` som repræsenterer højresiden ikke indeholder symboler men instanser af en anden indre klasse `Alternative`. En instans af `Alternative` repræsenterer én af de mulige højresider som et ikke-terminalt symbol kan afledes over i. `Alternative` svarer således ikke til et enkelt symbol men til en hel højreside for en enkelt regel i den begrebsmæssige grammatik.

For hvert ikke-terminale symbol i den begrebsmæssige grammatik findes der således kun én instans af `ConceptProductionRule` i modsætning til implementeringen af regler i den lingvistiske grammatik, som giver anledning til en instans af `ProductionRule` for hver mulig højreside. Reglen

$\langle A \rangle ::= \langle A \rangle\langle B \rangle \mid a\langle C \rangle \mid a$ ville således kun give anledning til 1 instans af `ConceptProductionRule`, hvis reglen var en del af den begrebsmæssige grammatik. I den situation ville hver af de tre højresider være repræsenteret ved en instans af `Alternative`. Var reglen derimod en del af den lingvistiske grammatik ville den være repræsenteret ved 3 instanser af `ProductionRule`.

Forskellen i implementeringen af grammatikkernes regler skyldes, at højresiderne for den begrebsmæssige grammatik følger en bestemt struktur. Denne struktur afspejles som nævnt af klassen `Alternative`.

ConceptGrammar.Alternative	
−concept	:Symbol
−features	:Vector
+**getConcept**()	:Symbol
+**getFeatures**()	:Vector

Klassen `Alternative` indeholder de dele som en højreside i den begrebsmæssige grammatik består af: Et begreb i form af et `Symbol` (Dvs. en instans af `Terminal` eller `NonTerminal`) og en mængde mulige semantiske relationer.

En semantisk relation er repræsenteret ved en instans af den indre klasse `Feature`, som indeholder en rolle og et begreb i form af symboler fra den begrebsmæssige grammatik.

ConceptGrammar.Feature	
−role	:NonTerminal
−concept	:Symbol
+**getRole**()	:NonTerminal
+**getConcept**()	:Symbol

5.3.2 Beskrivere

Berskrivere er repræsenteret ved den i `ConceptGrammar` indre klasse `SemParsing`.

ConceptGrammar.SemParsing	
−tConcept	:Symbol
−acceptedFeatures	:Vector
−possibleFeatures	:Vector
−concept	:NonTerminal
−bHeadDerive	:boolean
+**getConcept**()	:NonTerminal
+**getTConcept**()	:Symbol
+**getPossibleFeatures**()	:Vector
+**getAcceptedFeatures**()	:Vector
+**acceptFeature**(ParsedFeature pf)	
+**isHeadDerive**()	:boolean
+**containsNotUsedMandatory**()	:boolean

Instanser af klassen repræsenterer således sætningsformer med udgangspunkt i den begrebsmæssige grammatik. En beskriver består af et terminalt begreb samt et antal genkendte/accepterede semantiske relationer. Klassen indeholder også et felt til de mulige semantiske relationer, som er nedarvet på baggrund af udformningen af grammatikkens regler. Disse anvendes, som beskrevet i afsnit 4.4 som udgangspunkt for at udvide beskriveren. Derudover gemmes det begreb, som sætningsformen oprindeligt er afledt af, samt et felt der angiver om sætningsformen afspejler *HeadDerive* funktionen anvendt på det angivne begreb. Det vil sige, at en instans kan repræsentere den mængde af beskrivere, som ville være resultatet af *HeadDerive* funktionen anvendt på det tilhørende begreb. Denne mekanisme anvendes for at udskyde eksekveringen af den tidskrævende *HeadDerive* funktion og derved udnytte en mulig afgrænsning af funktionen. Fx vil `HeadDerive` operationen anvendt på startsymbolet aflede beskrivere svarende til alle terminale begreber i grammatikken. Størstedelen af de fundne beskrivere vil dog blive kasseret, så snart at mere information om sætningen bliver til rådighed. Derfor kan eksekveringen af *HeadDerive* ligeså godt udskydes, indtil denne begrænsende information bliver til rådighed og derved kan udnyttes i beregningen.

De mulige semantiske relationer for beskriverne er repræsenteret ved instanser af klassen `Feature`, som også anvendes i repræsentationen af grammatikkens regler. De accepterede semantiske relationer er repræsenteret ved instanser af en anden indre klasse `ParsedFeatures`.

ConceptGrammar.ParsedFeature	
−feature	:Feature
−subParsing	:SemParsing

Klassen indeholder en instans af `Feature`, som afspejler hvilken semantisk relation der er genkendt, samt af en `SemParsing` instans, som afspejler den delbeskriver, der tilknyttes af den pågældende semantiske relation.

En beskriver er således opbygget rekursivt ved hjælp af klasserne `SemParsing` og `ParsedFeature`.

Metoderne i `SemParsing` anvendes primært i forbindelse med attributfunktionerne for den lingvistiske grammatiks beregninger af attributværdier, der som bekendt indeholder mængder af beskrivere. Den mest interessante metode i `SemParsing` er metoden `acceptFeature()`, som anvendes når en af de mulige semantiske relationer er genkendt og derfor skal overføres til de genkendte semantiske relationer. Metoden tager derfor en instans af klassen `ParsedFeature`, som tilføjes mængden af accepterede semantiske relationer.

Som det fremgår afspejler den beskrevne struktur for beskrivere ikke komplette syntakstræer med hensyn til den begrebsmæssige grammatik. Strukturen ligger nærmere op af den ontologiske information, som grammatikken beskriver. Selvom syntakstræerne gennemløbes i takt med at sætningerne dannes, er der er dog ingen grund til at gemme på dem, da det er de dannede sætningsformer, der er væsentlige. Eventuel tvetydighed i den begrebsmæssige grammatik på baggrund af fx polysemi som beskrevet i afsnit 4.2 tillægges således ingen betydning i de endelige beskrivere.

5.4 Sætninger til analyse

InputString	
−symbols	:Vector
−concepts	:Vector
−corrupted	:boolean
+getSymbol(int i)	:Symbol
+getConcept(int i)	:String
+isCorrupted()	:boolean
+getSymbols()	:Vector
+size()	:int

En instans af klassen `InputString` repræsenterer en sætning, som tekstanalysatoren kan tage ind til analyse. Klassen indeholder således felter, der afspejler en sætnings indhold. Dette gøres ved hjælp af to instanser af JAVA-klassen `Vector` med henholdsvis de indlæste ordklasser og de indlæste ord.

Klassen indeholder også et felt, der afspejler om en sætning indeholder ugyldige ord eller ordklasser.

Klassen udbyder derudover metoder, der kan levere de indlæste symboler til tekstanalysatoren.

5.5 Tekstanalysatoren

SRTRBFBUParser	
# grm	:Grammar
# cgrm	:ConceptGrammar
# InputString	:ips
# readSymbols	:int
# reductionStacks	:Vector
# newStartRule	:ProductionRule
+**setParseArguments**(Grammar grm, ConceptGrammar cgrm, InputString ips)	:Vector
+**run**()	
+**getReductionStacks**()	:Vector
−**shift**(int inputIndex)	
−**stackScanner**(Stack st, Symbol s, String c)	:Vector
−**reduce**()	
−**completeReduceStep**(Stack st)	:Vector
−**succesReduceStacks**()	:boolean

En instans af klassen `SRTRBFBUParser` fungerer som en tekstanalysator, der ud fra instanser af `Grammar`, `ConceptGrammar` og `InputString`, kan danne en eller flere beskrivere i form af instanser af `SemParsing`.

Udover felter til grammatikkerne og den sætning, der ønskes analyseret, indeholder klassen et felt, der holder styr på hvor meget af den givne sætning, der er analyseret, samt en `Vector` som indeholder alle de igangværende stakke. Derudover findes også et felt til den nye startregel for den lingvistiske grammatik, som indføres forud for en tekstanalyse.

Klassen implementerer JAVA-*interfacet* `Runnable` i form af metoden `run()`, som gør det muligt at starte tekstanalysen i en selvstændig tråd. Når en tekstanalyse skal udføres, angives argumenterne til analysen først ved hjælp af metoden `setParseArguments()`, som tager de tre nødvendige instanser som argumenter. Dernæst startes en tråd med udgangspunkt i metoden `run()`. Metoden udfører de indledende operationer såsom at indføre det nye startsymbol og tilføje den nye startregel til den sproglige grammatik. Derefter holdes styr på hvor meget af den givne sætning der er læst ind, imens metoderne `shift()` og `reduce()` kaldes skiftevis.

`shift()` kaldes med et indeks for det næste endnu ikke indlæste symbol og nedlægger og opretter nye stakke og stakelementer på baggrund af metoden `stackScanner()`, som anvendes på hver igangværende stak i `reductionStacks`. På samme måde anvender `reduce()` metoden `completeReduceStep()` på hver igangværende stak, og nedlægger og opretter nye stakke og stakelementer. Hvilke stakke, der nedlægges og oprettes af de to metoder, afhænger selvfølgelig af det indskiftede symbol og mulige reduktioner samt de beregnede attributværdier, som er styrende for den semantiske begrænsning.

Efter endt analyse kaldes metoden `getReductionStacks()`, som returnerer de stakke som analysen er endt med. På baggrund af disse findes tekstanalysens resultat i form af beskrivere og lingvistiske syntakstræer.

Fordelen ved at lade tekstanalyser køre i en selvstændig tråd er, at brugergrænsefladen derved ikke påvirkes af igangværende tekstanalyser. Der kan således stadig arbejdes i brugergrænsefladen, selvom en analyse er igang. Desuden kan analyser stoppes, før de er afsluttet, uden at hele programmet afbrydes.

5.5.1 Stakelementer

Feltet `reductionStacks` repræsenterer tekstanalysatorens tilstand i form af de igangværende sætningsformer i den lingvistiske grammatik. Hver sætningsform er som bekendt repræsenteret ved en stak svarende til den i figur 3.1 viste. Disse stakke er repræsenteret ved JAVA-datatypen `Stack`. Hvert element i en sådan stak indeholder et antal *items* samt de tilknyttede attributværdier. I implementeringen er stakelementerne beskrevet ved den indre klasse `StackElement`.

SRTRBFBUParser.StackElement	
−itemVector	:Vector
−lastSynthValue	:Vector
−possibleParses	:Vector
−parseTree	:SubParseTree
+**getPossibleParses**()	:Vector
+**getLastSynthValue**()	:Vector
+**getParseTreeCopy**()	:SubParseTree

Klassen indeholder et felt til håndtering af den mængde af *items*, der definerer stakelementet i forhold til den lingvistiske syntaksanalyse, samt et felt der repræsenter det lingvistiske syntakstræ i form af en instans af klassen `SubParseTree`. Derudover indeholder klassen to felter, der afspejler attributværdierne som beskrevet i afsnit 4.6.2. De to felter indeholder således hver især mængder af mulige beskrivere i form af en `Vector`, der indeholder instanser af klassen `SemParsing`.

De *items*, som er indeholdt i hvert stakelement, er repræsenteret ved den indre klasse `Item`.

SRTRBFBUParser.Item	
−rule	:ProductionRule
−dotPosition	:int
+**isPossibleDotShift**(Symbol nextSymbol)	:boolean
+**dotIsAtEnd**()	:boolean
+**getPredictorItems**(Vector result)	:Vector

Klassen indeholder felter, der afspejler en regel i den lingvistiske grammatik, samt prikkens position i det pågældende *item*.

Metoderne udspringer af, hvilke behov tekstanalysatoren har i forhold til *items* i en tekstanalyse.

5.5.2 Eksekvering af attributfunktioner

Undervejs i tekstanalysen skal nye attributværdier beregnes på baggrund af de attributfunktioner, som er defineret i forbindelse med den lingvistiske grammatik. Disse beregninger udføres af metoder i den statiske klasse `AttributeCalculator`.

AttributeCalculator	
+NO_OPERATION = 0	:int
+OPERATION_HEADDERIVE1 = 1	:int
+OPERATION_HEADDERIVE2 = 2	:int
+OPERATION_MATCH = 3	:int
+OPERATION_FEATUREACCEPT1 = 4	:int
+OPERATION_FEATUREACCEPT2 = 5	:int
+BLANK_ARGUMENT = -9	:int
+**calcAttributeValue**(ConceptGrammar cgrm, ProductionRule pRule, int funNr, Vector inputValues)	:Vector
−**calcFun**(ConceptGrammar cgrm, ProductionRule.AttributeFunctionDef funDef, Vector inputValues)	:Vector
−**headDerive1**(ConceptGrammar cgrm, String concept)	:Vector
−**headDerive2**(ConceptGrammar cgrm, Vector semParses, String role)	:Vector
−**match**(ConceptGrammar cgrm,Vector semParses, String concept)	:Vector
−**featureAccept1**(ConceptGrammar cgrm,Vector semParses, String role, String fConcept)	:Vector
−**featureAccept2**(ConceptGrammar cgrm,Vector semParses, String role, Vector fConcepts)	:Vector
−**findReducedConcepts**(Vector possibleParses)	:Vector
−**calcHeadDerive**(ConceptGrammar cgrm, NonTerminal concept, NonTerminal subConcept, Vector inheritedFeatures)	:Vector
−**calcRestrictedHeadDerive**(ConceptGrammar cgrm, NonTerminal concept, NonTerminal subConcept, Vector inheritedFeatures, Terminal restrictionConcept)	:Vector

Konstanterne afspejler de forskellige basisfunktioner og anvendes som beskrevet i afsnit 5.2.3 i repræsentationen af de definerede attributfunktioner i klassen `ProductionRule.AttributeFunctionDef`.

Når tekstanalysatoren har behov for at beregne en attributværdi på baggrund af en af de definerede attributfunktioner, anvendes metoden `calcAttributeValue()`. Som argumenter tager metoden en begrebsmæssig grammatik, en regel fra den lingvistiske grammatik, en angivelse af hvilken attributfunktion for reglen der ønskes anvendt, samt de attributværdier som er til rådighed som argumenter. Metoden udgør således tekstanalysatoren grænseflade til beregning af attributværdier.

På baggrund af de givne informationer findes frem til hvilken instans af `ProductionRule.AttributeFunctionDef`, som skal ligge til grund for attributberegningen. Derefter kaldes metoden `calcFun()`, som iværksætter kald af den metode, der afspejler den angivne basisfunktion. `calcFun()` kalder desuden sig selv rekursivt, hvis en attributfunktions-definition indeholder argumenter, der først skal beregnes som resultatet af en basisfunktion.

5.6 Øvrige klasser

Ud over de beskrevne klasser findes et yderligere antal klasser, som ikke direkte relaterer sig til tekstanalysatoren. Disse er kort beskrevet i det følgende. For en nærmere beskrivelse af klasserne henvises til kildekoden i bilag B.

5.6.1 Håndtering af grammatikker

Den statiske klasse `Filehandler` udgør en grænseflade til filsystemet og den tekstuelle repræsentation af grammatikkerne. Klassen giver således mulighed for at indlæse og gemme de grammatikker der arbejdes med i den grafiske brugergrænseflade. Desuden indeholder klassen metoder, der opbygger de tidligere beskrevne grammatikinstanser på baggrund af den tekstuelle repræsentation, som grammatikkerne har i brugergrænsefladen. Den tekstuelle repræsentation af grammatikkerne går i øvrigt igen i grammatikfilerne. Derved er det muligt at indlæse grammatikker, som ikke nødvendigvis er opbygget i denne prototype, ligesom man kan definere grammatikker i enhver teksteditor.

5.6.2 Brugergrænsefladen

Brugergrænsefladen er som nævnt i starten af kapitlet repræsenteret ved den statiske klasse `GUI`. Klassen er således ansvarlig for opsætningen af det programvindue, som udgør brugegrænsefladen. Vinduet er opbygget ved hjælp af de brugegrænseflade-komponenter, som findes i JAVA-API'et.

Knapperne og menupunkterne i brugegrænsefladen er tilknyttet en *eventlistener* i form af en instans af klassen `GUIListener`. `GUIListener` opsamler således de brugerhandlinger, der udføres ved hjælp af brugergrænsefladen. Handlingerne medfører kald af metoder i `GUI`, som igangsætter fx en tekstanalyse.

Derudover håndterer klassen `GUI` de output, som genereres af tekstanalysatoren, og som sendes videre til brugeren igennem brugegrænsefladen.

Kapitel 6

Afprøvning

Der er ikke foretaget en fuld systematiseret test af prototypen, da dette ikke er skønnet nødvendigt på grund af systemets forholdsvis beskedne omfang og det faktum, at kun én person har været involveret i hele udviklingen og implementeringen i et fokuseret udviklingsforløb.

Prototypen er dog blevet afprøvet løbende i de forskellige faser af udviklingen med en voksende mængde eksempelmateriale. Afprøvningen har omfattet forskellige begrebsmæssige og lingvistiske grammatikker med regler, der afspejler de forskellige former for rekursion og tvetydighed, som skal kunne håndteres. Desuden er brugergrænsefladen, der tilbyder editering af de i tekstanalysen benyttede grammatikker og attributfunktioner, blevet afprøvet med henblik på at sikre systemets generalitet, hvad angår angivelse og editering af grammatikker og attributfunktioner.

På den baggrund må prototypen siges at fungere efter hensigten.

6.1 Køretider

For at få en vurdering af, hvordan den implementerede tekstanalysator fungerer i praksis, er den afprøvet på forskellige sætninger med henblik på vurdering af køretider. Desuden blev effekten af den semantske begrænsning undersøgt.

I det følgende er angivet køretiderne for forskellige sætninger, som er analyseret ud fra de grammatikker, der findes i figur 4.3 og figur 4.4. Desuden er angivet hvor mange lingvistiske syntakstræer, som sætningerne giver anledning til med og uden semantisk begrænsning.

mangel{Noun} på{Prep} vitamin{Noun}

→ $Lack\ [\ WRT : Vitamin\ ]$

	# lingv. syntakstræer	køretid
Uden sem. begrænsning:	**1**	∼ **7** ms
Med sem. begrænsning:	**1**	∼ **7** ms

sygdom{Noun} der{Prep} skyldes{Verb} mangel{Noun} på{Prep} organisk{Adj} C-vitamin{Noun}

→ $Disease\ [\ CBY : Lack\ [\ WRT : VitaminC\ [\ CHR : Organic\ ]\ ]\ ]$

	# lingv. syntakstræer	køretid
Uden sem. begrænsning:	**2**	∼ **26** ms
Med sem. begrænsning:	**1**	∼ **19** ms

symptom{Noun} på{Prep} mangel{Noun} på{Prep} C-vitamin{Noun}

→ $Symptom\ [\ CBY : Lack\ [\ WRT : VitaminC\ ]\ ]$

	# lingv. syntakstræer	køretid
Uden sem. begrænsning:	**2**	∼ **13** ms
Med sem. begrænsning:	**1**	∼ **10** ms

symptom{Noun} på{Prep} .. symptom{Noun} på{Prep} mangel{Noun} på{Prep} C-vitamin{Noun} (**15** ord ialt)

→ $Symptom\ [\ CBY : Symptom\ [\ CBY : Symptom\ [\ CBY :\ ...$
$...\ Symptom\ [\ CBY : Lack\ [\ WRT : VitaminC\ ]\ ...\ ]$

	# lingv. syntakstræer	køretid
Uden sem. begrænsning:	**429**	$\sim$ **71.000** ms
Med sem. begrænsning:	**1**	$\sim$ **90** ms

symptom{Noun} på{Prep} .. symptom{Noun} på{Prep} mangel{Noun} på{Prep} C-vitamin{Noun} (**17** ord ialt)

→ $Symptom\ [\ CBY : Symptom\ [\ CBY : Symptom\ [\ CBY :\ ...$
$...\ Symptom\ [\ CBY : Lack\ [\ WRT : VitaminC\ ]\ ...\ ]$

	# lingv. syntakstræer	køretid
Uden sem. begrænsning:	**1430**	$\sim$ **2.270.000** ms
Med sem. begrænsning:	**1**	$\sim$ **120** ms

symptom{Noun} på{Prep} .. symptom{Noun} på{Prep} mangel{Noun} på{Prep} C-vitamin{Noun} (**29** ord ialt)

→ $Symptom\ [\ CBY : Symptom\ [\ CBY : Symptom\ [\ CBY :\ ...$
$...\ Symptom\ [\ CBY : Lack\ [\ WRT : VitaminC\ ]\ ...\ ]$

	# lingv. syntakstræer	køretid
Uden sem. begrænsning:	**-**	**-**
Med sem. begrænsning:	**1**	$\sim$ **610** ms

De lange sætninger giver ikke umiddelbart mening, men er medtaget for at afprøve tekstanalysatoren på længere og tvetydige sætninger. Den kunstige sætningskonstruktion er nødvendig, da den beskedne størrelse af den her anvendte begrebsmæssige grammatik ikke gør det muligt at danne længere meningsfyldte sætninger.

For analyser foretaget uden semantisk begrænsning ses det, at køretiden øges væsentligt i takt med tvetydigheden af den givne sætning. Det er

hvad man kunne forvente, da alle de mulige syntakstræer i forhold til den lingvistiske grammatik opbygges fuldt ud, uanset om de giver anledning til en gyldig beskriver eller ej.

Sættes den semantiske begrænsning i funktion reduceres køretiden som det ses drastisk. Det skyldes selvfølgelig, at kun de lingvistiske syntakstræer, som også giver mening i forhold til den begrebsmæssige grammatik opbygges fuldt ud. De øvrige syntakstræer kasseres undervejs i syntaksanalysen efterhånden, som det fastslås, at de ikke fører til nogen beskriver. Derfor er det også for sætninger, som er tvetydige i forhold til den lingvistiske grammatik, hvor gevinsten er størst.

Som det ses, øges gevinsten i takt med et øget antal overflødige lingvistiske syntakstræer. For stærkt tvetydige sætninger af en vis længde vil forskellen være så stor, at sætningen i praksis ikke vil kunne analyseres uden den semantiske begrænsning. Den semantiske begrænsning formår således at reducere køretiden, for det ellers eksponentielt voksende problem, til en størrelse som er praktisk anvendelig.

Køretiderne for tekstanalysatoren med semantiske begrænsning må endvidere siges at være af en størrelsesorden, der kan accepteres i et søgesystem. Der skal dog tages et vist forbehold, da en konkret anvendelse vil medføre større grammatikker, hvilket selvsagt vil have en vis indflydelse på køretiden. Omvendt vil systemet også kunne optimeres væsentligt, hvis det ikke var møntet på eksperimenter med forskellige grammatikker og attributfunktioner, men i stedet var baseret på forud fastlagte grammatikker og attributfunktioner. Disse kunne derved i højere grad integreres i koden, med en øget effektivitet til følge.

Kapitel 7

Partialitet og robusthed

I forbindelse med en praktisk anvendelse af tekstanalysatoren vil man blive nødt til at overveje, hvordan den kan fungere uden for det beskyttede eksperimentielle miljø med trods alt kunstige eksempler. To væsentlige elementer i den sammenhæng er analysatorens evner i retning af de forbundne begreber; partialitet og robusthed.

Med partialitet menes evnen til at genkende brudstykker af sætninger, som det måske ikke kan lade sig gøre at genkende fuldt ud.

Med robusthed menes evnen til at gennemføre en analyse, og finde frem til et resultat trods uventede inddata.

Som det ses, er begreberne vagt defineret, men beskrivelserne giver alligevel en idé om nogle egenskaber, som er ønskelige og i nogen grad nødvendige for, at tekstanalysatoren ville kunne fungere godt i praksis.

I det følgende anvises og diskuteres nogle mulige tilgange til problemet.

7.1 Fjerne overflødig information

Som udgangspunkt klarer tekstanalysatoren sig relativt dårligt i forhold til de beskrevne begreber. Det skyldes blandt andet, at tekstanalysatoren er baseret på at indlæste sætninger, som udgangspunkt skal kunne genkendes

fuldt ud med hensyn til både den lingvistiske og den begrebsmæssige grammatik for, at et resultat i form af beskrivere findes.

Syntaksanalysen for et deltræ afbrydes således, hvis et symbol uden for den lingvistiske grammatik mødes, ligesom den semantiske begrænsning vil træde i funktion, hvis et ord, som ikke er repræsenteret i den begrebsmæssige grammatik, mødes.

En hel sætning kan således blive kasseret af årsager, der knytter sig til manglende partialitet og robusthed, selvom dele af sætningen måske godt kunne give anledning til en gyldig beskriver.

En umiddelbar løsning på problemet kunne være at fjerne de ord, hvis ordklasser ikke optræder i den lingvistiske grammatik forud for syntaksanalysen. Af samme grund fjernes de ord, som ikke er repræsenteret i den begrebsmæssige grammatik.

Ordene fjernes med det argument, at de er uden betydning i forhold til tekstanalysen, da deres tilstedeværelse alligevel aldrig ville kunne accepteres af den lingvistiske og begrebsmæssige grammatik. De vil derfor blot stå i vejen for en mulig genkendelse.

Den eneste hindring for genkendelse er så blot udformningen af den lingvistiske og den begrebsmæssige grammatik i forhold til den tilbageværende ordsekvens.

Et problem ved denne tilgang er, at den tilbageværende ordsekvens ikke nødvendigvis afspejler en naturligtsprogs-konstruktion. Dette kan selvsagt give problemer med genkendelsen i forhold til den lingvistiske grammatik, da fx et præpositionsled, der har fået fjernet selve præpositionen, ikke giver mening. Væsentlig lingvistisk information kan således risikere at blive kasseret.

Ord af visse ordklasser, som heller ikke er repræsenteret i den begrebsmæssige grammatik, kunne eventuelt få lov at blive, da visse sætningskonstruktioner eventuelt kan genkendes blot ud fra tilstedeværelsen af et ord af en bestemt ordklasse, uden at selve ordet anvendes i den semantiske analyse. Dette afgøres af, hvilken information den lingvistiske grammatik og de tilhørende attributfunktioner gør brug af.

7.2 Udvidelse af den lingvistiske grammatik

En anden mulighed for i højere grad at understøtte partialitet og robusthed er, at udvide den lingvistiske grammatik til at kunne håndtere flere ordklasser og sætningskonstruktioner. Også selvom de ikke direkte afspejles i de beskrivere, der opbygges på baggrund af den begrebsmæssige grammatik. Fx er artikler som "en" og "et" gode indikationer af et efterfølgende navneled, selvom de ikke er repræsenteret i eventuelle beskrivere for navneleddet.

På tilsvarende vis kunne grammatikken udvides med regler, der tillader ord, som ikke bidrager med nogen information. Begrundelsen for dette er, at gøre grammatikken mere robust ved at filtrere ord fra, som alligevel ikke kan genkendes, men uden at syntaksanalysen afbrydes. Fx som med følgende regler:

$$
\begin{aligned}
&S ::= NP \\
&S ::= S \;\; NP \\
&S ::= S \;\; WORD \;\; NP \\
&WORD ::= Prep \mid Verb \mid RelPron \mid Adj \mid Article \mid ...
\end{aligned}
$$

I attributfunktionerne for den tredje regel ignoreres bidraget fra $WORD$ blot, hvorved det genkendte $WORD$ ikke får nogen indflydelse på de dannede beskrivere. Den syntetiserede værdi for den anden og tredje regel er derfor den samme.

De viste regler forbedrer også partialiteten, da sætningen kan deles op i mindre dele, som hver især kan genkendes som navneled.

Et grundliggende problem ved at udvide den lingvistiske grammatik, som beskrevet, er at det i nogen grad udvander den semantiske begrænsning. Det skyldes, at den øgede robusthed i form af svagere krav til den lingvistiske udformning af accepterede sætninger medfører en øget tvetydighed. Den øgede tvetydighed og det dermed øgede antal syntakstræer for gyldige sætninger kan ikke umiddelbart holdes nede af den begrebsmæssige grammatik, da de mange nye syntakstræer er gyldige i forhold til både den lingvistiske og begrebsmæssige grammatik. Der vil således blive dannet mange overflødige syntakstræer, som vil give anledning til de samme beskrivere. Jo længere sætninger, jo værre vil problemet selvsagt være.

7.3 Modifikationer af tekstanalysatoren

En tredje mulighed er at tilpasse tekstanalysatoren på forskellige måder. Fx kunne den ændres til ikke at virke på sætninger af en defineret længde, men blot stoppe, når den ikke kan genkende længere. Det hidtil fundne resultat i form af beskrivere kunne så returneres, hvorefter der startes forfra med en ny tekstanalyse fra den pågældende position i teksten. Metoden vil til en vis grad sikre partialitet, men vil også kunne splitte sammenhænge unødigt op.

Modifikationer af tekstanalysatoren kunne også anvendes i kombination med de netop beskrevne tilgange.

Fx kunne problemet med den store tvetydighed reduceres ved at udvide den semantiske begrænsning til at omfatte en yderligere begrænsning på, hvilke deltræer der skal arbejdes videre med. Derved kunne genkendelsen af mindre dele fx stoppes, hvis de allerede indgår i genkendelsen af en større del af sætningen. Der er fx ingen interesse i beskriverne $Lack$ og $Vitamin$, hvis $Lack$ [$WRT : Vitamin$] er genkendt i et andet syntakstræ for den samme ordsekvens.

Kapitel 8

Konklusion

8.1 Tekstanalysatoren

Den beskrevne tekstanalysator formår som ønsket, at danne beskrivelser af det semantiske indhold af sætningsled som er genkendt med hensyn til en abstraheret lingvistisk grammatik for dansk. Det semantiske indhold beskrives ved hjælp af en begrebsmæssig grammatik, som samtidig tjener som en specifikation af en domæneontologi for det videndomæne, som sætningerne omhandler.

Tekstanalysatoren lever derfor op til de forventninger der blev stillet til den indledningsvis. Den kan derfor også anvendes som grundstenen i et ontologibaseret tekstsøgesystem, hvor søgning baseres på sammenligning af semantiske beskrivelser frem for på ordforekomster.

Tekstanalysatoren er endvidere konstrueret på en måde, så de semantiske beskrivelser opbygges sideløbende med den lingvistiske genkendelse af sætningsleddene. Den semantiske analyse af sætningsleddene er således ikke afhængig af fuldt genkendte lingvistiske syntakstræer, men udføres under opbygningen af dem.

Denne konstruktion baner vejen for at give den semantiske genkendelse indflydelse på analysens forløb i form af den indførte semantiske begrænsning. Den semantiske begrænsning udnytter således den opbyggede semantiske information til at afgrænse afsøgningen af de lingvistiske syntakstræer. Den

begrebsmæssige grammatik får derved også indflydelse på forløbet af tekstanalysen.

Den semantiske begrænsnings styrke er at den reducerer køretiden væsentligt for sætninger med en stærkt tvetydig lingvistisk opbygning. Det ellers eksponentielt voksende problem er således reduceret til en størrelse der lader sig håndtere i praksis.

Desuden gør den semantiske begrænsning det muligt at arbejde med stærkt tvetydige lingvistiske grammatikker, som ellers ikke ville kunne anvendes på grund af kompleksiteten. Der er således en øget frihed i designet af den lingvistiske grammatik.

8.2 Den implementerede prototype

Den implementerede prototype gør det muligt at afprøve tekstanalysatoren med udgangspunkt i forskellige begrebsmæssige og lingvistiske grammatikker. Systemet udgør således et udviklingsmiljø, hvor der kan eksperimenteres med vilkårlige udformninger af de to grammatikker som styrer tekstanalysatoren. Eneste begrænsning i brugergrænsefladens generalitet er at attributfunktionerne for den lingvistiske grammatik kun kan angives indenfor rammerne af de definerede basisfunktioner.

Systemet kan endvidere bidrage til en øget forståelse af hvordan ændringer i grammatikkerne påvirker tekstanalysens forløb, da systemet kan levere relevante uddata, som fx de lingvistiske syntakstræer som opbygges under en tekstanalyse.

Den modulære opbygning af koden gør det endvidere relativt simpelt at udvide systemet med nye komponenter eller at udskifte fx tekstanalysatoren eller basisfunktionerne.

8.3 Sammenligning med LKB

Sammenlignes tekstanalysatoren med det beskrevne LKB-system, som kan anvendes til tilsvarende tekstanalyser er der flere væsentlige forskelle.

Først og fremmest er LKB-systemet baseret på en anderledes underliggende struktur, *typed feature structures*. Det medfører en relativt besværlig

syntaks for angivelse af den lingvistiske grammatik og domæneontologien. I modsætning til dette står den her anvendte BNF-notation, som må siges at være simpel og overskuelig, samtidig med at den fuldt ud er i stand til at beskrive de relevante strukturer.

For den der skal opbygge en ontologi for et videndomæne må man formode at BNF-notationen klart vil være at foretrække.

Som beskrevet i afsnit 4.2.4 lader den ontologiske struktur sig heller ikke umiddelbart beskrive ved hjælp *typed feature structures*, hvilket blot understreger at LKB primært er beregnet på lingvistiske grammatikker kombineret med leksikal information og ikke på at arbejde med semantisk information i form af domæneontologier.

8.4 Videre arbejde

I forbindelse med tekstanalystoren vil det være oplagt at arbejde videre med hvordan begreberne partialitet og robusthed kan integreres i højere grad, hvilket vil kunne øge anvendeligheden af tekstanalysatoren.

For en afprøvning på et større eksempelmateriale vil der også kunne arbejdes med udbygninger af den lingvistiske og den begrebsmæssige grammatikker, hvilket måske vil kræve at kompetencer udenfor den tekniske verden inddrages.

Hvad angår den implementerede prototype kunne den også danne udgangspunkt for en udvidelse der omfatter et databasemodul og en søger, der er i stand til at sammenligne beskrivere. Derved vil det eksperimentielle miljø kunne udvides til at omfatte alle dele af et ontologibaseret søgesystem.

8.5 Afsluttende bemærkning

Afslutningsvis kan jeg erklære mig tilfreds med de opnåede resultater samt med mit personlige udbytte af arbejdet med projektet, der har været både spændende og udfordrernde.

Litteratur

[1] Jay Earley, *An Efficient Context-Free Parsing Algorithm*, Communications of the ACM, Volume 13, Number 2, 1970.

[2] Dick Grune & Ceriel Jacobs, *Parsing Techniques A Practical Guide*, Ellis Horwood Limited, 1998.

[3] Troels Andreasen, Jørgen Fischer Nilsson, & Hanne Erdman Thomsen, *Ontology-Based Querying*, Flexible Query Answering Systems, FQAS'2000, Physica-Verlag, 2000.

[4] Jørgen Fischer Nilsson, *Conceptual Grammar*, Internal working paper for OntoQuery, 2001.

[5] Patrizia Paggio, Bolette S. Pedersen and Dorte Haltrup, *Applying Language Technology to Content-based Querying*, Center for Sprogteknologi, København, 2001.

[6] Harry R. Lewis & Christos H. Papadimitriou, *Elements of the theory of computation*, Prentice-Hall, 1998.

[7] Alfred V. Aho, Ravi Sethi & Jeffrey D. Ullman, *Compilers, Principle, Techniques and Tools*, Addison-Wesley, 1986.

[8] Jørgen Fischer Nilsson & Troels Andreasen, *Grammatical Specification of Domain Ontologies*, 2001.

[9] Jørgen Fischer Nilsson, *A Logico-Algebraic Framework for Ontologies*, Ontology-based Interpretation of Noun Phrases, Proceedings of the

First International OntoQuery Workshop, Syddansk Universitet 2001.

[10] Jørgen Fischer Nilsson, *Concept Descriptions for Text Search*, Proceedings of the 11th European-Japanese Conference on Information Modelling and Knowledge Bases, University of Maribor, Slovenia, 2001.

[11] Hans Bruun, *Forelæsningsnoter til faget Avancerede Algoritmer*, DTU, 2000.

[12] Hans Bruun, *Forelæsningsnoter til faget Oversætterteknik*, DTU, 1987.

[13] Frank J. Oles, *An application of lattice theory to knowledge representation*, Theoretical Computer Science 249, 2000.

[14] Ann Copestake and others, *The (new) LKB system*, http:/www-csli.stanford.edu/ aac/lkb.html, 2000.

Bilag A

Brugervejledning

A.1 Programvinduet

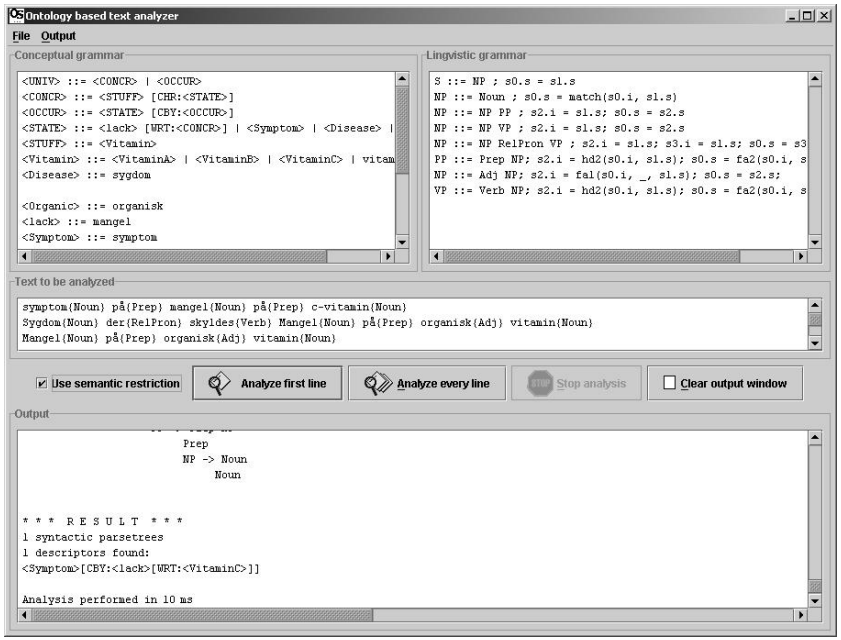
Ved start af programmet åbnes programvinduet, som kan ses i figur A.1. Vinduet er inddelt i fire felter: Et felt til den begrebsmæssige grammatik, et felt til den lingvistiske grammatik, et felt til sætninger som ønskes analyseret og et felt til resultatet af tekstanalyser og øvrige uddata i forbindelse med tekstanalysen. Derudover indeholder vinduet et antal knapper til følgende funktioner:

- Start af tekstanalyse af enkelt sætning
- Start af tekstanalyse af alle sætninger
- Standsning af igangværende tekstanalyse
- Ryd output-vinduet

Derudover angiver et flueben om den semantiske begrænsning ønskes anvendt.

A.2 Angivelse af grammatikker og sætninger

Den begrebsmæssige grammatik angives som den er. Et eksempel ses i figur 4.3. Startsymbolet er det første symbol i grammatikken.



Figur A.1: Programvinduet

Den lingvistiske grammatik angives med kun en enkelt regel pr. linie efterfulgt af et semikolon, hvorefter attributfunktionerne for reglen angives. Reglerne adskilles af semikolon. En attributværdi for et symbol angives ved et "s" efterfulgt af symbolets position. 0 er symbolet i venstresiden af reglen, 1 er det første symbol i højresiden o.s.v. Positionen efterfølges af et punktum, samt et "s" eller "i", som angiver om det er den syntetiserede eller nedarvede attributværdi. Attributværdier angives på denne måde både når de skal tildeles en attributfunktion og når de indgår som argumenter i en attributfunktion. Et eksempel på en lingvistisk grammatik med attributfunktioner ses nedenfor.

```
S ::= NP ; s0.s = s1.s
NP ::= Noun ; s0.s = Match(s0.i, s1.s)
NP ::= NP PP ; s2.i = s1.s; s0.s = s2.s
NP ::= NP VP ; s2.i = s1.s; s0.s = s2.s
NP ::= NP RelPron VP ; s2.i = s1.s; s3.i = s1.s;
                       s0.s = s3.i
PP ::= Prep NP ; s2.i = HD2(s0.i, s1.s);
                 s0.s = FA2(s0.i, s1.s, s2.s)
NP ::= Adj NP ; s2.i = FA1(s0.i, _, s1.s); s0.s = s2.s
VP ::= Verb NP ; s2.i = HD2(s0.i, s1.s);
                 s0.s = FA2(s0.i, s1.s, s2.s)
NP ::= Noun Noun ; s0.s = FA1(Match(s0.i, s2.s), _, s1.s)
```

I angivelsen af attributfunktionerne er følgende basisfunktioner til rådighed:

- `FeatureAccept` 1 og 2
- `Match`
- `HeadDerive` 1 og 2

Basisfunktionerne er beskrevet nærmere i afsnit 4.5.

Som det ses kan basisfunktionerne også angibes i forkortet form som `HD1`, `HD2`, `FA1` og `FA2`.

Da alle operationer returnerer mængder af beskrivere, kan de også anvendes som argumenter i andre operationer, som vist nedenfor.

```
NP ::= Noun Noun; s0.s = FA1(Match(s0.i, s2.s), _, s1.s)
```


Reglen kan anvendes til sammensatte ord der er blevet splittet op. Fx vil sætningen vitamin{Noun} mangel{Noun} kunne genkendes ved hjælp af reglen.

I angivelsen af attributfunktionerne skal man huske på at grammatikken skal være *leftattributed*.

A.3 Filer

I *File*-menuen er det muligt at hente grammatikker og sætninger fra filsystemet, ligesom det er muligt at gemme de grammatikker og sætninger der aktuelt arbejdes med. Indholdet af output-vinduet kan desuden gemmes i en fil. Filformatet er tekstbaseret, så grammatikker og tekster behøver ikke at være oprettet i programmets brugergrænseflade, men kan være dannet i enhver teksteditor.

A.4 Output

I *Output*-menuen vælges hvilke informationer om tekstanalysen, som ønskes skrevet ud i output-vinduet undervejs i tekstanalyser. Der er følgende muligheder:

- Vis lingvistiske syntakstræer
- Vis staktilstande
- Vis syntaksanalyse information

Udover disse valgfri oplysninger, afsluttes enhver tekstanalyse altid med resultatet i form af fundne beskrivere, samt antallet af fuldt afledte syntakstræer. Desuden vises den forbrugte tid i milisekunder.

Bilag B

Kildekode

B.1 GUI.java

```
import javax.swing.*;
import javax.swing.text.*;
import java.awt.*;
import java.io.*;
import java.awt.event.*;
import java.util.*;

/**
 * User interface and controlling thread of program
 */
public final class GUI {

  /**
   * variables controlling output information
   */
  public static boolean outputControl[] = new boolean[5];
  public static final int SHOWNOTHING = 0;
  public static final int SHOWALWAYS = 1;
  public static final int SHOWLINGVTREE = 2;
  public static final int SHOWSTACKSTATE = 3;
  public static final int SHOWPARSEINFO = 4;
```



```
/**
 * elements of user interface
 */
public static JFrame mainFrame, textFrame;
public static JTextArea semGrammarTA;
public static JTextArea synGrammarTA;
public static JTextArea textTA;
public static JButton parseButton, parseButton2, stopButton,
                      clearButton;
public static JCheckBox semParseCheckBox;
public static JTextArea outTA;
public static JMenuBar menuBar;
public static JMenu filemenu, loadsubmenu, savesubmenu,
                    outputmenu;
public static JMenuItem loadSemGramMenuItem,
              loadLingvGramMenuItem, loadInputTextMenuItem,
              saveSemGramMenuItem, saveLingvGramMenuItem,
              saveInputTextMenuItem, saveOutputTextMenuItem;
public static JCheckBoxMenuItem cbOutputMenuItem1,
                                cbOutputMenuItem2,
                                cbOutputMenuItem3,
                                cbOutputMenuItem4;
public static final JFileChooser fc = new JFileChooser();

/**
 * related textanalyzer
 */
private static SRTRBFBUParser srtrbfbup;

/**
 * related event listener
 */
private static GUIListener guiListener = new GUIListener();

/**
 * active lingvistic grammar
 */
private static Grammar lingvisticGrammar;

/**
 * active conceptual grammar
 */
```

```
private static ConceptGrammar semanticGrammar;

/**
 * open files
 */
public static File lastOpenedTextFile, lastOpenedLingvGramFile,
                   lastOpenedSemGramFile;

/**
 * reflects type of textanalysis
 */
public static boolean semanticParse = false;

/**
 * used in stopping execution of textanalysis
 */
public static boolean stopParse = false;

/**
 * used in measuring time consume of textanalysis
 */
private static long t1,t2;

/**
 * starts new textanalysis
 *
 * @param ips  sentence to be analyzed
 * @return   Thread runnig the analysis
 */
public static Thread parse(InputString ips) {
  writeToOutput("\nNEW ANALYSIS STARTED", SHOWALWAYS);

  writeToOutput("Input: "+ ips.printToString(), SHOWALWAYS);

  Vector reductionStacks = null;

  srtrbfbup.setParseArguments(lingvisticGrammar, semanticGrammar,
                              ips);

  stopParse = false;
  stopButton.setEnabled(true);
  parseButton.setEnabled(false);
  parseButton2.setEnabled(false);
```



```
  Thread trd = new Thread(srtrbfbup);
  t1 = System.currentTimeMillis();
  trd.start();
  return trd;

}


/**
 * called when first line parse is started
 */
public static void parseFirstLine() {

  updateSyntacticGrammar();
  updateSemanticGrammar();

  InputString ips = new InputString(lingvisticGrammar,
                                    semanticGrammar,
                                    getFirstInputLine());
  if (!ips.isCorrupted()) {
    parse(ips);
  } else {
    writeToOutput("Problems understanding input string.\nParsing
                  of string aborted.\n", SHOWPARSEINFO);
  }
}

/**
 * called when parse of every line is started
 */
public static void parseAll() {

  updateSyntacticGrammar();
  updateSemanticGrammar();

  Vector strings = getVectorOfInputLines();
  for (Enumeration e = strings.elements(); e.hasMoreElements();) {
    InputString ips = new InputString(lingvisticGrammar,
                                      semanticGrammar,
                                      (StringTokenizer)e.nextElement());
    if (!ips.isCorrupted()) {
```

```
      Thread trd = parse(ips);
      try {
        trd.join();
      } catch(InterruptedException ie) {};

    } else {
      writeToOutput("Problems understanding input string.\nParsing
                    of string aborted.\n", SHOWPARSEINFO);
    }
  }
}

/**
 * called when parsing has ended
 */
public static void parsingEnded() {
  long t2 = System.currentTimeMillis();
  stopButton.setEnabled(false);
  parseButton.setEnabled(true);
  parseButton2.setEnabled(true);

  if (GUI.stopParse) return;

  Vector reductionStacks = srtrbfbup.getReductionStacks();

  if (reductionStacks == null) {
    writeToOutput("Parsing stopped without result.", SHOWALWAYS);
  } else {
    String resultString = "";
    int numOfDescriptors = 0;

    for (Enumeration e = reductionStacks.elements();
         e.hasMoreElements(); ) {

      Stack st = (Stack) e.nextElement();

      SRTRBFBUParser.StackElement se =
                          (SRTRBFBUParser.StackElement) st.peek();

      writeToOutput("\nNumber of descriptors for lingvistic
                    parsetree: "+
                    se.getPossibleParses().size() +
                    "\n", SHOWLINGVTREE);
```



```
      for (Enumeration e2 =
                    ((Vector)se.getPossibleParses()).elements();
                    e2.hasMoreElements(); ) {

        String desc = ((ConceptGrammar.SemParsing)
                     e2.nextElement()).printToString();

        resultString += desc+"\n";
        numOfDescriptors++;
        writeToOutput( desc + "\n", SHOWLINGVTREE);
        writeToOutput("Syntactic parse tree\n", SHOWLINGVTREE);
      }

      while (!st.empty()) {
        se = (SRTRBFBUParser.StackElement) st.pop();
        writeToOutput(se.printParseTreeToString(), SHOWLINGVTREE);
      }
    }
    writeToOutput("\n* * *  R E S U L T  * * *", SHOWALWAYS);
    writeToOutput("\n"+(reductionStacks.size())+
                  " syntactic parsetrees", SHOWALWAYS);
    writeToOutput("\n"+numOfDescriptors+" descriptors found:" +
                  "\n"+resultString, SHOWALWAYS);
    writeToOutput("\nAnalysis performed in "+ (t2-t1) + " ms\n\n",
                  SHOWALWAYS);
  }
}

/**
 * clears output window
 */
public static void clearOutputWindow() {
  outTA.setText("");
}

/**
 * loads lingvistic grammar-file
 *
 * @param file  file to open
 */
public static void loadSyntacticGrammar(File file) {
  String grammarString = FileHandler.readFileToString(file);
```

```
  synGrammarTA.setText(grammarString);
}

/**
 * saves content of lingvistic grammar window
 *
 * @param file  file to save to
 */
public static void saveSyntacticGrammar(File file) {
  FileHandler.writeStringToFile(file, synGrammarTA.getText());
}

/**
 * updates instance of lingvistic grammar based on content of
 * window with lingvistic grammar
 *
 * @return
 */
public static void updateSyntacticGrammar() {
  lingvisticGrammar =
            FileHandler.readGrammarString(synGrammarTA.getText());
  lingvisticGrammar.print();
}

/**
 * loads conceptual grammar-file
 *
 * @param file  file to open
 */
 public static void loadSemanticGrammar(File file) {
  String grammarString = FileHandler.readFileToString(file);
  semGrammarTA.setText(grammarString);
}

/**
 * saves content of conceptual grammar window
 *
 * @param file  file to save to
 */
 public static void saveSemanticGrammar(File file) {
  FileHandler.writeStringToFile(file, semGrammarTA.getText());
}
```



```
/**
 * updates instance of conceptual grammar based on content of
 * window with conceptual grammar
 *
 * @return
 */
public static void updateSemanticGrammar() {
  semanticGrammar = FileHandler.readConceptGrammarString(
                                          semGrammarTA.getText());
  semanticGrammar.print();
}

/**
 * loads text-file to sentence window
 *
 * @param file  file to open
 */
 public static void loadText(File file) {
  String sText = FileHandler.readFileToString(file);
  lastOpenedTextFile = file;
  textTA.setText(sText);
}

/**
 * saves content of sentence window
 *
 * @param file  file to save to
 */
public static void saveText(File file) {
  FileHandler.writeStringToFile(file, textTA.getText());
}

/**
 * saves content of output window
 *
 * @param file  file to save to
 */
public static void saveOutputText(File file) {
  FileHandler.writeStringToFile(file, outTA.getText());
}

/**
 * gets first line of input from sentence window
```

```
 *
 * @return  StringTokenizer with input line
 */
public static StringTokenizer getFirstInputLine() {
  return FileHandler.readExpressionString(textTA.getText());
}

/**
 * gets every line of input from sentence window
 *
 * @return  Vector of StringTokenizers with input lines
 */
public static Vector getVectorOfInputLines() {
  return FileHandler.readExpressionStrings(textTA.getText());
}

/**
 * writes text to output window
 *
 * @param str   text to be written
 * @param type  type of message to be written to output window
 */
public static void writeToOutput(String str, int type) {
  if (outputControl[type] &&
      !outputControl[SHOWNOTHING] &&
      !stopParse) {
    outTA.append(str);
  }
  outTA.setCaretPosition(outTA.getText().length());
}

/**
 * initializes program window
 */
public static void init() {

  srtrbfbup = new SRTRBFBUParser();

  mainFrame = new JFrame("Ontology based text analyzer");
  Image w = Toolkit.getDefaultToolkit().getImage("logo.gif");
  mainFrame.setIconImage(w);

  outputControl[SHOWALWAYS] = true;
```



```
outputControl[SHOWNOTHING] = false;

// File menu
menuBar = new JMenuBar();

filemenu = new JMenu("File");
filemenu.setMnemonic(KeyEvent.VK_F);
menuBar.add(filemenu);

// Open
loadsubmenu = new JMenu("Open");

ImageIcon openIcon = new ImageIcon("open.gif");

loadSemGramMenuItem = new JMenuItem("Semantic grammar",
                                    openIcon);
loadSemGramMenuItem.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_1, ActionEvent.ALT_MASK));
loadSemGramMenuItem.setActionCommand("LoadSemGrammar");
loadSemGramMenuItem.addActionListener( guiListener );
loadsubmenu.add(loadSemGramMenuItem);

loadLingvGramMenuItem = new JMenuItem("Lingvistic grammar",
                                      openIcon);
loadLingvGramMenuItem.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_2, ActionEvent.ALT_MASK));
loadLingvGramMenuItem.setActionCommand("LoadLingvGrammar");
loadLingvGramMenuItem.addActionListener( guiListener );
loadsubmenu.add(loadLingvGramMenuItem);

loadInputTextMenuItem = new JMenuItem("Input text", openIcon);
loadInputTextMenuItem.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_3, ActionEvent.ALT_MASK));
loadInputTextMenuItem.setActionCommand("LoadInputText");
loadInputTextMenuItem.addActionListener( guiListener );
loadsubmenu.add(loadInputTextMenuItem);

filemenu.add(loadsubmenu);


// Save
savesubmenu = new JMenu("Save");
```

```
ImageIcon saveIcon = new ImageIcon("save.gif");

saveSemGramMenuItem = new JMenuItem("Semantic grammar",
                                    saveIcon);
saveSemGramMenuItem.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_1, ActionEvent.ALT_MASK));
saveSemGramMenuItem.setActionCommand("SaveSemGrammar");
saveSemGramMenuItem.addActionListener( guiListener );
savesubmenu.add(saveSemGramMenuItem);

saveLingvGramMenuItem = new JMenuItem("Lingvistic grammar",
                                      saveIcon);
saveLingvGramMenuItem.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_2, ActionEvent.ALT_MASK));
saveLingvGramMenuItem.setActionCommand("SaveLingvGrammar");
saveLingvGramMenuItem.addActionListener( guiListener );
savesubmenu.add(saveLingvGramMenuItem);

saveInputTextMenuItem = new JMenuItem("Input text", saveIcon);
saveInputTextMenuItem.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_3, ActionEvent.ALT_MASK));
saveInputTextMenuItem.setActionCommand("SaveInputText");
saveInputTextMenuItem.addActionListener( guiListener );
savesubmenu.add(saveInputTextMenuItem);

saveOutputTextMenuItem = new JMenuItem("Output window content",
                                       saveIcon);
saveOutputTextMenuItem.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_4, ActionEvent.ALT_MASK));
saveOutputTextMenuItem.setActionCommand("SaveOutputText");
saveOutputTextMenuItem.addActionListener( guiListener );
savesubmenu.add(saveOutputTextMenuItem);

filemenu.add(savesubmenu);


// Output menu
outputmenu = new JMenu("Output");
outputmenu.setMnemonic(KeyEvent.VK_O);

menuBar.add(outputmenu);

cbOutputMenuItem1 = new JCheckBoxMenuItem("Show lingvistic
```



```
                                          parsetree");
cbOutputMenuItem1.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_A, ActionEvent.ALT_MASK));
cbOutputMenuItem1.setSelected(true);
outputControl[SHOWLINGVTREE] = true;
cbOutputMenuItem1.addItemListener( guiListener );
outputmenu.add(cbOutputMenuItem1);

cbOutputMenuItem2 = new JCheckBoxMenuItem("Show stack state");
cbOutputMenuItem2.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_A, ActionEvent.ALT_MASK));
cbOutputMenuItem2.setSelected(true);
outputControl[SHOWSTACKSTATE] = true;
cbOutputMenuItem2.addItemListener( guiListener );
outputmenu.add(cbOutputMenuItem2);

cbOutputMenuItem3 = new JCheckBoxMenuItem("Show parse
                                          information");
cbOutputMenuItem3.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_A, ActionEvent.ALT_MASK));
cbOutputMenuItem3.setSelected(true);
outputControl[SHOWPARSEINFO] = true;
cbOutputMenuItem3.addItemListener( guiListener );
outputmenu.add(cbOutputMenuItem3);

cbOutputMenuItem4 = new JCheckBoxMenuItem("Show all");
cbOutputMenuItem4.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_A, ActionEvent.ALT_MASK));
cbOutputMenuItem4.setSelected(true);
cbOutputMenuItem4.addItemListener( guiListener );
outputmenu.add(cbOutputMenuItem4);

mainFrame.setJMenuBar(menuBar);

// Semantic grammar textArea
semGrammarTA = new JTextArea("");
semGrammarTA.setFont(new Font("Courier", Font.PLAIN, 12));
semGrammarTA.setMargin(new Insets(1,5,1,5));

JScrollPane semAreaScrollPane = new JScrollPane(semGrammarTA);
semAreaScrollPane.setVerticalScrollBarPolicy(
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
semAreaScrollPane.setPreferredSize(new Dimension(450, 250));
```

```
semAreaScrollPane.setBorder(
  BorderFactory.createCompoundBorder(
    BorderFactory.createCompoundBorder(
      BorderFactory.createTitledBorder("Conceptual grammar"),
      BorderFactory.createEmptyBorder(5,5,5,5)),
    semAreaScrollPane.getBorder()));


// Lingvistic grammar textArea
synGrammarTA = new JTextArea("");
synGrammarTA.setFont(new Font("Courier", Font.PLAIN, 12));
synGrammarTA.setMargin(new Insets(1,5,1,5));

JScrollPane synAreaScrollPane = new JScrollPane(synGrammarTA);
synAreaScrollPane.setVerticalScrollBarPolicy(
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
synAreaScrollPane.setPreferredSize(new Dimension(450, 250));
synAreaScrollPane.setBorder(
  BorderFactory.createCompoundBorder(
    BorderFactory.createCompoundBorder(
      BorderFactory.createTitledBorder("Lingvistic grammar"),
      BorderFactory.createEmptyBorder(5,5,5,5)),
    synAreaScrollPane.getBorder()));


// Text to be parsed textArea
textTA = new JTextArea("");
textTA.setFont(new Font("Courier", Font.PLAIN, 12));
textTA.setWrapStyleWord(true);
textTA.setMargin(new Insets(1,5,1,5));

JScrollPane textAreaScrollPane = new JScrollPane(textTA);
textAreaScrollPane.setVerticalScrollBarPolicy(
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
textAreaScrollPane.setPreferredSize(new Dimension(350, 100));
textAreaScrollPane.setBorder(
  BorderFactory.createCompoundBorder(
    BorderFactory.createCompoundBorder(
      BorderFactory.createTitledBorder("Text to be analyzed"),
      BorderFactory.createEmptyBorder(5,5,5,5)),
    textAreaScrollPane.getBorder()));
```



```
// Parse first line button
parseButton = new JButton("Analyze first line",
                          new ImageIcon("analyze1.gif"));
parseButton.setMnemonic(KeyEvent.VK_P);
parseButton.setActionCommand("ParseFirstLine");
parseButton.addActionListener( guiListener );

mainFrame.getRootPane().setDefaultButton(parseButton);

// Parse button
parseButton2 = new JButton("Analyze every line",
                           new ImageIcon("analyze2.gif"));
parseButton2.setMnemonic(KeyEvent.VK_A);
parseButton2.setActionCommand("ParseAll");
parseButton2.addActionListener( guiListener );

// Stop button
stopButton = new JButton("Stop analysis",
                         new ImageIcon("stop.gif"));
stopButton.setMnemonic(KeyEvent.VK_S);
stopButton.setActionCommand("stop");
stopButton.addActionListener( guiListener );
stopButton.setEnabled(false);

// Clear output window button
clearButton = new JButton("Clear output window",
                          new ImageIcon("clear.gif"));
clearButton.setMnemonic(KeyEvent.VK_C);
clearButton.setActionCommand("Clearow");
clearButton.addActionListener( guiListener );

//Enable sem parse check box
semParseCheckBox = new JCheckBox("Use semantic restriction");
semParseCheckBox.addItemListener( guiListener );

//Button panel
JPanel buttonPanel = new JPanel();

buttonPanel.add(semParseCheckBox);
buttonPanel.add(parseButton);
buttonPanel.add(parseButton2);
buttonPanel.add(stopButton);
buttonPanel.add(clearButton);
```

```
// Output textArea
outTA = new JTextArea("");
outTA.setFont(new Font("Courier", Font.PLAIN, 12));
outTA.setWrapStyleWord(true);
outTA.setMargin(new Insets(1,5,1,5));

JScrollPane outAreaScrollPane = new JScrollPane(outTA);
outAreaScrollPane.setVerticalScrollBarPolicy(
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
outAreaScrollPane.setPreferredSize(new Dimension(350, 300));
outAreaScrollPane.setBorder(
  BorderFactory.createCompoundBorder(
    BorderFactory.createCompoundBorder(
            BorderFactory.createTitledBorder("Output"),
            BorderFactory.createEmptyBorder(5,5,5,5)),
    outAreaScrollPane.getBorder()));

// Panel for textAreas
JPanel mainPane = new JPanel();

GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();
mainPane.setLayout(gridbag);

c.fill = GridBagConstraints.BOTH;
c.weightx = 1.0;
c.weighty = 1.0;

gridbag.setConstraints(semAreaScrollPane, c);
mainPane.add(semAreaScrollPane);

c.gridx = 1;

gridbag.setConstraints(synAreaScrollPane, c);
mainPane.add(synAreaScrollPane);

c.gridx = 0;
c.gridy = 1;
c.gridwidth = 2;
c.weighty = 0.2;

gridbag.setConstraints(textAreaScrollPane, c);
```



```
  mainPane.add(textAreaScrollPane);

  c.gridy = 2;
  c.fill = GridBagConstraints.NONE;
  c.weightx = 0.0;
  c.weighty = 0.0;

  gridbag.setConstraints(buttonPanel, c);
  mainPane.add(buttonPanel);

  c.gridy = 3;
  c.fill = GridBagConstraints.BOTH;
  c.weightx = 1.0;
  c.weighty = 1.0;

  gridbag.setConstraints(outAreaScrollPane, c);
  mainPane.add(outAreaScrollPane);

  mainFrame.getContentPane().add(mainPane);

  mainFrame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  });

  mainFrame.pack();
  mainFrame.setVisible(true);
}

/**
 * main method of program
 *
 * @param args  command line parameters
 */
public static void main(String[] args) {

  if (args.length > 0 && args[0].equals("-debug")) {
    dbg.setDebugMode(true);
  }

  init();
  lastOpenedLingvGramFile = new File("lingv.grm");
```

```
    lastOpenedSemGramFile = new File("sem.grm");
    lastOpenedTextFile = new File("text.exp");

    loadSyntacticGrammar(lastOpenedLingvGramFile);
    loadSemanticGrammar(lastOpenedSemGramFile);
    loadText(lastOpenedTextFile);
  }
}
```


B.2 Grammar.java

```
import java.util.*;

/**
 * A lingvistic grammar for use in textanalysis.
 */
public class Grammar {

  /**
   * Start symbol of grammar
   */
  private NonTerminal startSym;

  /**
   * The non-terminal symbols used in this grammar.
   */
  private Hashtable nonTerminals;

  /**
   * The terminal symbols used in this grammar.
   */
  private Hashtable terminals;

  /**
   * The production rules defining this grammar.
   */
  private Vector productionRules;

  /**
   * Creates new lingvistic grammar.
   *
   * @param startSym         The start symbol of the new grammar.
   * @param nonTerminals     Hashtabel containing the non-terminal
   *                         symbols of
   * @param terminals        Hashtable containing the terminal
   *                         symbols of the
   * @param productionRules  Vector containing the production
   *                         rules of the new
   */
  public Grammar(NonTerminal startSym, Hashtable nonTerminals,
                 Hashtable terminals, Vector productionRules) {
```

```
  this.startSym = startSym;
  this.nonTerminals = nonTerminals;
  this.terminals = terminals;
  this.productionRules = productionRules;
}

/**
 * Gets symbol and returns it if symbol exists in grammar.
 *
 * @param name  Name of the requested symbol.
 * @return      Requested symbol if found. Null othervise.
 */
public Symbol getSymbol(String name) {
  if (nonTerminals.containsKey(name)) {
    return (Symbol) nonTerminals.get(name);
  } else if (terminals.containsKey(name)) {
    return (Symbol) terminals.get(name);
  }
  return null;
}

/**
 * Finds left hand sides in productin rules which right hand side
 * matches the given symbols.
 *
 * @param rhs  Vector of symbols to be matched with right hand
 *             sides.
 * @return     Vector of symbols from left hand sides.
 */
public Vector rhsToLhs(Vector rhs) {

  Vector result = new Vector(1);
  for (Enumeration e = productionRules.elements();
       e.hasMoreElements(); ) {

    ProductionRule p = (ProductionRule) e.nextElement();

    if (p.rhsMatch(rhs)) {
      result.add(p.getLhs());
    }
  }
  return result;
```



```
}

/**
 * Gets startsymbol of grammar.
 *
 * @return  Startsymbol of grammar.
 */
public NonTerminal getStartSymbol() {
  return startSym;
}

/**
 * Gets production rules defining grammar.
 *
 * @return  Vector containing the production rules for this
 * grammar.
 */
public Vector getProductionRules() {
  return productionRules;
}

/**
 * Prints textual description of this lingvistic grammar to
 * System.out with use of static class dbg.
 */
public void print() {
  dbg.pp("Start symbol: ");
  startSym.print();
  dbg.pp("\n");
  dbg.p("Nonterminal symbols:");
  for (Enumeration e = nonTerminals.elements();
       e.hasMoreElements(); ) {

    NonTerminal nt = (NonTerminal) e.nextElement();
    nt.print();
    dbg.pp(", ");
  }
  dbg.pp("\n");
  dbg.p("Terminal symbols:");
  for (Enumeration e = terminals.elements();e.hasMoreElements();){
    Terminal t = (Terminal) e.nextElement();
    t.print();
    dbg.pp(", ");
```

```
    }
    dbg.pp("\n");
    dbg.p("Production rules:");
    for (Enumeration e = productionRules.elements();
         e.hasMoreElements(); ) {

      ProductionRule p = (ProductionRule) e.nextElement();
      dbg.pp("\n");
      p.print();
    }
    dbg.pp("\n");
  }
}
```


B.3 ProductionRule.java

```
import java.util.*;
import java.lang.reflect.*;

/**
 * Production rule in lingvistic grammar.
 */
public class ProductionRule {

  /**
   * Non terminal symbol in left hand side of this rule
   */
  private NonTerminal lhs;

  /**
   * Symbols in right hand side of this rule
   */
  private Vector rhs;

  /**
   * Attribute functions for this rule
   */
  private AttributeFunctionDef[] attributeFunctions;

  /**
   * Create ProductionRule with given left hand side and right
   * hand side.
   *
   * @param lhs  NonTerminal at left hand side.
   * @param rhs  Vector containing symbols at right hand side.
   */
  public ProductionRule(NonTerminal lhs, Vector rhs) {
    this.lhs = lhs;
    this.rhs = rhs;
    this.attributeFunctions = new AttributeFunctionDef[10];
  }

  /**
   * Adds definition of attribute function for this rule.
   *
   * @param funPos  int describing which function is defined.
```

```
 * @param funDef  Definition of attribute function.
 */
public void addFunDef(int funPos, AttributeFunctionDef funDef) {
  Array.set(attributeFunctions, funPos, funDef);
}

/**
 * Gets definition of attribute function for this rule.
 *
 * @param funNr  The number of the requested attribute function
 *               definition.
 * @return       Definition of requested attribute function.
 */
public AttributeFunctionDef getFunDef(int funNr) {
  return (AttributeFunctionDef)Array.get(attributeFunctions,
                                         funNr);
}

/**
 * Checks if sequence of symbols matches symbols in right hand
 * side of this rule.
 *
 * @param symbols  The sequence of symbels to be compared with
 *                 this.
 * @return         true if sequence of symbols equals right hand
 *                 side of this rule. false otherwise.
 */
public boolean rhsMatch(Vector symbols) {
  if (rhs.size() != symbols.size())
    return false;
  for (int i=0; i < rhs.size(); i++) {
    if (!rhs.get(i).equals(symbols.get(i))) {
      return false;
    }
  }
  return true;
}

/**
 * Gets symbol in right hand side of this rule.
 *
 * @param position  The position of the requested right hand side
 *                  symbol.
```



```
 * @return          The requested symbol from Vector rhs.
 */
public Symbol getRhsSymbol(int position) {
  if (position < rhs.size()) {
    return (Symbol) rhs.get(position);
  } else {
    return null;
  }
}

/**
 * Gets left hand side symbol of this rule.
 *
 * @return  The symbol lhs.
 */
public Symbol getLhs() {
  return lhs;
}

/**
 * Gets size of the right hand side of this rule.
 *
 * @return  The size of Vector rhs.
 */
public int rhsSize() {
  return rhs.size();
}

/**
 * Gets position of symbol in right hand size this rule.
 *
 * @param symbol  The requested symbol.
 * @return        The position of the first occurence of the
 *                symbol in
 * Vector rhs.
 */
public int rhsContains(Symbol symbol) {
  return ( rhs.indexOf(symbol) );
}

/**
 * Prints textual description of this rule to a String.
 *
```

```
 * @return  String containing a textual representation.
 */
public String printToString() {
  String result;
  result = lhs.printToString();
  result += " -> ";
  for (Enumeration e = rhs.elements(); e.hasMoreElements(); ) {
    result += ((Symbol) e.nextElement()).printToString();
    result += " ";
  }
  return result;
}

/**
 * Prints textual description of this rule to System.out with
 * use of method pp in class dbg.
 */
public void print() {
  try {
    lhs.print();
    dbg.pp(" -> ");
    for (Enumeration e = rhs.elements(); e.hasMoreElements(); ) {
      ((Symbol) e.nextElement()).print();
      dbg.pp(" ");
    }
  } catch(Exception e){}
}

/**
 * Definition of an attribute function. An operation and the
 * arguments to it. Arguments can be integers based on positions
 * in right hand side of corresponding production rule or
 * instances of this class if the argument value has to be
 * claculated by use of another operation.
 */
public class AttributeFunctionDef {

  /**
   * What operation. Possibilities defined in class
   * AttributeCalculation:
   */
  private int operation;
```



```
/**
 * Arguments to operation. Of type Integer or type
 * AttributeFunctionDef.
 */
private Vector arguments;

/**
 * Creates new AttributeFunctionDef.
 *
 * @param operation  The operation of the new instance.
 * @param arguments  The arguments for this operation.
 */
public AttributeFunctionDef(int operation, Vector arguments) {
  this.operation = operation;
  this.arguments = arguments;
}

/**
 * Adds new integer argument to Array arguments.
 *
 * @param argument  New argument.
 */
public void addArgument(int argument) {
  arguments.add(new Integer(argument));
}

/**
 * Adds new AttributeFunctionDef argument to Array arguments.
 *
 * @param argument  New argument.
 */
public void addArgument(AttributeFunctionDef argument) {
  arguments.add(argument);
}

/**
 * Gets operation of this AttributeFunctionDef.
 *
 * @return  variable operation.
 */
public int getOperation() {
  return operation;
}
```

```
    /**
     * Gets arguments of this AttributeFunctionDef.
     *
     * @return  variable arguments.
     */
    public Vector getArguments() {
      return arguments;
    }
  }
}
```


B.4 ConceptGrammar.java

```
import java.util.*;

/**
 * Represents conceptual grammar reflecting domain knowledge
 */
public class ConceptGrammar implements java.io.Serializable {

  /**
   * startsymbol of grammar
   */
  private NonTerminal startSym;

  /**
   * non terminal symbols used in grammar
   */
  private Hashtable nonTerminals;

  /**
   * terminal symbols used in grammar
   */
  private Hashtable terminals;

  /**
   * production rules ni grammar. Instances of ConceptProductionRule
   */
  private Vector productionRules;

  /**
   * Constructor creating new grammar
   *
   * @param startSym     start symbol
   * @param nonTerminals   non terminal symbols of new grammar
   * @param terminals    terminal symbols of new grammar
   * @param productionRules  production rules of new grammar
   */
  public ConceptGrammar(NonTerminal startSym,
                        Hashtable nonTerminals,
                        Hashtable terminals,
                        Vector productionRules) {
    this.startSym = startSym;
```

```
  this.nonTerminals = nonTerminals;
  this.terminals = terminals;
  this.productionRules = productionRules;
}

/**
 * default constructor
 *
 * @return
 */
public ConceptGrammar() {
}

/**
 * Finds symbol and returns it if symbol exists in grammar
 *
 * @param name  name of seeked symbol
 * @return    symbol if found, null otherwise
 */
public Symbol getSymbol(String name) {
  if (nonTerminals.containsKey(name)) {
    return (Symbol) nonTerminals.get(name);
  } else if (terminals.containsKey(name)) {
    return (Symbol) terminals.get(name);
  }
  return null;
}

/**
 * gets start symbol of grammar
 *
 * @return
 */
public NonTerminal getStartSymbol() {
  return startSym;
}

/**
 * gets production rules of grammar
 *
 * @return  requested production rules
 */
public Vector getProductionRules() {
```



```
  return productionRules;
}

/**
 * finds rule with a given left hand side
 *
 * @param lhs  non terminal symbol to look after in left hand
 *             sides
 * @return     rhe requested instance of ConceptProductionRule
 *             if found, null otherwise
 */
public ConceptProductionRule findRuleWithLhs(NonTerminal lhs) {
  for (Enumeration e = productionRules.elements();
       e.hasMoreElements(); ) {

    ConceptProductionRule p =
                      (ConceptProductionRule) e.nextElement();
    if (p.getLhs() == lhs) {
      return p;
    }
  }
  dbg.p("Error: ConceptProductionRule not found in grammar!");
  return null;
}

/**
 * prints textual representation of grammar with use of class dbg
 */
public void print() {
  dbg.p("Start symbol: ");
  startSym.print();
  dbg.p("");
  dbg.p("Nonterminal symbols:");
  for (Enumeration e = nonTerminals.elements();
       e.hasMoreElements(); ) {

    NonTerminal nt = (NonTerminal) e.nextElement();
    nt.print();
    dbg.pp(", ");
  }
  dbg.pp("\n");
  dbg.p("Terminal symbols:");
  for (Enumeration e = terminals.elements();
```

```
        e.hasMoreElements(); ) {

    Terminal t = (Terminal) e.nextElement();
    t.print();
    dbg.pp(", ");
  }
  dbg.p("");
  dbg.p("Production rules:");
  for (Enumeration e = productionRules.elements();
       e.hasMoreElements(); ) {
    ConceptProductionRule p =
                      (ConceptProductionRule) e.nextElement();
    dbg.pp("\n");
    p.print();
  }
  dbg.pp("\n");
}

/**
 * Gets symbol in left hand side of rule, with given right hand
 * side. Method used to get terminal concepts from terminal
 * strings in grammar.
 *
 * @param t  Terminal string
 * @return   The found terminal concept i form of NonTerminal
 *           instance.
 */
public NonTerminal getLhsTo(Terminal t) {
  for (Enumeration e = productionRules.elements();
       e.hasMoreElements(); ) {

    ConceptProductionRule p =
                     (ConceptProductionRule) e.nextElement();
    for (Enumeration e2 = p.getAlternatives().elements();
         e2.hasMoreElements(); ) {

      Alternative alt = (Alternative) e2.nextElement();
      if (alt.getConcept().isTerminalSymbol()) {
        if (alt.getConcept() == t) {
          return p.getLhs();
        }
      }
    }
```



```
  }
  return null;
}

/**
 * checks whether a concept is a subconcept of another
 *
 * @param sub      expected subconcept
 * @param concept  expected superconcept
 * @return         true if sub is subconcept of concept,
 *                 false otherwise
 */
public boolean isSubConceptOf(NonTerminal sub,
                              NonTerminal concept) {
  boolean res = false;
  ConceptProductionRule p = findRuleWithLhs(concept);
  for (Enumeration e = p.getAlternatives().elements();
       e.hasMoreElements(); ) {

    Alternative alt = (Alternative) e.nextElement();
    if (!alt.getConcept().isTerminalSymbol()) {
      if (alt.getConcept() == sub) {
        return true;
      } else {
        res = res ||
              isSubConceptOf(sub, (NonTerminal) alt.getConcept());
      }
    }
  }
  return res;
}

/**
 * Production rule in conceptual grammar. Notice: Only one
 * instance for each non terminal symbol in grammar.
 */
public class ConceptProductionRule
                                implements java.io.Serializable {

  /**
   * non terminal in left hand side of rule
   */
  private NonTerminal lhs;
```

```
/**
 * possible right hand sides of left hand side symbol
 */
private Vector alternatives;


/**
 * constructor
 *
 * @param lhs       left hand side symbol for new rule
 * @param alternatives  possible right hand sides
 */
public ConceptProductionRule(NonTerminal lhs,
                             Vector alternatives) {
  this.lhs = lhs;
  this.alternatives = alternatives;
}

/**
 * gets left hand side symbol
 *
 * @return  requested symbol
 */
public NonTerminal getLhs() {
  return lhs;
}

/**
 * gets possible right hand sides
 *
 * @return  Vector of Alternatives
 */
public Vector getAlternatives() {
  return alternatives;
}

/**
 * checks if this contains right hand side with symbol given
 *
 * @param s  symbol to look for
 * @return   true if symbol found, false otherwise
 */
```



```
  public boolean containsRhs(Symbol s) {
    for (Enumeration e = alternatives.elements();
         e.hasMoreElements(); ) {
      Alternative a = (Alternative) e.nextElement();
      if (a.getConcept() == s) return true;
    }
    return false;
  }

  /**
   * prints textual representation of rule with use of class dbg.
   */
  public void print() {
    lhs.print();
    dbg.pp(" ::= ");
    Enumeration e = alternatives.elements();
    ((Alternative) e.nextElement()).print();
    while (e.hasMoreElements()) {
      dbg.pp(" | ");
      ((Alternative) e.nextElement()).print();
    }
  }
}

/**
 * Represents right hand side of rule in conceptual grammar
 */
public class Alternative implements java.io.Serializable {


  /**
   * concept in right hand side
   */
  Symbol concept;

  /**
   * possible semantic relations
   */
  Vector features;

  /**
   * constructor
   *
```

```
 * @param concept   concept of new Alternative
 * @param features  possible semantic relations for new
 *                  Laternative
 */
public Alternative(Symbol concept, Vector features) {
  this.concept = concept;
  this.features = features;
}

/**
 * gets concept of this.
 *
 * @return  requested concept
 */
public Symbol getConcept() {
  return concept;
}

/**
 * gets possible semantic relations.
 *
 * @return  requested Vector
 */
public Vector getFeatures() {
  return features;
}

/**
 * prints this right hand side to string
 *
 * @return  textual representation of this.
 */
public String printToString() {
  String result;
  result = concept.printToString();
  for (Enumeration e = features.elements();
       e.hasMoreElements(); ) {

    result += ((Feature) e.nextElement()).printToString();
  }
  return result;
}
```



```
  /**
   * prints this right hand side with use of class dbg.
   */
  public void print() {
    concept.print();
    for (Enumeration e = features.elements();
         e.hasMoreElements(); ) {

      ((Feature) e.nextElement()).print();
    }
  }
}

/**
 * A possible semantic relation/feature
 */
public class Feature implements java.io.Serializable {

  /**
   * Role in semantic relation
   */
  NonTerminal role;

  /**
   * concept in semantic relation
   */
  Symbol concept;

  /**
   * constructor
   *
   * @param role     role for new semantic relation
   * @param concept  concept for new semantic relation
   */
  public Feature(NonTerminal role, Symbol concept) {
    this.role = role;
    this.concept = concept;
  }

  /**
   * gets role of this semantic relation
   *
   * @return  the role
```

```
 */
public NonTerminal getRole() {
  return role;
}

/**
 * gets concept of this semantic relation
 *
 * @return  the concept
 */
public Symbol getConcept() {
  return concept;
}

/**
 * gets copy of this.
 *
 * @return
 */
public Feature getCopy() {
  return new Feature(role, concept);
}

/**
 * prints this to string
 *
 * @return  textual representation of this.
 */
public String printToString() {
  String result;
  result = "[";
  result += role.printToString();
  result += ":";
  result += concept.printToString();
  result += "]";
  return result;
}

/**
 * prints this with use of class dbg.
 *
 */
public void print() {
```



```
    dbg.pp("[");
    role.print();
    dbg.pp(":");
    concept.print();
    dbg.pp("]");
  }

}

/**
 * Descriptor / sentence form in conceptual grammar
 */
public class SemParsing implements java.io.Serializable {

  /**
   * concept that led to this semantic parsing.
   */
  NonTerminal concept;

  /**
   * terminal concept beginning this semantic parsing
   */
  Symbol tConcept;

  /**
   * possible semantic relations
   */
  Vector possibleFeatures;

  /**
   * accepted semantic relations.
   */
  Vector acceptedFeatures;

  /**
   * tells if HeadDerive should be calculated before use
   */
  boolean bHeadDerive;

  /**
   * constructor
   *
   * @param concept           concept that led to this
```

```
 * @param tConcept          terminal concept of this
 * @param possibleFeatures  possible semantic relations
 * @param acceptedFeatures  accepted/recognized semantic
 *                          relations
 */
public SemParsing(NonTerminal concept, Symbol tConcept,
                  Vector possibleFeatures,
                  Vector acceptedFeatures) {
  this.concept = concept;
  this.tConcept = tConcept;
  this.possibleFeatures = possibleFeatures;
  this.acceptedFeatures = acceptedFeatures;
  this.bHeadDerive = false;
}

/**
 * constructor used if result of not executed HeadDerive
 * should be represented
 *
 * @param concept    concept to HeadDerive
 * @param bHeadDerive  marker of HeadDerive
 */
public SemParsing(NonTerminal concept, boolean bHeadDerive) {
  this.concept = concept;
  this.bHeadDerive = true;
}

/**
 * gets concept that led to this.
 *
 * @return  concept
 */
public NonTerminal getConcept() {
  return concept;
}

/**
 * gets terminal concept
 *
 * @return terminal concept
 */
public Symbol getTConcept() {
  return tConcept;
```



```
}

/**
 * gets possible semantic relations
 *
 * @return  requested Vector
 */
public Vector getPossibleFeatures() {
  return possibleFeatures;
}

/**
 * gets accepted semantic relations
 *
 * @return  requested Vector
 */
public Vector getAcceptedFeatures() {
  return acceptedFeatures;
}

/**
 * accepts possible semantic relation
 *
 * @param pf  accepted semantic relation
 */
public void acceptFeature(ParsedFeature pf) {
  acceptedFeatures.add(pf);
}

/**
 * chekcs if this represents result of not executed HeadDerive
 *
 * @return  true if it is, false otherwise
 */
public boolean isHeadDerive() {
  return bHeadDerive;
}

/**
 * gets copy of this descriptor / sentenceform
 *
 * @return  copy of this
 */
```

```
public SemParsing getCopy() {
  if (bHeadDerive) return new SemParsing(concept, true);
  Vector possibleFeaturesCopy =
                        new Vector(possibleFeatures.size());
  Vector acceptedFeaturesCopy =
                        new Vector(acceptedFeatures.size());
  for (Enumeration e = possibleFeatures.elements();
       e.hasMoreElements(); ) {

    Feature fea = (Feature) e.nextElement();
    possibleFeaturesCopy.add( fea );
  }
  for (Enumeration e = acceptedFeatures.elements();
       e.hasMoreElements(); ) {

    ParsedFeature fea = (ParsedFeature) e.nextElement();
    acceptedFeaturesCopy.add( fea.getCopy() );
  }
  return new SemParsing(concept, tConcept,
                        possibleFeaturesCopy,
                        acceptedFeaturesCopy);
}

/**
 * prints this to string.
 *
 * @return  textual representation of this.
 */
public String printToString() {
  if (bHeadDerive) return("-HD-");
  String result;
  result = tConcept.printToString();
  if (possibleFeatures.size() > 0 ||
      acceptedFeatures.size() > 0) {

    if (acceptedFeatures.size() > 0) {
      for (Enumeration e = acceptedFeatures.elements();
           e.hasMoreElements(); ) {

        ParsedFeature fea = (ParsedFeature) e.nextElement();
        result += fea.printToString();
      }
    }
```



```
    }
    return result;
  }

  /**
   * prints textual representation of this with use of class dbg.
   */
  public void print() {
    dbg.pp("\nSemParsing: ");
    if (bHeadDerive) dbg.pp("-HD-");
    else {
      dbg.pp(" concept: "); concept.print();
      dbg.pp(" tConcept: "); tConcept.print();
      dbg.pp(" Features:[");
      for (Enumeration e = possibleFeatures.elements();
           e.hasMoreElements(); ) {

        Feature fea = (Feature) e.nextElement();
        fea.print();
      }
      dbg.pp(" :: ");
      for (Enumeration e = acceptedFeatures.elements();
           e.hasMoreElements(); ) {

        ParsedFeature fea = (ParsedFeature) e.nextElement();
        fea.print();
      }
      dbg.pp("]");
    }
  }

  /**
   * prints textual representation of this without possible
   * semantic relations with use of class dbg.
   */
  public void prettyprint() {
    if (bHeadDerive) dbg.pp("-HD-");
    else {
      tConcept.print();
      if (possibleFeatures.size() > 0 ||
          acceptedFeatures.size() > 0) {

        if (acceptedFeatures.size() > 0) {
```

```
          for (Enumeration e = acceptedFeatures.elements();
               e.hasMoreElements(); ) {
            ParsedFeature fea = (ParsedFeature) e.nextElement();
            fea.print();
          }
        }
      }
    }
  }

}

/**
 * Parsed semantic relation
 */
public class ParsedFeature implements java.io.Serializable {

  /**
   * the parsed semantic relations
   */
  Feature feature;

  /**
   * the semantic parse that is part of this parsed semantic
   * relation.
   */
  SemParsing subParsing;

  /**
   * constructor
   *
   * @param feature     the parsed semantic relation
   * @param subParsing  the semantic parse that is part of this
   *                    parsed semantic relation
   */
  public ParsedFeature(Feature feature, SemParsing subParsing) {
    this.feature = feature;
    this.subParsing = subParsing;
  }

  /**
   * gets copy of this Feature
   *
```



```
   * @return  copy of this
   */
  public ParsedFeature getCopy() {
    return( new ParsedFeature(feature, subParsing.getCopy()) );
  }

  /**
   * prints this parsed semantic relation to string.
   *
   * @return  string with textual representation of this.
   */
  public String printToString() {
    String result;
    result = "[";
    result += feature.getRole().printToString();
    result += ":";
    result += subParsing.printToString();
    result += "]";
    return result;
  }

  /**
   * prints this parsed semantic relation with use of class dbg.
   */
  public void print() {
    dbg.pp("[");
    feature.getRole().print();
    dbg.pp(":");
    subParsing.prettyprint();
    dbg.pp("]");
  }
 }
}
```

B.5 Symbol.java

```
/**
 * A grammar symbol.
 */
public interface Symbol extends java.io.Serializable {

  /**
   * Prints textual description of this Symbol to System.out.
   */
  public void print();

  /**
   * Prints textual description of this Symbol to a String.
   *
   * @return  String containing textual representation of this
   * Symbol.
   */
  public String printToString();

  /**
   * Determines if this Symbol is a Terminal.
   */
  public boolean isTerminalSymbol();

  /**
   * Gets name of this Symbol.
   *
   * @return  Name of this terminal.
   */
  public String getName();
}
```


B.6 NonTerminal.java

```
/**
 * A non-terminal symbol.
 */
 public class NonTerminal implements Symbol, java.io.Serializable {

  /**
   * Name of the terminal symbol.
   */
  private String name;

  /**
   * Default constructor.
   */
  public NonTerminal() {
  }

  /**
   * Creates new NonTerminal.
   *
   * @param name  Name of NonTerminal.
   */
  public NonTerminal(String name) {
    this.name = name;
  }

  /**
   * Determines if NonTerminal equals this NonTerminal.
   *
   * @param sym  NonTerminal to compare with.
   * @return     true if NonTerminals equals
   */
  public boolean equals(NonTerminal sym) {
    return (name.equals(sym));
  }

  /**
   * Determines if Terminal equals this NonTerminal.
   *
   * @param sym  Terminal to compare with.
   * @return     false always.
```

```
 */
public boolean equals(Terminal sym) {
  return (false);
}

/**
 * Determines if name of NonTerminal equals name of this
 * NonTerminal.
 *
 * @param sym  The NonTerminal that should be compared
 *             with.
 * @return     true if names are equal
 */
public boolean nameEquals(NonTerminal sym) {
  return (name.compareTo(sym.getName()) == 0);
}

/**
 * Determines if this is a Terminal.
 *
 * @return    false always.
 */
public boolean isTerminalSymbol() {
  return false;
}

/**
 * Gets name of this Terminal.
 *
 * @return  Name of this terminal.
 */
public String getName() {
  return name;
}

/**
 * Prints textual description of this NonTerminal to a
 * String.
 *
 * @return The name of this NonTerminal.
 */
public String printToString() {
  return name;
```



```
  }

  /**
   * Prints textual description of this NonTerminal to
   * System.out with use of method pp in class dbg.
   */
  public void print() {
    dbg.pp(name);
  }
}
```

B.7 Terminal.java

```
/**
 * A terminal symbol.
 */
public class Terminal implements Symbol, java.io.Serializable{

  /**
   * Name of the terminal symbol.
   */
  private String name;

  /**
   * Default constructor.
   */
  public Terminal() {
  }

  /**
   * Creates new Terminal.
   *
   * @param name  Name of Terminal.
   */
  public Terminal(String name) {
    this.name = name;
  }

  /**
   * Determines if Terminal equals this Terminal.
   *
   * @param sym  Terminal to compare with.
   * @return     true if Terminals equals
   */
  public boolean equals(Terminal sym) {
    return (name.equals(sym));
  }

  /**
   * Determines if NonTerminal equals this Terminal.
   *
   * @param sym  NonTerminal to compare with.
   * @return     false always.
```



```
   */
  public boolean equals(NonTerminal sym) {
    return (false);
  }

  /**
   * Determines if name of Terminal equals name of this
   * Terminal.
   *
   * @param sym  The Terminal that should be compared with.
   * @return     true if names are equal
   */
  public boolean nameEquals(Terminal sym) {
    return ((name.compareTo(sym.getName()) == 0));
  }

  /**
   * Determines if this is a Terminal.
   *
   * @return  true always.
   */
  public boolean isTerminalSymbol() {
    return true;
  }

  /**
   * Gets name of this Terminal.
   *
   * @return  Name of this terminal.
   */
  public String getName() {
    return name;
  }

  /**
   * Prints textual description of this Terminal to a
   * String.
   *
   * @return The name of this Terminal.
   */
  public String printToString() {
    return name;
  }
```

```
  /**
   * Prints textual description of this Terminal to
   * System.out with use of method pp in class dbg.
   */
   public void print() {
     dbg.pp(name);
   }
}
```


B.8 InputString.java

```
import java.util.*;

/**
 * An input string for the parser.
 */
public class InputString {

  /**
   * Vector of Symbols containing the symbols of the input string.
   * The elements reflects word-classes.
   */
  private Vector symbols;

  /**
   * Vector of Strings containing the corresponding concpts.
   * The elements reflects words.
   */
  private Vector concepts;

  /**
   * true if input string is corrupted.
   */
  private boolean corrupted;

  /**
   * Default constructor.
   */
  public InputString() {
  }

  /**
   * Creates new InputString.
   *
   * @param grm   Lingvistic grammar connected to this input string.
   * @param cgrm  Conceptual grammar connected to this input string.
   * @param st    StringTokenizer containing the content of the new
   *              input string.
   */
  public InputString(Grammar grm, ConceptGrammar cgrm,
                     StringTokenizer st) {
```

```
  if (st.countTokens() == 0) {
    corrupted = true;
  } else {
    corrupted = false;
  }
  symbols = new Vector(st.countTokens());
  concepts = new Vector(st.countTokens());
  Symbol s = new Terminal();
  String c = "";
  while (st.hasMoreTokens()) {
    String token = st.nextToken();
    String sSymbol;
    String sConcept;

    if (token.indexOf('{') >= 0) {
      sSymbol = token.substring( token.indexOf('{') + 1,
                    token.indexOf('}') );
      sConcept = token.substring( 0, token.indexOf('{') );
    } else {
      sSymbol = token;
      sConcept = "";
    }

    s = grm.getSymbol(sSymbol);
    if (s != null) {
      symbols.add(s);
      concepts.add(sConcept);
    } else {
      corrupted = true;
      break;
    }
  }
}

/**
 * Checks if this input string is corrupted.
 *
 * @return  Value of variable corrupted.
 */
public boolean isCorrupted() {
  return corrupted;
}
```



```
/**
 * Gets lingvistic symbols of this input string.
 *
 * @return  Array with the symbols.
 */
public Vector getSymbols() {
  return symbols;
}

/**
 * Gets lingvistic symbol at a certain position.
 *
 * @param i  Position.
 * @return   The requested symbol.
 */
public Symbol getSymbol(int i) {
  return (Symbol) symbols.get(i);
}

/**
 * Gets conceptual string at a certain position.
 *
 * @param i  Position.
 * @return   The requested String.
 */
public String getConcept(int i) {
  return (String) concepts.get(i);
}

/**
 * Gets the length of this input string.
 *
 * @return  Length of Array symbols.
 */
public int size() {
  return symbols.size();
}

/**
 * Prints textual description of this InputString.
 *
 * @return  String containing all of the symbols in this input
 *          string.
```

```
   */
  public String printToString() {
    String result = "";
    Enumeration e2 = concepts.elements();
    for (Enumeration e = symbols.elements(); e.hasMoreElements();) {
      Symbol s = (Symbol) e.nextElement();
      result += (String) e2.nextElement();
      result += "{" + s.printToString() + "}";
      result += " ";
    }
    return result;
  }

  /**
   * Prints textual description of this InputString to System.out
   * with use of method pp in class dbg.
   */
  public void print() {
    Enumeration e2 = concepts.elements();
    for (Enumeration e = symbols.elements(); e.hasMoreElements();) {
      Symbol s = (Symbol) e.nextElement();
      s.print();
      dbg.pp("(" + (String) e2.nextElement() + ")");
      dbg.pp(" ");
    }
    dbg.pp("\n");
  }
}
```


B.9 SRTRBFBUParser.java

```
import java.util.*;

/**
 * Semantic restricted top-down restricted breadth-first bottom-up
 * parser
 */
public class SRTRBFBUParser implements Runnable{

  /**
   * Lingvistic grammar
   */
  protected Grammar grm;

  /**
   * Conceptual grammar
   */
  protected ConceptGrammar cgrm;

  /**
   * Sentence to be analyzed
   */
  protected InputString ips;

  /**
   * Number of symbols read from input string
   */
  protected int readSymbols;

  /**
   * The stacks reflecting the sentenceforms under analysis
   */
  protected Vector reductionStacks;

  /**
   * The new start rule
   */
  protected ProductionRule newStartRule;

  /**
   * Default constructor
```

```
 *
 * @return
 */
public SRTRBFBUParser() {
}

/**
 * Parses input string with respect to the given grammars
 *
 * @param grm   lingvistic grammar
 * @param cgrm  conceptal grammar
 * @param ips   input string to be analyzed
 * @return      the parsed sentenceforms in form of stacks
 */
public void setParseArguments(Grammar grm, ConceptGrammar cgrm,
                              InputString ips) {
  this.grm = grm;
  this.cgrm = cgrm;
  this.ips = ips;
}

/**
 * Parses input string with respect to the given grammars
 *
 * @param grm   lingvistic grammar
 * @param cgrm  conceptal grammar
 * @param ips   input string to be analyzed
 * @return      the parsed sentenceforms in form of stacks
 */
public void run() {

  this.readSymbols = 0;
  reductionStacks = new Vector();

  //New start rule S' -> S
  Vector newRhsV = new Vector(1);
  newRhsV.add(grm.getStartSymbol());
  newStartRule = new ProductionRule(new NonTerminal("$"),
                                    newRhsV);
  Vector newFunDefArg = new Vector(1); newFunDefArg.add(new Integer(1));
  newStartRule.addFunDef(1,
              newStartRule.new AttributeFunctionDef(
                      AttributeCalculator.NO_OPERATION,
```



```
                      newFunDefArg)
              );
  Stack firstElement = new Stack();
  Vector firstItemVector = new Vector();
  Item startItem = new Item(newStartRule, 0);
  firstItemVector.add(startItem);
  firstItemVector = startItem.getPredictorItems(firstItemVector);
  Vector hdSemParses = new Vector();
  hdSemParses.add(cgrm.new SemParsing(cgrm.getStartSymbol(),
                                      true));
  StackElement firstStackElement =
                  new StackElement(firstItemVector,
                                      hdSemParses, null);
  firstElement.push(firstStackElement);
  printItemVector("bottomelement:", firstStackElement.itemVector);
  reductionStacks.add(firstElement);

  while (readSymbols < ips.size()) {

    shift( readSymbols++ );
    reduce();

    if (GUI.stopParse) return;

    if (reductionStacks.size() == 0) {
      GUI.writeToOutput("Next symbol not recognized -
                         aborting parsing\n",
                        GUI.SHOWPARSEINFO);
      dbg.p("Next symbol not recognized - aborting parsing");
      reductionStacks = null;
      GUI.parsingEnded();
      return;
    }
  }
  GUI.writeToOutput("stacks"+reductionStacks.size(),
                    GUI.SHOWPARSEINFO);
  if (succesReduceStacks()) {
    dbg.p("Sentence parsed succesfully!");
  }
  GUI.parsingEnded();
}
```

```
public Vector getReductionStacks() {
  return reductionStacks;
}

/**
 * Discards stacks without succesfull parse
 *
 * @return  false if no stacks are succesfull, true otherwise
 */
private boolean succesReduceStacks() {
  Vector areToBeRemoved = new Vector();
  for (Enumeration e = reductionStacks.elements();
       e.hasMoreElements(); ) {

    Stack st = (Stack) e.nextElement();
    boolean startItemFound = false;
    for (Enumeration e2 =
                (((StackElement)st.peek()).itemVector).elements();
         e2.hasMoreElements(); ) {

      Item it = (Item) e2.nextElement();
      if ( it.getProductionRule() == newStartRule &&
                                     it.dotIsAtEnd()) {
        startItemFound = true;
      }
    }
    if (!startItemFound) {
      areToBeRemoved.add(st);
    }
  }
  for (Enumeration e = areToBeRemoved.elements();
       e.hasMoreElements(); ) {

    reductionStacks.removeElement((Stack) e.nextElement());
  }
  if (reductionStacks.size() > 0) {
    return true;
  } else {
    return false;
  }
}

/**
```



```
 * Prints to string the number of semantic parses in each stack.
 *
 * @return  String representing state
 */
public String printStateToString() {
  String result = "";
  for (Enumeration e = reductionStacks.elements();
       e.hasMoreElements(); ) {

    Stack st = (Stack) e.nextElement();
    result += "[";
    for (Enumeration e2 = st.elements(); e2.hasMoreElements(); ) {
      StackElement ste = (StackElement) e2.nextElement();
      result += "" + ste.getPossibleParses().size();
      if (e2.hasMoreElements()) {
        result += ":";
      }
    }
    result += "]";
    if (e.hasMoreElements()) {
      result += ", ";
    } else {
      result += "\n";
    }
  }
  return (result);
}

/**
 * shifts symbol and push new element on stacks that can accept
 * symbol and discards stacks that cannot.
 *
 * @param inputIndex  position of the next symbol to add to stacks
 */
private void shift( int inputIndex ) {
  Symbol s = ips.getSymbol(inputIndex);
  String c = ips.getConcept(inputIndex);
  s.print(); dbg.pp(" <<< shift symbol   ");
  dbg.p(c + " <<< shift concept");

  GUI.writeToOutput("\nSHIFT\""+c+"{"+s.printToString()+"}\"",
                    GUI.SHOWPARSEINFO);
```

```
Vector areToBeRemoved = new Vector();
Vector newStacks = new Vector();
for (Enumeration e = reductionStacks.elements();
     e.hasMoreElements(); ) {

  if (GUI.stopParse) return;
  Stack st = (Stack) e.nextElement();

  Vector newStackElements = stackScanner(st, s, c);
  if (newStackElements.size() > 0) {
    for (Enumeration e2 = newStackElements.elements();
         e2.hasMoreElements(); ) {

      StackElement newSE = (StackElement) e2.nextElement();
      if ( newSE == newStackElements.firstElement() ) {
        st.push( newSE );
      } else {
        Stack newStack = stackCopy(st);
        newStack.pop();
        newStack.push( newSE );
        newStacks.add( newStack );
      }
      printItemVector("Added by 'shift'", newSE.itemVector);
    }
  } else {
    areToBeRemoved.add(st);
  }
}
if (newStacks.size() > 0) {
  for (Enumeration e = newStacks.elements();
       e.hasMoreElements(); ) {

    reductionStacks.add((Stack) e.nextElement());
  }
  GUI.writeToOutput("\n"+newStacks.size() +
                    " new stacks due to new input symbol",
                    GUI.SHOWPARSEINFO);
}
if (areToBeRemoved.size() > 0) {
  for (Enumeration e = areToBeRemoved.elements();
       e.hasMoreElements(); ) {

    reductionStacks.removeElement((Stack) e.nextElement());
```



```
    }
    GUI.writeToOutput("\n"+areToBeRemoved.size() +
                      "stacks removed because of input symbol",
                      GUI.SHOWPARSEINFO);
    dbg.p(areToBeRemoved.size() + " stacks removed");
  }

  GUI.writeToOutput("\nStatus:\n" +printStateToString(),
                    GUI.SHOWSTACKSTATE);
}

/**
 * Scans stack with respect to read symbol and concept
 *
 * @param st  stack to be checked
 * @param s   read symbol
 * @param c   read concept
 * @return    new stack elements
 */
private Vector stackScanner(Stack st, Symbol s, String c) {

  Vector newStackElements = new Vector();

  StackElement topSE = (StackElement) st.peek();
  Vector topItemVector = topSE.itemVector;

  for (Enumeration e = topItemVector.elements();
       e.hasMoreElements(); ) {

    Item tmpItem = (Item) e.nextElement();
    itemCheck: if ( tmpItem.isPossibleDotShift( s ) ) {
      Vector newItemVector = new Vector();

      //wrt semantic parsing
      Vector inputValues = new Vector();
      int dotPos = tmpItem.getDotPosition();
      for (int i = dotPos; i >= 0; i--) {
        if (i < dotPos) {
          inputValues.add(
                  AttributeCalculator.copyOfInputValue(
                      ((StackElement)
                       st.elementAt( (st.size()-1) - i) ).
                       getLastSynthValue() ) );
```

```
        } else {
          inputValues.add(
                  AttributeCalculator.copyOfInputValue(
                      ((StackElement)
                       st.elementAt( (st.size()-1) - i) ).
                       getPossibleParses() ) );
        }
      }
      Vector lastSynthValue = new Vector(1);
      lastSynthValue.add( new String(c) );
      inputValues.add( lastSynthValue );
      Vector newPossibleParses =
              AttributeCalculator.
                  calcAttributeValue(cgrm,
                                     tmpItem.getProductionRule(),
                                     dotPos + 1,
                                     inputValues);

      if (newPossibleParses.size() == 0 && GUI.semanticParse) {
        GUI.writeToOutput("\n"+"Stackelement discarded in shift
                           operation because of the semantics",
                          GUI.SHOWPARSEINFO);
        break itemCheck;
      }

      //wrt syntactic parsing
      Item newItem = tmpItem.getCopy();
      newItem.shiftDotPosition();
      newItemVector.add( newItem );
      newItemVector = newItem.getPredictorItems( newItemVector );

      StackElement newSE = new StackElement(newItemVector,
                                            newPossibleParses,
                                            lastSynthValue,
                                            new SubParseTree(
                                                  (Terminal) s ));
      newStackElements.add( newSE );
    }
  }
  return newStackElements;
}

/**
```



```
 * Reduces stacks in reductionStacks if possible
 */
private void reduce() {
  if (reductionStacks.size() > 0) {

    GUI.writeToOutput("\n\n*** REDUCE ***"+"", GUI.SHOWPARSEINFO);

    for (Enumeration e = reductionStacks.elements();
         e.hasMoreElements(); ) {

      if (GUI.stopParse) return;
      Stack st = (Stack) e.nextElement();
      Vector newStacks = completeReduceStep(st);
      if (newStacks.size() > 0) {
        for (Enumeration e2 = newStacks.elements();
             e2.hasMoreElements(); ) {
          Stack newStack = (Stack) e2.nextElement();
          reductionStacks.add(newStack);
          printItemVector("Added by 'reduce'",
                          ((StackElement)
                                  newStack.peek()).itemVector );
        }
        GUI.writeToOutput("\n"+newStacks.size() + " new stacks due
                           to reduction", GUI.SHOWPARSEINFO);
      }
    }
  }
  GUI.writeToOutput("\nStatus:\n" + printStateToString(),
                    GUI.SHOWSTACKSTATE);
}

/**
 * Reduces one stack one step and returns one or more new reduced
 * stacks. (Vector of stacks)
 *
 * @param st  Stack to be checked for possible reductions
 * @return    Vector with new stacks
 */
private Vector completeReduceStep(Stack st) {

  Vector newStacks = new Vector();
  StackElement topStackElement = (StackElement) st.peek();
  for (Enumeration e = topStackElement.itemVector.elements();
```

```
   e.hasMoreElements(); ) {

Item tmpItem = (Item) e.nextElement();
if (tmpItem.dotIsAtEnd()) {
  Symbol activeSymbol = tmpItem.getProductionRule().getLhs();
  int rhsSize = tmpItem.getProductionRule().rhsSize();

  Stack newStack = stackCopy(st);
  SubParseTree spt[] = new SubParseTree[rhsSize];
  for (int i = 0; i < rhsSize; i++) {
    StackElement se = (StackElement) newStack.pop();
    if (se.parseTree == null)
      dbg.p("null");
    spt[rhsSize-1-i] = se.parseTree;
  }
  Vector items1 = ((StackElement) newStack.peek()).itemVector;

  boolean firstTime = true;
  for (Enumeration e2 = items1.elements();
       e2.hasMoreElements(); ) {

    Stack newStackT = stackCopy(newStack);

    Item tmpItem2 = (Item) e2.nextElement();
    itemCheck: if (tmpItem2.isPossibleDotShift(activeSymbol)){
      Vector newItems = new Vector();
      Item newItem = tmpItem2.getCopy();
      newItem.shiftDotPosition();
      newItems.add(newItem);
      newItems = newItem.getPredictorItems( newItems );

      //wrt semantic parsing
      int dotPos = tmpItem2.getDotPosition();
      Vector inputValues = new Vector();
      for (int i = dotPos; i >= 0; i--) {
        if (i < dotPos) {
          inputValues.add(
                  AttributeCalculator.copyOfInputValue(
                      ((StackElement)
                       newStackT.elementAt(
                           (newStack.size()-1) - i) ).
                             getLastSynthValue() ) );
        } else {
```



```
          inputValues.add(
                  AttributeCalculator.copyOfInputValue(
                      ((StackElement)
                       newStackT.elementAt(
                           (newStack.size()-1) - i) ).
                             getPossibleParses() ) );
        }
      }
      Vector lastSynthValue = AttributeCalculator.
                              copyOfInputValue(
                                  topStackElement.
                                  getPossibleParses() );

      inputValues.add( lastSynthValue );

      Vector newPossibleParses =
                AttributeCalculator.calcAttributeValue(
                                  cgrm,
                                  tmpItem2.getProductionRule(),
                                  dotPos + 1,
                                  inputValues);

      if (newPossibleParses.size() == 0 && GUI.semanticParse){
        GUI.writeToOutput("Stackelement discarded in reduce
                           operation because of the
                           semantics\n",
                          GUI.SHOWPARSEINFO);
        break itemCheck;
      }

      SubParseTree newSpt =
             new SubParseTree(tmpItem.getProductionRule(),
                              spt, null);

      StackElement newSE = new StackElement(newItems,
                                            newPossibleParses,
                                            lastSynthValue,
                                            newSpt );
      newStackT.push( newSE );
      newStacks.add( newStackT );
    }
  }
}
```

```
  }
  return newStacks;
}

/**
 * Copies stack
 *
 * @param st  Stack to be copied
 * @return  Copy of st
 */
private Stack stackCopy(Stack st) {
  Stack newStack  = new Stack();
  StackElement elems[] = new StackElement[st.size()];
  st.toArray(elems);
  for (int i = 0; i < elems.length; i++) {
    Vector newItems = new Vector();
    for (Enumeration e = elems[i].itemVector.elements();
         e.hasMoreElements(); ) {

      Item tmpItem = (Item) e.nextElement();
      newItems.add(tmpItem.getCopy());
    }
    Vector newPossibleParses =
            AttributeCalculator.copyOfInputValue(
                                  elems[i].getPossibleParses() );
    Vector newLastSynthValue =
            AttributeCalculator.copyOfInputValue(
                                  elems[i].getLastSynthValue() );
    StackElement newSE =
            new StackElement(newItems,
                             newPossibleParses,
                             newLastSynthValue,
                             elems[i].getParseTreeCopy());
    newStack.push(newSE);
  }
  return newStack;
}

/**
 * Prints item Vector
 *
 * @param s   tab string
 * @param iv  items
```



```
 */
public void printItemVector(String s, Vector iv) {
  dbg.pp("\n" + s + "\n");
  for (Enumeration e = iv.elements(); e.hasMoreElements(); ) {
    ((Item) e.nextElement()).print();
  }
  dbg.pp("\n");
}

/**
 * Class that represents the Stack elements
 * Contains an item Vector and a so far parsetree
 */
public class StackElement {

  /**
   * Items representing this stack element
   */
  private Vector itemVector;

  /**
   * Attributevalue. Vector of conceptGrammar.semParse
   */
  private Vector lastSynthValue;

  /**
   * Attributevalue. Vector of conceptGrammar.semParse
   */
  private Vector possibleParses;

  /**
   * The so far build lingvistic parse tree
   */
  private SubParseTree parseTree;

  /**
   * Constructor creating new stack element
   *
   * @param itemVector      items representing stack element
   * @param possibleParses  attribute value possible parses
   * @param parseTree       so far build lingvistic parsetree
   */
  public StackElement(Vector itemVector, Vector possibleParses,
```

```
                    SubParseTree parseTree) {

  this.itemVector = itemVector;
  this.lastSynthValue = possibleParses;
  this.possibleParses = possibleParses;
  this.parseTree = parseTree;
}

/**
 * Constructor creating new stack element
 *
 * @param itemVector      items representing stack element
 * @param possibleParses  attribute value possible parses
 * @param lastSynthValue  attribute value possible parses
 * @param parseTree       so far build lingvistic parsetree
 */
public StackElement(Vector itemVector, Vector possibleParses,
                    Vector lastSynthValue,
                    SubParseTree parseTree) {

  this.itemVector = itemVector;
  this.lastSynthValue = lastSynthValue;
  this.possibleParses = possibleParses;
  this.parseTree = parseTree;
}

/**
 * gets attribute value possibleParses
 *
 * @return  requested value
 */
public Vector getPossibleParses() {
  return possibleParses;
}

/**
 * gets attribute value lastSynthValue
 *
 * @return  requested value
 */
public Vector getLastSynthValue() {
  return lastSynthValue;
}
```



```
/**
 * gets copy of the so far build parse tree
 *
 * @return  the requested copy
 */
public SubParseTree getParseTreeCopy() {
  if (parseTree != null)
    return parseTree.getCopy();
  else
    return null;
}

/**
 * Prints textual description of parse tree to String
 *
 * @return  String with tree
 */
public String printParseTreeToString() {
  if (parseTree != null)
    return (parseTree.printToString(0));
  else
    return "";
}

/**
 * Prints textual description of parse tree.
 */
public void printParseTree() {
  if (parseTree != null)
  parseTree.print(0);
}

/**
 * Prints textual description of this stack element with use
 * of class dbg.
 */
public void print() {
  printItemVector("-- Stackelement --" , itemVector);
  dbg.p("lastSynthValue: " + lastSynthValue);
  dbg.p("possibleParses: " + possibleParses);
  dbg.p("  -- --");
}
```

```
}

/**
 * Class represents an item.
 * Consists of a production rule and a dot position
 */
public class Item {

  /**
   * Production rule used in item
   */
  private ProductionRule rule;

  /**
   * Position of dot in item
   */
  private int dotPosition;  //points at symbol AFTER dot

  /**
   * Conctructor
   *
   * @param rule     rule in new item.
   * @param dotPosition  position of dot in new item
   */
  public Item(ProductionRule rule, int dotPosition) {
    this.rule = rule;
    this.dotPosition = dotPosition;
  }

  /**
   * Checks if this item can accept a symbol as following.
   *
   * @param nextSymbol  next symbol
   * @return            true if symbol is after dot in item,
   *                    false otherwise.
   */
  public boolean isPossibleDotShift(Symbol nextSymbol) {
    if (dotIsAtEnd()) return false;
    return (rule.getRhsSymbol(dotPosition) == nextSymbol);
  }

  /**
   * Checks if dot is at end i production rule
```



```
   *
   * @return  true if dot is at end in item, false otherwise.
   */
  public boolean dotIsAtEnd() {
    return (dotPosition == rule.rhsSize());
  }

  /**
   * gets predicted items for this item. Used recursively
   *
   * @param result  the so far found result
   * @return        the given result added new predictable
   *                items.
   */
  public Vector getPredictorItems(Vector result) {
    Symbol activeSymbol = rule.getRhsSymbol(dotPosition);
    for (Enumeration e =
                ((Vector)grm.getProductionRules()).elements();
         e.hasMoreElements(); ) {

      ProductionRule tmpPr = (ProductionRule) e.nextElement();

      if (tmpPr.getLhs() == activeSymbol) {
        Item tmpI = new Item(tmpPr, 0);

        if (!tmpI.containedIn(result)) {
          result.add(tmpI);

          for (Enumeration e2 =
                  tmpI.getPredictorItems(result).elements();
               e2.hasMoreElements(); ) {

            Item tmpI2 = (Item) e2.nextElement();
            if (!tmpI2.containedIn(result)) {
              result.add(tmpI2);
            }
          }
        }
      }
    }
    return result;
  }
```

```
/**
 * Shifts dot position one step to the right.
 */
public void shiftDotPosition() {
  dotPosition++;
}

/**
 * gets production rule of this item
 *
 * @return  requested production rule
 */
public ProductionRule getProductionRule() {
  return rule;
}

/**
 * gets copy of this item
 *
 * @return  requested copy
 */
public Item getCopy() {
  return new Item(rule, dotPosition);
}

/**
 *
 *
 * @param pos
 * @return
 */
public Symbol getSymbolAt(int pos) {
  return rule.getRhsSymbol(pos);
}

/**
 * gets position of dot in this item
 *
 * @return  the dot position
 */
public int getDotPosition() {
  return dotPosition;
}
```



```
    /**
     * checks if this item is contained in given Vector
     *
     * @param v  Vector that may contain this item
     * @return   true if this item is contained in Vector, false
     *           otherwise.
     */
    public boolean containedIn(Vector v) {
      for (Enumeration e = v.elements(); e.hasMoreElements(); ) {
        Item tmpI = (Item) e.nextElement();
        if (rule.equals(tmpI.getProductionRule()) &&
            (dotPosition == tmpI.getDotPosition())) return true;
      }
      return false;
    }

    /**
     * Prints this item with use of class dbg.
     */
    public void print() {
      rule.getLhs().print();
      dbg.pp(" -> ");
      for (int i = 0; i < rule.rhsSize(); i++) {
        if (i == dotPosition) {
          dbg.pp(".");
        } else if (i > 0 && i < rule.rhsSize()) {
          dbg.pp(" ");
        }
        rule.getRhsSymbol(i).print();
      }
      if (dotPosition == rule.rhsSize()) dbg.pp(".");
      dbg.pp("\n");
    }
  }
}
```

B.10 AttributeCalculator.java

```
import java.util.*;

/**
 * Class contains methods for calculating attribute values
 */
public final class AttributeCalculator {

  /**
   * possible basis functions
   */
  public final static int NO_OPERATION = 0;
  public final static int OPERATION_HEADDERIVE1 = 1;
  public final static int OPERATION_HEADDERIVE2 = 2;
  public final static int OPERATION_MATCH = 3;
  public final static int OPERATION_FEATUREACCEPT1 = 4;
  public final static int OPERATION_FEATUREACCEPT2 = 5;
  public final static int OPERATION_UNITE = 6;
  public final static int BLANK_ARGUMENT = -9;

  /**
   * Calculates attribute functions. Method is invoked during
   * textanalysis
   *
   * @param cgrm         conceptual grammar to use
   * @param pRule        lingvistic production rule containing
   *                     attribute functions
   * @param funNr        position of symbol/attributefunktion in
   *                     rule
   * @param inputValues  attribute values to use as arguments in
   *                     calculation
   * @return             Vector containing the result (instances of
   *                     SemParsing)
   */
  public static Vector calcAttributeValue(ConceptGrammar cgrm,
                                          ProductionRule pRule,
                                          int funNr,
                                          Vector inputValues) {

    return calcFun(cgrm, pRule.getFunDef(funNr), inputValues);
  }
```



```
  /**
   * Calculates result of basis function used in defining
   * attributefunctions
   *
   * @param cgrm         conceptual grammar to use
   * @param funDef       attribute function definition
   * @param inputValues  attribute values to use as arguments in
   *                     calculation
   * @return             Vector containing the result (instances of
   *                     SemParsing)
   */
  private static Vector calcFun(
                  ConceptGrammar cgrm,
                  ProductionRule.AttributeFunctionDef funDef,
                  Vector inputValues) {

    if (dbg.getDebugMode()) {
      for (Enumeration e = inputValues.elements();
           e.hasMoreElements(); ) {

        Object o = e.nextElement();
        dbg.p(funDef+"CALCATTRIBUTEFUN_INPUTVALUE: " + o);

        try {
          Vector varg1 = (Vector) o;
          for (Enumeration eee = varg1.elements();
               eee.hasMoreElements(); ) {

            ((ConceptGrammar.SemParsing) eee.nextElement()).print();
          }

        } catch(ClassCastException cce){
          dbg.p("not vector");
        }
      }
    }

    Vector result = new Vector();

    if (funDef == null) {
      return copyOfPossibleParses( (Vector) inputValues.get(0) );
    }
```

```
if (funDef.getOperation() == NO_OPERATION) {

  int valuePos =
      ((Integer)funDef.getArguments().elementAt(0)).intValue();

  result =
      copyOfPossibleParses((Vector)inputValues.get(valuePos));

} else if (funDef.getOperation() == OPERATION_MATCH) {

  Vector varg1;
  try {
    int arg1 =
        ((Integer)funDef.getArguments().elementAt(0)).intValue();
    varg1 = (Vector) inputValues.get(arg1);
  } catch(ClassCastException cce){
    varg1 =
        calcFun(cgrm,
                (ProductionRule.AttributeFunctionDef)
                    funDef.getArguments().elementAt(0),
                inputValues);
  }

  int arg2 =
      ((Integer)funDef.getArguments().elementAt(1)).intValue();

  result =
      match(cgrm,
            copyOfPossibleParses(varg1),
            (String)( (Vector) inputValues.get(arg2)).get(0));

} else if (funDef.getOperation() == OPERATION_FEATUREACCEPT1) {

  Vector varg1;
  try {
    int arg1 =
        ((Integer)funDef.getArguments().elementAt(0)).intValue();
    varg1 =
        copyOfPossibleParses((Vector) inputValues.get(arg1));
  } catch(ClassCastException cce){
    varg1 =
        calcFun(cgrm,
```



```
                (ProductionRule.AttributeFunctionDef)
                    funDef.getArguments().elementAt(0),
                inputValues);
  }

  int arg2 =
      ((Integer)funDef.getArguments().elementAt(1)).intValue();
  int arg3 =
      ((Integer)funDef.getArguments().elementAt(2)).intValue();

  String sarg2 =
      (arg2 < 0)?
          "":
          (String)( (Vector) inputValues.get(arg2)).get(0);

  result =
      featureAccept1(cgrm,
                     varg1,
                     sarg2,
                     (String)((Vector)inputValues.get(arg3)).
                             get(0));

} else if (funDef.getOperation() == OPERATION_FEATUREACCEPT2) {

  Vector varg1;
  try {
    int arg1 =
        ((Integer)funDef.getArguments().elementAt(0)).intValue();
    varg1 =
        copyOfPossibleParses((Vector) inputValues.get(arg1));
  } catch(ClassCastException cce){
    varg1 =
        calcFun(cgrm,
                (ProductionRule.AttributeFunctionDef)
                    funDef.getArguments().elementAt(0),
                inputValues);
  }

  int arg2 =
      ((Integer)funDef.getArguments().elementAt(1)).intValue();
  String sarg2 =
      (arg2 < 0)?
          "":
```

```
          (String) ((Vector) inputValues.get(arg2)).get(0);

    Vector varg3;
    try {
      int arg3 =
          ((Integer)funDef.getArguments().elementAt(2)).intValue();
      varg3 =
          copyOfPossibleParses((Vector) inputValues.get(arg3));
    } catch(ClassCastException cce){
      varg3 =
          calcFun(cgrm,
                  (ProductionRule.AttributeFunctionDef)
                      funDef.getArguments().elementAt(2),
                  inputValues);
    }

    result = featureAccept2(cgrm, varg1, sarg2, varg3);

  } else if (funDef.getOperation() == OPERATION_HEADDERIVE1) {

    int arg1 =
        ((Integer)funDef.getArguments().elementAt(0)).intValue();

    result =
        headDerive1(cgrm,
                    (String)((Vector)inputValues.get(arg1)).
                            get(0));

  } else if (funDef.getOperation() == OPERATION_HEADDERIVE2) {

    Vector varg1;
    try {
      int arg1 =
          ((Integer)funDef.getArguments().elementAt(0)).intValue();
      varg1 =
          copyOfPossibleParses((Vector) inputValues.get(arg1));
    } catch(ClassCastException cce){
      varg1 =
          calcFun(cgrm,
                  (ProductionRule.AttributeFunctionDef)
                      funDef.getArguments().elementAt(0),
                  inputValues);
    }
```



```
    int arg2 =
        ((Integer)funDef.getArguments().elementAt(1)).intValue();

    result =
        headDerive2(cgrm,
                    varg1,
                    (String)((Vector) inputValues.get(arg2))
                            .get(0));

  } else if (funDef.getOperation() == OPERATION_UNITE) {

    Vector varg1;
    try {
      int arg1 =
          ((Integer)funDef.getArguments().elementAt(0)).intValue();
      varg1 =
          copyOfPossibleParses((Vector) inputValues.get(arg1));
    } catch(ClassCastException cce){
      varg1 =
          calcFun(cgrm,
                  (ProductionRule.AttributeFunctionDef)
                      funDef.getArguments().elementAt(0),
                  inputValues);
    }

    Vector varg2;
    try {
      int arg2 =
          ((Integer)funDef.getArguments().elementAt(1)).intValue();
      varg2 =
          copyOfPossibleParses((Vector) inputValues.get(arg2));
    } catch(ClassCastException cce){
      varg2 =
          calcFun(cgrm,
          (ProductionRule.AttributeFunctionDef)
              funDef.getArguments().elementAt(1),
          inputValues);
    }

    result.addAll(varg1);
    result.addAll(varg2);
```

```
  }

  if (dbg.getDebugMode()) {
    dbg.p("Result");
    for (Enumeration e = result.elements(); e.hasMoreElements();){
      dbg.p("result from Attributecalculation:");
      ((ConceptGrammar.SemParsing) e.nextElement()).print();
    }
    dbg.p("End result");
  }

  return result;
}

/**
 * Calculates result of basis function HeadDerive1
 *
 * @param cgrm     conceptual grammar to use
 * @param concept  concept to derive
 * @return         Vector with special HeadDerive instance of
 *                 class SemParsing
 */
private static Vector headDerive1(ConceptGrammar cgrm,
                                  String concept) {
  Vector hdSemParses = new Vector();
  hdSemParses.add(
      cgrm.new SemParsing((NonTerminal)cgrm.getSymbol( concept ),
      true));

  return hdSemParses;
}

/**
 * Calculates result of basis function HeadDerive2
 *
 * @param cgrm       conceptual grammar to use
 * @param semParses  semantic parses to use
 * @param role       string containing role restriction
 * @return           Vector with special HeadDerive instances of
 *                   class SemParsing
 */
private static Vector headDerive2(ConceptGrammar cgrm,
                                  Vector semParses,
```



```
                                  String role) {
  Vector result = new Vector();

  for (Enumeration e = semParses.elements(); e.hasMoreElements();){
    ConceptGrammar.SemParsing parse =
                  (ConceptGrammar.SemParsing) e.nextElement();

    for (Enumeration e2 =
            ((Vector) parse.getPossibleFeatures()).elements();
         e2.hasMoreElements(); ) {

      ConceptGrammar.Feature fea =
                    (ConceptGrammar.Feature) e2.nextElement();
      if ( cgrm.findRuleWithLhs(fea.getRole()).
               containsRhs( cgrm.getSymbol( role ) ) ) {

        Vector hdSemParses = new Vector();
        hdSemParses.add(
            cgrm.new SemParsing((NonTerminal)fea.getConcept(),
            true));

        result.addAll( hdSemParses );
      }
    }
  }

  return result;
}

/**
 * Executes calculation of HeadDerive without restrictions.
 * Calls itself recursively. Start call with same concept
 * and SubConcept and empty vector.
 *
 * @param concept            top concept of derivation
 * @param subConcept         concept to derive
 * @param inheritedFeatures  inherited possible semantic relations
 * @return                   Vector containing the result
 *                           (instances of SemParsing)
 */
private static Vector calcHeadDerive(ConceptGrammar cgrm,
                                     NonTerminal concept,
                                     NonTerminal subConcept,
```

```
                                    Vector inheritedFeatures) {
  Vector result = new Vector(0);

  ConceptGrammar.ConceptProductionRule p =
                               cgrm.findRuleWithLhs(subConcept);
  for (Enumeration e = p.getAlternatives().elements();
       e.hasMoreElements(); ) {

    ConceptGrammar.Alternative alt =
                  (ConceptGrammar.Alternative) e.nextElement();
    Vector newInheritedFeatureVector =
                  copyFeatureVector(inheritedFeatures);
    newInheritedFeatureVector.addAll( alt.getFeatures() );
    if (alt.getConcept().isTerminalSymbol()) {
      ConceptGrammar.SemParsing newSP =
                cgrm.new SemParsing(concept,
                                    subConcept,
                                    newInheritedFeatureVector,
                                    new Vector());
      result.add(newSP);
    } else {
      Vector subRes =
          calcHeadDerive(cgrm,
                         concept,
                         (NonTerminal) alt.getConcept(),
                         newInheritedFeatureVector);
      result.addAll(subRes);
    }
  }
  dbg.p("headDeriveResultSize " + result.size());
  return result;
}

/**
 * Executes calculation of HeadDerive with restriction concept.
 * Calls itself recursively. Start call with same concept and
 * SubConcept and empty vector.
 *
 * @param concept             top concept of derivation
 * @param subConcept          concept to derive
 * @param inheritedFeatures   inherited possible semantic relations
 * @param restrictionConcept  concept to restrict derivation
 * @return                    Vector containing the result
```



```
 *                            (instances of SemParsing)
 */
private static Vector calcRestrictedHeadDerive(
                                    ConceptGrammar cgrm,
                                    NonTerminal concept,
                                    NonTerminal subConcept,
                                    Vector inheritedFeatures,
                                    Terminal restrictionConcept) {
  restrictionConcept.print();
  Vector result = new Vector(0);

  ConceptGrammar.ConceptProductionRule p =
                               cgrm.findRuleWithLhs(subConcept);
  for (Enumeration e = p.getAlternatives().elements();
       e.hasMoreElements(); ) {

    ConceptGrammar.Alternative alt =
                  (ConceptGrammar.Alternative) e.nextElement();
    Vector newInheritedFeatureVector =
                  copyFeatureVector(inheritedFeatures);
    newInheritedFeatureVector.addAll( alt.getFeatures() );
    if (alt.getConcept().isTerminalSymbol() &&
        alt.getConcept() == restrictionConcept) {

      ConceptGrammar.SemParsing newSP =
              cgrm.new SemParsing(concept,
                                  subConcept,
                                  newInheritedFeatureVector,
                                  new Vector());
      result.add(newSP);
    } else if (!alt.getConcept().isTerminalSymbol()) {
      Vector subRes =
          calcRestrictedHeadDerive(cgrm,
                                   concept,
                                   (NonTerminal) alt.getConcept(),
                                   newInheritedFeatureVector,
                                   restrictionConcept);
      result.addAll(subRes);
    }
  }
  return result;
}
```

```
/**
 * Calculates result of basis function Match
 *
 * @param cgrm       conceptual grammar to use
 * @param semParses  semantic parses to use
 * @param concept    concept to use
 * @return           result as Vector with instances of class
 *                   SemParsing
 */
private static Vector match(ConceptGrammar cgrm,
                            Vector semParses,
                            String concept) {
  concept = concept.toLowerCase();
  Vector result = new Vector();
  for (Enumeration e = semParses.elements(); e.hasMoreElements();){
    ConceptGrammar.SemParsing parse =
                  (ConceptGrammar.SemParsing) e.nextElement();

    if (parse.isHeadDerive()) {
      Vector newParses =
          calcRestrictedHeadDerive(cgrm,
                                   parse.getConcept(),
                                   parse.getConcept(),
                                   new Vector(),
                                   (Terminal)cgrm.getSymbol(concept));
      semParses.addAll(newParses);
    } else {
      if (cgrm.findRuleWithLhs((NonTerminal) parse.getTConcept()).
          containsRhs( cgrm.getSymbol( concept.toLowerCase() ))){
        result.add(parse.getCopy());
      }
    }
  }
  return result;
}

/**
 * Calculates result of basis function FeatureAccept1
 *
 * @param cgrm       conceptual grammar to use
 * @param semParses  semantic parses to use
 * @param role       role to use
```



```
 * @param fConcept   concept string to use
 * @return           result as Vector with instances of class
 *                   SemParsing
 */
private static Vector featureAccept1(ConceptGrammar cgrm,
                                     Vector semParses,
                                     String role,
                                     String fConcept) {
  Vector result = new Vector();

  if (role != null && fConcept != null) {

    Terminal tfConcept = (Terminal) cgrm.getSymbol( fConcept );
    NonTerminal ntfConcept = cgrm.getLhsTo( tfConcept );

    for (Enumeration e = semParses.elements();
         e.hasMoreElements() ; ) {

      ConceptGrammar.SemParsing parse =
                    (ConceptGrammar.SemParsing) e.nextElement();
      if (parse.isHeadDerive()) {
        Vector newParses =
            calcHeadDerive(cgrm,
                           parse.getConcept(),
                           parse.getConcept(),
                           new Vector());
        semParses.addAll(newParses);
      } else {
        ConceptGrammar.SemParsing newParse = parse.getCopy();
        featurecheck:
        for (Enumeration e2 =
                 ((Vector) newParse.getPossibleFeatures()).
                     elements();
             e2.hasMoreElements(); ) {

          ConceptGrammar.Feature fea =
                     (ConceptGrammar.Feature) e2.nextElement();

          if (role == "" ||
              cgrm.findRuleWithLhs(fea.getRole()).
                   containsRhs( cgrm.getSymbol( role )) ) {

            if (ntfConcept == fea.getConcept() ||
```

```
                cgrm.isSubConceptOf(
                              ntfConcept,
                              (NonTerminal) fea.getConcept())) {

              ConceptGrammar.SemParsing spf =
                         cgrm.new SemParsing(
                             (NonTerminal) fea.getConcept(),
                             ntfConcept, new Vector(0),
                             new Vector(0));

              newParse.getPossibleFeatures().remove( fea );
              newParse.acceptFeature(
                             cgrm.new ParsedFeature(fea, spf) );
              result.add( newParse );
              break featurecheck;
            }
          }
        }
      }
    }
  }
  return result;
}

/**
 * Calculates result of basis function FeatureAccept2
 *
 * @param cgrm       conceptual grammar to use
 * @param semParses  semantic parses to use
 * @param role       role to use
 * @param fConcepts  concepts to use
 * @return           result as Vector with instances of class
 *                   SemParsing
 */
private static Vector featureAccept2(ConceptGrammar cgrm,
                                     Vector semParses,
                                     String role,
                                     Vector fConcepts) {
  Vector result = new Vector();

  Vector reducedConcepts = findReducedConcepts( fConcepts );
  for (Enumeration e = semParses.elements();e.hasMoreElements();){
    ConceptGrammar.SemParsing parse =
```



```
                  (ConceptGrammar.SemParsing) e.nextElement();
    if (parse.isHeadDerive()) {
      Vector newParses = calcHeadDerive(cgrm,
                                        parse.getConcept(),
                                        parse.getConcept(),
                                        new Vector());
      semParses.addAll(newParses);
    } else {

      featurecheck:
      for (Enumeration e2 = (
                 (Vector) parse.getPossibleFeatures()).elements();
           e2.hasMoreElements(); ) {

        ConceptGrammar.Feature fea =
                      (ConceptGrammar.Feature) e2.nextElement();
        if (role == "" ||
            cgrm.findRuleWithLhs(fea.getRole()).
                 containsRhs( cgrm.getSymbol( role ) ) ) {

          if ( reducedConcepts.contains( fea.getConcept() )) {
            ConceptGrammar.SemParsing newParse = parse.getCopy();
            for (Enumeration e3 = fConcepts.elements();
                 e3.hasMoreElements(); ) {

              ConceptGrammar.SemParsing spf =
                  (ConceptGrammar.SemParsing) e3.nextElement();
              if (spf.isHeadDerive()) {
                Vector newParses =
                        calcHeadDerive(cgrm,
                                       parse.getConcept(),
                                       parse.getConcept(),
                                       new Vector());
                fConcepts.addAll(newParses);
              } else {
                if (fea.getConcept() == spf.getConcept()) {
                  newParse.getPossibleFeatures().remove( fea );
                  newParse.acceptFeature(
                              cgrm.new ParsedFeature(fea, spf) );

                  result.add( newParse );
                  break featurecheck;
                }
```

```
            }
          }
        }
      }
    }
  }
  return result;
}

/**
 * finds reduced concepts in Vector if descriptors
 *
 * @param possibleParses  Vector with descriptors
 * @return                Vector with found concepts
 */
private static Vector findReducedConcepts(Vector possibleParses) {
  Vector result = new Vector();
  for (Enumeration e = possibleParses.elements();
       e.hasMoreElements(); ) {

    ConceptGrammar.SemParsing sp =
                    (ConceptGrammar.SemParsing) e.nextElement();
    NonTerminal c = sp.getConcept();
    if ( !result.contains(c) )
      result.add(c);
  }
  return result;
}

/**
 * copies Vector of descriptors (instances of class SemParsing)
 *
 * @param possibleParses  Vector to copy
 * @return                copy of given Vector
 */
public static Vector copyOfPossibleParses(Vector possibleParses) {
  Vector result = new Vector(possibleParses.size());
  for (Enumeration e = possibleParses.elements();
       e.hasMoreElements(); ) {

    ConceptGrammar.SemParsing sp =
                    (ConceptGrammar.SemParsing) e.nextElement();
```



```
    result.add( sp.getCopy() );
  }
  return result;
}

/**
 * copies Vector of possible semantic relations (instances of
 * class Feature)
 *
 * @param features  Vector to copy
 * @return          copy of given Vector
 */
private static Vector copyFeatureVector(Vector features) {
  Vector newFeatures = new Vector(features.size());
  for (Enumeration e = features.elements();
       e.hasMoreElements(); ) {

    newFeatures.add(
        ((ConceptGrammar.Feature) e.nextElement()).getCopy() );
  }
  return newFeatures;
}

/**
 * copies Vector of attribute values
 *
 * @param inputValue  Vector to copy
 * @return            copy of given vector
 */
public static Vector copyOfInputValue(Vector inputValue) {
  Vector result = new Vector(inputValue.size());
  for (Enumeration e = inputValue.elements();
       e.hasMoreElements(); ) {

    Object obj = e.nextElement();
    try {
      ConceptGrammar.SemParsing sp =
                              (ConceptGrammar.SemParsing) obj;
      result.add( sp.getCopy() );
    } catch(ClassCastException cce) {
      String s = (String) obj;
      result.add( s );
    }
```

```
    }
    return result;
  }
}
```


B.11 FileHandler.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

/**
 * Static class offering methods for handling file in- and output
 * and reading textual representations of grammars into objects.
 */
public final class FileHandler {

  /**
   * Reads a textfile into a string
   *
   * @param file  File to read.
   * @return    String containing content of the given file.
   */
  public static String readFileToString(File file) {
      String line, res;
      res = "";
    try {
      FileReader r = new FileReader(file);
      BufferedReader br = new BufferedReader(r);
      line = br.readLine();
      while (line != null) {
        res += line + System.getProperty("line.separator");
        line = br.readLine();
      }
      br.close();
    } catch(IOException e) {dbg.p("file read error: " + e);}
      return res;
  }

  /**
   * Writes a string into a textfile
   *
   * @param file  File to write to.
   * @param str   String to be written to file.
   * @return    true if it succeds. false otherwise
```

```
 */
public static boolean writeStringToFile(File file, String str) {
    int i = 0;
    char[] chars = str.toCharArray();
  try {
    FileWriter f = new FileWriter(file);

    while (i < chars.length) {
      if (chars[i] == '\n') {
        f.write('\r');
        f.write(chars[i]);
      } else if (chars[i] == '\r') {

      } else {
        f.write(chars[i]);
      }
      i++;
    }
    f.flush();
    f.close();
  } catch(IOException e)
        {dbg.p("file write error: " + e);return false;}
    return true;
}

/**
 * Converts textual representation of lingvistic grammar to
 * instance of class Grammar.
 *
 * @param r  String to be converted
 * @return   new instance of class Grammar reflecting the read
 *           String
 */
public static Grammar readGrammarString(String r) {

  BufferedReader br = new BufferedReader(new StringReader(r));

  Hashtable nonTerminals = new Hashtable(10);
  Hashtable terminals = new Hashtable(10);
  NonTerminal startSymbol = null;
  Vector productionRules = null;

try{
```



```
String line = br.readLine();
StringTokenizer st;

    boolean startSymbolFound = false;
    while (line != null) {
      if (line.length() > 1) {
        if (!line.substring(0,1).equals("\\")) {
          String str = line.substring(0, line.indexOf(" "));
          NonTerminal NewNT = new NonTerminal(str);
          if (!nonTerminals.containsKey(str))
            nonTerminals.put(str ,NewNT);
          if (!startSymbolFound) {
            startSymbol = NewNT;
            startSymbolFound = true;
          }
        }
      }
      line = br.readLine();
    }

br = new BufferedReader(new StringReader(r));
    line = br.readLine();
    productionRules = new Vector();
    NonTerminal tmpLhs;
    Vector tmpRhs;
    String tmps;
    while (line != null) {
      if (line.length() > 1) {
        if (!line.substring(0,1).equals("\\")) {
          st = new StringTokenizer(
                line.substring(0,line.indexOf(";")), " ");
          tmps = st.nextToken();
          tmpLhs = (NonTerminal) nonTerminals.get(tmps);
          tmps = st.nextToken();
          while (st.hasMoreTokens()) {
            tmps = st.nextToken();
            tmpRhs = new Vector(5);
            Terminal tmpT;

            rhside: while (!tmps.equals("|")) {
              // a terminal symbol is read
              if (!nonTerminals.containsKey(tmps)) {
                if (terminals.containsKey(tmps)) {
```

```
          tmpT =(Terminal)terminals.get(tmps);
        } else {
          tmpT = new Terminal(tmps);
          terminals.put(tmps, tmpT);
        }
        tmpRhs.addElement(tmpT);
      // a nonterminal symbol is read
      } else {
        tmpRhs.addElement(
                nonTerminals.get(tmps));
      }
      if (!st.hasMoreTokens()) {
        break rhside;
      }
      tmps = st.nextToken();
    }
    ProductionRule pr = new ProductionRule(tmpLhs,
                                           tmpRhs);

    StringTokenizer st2 =
        new StringTokenizer(
          line.substring(
              line.indexOf(";"),
              line.length()),
          ";");
    while (st2.hasMoreTokens()) {
      String csr = st2.nextToken();
      int pos = csr.indexOf("=");
      String leftside = csr.substring(0, pos);
      int funPos = convertToPos(leftside);
      if (funPos == 0) funPos = pr.rhsSize();
      else funPos--;

      csr = csr.substring(pos+1);
      ProductionRule.AttributeFunctionDef funDef =
                                resolveFunDef(pr, csr);

      pr.addFunDef(funPos, funDef);
    }

    productionRules.add(pr);
  }
}
```



```
        }
        line = br.readLine();
      }
} catch(IOException e) {dbg.p("Error in filehandler.java: "+e);}
  return new Grammar(startSymbol, nonTerminals,
                     terminals, productionRules);
}

/**
 * Creates definition of attribute function from textual
 * representation of function.
 *
 * @param pr   Corresponding production rule.
 * @param csr  String containng the textual representation of
 *             attribute function.
 * @return     new instance of class
 *             ProductionRule.AttributeFunctionDef reflecting
 *             the read attributr function definition.
 */
private static ProductionRule.AttributeFunctionDef resolveFunDef(
                      ProductionRule pr, String csr) {

  int operation = -1;
  Vector arguments = new Vector();

  dbg.p("input: "+csr);

  int pos = csr.indexOf("(");
  if (pos >= 0) {
    String fun = csr.substring(0, pos);
    fun = fun.trim();
    csr = csr.substring(pos+1);

    dbg.p("fun1: "+fun);

    if (fun.compareTo("hd1") == 0 ||
      fun.compareTo("headderive1") == 0) {

      operation = AttributeCalculator.OPERATION_HEADDERIVE1;

    } else if (fun.compareTo("hd2") == 0 ||
          fun.compareTo("headderive2") == 0) {
```

```
  operation = AttributeCalculator.OPERATION_HEADDERIVE2;

} else if (fun.compareTo("fa1") == 0 ||
       fun.compareTo("featureaccept1") == 0) {

  operation = AttributeCalculator.OPERATION_FEATUREACCEPT1;

} else if (fun.compareTo("fa2") == 0 ||
       fun.compareTo("featureaccept2") == 0) {

  operation = AttributeCalculator.OPERATION_FEATUREACCEPT2;

} else if (fun.compareTo("match") == 0) {

  operation = AttributeCalculator.OPERATION_MATCH;

} else if (fun.compareTo("unite") == 0) {

  operation = AttributeCalculator.OPERATION_UNITE;

}

//read parameters
do {
  if (csr.indexOf(",") == 0) csr = csr.substring(1);

  if (csr.indexOf("(") >= 0 &&
    (csr.indexOf("(") < csr.indexOf(",") ||
     csr.indexOf(",") < 0)) {

    //next argument is function
    pos = csr.indexOf(")");
    ProductionRule.AttributeFunctionDef arg =
             resolveFunDef(pr, csr.substring(0, pos+1));
    arguments.add(arg);
    csr = csr.substring(pos+1);
  } else {

    //next argument is value
    if (csr.indexOf(",") >= 0) {
      pos = csr.indexOf(",");
    } else {
      pos = csr.indexOf(")");
```



```
      }
      String argString = csr.substring(0, pos);
      arguments.add(new Integer(convertToPos(argString)));
      csr = csr.substring(pos);
    }

  } while (csr.indexOf(",") >= 0);
 } else {
  arguments.add(new Integer(convertToPos(csr.trim())));
  operation = AttributeCalculator.NO_OPERATION;
 }

 return pr.new AttributeFunctionDef(operation, arguments);
}

/**
 * Converts attribute argument symbol to right hand side position
 * in a production rule.
 *
 * @param argString  String containing argument.
 * @return           Position if found, else -1.
 */
private static int convertToPos(String argString) {
 int pPos = argString.indexOf(".");
 if (pPos >= 0) {
  String si = argString.substring(pPos-1, pPos);
  return new Integer(si).intValue();
 } else if (argString.trim().equals("_")) {
  return AttributeCalculator.BLANK_ARGUMENT;
 } else {
  dbg.p("Error in definition of attribute functions");
  return -1;
 }

}

/**
 * Reads input expression from string into tokenized string
 *
 * @param r  String containing expression.
 * @return   StringTokenizer containing words in expression
 *           string.
 */
```

```
public static StringTokenizer readExpressionString(String r) {
  String line = "";
try {
  BufferedReader br = new BufferedReader(new StringReader(r));
  line = br.readLine();
} catch (IOException e) {dbg.p("Error in expression string: "+e);}
  return (new StringTokenizer(line, " "));
}

/**
 * Reads sequence of input expression from string into vector of
 * tokenized strings.
 *
 * @param r  String containing expressions.
 * @return   Vector containing StringTokenizers with words in
 *           expressions.
 */
public static Vector readExpressionStrings(String r) {
  BufferedReader br = new BufferedReader(new StringReader(r));
  Vector result = new Vector();
  try {
    String line = br.readLine();

    while (line != null) {
      StringTokenizer strings = new StringTokenizer(line, ".");
      while (strings.hasMoreTokens()) {
        result.add(new StringTokenizer(strings.nextToken(), " "));
      }
      line = br.readLine();

    }
  } catch (IOException e) {dbg.p("Error in expression : "+e);}
  return result;
}


/**
 * Reads concept grammar from string and creates instance of
 * ConceptGrammar
 *
 * @param r  String to convert
 * @return   created ConceptGrammar
 */
```



```
public static ConceptGrammar readConceptGrammarString(String r) {
  BufferedReader br = new BufferedReader(new StringReader(r));
  ConceptGrammar result = new ConceptGrammar();
  Hashtable nonTerminals = new Hashtable(10);
  Hashtable terminals = new Hashtable(10);
  NonTerminal startSymbol = null;
  Vector productionRules = null;
try {
  String line = br.readLine();

  // First loop run finds non terminals

      boolean startSymbolFound = false;
      while (line != null) {
        if (line.length() > 1) {
          if (!line.substring(0,1).equals("\\")) {
            String str = line.substring(0, line.indexOf(" "));
            NonTerminal NewNT = new NonTerminal(str);
            nonTerminals.put(str ,NewNT);

            if (!startSymbolFound) {
              startSymbol = NewNT;
              startSymbolFound = true;
            }
          }
        }
        line = br.readLine();
      }

  br = new BufferedReader(new StringReader(r));

  // Second loop run finds productions

      line = br.readLine();
      productionRules = new Vector();

      while (line != null) {
        if (line.length() > 1) {
          if (!line.substring(0,1).equals("\\")) {

            NonTerminal lhs;
            Symbol rhs;
            Vector alternatives = new Vector(2);
```

```
String str = line.substring(0, line.indexOf(" "));
line = line.substring(line.indexOf(" ") + 1);
lhs = (NonTerminal) nonTerminals.get(str);
line = line.substring(line.indexOf(" ") + 1);
StringTokenizer st = new StringTokenizer(line, "|");

while (st.hasMoreTokens()) {

  // rhs concept
  String tmps = st.nextToken();
  String tmps2;
  if (tmps.indexOf("[") < 0) {
    tmps2 = tmps.trim();
  } else {
    tmps2 = tmps.substring(0, tmps.indexOf(" "));
    tmps = tmps.substring(tmps.indexOf(" "));
  }

  // a terminal symbol is read
  Terminal tmpT;
  if (!nonTerminals.containsKey(tmps2)) {
    if (terminals.containsKey(tmps2)) {
      tmpT = (Terminal) terminals.get(tmps2);
    } else {
      tmpT = new Terminal(tmps2);
      terminals.put(tmps2, tmpT);
    }
    rhs = (Symbol) tmpT;
  // a nonterminal symbol is read
  } else {
    rhs  = (Symbol) nonTerminals.get(tmps2);
  }
  // Features
  Vector features = new Vector(2);
  StringTokenizer st2 = new StringTokenizer(tmps, "[");
  while (st2.hasMoreTokens()) {
    String tmps3 = st2.nextToken();
    if (tmps3.indexOf("]") > 0) {
      String sAtb = tmps3.substring(0, tmps3.indexOf(":"));
      String sCon = tmps3.substring(
                           tmps3.indexOf(":") + 1 ,
                           tmps3.indexOf("]"));
```



```
                    ConceptGrammar.Feature newf =
                            result.new Feature(
                                (NonTerminal) nonTerminals.get(sAtb),
                                (NonTerminal) nonTerminals.get(sCon));

                    features.add(newf);
                  }
                }
                alternatives.add(result.new Alternative(rhs,
                                                        features));
              }
              productionRules.add(
                  result.new ConceptProductionRule(lhs,
                                                   alternatives));
            }
          }
          line = br.readLine();
        }

  } catch(IOException e) {dbg.p("Error in filehandler.java: "+e);}
    return new ConceptGrammar(startSymbol, nonTerminals,
                              terminals, productionRules);
  }
}
```

B.12 GUIListener.java

```
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

/**
 * Eventlistener that listens for events triggered by the user
 * interface.
 */
public class GUIListener implements ActionListener, ItemListener {

  /**
   * Default constructor.
   */
  public GUIListener() {
  }

  /**
   * Method called when actionEvent is registered. Calls methods
   * in class GUI when a button or menu item is activated in the
   * user interface.
   *
   * @param e  The performed ActionEvent
   */
  public void actionPerformed(java.awt.event.ActionEvent e) {

    if (e.getActionCommand().equals("ParseFirstLine")) {

      GUI.parseFirstLine();

    } else if (e.getActionCommand().equals("ParseAll")) {

      GUI.parseAll();

    } else if (e.getActionCommand().equals("stop")) {

      GUI.writeToOutput("\n*** Parsing cancelled ***\n",
                        GUI.SHOWALWAYS);
      GUI.stopParse = true;
      GUI.stopButton.setEnabled(false);
```



```
      GUI.parseButton.setEnabled(true);
      GUI.parseButton2.setEnabled(true);

    } else if (e.getActionCommand().equals("Clearow")) {

      GUI.clearOutputWindow();

    } else if (e.getActionCommand().equals("LoadSemGrammar")) {

      GUI.fc.setDialogTitle("Open semantic grammar file");
      GUI.fc.setCurrentDirectory(GUI.lastOpenedSemGramFile);
      GUI.fc.setSelectedFile(GUI.lastOpenedSemGramFile);
      int returnVal = GUI.fc.showOpenDialog(GUI.mainFrame);

      if (returnVal == JFileChooser.APPROVE_OPTION) {
        File file = GUI.fc.getSelectedFile();
        GUI.loadSemanticGrammar(file);
      }

    } else if (e.getActionCommand().equals("LoadLingvGrammar")) {

      GUI.fc.setDialogTitle("Open lingvistic grammar file");
      GUI.fc.setCurrentDirectory(GUI.lastOpenedLingvGramFile);
      GUI.fc.setSelectedFile(GUI.lastOpenedLingvGramFile);
      int returnVal = GUI.fc.showOpenDialog(GUI.mainFrame);

      if (returnVal == JFileChooser.APPROVE_OPTION) {
        File file = GUI.fc.getSelectedFile();
        GUI.loadSyntacticGrammar(file);
      }

    } else if (e.getActionCommand().equals("LoadInputText")) {

      GUI.fc.setDialogTitle("Open input text file");
      GUI.fc.setCurrentDirectory(GUI.lastOpenedTextFile);
      GUI.fc.setSelectedFile(GUI.lastOpenedTextFile);
      int returnVal = GUI.fc.showOpenDialog(GUI.mainFrame);

      if (returnVal == JFileChooser.APPROVE_OPTION) {
        File file = GUI.fc.getSelectedFile();
        GUI.loadText(file);
      }
```

```
} else if (e.getActionCommand().equals("SaveSemGrammar")) {

  GUI.fc.setDialogTitle("Save semantic grammar");
  GUI.fc.setCurrentDirectory(GUI.lastOpenedSemGramFile);
  GUI.fc.setSelectedFile(GUI.lastOpenedSemGramFile);
  int returnVal = GUI.fc.showSaveDialog(GUI.mainFrame);

  if (returnVal == JFileChooser.APPROVE_OPTION) {
    File file = GUI.fc.getSelectedFile();
    GUI.saveSemanticGrammar(file);
  }

} else if (e.getActionCommand().equals("SaveLingvGrammar")) {

  GUI.fc.setDialogTitle("Save lingvistic grammar");
  GUI.fc.setCurrentDirectory(GUI.lastOpenedLingvGramFile);
  GUI.fc.setSelectedFile(GUI.lastOpenedLingvGramFile);
  int returnVal = GUI.fc.showSaveDialog(GUI.mainFrame);

  if (returnVal == JFileChooser.APPROVE_OPTION) {
    File file = GUI.fc.getSelectedFile();
    GUI.saveSyntacticGrammar(file);
  }

} else if (e.getActionCommand().equals("SaveInputText")) {

  GUI.fc.setDialogTitle("Save input text");
  GUI.fc.setCurrentDirectory(GUI.lastOpenedTextFile);
  GUI.fc.setSelectedFile(GUI.lastOpenedTextFile);
  int returnVal = GUI.fc.showSaveDialog(GUI.mainFrame);

  if (returnVal == JFileChooser.APPROVE_OPTION) {
    File file = GUI.fc.getSelectedFile();
    GUI.saveText(file);
  }

} else if (e.getActionCommand().equals("SaveOutputText")) {

  GUI.fc.setDialogTitle("Save output text");
  GUI.fc.setCurrentDirectory(GUI.lastOpenedTextFile);
  GUI.fc.setSelectedFile(new File("output.txt"));
  int returnVal = GUI.fc.showSaveDialog(GUI.mainFrame);
```



```
    if (returnVal == JFileChooser.APPROVE_OPTION) {
      File file = GUI.fc.getSelectedFile();
      GUI.saveOutputText(file);
    }

  } else {
    GUI.writeToOutput("Not implemented.", GUI.SHOWALWAYS);
  }
}


private boolean dontDisable = false;

/**
 * Method called when ItemEvent is registered. Calls methods in
 * class GUI when a item state is changed in the user interface.
 *
 * @param e  The performed ItemEvent
 */
public synchronized void itemStateChanged(ItemEvent e) {

  if (e.getItem() == GUI.cbOutputMenuItem1) {

    if (e.getStateChange() == ItemEvent.DESELECTED) {
      GUI.outputControl[GUI.SHOWLINGVTREE] = false;
      if (GUI.cbOutputMenuItem4.isSelected()) {
        dontDisable = true;
        GUI.cbOutputMenuItem4.setSelected(false);
        dontDisable = false;
      }
    } else {
      GUI.outputControl[GUI.SHOWLINGVTREE] = true;
    }

  } else if (e.getItem() == GUI.cbOutputMenuItem2) {

    if (e.getStateChange() == ItemEvent.DESELECTED) {
      GUI.outputControl[GUI.SHOWSTACKSTATE] = false;
      if (GUI.cbOutputMenuItem4.isSelected()) {
        dontDisable = true;
        GUI.cbOutputMenuItem4.setSelected(false);
        dontDisable = false;
      }
```

```
    } else {
      GUI.outputControl[GUI.SHOWSTACKSTATE] = true;
    }

  } else if (e.getItem() == GUI.cbOutputMenuItem3) {

    if (e.getStateChange() == ItemEvent.DESELECTED) {
      GUI.outputControl[GUI.SHOWPARSEINFO] = false;
      if (GUI.cbOutputMenuItem4.isSelected()) {
        dontDisable = true;
        GUI.cbOutputMenuItem4.setSelected(false);
        dontDisable = false;
      }
    } else {
      GUI.outputControl[GUI.SHOWPARSEINFO] = true;
    }

  } else if (e.getItem() == GUI.cbOutputMenuItem4) {

    if (e.getStateChange() == ItemEvent.DESELECTED) {
      if (!dontDisable) {
        if (GUI.cbOutputMenuItem1.isSelected())
          GUI.cbOutputMenuItem1.setSelected(false);
        if (GUI.cbOutputMenuItem2.isSelected())
          GUI.cbOutputMenuItem2.setSelected(false);
        if (GUI.cbOutputMenuItem3.isSelected())
          GUI.cbOutputMenuItem3.setSelected(false);
      }
    } else {
      if (!GUI.cbOutputMenuItem1.isSelected())
        GUI.cbOutputMenuItem1.setSelected(true);
      if (!GUI.cbOutputMenuItem2.isSelected())
        GUI.cbOutputMenuItem2.setSelected(true);
      if (!GUI.cbOutputMenuItem3.isSelected())
        GUI.cbOutputMenuItem3.setSelected(true);
    }

  } else if (e.getItem() == GUI.semParseCheckBox) {

    if (e.getStateChange() == ItemEvent.DESELECTED) {
      GUI.semanticParse = false;
    } else {
      GUI.semanticParse = true;
```



```
      }
    }
  }
}
```

B.13 dbg.java

```
/**
 * Class to print debug information.
 */
public final class dbg {

  /**
   * debug state. set to true if program started with
   * parameter -debug.
   */
  private static boolean debugMode = false;

  /**
   * sets debug mode
   *
   * @param s  new debug mode
   */
    public static void setDebugMode(boolean dm) {
        debugMode = dm;
    }

  /**
   * gets debug mode
   *
   * @return  debug mode
   */
    public static boolean getDebugMode() {
        return debugMode;
    }

  /**
   * prints line of text to System.out if in debig mode
   *
   * @param s  text to print
   */
    public static void p(String s) {
        if (debugMode) {
          System.out.println("\n" + s);
        }
    }
```



```
  /**
   * prints text to System.out, if in debug mode
   *
   * @param s  text to print
   */
    public static void pp(String s) {
        if (debugMode) {
          System.out.print(s);
        }
    }
}
```

B.14 SubParseTree.java

```
import java.util.*;

/**
 * A part of a parse tree.
 */
public class SubParseTree {

  /**
   * The production rule reflecting this part of parse tree.
   * Null if this is a bottom element.
   */
  private ProductionRule rule;

  /**
   * The Terminal reflecting this part of parse tree.
   */
  private Terminal sym;

  /**
   * The sub parse trees just below this part of tree.
   */
  private SubParseTree[] subParseTrees;

  /**
   * Constructor for creating instance representing a bottom
   * element.
   *
   * @param sym  The terminal symbol in the bottom element.
   */
  public SubParseTree(Terminal sym) {
    this.sym = sym;
  }

  /**
   * Constructor for creating instance representing a part of
   * a tree without
   * having the subtrees ready or the symbol ready.
   *
   * @param rule  The rule used in this part of the tree.
   */
```



```
public SubParseTree(ProductionRule rule) {
  this.rule = rule;
  subParseTrees = new SubParseTree[rule.rhsSize()];
}

/**
 * Constructor for creating instance representing a part of a
 * tree.
 *
 * @param rule           The rule used in this part of the tree.
 * @param subParseTrees  The trees just below this part of tree.
 * @param sym            The symbol in this part of tree.
 */
public SubParseTree(ProductionRule rule,
          SubParseTree[] subParseTrees, Terminal sym){

  this.rule = rule;
  this.subParseTrees = subParseTrees;
  this.sym = sym;
}

/**
 * Connects a given SubParseTree to this.
 *
 * @param pos   Which sub tree position.
 * @param tree  The tree to connect to this.
 */
public void setSubParseTree(int pos, SubParseTree tree) {
  subParseTrees[pos] = tree;
}

/**
 * Creates a copy of this SubParseTree including sub trees.
 *
 * @return  a copy of this.
 */
public SubParseTree getCopy() {
  SubParseTree newTrees[] = new SubParseTree[countSubTrees()];
  for (int i = 0; i < countSubTrees(); i++) {
    newTrees[i] = subParseTrees[i].getCopy();
  }
  return new SubParseTree(rule, newTrees, sym);
}
```

```
/**
 * Gets one of the sub trees connected to this.
 *
 * @param pos  The sub trees position.
 * @return     The sub tree if exists. Null otherwise.
 */
public SubParseTree getSubParseTree(int pos) {
  if (subParseTrees != null)
    return subParseTrees[pos];
  else
    return null;
}

/**
 * Gets the number of trees just below this.
 *
 * @return  The number of trees if they exist. 0 otherwise.
 */
public int countSubTrees() {
  if (subParseTrees != null)
    return subParseTrees.length;
  else
    return 0;
}

/**
 * Prints textual description of this SubParseTree to a String.
 * Including sub trees.
 *
 * @param tab  The number of spaces to displace when printing
 *             subtrees.
 * @return     String containing the the full tree with sub
 *             trees.
 */
public String printToString(int tab) {
  String result = "";
  String s = "";
  for (int i = 0; i < tab; i++) {
    s += "   ";
  }

  if (sym != null) {
```



```
    result += s;
    result += sym.printToString();
    result += "\n";
  } else {
    result += s;
    result += rule.printToString();
    result += "\n";
    for (int i = 0; i < countSubTrees(); i++) {
      result += subParseTrees[i].printToString(tab+1);
    }
  }
  return result;
}

/**
 * Prints textual description of this SubParseTree including sub
 * trees to System.out with use of method pp in class dbg.
 *
 * @param tab  The number of spaces to displace when printing
 *             subtrees.
 */
public void print(int tab) {
  String s = "";
  for (int i = 0; i < tab; i++) {
    s += "   ";
  }

  if (sym != null) {
    dbg.pp(s);
    sym.print();
    dbg.pp("\n");
  } else {
    dbg.pp(s);
    rule.print();
    dbg.pp("\n");
    for (int i = 0; i < countSubTrees(); i++) {
      subParseTrees[i].print(tab+1);
    }
  }
}
}
```